Programming With System Calls For Interprocess Communication

Order No. 005696
Revision 00
Software Release 9.0

Apollo Computer Inc.
330 Billerica Road
Chelmsford, MA 01824

Printed in U.S.A.

First Printing: July, 1985

This document was produced using the SCRIBE® document preparation system. (SCRIBE is a registered trademark of Unilogic, Ltd.)

Preface

Programming With System Calls for Interprocess Communication describes the DOMAIN® system calls you can use for interprocess communication. Interprocess communication can involve data transfer, event notification, and synchronization.

Audience

This manual is intended for programmers who write applications involving several programs that run from separate processes. Before using this manual, you should be familiar with programming concepts and terminology, and should know how to use general-purpose DOMAIN system calls. You should also understand the DOMAIN implementation of the programming language you are using.

This manual describes different techniques for providing interprocess communication, and uses programming examples to explain the techniques. However, the manual does not provide complete reference information for each call that it demonstrates. For complete reference information, see the *DOMAIN System Call Reference (Volumes I and II).*

Organization of This Manual

This manual contains six chapters and two appendices:

- Chapter 1 gives an overview of the programming calls for interprocess communication.
- Chapter 2 describes sharing data through mapping.
- Chapter 3 describes using user-defined eventcounts for interprocess synchronization and event notification.
- Chapter 4 describes using mutex locks to control access to a shared resource.
- Chapter 5 describes interprocess communication using mailboxes.
- Chapter 6 describes interprocess communication using datagrams.
- Appendix A contains sample Pascal programs that show how to use the calls described throughout the manual.
- Appendix B contains translations, written in C, of the programs in Appendix A.

This manual uses excerpts of Pascal programs to illustrate the narrative descriptions. Each excerpt begins with the name of the program from which it was taken. To see the complete Pascal program, find the corresponding program in Appendix A. To see the C translation, find the corresponding program in Appendix B.

You can also view the programs on-line, as described in the next section.

On-Line Sample Programs

The programs from this manual are stored on-line, along with sample programs from other DOMAIN manuals. We include sample programs in Pascal and C. All programs in each language have been stored in master files (to conserve disk space). There is a master file for each language.

In order to access any of the on-line sample programs you must create one or more of the following links:

```
(For Pascal examples)  $ CRL ~COM/GETPAS /DOMAIN_EXAMPLES/PASCAL_EXAMPLES/GETPAS

(For C examples)       $ CRL ~COM/GETCC  /DOMAIN_EXAMPLES/CC_EXAMPLES/GETCC
```

To extract a sample program from one of the master files, all you have to do is execute one of the following programs:

```
(To get a Pascal program)     $ GETPAS

(To get a C program )         $ GETCC
```

These programs prompt you for the name of the sample program and the pathname of the file to copy it to. Here is a demonstration:

```
$ GETPAS
Enter the name of the program you want to retrieve -- MS_MAP
What file would you like to store the program in? -- MAP1.PAS

Done.
$
```

You can also enter the information on the command line in the following format:

```
$ GETPAS name_of_program_to_retrieve  name_of_output_file
```

For example, here is an alternate version of our earlier demonstration:

```
$ GETPAS MS_MAP MAP1.PAS
```

GETPAS and GETCC warn you if you try to write over an existing file.

For a complete list of on-line DOMAIN programs in a particular language, enter one of the following commands:

```
(for Pascal)   $ GETPAS HELP
(for C)        $ GETCC HELP
```

Technical Changes

This manual is part of a new programming documentation set for SR9. The programming documentation set includes the *DOMAIN System Call Reference*, plus the following programming guidebooks:

- *Programming With General System Calls*

- *Programming With System Calls for Interprocess Communication*
- *Programming With DOMAIN 2D Graphics Metafile Resource*
- *Programming with DOMAIN Graphics Primitives*

The *DOMAIN System Call Reference* includes complete information on call syntax, data types, constants, and error codes. The programming guidebooks describe how to use system calls within programs. Although the guidebooks describe many pre-SR9 system calls, the material has been significantly revised to include more usage information and examples.

Programming With System Calls for Interprocess Communication incorporates the following information from the SR8 *DOMAIN System Programmer's Reference Manual*:

- Mapped segments
- Eventcounts
- Mailboxes

In addition, this guidebook includes new technical information regarding mapped segments, mailboxes, and two new groups of system calls: mutual exclusion (MUTEX) and interprocess communication (IPC). The following list describes the technical changes for SR9:

- The mapped segment (MS) manager includes four new calls:

  MS_$ADVICE
  MS_$ATTRIBUTES
  MS_$FW_FILE
  MS_$TRUNCATE

  In addition, new there are two new MS data types for use with MS_$ADVICE: MS_$ACCESS_T and MS_$ADVICE_OPT_T. See Chapter 2 for more information.

- There is a new group of system calls that let programs share a file, or other resource, with mutual exclusion. See Chapter 4 for information on using the mutual exclusion (MUTEX) system calls.

- Mailbox (MBX) calls now accept larger messages. A server can now put and get messages that are up to 32767 bytes long; a client can now put and get messages that are up to 32761 bytes long. However, if a server sends a message that is larger than 1158 bytes to a client on a remote node, the remote node's MBX_HELPER must be able to handle the message. To handle the message, the remote node's MBX_HELPER must have a queue data size that is at least as large as the message.

  Also, three new MBX calls have been added:

  MBX_$CLIENT_WINDOW
  MBX_$PUT_CHR_COND
  MBX_$SERVER_WINDOW

  Three new MBX error codes have been added:

MBX_$SEQUENCED_SEND_FAILED
MBX_$CLIENT_NO_RIGHTS
MBX_$HELPER_NO_RIGHTS

See Chapter 5 for information on using MBX calls.

- There is a new group of system calls that perform fast interprocess communication. These calls let programs use communications buffers, called sockets, to send and receive messages, called datagrams. See Chapter 6 for information on using the interprocess communication (IPC) calls.

Suggested Reading Paths

Before you read this manual, you should be familiar with the following:

- *Getting Started With Your DOMAIN System* and *DOMAIN System User's Guide.* These manuals provides general information about using your node.
- *Programming With General System Calls.* This manual provides general information about using system calls. It also describes the general-purpose system calls.
- *DOMAIN System Call Reference.* These manuals give complete reference information on all DOMAIN system calls.

In addition, you should be familiar with the DOMAIN language manuals for your programming language.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following conventions:

UPPERCASE	Uppercase words or characters in formats and command descriptions represent commands or keywords that you must use literally.
lowercase	Lowercase words or characters in formats and command descriptions represent values that you must supply.
[ ]	Square brackets enclose optional items in formats and command descriptions. In sample Pascal statements, square brackets assume their Pascal meanings.
{ }	Braces enclose a list from which you must choose an item in format and command descriptions. In simple Pascal statements, braces assume their Pascal meanings.
|	A vertical bar separates items in a list of choices.
< >	Angle brackets enclose the name of a key on the keyboard.
CTRL/Z	The notation CTRL/ followed by the name of a key indicates a control character sequence. You should hold down the <CTRL> key while typing the character.

... Horizontal ellipsis points indicate that the preceding item can be repeated one or more times.

⋮ Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.

Problems, Questions, and Suggestions

We appreciate comments from the people who use our system. In order to make it easy for you to communicate with us, we provide the User Change Request (UCR) system for software-related comments, and the Reader's Response form for documentation comments. By using these formal channels you make it easy for us to respond to your comments.

You can get more information about how to submit a UCR by consulting the *DOMAIN System Command Reference* manual. Refer to the CRUCR (Create User Change Request) Shell command description. You can view the same information on-line by typing:

```
$ HELP CRUCR <RETURN>
```

For documentation comments, a Reader's Response form is located at the back of this manual.

Contents

Chapter 4 Using Mutual Exclusion Locks

Chapter 5 Using MailBoxes

Chapter 6 Sending Datagrams

Illustrations

Tables

Chapter 1
Introduction

Programming applications within a DOMAIN local area network often involve separate programs that need to communicate. For example, a program may need to send data to another program or may need to notify another program when an event has occurred. Programs may also need to share code or data. Communication between programs is called interprocess communication when the programs run from separate processes.

Related programs can run on the same, or on different, nodes. When an application involves programs on different nodes, it is a distributed application (or program). Distributed programming lets you share the resources of many nodes in support of a single application.

Your DOMAIN system includes several types of calls that let you perform interprocess communication. Some calls are used primarily on a single node, while others can be used in a distributed application.

This chapter gives an overview of the DOMAIN system calls for interprocess communication. Table 1-1 summarizes the calls and shows some of their common uses; Sections 1.1 through 1.5 give additional information about each group of calls.

Table 1-1. Summary of DOMAIN System Calls for Interprocess Communication

Call	Common Uses
Mapped segment (MS)	Share a common file; transfer data between programs. Used primarily by programs on the same node.
Eventcount (EC2)	Indicate that an event has occurred; synchronize program activities such as data transfer. Used by programs on the same node.
Mutual exclusion lock (MUTEX)	Control access to a shared file. Used by programs on the same node.
Mailbox (MBX)	Transfer data between programs. Used by programs on the same, or on different, nodes.
Interprocess Communication (IPC)	Transfer data between programs. Used primarily by programs on different nodes.

In addition to the calls described in this book, you may also find PGM_$INVOKE useful when you work with multi-process applications. PGM_$INVOKE allows one program to invoke another in a separate process. (The separate process is created on the caller's node.) Some of the programs in this manual illustrate PGM_$INVOKE. However, for additional information on this call, see *Programming With General System Calls*.

1.1. Mapped Segments

Mapping associates part of your address space with a disk file. After you map a file, you can access it by supplying the appropriate address. The system brings the required pages of the file into memory as needed.

For a single program, mapping is useful because it provides quick access to a file. For multiple programs that map the same file, mapping can provide a fast way to share or exchange data. Chapter 2 describes how to use the mapped segment (MS) calls to share data through mapping.

Programs on the same node can map a common file for both reading and writing. However, programs on different nodes can use mapping to share a file with read, but not write, access to the file. Because of this restriction, mapping is most useful for programs running on the same node.

Note that you must provide your own synchronization when several programs use mapping to update a common file or to transfer data. You can use eventcounts or mutual exclusion locks to provide this synchronization.

1.2. Eventcounts

An eventcount is a value that is incremented when an associated event occurs. The DOMAIN system provides eventcounts that programs can use to wait for certain types of system-defined events. In addition, the DOMAIN system provides eventcount calls that you can use to create your own (user-defined) eventcounts. Chapter 3 describes how to use the eventcount (EC2) calls to work with user-defined eventcounts. For information on system-defined eventcounts, see *Programming With General System Calls.*

User-defined eventcounts are useful for event notification and interprocess synchronization. For example, one program can establish an eventcount and advance the eventcount when a particular event occurs. A waiting program can then be notified that the eventcount has advanced. In this way, one program uses the eventcount to notify another that a relevant event has occurred, which allows the programs to synchronize their activities.

In order for many programs to use the same eventcount, the eventcount must be located in a memory location (such as a mapped file) that all the programs share. Because of the restrictions on mapping files, only programs that run on the same node can use the same eventcount.

1.3. Mutual Exclusion Locks

A mutual exclusion, or mutex, lock provides controlled access to a shared resource. If you associate a mutex lock with a resource, then programs must obtain the lock in order to use the resource. Only the program with the mutex lock can use the resource; other programs must wait until the lock is available. Chapter 4 describes how to use the mutual exclusion (MUTEX) calls to control access to a shared resource.

Mutex locks are often used to control access to a shared file. The mutex lock ensures that only one program at a time uses the file. In addition, the mutex lock provides a way to notify waiting programs when the file becomes available.

Programs use a mutex lock record to store information about whether the associated resource is in use. A mutex lock record, like an eventcount, must be located in a memory location (such as a mapped file) that all the programs share. Because of the restrictions on mapping files, only programs that run on the same node can use the same mutex lock.

1.4. Mailboxes

A mailbox is a file that programs use to send information to each other. A program that creates a mailbox is called a server; a program that opens a channel to an existing mailbox is called a client. Chapter 5 describes how to use the mailbox (MBX) calls for interprocess communication.

A mailbox provides a virtual circuit between a server and each of its clients. That is, the mailbox provides a guaranteed connection that maintains the message sequence. Programs on the same, or on different, nodes can communicate using a mailbox. However, if a server and a client are on different nodes, then you must run a mailbox helper (/SYS/MBX/MBX_HELPER) on each node.

Typically, programs use a mailbox for one-way data transfer (in which a client sends data to the server) or two-way data transfer (in which a client sends a request to the server and the server sends a response.) However, you can also use a mailbox for interprocess synchronization.

1.5. Interprocess Communication Datagrams

A interprocess communication (IPC) datagram is a message that one program sends to another through a communications buffer called a socket. Chapter 6 describes how to use the IPC calls to send datagrams.

A datagram connection is a high-speed, but unguaranteed, connection between two programs. That is, when one program sends a datagram to another, the system makes its best effort to deliver it. The system does not, however, guarantee message delivery and does not guarantee the message sequence. Therefore, programs that use IPC datagrams must verify that each datagram is successfully received.

You can use datagrams to send large amounts of data from one program to another. However, you can also use datagrams for transaction-oriented applications in which one program sends a request to another, and then waits for a response.

Chapter 2
Sharing Data Through Mapping

This chapter describes how programs can share data by mapping a common file. This chapter includes:

- An overview of mapping.
- A summary of DOMAIN system calls for mapping files.
- A description of the steps for mapping and working with shared files.

2.1. Overview

Mapping associates part of your address space with a disk file. When you map a file, the system reserves part of your address space for the file. In addition, the system returns a pointer with which you can access the file. As you reference different parts of the file, the system brings the required pages into memory.

You should map files that contain data in a user-defined format; do not map a DOMAIN record structured file, such as one created by a STREAM call. When you map a file, be sure that the program understands the file's format.

A single program uses mapping for fast access to a file. However, if an application involves many programs, each program can map the same file to share or exchange data. For example, several programs can map a common file to:

- Transfer data from one program to another.
- Read from a common data file.
- Read and write to a common file.

When programs on the same node map the same file, they share the same memory for the same pages of the file. Therefore, these programs can simultaneously map a file for reading and writing. If one program writes to the file, the others can access the changes. Note, however, that you must provide your own synchronization mechanisms when several programs write to the same file. To synchronize file access, use user-defined eventcounts or mutex locks. See Chapter 3 for information on eventcounts; see Chapter 4 for information on mutex locks.

When programs on different nodes map the same file, they do not share the same memory for pages of the file. Instead, each node stores pages from the file in its own memory. For this reason, programs on different nodes can map the same file only for reading. If programs on different nodes need to write to a common file, you can use one of the following techniques:

- Allow one program at a time to map the file. For example, one program can map, update, and unmap the file while programs on other nodes wait to map the file. To determine when a file is available, a waiting program would periodically attempt a map operation.

- Write a program that controls access to the file. Let other programs send requests to access the file, using mailbox (MBX) or interprocess communication (IPC) calls. See Chapter 5 for more information on MBX; see Chapter 6 for more information on IPC.

2.2. MS System Calls, Insert Files, and Data Types

To map a file, use MS system calls. These calls invoke the mapped segment (MS) manager, the system component that is responsible for mapping. Table 2-1 summarizes the MS calls.

Table 2-1. Summary of MS Calls

Operation	Call
Map and lock a file	MS_$CRMAPL MS_$MAPL
Provide file access advice	MS_$ADVICE
Change how the file is mapped	MS_$RELOCK MS_$REMAP
Obtain/change file attributes	MS_$ATTRIBUTES MS_$TRUNCATE
Force write a file	MS_$FW_FILE
Unmap an object	MS_$UNMAP

In order to use MS calls, you must include the appropriate insert file in your program. The MS insert files are:

/SYS/INS/MS.INS.C (for C)
/SYS/INS/MS.INS.FTN (for FORTRAN)
/SYS/INS/MS.INS.PAS (for Pascal)

Some of the MS calls require that you specify parameters using special DOMAIN data types. These include:

MS_$ACC_MODE_T An access mode.

MS_$ACCESS_T A file usage pattern.

MS_$ADVICE_OPT_T Reserved for future use.

MS_$CONC_MODE_T A concurrency mode.

In FORTRAN, use a 2-byte integer to represent each of these data types.

For complete information on the MS system calls and data types, see the *DOMAIN System Call Reference*.

2.3. Steps For Mapping a File

To access a file by mapping it, follow these steps:

- Use MS_$MAPL or MS_$CRMAPL to map the file and get a pointer to the file's location in your address space. MS_$MAPL maps an existing file; MS_$CRMAPL creates and maps a new file.

- When you map the file, specify a lock. A lock determines the type of access you can have to the file. The lock also affects the type of access that other programs can have to the file. Once you have a lock, other programs can only get locks that are compatible with yours.

- If you have a predicted pattern of file usage, use MS_$ADVICE to provide advice to the mapped segment (MS) manager. This helps the system provide better performance when managing the file on your behalf.

- Access data in the file by using the pointer returned by MS_$MAPL or MS_$CRMAPL.

- Perform any required map operations. To change your lock on a file, use MS_$RELOCK. To map a different section of a file, use MS_$REMAP. To obtain file attributes, use MS_$ATTRIBUTES. To truncate a file, use MS_$TRUNCATE. To force write a file, use MS_$FW_FILE.

- When you are through with the file, use MS_$UNMAP to unmap and unlock the file.

2.4. Mapping a File

There are two calls for mapping files:

- MS_$MAPL maps an existing file.

- MS_$CRMAPL creates and maps a new file.

Both calls require parameters to specify the information about the file you are mapping and the lock you are requesting. (A lock defines a type of file access. See Section 2.5 for more information.) In addition, both calls return a pointer to the mapped file. Use this pointer to reference the file. (A pointer is the same as an address.) A file remains mapped until you unmap it with MS_$UNMAP. See Section 2.10 for information on unmapping files.

The following sections describe different aspects of mapping a file:

- Section 2.4.1 describes how to define a user-defined record to help you access data in a mapped file.

- Section 2.4.2 describes the syntax for MS_$MAPL and MS_$CRMAPL.

- Section 2.4.3 describes how to provide usage advice after you map a file.

- Section 2.4.4 describes how to obtain file attributes after you map a file.

2.4.1. Creating a User-Defined Record

When you call MS_$MAPL and MS_$CRMAPL, you may find it useful to define a variable that is a pointer to a user-defined record. You can then use this variable to receive the pointer returned by MS_$MAPL or MS_$CRMAPL. To access data in the mapped file, use the pointer. (In FORTRAN, a pointer is an address.)

Figure 2-1 shows how to map a file that contains a user-defined record. First, the example defines a record and a pointer to this record. Then the program uses MS_$MAPL to map the file and obtain a pointer. The program then uses this pointer to access different fields in the record.

Note that MS_$MAPL returns a status code using the variable **status**, which is a record in STATUS_$T format. This record is defined in the BASE insert file. See *Programming With General System Calls* for more information on the structure of status codes.

```
PROGRAM ms_map;

{ This program maps a file and uses the pointer returned by MS_$MAPL
  to reference a user-defined record.  }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';

TYPE
    sales_record_t =
        RECORD
        item_code       : integer;
        units_sold      : integer;
        END;
VAR
    status              : status_$t;
    mapped_seg_ptr      : ^sales_record_t;
                .
                .
                .
BEGIN
    mapped_seg_ptr := ms_$mapl('sales_stats',   { file to map }
                                .
                                .                { other parameters }
                                .
                            status);

    IF status.all <> status_$ok THEN  RETURN;

    { Write a record to the file. }

    mapped_seg_ptr^.item_code := 1;
    mapped_seg_ptr^.units_sold := 10;
                .
                .
                .
END;   { program  }
```

Figure 2-1. Getting a Pointer to a User-Defined Record

If a file contains many records, you can define the area in which you map the file as an array of records. In this way, you can define the data file to be as simple or as complex as your application requires. Note that, in C, you may need to type-cast a pointer returned by MS_$MAPL or MS_$CRMAPL.

2.4.2. Using MS_$MAPL and MS_$CRMAPL

MS_$MAPL maps an existing file into your process address space, and obtains a lock on the file. MS_$CRMAPL creates, maps, and locks a new file.

MS_$MAPL has the following format:

```
address = MS_$MAPL (name, name-length, start, desired-length,
                    concurrency, access, extend, length-mapped, status)
```

The parameters are defined as follows:

- The **name** and **name-length** parameters indicate the pathname of the file that you are mapping, and the length of the pathname.

- The **start** parameter indicates the first byte to map. Use the value zero to indicate the first byte of the object.

- The **desired-length** parameter indicates the number of bytes to map. Note that MS_$MAPL may map more bytes than you request. MS_$MAPL returns the actual number of mapped bytes in the **length-mapped** parameter.

- The **concurrency** and **access** parameters determine the type of lock that you are requesting. The concurrency indicates the number of programs that can access a file; the access indicates the type of access that each program can have. To select concurrency and access values, first determine the type of lock you want to get. Then select the correct combination of concurrency and access modes to get this lock. See Section 2.5 for a description of locks, and for the combinations of concurrency and access modes that produce each lock. The concurrency mode must be of type MS_$CONC_MODE_T; in FORTRAN this is a 2-byte integer. MS_$MAPL allows the following concurrency modes:

MS_$NR_XOR_1W	Allows one writer or any number of readers
MS_$COWRITERS	Allows any number or readers and/or writers

  The access must be of type MS_$ACC_MODE_T; in FORTRAN this is a 2-byte integer. MS_$MAPL allows the following access types:

MS_$R	Read
MS_$RX	Read and execute
MS_$RIW	Read with intent to write (RIW)
MS_$WR	Read and write
MS_$WRX	Read, write, and execute

- The **extend** parameter indicates whether the entire length you specify (in the **desired-length** parameter) should be mapped. The value TRUE allows you to extend the size of the file to the desired length, even if the file is shorter. (That is, you can write data that increases the length of the file.) FALSE indicates that the amount mapped should be no greater than the actual length of the file.

- The **length-mapped** and **status** parameters are returned by MS_$MAPL. The length-mapped parameter indicates the number of bytes actually mapped. Note that in certain cases, MS_$MAPL maps a file in 32-page units, called segments. Therefore, the length mapped may exceed the desired length.

MS_$CRMAPL has a format that is similar to MS_$MAPL:

```
address = MS_$CRMAPL (name, name-length, start, desired-length,
                      concurrency, status)
```

The parameters for MS_$CRMAPL accept the same values as the corresponding parameters for MS_$CRMAPL. Note, however, that MS_$CRMAPL does not allow you to select an access mode or an extend value. Instead, MS_$CRMAPL always uses an access mode of MS_$WR and an extend value of TRUE.

Figure 2-2 shows a program that uses MS_$MAPL and MS_$CRMAPL. First, the program tries to map a file with MS_$MAPL. If the file does not exist, the program uses MS_$CRMAPL to create and map the file.

Note that MS_$MAPL returns the status NAME_$NOT_FOUND if no file currently exists. In order to test for this value, you must include the naming server insert file in your program.

```
PROGRAM ms_map;

{ This program tries to map a file.  If the file does
  not exist, the program creates and maps it.  }


%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ms.ins.pas';
%include '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record  }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);
```

Figure 2-2. Using MS_$MAPL and MS_$CRMAPL

```
VAR
  status         : status_$t;           { status }
  mapped_seg_ptr : ^sales_record_t;     { pointer to record }
  len_mapped     : integer32;           { length mapped }


BEGIN

{ Try to map existing file. }

mapped_seg_ptr := ms_$mapl (pathname,
                            namelength,
                            0,                          { where to start }
                            sizeof(sales_record_t),     { desired length }
                            ms_$nr_xor_1w,              { concurrency    }
                            ms_$wr,                     { access         }
                            true,                       { extension      }
                            len_mapped,
                            status);

IF(status.all <> status_$ok)   AND (status.all <> name_$not_found) THEN

  BEGIN
    error_$print (status);
    RETURN;
  END;

IF status.all = name_$not_found THEN

  BEGIN    { create and map   }
    mapped_seg_ptr := ms_$crmapl (pathname,
                                  namelength,
                                  0,                        { where to start  }
                                  sizeof(sales_record_t),   { desired length  }
                                  ms_$nr_xor_1w,            { concurrency     }
                                  status);

    IF status.all <> status_$ok THEN
      BEGIN
        error_$print (status);
        RETURN;
      END;
  END;     {create and map  }

{ Write a record to the file. }

mapped_seg_ptr^.item_code := 1;
mapped_seg_ptr^.units_sold := 10;
                .
                .
                .
END.
```

Figure 2-2. Using MS_$MAPL and MS_$CRMAPL (continued)

2.4.3. Providing Usage Advice

If you have a predicted type of file usage, you can use MS_$ADVICE to provide advice to the mapped segment manager. MS_$ADVICE can help the operating system optimize performance when managing your mapped file.
You can specify the following types of file access:

MS_$NORMAL — You do not have a predicted manner for accessing the file. This is the default if you never use MS_$ADVICE.

MS_$RANDOM — You will access the object randomly.

MS_$SEQUENTIAL — You will access the object sequentially.

Although it is not required that you use MS_$ADVICE, it is recommended that you provide file usage advice after you map a file. Note that you can call MS_$ADVICE more than once to change the advice for a mapped file.

MS_$ADVICE has the following format:

```
MS_$ADVICE (address, length, access, options, record-length, status)
```

The parameters are defined as follows:

- The **address** parameter is a pointer to the first byte for which to provide advice.
- The **length** parameter indicates the number of bytes for which to provide advice.
- The **access** parameter indicates the type of file access you are performing.
- The **options** parameter is reserved for future use. In Pascal, specify the empty set [ ]. In C and FORTRAN, declare a variable as a 4-byte integer and initialize it to zero (0).
- The **record-length** parameter indicates the number of bytes in one record of the mapped file. If you do not know the record length, or if the file is not record-structured, specify zero (0).
- The **status** parameter is returned by MS_$ADVICE to indicate whether the call was successful.

Figure 2-3 shows how to use MS_$ADVICE to inform the mapped segment manager that you plan to access a file sequentially.

```
PROGRAM ms_advice;

{ This program maps a file and then provides file usage advice. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ms.ins.pas';
%include '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;        { status }
    mapped_seg_ptr  : ^sales_record_t;  { pointer to record }
    len_mapped      : integer32;        { length mapped }

  BEGIN

  { Map existing file. }

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                                  { where to start  }
                              sizeof (sales_record_t) * 100,      { map 100 records }
                              ms_$nr_xor_1w,                      { concurrency     }
                              ms_$wr,                             { access          }
                              true,                               { extension       }
                              len_mapped,
                              status);
  IF(status.all <> status_$ok) THEN
    BEGIN
      error_$print (status);
      RETURN;
    END;

 { Provide advice to say you will access the file sequentially. }

  ms_$advice ( mapped_seg_ptr,            { first mapped byte }
               len_mapped,                { length            }
               ms_$sequential,            { access type       }
               [],                        { reserved          }
               sizeof ( sales_record_t ), { size of record    }
               status );
                    .
                    .
                    .
END.
```

Figure 2-3. Using MS_$ADVICE

2.4.4. Obtaining File Attributes

After you map a file, you can get information about the following file attributes:

- Permanence
- Immutability
- Length, in bytes
- Length, in blocks
- Date-time used
- Date-time modified
- Date-time created

To get attribute information, use MS_$ATTRIBUTES. This call has the following format:

```
MS_$ATTRIBUTES (address, attrib-buf, attrib-len, attrib-max, status)
```

The parameters are defined as follows:

- The **address** parameter specifies the pointer to the first byte of the currently mapped portion of the object.
- The **attrib-buf** and **attrib-len** are output parameters. The **attrib-buf** specifies a buffer in which the attributes are received. The **attrib-len** specifies the length of the returned attributes. When you specify an attributes buffer in Pascal and C, declare this buffer to be of type MS_$ATTRIB_T. MS_$ATTRIB_T is a record with the following fields:

PERMANENT	A Boolean value that indicates whether the object is permanent (TRUE) or temporary (FALSE)
IMMUTABLE	A Boolean value that indicates whether the object can be modified. The value TRUE means that the object is immutable. The value FALSE means that the object is not immutable and can be modified. (Even if an object is not immutable, you must still have a write lock in order to modify the object.)
CUR_LEN	Current length, in bytes, of the object.
BLOCKS_USED	The number of disk blocks used for the object.
DTU	Date-time used, in TIME_$CLOCKH_T format.
DTM	Date-time modified, in TIME_$CLOCKH_T format.
DTCR	Date-time created, in TIME_$CLOCKH_T format.

In FORTRAN, you can use a declaration like the following to declare an attributes buffer:

```
      CHARACTER ms_$attrib_t(22)
      CHARACTER permanent, immutable
      INTEGER*4 cur_len, blocks_used, dtu, dtm, dtcr
      EQUIVALENCE ( ms_$attrib_t(1), permanent ),
     2            ( ms_$attrib_t(2), immutable ),
     3            ( ms_$attrib_t(3), cur_len ),
     4            ( ms_$attrib_t(7), blocks_used ),
     5            ( ms_$attrib_t(11), dtu ),
     6            ( ms_$attrib_t(15), dtm ),
     7            ( ms_$attrib_t(19), dtcr )
```

Note, that the **permanent** and **immutable** fields are Boolean values that are one byte long. To determine the contents of these fields, use the FORTRAN statement ICHAR. For example, if ICHAR(PERMANENT) is zero, then the value is FALSE. If ICHAR(PERMANENT) is not zero, then the value is TRUE.

- The **attrib-max** is an input parameter that defines the length of the buffer where the attributes will be returned. This value defines the maximum amount of information that MS_$ATTRIBUTES can return.

- The **status** parameter returns a status code.

Figure 2-4 uses MS_$ATTRIBUTES to obtain the attributes of a mapped file.

```
PROGRAM ms_attributes;

{ This program maps an existing file and obtains its attributes. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ms.ins.pas';
%include '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map  }
     namelength = sizeof(pathname);
     attrib_max = sizeof (ms_$attrib_t);  { attribute buffer length }
```

Figure 2-4. Using MS_$ATTRIBUTES

```
VAR
  status          : status_$t;          { status }
  mapped_seg_ptr  : ^sales_record_t;    { pointer to record }
  len_mapped      : integer32;          { length mapped     }
  attrib_buf      : ms_$attrib_t;       { attribute buffer  }
  attrib_len      : integer;            { length of attribute record }


BEGIN

{ Map existing file. }

mapped_seg_ptr := ms_$mapl (pathname,
                            namelength,
                            0,                          { where to start  }
                            sizeof(sales_record_t),     { desired length  }
                            ms_$nr_xor_1w,              { concurrency     }
                            ms_$r,                      { access          }
                            true,                       { extension       }
                            len_mapped,
                            status);

IF(status.all <> status_$ok) THEN

  BEGIN
    error_$print (status);
    RETURN;
  END;

{ Determine current file length. }

MS_$ATTRIBUTES ( mapped_seg_ptr,{ start byte }
                 attrib_buf,    { attribute buffer }
                 attrib_len,    { length of returned attributes }
                 attrib_max,    { length of attribute buffer    }
                 status );

WRITELN ( 'The file is ', attrib_buf.cur_len,' bytes long.');
WRITELN ( 'The file uses ', attrib_buf.blocks_used, ' blocks.');
                 .
                 .
                 .
END.
```

Figure 2-4. Using MS_$ATTRIBUTES (continued)

2.5. Getting a Lock

You request a lock when you map the file with MS_$MAPL or MS_$CRMAPL. A lock determines the type of access your program can have to a file. In addition, a lock affects the types of access that programs from other processes can have to the file. Table 2-2 describes the allowable locks.

To get a lock, specify the appropriate concurrency and access modes when you map the file. Table 2-3 shows the concurrency and access modes for each lock.

Table 2-2. Types of MS Locks

Lock	Definition
Protected Read	You can read the file. Others can get any type of read lock, but no one can get any write locks.
Protected Read With Intent to Write (RIW)	You can read the file. Others can get protected or shared read locks. However, no one can get another RIW lock. After other readers unmap the file, you can get an exclusive write lock.
Shared Read	You can read the file. Others can get any type of read lock. Others can also get shared write locks.
Exclusive Write	You can read and write to the file. No one else can get any locks on the file.
Shared Write	You can read and write to the file. Others can get shared read or shared write locks.

Table 2-3. Concurrency and Access Modes for MS Locks

Lock	Concurrency mode	Access Mode
Protected Read	MS_$NR_XOR_1W	MS_$R or MS_$RX
Protected RIW	MS_$NR_XOR_1W	MS_$RIW
Shared Read	MS_$COWRITERS	MS_$R or MS_$RX or MS_$RIW
Exclusive Write	MS_$NR_XOR_1W	MS_$WR or MS_$WRX
Shared Write	MS_$COWRITERS	MS_$WR or MS_$WRX

Once you have locked a file, other programs can map the file only if these programs request a lock that is compatible with your lock. Table 2-4 shows the combinations of locks that are allowed and prohibited. Y means that the combinations are allowed; N means that the combinations are prohibited.

Note that when you map a file, your call returns immediately, even if the call is unable to perform your map request. For example, if you try to map a file with a lock that is incompatible with a lock that another file holds, your map call will return with an error status.

2.5.1. Protected Read Locks

Use protected read locks when several programs read a file, but do not write to it. No program can write to the file as long as other programs have protected read locks. You must use

Table 2-4. MS Lock Combinations

Existing Lock	Requested Lock*				
	Protected Read	Protected RIW	Shared Read	Exclusive Write	Shared Write
Protected Read	Y	Y	Y	N	N
Protected RIW	Y	N	Y	N	N
Shared Read	Y	Y	Y	N	Y **
Exclusive Write	N	N	N	N	N
Shared Write	N	N	Y **	N	Y **

* Y means that a combination is allowed. N means that a combination is prohibited.

** These locks are allowed only if the programs are on the same node.

MS_$MAPL to get a protected read lock; you cannot get a protected read lock with MS_$CRMAPL. However, if you use MS_$CRMAPL to get an exclusive write lock, you can later call MS_$RELOCK to change to a protected read lock.

2.5.2. Protected RIW Locks

Use protected RIW locks to read a file that you later want to update. The protected RIW lock allows other programs to map and read the file, but no one else can get a protected RIW lock. When you get a protected RIW lock, you do not block anyone else from reading the file. When you want to write to the file, you must wait for other programs to unmap the file. Then, you can change your protected RIW lock to an exclusive write lock.

The MS manager avoids a potential deadlock (where two programs wait to change protected RIW to write locks) by preventing two programs from getting protected RIW locks on the same file. However, the MS manager does not prevent deadlocks in which a program with an RIW lock waits for other programs to unlock the file. For example, a program may periodically try to use MS_$RELOCK to change an RIW lock to exclusive write; be sure that the program does not try indefinitely to relock the file. See Section 2.6 for more information on MS_$RELOCK.

You must use MS_$MAPL to get a protected RIW lock; you cannot get a protected RIW lock with MS_$CRMAPL. However, if you use MS_$CRMAPL to get an exclusive write lock, you can later call MS_$RELOCK to change to a protected read lock. See Section 2.6 for more information on MS_$RELOCK.

2.5.3. Shared Read Locks

Use a shared read lock when you want to read a file, but also allow other programs on your node to write to the file (by getting shared write locks). You must, however, provide your own synchronization if several programs read and write to the same file.

You must use MS_$MAPL to get a shared read lock; you cannot get a shared read lock with MS_$CRMAPL. Note that when you request a shared read lock, you can specify an access of MS_$R, MS_$RX, or MS_$RIW. For shared read locks, MS_$R and MS_$RIW are identical.

2.5.4. Exclusive Write Locks

Use an exclusive write lock when you want to write to a file, and you want to deny other programs access to the file. You can use either MS_$CRMAPL or MR_$MAPL to get an exclusive write lock.

2.5.5. Shared Write Locks

Use shared write locks to allow many programs to read from and write to the same file. Only programs on the same node can get shared write locks to the same file. You must provide your own synchronization if you allow more than one program to get a shared write lock.

Chapter 3 shows examples of two programs that use shared write locks to transfer data. Both programs map the same file with a shared write lock. One program places data in the shared file; the other reads data from the file. The programs use user-defined eventcounts to synchronize the data transfer.

2.6. Changing a Lock

Use the MS_$RELOCK call to change the lock on a file. MS_$RELOCK allows you to specify a new access mode. This new mode, in combination with the current concurrency mode, forms a new lock. Table 2-5 shows how to change from one lock to another. The table also describes any restrictions on changing locks. Note, however, that your ability to change a lock depends on the locks that other programs hold on the object. For example, it is legal to change a protected RIW lock to an exclusive write lock. However, you can change the lock only when no other programs hold locks on the file.

Figure 2-5 shows a program that maps a file with a protected read lock. Then it changes the lock to an exclusive write lock. In order to change the lock, the program must wait until no other programs have locks on the file.

```
PROGRAM ms_relock;

{ This program maps a file with a protected read lock.  Then it tries
  to change the lock to an exclusive write lock. If the MS_$RELOCK call
  is not successful, the program waits five seconds and then tries again
  to relock the file. }

%INCLUDE '/SYS/INS/BASE.INS.PAS';
%INCLUDE '/SYS/INS/MS.INS.PAS';
%INCLUDE '/SYS/INS/TIME.INS.PAS';
%INCLUDE '/SYS/INS/CAL.INS.PAS';
```

Figure 2-5. Relocking a File

```
LABEL
  unmap;

TYPE
  sales_record_t =          { user-defined record }
    RECORD
    item_code     : integer;
    units_sold    : integer;
    END;

CONST
  pathname = 'data_file';
  namelength = sizeof( pathname );

VAR
  status          : status_$t;
  mapped_seg_ptr  : ^sales_record_t;
  len_mapped      : integer32;
  wait_time       : time_$clock_t;

BEGIN

{ Map file with protected RIW lock. }

mapped_seg_ptr := ms_$mapl ( pathname,
                             namelength,
                             0,                          { where to start   }
                             sizeof( sales_record_t ),   { desired length   }
                             ms_$nr_xor_iw,              { concurrency      }
                             ms_$riw,                    { access           }
                             true,
                             len_mapped,
                             status );

IF status.all <> status_$ok THEN
  RETURN;

{ Get system clock value for 5 seconds. }

cal_$sec_to_clock ( 5, wait_time );

REPEAT

  { Keep trying to relock file until the MS_$RELOCK
    call is successful.  After each try, wait 5
    seconds and then try again. }

  ms_$relock( mapped_seg_ptr,
              ms_$wr,
              status );
  IF status.all <> ms_$in_use THEN
    EXIT;
  time_$wait( time_$relative, wait_time, status );
UNTIL false;
```

Figure 2-5. Relocking a File (continued)

```
  IF status.all <> status_$ok THEN
    GOTO unmap;

  { Write new data into the file. }

  mapped_seg_ptr^.item_code  := 1;
  mapped_seg_ptr^.units_sold := 300;

  { Unmap the file. }

unmap:

  ms_$unmap( mapped_seg_ptr,           { pointer to file }
             sizeof( sales_record_t ), { size of file    }
             status );

  IF status.all <> status_$ok THEN
    RETURN;
  writeln( 'File was unmapped' );

  END.
```

Figure 2-5. Relocking a File (continued)

Table 2-5. Ways to Relock a File

Current Lock	Changes
Protected read	Change to exclusive write by specifying the access mode MS_$WR or MS_$WRX. Change to protected RIW by specifying the access mode MS_$RIW.
Protected RIW	Change to exclusive write by specifying the access mode MS_$WR or MS_$WRX. Cannot change to protected read by specifying the access mode MS_$R.
Shared read	Change to shared write by specifying the access mode MS_$WR or MS_$WRX.
Exclusive write	Change to protected read by specifying the access mode MS_$R. Change to protected RIW by specifying the access mode MS_$RIW.
Shared write	Change to shared read by specifying MS_$R or MS_$RIW.

2.7. Remapping a File

If you are working with a very large file, you may not be able to map the entire file at one time. If you try to map a file that is larger than your virtual address space, you will get the error MS_$NO_SPACE. In such a case, map part of the file and then use the MS_$REMAP call to map another section. By mapping different sections of a file, you can use MS_$REMAP to move a sliding window over the file. MS_$REMAP does not change the lock mode of the object.

The program in Figure 2-6 maps a file one segment at a time. The file contains user-defined records that are four bytes long. First, the program prompts for the number of pages to map. (A page contains 1024 bytes, or 256 4-byte records.) Next, the program determines the number of segments needed to map the specified number of pages. (A segment contains 32 pages.)

The program maps the first segment of the file, starting at byte 0. After mapping the file, the program creates a loop to remap the file and get the next segment. After remapping the last segment, the program exits from the loop and unmaps the file.

Note that the program maps the file with an extend value of TRUE. Therefore, the program will map enough segments to contain the number of pages you specify, even if the actual file length is shorter.

```
PROGRAM ms_remap;

{ This program uses MS_$REMAP to map each segment of a
  large data file.  The file contains user-defined records. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/ms.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/pgm.ins.pas';

  TYPE
    sales_record_t =      { A sales record is 4 bytes long }
      RECORD
      item_code  : integer;
      units_sold : integer;
      END;
    data_page_t  = ARRAY [0..255] OF sales_record_t; { 256 records in a page  }
    mapped_seg_t = ARRAY [0..31] OF data_page_t;     { 32 pages in a segment  }

  CONST
    pathname    = 'data_file';
    namelength  = sizeof( pathname );

  VAR
    mapped_seg_ptr : ^mapped_seg_t;    { address returned by MS calls }
    status         : status_$t;        { status code                  }
    num_pages      : integer32;        { number of pages to map       }
    first_seg      : integer32;        { first segment to map         }
    last_seg       : integer32;        { last segment to map          }
    start_byte     : integer32;        { first byte                   }
    len_mapped     : integer32;        { length mapped                }
    seg_num        : integer32;        { counter                      }
```

Figure 2-6. Remapping a File

```
PROCEDURE check_status;  { for error handling }

  BEGIN
  IF status.all <> status_$ok THEN
    BEGIN
    error_$print( status );
    pgm_$exit;
    END;
  END;

BEGIN

 WRITE('Enter file size in pages: ');
 READLN(num_pages);

 { Determine the range of segments you need to map. }

 first_seg := 0;                          { first segment }
 last_seg := ( num_pages - 1 ) div 32;    { last segment  }

{ Initialize variable to indicate first byte to map. Map the first
  segment, starting at byte 0 in the file. }

 start_byte := 0;
 mapped_seg_ptr := ms_$mapl( pathname,          { file to map              }
                             namelength,        { length of file name      }
                             start_byte,        { start at first byte      }
                             32 * 1024,         { map 1 segment            }
                             ms_$nr_xor_1w,     { concurrency              }
                             ms_$wr,            { access                   }
                             true,              { extend                   }
                             len_mapped,        { bytes mapped - returned }
                             status);

 check_status;

{ Print message about the current segment.  Then remap the file
  to get the next segment.  Keep looping until you finish. }

 FOR seg_num := first_seg to last_seg DO
     BEGIN           { map loop }

       writeln('Finished mapping segment ',seg_num);
       writeln('Segment contains data starting at byte ', start_byte);
       writeln('The first record at this address has the item code ');
       writeln( mapped_seg_ptr^[0,0].item_code);

       { Remap the file to get the next segment unless you are done. }

       IF seg_num < last_seg THEN
```

Figure 2-6. Remapping a File (continued)

```
        BEGIN     { remap }

          start_byte := ( seg_num + 1 ) * 32 * 1024;
          mapped_seg_ptr :=  ms_$remap( mapped_seg_ptr, { previous segment }
                                        start_byte,     { new segment      }
                                        32 * 1024,      { map 1 segment    }
                                        len_mapped,     { length mapped    }
                                        status);
          check_status;

        END;      { remap }

      END;        { map loop }


{ Unmap the file and exit. }

ms_$unmap(mapped_seg_ptr,
          len_mapped,
          status);
check_status;

END.
```

Figure 2-6. Remapping a File (continued)

2.8. Truncating a File

Use MS_$TRUNCATE to set the length of a mapped file to the length that you specify. With MS_$TRUNCATE, you specify the pointer to the currently-mapped portion of the object. Also, specify the number of bytes you want to save (starting from the first byte.)

There are two common uses for MS_$TRUNCATE:

- To delete existing sections of a file.
- To set the file length to the number of bytes that the file actually uses.

When you want to delete existing sections of a mapped file, use MS_$TRUNCATE and specify the length at which to truncate the file. For example, if you map a file and you want to overwrite it, use MS_$TRUNCATE and specify a length of zero. Any existing information in the file will then be deleted.

You can also use MS_$TRUNCATE to define a length for a file, even if you are not throwing away data. For example, when you unmap a file, the system may set the file length to a value that is larger than the number of bytes that your file needs. To tell the system exactly how many bytes the file uses, call MS_$TRUNCATE and specify the length of your file.

Figure 2-7 illustrates MS_$TRUNCATE.

```
PROGRAM ms_truncate;

  { This program maps a file and uses MS_$TRUNCATE to delete
  the contents of the file.  After writing a new record to
  the file, the program uses MS_$TRUNCATE to set the file
  length to the number of used bytes. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ms.ins.pas';
%include '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =          { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;          { status            }
    mapped_seg_ptr  : ^sales_record_t;    { pointer to record }
    len_mapped      : integer32;          { length mapped     }


  BEGIN

  { Map existing file.}

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                          { where to start }
                              sizeof(sales_record_t),     { desired length }
                              ms_$nr_xor_1w,              { concurrency    }
                              ms_$wr,                     { access         }
                              true,                       { extension      }
                              len_mapped,
                              status);

  IF(status.all <> status_$ok)   THEN

    BEGIN
      error_$print (status);
      RETURN;
    END;

  { Truncate the file to zero to delete existing contents. }

  ms_$truncate ( mapped_seg_ptr,        { where to start    }
                 0,                     { no. bytes to keep }
                 status );
```

Figure 2-7. Truncating a File

```
  { Write a new record to the file. }

  mapped_seg_ptr^.item_code := 1;
  mapped_seg_ptr^.units_sold := 10;


  { Truncate the length to 4 - the number of used bytes.}

  MS_$TRUNCATE ( mapped_seg_ptr,          { where to start    }
                 sizeof (sales_record_t), { no. bytes to keep }
                 status );


  { Unmap file and exit. }

  ms_$unmap( mapped_seg_ptr,
             len_mapped,
             status );

  IF status.all <> status_$ok THEN
    BEGIN
      error_$print (status);
      RETURN;
    END;

END.
```

Figure 2-7. Truncating a File (continued)

2.9. Force Writing a File

When you work with mapped files, the system uses a predefined set of conditions to determine when to move information out of memory and back to disk. If you have mapped a local file, the system moves information back onto the disk. If you have mapped a remote file, the system moves the information back to the node where the disk is located.

If you need to supplement the system's actions, you can use MS_$FW_FILE to force write a file. Typically, you use MS_$FW_FILE when you are writing a database or transaction manager. In these cases, MS_$FW_FILE can help protect data integrity.

NOTE: You must use MS_$FW_FILE in combination with other mechanisms to ensure data integrity. Also, if you use MS_$FW_FILE inappropriately, you can significantly degrade system performance.

See the *DOMAIN System Call Reference* for more information on MS_$FW_FILE.

2.10. Unmapping a File

When you are through with the file, use MS_$UNMAP to unmap and unlock it. When you unmap a file, you release the sections of your process address space that were allocated for the file.

When you use MS_$UNMAP, specify the address of the currently mapped portion of the file, and the number of mapped bytes. MS_$UNMAP returns an error status. Figure 2-8 shows how to unmap a file.

```
PROGRAM ms_map;

{ This program maps an existing file named DATA_FILE.
  After working with the file, the program unmaps
  the file and exits. }

%include '/sys/ins/base.ins.pas';
%include '/sys/ins/error.ins.pas';
%include '/sys/ins/ms.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;          { status            }
    mapped_seg_ptr  : ^sales_record_t;    { pointer to record }
    len_mapped      : integer32;          { length mapped     }

  BEGIN

  { Map existing file. }

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                          { where to start }
                              400,                        { desired length }
                              ms_$nr_xor_1w,              { concurrency    }
                              ms_$wr,                     { access         }
                              true,                       { extension      }
                              len_mapped,
                              status);
                    .
                    .
                    .
```

Figure 2-8. Unmapping a File

```
  { Unmap file and exit. }

  ms_$unmap( mapped_seg_ptr,
             len_mapped,
             status );

  IF status.all <> status_$ok THEN
    BEGIN
      error_$print (status);
      RETURN;
    END;

END.
```

Figure 2-8. Unmapping a File (continued)

Chapter 3
Using User-Defined Eventcounts

This chapter describes how to use user-defined eventcounts. It includes:

- An overview of user-defined eventcounts.
- A summary of DOMAIN system calls for working with eventcounts.
- Examples of how to use user-defined eventcounts.
- Information on how asynchronous faults affect eventcount waits.

3.1. Overview

A user-defined eventcount is a value that a user program creates and advances when the associated event occurs. The event occurs within the program, and is therefore under the control of the program. You use a user-defined eventcount to synchronize activities among multiple programs involved in the same application. That is, one program advances the eventcount when the event occurs. If other programs are waiting for the eventcount, the change notifies these programs that an event has occurred.

In order for several programs to access an eventcount, the eventcount must be located in a memory location that all the programs can read and write to. Usually, the programs share memory by mapping a common file with read and write access. Because of the restrictions on mapping files, all the programs must be running on the same node.

A user-defined eventcount should be associated with only one type of event. The program that determines when this event occurs is called an event producer (or producer). The producer advances the eventcount after the event occurs. A program that needs to find out about an event is called a consumer. The consumer waits for the producer to advance the eventcount and then responds to the event. While waiting for the event, the consumer does not use computer processing time.

In addition to user-defined eventcounts, the DOMAIN system provides eventcounts that are associated with system objects. These system-defined eventcounts allow programs to wait for, and respond to, certain types of system events. The main difference between user and system-defined eventcounts is that user programs control the user-defined eventcounts, whereas the system controls the system-defined eventcounts. Thus, with user-defined eventcounts, you write both the producer and the consumer programs. However, with system-defined eventcounts, the system is the producer; your program is the consumer.

This chapter describes how to use user-defined eventcounts for interprocess communication. For information on system-defined eventcounts, see *Programming With General System Calls*.

3.2. EC2 System Calls, Insert Files, and Data Types

To work with user-defined eventcounts, use EC2 system calls. Table 3-1 summarizes the EC2 calls.

Table 3-1. Summary of EC2 System Calls

Operation	Call
Create a user-defined eventcount.	EC2_$INIT
Advance a user-defined eventcount.	EC2_$ADVANCE
Read the current value of an eventcount.	EC2_$READ
Wait until an eventcount reaches a trigger value.	EC2_$WAIT EC2_$WAIT_SVC

In order to use EC2 calls, you must include the appropriate insert file in your program. The EC2 insert files are:

/SYS/INS/EC2.INS.C	(for C)
/SYS/INS/EC2.INS.FTN	(for FORTRAN)
/SYS/INS/EC2.INS.PAS	(for Pascal)

Some of the EC2 calls require that you specify an eventcount directly. In these cases, specify a variable in EC2_$EVENTCOUNT_T format. The data type EC2_$EVENTCOUNT_T requires six bytes of storage. In FORTRAN, define this as an array of three INTEGER*2 elements.

Certain calls, however, require that you specify eventcounts using pointers. For these calls, specify an eventcount using a variable in EC2_$PTR_T format. EC2_$PTR_T is a pointer to an eventcount. In FORTRAN, use a declaration like the following:

```
INTEGER*4 ec2_pointer
```

For complete information on the EC2 system calls and data types, see the *DOMAIN System Call Reference*.

3.3. Steps For Using User-Defined Eventcounts

The following steps show how an event producer uses a user-defined eventcount to notify a consumer about an event. These steps describe a simple case, in which the producer "produces" an event, advances the eventcount to notify the consumer, and then exits.

The producer must perform the following steps:

1. Map a file in which to create the eventcount and use EC2_$INIT to initialize the eventcount in this file.

2. Create a condition field (in the mapped file) to indicate whether the event has occurred. To start, the producer sets this field to FALSE. (The consumer uses this field to avoid a deadlock the first time it waits for the eventcount.)

3. Perform the work needed to produce the event.

4. Set the condition field to TRUE.

5. Use EC2_$ADVANCE to increment the eventcount.

To wait for the event, the consumer must perform the following steps. (Note that you must start the consumer after the producer has initialized the eventcount and condition field.)

1. Map the file containing the eventcount.

2. Use EC2_$READ to read the current value of the eventcount.

3. Define an eventcount trigger value by adding one to the value you obtained from EC2_$READ.

4. Check the condition field. If the condition is FALSE, wait for the event to occur. If the condition is TRUE, respond to the event now.

5. Use EC2_$WAIT or EC2_$WAIT_SVC to wait for the eventcount to reach its trigger. These calls are the same, except for how they respond to asynchronous faults. See Section 3.6 for more information.

6. Respond to the event when EC2_$WAIT(_SVC) returns.

The above sequence of events is sufficient when a producer produces one event and exits. However, in many applications, a producer will repeatedly produce an event within a loop. Before repeating the loop, the producer needs to know whether the consumer responded to the event. In such a case, the producer and consumer need two eventcounts. The producer advances one eventcount to indicate that the event has occurred; the consumer advances the other eventcount to indicate a response.

Sections 3.4 and 3.5 show how to write a producer and a consumer that synchronize their activities using two eventcounts. The producer places data in a shared data buffer, and the consumer reads this data. The producer advances an eventcount to tell the consumer when there is data to read; the consumer advances an eventcount to tell the producer when to write new data. The two programs synchronize their actions so that:

- The consumer does not try to read data until the producer puts data into the buffer.

- The producer does not put new data into the buffer until after the consumer reads the current data.

Figure 3-1 illustrates how the producer and consumer use eventcounts to synchronize data transfer. In addition, the figure shows how the programs use a condition field. The condition field ensures that the initial synchronization is correct.

3.4. Writing An Event Producer

In order to write an event producer, you need to understand how to use EC2 calls to perform the basic eventcount operations. In addition, you need to understand how to synchronize actions among programs so that deadlocks do not occur.

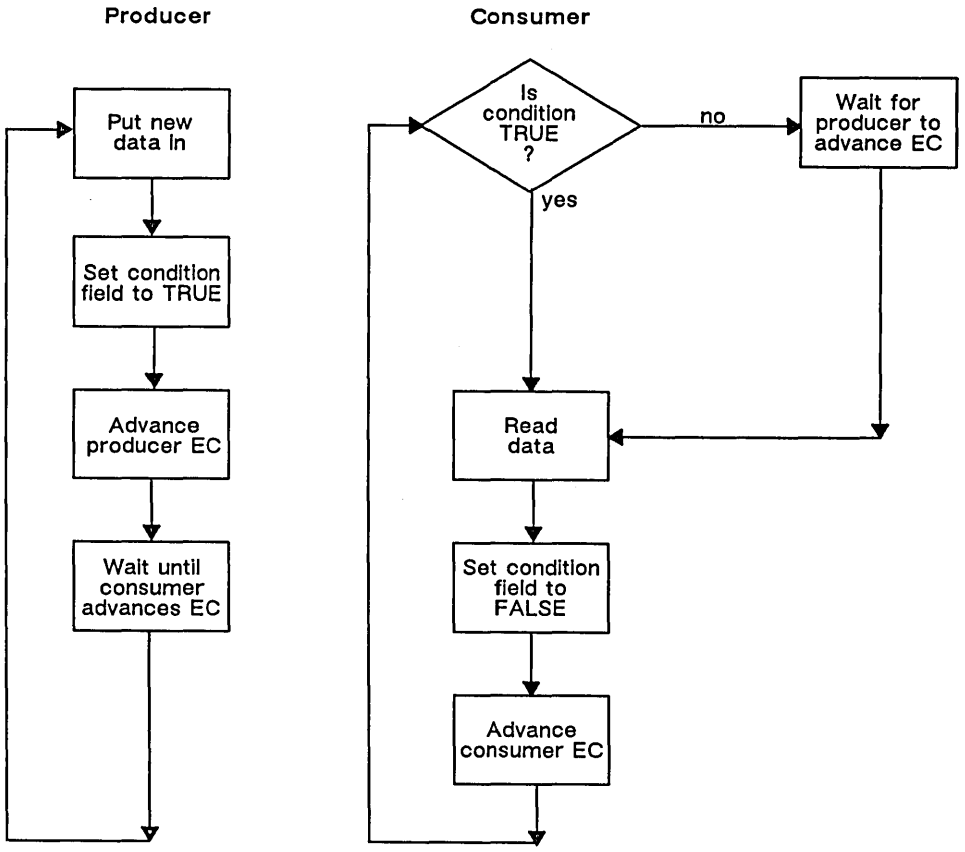

Figure 3-1. Eventcount Synchronization Between Two Programs

Section 3.4.1 describes how to perform the basic eventcount operations in a producer that puts data in a shared file, waits for the consumer to read this data, and then puts new data into the file. Section 3.4.2 shows the same program, adding some techniques to ensure proper synchronization.

3.4.1. Performing Basic Eventcount Operations

The producer performs the following eventcount operations. It:

- Maps a file in which to initialize eventcounts.
- Initializes eventcounts.

- Reads eventcounts.
- Advances eventcounts.
- Waits for events.

To map a file for user-defined eventcounts, use MS_$MAPL (to map an existing file) or MS_$CRMAPL (to create and map a file.) Specify a locking mode of MS_$COWRITERS and an access type of MS_$WR so that other programs can access the eventcounts. See Chapter 2 for more information on mapping files.

Map enough bytes to contain the eventcounts that the cooperating programs use, and any other information that the programs need to share. As described in Chapter 2, MS_$MAPL and MS_$CRMAPL return a pointer (or address) for the mapped file. Use this pointer to access data in the file.

Next, use EC2_$INIT to initialize eventcounts within the mapped file. This program initializes two eventcounts: one for the producer to advance and one for the consumer to advance. EC2_$INIT accepts one parameter: the eventcount to initialize. The eventcount must be in EC2_$EVENTCOUNT_T format, which requires six bytes of storage. Specify the eventcount in relation to the pointer you received from your MS call. For example, you can initialize the producer eventcount in the first six bytes of the file, and the consumer eventcount in the next six bytes.

Figure 3-2 shows how to map a file and initialize eventcounts.

```
PROGRAM ec2_producer;

{This program uses user-defined eventcounts to
 synchronize data transfer to another program. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

    TYPE
        shared_info_t =
            RECORD                                  { fields in mapped
                                                      storage record      }
            producer_ec     : ec2_$eventcount_t;    { producer eventcount }
            consumer_ec     : ec2_$eventcount_t;    { consumer eventcount }
                    .
                    .
                    .


            END;
```

Figure 3-2. Initializing User-Defined Eventcounts in a Producer

```
VAR
    status              : status_$t;          { status code           }
    length_mapped       : integer32;          { length of mapped data }
    shared_info         : ^shared_info_t;     { address of first byte }
                    .
                    .
                    .


PROCEDURE check_status;                      { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
        BEGIN
        error_$print( status );
        pgm_$exit;
        END;
    END;


BEGIN
                    .
                    .
                    .

{ Map a shared data file containing the eventcounts.
  Use this file to hold a record of type SHARED_INFO_T. }

shared_info := ms_$mapl( 'shared_file',       { object to be mapped     }
                         11,                  { length of name          }
                         0,                   { first byte to map       }
                         sizeof( shared_info_t ),
                                              { no. bytes to map        }
                         ms_$cowriters,       { locking mode            }
                         ms_$wr,              { access type             }
                         true,                { map length in 4th
                                                parameter, even if
                                                the object is shorter   }
                         length_mapped,       { bytes mapped - returned }
                         status );
check_status;
WITH shared_info^ DO
    BEGIN

    { Initialize the eventcounts }

    ec2_$init( producer_ec );
    ec2_$init( consumer_ec );
```

Figure 3-2. Initializing User-Defined Eventcounts in a Producer (continued)

Once you have initialized the eventcounts, any program that maps the shared file can read, wait for, or advance the eventcounts. In this example, the producer should read and wait for the consumer eventcount. In addition, the producer should advance its own eventcount.

To read an eventcount value, use EC2_$READ. To wait for an eventcount, use EC2_$WAIT or EC2_$WAIT_SVC. Usually, you read an eventcount value so you can define a trigger value for when you call EC2_$WAIT(_SVC). EC2_$WAIT(_SVC) suspends your program until the actual value of an eventcount reaches the trigger that you specify.

EC2_$READ accepts one parameter: the eventcount whose value you want to read. Specify the eventcount in EC2_$EVENTCOUNT_T format.

NOTE: You must use EC2_$READ to read eventcount values; if you attempt to refer to the eventcount directly, you may obtain an incorrect value, or you may incur a fault such as "odd address error", "access violation", or "reference to illegal address".

EC2_$WAIT(_SVC) has the following format:

```
ec-satisfied = EC2_$WAIT(_SVC) (ec-plist, ec-vlist, ec-count, status)
```

The parameters are described below:

- The **ec-plist** is an array of pointers to the eventcounts you are waiting for. The producer is waiting for only one eventcount: the consumer eventcount. Therefore, supply a pointer to the eventcount.

- The **ec-vlist** is an array of trigger values for each of the eventcounts. The order of the trigger values must correspond to the order of the eventcount pointers. In this example, there is only one eventcount pointer in the **ec-plist**. Therefore, supply only one trigger value.

- The **ec-count** is the number of eventcount pointers in the array. As stated previously, there is only one eventcount in the **ec-plist**.

- The **status** is the status code returned by the call.

When the eventcount in the **ec-plist** reaches its trigger value, EC2_$WAIT(_SVC) returns an ordinal number indicating the array subscript of the eventcount that is satisfied. Therefore, a return value of 1 indicates that the first (and, in this case, only) eventcount is satisfied.

In addition to reading and waiting for the consumer eventcount, the producer is responsible for advancing its own eventcount. The producer must advance its eventcount to tell the consumer when to read the data. To advance an eventcount, use EC2_$ADVANCE. This call accepts an eventcount in EC2_$EVENTCOUNT_T format and returns a status code.

Figure 3-3 shows how to read, wait for, and advance an eventcount. In this program, the producer reads the value of the consumer eventcount and adds one to it to form a trigger value. Next, the producer places data in a shared buffer, and advances its (the producer's) eventcount. Finally, the producer uses EC2_$WAIT to wait for the consumer to read the data.

```
PROGRAM ec2_producer;

{This program uses user-defined eventcounts to
 synchronize data transfer to another program. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

    TYPE
        shared_info_t =
            RECORD                                  { fields in mapped record }
            producer_ec     : ec2_$eventcount_t;    { producer eventcount     }
            consumer_ec     : ec2_$eventcount_t;    { consumer eventcount     }
                        .
                        .
                        .
            END;

    VAR
        status              : status_$t;            { status code               }
        length_mapped       : integer32;            { length of mapped data     }
        shared_info         : ^shared_info_t;       { address of first byte
                                                      of mapped storage         }
        consumer_wait_value : integer32;            { trigger value when
                                                      waiting for consumer
                                                      eventcount                }
        i                   : integer;              { value returned by ec2_$wait}

    PROCEDURE check_status;                    { for error handling }
                        .
                        .
                        .

    BEGIN
                        .
                        .
                        .
    { Map a shared data file containing the eventcounts.
      Use this file to hold a record of type SHARED_INFO_T. }

    shared_info := ms_$mapl( 'shared_file',         { object to be mapped     }
                             11,                    { length of name          }
                             0,                     { first byte to map       }
                             sizeof( shared_info_t ),
                                                    { no. bytes to map        }
                             ms_$cowriters,         { locking mode            }
                             ms_$wr,                { access type             }
                             true,                  { map length in 3rd
                                                      parameter, even if
                                                      object is shorter       }
                             length_mapped,         { bytes mapped - returned }
                             status );
    check_status;
```

Figure 3-3. Advancing a User-Defined Eventcount in a Producer

```
WITH shared_info^ DO
    BEGIN

    { Initialize the eventcounts }

    ec2_$init( producer_ec );
    ec2_$init( consumer_ec );

    { Store the current value of the consumer eventcount
      so you can use it as a trigger value later. }

    consumer_wait_value := ec2_$read( consumer_ec );
                    .
                    .
                    .
    REPEAT { loop over input file }

        { Put record into buffer for consumer to read.
          Then advance the producer eventcount. }
                    .
                    .
                    .

        ec2_$advance( producer_ec, status );
        check_status;

        { Wait for consumer to read the record. }

        consumer_wait_value := consumer_wait_value +1;
        i := ec2_$wait( addr( consumer_ec ),  { eventcount      }
                        consumer_wait_value,  { trigger value   }
                        1,                    { no. eventcounts }
                        status );
        check_status;

    UNTIL done;

                    .
                    .
                    .
```

Figure 3-3. Advancing a User-Defined Eventcount in a Producer (continued)

3.4.2. Synchronizing Reading and Writing

Once you understand the basic eventcount operations, you can add some techniques to ensure synchronization between the producer and consumer. For example, you can include a condition field in the shared file that you have mapped. Set this field to TRUE when there is data to read, and set it to FALSE when there is no data. The consumer uses the condition field to avoid a deadlock the first time it waits for the producer eventcount.

Another technique is to add a termination field to the mapped file. Set this field to TRUE when the data transfer is complete.

A third technique is to use PGM_$INVOKE to invoke the consumer from the producer. In this way, you can ensure that the eventcount file is initialized before the consumer tries to use it. See *Programming With General System Calls* for more information on PGM_$INVOKE.

To incorporate these synchronization techniques in a producer, follow these steps:

- Map a file to contain the eventcounts, the termination field, and the condition field. In addition, create fields for the data the producer will send, and the data length. Now all the information that the producer and consumer need to share will be in a single file.
- Initialize the producer and consumer eventcounts.
- Read the consumer eventcount so you can later define a trigger value.
- Set the condition field to FALSE because you have not yet written any data.
- Invoke the consumer program in a separate process. This step is not required within the producer. However, if you use a different method to start the consumer, you must ensure that the consumer does not map and read the shared file until after the producer initializes the appropriate fields.
- Design a loop to get data, put it into the mapped file, and wait for the consumer to read the data. Keep transferring data until you are through.
- At the end of the data transfer, unmap the shared file and exit.

To synchronize the data transfer, perform the tasks in the following order:

- Place the data into the shared file.
- Set the condition field to TRUE.
- Advance the producer eventcount to notify the consumer that there is data to read. You must advance the eventcount *after* you set the condition field or the synchronization may be wrong. For example, if the producer advances the eventcount before setting the condition field, the consumer might check the condition field, see that there is no data, and then wait for the producer eventcount to advance. In such a case, the consumer would be waiting for an eventcount that the producer had already advanced.

Figure 3-4 shows a producer that gets records from an input file and puts them into a mapped file. The producer uses stream calls to get and move records. See *Programming With General System Calls* for more information on streams.

The producer synchronizes reading and writing by using eventcounts and a condition value. After getting all the records, the producer unmaps the eventcount file and closes the input file.

```
PROGRAM ec2_producer;

{This program uses user-defined eventcounts to
 synchronize data transfer to another program. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

    TYPE
        shared_info_t =
            RECORD                                  { fields in mapped record    }
            producer_ec     : ec2_$eventcount_t;    { producer eventcount        }
            consumer_ec     : ec2_$eventcount_t;    { consumer eventcount        }
            done            : boolean;              { set to TRUE when you
                                                      reach end of input file    }
            record_present  : boolean;              { set to TRUE when you copy
                                                      record from input file     }
            useful_data     : string;               { record from input file     }
            data_len        : integer32;            { length of useful_data      }
            END;

    VAR
        status              : status_$t;            { status code                }
        length_mapped       : integer32;            { length of mapped data      }
        stream_id           : stream_$id_t;         { stream ID                  }
        seek_key            : stream_$sk_t;         { where stream data starts   }
        shared_info         : ^shared_info_t;       { address of first byte
                                                      of mapped storage          }
        res                 : uid_$t;               { reserved parameter for
                                                      pgm_$invoke                }
        consumer_wait_value : integer32;            { trigger value when
                                                      waiting for consumer
                                                      eventcount                 }
        local_buf           : string;               { buffer where stream_$get_ rec
                                                      may copy data              }
        data_ptr            : ^string;              { buffer where stream_$get_rec
                                                      copies data                }
        i                   : integer;              { value returned by ec2_$wait}


    PROCEDURE check_status;                  { for error handling }

        BEGIN
        IF status.all <> status_$ok THEN
            BEGIN
            error_$print( status );
            pgm_$exit;
            END;
        END;
```

Figure 3-4. Event Synchronization in a Producer

```
BEGIN
{ Open a stream to get records from. }

stream_$open( 'input_file',          { object to be opened }
              10,                    { length of pathname  }
              stream_$read,          { access type         }
              stream_$no_conc_write, { concurrency type    }
              stream_id,             { stream ID returned  }
              status );
check_status;

{ Map a shared data file containing the eventcounts.
  Use this file to hold a record of type SHARED_INFO_T. }

shared_info := ms_$mapl( 'shared_file',          { object to be mapped }
                         11,                     { length of name      }
                         0,                      { first byte to map   }
                         sizeof( shared_info_t ), { no. bytes to map    }
                         ms_$cowriters,          { locking mode        }
                         ms_$wr,                 { access type         }
                         true,            { map length in 3rd parameter,
                                            even if object is shorter   }
                         length_mapped,   { bytes mapped - returned     }
                         status );
check_status;
WITH shared_info^ DO
    BEGIN

    { Initialize the eventcounts }

    ec2_$init( producer_ec );
    ec2_$init( consumer_ec );

    { Store the current value of the consumer eventcount
      so you can use it as a trigger value later. }

    consumer_wait_value := ec2_$read( consumer_ec );

    { Set the condition field to FALSE. }

    record_present := false;

    { Invoke the consumer program in a separate process. }

    pgm_$invoke( 'ec2_consumer.bin', { program to invoke           }
                 16,                 { name length                 }
                 0,                  { no. args. to pass           }
                 0,                  { addresses of args.          }
                 0,                  { no. of streams to pass      }
                 0,                  { stream IDs to pass          }
                 [],                 { invoke in a separate process }
                 res,                { reserved                    }
                 status );
    check_status;
```

Figure 3-4. Event Synchronization in a Producer (continued)

```
    REPEAT { loop over input file }

    { Get a record.  If the input record is longer than your buffer,
      the get call returns a negative value.  To correct this, define
      the record length to be the buffer length. }

        stream_$get_rec( stream_id,          { stream ID to get record from}
                         addr( local_buf ), { where data may be read      }
                         sizeof( local_buf ), { buffer size               }
                         data_ptr,           { pointer to data - returned  }
                         data_len,           { length of data              }
                         seek_key,
                         status );

        IF data_len < 0 THEN
            data_len := sizeof( local_buf );

        { Set DONE to TRUE or FALSE, depending on result
          of stream_$get_rec.  Store DONE in shared memory. }

        done := status.all <> status_$ok;

        { Copy record to shared buffer. }

        useful_data := data_ptr^;

        { Set condition field to TRUE and advance producer
          eventcount. }

        record_present := true;
        ec2_$advance( producer_ec, status );
        check_status;

        { Wait for consumer to get the record. }

        consumer_wait_value := consumer_wait_value +1;
        i := ec2_$wait( addr( consumer_ec ),  { eventcount      }
                        consumer_wait_value,  { trigger value   }
                        1,                    { no. eventcounts }
                        status );
        check_status;

    UNTIL done;

    END;

{ Clean up and terminate.}

ms_$unmap( shared_info,   { first byte of mapped object }
           length_mapped, { length of mapped object     }
           status );
stream_$close( stream_id, status );

END.
```

Figure 3-4. Event Synchronization in a Producer (continued)

3.5. Writing An Event Consumer

To write an event consumer, follow these steps:

- Map the file containing the eventcounts.
- Read the current value of the producer eventcount.
- Create a loop to wait for and read records from the mapped file.

To map the file, use MS_$MAPL; the producer will have already created the file. When you map the file, specify a locking mode of MS_$COWRITERS and an access mode of MS_$WR. Map enough bytes to contain the eventcounts that cooperating programs will use, and the data that the programs will read or write.

To read the current value of the producer eventcount, use EC2_$READ. Later in the program you will use this value to create a trigger value for EC2_$WAIT.

Next, create a loop to wait for and read data. The consumer uses two types of information to determine when there is data to read:

- A condition field
- The producer eventcount

In order to read a record, the consumer must know that the producer has written a new record. Therefore, at the top of the loop, the consumer checks the condition field. If the condition is true, the consumer reads data from the mapped file. If the condition is false, the consumer waits for the producer eventcount to increase, and then checks the condition again. (See Section 3.4.2 for more information about the condition field in the mapped file.)

NOTE: The consumer must check the condition before waiting for the producer eventcount to increase. Otherwise, the initial synchronization may be incorrect.

When the condition is TRUE, the consumer can read data from the mapped file. After reading the data, the consumer sets the condition to FALSE and increments the consumer eventcount. The consumer then returns to the top of the loop to wait for more records.

Figure 3-5 shows a consumer that waits for and reads records that another program (the producer) places in a mapped file. The consumer uses eventcounts and a test condition to make sure that the reading and writing operations are synchronized.

```
PROGRAM ec2_consumer;

{ This program uses user-defined eventcounts to
  read data sent by another program. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';
```

Figure 3-5. Sample Event Consumer

```
TYPE
    shared_info_t =    { fields in mapped
                         storage record  }
        RECORD
        producer_ec    : ec2_$eventcount_t; { producer eventcount       }
        consumer_ec    : ec2_$eventcount_t; { consumer eventcount       }
        done           : boolean;           { set to TRUE when you
                                              reach end of input file   }
        record_present : boolean;           { set to TRUE when you copy
                                              record from input file    }
        useful_data    : string;            { record from input file    }
        data_len       : integer32;         { length of useful_data     }
        END;

VAR
    status              : status_$t;        { status code                }
    length_mapped       : integer32;        { length of mapped data      }
    stream_id           : stream_$id_t;     { stream ID                  }
    seek_key            : stream_$sk_t;     { where stream data starts   }
    shared_info         : ^shared_info_t;   { address of first byte
                                              of mapped storage          }
    producer_wait_value : integer32;        { trigger value when
                                              waiting for consumer
                                              eventcount                 }
    i                   : integer;          { value returned by ec2_$wait}

PROCEDURE check_status;

    BEGIN
    IF status.all <> status_$ok THEN
        BEGIN
        error_$print( status );
        pgm_$exit;
        END;
    END;

BEGIN

{ Open a stream to "consume" records into. }

stream_$create( 'output_file',          {object to be created   }
                11,                     { name length           }
                stream_$overwrite,      { access type           }
                stream_$no_conc_write,  { concurrency           }
                stream_id,              { stream ID - returned  }
                status );               { completion status     }
check_status;

{ Map the shared data file containing the eventcounts. }
```

Figure 3-5. Sample Event Consumer (continued)

```
shared_info := ms_$mapl( 'shared_file',  { object to be mapped     }
                         11,             { name length            }
                         0,              { first byte to map      }
                         sizeof( shared_info_t ),
                                         { no. bytes to map       }
                         ms_$cowriters,  { locking mode           }
                         ms_$wr,         { access type            }
                         true,           { map length in 3rd
                                           parameter, even if
                                           object is shorter      }
                         length_mapped,  { bytes mapped - returned }
                         status );
check_status;

WITH shared_info^ DO
    BEGIN
    { Store the current value of the producer eventcount
      so you can use it as a trigger value later. }

    producer_wait_value := ec2_$read( producer_ec );

    REPEAT { wait for and process records }

    { Wait until a record is available. Check RECORD_PRESENT
      before you wait. }

        WHILE NOT record_present DO
            BEGIN
            producer_wait_value := producer_wait_value + 1;
            i := ec2_$wait( addr( producer_ec ),  { ptr to eventcount   }
                            producer_wait_value,  { trigger value       }
                            1,                    { no. of eventcounts  }
                            status );
            END;

        { If a record is available, write it to the output file }

        IF NOT done THEN
            BEGIN
            stream_$put_rec( stream_id,           { stream ID      }
                             addr( useful_data ),{ data to put    }
                             data_len,            { length of data }
                             seek_key,            { seek key       }
                             status );
            check_status;
            END;

        { Set condition field to FALSE and advance consumer eventcount. }

        record_present := false;
        ec2_$advance( consumer_ec, status );
        check_status;

    UNTIL done;

    END;
```

Figure 3-5. Sample Event Consumer (continued)

```
{ Clean up and terminate.}

ms_$unmap( shared_info,    { first byte of mapped object }
           length_mapped,  { length of mapped object     }
           status );
stream_$close( stream_id, status );

END.
```

Figure 3-5. Sample Event Consumer (continued)

3.6. Asynchronous Faults During Eventcount Waits

When you use EC2_$WAIT or EC2_$WAIT_SVC, you cause a program to wait until the eventcount reaches its trigger value. However, during the wait, an asynchronous fault can occur. An asynchronous fault, such as a "quit", is generated outside your program.

If an asynchronous fault occurs during an EC2_$WAIT or EC2_$WAIT_SVC, the program's response depends on:

- The type of error handling the program uses.
- Whether the program uses EC2_$WAIT or EC2_$WAIT_SVC.

A program can handle asynchronous faults in the following ways:

- Declare a clean-up handler (with PFM_$CLEANUP) to perform clean-up operations and exit.
- Declare a fault handler (with PFM_$ESTABLISH_FAULT_HANDLER) to handle the fault. After taking corrective action, a fault handler often returns control to the program.
- Inhibit asynchronous faults (with PFM_$INHIBIT). This causes the program to ignore asynchronous faults. To re-enable faults, call PFM_$ENABLE.

If a program does not use one of the above techniques to handle asynchronous faults, then the system aborts the program if an asynchronous fault occurs. See *Programming With General System Calls* for more information on fault and cleanup handlers.

Table 3-2 shows how EC2_$WAIT and EC2_$WAIT_SVC respond to an asynchronous fault, if faults are enabled. Note that if a program uses a clean-up handler, an asynchronous fault aborts both EC2_$WAIT and EC2_$WAIT_SVC. However, if a program uses a fault handler, EC2_$WAIT and EC2_$WAIT_SVC perform differently. When the fault handler returns control to the program, EC2_$WAIT continues waiting. However, EC2_$WAIT_SVC returns the error EC2_$WAIT_QUIT.

Table 3-3 shows how EC2_$WAIT and EC2_$WAIT_SVC act when asynchronous faults are disabled. Note that EC2_$WAIT always defers asynchronous faults and continues waiting. In contrast, EC2_$WAIT_SVC defers asynchronous faults and returns the error EC2_$WAIT_QUIT. See *Programming With General System Calls* for more information on how faults are deferred when asynchronous faults are disabled.

Table 3-2. Wait Actions When Asynchronous Faults Are Enabled

Call	Error-Handling Technique	
	Clean-Up Handler	**Fault Handler**
EC2_$WAIT	Executes clean-up handler.	Executes fault handler. If fault handler returns control to the interrupted code, EC2_$WAIT continues waiting.
EC2_$WAIT_SVC	Executes clean-up handler.	Executes fault handler. If fault handler returns control to the interrupted code, EC2_$WAIT_SVC returns the error EC2_$WAIT_QUIT.

Table 3-3. Wait Actions When Asynchronous Faults Are Inhibited

Call	Error-Handling Technique	
	Clean-Up Handler	**Fault Handler**
EC2_$WAIT	Defers the fault and continues waiting.	Defers the fault and continues waiting.
EC2_$WAIT_SVC	Defers the fault, but returns the error EC2_$WAIT_QUIT.	Defers the fault, but returns the error EC2_$WAIT_QUIT.

When you use EC2_$WAIT(_SVC), you need to understand how your program will respond if an asynchronous fault occurs. First, you must ensure that the program performs any required clean-up actions if a fault occurs. Secondly, if faults are disabled, you must ensure that your program does not wait indefinitely if an asynchronous fault prevents the event from occurring.

If you inhibit asynchronous faults and use EC2_$WAIT, you must know that the wait can be satisfied in a short period of time. Otherwise, you should include a timer as one of the events you are waiting for. In this way, even though your program ignores faults, it remains in the wait state only for a predefined time. (See *Programming With General System Calls* for information on time eventcounts.)

Another way to inhibit faults, and still have a way to terminate a wait if a fault occurs, is to use EC2_$WAIT_SVC. EC2_$WAIT_SVC returns an error when an asynchronous fault occurs.

Figure 3-6 shows an example that inhibits asynchronous faults and uses EC2_$WAIT_SVC within a REPEAT loop. If the wait is not interrupted, the example responds to the wait and then drops through the bottom of the loop. However, if an asynchronous fault occurs during the wait, EC2_$WAIT_SVC returns an error. The program either exits or repeats the loop, depending on the type of fault handling that is in effect.

```
          .
          .
          .
REPEAT
    pfm_$inhibit;
    { Do work here.  For example, establish a temporary state
      and then wait for events before you continue. }
          .
          .
          .
    { Use ec2_$wait_svc to get error status if asynchronous
      fault occurs. }
    ec2_$wait_svc( pointer_list, trigger_list, status);
    IF status.all = status_$ok
         THEN
         { handle event }
           .
           .
           .
          ELSE
          { Fix things you did before you waited for events.
            If fault occurred during wait, status.all equals
             ec2_$wait_quit. }
           .
           .
           .
    pfm_$enable;
    { If program did not inhibit faults before the REPEAT
      loop, the condition or fault handler takes over here.
      If program previously inhibited faults or if fault
      handler returns control, continue. }

    { If fault occurred during ec2_$wait_svc, repeat loop and
      try again.  Otherwise, drop through loop and continue. }
UNTIL status.all <>  ec2_$wait_quit;
```

Figure 3-6. Handling Asynchronous Faults During Eventcount Waits

If you use a loop like the one in Figure 3-6, your program will work correctly whether or not the program disabled faults before entering the REPEAT loop. Also, the program will work correctly regardless of the type of error-handling in effect. The following list describes some possible situations:

- The program did not inhibit asynchronous faults before entering the loop, and the program does not use a fault handler. If a fault occurs during the wait, the ELSE clause restores items that were set before the wait. When the loop re-enables faults with PFM_$ENABLE, the fault occurs. The clean-up handler deals with the fault and the program exits.

- The program did not inhibit asynchronous faults before entering the loop, and the program uses a fault handler. If a fault occurs during the wait, the ELSE clause restores items that were set before the wait. When the loop re-enables faults with PFM_$ENABLE, the fault occurs. The fault handler deals with the fault, and returns control to the UNTIL condition. Because STATUS.ALL equals EC2_$WAIT_QUIT, the loop is repeated.

- The program previously inhibited asynchronous faults before entering the loop. If a fault occurs during the wait, the ELSE clause restores items that were set before the wait. The PFM_$ENABLE call decrements the inhibit count. However, this does not re-enable faults because the inhibit count is not zero. Because STATUS.ALL equals EC2_$WAIT_QUIT, the UNTIL condition is false and the loop is repeated.

Chapter 4
Using Mutual Exclusion Locks

This chapter describes how to use mutual exclusion (mutex) locks. It includes:

- An overview of mutual exclusion locks.
- A summary of DOMAIN system calls for working with mutex locks.
- Examples of programs that use mutex locks.

4.1. Overview

Mutual exclusion is a condition in which several programs share a resource, but only one program at a time can use it. When programs share with mutual exclusion, they can use a mutual exclusion (mutex) lock to control access to the resource.

In order to use the shared resource, a program must first request and receive the lock. If the resource is locked, the requesting program waits until the lock is available or until a specified amount of time has elapsed. When the program gets the lock, it uses the resource; when the program is through, it releases the lock for others to use.

A program determines whether it can get a lock by examining a mutex lock record. This record contains information about a resource's current lock status. The lock record is stored in a file; all programs using the lock must map this file with read and write access. To perform this map operation, the programs must be running on the same node. (See Chapter 2 for more information on mapping.)

Mutex locks are especially useful when you have several programs that share a file, but the order in which they use the file is unimportant. All that matters is that one program at a time reads (or writes to) the file. Otherwise, the file can become corrupted. When you use a mutex lock to control access to a file, you can store the lock record within the shared file.

NOTE: A mutex lock is a convention that cooperating programs use to control access to a resource. If a program bypasses the lock request and accesses the resource directly, you cannot guarantee mutual exclusion.

4.2. Mutex System Calls, Insert Files, and Data Types

Use the mutex system calls to work with mutex locks. These system calls begin with the prefix MUTEX. Table 4-1 summarizes the mutex calls.

In order to use mutex calls, you must include the appropriate insert file in your program. The mutex insert files are:

/SYS/INS/MUTEX.INS.C	(for C)
/SYS/INS/MUTEX.INS.FTN	(for FORTRAN)
/SYS/INS/MUTEX.INS.PAS	(for Pascal)

Table 4-1. Summary of MUTEX System Calls

Operation	Call
Initialize a mutex lock record	MUTEX_$INIT
Request the mutex lock	MUTEX_$LOCK
Release the mutex lock	MUTEX_$UNLOCK

When you refer to a mutex lock record from any of the mutex calls, you must use a value in MUTEX_$LOCK_REC_T format. In Pascal and C, you can simply declare a lock record variable to be of type MUTEX_$LOCK_REC_T. In FORTRAN, declare a mutex lock record variable as a 4-element array of 2-byte integers.

For complete information on MUTEX system calls and data types, see the *DOMAIN System Call Reference.*

4.3. Steps For Using Mutex Locks

Before you can use a mutex lock, one program must initialize the mutex lock record. Be sure that only one program initializes the record, regardless of how many programs use the lock. To initialize the lock record, map the file containing the record and then call MUTEX_$INIT. Now other programs can use the lock. (Note that the program that initializes a lock can also use it.)

To use a mutex lock after the lock record has been initialized, follow these steps:

- Map the file containing the lock record, if you have not already mapped it.
- Use MUTEX_$LOCK to request the lock. When you call MUTEX_$LOCK, specify the lock record and include a time-out value to indicate how long you want to wait. If you supply a negative value, or if you specify the constant MUTEX_$WAIT_FOREVER, you will wait indefinitely for the lock.
- Examine the value returned by MUTEX_$LOCK to determine whether you obtained the lock. If you have the lock, use the shared resource. Otherwise, perform an alternate action such as trying again or reporting an error.
- Use MUTEX_$UNLOCK to release the lock when you are through with the resource. Repeat the lock/unlock sequence as needed.
- Unmap the file that contains the mutex lock record when your program exits.

When several programs use a mutex lock, you must ensure that the mutex lock record has been initialized before it is used by any programs.

Remember that all programs using the same mutex lock record must be on the same node. The file containing the lock record can be on the same, or on another, node.

4.4. Initializing a Mutex Lock Record

Before you initialize a lock record, you must map a file to initialize the record in. Typically, you initialize the lock record in a file that also contains data. Programs can then access both the lock record and the data from a single file. However, if you prefer, you can initialize the lock record in a separate file.

To map the lock record file, use MS_$MAPL (to map an existing file) or MS_$CRMAPL (to create and map a new file.) When you map the file, specify a concurrency mode of MS_$NR_XOR_1W and an access type of MS_$WR. This concurrency/access combination obtains an exclusive write lock. No other programs can map the file while you are initializing the lock.

Map enough bytes to contain the mutex lock record and any data that the user programs will share. See Chapter 2 for more information on mapping files.

When you call MS_$MAPL or MS_$CRMAPL, you obtain a pointer to the mapped file. Use this pointer to refer to the mutex lock record. (This pointer is the same as the address of the file.)

After you map the file, call MUTEX_$INIT, supplying the mutex lock record as a parameter. Next, perform any other actions that are required in order to use the shared resource. Be sure to unmap the mutex lock record file before your program exits.

Figure 4-1 illustrates a program that initializes a lock record and then exits. This program prepares a file that others will use to process ticket sales for a concert. The file contains a mutex lock record and a count of the tickets sold.

First, the program defines the format for the shared file. This file contains:

- The mutex lock record, which is of type MUTEX_$LOCK_REC_T. (This data type is eight bytes long.)
- The count of tickets sold, which is an integer.

The program maps the shared file, initializes the mutex lock record, and sets the ticket count to zero. Then the program writes an informational message, unmaps the file, and exits. Now that the shared file is initialized, you can run the programs that accept ticket sales and update the ticket count.

```
PROGRAM mutex_init;

{ This program shows how to initialize a mutex lock record.
  It maps a file that contains a lock record and ticket sales
  count.  The program initializes the lock record and sales
  count and then exits. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/mutex.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
```

Figure 4-1. A Program That Initializes a Mutex Lock Record

```
TYPE
  ticket_sales_t =                        { ticket sales file }
    RECORD
    lock_record    : mutex_$lock_rec_t;   { mutex lock record }
    tickets_sold   : integer;             { counter          }
    END;

VAR
  status              : status_$t;          { status  code }
  ticket_info         : ^ticket_sales_t;
  length_mapped       : integer32;          { length of mapped data }
  tickets_wanted      : integer;
  wait_time           : time_$clock_t;
  lock                : boolean;

PROCEDURE check_status;                       { for error handling }

  BEGIN
  IF status.all <> status_$ok THEN
    BEGIN
    error_$print( status );
    pgm_$exit;
    END;
  END;

BEGIN

{ Map a data file containing the mutex lock. Use this file to hold data
  of type TICKET_SALES_T. }

ticket_info := ms_$mapl( 'ticket_sales_file',    { object to be mapped      }
                         17,                     { length of name           }
                         0,                      { first byte to map        }
                         sizeof( ticket_sales_t ),{ no. bytes to map        }
                         ms_$nr_xor_1w,          { locking mode             }
                         ms_$wr,                 { access type              }
                         true,                 { map length in 3rd parameter,
                                                 even if object is shorter  }
                         length_mapped,        { bytes mapped - returned    }
                         status );
check_status;

ticket_info^.tickets_sold := 0;

{ Initialize the mutex lock. }
mutex_$init ( ticket_info^.lock_record );

writeln( 'Mutex lock file has been initialized.  Start lock users now.' );

{ Unmap file and exit. }

ms_$unmap( ticket_info,
           sizeof( ticket_sales_t ),
           status );
check_status;
END.
```

Figure 4-1. A Program That Initializes a Mutex Lock Record (continued)

Figure 4-1 shows a program whose only function is to initialize a mutex lock record. If you execute this program before you start any program that uses the lock, you can ensure that the lock record is properly initialized. However, you can use other techniques to insure that a lock record is initialized before any program uses it. For example, you can ensure proper initialization if one program initializes the lock and then calls PGM_$INVOKE to invoke each program that uses the lock.

Another way to ensure proper initialization is to have each program that uses the lock determine whether the lock has been initialized. If the lock has not been initialized, any of the programs can initialize it. To do this, each program would:

- Try to map the file containing the lock record, obtaining an exclusive write lock.
- If the map operation succeeds, then no other program is currently mapping the lock record. (Only one program at a time can obtain an exclusive write lock.) Thus, the program should initialize the mutex lock record. However, the program must then unmap the file, and map the file again with a shared write lock.
- If the map operation fails, then another program is currently mapping the file. Thus, another program has already initialized the lock (or another program is in the process of initializing the lock.) In either case, you should try to map the file with a shared write lock. If your map operation fails, wait a few seconds and then try again.

4.5. Using Mutex Locks

Before you can use a mutex lock, you must map the file containing the lock record. When you map the file, specify a concurrency mode of MS_$COWRITERS and an access type of MS_$WR. This concurrency/access combination specifies a shared write lock. This allows you to read and write to the mutex lock record. In addition, other programs on your node can also map the mutex lock record with read and write access.

After you map the file, design a loop to:

- Perform work that requires you to use the shared resource.
- To use the resource, call MUTEX_$LOCK to request the mutex lock.
- If you get the lock, access the shared resource. Otherwise, perform an alternate action.
- Use MUTEX_$UNLOCK to release the lock.

When you are through, exit from the loop and unmap the mutex lock record file.

MUTEX_$LOCK accepts two parameters: the lock record and a time-out value. The time-out value indicates how long you want to wait for the lock. If you supply a negative value, you will wait indefinitely. MUTEX_$LOCK returns either TRUE or FALSE to indicate whether you obtained the lock.

NOTE: Whenever you call MUTEX_$LOCK, the system uses PFM_$INHIBIT to inhibit asynchronous faults. When you call MUTEX_$UNLOCK, the sytem uses PFM_$ENABLE to reenable them. See Section 4.6 for more information on fault handling.

Figure 4-2 shows a program that accepts concert ticket orders, using a shared ticket sales file to see if tickets are available. To process an order, the program requests a mutex lock on the ticket sales file. (The lock record is at the beginning of the ticket sales file.) When it has the lock, the program examines the ticket sales count to see if there are enough tickets to fill the order. If there are tickets, the program writes the new sales total and unlocks the file. The program accepts orders until the tickets are sold out.

If you run this program concurrently from different processes, each program will process orders using the same ticket file. However, only one program at a time will access the file.

```
PROGRAM mutex_user;

{ This program uses MUTEX calls to lock a file that contains ticket sales
  information.  When the program gets a mutex lock, it determines whether
  there are enough tickets to fill the ticket order.  If there are tickets,
  the program updates the ticket count and unlocks the file.  The program
  processes orders until tickets are sold out. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/mutex.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  TYPE
    ticket_sales_t =                      { ticket sales file }
      RECORD
      lock_record    : mutex_$lock_rec_t;  { mutex lock record }
      tickets_sold   : integer;            { counter          }
      END;

  VAR
    status              : status_$t;          { status  code }
    ticket_info         : ^ticket_sales_t;
    length_mapped       : integer32;          { length of mapped data }
    tickets_wanted      : integer;
    wait_time           : time_$clock_t;
    lock                : boolean;


  PROCEDURE check_status;                       { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;
```

Figure 4-2. A Program That Uses Mutex Locks

```
BEGIN
{ Map a shared data file containing the mutex lock.
  Use this file to hold data of type TICKET_SALES_T. }

ticket_info := ms_$mapl( 'ticket_sales_file',     { object to be mapped     }
                         17,                      { length of name          }
                         0,                       { first byte to map       }
                         sizeof( ticket_sales_t ), { no. of bytes to map    }
                         ms_$cowriters,           { concurrency mode        }
                         ms_$wr,                  { access type             }
                         true,                  { map length in 3rd parameter,
                                                  even if object is shorter  }
                         length_mapped,           { bytes mapped - returned }
                         status );
check_status;

{ Get system clock value for 30 seconds. }
cal_$sec_to_clock ( 30, wait_time );

{ Keep looping as long as you can get a lock and there are tickets left. }

WITH ticket_info^ DO

  BEGIN
  WHILE tickets_sold < 100 DO

    BEGIN {while }
    writeln ( 'Input the number of tickets you want : ' );
    readln ( tickets_wanted );

    { Get a mutex lock on the file.  If you get the lock, try
      to buy tickets.  Otherwise, assume there's a problem and exit.
      The theater holds 100 people.}

    lock := mutex_$lock( lock_record, wait_time );
    IF NOT lock THEN
      BEGIN
      writeln( 'Problem locking file.  Try again later.' );
      ms_$unmap( ticket_info,
                 sizeof( ticket_sales_t ),
                 status );
      check_status;
      RETURN;
      END;
    IF tickets_sold + tickets_wanted > 100  THEN
      writeln ( 'Only', 100 - tickets_sold:4, ' tickets left.' )
    ELSE
      BEGIN
      tickets_sold := tickets_sold + tickets_wanted;
      writeln ( ' You got', tickets_wanted:4, ' tickets.' );
      END;

    mutex_$unlock ( lock_record );
    END;
  END; { while }
```

Figure 4-2. A Program That Uses Mutex Locks (continued)

```
{ Unmap file and exit. }

ms_$unmap( ticket_info,
           sizeof( ticket_sales_t ),
           status );
check_status;

END.
```

Figure 4-2. A Program That Uses Mutex Locks (continued)

4.6. Mutex Locks and Fault Handling

As part of the MUTEX_$LOCK call, the system calls PFM_$INHIBIT to disable asynchronous faults (such as "quits".) As part of the MUTEX_$UNLOCK call, the system calls PFM_$ENABLE to re-enable these faults. The system performs these actions so that an asynchronous fault cannot abort your program while it is holding a mutex lock. If your program exits without releasing a lock, no other program will be able to obtain the lock. That is, the system does *not* automatically release a program's mutex lock when the program exits. Note that the system inhibits asynchronous faults only while you hold the lock, not while you are waiting for it.

If, while you hold a mutex lock, you call a procedure that may wait for a long time, you may want to restore the ability to use a "quit" to stop the program. To do this, temporarily re-enable asynchronous faults, as described below:

- Call MUTEX_$LOCK to get the mutex lock.
- Declare a clean-up handler that uses MUTEX_$UNLOCK to release the mutex lock. This clean-up handler will be in effect during the period in which you re-enable asynchronous faults.
- Re-enable asynchronous faults before you call the procedure. In this way, an asynchronous fault can interrupt the program.
- Call the procedure. If an asynchronous fault occurs, the program uses the clean-up handler to release the lock before exiting. If no asynchronous fault occurs, disable asynchronous faults when you return from the procedure.
- Release the clean-up handler.
- Call MUTEX_$UNLOCK to release the lock.

Figure 4-3 illustrates how to declare a clean-up handler and re-enable asynchronous faults while you have a mutex lock. See *Programming With General System Calls* for more information on clean-up handlers.

```
{ Make declarations and start the program. }
                        .
                        .
                        .

{ Get the mutex lock. }

lock := mutex_$lock( lock_record, wait_time );

{ Establish the clean-up handler. }

status := pfm_$cleanup ( handler_id );
IF ( status.all <> pfm_$cleanup_set ) THEN
   BEGIN
     mutex_$unlock ( lock_record ) ;
     pfm_$signal ( status );
   END;

{ Re-enable asynchronous faults. }

pfm_$enable;

{ Make a call that can be interrupted with a "quit."
  If a "quit" occurs, the cleanup-handler is executed. }
                        .
                        .
                        .

{ If no fault occurred, disable asynchronous faults and
  release the clean-up handler. }

pfm_$inhibit;
pfm_$release_cleanup ( handler_id );

{ Release the mutex lock. }

mutex_$unlock ( lock_record );
                        .
                        .
                        .
```

Figure 4-3. A Clean-Up Handler for Use After Calling MUTEX_$LOCK

You may also want to write a clean-up handler to handle synchronous faults (such as network failures) that may occur while you hold a mutex lock. If a synchronous fault can abort your program, be sure that a clean-up handler releases the program's mutex lock before the program exits. To do this:

- Call MUTEX_$LOCK to get the mutex lock.
- Declare a clean-up handler that uses MUTEX_$UNLOCK to release the mutex lock. Thus, if a synchronous fault occurs, the program will use the clean-up handler to release the lock before exiting.

- Perform the required work while you have the lock.
- Release the clean-up handler.
- Call MUTEX_$UNLOCK to release the lock.

See Figure 4-3 for an example of how to write a clean-up handler.

Chapter 5
Using Mailboxes

This chapter describes how to perform interprocess communication using mailboxes. It includes:

- An overview of message transmission using mailboxes.
- A summary of the MBX system calls and data types.
- A description of how to write a mailbox server and a client.
- A discussion of special considerations for sending long messages.
- A description of how to use mailbox eventcounts.
- A description of the MBX_HELPER.

5.1. Using Mailboxes To Transmit Messages

A mailbox is a file that two programs use to exchange information. A program that creates a mailbox is called a server. Other programs, called clients, can open mailbox channels in order to communicate with the server.

A mailbox has only one server, although a mailbox can have up to 255 channels that clients can open. A server can "service requests" from any client that has opened a channel to the mailbox. However, a client can communicate only with the server.

Figure 5-1 shows a server, a mailbox, and two clients that have opened mailbox channels. The server and the clients can get messages from the mailbox, and can put messages in. When the server puts a message into the mailbox, the server must indicate which channel the message is for; only the client using that channel can get the message out. When a client puts a message into the mailbox, only the server can get the message out.

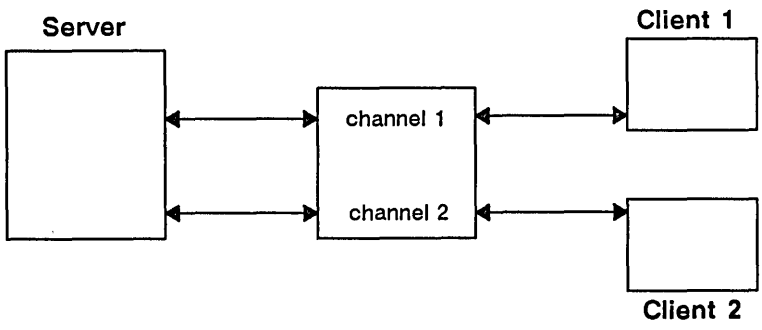

Figure 5-1. A Mailbox Server with Two Clients

A mailbox channel provides a guaranteed connection, called a virtual circuit, between a server and a client. Because it is a virtual circuit, a mailbox channel maintains the sequence for the messages that pass through it. Every mailbox channel has two buffers: one for the server and one for the client. Figure 5-2 illustrates a mailbox channel.

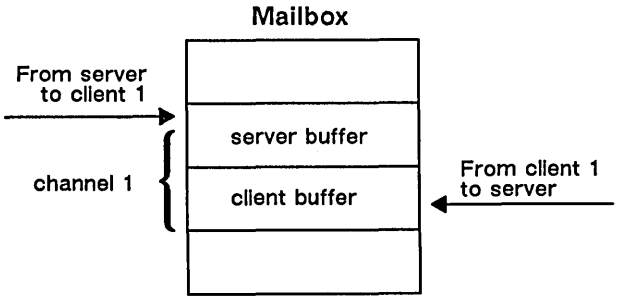

Figure 5-2. A Mailbox Channel

You can use mailboxes to communicate between programs on the same, or on different, nodes. However, if programs on different nodes use the same mailbox, they cannot both access the mailbox at the same time. Therefore, the DOMAIN system includes an MBX_HELPER program to assist the interprocess communication. You must run the MBX_HELPER on both the server's and client's nodes. See Section 5.8 for information on starting and using the MBX_HELPER.

You can use a mailbox for any task that involves data transfer between programs. For example, you can:

- Send data from a client to a server. The client sends data by putting it into the server's mailbox. The server then gets the data from the mailbox and processes it.

- Exchange information between a client and a server. The client sends a message that indicates a request for service. The server processes the request and then sends a reply to the client.

5.2. MBX System Calls, Insert Files, and Data Types

To send and receive mailbox messages, use MBX system calls. These calls invoke the mailbox manager, the system component that is responsible for mailboxes. Table 5-1 summarizes the MBX calls.

In order to use MBX calls, you must include the appropriate insert file in your program. The MBX insert files are:

/SYS/INS/MBX.INS.C	(for C)
/SYS/INS/MBX.INS.FTN	(for FORTRAN)
/SYS/INS/MBX.INS.PAS	(for Pascal)

Table 5-1. Summary of MBX Calls

Operation	Calls
Create mailboxes and open channels	MBX_$OPEN MBX_$CREATE_SERVER
Get messages from a mailbox	MBX_$GET_REC MBX_$GET_REC_CHAN MBX_$GET_REC_CHAN_SET MBX_$GET_CONDITIONAL MBX_$COND_GET_REC_CHAN MBX_$COND_GET_REC_CHAN_SET
Put messages into a mailbox	MBX_$PUT_CHR MBX_$PUT_CHR_COND MBX_$PUT_REC MBX_$PUT_REC_COND
Determine maximum message size	MBX_$SERVER_WINDOW MBX_$CLIENT_WINDOW
Use eventcounts	MBX_$GET_EC
Close mailboxes and deallocate channels	MBX_$CLOSE MBX_$DEALLOCATE

Some of the MBX calls require that you specify parameters using special DOMAIN data types. These data types include:

MBX_$CHAN_NUM_T	A mailbox channel number.
MBX_$CHAN_SET_T	A set of mailbox channel numbers.
MBX_$EC_KEY_T	An event associated with a mailbox.
MBX_$MTYPE_T	A mailbox message type.
MBX_$NAME_T	A mailbox name.
MBX_$MSG_HDR_T	A header for a mailbox message.
MBX_$SERVER_MSG_T	A mailbox message, as seen by a server. The message includes a header and a data section.

For complete information on MBX system calls and data types, see the *DOMAIN System Call Reference*.

5.3. Mailbox Messages

When a program sends a message to a mailbox, the message contains two parts:

- A 6-byte header that contains control information for the mailbox manager.
- Up to 32760 (MBX_$REC_DATA_MAX) bytes of user data.

The message header contains three fields, each of which is two bytes long. The fields include:

Message length — The total number of bytes in the message, including the header.

Message type — The purpose of a message. This field can contain one of the following values:

- MBX_$ACCEPT_OPEN_MT -- A response from a server accepting a client's open request.
- MBX_$CHANNEL_OPEN_MT -- A request from a client to open a channel to the mailbox.
- MBX_$DATA_MT -- A data transmission.
- MBX_$DATA_PARTIAL_MT -- A partial data transmission.
- MBX_$EOF_MT -- An end-of-transmission notice.
- MBX_$REJECT_OPEN_MT -- A response from a server, rejecting a client's open request.

Channel number — The channel of the client that sent the message, or that should receive the message. This field is two bytes long.

Although a message contains both a header and a data, a client gets and sends only the data section. In contrast, a server gets and sends both the header and data sections. Figure 5-3 shows a message as seen by a server and a client.

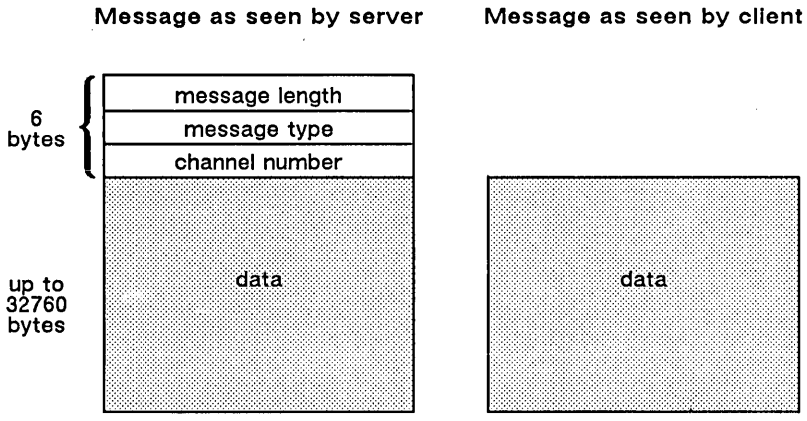

Figure 5-3. Mailbox Message Formats

The MBX insert files include constants that describe short and long messages. These constants are:

MBX_$MSG_MAX	A 1024-byte data section.
MBX_$SERV_MSG_MAX	A short message, including the 6-byte header and the 1024-byte data section.
MBX_$REC_DATA_MAX	A 32760-byte data section.
MBX_$REC_MSG_MAX	A long message, including the 6-byte header and the 32760-byte data section.

Although messages can be up to MBX_$REC_MSG_MAX (32766) bytes long, the following restrictions apply when you work with messages that are longer than 1158 bytes:

- If a server sends a long message to a client on a remote node, the remote node's MBX_HELPER must be able to accept the message. Use MBX_$SERVER_WINDOW to determine the largest message that the remote node's MBX_HELPER can accept. See Section 5.5.4 for more information on sending long messages.

- A mailbox client can optionally send data when using MBX_$OPEN to open a channel to a mailbox. However, the largest message you can send with MBX_$OPEN is 1024 bytes, even if the mailbox accepts larger messages. See Section 5.6.1 for more information.

The following sections describe how to define mailbox message buffers in servers and clients. Because servers and clients see different versions of a mailbox message, you must define different types of message buffers. You use a message buffer whenever you get a message from a mailbox, or whenever you send one.

5.3.1. Defining Message Buffers in a Server

When you define a message buffer in a server, you must include fields for the header and data sections of the message. In Pascal and C, you can use the predefined data type MBX_$MSG_HDR_T when you define message buffers for servers. MBX_$MSG_HDR_T is a record with the three fields in a message header:

CNT	A 2-byte integer giving the message length.
MT	A 2-byte integer giving the message type.
CHAN	A 2-byte integer giving the message channel.

The following example shows Pascal type definitions for a message and a pointer to a message:

```
TYPE
  server_rec_t =
    RECORD          { a server message }
    mbx_hdr   : mbx_$msg_hdr_t;   { header has 3 fields:
                                      cnt  - message length
                                      mt   - message type
                                      chan - message channel }
    msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
    END;

  server_rec_ptr_t  =  ^server_rec_t; { pointer to a server message }
```

In FORTRAN, you can define a server message header as a three-element INTEGER*2 array; declare the data as a character array. For example:

```
INTEGER*2 header(3)
CHARACTER message(1030)
EQUIVALENCE (header(1), message(1))
```

The server messages in the previous examples can contain up to MBX_$MSG_MAX (1024) bytes of data. The total message is MBX_$SERV_MSG_MAX (1030) bytes long, including the header. However, you can define the data portion of the message to be up to MBX_$REC_DATA_MAX (32760) bytes long.

5.3.2. Defining Message Buffers in a Client

A mailbox client sees only the data section of a message; the mailbox manager appends or removes the appropriate header. Therefore, you need only define a message buffer that is large enough to hold the messages you are sending and receiving. The largest possible data message (MBX_$REC_DATA_MAX) is 32760 bytes. You can use MBX_$MSG_MAX to define a short (1024-byte) data message.

5.4. Steps For Using Mailboxes

To communicate using mailboxes, follow these steps:

- Write the mailbox server, the program that creates the mailbox. In addition to creating the mailbox, the server defines the size of the mailbox and the number of channels that clients can open. Section 5.5 describes how to write a mailbox server.

- Write the mailbox client (or clients), the programs that use the mailbox. Each client must open its own channel to the mailbox. Section 5.6 describes how to write a mailbox client.

- Start the mailbox helper (MBX_HELPER), if required. If the server and client(s) will run on different nodes, you must be sure that MBX_HELPER is running on each node. Section 5.8 explains how to check for and start MBX_HELPER.

- Start the server program; clients cannot open channels until the server creates the mailbox.

- Start the client program(s); each client can now open a channel to the mailbox and send messages.

In general, servers and clients use MBX calls in the following order:

1. The server creates the mailbox with MBX_$CREATE_SERVER.

2. A potential client calls MBX_$OPEN to open a channel to the mailbox.

3. The server uses one of the MBX_$GET calls to get the open request. To accept the request, the server uses one of the MBX_$PUT calls to send a message with the type MBX_$ACCEPT_OPEN_MT; to reject the request, the server sends a message with the type MBX_$REJECT_OPEN_MT.

4. The client's MBX_$OPEN call returns and the status code indicates whether the call succeeded. If the server accepted the message, the MBX_$OPEN call returns with STATUS_$OK. If the server did not accept the message, the client's call returns with MBX_$OPEN_REJECTED.

5. The server and client exchange data using MBX_$PUT and MBX_$GET calls. Depending on the call, either complete messages (MBX_$DATA_MT) or partial messages (MBX_$DATA_PARTIAL_MT) are sent.

6. Either the server or a client ends the communication. The server can call MBX_$CLOSE or MBX_$DEALLOCATE; MBX_$CLOSE closes the mailbox, while MBX_$DEALLOCATE deallocates a channel and frees it for use by another client. If a server closes a mailbox or deallocates a channel while a client is still using it, the client gets an error message the next time it tries to use the mailbox.

   A client can stop using a channel by calling MBX_$CLOSE. This sends the server an MBX_$EOF_MT message. However, MBX_$CLOSE (when called from a client) does not close the mailbox or deallocate the channel. The server must receive the client's message and either deallocate the channel or close the mailbox.

For more information on how you send messages between servers and clients, refer to Sections 5.5 and 5.6.

5.5. Writing a Mailbox Server

A mailbox server has the following general design. The server:

- Creates the mailbox.

- Enters a loop to get messages from the mailbox and service the requests. The server should be able to handle open requests, data transmissions, partial data transmissions, and end-of-transmission notices.

- Exits from the loop and closes the mailbox when a terminating condition has been satisfied.

The following sections show how to write a mailbox server. Section 5.5.1 shows how to create and close a mailbox; Section 5.5.2 shows how to get messages; Section 5.5.3 shows how to respond to messages. Note that each section uses examples from the same sample program.

5.5.1. Creating and Closing a Mailbox

To create a mailbox, use the call MBX_$CREATE_SERVER; to close a mailbox, use the call MBX_$CLOSE.

MBX_$CREATE_SERVER specifies the name for the mailbox, the maximum size of the buffers for each channel, the number of channels that can be opened, and the name of a "handle" for the mailbox. The handle is an identifier for the mailbox.

Note that a mailbox is a special kind of file; therefore, specify the mailbox name as a DOMAIN pathname. If you use the name of an existing file when you call MBX$_CREATE_SERVER, the mailbox manager deletes the contents of the file before creating your mailbox.

When a program (either a server or a client) sends messages to a mailbox, the messages are stored in a buffer for the appropriate channel. The messages remain in the buffer until another program reads them. A buffer continues to accept messages until it is full. If a program tries to send a message when the buffer is full, the mailbox manager either returns an error or suspends the calling program (depending on the MBX call you used). Each channel has two buffers: one for the server and one for the client.

The buffer size defines the number of message bytes that the server and a client can each store in a channel. For example, if you specify a buffer of 2000, the mailbox manager allocates 4000 bytes per channel: 2000 bytes for the server and 2000 bytes for the client.

You must specify a buffer size of at least 64 bytes; the maximum buffer size is 32767. Be sure to specify a buffer that is at least as large as the largest message you plan to send from either a server or a client.

A mailbox can have between 1 and 255 channels. Remember that buffer space is allocated for both the server and the client using a channel. Thus, if you specify a buffer size of 2000 and a channel number of 4, the mailbox will be created with 16,000 bytes of buffer space. (That is, there will be 4,000 bytes apiece for four channels.) Specify enough channels to handle open requests from each client you anticipate.

The mailbox handle is a pointer to the mailbox. The value for this pointer is returned when you call MBX_$CREATE_SERVER. Most MBX calls use the handle, rather than the name, to refer to the mailbox.

Note that, in a secure network, a mailbox's access control list (ACL) is determined by the ACL of the directory in which it is created. If clients on other nodes use the mailbox, you must be sure that the server node's MBX_HELPER has read and write access to the mailbox. See Section 5.8.3 for more information.

Figure 5-4 shows a server that uses MBX_$CREATE_SERVER to create a mailbox named TEST_MAILBOX on the local node. This mailbox allows up to 255 open channels, with 2060 bytes in each channel. The program uses MBX_$CLOSE to close the mailbox.

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles
  requests from clients.  The server handles open
  requests, close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    done;

  CONST
    mbx_bufsize = mbx_$serv_msg_max;   { channel buffer size         }
    mbx_maxchan  = mbx_$chn_max;       { no. mailbox channels - 255  }
    mbx_name     = 'test_mailbox';     { mailbox name                }
    mbx_namelen  = sizeof( mbx_name ); { length of mailbox name      }

  TYPE
              .
              .
              .
  VAR

    mbx_handle    : univ_ptr;          { a pointer to the mailbox }
    status        : status_$t;         { a status code }

  PROCEDURE check_status;  .                    { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;

  BEGIN {program test_server }

  { Create the mailbox. }

  mbx_$create_server( mbx_name,     { name             }
                      mbx_namelen,  { name length      }
                      mbx_bufsize,  { buffer size      }
                      mbx_maxchan,  { maximum channels }
                      mbx_handle,   { handle           }
                      status );
  check_status;

  writeln( 'Mailbox ', mbx_name, ' was successfully opened.' );

  { Get messages from clients. }
              .
              .
              .
```

Figure 5-4. A Server That Creates and Closes a Mailbox

```
  { Close mailbox and exit. }

done:

  mbx_$close( mbx_handle,
              status );
  check_status;
  writeln ( 'The mailbox has been closed.' );

  END.    {program test_server }
```

Figure 5-4. A Server That Creates and Closes a Mailbox (continued)

5.5.2. Getting Messages

You can use two types of MBX calls to get messages from a mailbox. Nonconditional calls suspend your program until a message is available and the call can complete; conditional calls return immediately with the completion status MBX_$CHANNEL_EMPTY if no message is available.

Each MBX get call has both a conditional and a nonconditional form. Table 5-2 summarizes these calls.

Table 5-2. Summary of MBX Get Calls

	Nonconditional Call	Conditional Call
Get a message from any channel	MBX_$GET_REC	MBX_$GET_CONDITIONAL
Get a message from a specified channel	MBX_$GET_REC_CHAN	MBX_$COND_GET_REC_CHAN
Get a message from a specified group of channels	MBX_$GET_REC_CHAN_SET	MBX_$COND_GET_REC_CHAN_SET

For each of these calls, supply the handle for the mailbox and the name of a buffer where the mailbox manager can place the returned message. Note that you must also supply an output parameter so these calls can return the actual location of the retrieved message; the calls do not always place the message in the buffer you supply. Always use the output parameter to refer to the message.

Figure 5-5 shows a server that uses MBX_$GET_REC to search each channel, in sequence, until it finds a message to retrieve. If no messages are waiting, the server waits until a message is available. (See Section 5.7 for information on using eventcounts with the conditional MBX calls.)

When the server gets a message, it interprets the message type and branches to the appropriate section. The server can process open requests, end-of-transmission notices, data, and partial data transmissions. To determine the message type, the server looks in the second field of the message header.

In this example, the GET_MESSAGE_LOOP is controlled by a counter that keeps track of the number of open channels to the mailbox; the program continues in the loop until all channels have been closed. Note, however, that some applications may require that the server run continuously, regardless of whether there are open channels.

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles requests from
  clients.  It handles open and close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    get_message_loop,
    done;

  CONST
    mbx_bufsize = mbx_$serv_msg_max;   { channel buffer size              }
    mbx_maxchan  = mbx_$chn_max;       { no. mailbox channels - 255       }
    mbx_name     = 'test_mailbox';     { mailbox name                     }
    mbx_namelen  = sizeof( mbx_name ); { length of mailbox name           }
    srv_msg_len  = mbx_$serv_msg_max;  { length of buffer that server can
                                         use to receive messages          }

  TYPE
    server_rec_t =
      RECORD                          { server message }
      mbx_hdr   : mbx_$msg_hdr_t;     { header has 3 fields:
                                          cnt  - message length
                                          mt   - message type
                                          chan - message channel }
      msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
      END;
    server_rec_ptr_t  =
      ^server_rec_t;  { pointer to a server message }

  VAR
    mbx_handle    : univ_ptr;          { a pointer to the mailbox            }
    status        : status_$t;         { a status code                       }
    srv_msg_buf   : server_rec_t;      { a buffer that the server can use to
                                         receive messages                    }
    mbx_retptr    : server_rec_ptr_t;  { a pointer to the buffer where MBX
                                         placed a retrieved message    }
    mbx_retlen    : integer32;         { length of a retrieved message }
    send_msg_buf  : server_rec_t;      { a buffer where the server places
                                         a message to send }
    open_chan     : integer16;         { number of open channels to
                                         the mailbox }
```

Figure 5-5. A Server That Gets Messages

```
  PROCEDURE check_status;                          { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;

  BEGIN {program test_server }

  {  Initialize variables. }

  open_chan := 0;

  { Create the mailbox. }

  mbx_$create_server( mbx_name,      { name             }
                      mbx_namelen,   { name length      }
                      mbx_bufsize,   { buffer size      }
                      mbx_maxchan,   { maximum channels }
                      mbx_handle,    { handle           }
                      status );
  check_status;

  writeln( 'Mailbox ', mbx_name, ' was successfully opened.' );

  { Keep getting messages until there are no more clients. }

get_message_loop:

  REPEAT
    mbx_$get_rec( mbx_handle,
                  addr( srv_msg_buf ),  { where message may be received }
                  srv_msg_len,          { length of message buffer      }
                  mbx_retptr,           { where message is received     }
                  mbx_retlen,           { message length                }
                  status );
    check_status;

    WITH mbx_retptr^ DO
      BEGIN
      writeln( 'Message received from channel ', mbx_hdr.chan:4 );

      CASE mbx_hdr.mt OF

      { If message is an open channel request, accept it.
        Also, keep track of the number of open channels. }

        mbx_$channel_open_mt :
                        .
                        .
                        .
```

Figure 5-5. A Server That Gets Messages (continued)

```
        { If message is a close channel request, deallocate
          the channel and decrement the open channel count. }

        mbx_$eof_mt :
                        .
                        .
                        .

        { If message is a data transmission or
          a partial data transmission, process the data. }

        mbx_$data_mt :
                        .
                        .
                        .

        mbx_$data_partial_mt :
                        .
                        .
                        .

        OTHERWISE
          writeln ( 'Invalid message type' );

        END;   {case statement  }
      END;      {with statement }
  UNTIL open_chan = 0;

  { Close mailbox and exit. }

done:

  mbx_$close( mbx_handle,
              status );
  check_status;
  writeln ( 'The mailbox has been closed.' );

  END.   {program test_server }
```

Figure 5-5. A Server That Gets Messages (continued)

5.5.3. Responding to Messages

After a server gets a message, it must perform the appropriate action. Usually, this involves sending a reply to the client. A server uses one of the following PUT calls to send replies:

MBX_$PUT_REC	Puts a record into the mailbox. If the receiving channel is full, the program is suspended until there is room to accept the message.
MBX_$PUT_REC_COND	Puts a record into the mailbox if there is room in the receiving channel. If the channel is full, the call returns the status MBX_$NO_ROOM_IN_CHANNEL.

A server should be able to handle open requests, end-of-transmission notices, data, and partial data transmissions. The following sections describe how to respond to these message types.

Section 5.5.3.1 describes open requests, Section 5.5.3.2 desribes end-of-transmission notices, and Section 5.5.3.3 describes data and partial data messages.

5.5.3.1. Open Requests

You receive an MBX_$CHANNEL_OPEN_MT message whenever a client calls MBX_$OPEN and you have an available channel. Note that if you have no available channels, the mailbox manager does not transmit the open request to you; the mailbox manager immediately returns MBX_$NO_MORE_CHANNELS to the client's open call.

To respond to the open request, send one of the messages MBX_$ACCEPT_OPEN_MT or MBX_$REJECT_OPEN_MT back to the client. This causes the mailbox manager to return either STATUS_$OK or MBX_$OPEN_REJECTED to the client.

To send the message, use one of the calls MBX_$PUT_REC or MBX_$PUT_CONDITIONAL. The first call suspends the calling process until there is room in the channel for the message; the second call returns immediately with the completion status MBX_$NO_ROOM_IN_CHANNEL if there is no room. To use either call, specify the mailbox handle and a pointer to the message you are sending. Note that whenever you send a message from a server, you must include the message header.

Whenever you accept an open request, check the length of the message to see whether data is included. If the message is longer than the standard message header (MBX_$SERV_MSG_HDR_LEN), then the open request contains data that you should process.

Figure 5-6 shows a server that accepts an open request and checks whether the message contains data. If the message contains data, the server calls a subroutine to process the data.

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles requests from
  clients.  It handles open and close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    get_message_loop,
    done;

  CONST
              .
              .
              .
    srv_msg_len  = mbx_$serv_msg_max;  { length of buffer that server can
                                         use to receive messages }
```

Figure 5-6. A Server That Responds to Open Requests

```
TYPE
  server_rec_t =
    RECORD                          { server message }
    mbx_hdr  : mbx_$msg_hdr_t;      { header has 3 fields:
                                        cnt  - message length
                                        mt   - message type
                                        chan - message channel }
    msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
    END;
  server_rec_ptr_t  =
    ^server_rec_t;  { pointer to a server message }

VAR
  mbx_handle    : univ_ptr;          { a pointer to the mailbox            }
  status        : status_$t;         { a status code                       }
  srv_msg_buf   : server_rec_t;      { a buffer that the server can use to
                                       receive messages                    }
  mbx_retptr    : server_rec_ptr_t;  { a pointer to the buffer where MBX
                                       placed a retrieved message          }
  mbx_retlen    : integer32;         { length of a retrieved message       }
  send_msg_buf  : server_rec_t;      { a buffer where the server places
                                       a message to send                   }
  open_chan     : integer16;         { number of open channels to
                                       the mailbox                         }

PROCEDURE check_status;             { for error handling }
            .
            .
            .
PROCEDURE process_data;
            .
            .
            .
  END;

BEGIN {program test_server }

{Initialize variables. }

open_chan := 0;

{ Create mailbox and get handle. }
            .
            .
            .
 { Keep getting messages until there are no more clients. }

REPEAT
  mbx_$get_rec( mbx_handle,
                addr( srv_msg_buf ),  { where message may be received }
                srv_msg_len,          { length of message buffer      }
                mbx_retptr,           { where message is received     }
                mbx_retlen,           { message length                }
                status );
  check_status;
```

Figure 5-6. A Server That Responds to Open Requests (continued)

```
  WITH mbx_retptr^ DO
    BEGIN
    writeln( 'Message received from channel ', mbx_hdr.chan:4 );

    CASE mbx_hdr.mt OF

    { If message is an open channel request, accept it.
      Also, keep track of the number of open channels. }

      mbx_$channel_open_mt :
        BEGIN
        send_msg_buf.mbx_hdr.cnt := mbx_$serv_msg_hdr_len;
                        {there's no data in this message }
        send_msg_buf.mbx_hdr.mt  := mbx_$accept_open_mt;
                        {type of message you are sending }
        send_msg_buf.mbx_hdr.chan := mbx_hdr.chan;
                        {channel to send to }

        mbx_$put_rec( mbx_handle,
                      addr( send_msg_buf ),     { message to send   }
                      mbx_$serv_msg_hdr_len,    { length of message }
                      status );
        check_status;

        writeln ( ' Open request from channel ',
                  mbx_hdr.chan:4, ' has been accepted.' );

        open_chan := open_chan + 1;

        IF mbx_retlen <> mbx_$serv_msg_hdr_len THEN
          process_data;
        END;
            .
            .
            .
      END;    {case statement }
    END;       {with statement }
UNTIL open_chan = 0;

{ Close mailbox and exit.}
            .
            .
            .
END.    {program test_server }
```

Figure 5-6. A Server That Responds to Open Requests (continued)

5.5.3.2. End-of-Transmission Notices

When you receive a message of type MBX_$EOF_MT, call MBX_$DEALLOCATE to free the channel for use by other clients. Specify the channel by using the value in the third field of the message header.

Figure 5-7 shows a server that deallocates a channel after receiving an MBX_$EOF_MT message. The server also decrements a channel counter. When there are no open channels to the mailbox, the program exits.

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles requests from
  clients.  It handles open and close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    get_message_loop,
    done;

  CONST
              .
              .
              .
    srv_msg_len  = mbx_$serv_msg_max;  { length of buffer that server can
                                         use to receive messages }
  TYPE
    server_rec_t =
      RECORD                          { server message }
      mbx_hdr   : mbx_$msg_hdr_t;     { header has 3 fields:
                                            cnt  - message length
                                            mt   - message type
                                            chan - message channel }
      msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
      END;
    server_rec_ptr_t  =
      ^server_rec_t;  { pointer to a server message }

  VAR

    mbx_handle    : univ_ptr;          { a pointer to the mailbox }
    status        : status_$t;         { a status code }
    srv_msg_buf   : server_rec_t;      { a buffer that the server can use to
                                         receive messages }
    mbx_retptr    : server_rec_ptr_t;  { a pointer to the buffer where MBX
                                         placed a retrieved message }
    mbx_retlen    : integer32;         { length of a retrieved message }
    send_msg_buf  : server_rec_t;      { a buffer where the server places
                                         a message to send }
    open_chan     : integer16;         { number of open channels to
                                         the mailbox }

  PROCEDURE check_status;                    { for error handling }
              .
              .
              .

  PROCEDURE process_data;
              .
              .
              .
    END;
```

Figure 5-7. A Server That Responds to End-of-Transmission Notices

```
  BEGIN {program test_server }

  { Initialize variables. }
  open_chan := 0;

  { Create mailbox and get handle. }
              .
              .
              .
  { Keep getting messages until there are no more clients. }

get_message_loop:

  REPEAT
    mbx_$get_rec( mbx_handle,
                  addr( srv_msg_buf ),  { where message may be received }
                  srv_msg_len,          { length of message buffer      }
                  mbx_retptr,           { where message is received     }
                  mbx_retlen,           { message length                }
                  status );
    check_status;

    WITH mbx_retptr^ DO
      BEGIN
      writeln( 'Message received from channel ', mbx_hdr.chan:4 );

      CASE mbx_hdr.mt OF

        { If message is a close channel request, deallocate
          the channel and decrement the open channel count. }

        mbx_$eof_mt :
          BEGIN
          mbx_$deallocate( mbx_handle,
                           mbx_hdr.chan,  { channel number }
                           status );
          check_status;

          writeln ( 'Channel ', mbx_hdr.chan:4,' was deallocated.' );

          open_chan := open_chan - 1;

          IF open_chan = 0 THEN
            GOTO done;
          END; { mbx_hdr.mt}
          .
          .
          .
        END;   {case statement  }
      END;      {with statement }
  UNTIL open_chan = 0;

  { Close mailbox and exit. }

  END.   {program test_server }
```

Figure 5-7. A Server That Responds to End-of-Transmission Notices (continued)

5.5.3.3. Data and Partial Data Transmissions

You receive an MT_$DATA_MT message when a client sends a data transmission; you receive an MT_$DATA_PARTIAL_MT message when a client sends partial data. When you receive one of these message types, process the data portion of the message. Send a return message if one is required by the application; for some applications, you may need to send a message to another client. You may find it useful to write a subprogram to perform the data processing.

NOTE: When a server uses any of the MBX_$GET calls, the call returns STATUS_$OK for both data and partial data messages. This is in contrast to how MBX_$GET calls act when called from clients. When a client gets a data message, the MBX_$GET call returns STATUS_$OK; when a client gets a partial data message, the call returns MBX_$PARTIAL_RECORD.

Figure 5-8 shows a server that processes data transmissions by displaying the data on the node's monitor. The server uses the subroutine PROCESS_DATA to find the data portion of the message and write it to the screen. The server also sends a return message to indicate that the data was written. This example processes complete and partial data transmissions in the same way. However, you may want to design special methods to concatenate data from partial data transmissions.

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles requests from
  clients.  It handles open and close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    get_message_loop,
    done;

  CONST      .
             .
             .
    srv_msg_len  = mbx_$serv_msg_max;  { length of buffer that server can
                                         use to receive messages }
  TYPE
    server_rec_t =
      RECORD                          { server message  }
      mbx_hdr   : mbx_$msg_hdr_t;     { header has 3 fields:
                                            cnt  - message length
                                            mt   - message type
                                            chan - message channel }
      msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
      END;
    server_rec_ptr_t  =
      ^server_rec_t;  { pointer to a server message }
```

Figure 5-8. A Server That Responds to Data and Partial Data Transmissions

```
VAR
  mbx_handle    : univ_ptr;          { a pointer to the mailbox }
  status        : status_$t;         { a status code }
  srv_msg_buf   : server_rec_t;      { a buffer that the server can use to
                                       receive messages }
  mbx_retptr    : server_rec_ptr_t;  { a pointer to the buffer where MBX
                                       placed a retrieved message    }
  mbx_retlen    : integer32;         { length of a retrieved message }
  send_msg_buf  : server_rec_t;      { a buffer where the server places
                                       a message to send }
  open_chan     : integer16;         { number of open channels to
                                       the mailbox }

PROCEDURE check_status;              { for error handling }
            .
            .
            .
PROCEDURE process_data;

  CONST
    reply       = 'Message written.';
    reply_len   = sizeof( reply );

  VAR
    reply_array : ARRAY [ 1..reply_len ] OF char;
    i           : integer;           { a counter }

  BEGIN
  { Display the client's message. }

  writeln ( mbx_retptr^.msg_data : mbx_retlen
          - mbx_$serv_msg_hdr_len );

  { Construct a header for a return message. }

  send_msg_buf.mbx_hdr.cnt   := reply_len
                                + mbx_$serv_msg_hdr_len;
  send_msg_buf.mbx_hdr.mt    := mbx_$data_mt;
  send_msg_buf.mbx_hdr.chan  := mbx_retptr^.mbx_hdr.chan;

  { Construct the data portion of a message. }

  reply_array                := reply;
  FOR i := 1 TO reply_len DO
    send_msg_buf.msg_data[i] := reply_array[i];

  { Send the return message. }
  mbx_$put_rec( mbx_handle,
                addr( send_msg_buf ),
                send_msg_buf.mbx_hdr.cnt,
                status );
  check_status;
  END;
```

Figure 5-8. Server That Responds to Data and Partial Data Transmissisons (cont.)

```
  BEGIN {program test_server }

  {Initialize variables. }

  open_chan := 0;

  { Create the mailbox and get a handle. }
              .
              .
              .
  { Keep getting messages until there are no more clients. }

get_message_loop:

  REPEAT
    mbx_$get_rec( mbx_handle,
                  addr( srv_msg_buf ),  { where message may be received }
                  srv_msg_len,          { length of message buffer      }
                  mbx_retptr,           { where message is received     }
                  mbx_retlen,           { message length                }
                  status );
    check_status;

    WITH mbx_retptr^ DO
      BEGIN
      writeln( 'Message received from channel ', mbx_hdr.chan:4 );

      CASE mbx_hdr.mt OF
              .
              .
              .
        { If message is a data transmission or
          a partial data transmission, process the data. }

        mbx_$data_mt :
          process_data;

        mbx_$data_partial_mt :
          process_data;

        OTHERWISE
          writeln ( 'Invalid message type' );

        END;   {case statement }
      END;      {with statement }
  UNTIL open_chan = 0;

  { Close mailbox and exit. }
              .
              .
              .
  END.   {program test_server }
```

Figure 5-8. Server That Responds to Data and Partial Data Transmissisons (cont.)

5.5.4. Sending Long Messages

When a server puts a message into a mailbox and the message is intended for a client on a remote node, the message must pass through a system mailbox maintained by the client node's MBX_HELPER. (See Section 5.8 for more information on sending messages when servers and clients are on different nodes.) The buffer size for this mailbox is defined when the client node's MBX_HELPER is started. By default, a system mailbox has channel buffers that can hold 1158 bytes of data.

However, a server can create a mailbox with buffers that hold up to 32767 bytes. Thus, a server might create a buffer that is larger than the buffer for a system mailbox on a remote node. If a server creates such a mailbox, and then tries to send a message that is larger than the system mailbox buffer on the remote node, the send operation will fail.

To prevent this type of error, a server can call MBX_$SERVER_WINDOW to determine the buffer size for the mailbox maintained by the MBX_HELPER on a remote client's node. If the remote node's mailbox is too small, the server can perform one of the following actions:

- Separate a long message into pieces that are small enough to fit in the remote node's system mailbox.

- Report the problem so you can change the buffer size for the system mailbox on the remote node. To change this buffer size, stop the MBX_HELPER on the client's node. Then restart the MBX_HELPER, using the -DATASIZE option to specify a larger buffer size. See Section 5.8.2 for more information.

See the *DOMAIN System Call Reference* for more information on using MBX_$SERVER_WINDOW.

Note that, even if you send a message that is smaller than the remote node's mailbox buffer, a PUT call can fail if the buffer is full. For example, if you send a new message before the client got your previous one, the buffer may be full and your call may fail.

5.5.5. Closing a Channel

A server can close a channel even if a client is still using the channel. To close an active channel, use one of the following methods:

- Deallocate the channel with MBX_$DEALLOCATE.

- Send a message of type MBX_$EOF_MT to the client using the channel.

When you deallocate the channel, you free it for use by another client. When the affected client tries to use the mailbox, the client receives a completion status of MBX_$CHANNEL_NOT_OPEN.

If you send a message of type MBX_$EOF_MT to a client, the client receives a completion status of MBX_$EOF when the client tries to read the message. In such a case, the client should call MBX_$CLOSE to close the channel.

5.6. Writing a Mailbox Client

A mailbox client has the following general design. The client:

- Opens a channel to the mailbox.
- Enters a loop to send messages to the mailbox and get replies.
- Exits from the loop and closes the channel to the mailbox when a terminating condition has been satisfied.

The following sections describe how to write a mailbox client. Section 5.6.1 describes how to open and close channels; Section 5.6.2 describes how to send and receive messages. Each section uses examples from the same sample program.

NOTE: A client can use STREAM calls to open a channel to a mailbox, get and send messages, and close the channel. See *Programming With General System Calls* for more information on using stream calls.

5.6.1. Opening and Closing Channels

To open a channel to a mailbox, use MBX_$OPEN; to close a channel, use MBX_$CLOSE.

MBX_$OPEN specifies the name of the mailbox you want to open a channel to, and gives a name for the mailbox handle. The mailbox handle is a pointer to the mailbox; the mailbox manager returns a value for the handle when you call MBX_$OPEN.

When you open a channel, you can optionally include data to be sent if the open request is successful. To include data, specify a pointer to the buffer that contains the data when you call MBX_$OPEN. Also include the length of the data you are sending. Note, however, that you can send only 1024 bytes of data with an MBX_$OPEN call, even if the mailbox buffer can handle larger messages.

If you do not include data with MBX_$OPEN, specify a nil pointer and a data length of zero. Use the following to specify a nil pointer:

Pascal	NIL
FORTRAN	0
C	NULL

When you call MBX_$OPEN, the mailbox manager sends the server an open request (MBX_$CHANNEL_OPEN_MT). When the server responds, your MBX_$OPEN call returns a status code to indicate whether the server accepted your open request. Test this status code to see whether the call was successful; MBX_$OPEN returns STATUS_$OK if the server accepted your open request.

To indicate that you have finished using a channel, call MBX_$CLOSE. When you call MBX_$CLOSE, the mailbox manager sends the server an MBX_$EOF_MT message. The server must deallocate the channel to free it for use by other clients. Note that an MBX_$CLOSE call from a client does not close a mailbox.

Figure 5-9 shows a client that opens a channel to the mailbox TEST_MAILBOX on the local node. When the client has finished using the mailbox, it informs the server by calling MBX_$CLOSE.

```
PROGRAM mbx_client ( input,output );

{ This mailbox client opens a channel to a mailbox.  The client
  sends messages to the server, and gets the server's reply. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  CONST
    mbx_name    = 'test_mailbox';   { mailbox name  }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name }

  VAR
    mbx_handle  : univ_ptr;         { mailbox handle }
    status      : status_$t;        { a status code  }

  PROCEDURE check_status;           { for error handling }
              .
              .
              .

  BEGIN

{ Open a channel to the mailbox. }

  mbx_$open ( mbx_name,
              mbx_namelen,
              NIL,
              0,
              mbx_handle,
              status );
  check_status;

{ Send and receive messages.}
              .
              .
              .
{ Close the channel. }

  mbx_$close( mbx_handle,
              status );

  check_status;

  END.
```

Figure 5-9. A Client That Opens and Closes Channels

5.6.2. Sending and Receiving Messages

To send messages to the server, use one of the following calls:

MBX_$PUT_REC	Puts a record into the mailbox. If the receiving channel is full, the program is suspended until there is room to accept the message. This call sends a message with the type MBX_$DATA_MT.
MBX_$PUT_CHR	Puts a partial message into the mailbox. This call is equivalent to MBX_$PUT_REC, except that it sends a partial data message with the type MBX_$DATA_PARTIAL_MT.
MBX_$PUT_REC_COND	Puts a record into the mailbox if there is room in the receiving channel. If the channel is full, the call returns the status MBX_$NO_ROOM_IN_CHANNEL. This call sends a message with the type MBX_$DATA_MT.
MBX_$PUT_CHR_COND	Puts a partial message into the mailbox if there is room in the receiving channel. If the channel is full, the call returns the status MBX_$NO_ROOM_IN_CHANNEL. This call sends a message with the type MBX_$DATA_PARTIAL_MT.

When you use these calls, specify the mailbox handle and a pointer to the buffer containing the message. Note that you send only the data portion of a message from a client; the mailbox manager appends the message header.

NOTE: When you send a message, the mailbox's channel buffer must be large enough to hold your message. (The server defined the buffer size when it called MBX_$CREATE_SERVER.) Thus, the mailbox buffer defines the largest message you can send. If you do not know the mailbox buffer size, call MBX_$CLIENT_WINDOW to obtain it. See the *DOMAIN System Call Reference* for more information on using MBX_$CLIENT_WINDOW.

After you send a message to the server, you usually want a response. To get a response, use either MBX_$GET_REC or MBX_$GET_CONDITIONAL. MBX_$GET_REC suspends the calling process until a record is available, whereas MBX_$GET_CONDITIONAL returns immediately with the status MBX_$CHANNEL_EMPTY. When you use either of these calls, specify the mailbox handle and the name of a buffer where the mailbox manager can place the returned message. As described in Section 5.5.2, the mailbox manager does not always place messages in the buffer you supply. However, the get call always returns the location of the message in one of the output parameters. Use this returned location to obtain the message.

Note that when you get a data message, the MBX_$GET calls return the status STATUS_$OK. However, when you get a partial data message, the calls return the status MBX_$PARTIAL_RECORD.

Figure 5-10 shows a mailbox client that establishes a loop to send and receive mailbox messages. First, the client prompts for user input and places this input into a mailbox. Then the client gets a reply from the server and displays the reply. The client continues in this loop until you enter CTRL/Z. The client checks for both STATUS_$OK and MBX_$PARTIAL_RECORD in its error-checking routine.

```
PROGRAM mbx_client ( input,output );

{ This mailbox client opens a channel to a mailbox.  The client
  sends messages to the server, and gets the server's reply. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    done;

  CONST
    mbx_name    = 'test_mailbox';   { mailbox name             }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name }
    buf_len     = mbx_$msg_max;     { length of data buffer    }
    msg_buf_len = mbx_$msg_max;     { length of message buffer }
    prompt_str  = 'Enter a new message: ';

  TYPE
    msg_t       =
      ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =
      ^msg_t;     { pointer to a client message }

  VAR
    data_buf    : msg_t;            { buffer for data to be sent     }
    mbx_handle  : univ_ptr;         { mailbox handle                 }
    status      : status_$t;        { a status code                  }
    msg_buf     : msg_t;            { a buffer that the client
                                      can use to receive data        }
    msg_retptr  : msg_ptr_t;        { a pointer to the buffer where
                                      MBX placed a retrieved message }
    msg_retlen  : integer32;        { length of a retrieved message  }

  PROCEDURE check_status;                     { for error handling }
              .
              .
              .

  BEGIN

{ Open a channel to the mailbox. }

  mbx_$open ( mbx_name,
              mbx_namelen,
              NIL,
              0,
              mbx_handle,
              status );
  check_status;
```

Figure 5-10. A Client That Gets and Sends Messages

```
  { Read data and put it in the mailbox. }

  write( 'Enter a message; end with CTRL/Z: ' );
  WHILE NOT eof DO
    BEGIN
    readln( data_buf );
    mbx_$put_rec ( mbx_handle,
                   addr( data_buf ),
                   buf_len,
                   status );

{ Get a response from the server. }

    mbx_$get_rec( mbx_handle,
                  addr( msg_buf ),
                  msg_buf_len,
                  msg_retptr,
                  msg_retlen,
                  status );
    check_status;

    writeln ( msg_retptr^: msg_retlen );
    write( prompt_str );
    END;

done:

  mbx_$close( mbx_handle,
              status );
  check_status;

  END.
```

Figure 5-10. A Client That Gets and Sends Messages (continued)

5.7. Using Mailbox Eventcounts

An eventcount is a number that increases when a certain type of event happens. Use an eventcount to allow your program to wait for events, process them when they occur, and then wait for new events.

Whenever you create a mailbox, the system establishes two eventcounts for the mailbox. They are:

MBX_$GETREC_EC_KEY	Indicates that a new message may have arrived.
MBX_$PUTREC_EC_KEY	Indicates that a channel that was previously full may now be able to accept a message.

The mailbox manager uses these eventcounts when managing the mailbox. However, you can also use these eventcounts within your programs. Call MBX_$GET_EC to get a pointer to the MBX eventcount you want to use.

The mailbox eventcounts are system-defined eventcounts. Thus, these eventcounts, when satisfied, indicate that there *may* be an event to respond to. When a mailbox eventcount is

satisfied, your program should not assume that there *is* an event. See *Programming With General System Calls* for complete information on system-defined eventcounts.

After you get a pointer to an eventcount, use EC2_$WAIT or EC2_$WAIT_SVC to wait for the eventcount to increment. You pass EC2_$WAIT(_SVC) a "trigger" value; this is the value at which you want to be notified. EC2_$WAIT suspends your process until the eventcount reaches or exceeds your trigger value. When this condition is satisfied, EC2_$WAIT returns and allows your program to process the event.

If you need to wait for only one type of mailbox event, do not use an eventcount. Instead, use a nonconditional MBX call. By definition, this type of call waits for an event and then responds to the event. For example, a call such as MBX_$GET waits for a message and then gets it; a call such as MBX_$PUT waits until there is room in the channel, and then puts the message in.

However, if you want to wait for two or more types of events, use eventcounts. In this way, you can wait for any of the events to happen, respond to the type of event that occurs, and then go back to waiting for more events to happen. Your process is awakened whenever there is an event to respond to.

Follow these steps when you use eventcounts:

- Use the appropriate GET_EC calls to get pointers to each event that you want to wait for.
- Define the trigger values using the current value of each of your eventcounts. Use the call EC2_$READ to determine these values.
- Establish a loop to wait for events and process them. Use EC2_$WAIT or EC2_$WAIT_SVC to wait for events.
- Establish inner loops to process each of the events. Within each inner loop, define a new trigger value for your return to the EC2_$WAIT loop.
- Use conditional calls to process the event -- these calls don't wait for an event to happen before completing. To see if the call performed the requested action, check the return status.
- Keep using the conditional call until the return status indicates that there are no more events to process. Then return to the EC2_$WAIT(_SVC) loop.

The following sections describe how to use the MBX_$GET_REC_EC and MBX_$PUT_REC_EC eventcounts.

5.7.1. Waiting to Get a Message

Figure 5-11 shows a mailbox client that sets up two eventcounts:

- A stream get eventcount (STREAM_$GETREC_EC_KEY). This eventcount indicates when there may be input to get from a stream.
- A mailbox get eventcount (MBX_$GETREC_EC_KEY). This eventcount indicates when there may be input to get from a mailbox.

After establishing eventcounts, the program initializes the trigger values for responding to events. Then, the program uses EC2_$WAIT within a loop to wait for each eventcount to reach its trigger.

Note that before entering the EC2_$WAIT loop for the first time, the program sets the trigger values so that each eventcount is immediately satisfied. This insures that you will process any events that occurred before you entered the EC2_$WAIT loop.

After checking for each type of event, the program then waits for new events to occur. When a stream event occurs, the program gets input from the keyboard and puts it into the mailbox. When a mailbox event occurs, the program gets messages from the mailbox and displays them on the screen.

When getting stream and mailbox data, the program uses conditional get calls. Therefore, the program can keep getting stream input or mailbox messages until there is nothing more to get. The program then returns to the EC2_$WAIT loop to wait for another event.

Every time the program gets stream or mailbox input, the program redefines the trigger value for the EC2_$WAIT call. Therefore, when the program returns to the EC2_$WAIT loop, the trigger value is correctly defined.

```
PROGRAM mbx_get_ec ( input, output );

{ This program uses a mailbox eventcount to determine when to get messages
  from a mailbox. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

  LABEL
    done;

  CONST
    kbd_ec      = 1;                { index for keyboard event }
    mbx_ec      = 2;                { index for mbx event      }
    mbx_name    = 'test_mailbox';   { mailbox name             }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name }
    msg_buf_len = mbx_$msg_max;     { length of message buffer }

  TYPE
    msg_t       =
      ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =
      ^msg_t;     { pointer to a client message }
```

Figure 5-11. A Client That Uses MBX_$GETREC_EC_KEY

```
  VAR
    empty       : boolean;          { true if mailbox is empty             }
    ec2_ptr     : ARRAY [ 1..2 ] OF ec2_$ptr_t;
                                    { array of eventcount pointers         }
    ec2_val     : ARRAY [ 1..2 ] OF integer32;
                                    { array of trigger values              }
    which       : integer;          { integer that indicates an event type }
    data_buf    : msg_t;            { buffer where stream data can be read }
    data_retptr : msg_ptr_t;        { buffer where stream data is read     }
    linelen     : integer32;        { size of stream data line             }
    seek_key    : stream_$sk_t;
    mbx_handle  : univ_ptr;         { mailbox handle                       }
    status      : status_$t;        { status code                          }
    msg_buf     : msg_t;            { buffer that the client can use to
                                      get data                             }
    msg_retptr  : msg_ptr_t;        { buffer where MBX places a retrieved
                                      message                              }
    msg_retlen  : integer32;        { length of a retrieved message        }

  BEGIN

{ Get an eventcount that changes when there's input from the keyboard. }

  stream_$get_ec ( stream_$stdin,           { stream ID  }
                   stream_$getrec_ec_key,   { type of eventcount }
                   ec2_ptr[kbd_ec],         { where pointer is returned }
                   status );
  IF (status.all <> status_$ok) THEN
    RETURN;

{ Open a channel to the mailbox.  Then get an eventcount that changes when
  there are messages in the mailbox. }

  mbx_$open ( mbx_name,                     { name           }
              mbx_namelen,                  { length of name }
              NIL,                          { no data to send }
              0,                            { length of data }
              mbx_handle,                   { handle         }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,                  { mailbox }
               mbx_$getrec_ec_key,          { type of eventcount }
               ec2_ptr[mbx_ec],             { where pointer is returned }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  { Initialize the trigger values using the current eventcount values.}

  ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
  ec2_val[mbx_ec] := ec2_$read( ec2_ptr[mbx_ec]^ );
```

Figure 5-11. A Client That Uses MBX_$GETREC_EC_KEY (continued)

```
  { prompt for input }

  writeln( 'Enter a message; end with CTRL/Z: ' );


{ Now go into an infinite loop to wait for events.  When there is keyboard
  input, go to the kbd_ec loop. When there is a mailbox message, go to
  the mbx_ec loop. }

  REPEAT
    which := ec2_$wait( ec2_ptr,          { list of pointers to eventcounts }
                        ec2_val,          { trigger values                  }
                        2,                { no. eventcounts in the list     }
                        status );
    IF status.all <> status_$ok THEN
      RETURN;

    CASE which OF

    { For keyboard input, create a loop to read the current eventcount and
      increase it by one -- this is the new trigger value. Then get
      keyboard input. When there is no more input, exit from the loop.}

      kbd_ec:

        REPEAT
          ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ ) + 1;
          stream_$get_conditional( stream_$stdin,      { stream ID           }
                                   addr( data_buf ),   { buffer for data     }
                                   sizeof( data_buf ), { length of buffer    }
                                   data_retptr,        { where data is read  }
                                   linelen,            { length of data      }
                                   seek_key,           { seek key            }
                                  status );

          IF (status.all <> status_$ok)
             AND (status.code <> stream_$end_of_file)
          THEN
            RETURN;

          IF (status.subsys = stream_$subs)
             AND (status.code = stream_$end_of_file)
          THEN
            GOTO done;

          IF linelen > 0 THEN
            BEGIN
            mbx_$put_rec ( mbx_handle,                { handle         }
                           data_retptr,               { data to send   }
                           linelen,                   { length of data }
                           status );
            IF status.all <> status_$ok  THEN
              RETURN;
            END;
        UNTIL linelen = 0;
```

Figure 5-11. A Client That Uses MBX_$GETREC_EC_KEY (continued)

```
{ For a mailbox message, create a loop to read the current eventcount and
  increase it by one -- this is the new trigger value. Then get the
  mailbox message. When there are no more messages, exit from the loop. }

     mbx_ec:

       REPEAT
         empty := false;
         ec2_val[mbx_ec] := ec2_$read( ec2_ptr[mbx_ec]^ ) + 1;
         mbx_$get_conditional( mbx_handle,        { handle           }
                               addr( msg_buf ),   { buffer for data  }
                               msg_buf_len,       { length of buffer }
                               msg_retptr,        { actual data      }
                               msg_retlen,        { length of data   }
                               status );
         IF (status.all <> status_$ok)
            AND (status.all <> mbx_$channel_empty)
            AND (status.all <> mbx_$partial_record)
         THEN
           RETURN;

         IF (status.all = mbx_$channel_empty) THEN

           empty := true
         ELSE
           writeln ( msg_retptr^: msg_retlen );
       UNTIL empty;
     END; { case }
  UNTIL false;

done:

  mbx_$close( mbx_handle,
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  END.
```

Figure 5-11. A Client That Uses MBX_$GETREC_EC_KEY (continued)

5.7.2. Waiting to Put a Message

In order to wait for room in a full channel, you must first have an MBX_$PUT_REC_COND call that fails and returns an MBX_$NO_ROOM_IN_CHANNEL status code. When you receive such a status, you can use EC2_$WAIT to wait for the MBX_$PUTREC_EC_KEY eventcount to increase. When this count increases, there may be enough room in the channel to accept the message that you tried to put there.

Figure 5-12 shows a client that sets up three eventcounts:

- A stream get eventcount (STREAM_$GETREC_EC_KEY).
- A mailbox get eventcount (MBX_$GETREC_EC_KEY).
- A mailbox put eventcount (MBX_$PUTREC_EC_KEY).

The client uses the call EC2_$WAIT to wait for any of these events to occur. This program is similar to Figure 5-11 in Section 5.7.1. However, the following program (Figure 5-12) uses MBX_$PUT_REC_COND instead of MBX_$PUT_REC to put the keyboard data into the mailbox. If MBX_$PUT_REC_COND fails, then the program returns to the EC2_$WAIT loop. The program does not read new keyboard input until the MBX_$PUTREC_EC_KEY count is satisfied, and the program can put the data into the mailbox.

Note that when you initialize the trigger value for an MBX_$PUTREC_EC_KEY eventcount, you should add one to the current value before you use EC2_$WAIT to wait for events. This is in contrast to how you initialize the STREAM_$GETREC_EC_KEY and the MBX_$GETREC_EC_KEY counts. You should not force the MBX_$PUTREC_EC_KEY to be satisfied because this event occurs only after an MBX_PUT_REC_COND has failed.

```
PROGRAM mbx_put_ec ( input,output );

{ This mailbox client uses eventcounts to wait for input from the
  keyboard or from a mailbox.  The client also uses the putrec eventcount
  to wait for a previously full channel to accept a message.}

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

  LABEL
    done;

  CONST
    kbd_ec      = 1;                  { index for keyboard event }
    get_ec      = 2;                  { index for mbx get event  }
    put_ec      = 3;                  { index for mbx put event  }
    mbx_name    = 'test_mailbox';     { mailbox name             }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name   }
    msg_buf_len = mbx_$msg_max;       { length of message buffer }

  TYPE
    msg_t       =  ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =  ^msg_t;           { pointer to a client message     }

  VAR
    channel_full : boolean;          { TRUE if channel is full  }
    empty        : boolean;          { TRUE if mailbox is empty }
    ec2_ptr      : ARRAY [ 1..3 ] OF ec2_$ptr_t;
                                     { array of eventcount pointers }
    ec2_val      : ARRAY [ 1..3 ] OF integer32;
                                     { array of trigger values }
    which        : integer;          { integer that indicates an event type }
    data_buf     : msg_t;            { buffer where stream data can be read }
    data_retptr  : msg_ptr_t;        { buffer where stream data is read     }
    linelen      : integer32;        { size of stream data line             }
    seek_key     : stream_$sk_t;
```

Figure 5-12. A Client That Uses MBX_$PUTREC_EC_KEY

```
{ VAR cont. }
    mbx_handle    : univ_ptr;          { mailbox handle }
    status        : status_$t;         { status code    }
    msg_buf       : msg_t;             { buffer that the client can use to get
                                         data  }
    msg_retptr    : msg_ptr_t;         { buffer where MBX places a retrieved
                                         message }
    msg_retlen    : integer32;         { length of a retrieved message }

  BEGIN

{ Get an event count that shows when there's input from the keyboard. }

  stream_$get_ec ( stream_$stdin,          { stream to get eventcount for }
                   stream_$getrec_ec_key,  { type of eventcount   }
                   ec2_ptr[kbd_ec],        { where to put pointer }
                   status );
  IF (status.all <> status_$ok) THEN
    RETURN;


{ Open a channel to the mailbox.  Then get the getrec and putrec eventcounts. }

  mbx_$open ( mbx_name,           { name             }
              mbx_namelen,        { name length      }
              NIL,                { no data to send  }
              0,                  { length of data   }
              mbx_handle,         { handle           }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,         { mailbox to get eventcount for }
               mbx_$getrec_ec_key, { type of eventcount            }
               ec2_ptr[get_ec],    { where to put the pointer      }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,         { mailbox to get eventcount for }
               mbx_$putrec_ec_key, { type of eventcount        }
               ec2_ptr[put_ec],    { where to put the pointer  }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  { Initialize the trigger values for kbd_ec and get_ec using the current
    event count values.  Initialize the trigger value for put_ec as one
    greater than the current value. }

  ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
  ec2_val[get_ec] := ec2_$read( ec2_ptr[get_ec]^ );
  ec2_val[put_ec] := ec2_$read( ec2_ptr[put_ec]^ ) + 1;

  { Initialize variable. }
  channel_full := false;
```

Figure 5-12. A Client That Uses MBX_$PUTREC_EC_KEY (continued)

```
  { Prompt for input. }

  writeln( 'Enter a message; end with CTRL/Z: ' );

{ Now go into an infinite loop to wait for events.  When there is keyboard
  input, go to the kbd_ec loop. When there is a mailbox message to get,
  go to the get_ec loop. When a previously full channel has room to accept
  a mailbox message, go to the put_ec loop. }

  REPEAT
    which := ec2_$wait( ec2_ptr,        { list of eventcount pointers }
                        ec2_val,        { list of eventcount values   }
                        3,              { number of pointers in list  }
                        status );
    IF status.all <> status_$ok THEN
      RETURN;

    CASE which OF

    { For keyboard input, create a loop to read the current eventcount
      and increase it by one -- this is the new satisfaction value
      for the ec2_$wait call in the outer loop. Then get keyboard input.
      When there is no more input, exit from the loop. }

      kbd_ec:

        REPEAT
          ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ ) + 1;

          { If the channel is full, return to EC2_$WAIT.  When there is room
            in the channel, the put_ec section will change channel_full
            to FALSE. }

          IF channel_full = true THEN
            EXIT;

          stream_$get_conditional( stream_$stdin,       { stream ID           }
                                   addr( data_buf ),    { buffer for data     }
                                   sizeof( data_buf ),  { size of buffer      }
                                   data_retptr,         { where data is copied}
                                   linelen,             { length of data      }
                                   seek_key,
                                   status );
          IF (status.all <> status_$ok)
             AND (status.code <> stream_$end_of_file)
          THEN
            RETURN;

          IF (status.subsys = stream_$subs)
             AND (status.code = stream_$end_of_file)
          THEN
            GOTO done;
```

Figure 5-12. A Client That Uses MBX_$PUTREC_EC_KEY (continued)

```
        IF linelen > 0 THEN

        { If mbx_$put_rec_cond fails because the channel is full, set
          channel_full to TRUE. }

          BEGIN
          mbx_$put_rec_cond ( mbx_handle,        { handle       }
                              data_retptr,       { data to send }
                              linelen,           { length of data }
                              status );
          IF (status.all <> status_$ok)
             AND (status.all <> mbx_$no_room_in_channel)
          THEN
            RETURN;
          IF status.all = mbx_$no_room_in_channel THEN
            channel_full := true;
          END;
      UNTIL linelen = 0;

{ For a mailbox message, create a loop to read the current eventcount and
  increase it by one -- this is the new satisfaction value. Then get the
  mailbox message. When there are no more messages, exit from the loop. }

    get_ec:

      REPEAT
        empty := false;
        ec2_val[get_ec] := ec2_$read( ec2_ptr[get_ec]^ ) + 1;
        mbx_$get_conditional( mbx_handle,          { handle              }
                              addr( msg_buf ),     { buffer for data     }
                              msg_buf_len,         { length of buffer    }
                              msg_retptr,          { where data is copied }
                              msg_retlen,          { length of data      }
                              status );
        IF (status.all <> status_$ok) AND
           (status.all <> mbx_$channel_empty) AND
           (status.all <> mbx_$partial_record) THEN
           RETURN;

        IF (status.all = mbx_$channel_empty) THEN
          empty := true;

        IF NOT empty THEN
          writeln ( msg_retptr^: msg_retlen );
      UNTIL empty;
```

Figure 5-12. A Client That Uses MBX_$PUTREC_EC_KEY (continued)

```
      { When the putrec eventcount is satisfied, try to put the message
        in the mailbox.  If there still is no room, go back to EC2_$WAIT.
        If the MBX_$PUT_REC_COND succeeded, set the keyboard eventcount
        so that you will check to see if there are any more keyboard messages. }

      put_ec:

        BEGIN
        ec2_val[put_ec] := ec2_$read( ec2_ptr[put_ec]^ ) + 1;
        mbx_$put_rec_cond( mbx_handle,     { handle           }
                           data_retptr,    { data to send     }
                           linelen,        { length of data   }
                           status );
        IF status.all <> status_$ok THEN
          IF status.all = mbx_$no_room_in_channel THEN
            EXIT
          ELSE RETURN;
        channel_full := false;

        { Set the keyboard satisfaction value so that the ec2_$wait call will
          indicate a keyboard event when you return to the ec2_$wait loop. }

        ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
        END;

      END; { case }

  UNTIL false;

done:

  mbx_$close( mbx_handle,        { mailbox to close channel in }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  END.
```

Figure 5-12. A Client That Uses MBX_$PUTREC_EC_KEY (continued)

5.8. Using The Mailbox Helper

Whenever a mailbox server and client are on different nodes, the programs can not directly access the mailbox at the same time. Therefore, you must run the program /SYS/MBX/MBX_HELPER (the mailbox helper) to help remote servers and clients send messages. The MBX_HELPER maintains a system mailbox (SYSMBX) in the 'node_data directory.

When a server sends a message to a remote client, the message first goes to the client node's MBX_HELPER. The MBX_HELPER then puts the message into the SYSMBX on the client's node. When the client uses the mailbox handle to get the message, the system gets the message from the SYSMBX, rather than from the server's mailbox.

When a client sends a message to a remote server, the message follows a slightly different path. First, the message goes to the MBX_HELPER on the server's node. The MBX_HELPER then

puts the message into the server's mailbox. When the server gets the message, it uses its own mailbox as it normally would.

Figure 5-13 shows how the MBX_HELPER assists a client and a server. In this figure, the server is on Node B, the client is on Node C, and the server's mailbox is on a disk on Node A. The server's and client's nodes are each running the MBX_HELPER. On Node B, the MBX_HELPER does not need to use the SYSMBX to assist the server. However, on Node C, the MBX_HELPER uses its SYSMBX to assist the client.

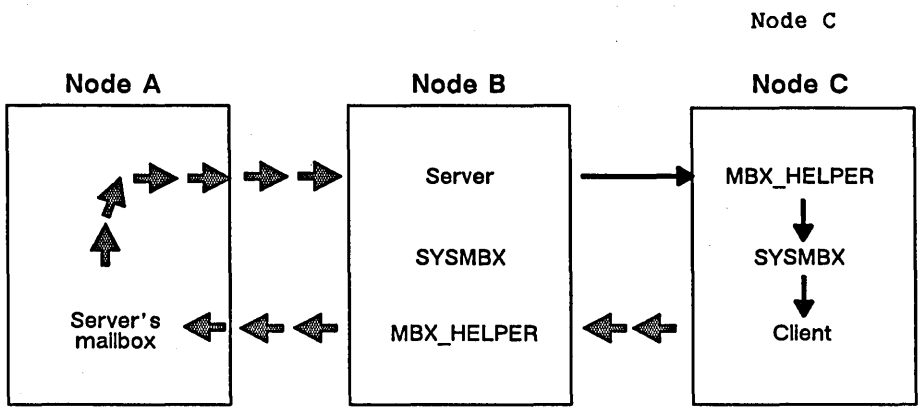

Figure 5-13. How MBX_HELPER Assists Interprocess Communication

Note that the MBX_HELPER's role is transparent to server and client programs. That is, these programs always use the same mailbox handle and calls to send and receive messages, regardless of how the messages are routed. The only times you need to consider the MBX_HELPER are:

- When a server sends long messages to a client (see Section 5.8.2).
- When you use unusually restrictive access control lists (ACLs) (see Section 5.8.3).

5.8.1. Starting the MBX_HELPER

If you plan to execute a mailbox server and its clients on different nodes, use the Shell command PST to see whether the MBX_HELPER is running on each of these nodes. PST shows the processes that are currently running on a node. Usually, if the MBX_HELPER is running, it will be displayed under the name MBX_HELPER. However, the MBX_HELPER is listed under a different name if the person who started the MBX_HELPER specified another name.

If you need to start the MBX_HELPER, use one of the following DM commands:

CPO	Creates a process without pads or windows. The process terminates when you log out.
CPS	Creates a process without pads or windows. The process runs whether or not anyone is logged in.

The following command creates a process that runs the MBX_HELPER. Because no name is specified, the process will be named MBX_HELPER.

Command: CPS /SYS/MBX/MBX_HELPER

You may want to keep the MBX_HELPER running as a background process on your node because many DOMAIN products rely on it. To ensure that the MBX_HELPER is always running on your node, you can include the CPO or CPS command in your node's start-up command file. The mailbox helper only needs to run once on any node, no matter how many clients or servers there are on the node.

When you start MBX_HELPER, you can use the following options:

-MAXCHAN n Set the number you specify with **n** as the maximum number of remote channels to and from the node using MBX. By default, **n** is 128 (channels.)

-DATASIZE b Sets the number you specify with **b** as the buffer size for the SYSMBX channel. (This buffer size is also called a queue data size.) The buffer size is the maximum number of bytes that the SYSMBX can buffer at one time for a remote server. By default, **b** is 1158.

5.8.2. Adjusting the Buffer Size for Long Messages

You may need to increase the SYSMBX default buffer size if a local client is expecting long messages from a remote server. For example, if the SYSMBX has a buffer of 1158 bytes (the default), and a client is expecting a message of 9000 bytes, you must change the buffer size.

To change the buffer size for the system mailbox, use the Shell command SIGP to stop the MBX_HELPER. Then restart the MBX_HELPER with a new buffer size. For example:

Command: CPS /SYS/MBX/MBX_HELPER -DATASIZE 9000

NOTE: You should increase the buffer size only if you have a special reason for allowing clients to receive large messages. If you arbitrarily increase the buffer size, you may force the MBX_HELPER to use up too much of its virtual address space for maintaining the SYSMBX. In such a case, you will limit the number of servers that the MBX_HELPER can assist.

5.8.3. Using MBX_HELPER in a Secure Network

In a secure network, a mailbox receives its access control list (ACL) from the directory in which it is created. Be sure that this ACL allows the MBX_HELPER (on the server's node) to access the mailbox. Otherwise, clients on other nodes (that use the server's MBX_HELPER) will not be able to use the mailbox. If a server's MBX_HELPER does not have access to the server's mailbox, a client will receive the error MBX_$HELPER_NO_RIGHTS when it tries to open a channel to a remote server's mailbox.

An MBX_HELPER can access a mailbox only if the mailbox's ACL allows read and write access to the MBX_HELPER. To have access, the MBX_HELPER's subject identifier (SID) must be included in the mailbox's ACL. An MBX_HELPER usually has a SID of:

user.server.none.node-id

The node-id represents the node where the MBX_HELPER is running. However, if you used the Shell command CPO to start the MBX_HELPER, the process will have your SID. Use the following Shell command to find out the SID of your MBX_HELPER:

```
$ LUSR -ME -ALLP
```

To be sure that a server node's MBX_HELPER can access the server's mailbox, use the EDACL Shell command to add the MBX_HELPER's SID to the mailbox's access control list. Give the MBX_HELPER write and read access.

The following example shows how to add an entry to a mailbox's access control list. The example gives write and read access to all MBX_HELPERs with SIDs beginning with user.server.none.

```
$ EDACL -A USER.SERVER.NONE.% WR TEST_MAILBOX
$ ACL -L TEST_MAILBOX                    # verify that the SID is there
Acl for test_mailbox:
  oconnell.%.group1.%              pgndwrx
  user.server.none.%               ----wr-
  %.%.group1.%                     pgn-wrx
  %.%.%.%                          -----rx
```

Note that you cannot change a mailbox's ACL while the mailbox is in use.

You must also be sure that a client has read and write access to its local SYSMBX, as the client uses this mailbox to get messages that are routed through the local MBX_HELPER. If a client can't access its SYSMBX, the client will get the error MBX_$CLIENT_NO_RIGHTS when it tries to open a channel to a remote server's mailbox.

Chapter 6
Sending Datagrams

This chapter describes how to send interprocess communication (IPC) datagrams using IPC system calls. The chapter includes:

- An overview of using datagrams for interprocess communication.
- A summary of the IPC system calls and data types.
- An explanation of the IPC datagram format.
- Examples of two programs that use IPC datagrams to communicate.

6.1. Overview

An IPC datagram is a message that one program sends to another through a communications buffer called a socket. To receive datagrams, a program must open a socket, receive a handle for the socket, and store the handle in a file. Another program can then look in the file, obtain the socket handle, and send a datagram. Programs that use datagrams can be running anywhere in the DOMAIN system network; the programs do not have to be on the same node.

Figure 6-1 shows two programs that communicate using sockets. Each program knows the other's socket handle. (These handles are stored in files, which are not represented in the diagram.) Program 1 sends datagrams to Program 2's socket; Program 2 sends datagrams to Program 1's socket. Note that a program always receives datagrams through a socket on its local node. However, the socket handle can be stored in a file anywhere in the network.

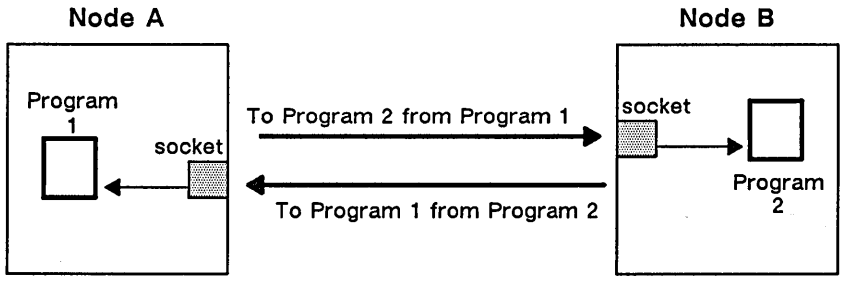

Figure 6-1. Using Sockets to Receive Datagrams

A datagram connection is a fast, but unreliable, connection between two programs. That is, when you send a datagram, the system makes its best effort to deliver it. However, the programs that communicate using datagrams are responsible for verifying that each datagram is successfully received. Usually, programs verify the datagram transmission by using acknowledgments. That is, when one program sends a datagram to another, the sender waits for an acknowledgment. If the acknowledgment does not arrive within a specified amount of time, the sender retransmits the original datagram.

In addition, programs that use datagrams must be able to recognize duplicates in order to remain synchronized. For example, it is possible for a program to receive a datagram, and send an acknowledgment to the sender. However, the acknowledgment may get lost. In such a case, the sender would retransmit the original datagram. The receiver should be able to identify the second datagram as a duplicate.

In conclusion, IPC datagrams provide two important features:

- They provide a fast way to transfer information between programs on the same, or on different, nodes.
- They allow programs to use the DOMAIN file system to catalog socket handles. A program that wants to communicate with another can easily obtain the socket handle by looking in the correct file.

6.2. IPC System Calls, Insert Files, and Data Types

To send and receive datagrams, use the IPC system calls. These calls invoke the IPC manager, the system component that is responsible for datagrams. Table 6-1 summarizes the IPC calls.

Table 6-1. Summary of IPC Calls

Operation	Calls
Opening sockets	IPC_$CREATE IPC_$OPEN
Sending and receiving datagrams	IPC_$SEND IPC_$RCV IPC_$SAR
Waiting for datagrams	IPC_$WAIT IPC_$GET_EC
Closing sockets	IPC_$CLOSE IPC_$DELETE
Determining a socket's handle	IPC_$RESOLVE

In order to use the IPC calls, you must include the appropriate insert files in your program. These insert files are:

/SYS/INS/IPC.INS.C	(for C)
/SYS/INS/IPC.INS.FTN	(for FORTRAN)
/SYS/INS/IPC.INS.PAS	(for Pascal)

Some of the IPC calls allow you to specify parameters using special DOMAIN data types. These include:

IPC_$DATA_T	The data portion of an IPC datagram.
IPC_$HDR_INFO_T	The header portion of an IPC datagram.
IPC_$SOCKET_HANDLE_T	A handle for an IPC socket.

For complete information on IPC system calls and data types, see the *DOMAIN System Call Reference.*

6.3. IPC Datagram Format

An IPC datagram has two parts:

- A header that can contain up to 128 bytes.
- A data section that can contain up to 1024 bytes.

If you are sending large amounts of data, you can place control information in the header and put the data in the data section. The following example shows variables for a program that sends a page (1024 bytes) of data, with control information in the datagram's header. The example uses the predefined types IPC_$HDR_INFO_T and IPC_$DATA_T to define the length of each buffer. IPC_$HDR_INFO_T defines a 128-byte header buffer; IPC_$DATA_T defines a 1024-byte data buffer.

```
VAR
  header_buf : IPC_$HDR_INFO_T; { header buffer for control information }
  data_buf   : IPC_$DATA_T;     { data buffer }
```

Note that a program does not have to use both the header and data sections of a datagram. For example, if you are sending less than 128 bytes of information, you can put all the information in the header. The following example shows a Pascal type definition for a datagram that contains 128 bytes. The first two bytes contain an integer that is used as control information; the following 126 bytes contain data. Note that even though the example does not need a data section, you should still declare a buffer for it. (The IPC send and receive calls require that you specify buffers for both the header and data portions of a datagram.) To declare a data buffer as a placeholder, use a small, 2-byte buffer. For example:

```
TYPE                                        { user-defined format for datagram }
  datagram_t =                              { that fits into a 128-byte header }
    RECORD
    msg_number : integer;                   { 2 bytes for control info }
    msg_text   : ARRAY [ 1..126 ] OF char; { 126 bytes for data       }
    END;

VAR
  header_buf            : datagram_t;     { header contains entire datagram }
  data_buf              : integer;        { placeholder for data section    }
```

If you are sending datagrams that contain less than 128 bytes, it is more efficient to put all the information into the header. It takes less time to send a datagram that contains only a header.

6.4. Using The IPC Calls

When you use datagrams, your application usually involves two programs:

- An IPC server, whose primary purpose is to receive information.
- An IPC client, whose primary purpose is to send information.

An IPC server waits for datagrams from a client. When a datagram arrives, the server processes the datagram and sends a reply if necessary. In contrast, an IPC client initiates communication by sending information to the server. The primary flow of information from a client to a server can be either one- or two-way. For example, a client can send data for the server to process. Alternatively, the client can send a request for information that the server must respond to.

When a client and server use IPC datagrams to communicate, the programs must provide mechanisms for message verification. Because IPC is an unreliable datagram service, a client does not know whether any of its datagrams are received. Therefore, servers should acknowledge the receipt of datagrams.

In addition, servers and clients must ensure that they remain synchronized. For example, a server needs to recognize duplicate messages from a client. Also, a client needs to recognize duplicate acknowledgments from a server.

IPC servers and clients use the same calls, but in different sequences. Sections 6.4.1 through 6.4.5 give an overview of the tasks you perform with IPC calls. Section 6.5 describes how to use the calls in a server; Section 6.6 describes how to use the calls in a client. In addition, Sections 6.5 and 6.6 show some techniques for datagram verification and interprocess synchronization.

6.4.1. Creating a Handle File and Opening a Socket

In order to receive datagrams, a program must open one of the node's sockets and obtain the socket's handle. Before opening a socket, however, you must create a file in which to store the handle. To create such a file, use IPC_$CREATE. For example:

```
ipc_$create ( 'prog1_handle_file',     { file for handle    }
              17,                      { length of filename }
              status );
```

After you create the handle file, you can open a socket with IPC_$OPEN. This call opens an available socket and stores the handle in the handle file. In addition, IPC_$OPEN returns the handle in one of the output parameters.

When you use IPC_$OPEN, you must specify the name and length of the handle file. You must also include a socket depth to specify how many datagrams the socket can hold at one time. Allowable values are one through four.

You must also declare a variable that IPC_$OPEN uses to return the handle for the socket it opens. Declare this variable to be of type IPC_$SOCKET_HANDLE_T. In FORTRAN, specify this as an array of 20 characters. C users should note that the IPC_$SOCKET_HANDLE_T string is *not* null-terminated. Therefore, in an operation such as comparing two handles, use **strncmp** not **strcmp**. Figure 6-2 illustrates IPC_$OPEN.

```
VAR
  status                : status_$t;            { status code }
  local_socket_handle   : ipc_$socket_handle_t; { handle for your socket }

BEGIN

{ Create file to contain a socket handle. }
                .
                .
                .
{ Open a socket so you can receive datagrams. }

ipc_$open( 'prog1_handle_file',    { file for handle        }
           17,                     { length of filename     }
           4,                      { socket depth           }
           local_socket_handle,    { handle for your socket }
           status );
```

Figure 6-2. Opening a Socket

6.4.2. Receiving Datagrams

Once you have opened a socket, you can receive datagrams. To get a datagram that has arrived in your socket, use IPC_$RCV. This call copies the header portion of the datagram to one buffer, and the data portion to another. Note that IPC_$RCV also obtains the handle you can use to reply to the datagram.

When you use IPC_$RCV, you must specify the lengths of the buffers where you want to receive the datagram header and data. IPC_$RCV gets the amount of information that will fit in these buffers. If the actual header or data is longer, you will lose part of the information.

IPC_$RCV has the following format:

```
IPC_$RCV ( handle, hdr-buflen, data-buflen, from-handle, hdr-buf, hdr-length,
           data-buf, data-length, status )
```

These are the input parameters:

- **handle** -- The handle from which to get the datagram. You should specify the handle for your socket. (In C, the handle is not null-terminated.)
- **hdr-buflen** -- The length of the buffer where IPC_$RCV should copy the header portion of the datagram.
- **data-buflen** -- The length of the buffer where you are copying the data portion of the datagram.

These are the output parameters:

- **from-handle** -- The handle to which you can send a reply. This is the handle identifies a socket belonging to the program that sent the datagram.

- **hdr-buf** -- The buffer where IPC_$RCV copies the data.
- **hdr-length** -- The length of the header that is copied.
- **data-buf** -- The buffer where IPC_$RCV copies the data.
- **data-length** -- The length of the data that is copied.
- **status** -- The return status.

Note that you can define datagrams that do not use the data portion of a datagram, and instead store user data in the datagram header. (See Section 6.3.) In such a case, you can specify a small data buffer with IPC_$RCV, even though your program does not use this buffer.

NOTE: There is an alternate method of specifying the data buffer when no data is included in this buffer. Instead of declaring a small buffer to act as a placeholder, you can specify the **data-buf** as 0. However, if you do this, be sure that you have also specifed the **data-buflen** as 0.

Figure 6-3 uses IPC_$RCV to get a datagram. The datagram contains two fields of information: a message number and some text. The datagram is only 128 bytes long, and is stored entirely in the header. Therefore, the IPC_$RCV call specifies that the length of the header buffer is 128 and the length of the data buffer is 0.

```
TYPE                                        { user-defined format for datagram }
  datagram_t =                              { that fits into 128-byte header }
    RECORD
    msg_number : integer;                   { 2 bytes for sequence no. }
    msg_text   : ARRAY [ 1..126 ] OF char; { 126 bytes for data       }
    END;

VAR
  status                : status_$t;            { status code                  }
  local_socket_handle   : ipc_$socket_handle_t;{ handle for your socket       }
  rcv_socket_handle     : ipc_$socket_handle_t;{ handle for received datagram }
  expected_msg_number   : integer;              { datagram you expect          }
  receive_buf           : datagram_t;           { buffer to receive datagram   }
  receive_len           : integer;              { length of received datagram  }
  data_buf              : integer;              { buffer for data portion      }
  data_len              : integer;              { length of data portion       }

BEGIN

{ Create a file for the handle and open a socket. }
                .
                .
                .
```

Figure 6-3. Receiving a Datagram from a Socket

```
{ Get a datagram. }

 ipc_$rcv( local_socket_handle,        { your socket handle          }
           sizeof( ipc_$hdr_info_t ),  { maximum size of header      }
           0,                          { maximum size of data        }
           rcv_socket_handle,          { where the datagram came from }
           receive_buf,                { buffer for header           }
           receive_len,                { length of header            }
           data_buf,                   { buffer for data             }
           data_len,                   { length of data              }
           status );
```

Figure 6-3. Receiving a Datagram from a Socket (continued)

6.4.3. Waiting for Datagrams

In general, you use IPC_$RCV when you know that there is a datagram to get. If you call IPC_$RCV when your socket is empty, the call returns immediately with the status IPC_$SOCKET_EMPTY. IPC provides two ways to wait for datagrams:

- Use the call IPC_$WAIT.
- Use the call IPC_$GET_EC to get a pointer to an eventcount associated with a socket. Then use either EC2_$WAIT or EC2_$WAIT_SVC to wait for datagrams.

In general, use IPC_$WAIT if you are waiting for datagrams in only one socket. With IPC_$WAIT, specify the handle for the socket you are waiting on, and the amount of time to wait. If a datagram is already in the socket, IPC_$WAIT returns immediately with a success status. Otherwise, IPC_$WAIT waits for the specified amount of time. If the call times out before a datagram arrives, IPC_$WAIT returns the status IPC_$TIMEOUT.

With IPC_$WAIT, specify the waiting time in quarter seconds. Thus, if you want the call to wait for one minute, you can specify the waiting time as **60 * 4**. The following example illustrates IPC_$WAIT.

```
VAR
  status                : status_$t;           { status code              }
  local_socket_handle   : ipc_$socket_handle_t; { handle for your socket }

BEGIN
                .
                .
                .

    ipc_$wait( local_socket_handle,   { your socket handle }
               300 * 4,               { wait 5 minutes     }
               status );

    { If the wait is successful, call IPC_$RCV to get the datagram. }
```

To wait for datagrams in more than one socket, use IPC_$GET_EC to get a pointer to each socket's eventcount. Then use EC2_$READ to read the values for each eventcount, and use EC2_$WAIT (or EC2_$WAIT_SVC) to wait for any of the eventcounts to increment.

The socket eventcount works in the same way as the other system-defined eventcounts:

- Use IPC_$GET_EC to get a pointer to each socket's eventcount.
- Use EC2_$READ to read the current value of each eventcount. Place these values into an array of eventcount trigger values.
- Design a loop to wait for any of the eventcounts to reach their trigger values. When an eventcount is satisfied, try to get a datagram from the corresponding socket. Use EC2_$WAIT(_SVC) for eventcounts to reach their triggers.
- The first time through the loop, define the trigger values so that each eventcount wait is immediately satisfied. This prevents you from waiting if a datagram is currently available in a socket.

Figure 6-4 shows how to use IPC_$GET_EC to get pointers to eventcounts for two sockets. See *Programming With General System Calls* for more information on using system-defined eventcounts.

```
{ Program that shows how to wait for eventcounts from to sockets. }

PROGRAM wait_socket;

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ipc.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

    CONST
       { define indexes for arrays  }
        sock1_ec = 1;  { The first element is for socket 1.  }
        sock2_ec = 2;  { The second element is for socket 2. }

    VAR
        ec2_ptr : ARRAY [ 1..2 ] OF ec2_$ptr_t;{ array of pointers to
                                                 two eventcounts              }
        ec2_val : ARRAY [ 1..2 ] OF integer32; { array of eventcount
                                                 trigger values               }
        which          : integer;             { number returned by EC2_$WAIT }
        socket1_handle : ipc_$socket_handle_t; { handle for socket 1          }
        socket2_handle : ipc_$socket_handle_t; { handle for socket 2          }
        status         : status_$t;            { status code                  }

    BEGIN

    { Create socket handle files and open each socket. }
                          .
                          .
                          .
```

Figure 6-4. Waiting for Datagrams from Two Sockets

```
  { Get pointer to eventcount for socket 1. Store the eventcount pointer
    in first element of pointer array. }

  ipc_$get_ec( socket1_handle,    { socket to get eventcount ptr for    }
               ec2_ptr[sock1_ec], { eventcount pointer returned by call }
               status );
  IF status.all <> status_$ok THEN
      RETURN;

  { Get pointer to eventcount for socket 2. Store the eventcount pointer
    in second element of pointer array. }

  ipc_$get_ec( socket2_handle,    { socket to get eventcount ptr for    }
               ec2_ptr[sock2_ec], { eventcount pointer returned by call }
               status );
  IF status.all <> status_$ok THEN
      RETURN;

{ Read the value of socket 1's eventcount and store it in the first
  element of the trigger value array.  Read the value of socket 2's
  eventcount and store it in the second element of the trigger value array. }

  ec2_val[sock1_ec] := ec2_$read( ec2_ptr[sock1_ec]^ );
  ec2_val[sock2_ec] := ec2_$read( ec2_ptr[sock2_ec]^ );

  { Go into an infinite loop to wait for  input from the two sources.
    The first time through, both eventcounts are satisfied. }

  REPEAT
      which := ec2_$wait( ec2_ptr,          { list of pointers   }
                          ec2_val,          { list of triggers   }
                          2,                { no. of eventcounts }
                          status );
      IF status.all <> status_$ok THEN
          RETURN;

      CASE which OF
          sock1_ec:

              { If WHICH is 1,  get datagrams from socket 1
                and then return to EC2_$WAIT. }
                      .
                      .
                      .

          sock2_ec:

              { If WHICH is 2, get datagrams from socket 2
                and then return to EC2_$WAIT. }
                      .
                      .
                      .


          END; {case }

  UNTIL false;
  END. {program }
```

Figure 6-4. Waiting for Datagrams from Two Sockets (continued)

6.4.4. Sending Datagrams

To send a datagram, you must know the socket handle for the program you want to communicate with. If you are responding to a datagram you just received, use the socket handle returned by IPC_$RCV. However, if you are initiating the communication, you can obtain the handle if you know the pathname for the socket handle file. Use IPC_$RESOLVE to obtain a handle from a handle file. For example:

```
VAR
  status                : status_$t;            { status code            }
  prog2_socket_handle   : ipc_$socket_handle_t; { handle for your socket }

BEGIN
                .
                .
                .

ipc_$resolve( 'prog2_socket_file', { file with prog2's handle }
              17,                  { length of filename       }
              prog2_socket_handle, { prog2's handle           }
              status );
```

IPC_$RESOLVE returns a handle only if the socket is currently open. If you call IPC_$RESOLVE and the file you specify does not contain a handle for an open socket, IPC_$RESOLVE returns the error IPC_$SOCKET_NOT_OPEN.

To send a datagram, use IPC_$SEND or IPC_$SAR. IPC_$SEND sends a datagram, while IPC_$SAR sends a datagram, waits a specified amount of time for a response, and gets the response if one arrives. This section describes IPC_$SEND; for information on IPC_$SAR, see the *DOMAIN System Call Reference*.

IPC_$SEND has the following format:

```
IPC_$SEND ( to-handle, reply-handle, hdr-buf, hdr-length, data-buf,
            data-length, status )
```

The following are input parameters:

- **to-handle** -- The handle for the socket where you are sending the datagram.
- **reply-handle** -- The handle for the socket where you want to get a reply. The program that gets your datagram uses this socket to send a reply.
- **hdr-buf** -- The buffer that contains the header for the datagram you are sending.
- **hdr-length** -- The length of the header.
- **data-buf** -- The buffer that contains the data portion of the datagram you are sending.
- **data-length** -- The length of the data.

IPC_$SEND returns a status code to indicate whether the call completed successfully. Note that even if IPC_$SEND completes successfully, there is no guarantee that the datagram has reached

the target socket, or that the target program has received the datagram. For example, IPC_$SEND will complete successfully even if the target socket is closed. Therefore, programs that use IPC calls must acknowledge the receipt of datagrams, and must synchronize their activities. Sections 6.5 and 6.6 describe some techniques to provide this synchronization.

Figure 6-5 uses IPC_$SEND to send a datagram. The datagram contains two fields of information: a message number and some text. The datagram is only 128 bytes long, and is stored entirely in the header. Therefore, the IPC_$SEND call uses a small buffer as a placeholder, and specifies the data length as 0.

```
CONST

  socket_file         = 'prog1_handle_file'; { file for own socket handle }

TYPE                                        { user-defined format for datagram }
  datagram_t =                              { that fits into 128-byte header   }
    RECORD
    msg_number : integer;                   { 2 bytes for sequence no.         }
    msg_text : ARRAY [ 1..126 ] OF char;    { 126 bytes for data               }
    END;

VAR
  status                : status_$t;
  local_socket_handle   : ipc_$socket_handle_t;  { handle for your socket    }
  prog2_socket_handle   : ipc_$socket_handle_t;  { handle for prog2's socket }
  send_buf              : datagram_t;  { buffer with datagram to send        }
  send_len              : integer;     { length of datagram you are sending  }
  data_buf              : integer;     { buffer for data portion of datagram }

BEGIN

{ Create file for socket handle and open socket
  Then get handle for socket you want to send to.}
                .
                .
                .

    ipc_$send( prog2_socket_handle,     { where to send datagram }
               local_socket_handle,     { your handle            }
               send_buf,                { header portion         }
               send_len,                { length of header       }
               data_buf,                { data portion           }
               0,                       { length of the data     }
               status );
```

Figure 6-5. Sending a Datagram

6.4.5. Closing Sockets and Deleting Handle Files

When a program is through with a socket, the program should close it. To close a socket, use IPC_$CLOSE. This call removes the socket handle from the handle file and unlocks the handle file. You should close sockets when you no longer need them, as the number of sockets on your node is limited. When you close a socket, you make it available for other programs that need to use it.

A program should always close any open sockets before exiting. However, if a program fails to do so, the IPC manager will automatically close sockets and unlock handle files when a program exits.

In addition to closing a socket, you can also delete the handle file. To delete a handle file, use IPC_$DELETE. A program does not have to delete its handle file before exiting. For example, if you execute a program frequently, you may save the handle file rather than creating and deleting it each time you run the program.

Both IPC_$CLOSE and IPC_$DELETE accept the name and length of the pathname for the handle file. In addition, both calls return a status code. Figure 6-6 illustrates IPC_$CLOSE and IPC_$DELETE:

```
{ Declarations for program }
              .
              .
              .

{ Use IPC for interprocess communication. }
              .
              .
              .

{ Close socket and delete handle file when you are through. }

ipc_$close( 'prog1_handle_file',     { file containing handle }
            17,                      { length of filename     }
            status );

IF status.all <> status_$ok THEN
  error_$print_name (status, 'Error closing socket.', 21);

ipc_$delete( 'prog1_handle_file',     { file containing handle }
             17,                      { length of filename     }
             status );

IF status.all <> status_$ok THEN
  error_$print_name (status, 'Error deleting handle file.', 27);
```

Figure 6-6. Closing a Socket and Deleting a Handle File

6.5. Writing a Server

The main function of an IPC server is to service client requests. Therefore, an IPC server waits for incoming datagrams, gets them, and processes them. However, because a datagram connection is unreliable, an IPC server needs to acknowledge the receipt of each datagram. In addition, the IPC server needs to remain synchronized with the client. For example, if the client retransmits a datagram (because the server's acknowledgment got lost), the server must recognize that the second datagram is a duplicate.

One way to provide synchronization is to have the client include a sequence number with every datagram. In this way, a server can check to see that a datagram contains the expected sequence number. If the client increments the sequence number for each new datagram, the server can then know which datagram to expect. For example, after processing datagram number 5, the server expects number 6. If the server receives the same sequence number twice, the server can identify a duplicate.

The sequence number is also useful when the server sends acknowledgments to the client. For example, the server can include the sequence number in an acknowledgment to identify which datagram is being acknowledged.

The following algorithm describes one way a server can receive datagrams, acknowledge them, and remain synchronized with the client:

1. Use IPC_$WAIT to wait for datagrams. When a datagram arrives, use IPC_$RCV to get it.

2. Examine the datagram and determine whether you should process it. First, check the length. If the length is not within the limits you are expecting, ignore the datagram and wait for another.

3. If the length is acceptable, check the sequence number. If the sequence number is not what you're expecting, ignore the datagram and wait for another.

4. If the datagram contains the correct sequence number, process the datagram and increment your sequence count. Then use IPC_$SEND to send an acknowledgment to the client. Include the sequence number for the datagram you are acknowledging.

5. If you receive a datagram that contains a sequence number that is one less than what you're expecting, don't process it. (You processed it when you got it the first time.) However, send an acknowledgment because your original acknowledgment may have gotten lost.

6. Keep waiting for and processing datagrams until the data transmission is complete.

Figure 6-7 shows a server that waits for and processes IPC datagrams, following the algorithm described above. Note that the server expects datagrams that contain the 128-byte header, but do not contain a data section. Within the header, the first two bytes contain the sequence number. The following 126 bytes contain a message sent by the client.

First, the server creates a file for its socket handle and opens a socket. Then the server waits for datagrams. When it gets a datagram, the server determines whether the datagram is valid. The server first checks to see if the datagram's length is between 1 and 128. Next, the server checks that the sequence number matches the one that the server is expecting. If the sequence number is correct, the server displays the client's message on the screen and then sends an acknowledgment.

If the sequence number is one less than what the server is expecting, the datagram contains a second copy of a previously received datagram. The server does not display the message a second time. However, the server sends another acknowledgment to the client. The server continues waiting for datagrams, displaying them, and sending acknowledgments until the message "q" arrives.

```
PROGRAM ipc_server;

{ This program accepts datagrams from a client and displays
  the message portion of the datagram.  The program also
  illustrates techniques for acknowledging datagrams and
  remaining synchronized with the client. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ipc.ins.pas';


  CONST
    socket_file =  'server_handle_file'; { file for socket handle    }
    wait_sec    = 300;                   { number of seconds to wait }


  TYPE                                    { user-defined format for datagram }
    datagram_t =                          { that fits into 128-byte header   }
      RECORD
      msg_number : integer;               { 2 bytes for sequence no. }
      msg_text   : ARRAY [ 1..126 ] OF char; { 126 bytes for data      }
      END;

  VAR
    status                : status_$t;           { status code                 }
    local_socket_handle   : ipc_$socket_handle_t;{ handle for your socket      }
    rcv_socket_handle     : ipc_$socket_handle_t;{ handle for received datagram }
    expected_msg_number   : integer;             { datagram you expect         }
    receive_buf           : datagram_t;          { buffer to receive datagram  }
    receive_len           : integer;             { length of received datagram }
    text_len              : integer;             { length of message text      }
    ack_buf               : integer;             { acknowledgment              }
    data_buf              : integer;             { buffer for data portion     }
    data_len              : integer;             { length of data portion      }

  PROCEDURE program_cleanup;          { Cleanup procedure }
    BEGIN

      ipc_$close( socket_file,              { file containing handle }
                  sizeof( socket_file ),    { length of filename     }
                  status );
      IF status.all <> status_$ok THEN
        error_$print_name (status, 'Error closing socket.', 21);

      ipc_$delete( socket_file,             { file containing handle }
                   sizeof( socket_file ),   { length of filename     }
                   status );
      IF status.all <> status_$ok THEN
        error_$print_name (status, 'Error deleting handle file.', 27);
    END;
```

Figure 6-7. An IPC Server

```
PROCEDURE program_exit;    { Program exit procedure }

  BEGIN

    program_cleanup;
    pgm_$exit;

  END;


BEGIN          { ipc_server }

{ Create file to contain a socket handle. }

ipc_$create ( socket_file,            { file for handle    }
              sizeof( socket_file ),  { length of filename }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error creating handle file.', 27);
    pgm_$exit;
  END;

{ Open a socket so you can receive datagrams. }

ipc_$open( socket_file,            { file for handle        }
           sizeof( socket_file ),  { length of filename     }
           4,                      { socket depth           }
           local_socket_handle,    { handle for your socket }
           status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error opening socket.', 21);
    program_exit;
  END;

writeln ( 'Socket opened successfully.');

{ Initialize datagram sequence number. }

expected_msg_number := 1;
```

Figure 6-7. An IPC Server (continued)

```
{ Enter loop to wait for and acknowledge incoming datagrams.
  Acknowledge a datagram if it is valid and it contains a
  sequence number that you are expecting. }

REPEAT                                  { begin receive loop }
    { Wait up to 5 minutes for a datagram. }

  REPEAT                                {  begin wait loop }

    ipc_$wait( local_socket_handle,     { your socket handle }
               wait_sec * 4,            { wait 5 minutes     }
               status );

    { If a datagram arrived, exit from the wait loop and get the datagram. }

    IF status.all = status_$ok THEN
      EXIT;

    { It the call timed out, wait again. }

    IF status.all = ipc_$timeout THEN
      NEXT;

    { If another error occurred, print error and repeat wait loop. }
    error_$print( status );

  UNTIL false;                          { end wait loop }

  { Get the datagram. }

  ipc_$rcv( local_socket_handle,          { your socket handle            }
            sizeof( ipc_$hdr_info_t ),    { maximum size of header        }
            0,                            { maximum size of data          }
            rcv_socket_handle,            { where the datagram came from  }
            receive_buf,                  { buffer for header             }
            receive_len,                  { length of header              }
            data_buf,                     { buffer for data               }
            data_len,                     { length of data                }
            status );

  { If there's an error, print an error message and return to the top
    of the receive loop to wait for a new datagram. }

  IF status.all <> status_$ok THEN
    BEGIN
    error_$print_name (status, 'Error getting the datagram.', 27);
    NEXT;
    END;

  { If you successfully got a datagram, make sure that it's valid before
    you use its contents.   First, check that the size is valid. If the
    size is not valid, print an error message and return to the top
    of the receive loop to wait for a new datagram. }
```

Figure 6-7. An IPC Server (continued)

```
IF receive_len > sizeof( datagram_t ) OR ELSE
   receive_len < sizeof( datagram_t.msg_number )
THEN
  BEGIN
  error_$print_name (status, 'Bad datagram length.', 20);
  NEXT;
  END;

{ Check the sequence number to make sure you remain synchronized
  with the sender.  If the sequence number is bad, return to the
  top of the receive loop to wait for a new datagram.  Discard any
  datagram whose sequence number is larger than you expect.  Also,
  discard any datagram whose sequence number is two less than the
  number you expect because this is an old datagram that you have
  already processed. }

IF receive_buf.msg_number > expected_msg_number OR ELSE
   receive_buf.msg_number < (expected_msg_number - 1)
THEN
  BEGIN
  error_$print_name (status, 'Received out of sequence.', 26);
  NEXT;
  END;

{ If you got the datagram you expected, process it by
  displaying it on the screen. Then increment the sequence number.
  If you got a datagram whose sequence number is one less than
  what you expected, assume that the sender never got your
  acknowledgment.  Don't process the datagram, but send another
  acknowledgment. }

IF receive_buf.msg_number = expected_msg_number THEN
  BEGIN
    text_len := receive_len - sizeof( datagram_t.msg_number );
    writeln ( 'Received message: ', receive_buf.msg_text : text_len );
    expected_msg_number := expected_msg_number + 1;
  END;

{ Send an acknowledgment for the datagram you just processed,
  or for a datagram whose acknowledgment was lost.
  Send the sequence number for the datagram you are acknowledging. }

ack_buf := receive_buf.msg_number;
ipc_$send( rcv_socket_handle,       { where to send acknowledgment }
           local_socket_handle,     { your handle                  }
           ack_buf,                 { the number you're sending    }
           sizeof( ack_buf ),       { length of the number         }
           data_buf,                { data portion                 }
           0,                       { length of data portion       }
           status );

IF status.all <> status_$ok THEN
  error_$print_name (status, 'Error sending acknowledgment.', 29);
```

Figure 6-7. An IPC Server (continued)

```
  { When you receive a 'q' then quit running. }

  IF ( text_len = 1 ) AND
     ( receive_buf.msg_text[1] = 'q' ) THEN
     program_exit;

UNTIL false;          { end receive loop }

END.
```

Figure 6-7. An IPC Server (continued)

6.6. Writing a Client

The main function of an IPC client is to send requests to a server. Therefore, the client must:

- Obtain the information to send.
- Send a datagram containing this information.
- Wait for the server's acknowledgment.
- If the acknowledgment does not arrive within a specified amount of time, resend the datagram.

In addition, the client must perform special checks to remain synchronized with the server. For example, if an acknowlegment is late and you resend a datagram, you might eventually receive two acknowledgments -- one for each datagram that you sent. Your program should be able to identify duplicate acknowledgments.

Another condition you need to watch for is if the server (or the network) becomes unavailable. In such a case, you will receive no acknowledgments and should probably report an error condition.

As described in Section 6.5, you can use sequence numbers to provide synchronization. Include a sequence number with every datagram that you send; check the sequence number from every acknowledgment that you receive.

The following algorithm describes one way a client can send datagrams and wait for acknowledgments:

1. Use IPC_$RESOLVE to obtain the handle for the server you want to communicate with. Note that you must get the handle after the server has opened its socket. Therefore, always start the client after you start the server.
2. Obtain the information you want to include in the datagram.
3. Use IPC_$SEND to send the datagram. Include the sequence number within the datagram.
4. Wait for an acknowledgment using IPC_$WAIT. If a reply arrives, use IPC$_RCV to get it. Then verify that the acknowledgment is for the datagram you just sent. If it is, increment the sequence number and send the next datagram. If the acknowledgment is bad (e.g., it contains the wrong sequence number or if it came

from the wrong socket) wait for another datagram. Wait for, and receive, up to three datagrams. If you still don't get a valid acknowledgment, resend the datagram.

5. If you are waiting for an acknowledgment and it does not arrive within the allotted time, resend the the original datagram. Try up to five times to send the datagram and get the correct acknowledgment. If you don't get an acknowledgment after five tries, then display an error message and exit.

6. Continue sending datagrams and waiting for acknowledgments until the data transfer is complete.

Figure 6-8 shows a client that sends IPC datagrams using the algorithm described above. Note that the client sends datagrams containing the 128-byte header, but they do not contain a data section. Within the header, the first two bytes contain the sequence number. The following 126 bytes contain a message.

First, the client creates a socket handle file and opens a socket. Next, the client uses IPC_$RESOLVE to obtain the server's socket handle. Then the client prompts the user to enter a message.

The client places a sequence number in front of the message and then uses IPC_$SEND to send the datagram to the server. The IPC_$SEND call includes the client's socket handle; the server will use this handle to send an acknowledgment.

Then the client waits for an acknowledgment. If the acknowledgment arrives within the specified amount of time, the client verifies that it is valid. To be valid, the acknowledgment must be two bytes long and must contain the current sequence number.

If the correct acknowledgment arrives, the client prompts for a new message to send. However, if the correct acknowledgment does not arrive, the client resends the current message. The client continues prompting for and sending messages until the user enters "q" (or until the client is unable to communicate with the server.)

```
PROGRAM ipc_client;

{ This program prompts for input and sends it to an IPC server,
  using a datagram.  The program waits for an acknowledgment
  before prompting for new input. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/ipc.ins.pas';


  CONST

    socket_file         = 'client_handle_file'; { file for own socket handle    }
    server_socket_file = 'server_handle_file';  { file with server's handle     }
    wait_sec = 10;                              { no. seconds to wait           }
```

Figure 6-8. An IPC Client

```
TYPE                                    { user-defined format for datagram   }
  datagram_t =                          { that fits into the 128-byte header }
    RECORD
    msg_number : integer;               { 2 bytes for sequence no.           }
    msg_text : ARRAY [ 1..126 ] OF char; { 126 bytes for data                }
    END;

VAR
  status                : status_$t;
  local_socket_handle   : ipc_$socket_handle_t;{ handle for your socket       }
  server_socket_handle  : ipc_$socket_handle_t;{ handle for server's socket   }
  rcv_socket_handle     : ipc_$socket_handle_t;{ handle for received datagram }
  current_msg_number    : integer;      { datagram you are sending           }
  send_buf              : datagram_t;   { buffer with datagram to send       }
  send_len              : integer;      { length of datagram you are sending }
  text_len              : integer;      { length of message text             }
  receive_buf           : datagram_t;   { buffer in which to receive datagram }
  receive_len           : integer;      { length of received datagram        }
  data_buf              : integer;      { buffer for data portion of datagram }
  data_len              : integer;      { length of data portion of datagram }
  count                 : integer;
  send_count            : integer;
  wait_count            : integer;

LABEL
  message_received;

PROCEDURE program_cleanup;          { Clean-up procedure }
  BEGIN

  ipc_$close( socket_file,            { file containing handle }
              sizeof( socket_file ),  { length of filename     }
              status );
  IF status.all <> status_$ok THEN
    error_$print_name (status, 'Error closing socket.', 21);

    ipc_$delete( socket_file,           { file containing handle }
                 sizeof( socket_file ), { length of filename     }
                 status );
    IF status.all <> status_$ok THEN
      error_$print_name (status, 'Error deleting handle file.', 27);
  END;

PROCEDURE program_exit;       { Program exit procedure }

  BEGIN
    program_cleanup;
    pgm_$exit;
  END;
```

Figure 6-8. An IPC Client (continued)

```
BEGIN       { ipc_client }

{ Create file to contain a socket handle. }

ipc_$create ( socket_file,            { file for handle    }
              sizeof( socket_file ),  { length of filename }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error creating handle file.', 27);
    pgm_$exit;
  END;

{ Open a socket so you can receive datagrams. }

ipc_$open( socket_file,             { file for handle        }
           sizeof( socket_file ),   { length of filename     }
           4,                       { socket depth           }
           local_socket_handle,     { handle for your socket }
           status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error opening socket.', 21);
    program_exit;
  END;

{ Get the server's socket handle. }

ipc_$resolve( server_socket_file,           { file with server's handle }
              sizeof( server_socket_file ), { length of filename        }
              server_socket_handle,         { server's handle           }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error resolving socket name.', 28);
    program_exit;
  END;

{ Initialize datagram sequence number. }

current_msg_number := 1;

{ Enter loop to prompt for messages, send datagrams, and
  wait for acknowledgments. }
```

Figure 6-8. An IPC Client (continued)

```
REPEAT      { begin get message loop }

  { Prompt for message. }

  writeln ( 'Enter a message ( q to quit) : ');
  readln ( send_buf.msg_text );

  { Determine the message length. }

  text_len := sizeof ( send_buf.msg_text );
  WHILE ( send_buf.msg_text[text_len] = ' ' ) AND
        ( text_len > 0 ) DO
    text_len := text_len - 1;

  { Define the sequence number and length for the datagram. }

  send_buf.msg_number := current_msg_number;
  send_len := text_len + sizeof( datagram_t.msg_number );

  { Send the datagram and wait for an acknowledgment. If you don't
    receive the acknowledgment within 10 seconds, then resend the
    datagram. Try to send up to 5 times. }

  FOR send_count := 1 TO 5 DO           { begin send loop        }
    BEGIN
    ipc_$send( server_socket_handle,    { where to send datagram }
               local_socket_handle,     { your handle            }
               send_buf,                { header you're sending  }
               send_len,                { length of the  header  }
               data_buf,                { data portion           }
               0,                       { length of the data     }
               status );

    { If there's an error, try sending again. }

    IF status.all <> status_$ok THEN
      BEGIN
        error_$print_name (status, 'Error sending datagram.', 23);
        NEXT;
      END;

    { If the send completed successfully, wait for an acknowledgment.
      If you get a bad acknowledgment, ignore it and wait for another one.
      Repeat the wait loop up to 3 times, then resend the datagram. }

    FOR wait_count := 1 TO 3 DO   { begin wait loop }

      BEGIN

      ipc_$wait( local_socket_handle,  { your handle     }
                 wait_sec * 4,         { wait 10 seconds }
                 status );
```

Figure 6-8. An IPC Client (continued)

```
IF status.all <> status_$ok THEN
  BEGIN

  { If the wait timed out, exit from the wait loop and resend the
    datagram. If there was another type of error, display an
    error message and repeat the wait loop. }

  IF status.all = ipc_$timeout THEN
    EXIT;

  error_$print_name (status, 'Error waiting for datagram.', 27);
  NEXT;
  END;

{ If the wait completed successfully, get the datagram and
  verify that it contains a valid acknowledgment. }

ipc_$rcv( local_socket_handle,        { your socket              }
          sizeof ( ipc_$hdr_info_t ), { maximum size of header   }
          0 ,                         { maximum size of data     }
          rcv_socket_handle,          { where datagram came from }
          receive_buf,                { buffer for header        }
          receive_len,                { length of header         }
          data_buf,                   { buffer for data          }
          data_len,                   { length of data           }
          status );

{ If there's a receive error, repeat the wait loop. }

IF status.all <> status_$ok THEN
  BEGIN
  error_$print_name (status, 'Error receiving datagram.', 25);
  NEXT;
  END;

{ If you got a datagram, make sure it came from the server.
  Otherwise, display an error message and repeat the wait loop. }

IF server_socket_handle <> rcv_socket_handle THEN
  BEGIN
  error_$print_name (status,
                     'Received message from unexpected socket.', 40);
  NEXT;
  END;

{ If the datagram came from the right socket, make sure it's
  the right length.  If it is, then check that the sequence number
  is correct.If either condition is FALSE, then display an error
  message and repeat the wait loop. }
```

Figure 6-8. An IPC Client (continued)

```
      IF receive_len <> sizeof( datagram_t.msg_number ) OR ELSE
         receive_buf.msg_number <> current_msg_number

      THEN
        BEGIN
          error_$print_name (status, 'Received bad acknowledgment.', 28);
          NEXT;
        END;

      { If you got a good acknowledgment, exit from the wait loop
        and the send loop. }

      GOTO message_received;

      END;              { end of wait loop }

    END;                     { end of send message loop }

{ If you failed after 5 send attempts, display an error message and exit. }

  writeln ('Unable to communicate with foreign socket.');
  program_exit;


message_received:

  { If the message started with a 'q', exit from the program.
    Otherwise, increment the sequence number and repeat the get
    message loop. }

  IF ( text_len = 1 ) AND ( send_buf.msg_text[1] = 'q') THEN
    program_exit;

  current_msg_number := current_msg_number + 1;

UNTIL false;          { end get message loop }

END.
```

Figure 6-8. An IPC Client (continued)

Appendix A
Sample Pascal Programs

Appendix A contains complete versions of the Pascal examples that are used throughout this manual. The MS programs are described in Chapter 2; the EC2 programs are described in Chapter 3; the MUTEX programs are described in Chapter 4; the MBX programs are described in Chapter 5; the IPC programs are described in Chapter 6.

Table A-1 summarizes the programs that appear in this appendix.

Table A-1. Summary of Programs in Appendix A

Name	Function
MS_MAP.PAS	Maps, accesses, and unmaps a file (page A-3).
MS_ADVICE.PAS	Provides file usage advice after you map a file (page A-5).
MS_ATTRIBUTES.PAS	Obtains attributes of a mapped file (page A-7).
MS_RELOCK.PAS	Waits for other programs to unmap a file and then relocks it (page A-9).
MS_REMAP.PAS	Maps different sections of a file (page A-11).
MS_TRUNCATE.PAS	Deletes the contents of a file, writing new information, and setting the file length (page A-13).
EC2_PRODUCER.PAS	Uses user-defined eventcounts to synchronize data transfer to another program (page A-15).
EC2_CONSUMER.PAS	Uses user-defined eventcounts to synchronize data transfer from another program (page A-19).
MUTEX_INIT.PAS	Initializes a mutex lock (page A-22).
MUTEX_USER.PAS	Uses a mutex lock (page A-24).
MBX_SERVER.PAS	Uses a mailbox to wait for and service requests from other programs (page A-27).
MBX_CLIENT.PAS	Uses a mailbox to send requests to another program and receive replies (page A-31).
MBX_GET_EC.PAS	Waits to receive a mailbox message (page A-33).

Table A-1. Summary of Programs in Appendix A (continued)

Name	Function
MBX_PUT_EC.PAS	Waits to put a message into a mailbox (page A-37).
IPC_SERVER.PAS	Uses datagrams to obtain data from another program (page A-42).
IPC_CLIENT.PAS	Uses datagrams to send data to another program (page A-47).

A.1. MS_MAP.PAS

```
PROGRAM ms_map;

{ This program tries to map a file.  If the file does
  not exist, the program creates and maps it. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;          { status             }
    mapped_seg_ptr  : ^sales_record_t;    { pointer to record  }
    len_mapped      : integer32;          { length mapped      }


  BEGIN

  { Try to map existing file. }

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                          { where to start }
                              sizeof(sales_record_t),     { desired length }
                              ms_$nr_xor_1w,              { concurrency    }
                              ms_$wr,                     { access         }
                              true,                       { extension      }
                              len_mapped,
                              status);

  IF(status.all <> status_$ok)   AND (status.all <> name_$not_found) THEN

    BEGIN
      error_$print (status);
      RETURN;
    END;
```

```
  IF status.all = name_$not_found THEN

    BEGIN    { create and map  }
      mapped_seg_ptr := ms_$crmapl (pathname,
                                    namelength,
                                    0,                          { where to start  }
                                    sizeof(sales_record_t),     { desired length  }
                                    ms_$nr_xor_1w,              { concurrency     }
                                    status);

      IF status.all <> status_$ok THEN
        BEGIN
          error_$print (status);
          RETURN;
        END;
    END;     {create and map }

  { Write a record to the file. }

  mapped_seg_ptr^.item_code := 1;
  mapped_seg_ptr^.units_sold := 10;


  { Unmap file and exit. }

  ms_$unmap( mapped_seg_ptr,
             sizeof(sales_record_t),
             status );

  IF status.all <> status_$ok THEN
    BEGIN
      error_$print (status);
      RETURN;
    END;

END.
```

A.2. MS_ADVICE.PAS

Before you execute this program, use MS_MAP to create the file DATA_FILE.

```
PROGRAM ms_advice;

{ This program maps the file DATA_FILE and then provides file usage advice.
  Before running this program, run MS_MAP to create the file DATA_FILE. }



%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map   }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;        { status           }
    mapped_seg_ptr  : ^sales_record_t;  { pointer to record }
    len_mapped      : integer32;        { length mapped    }


  BEGIN

  { Map existing file. }

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                                { where to start   }
                              sizeof (sales_record_t) * 100,    { map 100 records  }
                              ms_$nr_xor_1w,                    { concurrency      }
                              ms_$wr,                           { access           }
                              true,                             { extension        }
                              len_mapped,
                              status);

  IF(status.all <> status_$ok) THEN

    BEGIN
      error_$print (status);
      RETURN;
    END;
```

```
{ Provide advice to say you will access the file sequentially. }

 ms_$advice ( mapped_seg_ptr,            { first mapped byte }
              len_mapped,                { length            }
              ms_$sequential,            { access type       }
              [],                        { reserved          }
              sizeof ( sales_record_t ), { size of record    }
              status );

 { Unmap file and exit. }

 ms_$unmap( mapped_seg_ptr,              { first mapped byte }
            len_mapped,                  { length            }
            status );

 IF status.all <> status_$ok THEN
   BEGIN
     error_$print (status);
     RETURN;
   END;

END.
```

A.3. MS_ATTRIBUTES.PAS

Before you execute this program, use MS_MAP to create the file DATA_FILE in the directory where you will execute MS_ATTRIBUTES.

```
PROGRAM ms_attributes;

{ This program obtains the attributes of a mapped file.
  Before running this program, run MS_MAP to create the file DATA_FILE. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);
     attrib_max = sizeof (ms_$attrib_t); { attribute buffer length }

  VAR
    status          : status_$t;        { status                      }
    mapped_seg_ptr  : ^sales_record_t;  { pointer to record           }
    len_mapped      : integer32;        { length mapped               }
    attrib_buf      : ms_$attrib_t;     { attribute buffer            }
    attrib_len      : integer;          { length of attribute record  }


  BEGIN

  { Map existing file. }

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                          { where to start }
                              sizeof(sales_record_t),     { desired length }
                              ms_$nr_xor_1w,              { concurrency    }
                              ms_$r,                      { access         }
                              true,                       { extension      }
                              len_mapped,
                              status);

  IF(status.all <> status_$ok) THEN

    BEGIN
      error_$print (status);
      RETURN;
    END;
```

```
{ Determine current file length. }

 MS_$ATTRIBUTES ( mapped_seg_ptr,{ start byte                    }
                  attrib_buf,    { attributes buffer             }
                  attrib_len,    { length of returned attributes }
                  attrib_max,    { length of attributes buffer   }
                  status );

 WRITELN ( 'The file is ', attrib_buf.cur_len,' bytes long.');
 WRITELN ( 'The file uses ', attrib_buf.blocks_used, ' blocks.');

 { Unmap file and exit. }

 ms_$unmap( mapped_seg_ptr,
            len_mapped,
            status );

 IF status.all <> status_$ok THEN
   BEGIN
     error_$print (status);
     RETURN;
   END;

END.
```

A.4. MS_RELOCK.PAS

Before you execute this program, use MS_MAP create the file DATA_FILE in the directory where you will execute MS_RELOCK.

```
PROGRAM ms_relock;

{ This program maps a file with a protected read lock.
  Then it tries to change the lock to an exclusive write lock.
  If the MS_$RELOCK call is not successful, the program
  waits five seconds and then tries again to relock
  the file.

  Before running this program, run MS_MAP to create the file DATA_FILE. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/time.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';

  LABEL
    unmap;

  TYPE
    sales_record_t =
      RECORD
      first        : integer;
      second       : integer;
      END;

  CONST
    pathname = 'data_file';
    namelength = sizeof( pathname );

  VAR
    status         : status_$t;
    mapped_seg_ptr : ^sales_record_t;
    len_mapped     : integer32;
    wait_time      : time_$clock_t;

  BEGIN

  { Map file with protected RIW lock. }

  mapped_seg_ptr := ms_$mapl ( pathname,
                               namelength,
                               0,                          { where to start }
                               sizeof( sales_record_t ),   { desired length }
                               ms_$nr_xor_1w,              { concurrency    }
                               ms_$riw,                    { access         }
                               true,
                               len_mapped,
                               status );

  IF status.all <> status_$ok THEN
    RETURN;
  writeln( 'File was mapped' );
```

```
  { Get system clock value for 5 seconds. }

  cal_$sec_to_clock ( 5, wait_time );

  REPEAT

    { Keep trying to relock file until the MS_$RELOCK
      call is successful.  After each try, wait 5
      seconds and then try again. }

    ms_$relock( mapped_seg_ptr,
                ms_$wr,
                status );
    IF status.all <> ms_$in_use THEN
      EXIT;
    time_$wait( time_$relative, wait_time, status );
  UNTIL false;

  IF status.all <> status_$ok THEN
    GOTO unmap;

  { Write new data into the file. }

  mapped_seg_ptr^.first  := 1;
  mapped_seg_ptr^.second := 3000;

  { Unmap the file. }

unmap:

  ms_$unmap( mapped_seg_ptr,            { pointer to file }
             sizeof( sales_record_t ),  { size of file    }
             status );

  IF status.all <> status_$ok THEN
    RETURN;
  writeln( 'File was unmapped' );

  END.
```

A.5. MS_REMAP.PAS

Before you execute this program, create the file DATA_FILE in the directory where you will execute MS_REMAP.

```
PROGRAM ms_remap;

{ This program uses MS_$REMAP to map each segment of a
  large data file.  The file contains user-defined records.
  Before running this program, run MS_MAP to create the file DATA_FILE. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  TYPE
    sales_record_t =      { A sales record is 4 bytes long }
      RECORD
      item_code  : integer;
      units_sold : integer;
      END;
    data_page_t  = ARRAY [0..255] OF sales_record_t; { 256 records in a file }
    mapped_seg_t = ARRAY [0..31] OF data_page_t;     { 32 pages in a segment }

  CONST

    pathname    = 'data_file';
    namelength  = sizeof( pathname );

  VAR

    mapped_seg_ptr : ^mapped_seg_t;    { address returned by MS calls }
    status         : status_$t;        { status code                  }
    num_pages      : integer32;        { number of pages to map       }
    first_seg      : integer32;        { first segment to map         }
    last_seg       : integer32;        { last segment to map          }
    start_byte     : integer32;        { first byte                   }
    len_mapped     : integer32;        { length mapped                }
    seg_num        : integer32;        { counter                      }

  PROCEDURE check_status;  { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;

  BEGIN

   WRITE('Enter file size in pages: ');
   READLN(num_pages);
```

```
{ Determine the range of segments you need to map. }

first_seg := 0;                          { first segment }
last_seg := ( num_pages - 1 ) div 32;    { last segment  }

{ Initialize variable to indicate first byte to map. Map the first
  segment, starting at byte 0 in the file. }

start_byte := 0;
mapped_seg_ptr := ms_$mapl( pathname,          { file to map              }
                            namelength,        { length of file name      }
                            start_byte,        { start at first byte      }
                            32 * 1024,         { map 1 segment            }
                            ms_$nr_xor_1w,     { concurrency              }
                            ms_$wr,            { access                   }
                            true,              { extend                   }
                            len_mapped,        { bytes mapped - returned  }
                            status);

check_status;

{ Print message about the current segment.  Then remap the file
  to get the next segment.  Keep looping until you finish. }

FOR seg_num := first_seg to last_seg DO
    BEGIN           { map loop }

      writeln('Finished mapping segment ',seg_num);
      writeln('Segment contains data starting at byte ', start_byte);
      writeln('The first record at this address has this item code: ');
      writeln( mapped_seg_ptr^[0,0].item_code);

      { Remap the file to get the next segment unless you are done. }

      IF seg_num < last_seg THEN

        BEGIN     { remap }

          start_byte := ( seg_num + 1 ) * 32 * 1024;
          mapped_seg_ptr :=  ms_$remap( mapped_seg_ptr, { previous segment }
                                        start_byte,     {  new segment     }
                                        32 * 1024,      { map 1 segment    }
                                        len_mapped,     { length mapped    }
                                        status);
          check_status;

        END;      { remap }

      END;        { map loop }


{ Unmap the file and exit. }

ms_$unmap(mapped_seg_ptr,
          len_mapped,
          status);
check_status;

END.
```

A.6. MS_TRUNCATE.PAS

Before you execute this program, create the file DATA_FILE in the directory where you will execute MS_TRUNCATE.

```
PROGRAM ms_truncate;

{ This program maps the file DATA_FILE and uses MS_$TRUNCATE to delete
  the contents of the file.  After writing a new record to
  the file, the program uses MS_$TRUNCATE to set the file
  length to the number of used bytes.

  Before running this program, run MS_MAP to create the file DATA_FILE. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/name.ins.pas';

  TYPE
    sales_record_t =           { user-defined record }
        RECORD
        item_code     : integer;
        units_sold    : integer;
        END;

  CONST
     pathname = 'data_file';              { file to map }
     namelength = sizeof(pathname);

  VAR
    status          : status_$t;          { status            }
    mapped_seg_ptr  : ^sales_record_t;    { pointer to record }
    len_mapped      : integer32;          { length mapped     }


  BEGIN

  { Map existing file.}

  mapped_seg_ptr := ms_$mapl (pathname,
                              namelength,
                              0,                          { where to start  }
                              sizeof(sales_record_t),     { desired length  }
                              ms_$nr_xor_1w,              { concurrency     }
                              ms_$wr,                     { access          }
                              true,                       { extension       }
                              len_mapped,
                              status);

  IF(status.all <> status_$ok)   THEN

    BEGIN
      error_$print (status);
      RETURN;
    END;
```

```
  { Truncate the file to 0 to delete existing contents. }

  ms_$truncate ( mapped_seg_ptr,        { where to start    }
                 0,                     { no. bytes to keep }
                 status );


  { Write a new record to the file. }

  mapped_seg_ptr^.item_code := 1;
  mapped_seg_ptr^.units_sold := 10;


  { Truncate the length to 4 - the number of used bytes. }

  MS_$TRUNCATE ( mapped_seg_ptr,          { where to start    }
                 sizeof (sales_record_t), { no. bytes to keep }
                 status );

  { Unmap file and exit. }

  ms_$unmap( mapped_seg_ptr,
             len_mapped,
             status );

  IF status.all <> status_$ok THEN
    BEGIN
      error_$print (status);
      RETURN;
    END;

END.
```

A.7. EC2_PRODUCER.PAS

After initializing user-defined eventcounts, this program invokes EC2_CONSUMER.PAS (see Section A.8).

Before you execute this program, create the file INPUT_FILE in the directory where you will execute EC2_PRODUCER. Also, be sure that EC2_CONSUMER is in this directory.

```
PROGRAM ec2_producer;

{ This program uses user-defined eventcounts to synchronize data transfer
  to another program, EC2_CONSUMER.  Be sure that EC2_CONSUMER is in the
  directory where you will execute EC2_PRODUCER.  Also, before you execute
  this program, use the DM editor to create a file named INPUT_FILE. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

    TYPE
        shared_info_t =
            RECORD                                  { fields in mapped record   }
            producer_ec     : ec2_$eventcount_t;    { producer eventcount       }
            consumer_ec     : ec2_$eventcount_t;    { consumer eventcount       }
            done            : boolean;              { set to TRUE when you
                                                      reach end of input file   }
            record_present  : boolean;              { set to TRUE when you copy
                                                      record from input file    }
            useful_data     : string;               { record from input file    }
            data_len        : integer32;            { length of useful_data     }
            END;

    VAR
        status              : status_$t;            { status code               }
        length_mapped       : integer32;            { length of mapped data     }
        stream_id           : stream_$id_t;         { stream ID                 }
        seek_key            : stream_$sk_t;         { where stream data starts  }
        shared_info         : ^shared_info_t;       { address of first byte of
                                                      mapped storage            }
        res                 : uid_$t;               { reserved parameter for
                                                      pgm_$invoke               }
        consumer_wait_value : integer32;            { trigger value when waiting
                                                      for consumer eventcount   }
        local_buf           : string;               { buffer where stream_$get_ rec
                                                      may copy data }
        data_ptr            : ^string;              { buffer where stream_$get_rec
                                                      copies data }
        i                   : integer;              { value returned by ec2_$wait}
```

```
PROCEDURE check_status;                          { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
        BEGIN
        error_$print( status );
        pgm_$exit;
        END;
    END;

BEGIN

{ Open a stream to get records from. }

stream_$open( 'input_file',          { object to be opened }
              10,                    { length of pathname  }
              stream_$read,          { access type         }
              stream_$no_conc_write, { concurrency type    }
              stream_id,             { stream ID returned  }
              status );
check_status;

{ Map a shared data file containing the eventcounts.
  Use this file to hold a record of type SHARED_INFO_T. }

shared_info := ms_$mapl( 'shared_file',       { object to be mapped }
                         11,                  { length of name      }
                         0,                   { first byte to map   }
                         sizeof( shared_info_t ),
                                              { no. bytes to map    }
                         ms_$cowriters,       { locking mode        }
                         ms_$wr,              { access type         }
                         true,                { map length in 3rd
                                                parameter, even if
                                                object is shorter    }
                         length_mapped,       { bytes mapped - returned }
                         status );
check_status;

WITH shared_info^ DO
    BEGIN

    { Initialize the eventcounts }

    ec2_$init( producer_ec );
    ec2_$init( consumer_ec );

    { Store the current value of the consumer eventcount
      so you can use it as a trigger value later. }

    consumer_wait_value := ec2_$read( consumer_ec );

    { Set the condition field to FALSE. }

    record_present := false;
```

```
{ Invoke the consumer program in a separate process. }

pgm_$invoke( 'ec2_consumer.bin', { program to invoke           }
             16,                 { name length                 }
             0,                  { no. args. to pass           }
             0,                  { addresses of args.          }
             0,                  { no. of streams to pass      }
             0,                  { stream IDs to pass          }
             [],                 { invoke in a separate process }
             res,                { reserved                    }
             status );
check_status;

REPEAT { loop over input file }

{ Get a record.  If the input record is longer than your buffer,
  the get call returns a negative value.  To correct this, define
  the record length to be the buffer length. }

    stream_$get_rec( stream_id,          { stream ID to get record from}
                     addr( local_buf ), { where data may be read }
                     sizeof( local_buf ), { buffer size }
                     data_ptr,           { where data is read - returned }
                     data_len,           { length of data }
                     seek_key,
                     status );
    IF data_len < 0 THEN
        data_len := sizeof( local_buf );

    { Set DONE to TRUE or FALSE, depending on result
      of stream_$get_rec.  Store DONE in shared memory. }

    done := status.all <> status_$ok;

    { Copy record to shared buffer. }

    useful_data := data_ptr^;

    { Set condition field to TRUE and advance producer
      eventcount. }

    record_present := true;
    ec2_$advance( producer_ec, status );
    check_status;

    { Wait for consumer to get the record. }

    consumer_wait_value := consumer_wait_value +1;
    i := ec2_$wait( addr( consumer_ec ),  { eventcount      }
                    consumer_wait_value,  { trigger value   }
                    1,                    { no. eventcounts }
                    status );
    check_status;

UNTIL done;

END;
```

```
{ Clean up and terminate. }

ms_$unmap( shared_info,   { first byte of mapped object }
           length_mapped, { length of mapped object     }
           status );
stream_$close( stream_id, status );

END.
```

A.8. EC2_CONSUMER.PAS

This program is invoked by the program EC2_PRODUCER.PAS. (see Section A.7).

```
PROGRAM ec2_consumer;

{ This program uses user-defined eventcounts to read data sent by another
  program, EC2_PRODUCER. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

    TYPE
        shared_info_t =    { fields in mapped storage record }
            RECORD
            producer_ec    : ec2_$eventcount_t; { producer eventcount       }
            consumer_ec    : ec2_$eventcount_t; { consumer eventcount       }
            done           : boolean;           { set to true when you
                                                  reach end of input file   }
            record_present : boolean;           { set to true when you copy
                                                  record from input file    }
            useful_data    : string;            { record from input file    }
            data_len       : integer32;         { length of useful_data     }
            END;

    VAR
        status              : status_$t;       { status code                 }
        length_mapped       : integer32;       { length of mapped data       }
        stream_id           : stream_$id_t;    { stream ID                   }
        seek_key            : stream_$sk_t;    { where stream data starts    }
        shared_info         : ^shared_info_t;  { address of first byte
                                                 of mapped storage          }
        producer_wait_value : integer32;       { trigger value when waiting
                                                 for consumer eventcount    }
        i                   : integer;         { value returned by ec2_$wait }

    PROCEDURE check_status;

        BEGIN
        IF status.all <> status_$ok THEN
            BEGIN
            error_$print( status );
            pgm_$exit;
            END;
        END;
```

```
BEGIN

{ Open a stream to "consume" records into. }

stream_$create( 'output_file',          {object to be created  }
                11,                     { name length          }
                stream_$overwrite,      { access type          }
                stream_$no_conc_write,  { concurrency          }
                stream_id,              { stream ID - returned }
                status );               { completion status    }
check_status;

{ Map the shared data file containing the eventcounts. }

shared_info := ms_$mapl( 'shared_file',  { object to be mapped          }
                         11,             { name length                  }
                         0,              { first byte to map            }
                         sizeof( shared_info_t ),
                                         { no. bytes to map             }
                         ms_$cowriters,  { locking mode                 }
                         ms_$wr,         { access type                  }
                         true,           { map length in 3rd parameter,
                                           even if object is shorter    }
                         length_mapped,  { bytes mapped - returned      }
                         status );
check_status;

WITH shared_info^ DO
    BEGIN

    { Store the current value of the producer eventcount
      so you can use it as a trigger value later. }

    producer_wait_value := ec2_$read( producer_ec );
```

```
    REPEAT { wait for and process records }

    { Wait until a record is available. Check RECORD_PRESENT
      before you wait. }

        WHILE NOT record_present DO
            BEGIN
            producer_wait_value := producer_wait_value + 1;
            i := ec2_$wait( addr( producer_ec ),  { ptr to eventcount  }
                            producer_wait_value,  { trigger value      }
                            1,                    { no. of eventcounts }
                            status );
            END;

        { If a record is available, write it to the output file. }

        IF NOT done THEN
            BEGIN
            stream_$put_rec( stream_id,           { stream ID      }
                             addr( useful_data ),{ data to put    }
                             data_len,            { length of data }
                             seek_key,            { seek key       }
                             status );
            check_status;
            END;

        { Set condition field to false and advance consumer eventcount. }

        record_present := false;
        ec2_$advance( consumer_ec, status );
        check_status;

    UNTIL done;

    END;

{ Clean up and terminate. }

ms_$unmap( shared_info,    { first byte of mapped object }
           length_mapped,  { length of mapped object     }
           status );
stream_$close( stream_id, status );

END.
```

A.9. MUTEX_INIT.PAS

Before you execute this program, create the file TICKET_SALES_FILE in the directory where you will execute MUTEX_INIT.

```
PROGRAM mutex_init;

{ This program shows how to initialize a mutex lock record.  It maps a file
  that contains a lock record and ticket sales count.  The program
  initializes the lock record and sales count and then exits. Before you execute
  this program, use the DM editor to create a file named TICKET_SALES_FILE. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';
%INCLUDE '/sys/ins/mutex.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  TYPE
    ticket_sales_t =
      RECORD
      lock_record    : mutex_$lock_rec_t;  { mutex lock record }
      tickets_sold   : integer;            { counter          }
      END;

  VAR
    status              : status_$t;          { status  code }
    ticket_info         : ^ticket_sales_t;
    length_mapped       : integer32;          { length of mapped data }
    tickets_wanted      : integer;
    wait_time           : time_$clock_t;
    lock                : boolean;

  PROCEDURE check_status;                     { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;
```

```
BEGIN


{ Map a data file containing the mutex lock. Use this file to hold data
  of type TICKET_SALES_T. }

ticket_info := ms_$mapl( 'ticket_sales_file',    { object to be mapped      }
                         17,                     { length of name           }
                         0,                      { first byte to map        }
                         sizeof( ticket_sales_t ),{ no. bytes to map        }
                         ms_$nr_xor_1w,          { locking mode             }
                         ms_$wr,                 { access type              }
                         true,                   { map length in
                                                   parameter, even if
                                                   object is shorter        }
                         length_mapped,          { bytes mapped - returned  }
                         status );

check_status;

ticket_info^.tickets_sold := 0;

{ Initialize the mutex lock. }
mutex_$init ( ticket_info^.lock_record );

writeln( 'Mutex lock file has been initialized.  Start lock users now.' );

{ Unmap file and exit. }

ms_$unmap( ticket_info,
           sizeof( ticket_sales_t ),
           status );
check_status;

END.
```

A.10. MUTEX_USER.PAS

Before you execute this program, use MUTEX_INIT.PAS (see Section A.9) to initialize the lock record. You can run this program concurrently from different processes.

```
PROGRAM mutex_user;

{ This program uses MUTEX calls to lock a file that contains ticket sales
  information.  When the program gets a mutex lock, it determines whether
  there are enough tickets to fill the ticket order.  If there are tickets,
  the program updates the ticket count and unlocks the file.  The program
  processes orders until tickets are sold out.

  Before you execute this program, run MUTEX_INIT to initialize the ticket
  sales file and the mutex lock record.}


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/cal.ins.pas';
%INCLUDE '/sys/ins/ms.ins.pas';
%INCLUDE '/sys/ins/mutex.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  TYPE
    ticket_sales_t =                      { ticket sales file }
      RECORD
      lock_record    : mutex_$lock_rec_t;  { mutex lock record }
      tickets_sold   : integer;            { counter          }
      END;

  VAR
    status              : status_$t;          { status  code }
    ticket_info         : ^ticket_sales_t;
    length_mapped       : integer32;          { length of mapped data }
    tickets_wanted      : integer;
    wait_time           : time_$clock_t;
    lock                : boolean;


  PROCEDURE check_status;                       { for error handling }

    BEGIN
    IF status.all <> status_$ok THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;

  BEGIN

  { Map a shared data file containing the mutex lock.
    Use this file to hold data of type TICKET_SALES_T. }
```

```
ticket_info := ms_$mapl( 'ticket_sales_file',     { object to be mapped     }
                         17,                      { length of name          }
                         0,                       { first byte to map       }
                         sizeof( ticket_sales_t ), { no. bytes to map       }
                         ms_$cowriters,           { concurrency mode        }
                         ms_$wr,                  { access type             }
                         true,                    { map length in 3rd
                                                    parameter, even if
                                                    object is shorter       }
                         length_mapped,           { bytes mapped - returned }
                         status );

check_status;

{ Get system clock value for 30 seconds. }
cal_$sec_to_clock ( 30, wait_time );

{ Keep looping as long as you can get a lock and there are tickets left. }

WITH ticket_info^ DO

  BEGIN
  WHILE tickets_sold < 100 DO

    BEGIN {while }

    write ( 'Input the number of tickets you want : ' );
    readln ( tickets_wanted );

    { Get a mutex lock on the file.  If you get the lock, try
      to buy tickets.  Otherwise, assume there's a problem and exit.
      The theater holds 100 people.}

    lock := mutex_$lock( lock_record, wait_time );
    IF NOT lock THEN
      BEGIN
      writeln( 'Problem locking file.  Try again later.' );
      ms_$unmap( ticket_info,
                 sizeof( ticket_sales_t ),
                 status );
      check_status;
      RETURN;
      END;

    IF tickets_sold + tickets_wanted > 100  THEN
      writeln ( 'Only', 100 - tickets_sold:4, ' tickets left.' )
    ELSE
      BEGIN
      tickets_sold := tickets_sold + tickets_wanted;
      writeln ( 'You got', tickets_wanted:4, ' tickets.' );
      END;

    mutex_$unlock ( lock_record );

    END; { while }
  END;   { with  }
```

```
{ Unmap file and exit. }

ms_$unmap( ticket_info,
           sizeof( ticket_sales_t ),
           status );
check_status;

END.
```

A.11. MBX_SERVER.PAS

```
PROGRAM mbx_server ( input,output );

{ This mailbox server creates a mailbox and handles requests from clients.
  The server handles open requests, close requests, and data transmissions. }

%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    get_message_loop,
    done;


  CONST
    mbx_bufsize = mbx_$serv_msg_max;   { channel buffer size              }
    mbx_maxchan  = mbx_$chn_max;       { no. mailbox channels - 255       }
    mbx_name     = 'test_mailbox';     { mailbox name                     }
    mbx_namelen  = sizeof( mbx_name ); { length of mailbox name           }
    srv_msg_len  = mbx_$serv_msg_max;  { length of buffer that server can
                                         use to receive messages          }

  TYPE
    server_rec_t =
      RECORD                           { server message  }
      mbx_hdr   : mbx_$msg_hdr_t;      { header has 3 fields:
                                           cnt  - message length
                                           mt   - message type
                                           chan - message channel }
      msg_data : ARRAY [ 1..mbx_$msg_max ] OF char;
      END;
    server_rec_ptr_t  =
      ^server_rec_t;                   { pointer to a server message }

  VAR

    mbx_handle    : univ_ptr;          { a pointer to the mailbox            }
    status        : status_$t;         { a status code                       }
    srv_msg_buf   : server_rec_t;      { a buffer that the server can use to
                                         receive messages                    }
    mbx_retptr    : server_rec_ptr_t;  { a pointer to the buffer where MBX
                                         placed a retrieved message          }
    mbx_retlen    : integer32;         { length of a retrieved message       }
    send_msg_buf  : server_rec_t;      { a buffer where the server places
                                         a message to send                   }
    open_chan     : integer16;         { number of open channels to the mailbox}
```

```
PROCEDURE check_status;                          { for error handling }

  BEGIN
  IF status.all <> status_$ok THEN
    BEGIN
    error_$print( status );
    pgm_$exit;
    END;
  END;


PROCEDURE process_data;

  CONST
    reply       = 'Message written.';
    reply_len   = sizeof( reply );

  VAR
    reply_array : ARRAY [ 1..reply_len ] OF char;
    i           : integer;           { a counter }

  BEGIN
  { Display the client's message. }

  writeln ( mbx_retptr^.msg_data : mbx_retlen
          - mbx_$serv_msg_hdr_len );

  { Construct a header for a return message. }

  send_msg_buf.mbx_hdr.cnt    := reply_len
                                 + mbx_$serv_msg_hdr_len;
  send_msg_buf.mbx_hdr.mt     := mbx_$data_mt;
  send_msg_buf.mbx_hdr.chan   := mbx_retptr^.mbx_hdr.chan;

  { Construct the data portion of a message. }

  reply_array                 := reply;
  FOR i := 1 TO reply_len DO
    send_msg_buf.msg_data[i] := reply_array[i];

  { Send the return message. }
  mbx_$put_rec( mbx_handle,
                addr( send_msg_buf ),
                send_msg_buf.mbx_hdr.cnt,
                status );
  check_status;
  END;

BEGIN { program test_server }

{Initialize variables.}
```

```
  open_chan := 0;

  { Create the mailbox. }

  mbx_$create_server( mbx_name,     { name             }
                     mbx_namelen,  { name length      }
                     mbx_bufsize,  { buffer size      }
                     mbx_maxchan,  { maximum channels }
                     mbx_handle,   { handle           }
                     status );
  check_status;

  writeln( 'Mailbox ', mbx_name, ' was successfully opened.' );

  { Keep getting messages until there are no more clients. }

get_message_loop:

  REPEAT
    mbx_$get_rec( mbx_handle,
                  addr( srv_msg_buf ),  { where message may be received }
                  srv_msg_len,          { length of message buffer      }
                  mbx_retptr,           { where message is received     }
                  mbx_retlen,           { message length                }
                  status );
    check_status;

    WITH mbx_retptr^ DO
      BEGIN
      writeln( 'Message received from channel ', mbx_hdr.chan:4 );

      CASE mbx_hdr.mt OF

      { If message is an open channel request, accept it.  Also, keep
        track of the number of open channels. }

        mbx_$channel_open_mt :
          BEGIN
          send_msg_buf.mbx_hdr.cnt := mbx_$serv_msg_hdr_len;
                                      { there's no data in this message }
          send_msg_buf.mbx_hdr.mt  := mbx_$accept_open_mt;
                                      { type of message you are sending }
          send_msg_buf.mbx_hdr.chan := mbx_hdr.chan;
                                      { channel to send to              }

          mbx_$put_rec( mbx_handle,
                        addr( send_msg_buf ),    { message to send      }
                        mbx_$serv_msg_hdr_len,   { length of message    }
                        status );
          check_status;

          writeln ( ' Open request from channel ',
                    mbx_hdr.chan:4, ' has been accepted.' );

          open_chan := open_chan + 1;

          IF mbx_retlen <> mbx_$serv_msg_hdr_len THEN
            process_data;
          END;
```

```
        { If message is a close channel request, deallocate the channel
          and decrement the open channel count. }

        mbx_$eof_mt :
          BEGIN
          mbx_$deallocate( mbx_handle,
                           mbx_hdr.chan,  { channel number }
                           status );
          check_status;

          writeln ( 'Channel ', mbx_hdr.chan:4,
                    ' was deallocated.' );

          open_chan := open_chan - 1;

          IF open_chan = 0 THEN
            GOTO done;
          END;

        { If message is a data transmission or a partial data transmission,
          process the data. }

        mbx_$data_mt :
          process_data;

        mbx_$data_partial_mt :
          process_data;

        OTHERWISE
          writeln ( 'Invalid message type' );

        END;    { case statement }
      END;      { with statement }
  UNTIL open_chan = 0;

  { Close mailbox and exit. }

done:

  mbx_$close( mbx_handle,
              status );
  check_status;
  writeln ( 'The mailbox has been closed.' );

  END.    { program test_server }
```

A.12. MBX_CLIENT.PAS

This program uses the mailbox created by MBX_SERVER.PAS (Section A.11). Execute the client from the same directory where you execute the server.

```
PROGRAM mbx_client ( input,output );

{ This mailbox client opens a channel to a mailbox.  The client sends
  messages to the server, and gets the server's reply. Execute this
  program from the same directory where MBX_SERVER is running. }



%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';

  LABEL
    done;

  CONST
    mbx_name    = 'test_mailbox';   { mailbox name              }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name  }
    buf_len     = mbx_$msg_max;     { length of data buffer     }
    msg_buf_len = mbx_$msg_max;     { length of message buffer  }
    prompt_str  = 'Enter a new message: ';

  TYPE
    msg_t       =
      ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =
      ^msg_t;     { pointer to a client message }

  VAR
    data_buf    : msg_t;            { buffer for data to be sent          }
    mbx_handle  : univ_ptr;         { mailbox handle                      }
    status      : status_$t;        { a status code                       }
    msg_buf     : msg_t;            { a buffer that the client can use to
                                      receive data                        }
    msg_retptr  : msg_ptr_t;        { a pointer to the buffer where MBX
                                      placed a retrieved message          }
    msg_retlen  : integer32;        { length of a retreived message       }

  PROCEDURE check_status;                       { for error handling }

    BEGIN
    IF (status.all <> status_$ok) AND
       (status.all <> mbx_$partial_record) THEN
      BEGIN
      error_$print( status );
      pgm_$exit;
      END;
    END;
```

```
  BEGIN

{ Open a channel to the mailbox. }

  mbx_$open ( mbx_name,       { name                }
              mbx_namelen,    { length of name      }
              NIL,            { no data being sent  }
              0,              { length of data      }
              mbx_handle,     { handle              }
              status );
  check_status;

  { Read data and put it in the mailbox. }

  write( 'Enter a message; end with CTRL/Z: ' );
  WHILE NOT eof DO
    BEGIN
    readln( data_buf );
    mbx_$put_rec ( mbx_handle,              { handle           }
                   addr( data_buf ),        { buffer with data }
                   buf_len,                 { length of message }
                   status );

{ Get a response from the server. }

    mbx_$get_rec( mbx_handle,               { handle                    }
                  addr( msg_buf ),          { buffer for message        }
                  msg_buf_len,              { length of buffer          }
                  msg_retptr,               { actual location of message }
                  msg_retlen,               { length of message         }
                  status );
    check_status;

    writeln ( msg_retptr^: msg_retlen );
    write( prompt_str );
    END;

done:

  mbx_$close( mbx_handle,
              status );

  check_status;

  END.
```

A.13. MBX_GET_EC.PAS

This program uses the mailbox created by MBX_SERVER.PAS (see Section A.11). Execute the client from the same directory where you execute the server.

```
PROGRAM mbx_get_ec ( input, output );

{ This program uses a mailbox eventcount to determine when to get messages
  from a mailbox. Execute this program from the same directory where
  MBX_SERVER is running. }



%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

  LABEL
    done;

  CONST
    kbd_ec      = 1;                { index for keyboard event }
    mbx_ec      = 2;                { index for mbx event      }
    mbx_name    = 'test_mailbox';   { mailbox name             }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name }
    msg_buf_len = mbx_$msg_max;     { length of message buffer }

  TYPE
    msg_t       =
      ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =
      ^msg_t;     { pointer to a client message }

  VAR
    empty       : boolean;          { TRUE if mailbox is empty }
    ec2_ptr     : ARRAY [ 1..2 ] OF ec2_$ptr_t;
                                    { array of eventcount pointers }
    ec2_val     : ARRAY [ 1..2 ] OF integer32;
                                    { array of trigger values }
    which       : integer;          { integer that indicates an event type }
    data_buf    : msg_t;            { buffer where stream data can be read }
    data_retptr : msg_ptr_t;        { buffer where stream data is read     }
    linelen     : integer32;        { size of stream data line             }
    seek_key    : stream_$sk_t;
    mbx_handle  : univ_ptr;         { mailbox handle }
    status      : status_$t;        { status code    }
    msg_buf     : msg_t;            { buffer that the client can use to
                                      get data }
    msg_retptr  : msg_ptr_t;        { buffer where MBX places a retrieved
                                      message }
    msg_retlen  : integer32;        { length of a retrieved message }
```

```
  BEGIN

{ Get an eventcount that changes when there's input from the keyboard. }

  stream_$get_ec ( stream_$stdin,           { stream id                 }
                   stream_$getrec_ec_key,   { type of eventcount        }
                   ec2_ptr[kbd_ec],         { where pointer is returned }
                   status );
  IF (status.all <> status_$ok) THEN
    RETURN;

{ Open a channel to the mailbox.  Then get an eventcount that changes when
  there are messages in the mailbox. }

  mbx_$open ( mbx_name,                     { name            }
              mbx_namelen,                  { length of name  }
              NIL,                          { no data to send }
              0,                            { length of data  }
              mbx_handle,                   { handle          }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,                  { mailbox                   }
               mbx_$getrec_ec_key,          { type of eventcount        }
               ec2_ptr[mbx_ec],             { where pointer is returned }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  { Initialize the trigger values using the current eventcount values. }

  ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
  ec2_val[mbx_ec] := ec2_$read( ec2_ptr[mbx_ec]^ );

  { prompt for input }

  writeln( 'Enter a message; end with CTRL/Z: ' );


{ Now go into an infinite loop to wait for events.  When there is keyboard
  input, go to the kbd_ec loop. When there is a mailbox message, go to
  the mbx_ec loop. }

  REPEAT
    which := ec2_$wait( ec2_ptr,        { list of pointers to eventcounts }
                        ec2_val,        { trigger values                  }
                        2,              { no. eventcounts in the list     }
                        status );
    IF status.all <> status_$ok THEN
      RETURN;
```

```
CASE which OF

{ For keyboard input, create a loop to read the current eventcount and
  increase it by one -- this is the new trigger value. Then get
  keyboard input. When there is no more input, exit from the loop. }

  kbd_ec:

    REPEAT
      ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ ) + 1;
      stream_$get_conditional( stream_$stdin,      { stream id          }
                               addr( data_buf ),   { buffer for data    }
                               sizeof( data_buf ), { length of buffer   }
                               data_retptr,        { where data is read }
                               linelen,            { length of data     }
                               seek_key,           { seek key           }
                               status );
      IF (status.all <> status_$ok)
         AND (status.code <> stream_$end_of_file)
      THEN
        RETURN;

      IF (status.subsys = stream_$subs)
         AND (status.code = stream_$end_of_file)
      THEN
        GOTO done;

      IF linelen > 0 THEN
        BEGIN
        mbx_$put_rec ( mbx_handle,                 { handle         }
                       data_retptr,                { data to send   }
                       linelen,                    { length of data }
                       status );
        IF status.all <> status_$ok  THEN
          RETURN;
        END;
    UNTIL linelen = 0;
```

```
{ For a mailbox message, create a loop to read the current eventcount and
  increase it by one -- this is the new trigger value. Then get the
  mailbox message. When there are no more messages, exit from the loop. }

    mbx_ec:

      REPEAT
        empty := false;
        ec2_val[mbx_ec] := ec2_$read( ec2_ptr[mbx_ec]^ ) + 1;
        mbx_$get_conditional( mbx_handle,        { handle          }
                              addr( msg_buf ),   { buffer for data }
                              msg_buf_len,       { length of buffer }
                              msg_retptr,        { actual data     }
                              msg_retlen,        { length of data  }
                              status );
        IF (status.all <> status_$ok)
           AND (status.all <> mbx_$channel_empty)
           AND (status.all <> mbx_$partial_record)
        THEN
          RETURN;

        IF (status.all = mbx_$channel_empty) THEN
          empty := true
        ELSE
          writeln ( msg_retptr^: msg_retlen );
      UNTIL empty;

    END; { case }

  UNTIL false;

done:

  mbx_$close( mbx_handle,
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  END.
```

A.14. MBX_PUT_EC.PAS

This program uses the mailbox created by MBX_SERVER.PAS (see Section A.11). Execute the client from the same directory where you execute the server.

```
PROGRAM mbx_put_ec ( input,output );

{ This mailbox client uses eventcounts to wait for input from the
  keyboard or from a mailbox.  The client also uses the putrec eventcount
  to wait for a previously full channel to accept a message.  Execute this
  program from the same directory where MBX_SERVER is running. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/mbx.ins.pas';
%INCLUDE '/sys/ins/streams.ins.pas';
%INCLUDE '/sys/ins/ec2.ins.pas';

  LABEL
    done;

  CONST
    kbd_ec      = 1;                  { index for keyboard event }
    get_ec      = 2;                  { index for mbx get event  }
    put_ec      = 3;                  { index for mbx put event  }
    mbx_name    = 'test_mailbox';     { mailbox name             }
    mbx_namelen = sizeof( mbx_name ); { length of mailbox name   }
    msg_buf_len = mbx_$msg_max;       { length of message buffer }

  TYPE
    msg_t       =  ARRAY [ 1..mbx_$msg_max ] OF char; { client message }
    msg_ptr_t   =  ^msg_t;           { pointer to a client message     }

  VAR
    channel_full : boolean;          { TRUE if channel is full  }
    empty        : boolean;          { TRUE if mailbox is empty }
    ec2_ptr      : ARRAY [ 1..3 ] OF ec2_$ptr_t;
                                     { array of eventcount pointers }
    ec2_val      : ARRAY [ 1..3 ] OF integer32;
                                     { array of trigger values }
    which        : integer;          { integer that indicates an event type }
    data_buf     : msg_t;            { buffer where stream data can be read }
    data_retptr  : msg_ptr_t;        { buffer where stream data is read     }
    linelen      : integer32;        { size of stream data line             }
    seek_key     : stream_$sk_t;
    mbx_handle   : univ_ptr;         { mailbox handle }
    status       : status_$t;        { status code    }
    msg_buf      : msg_t;            { buffer that the client can use to get
                                       data  }
    msg_retptr   : msg_ptr_t;        { buffer where MBX places a retrieved
                                       message }
    msg_retlen   : integer32;        { length of a retrieved message }
```

```
  BEGIN

{ Get an eventcount that shows when there's input from the keyboard. }

  stream_$get_ec ( stream_$stdin,          { stream to get eventcount for }
                   stream_$getrec_ec_key,  { type of eventcount   }
                   ec2_ptr[kbd_ec],        { where to put pointer }
                   status );
  IF (status.all <> status_$ok) THEN
    RETURN;

{ Open a channel to the mailbox.  Then get the getrec and putrec eventcounts. }

  mbx_$open ( mbx_name,           { name             }
              mbx_namelen,        { name length      }
              NIL,                { no data to send  }
              0,                  { length of data   }
              mbx_handle,         { handle           }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,         { mailbox to get eventcount for }
               mbx_$getrec_ec_key, { type of eventcount            }
               ec2_ptr[get_ec],    { where to put the pointer      }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  mbx_$get_ec( mbx_handle,         { mailbox to get eventcount for }
               mbx_$putrec_ec_key, { type of eventcount        }
               ec2_ptr[put_ec],    { where to put the pointer  }
               status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  { Initialize the trigger values for kbd_ec and get_ec using the current
    event count values.  Initialize the trigger value for put_ec as one
    greater than the current value. }

  ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
  ec2_val[get_ec] := ec2_$read( ec2_ptr[get_ec]^ );
  ec2_val[put_ec] := ec2_$read( ec2_ptr[put_ec]^ ) + 1;

  { Initialize variable. }
  channel_full := false;

  { Prompt for input. }

  writeln( 'Enter a message; end with CTRL/Z: ' );
```

```
{ Now go into an infinite loop to wait for events.  When there is keyboard
  input, go to the kbd_ec loop. When there is a mailbox message to get,
  go to the get_ec loop. When a previously full channel has room to accept
  a mailbox message, go to the put_ec loop. }

  REPEAT
    which := ec2_$wait( ec2_ptr,        { list of eventcount pointers }
                        ec2_val,        { list of eventcount values   }
                        3,              { number of pointers in list  }
                        status );
    IF status.all <> status_$ok THEN
      RETURN;

    CASE which OF

    { For keyboard input, create a loop to read the current eventcount
      and increase it by one -- this is the new satisfaction value
      for the ec2_$wait call in the outer loop. Then get keyboard input.
      When there is no more input, exit from the loop. }

      kbd_ec:

        REPEAT
          ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ ) + 1;

          { If the channel is full, return to EC2_$WAIT.  When there is room
            in the channel, the put_ec section will change channel_full
            to FALSE. }

          IF channel_full = true THEN
            EXIT;
          stream_$get_conditional( stream_$stdin,       { stream id           }
                                   addr( data_buf ),    { buffer for data     }
                                   sizeof( data_buf ),  { size of buffer      }
                                   data_retptr,         { where data is copied}
                                   linelen,             { length of data      }
                                   seek_key,
                                   status );
          IF (status.all <> status_$ok)
             AND (status.code <> stream_$end_of_file)
          THEN
            RETURN;

          IF (status.subsys = stream_$subs)
             AND (status.code = stream_$end_of_file)
          THEN
            GOTO done;
```

```
        IF linelen > 0 THEN

        { If mbx_$put_rec_cond fails because the channel is full, set
          channel_full to TRUE. }

          BEGIN
          mbx_$put_rec_cond ( mbx_handle,        { handle       }
                              data_retptr,       { data to send }
                              linelen,           { length of data }
                              status );
          IF (status.all <> status_$ok)
             AND (status.all <> mbx_$no_room_in_channel)
          THEN
            RETURN;
          IF status.all = mbx_$no_room_in_channel THEN
            channel_full := true;
          END;
        UNTIL linelen = 0;

{ For a mailbox message, create a loop to read the current eventcount and
  increase it by one -- this is the new satisfaction value. Then get the
  mailbox message. When there are no more messages, exit from the loop. }

      get_ec:

        REPEAT
          empty := false;
          ec2_val[get_ec] := ec2_$read( ec2_ptr[get_ec]^ ) + 1;
          mbx_$get_conditional( mbx_handle,          { handle              }
                                addr( msg_buf ),     { buffer for data     }
                                msg_buf_len,         { length of buffer    }
                                msg_retptr,          { where data is copied }
                                msg_retlen,          { length of data      }
                                status );
          IF (status.all <> status_$ok) AND
             (status.all <> mbx_$channel_empty) AND
             (status.all <> mbx_$partial_record) THEN
             RETURN;

          IF (status.all = mbx_$channel_empty) THEN
            empty := true;

          IF NOT empty THEN
            writeln ( msg_retptr^: msg_retlen );
        UNTIL empty;

      { When the putrec eventcount is satisfied, try to put the message
        in the mailbox.  If there still is no room, go back to EC2_$WAIT.
        If the MBX_$PUT_REC_COND succeeded, set the keyboard eventcount
        so that you will check to see if there are any more keyboard messages. }
```

```
      put_ec:

        BEGIN
        ec2_val[put_ec] := ec2_$read( ec2_ptr[put_ec]^ ) + 1;
        mbx_$put_rec_cond( mbx_handle,     { handle           }
                           data_retptr,    { data to send     }
                           linelen,        { length of data   }
                           status );
        IF status.all <> status_$ok THEN
          IF status.all = mbx_$no_room_in_channel THEN
            EXIT
          ELSE RETURN;
        channel_full := false;

        { Set the keyboard satisfaction value so that the ec2_$wait call will
          indicate a keyboard event when you return to the ec2_$wait loop. }

        ec2_val[kbd_ec] := ec2_$read( ec2_ptr[kbd_ec]^ );
        END;

      END; { case }

  UNTIL false;

done:

  mbx_$close( mbx_handle,        { mailbox to close channel in }
              status );
  IF (status.all <> status_$ok) THEN
    RETURN;

  END.
```

A.15. IPC_SERVER.PAS

```
PROGRAM ipc_server;

{ This program accepts datagrams from a client and displays
  the message portion of the datagram.  The program also
  illustrates techniques for acknowledging datagrams and
  remaining synchronized with the client. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/ipc.ins.pas';


  CONST

    socket_file =  'server_handle_file'; { file for socket handle    }
    wait_sec    = 300;                   { number of seconds to wait }


  TYPE                                    { user-defined format for datagram}
    datagram_t =                          { that fits into 128-byte header  }
      RECORD
      msg_number : integer;               { 2 bytes for sequence no.        }
      msg_text   : ARRAY [ 1..126 ] OF char; { 126 bytes for data           }
      END;

  VAR
    status                : status_$t;           { status code                }
    local_socket_handle   : ipc_$socket_handle_t;{ handle for your socket     }
    rcv_socket_handle     : ipc_$socket_handle_t;{ handle for received datagram}
    expected_msg_number   : integer;             { datagram you expect        }
    receive_buf           : datagram_t;          { buffer to receive datagram }
    receive_len           : integer;             { length of received datagram }
    text_len              : integer;             { length of message text     }
    ack_buf               : integer;             { acknowledgment             }
    data_buf              : integer;             { buffer for data portion    }
    data_len              : integer;             { length of data portion     }
```

```
PROCEDURE program_cleanup;          { Cleanup procedure }

  BEGIN

    ipc_$close( socket_file,              { file containing handle }
                sizeof( socket_file ),    { length of filename     }
                status );
    IF status.all <> status_$ok THEN
      error_$print_name (status, 'Error closing socket.', 21);

    ipc_$delete( socket_file,             { file containing handle }
                 sizeof( socket_file ),   { length of filename     }
                 status );
    IF status.all <> status_$ok THEN
      error_$print_name (status, 'Error deleting handle file.', 27);

  END;

PROCEDURE program_exit;     { Program exit procedure }

  BEGIN

    program_cleanup;
    pgm_$exit;

  END;


BEGIN          { ipc_server }

{ Create file to contain a socket handle.}

ipc_$create ( socket_file,            { file for handle    }
              sizeof( socket_file ),  { length of filename }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error creating handle file.', 27);
    pgm_$exit;
  END;

{ Open a socket so you can receive datagrams. }

ipc_$open( socket_file,             { file for handle        }
           sizeof( socket_file ),   { length of filename     }
           4,                       { socket depth           }
           local_socket_handle,     { handle for your socket }
           status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error opening socket.', 21);
    program_exit;
  END;

writeln ( 'Socket opened successfully.');
```

```
{ Initialize datagram sequence number. }

expected_msg_number := 1;


{ Enter loop to wait for and acknowledge incoming datagrams.
  Acknowledge a datagram if it is valid and it contains a
  sequence number that you are expecting. }


REPEAT                                  { begin receive loop }

    { Wait up to 5 minutes for a datagram. }

  REPEAT                                {  begin wait loop }

    ipc_$wait( local_socket_handle,     { your socket handle }
               wait_sec * 4,            { wait 5 minutes     }
               status );

    { If a datagram arrived, exit from the wait loop and get the datagram. }

    IF status.all = status_$ok THEN
      EXIT;

    { It the call timed out, wait again. }

    IF status.all = ipc_$timeout THEN
      NEXT;

    { If another error occurred, print error and repeat wait loop. }
    error_$print( status );

  UNTIL false;                          { end wait loop }


  { Get the datagram. }

  ipc_$rcv( local_socket_handle,          { your socket handle             }
            sizeof( ipc_$hdr_info_t ),    { maximum size of header         }
            0,                            { maximum size of data           }
            rcv_socket_handle,            { where the datagram came from   }
            receive_buf,                  { buffer for header              }
            receive_len,                  { length of header               }
            data_buf,                     { buffer for data                }
            data_len,                     { length of data                 }
            status );

  { If there's an error, print an error message and return to the top
    of the receive loop to wait for a new datagram. }

  IF status.all <> status_$ok THEN
    BEGIN
    error_$print_name (status, 'Error getting the datagram.', 27);
    NEXT;
    END;
```

```
{ If you successfully got a datagram, make sure that it's valid before
  you use its contents.  First, check that the size is valid. If the
  size is not valid, print an error message and return to the top
  of the receive loop to wait for a new datagram. }

IF receive_len > sizeof( datagram_t ) OR ELSE
   receive_len < sizeof( datagram_t.msg_number )
THEN
  BEGIN
  error_$print_name (status, 'Bad datagram length.', 20);
  NEXT;
  END;

{ Check the sequence number to make sure you remain synchronized
  with the sender.  If the sequence number is bad, return to the
  top of the receive loop to wait for a new datagram.  Discard any
  datagram whose sequence number is larger than you expect.  Also,
  discard any datagram whose sequence number is two less than the
  number you expect because this is an old datagram that you have
  already processed. }

IF receive_buf.msg_number > expected_msg_number OR ELSE
   receive_buf.msg_number < (expected_msg_number - 1)
THEN
  BEGIN
  error_$print_name (status, 'Received out of sequence.', 26);
  NEXT;
  END;

{ If you got the datagram you expected, process it by
  displaying it on the screen. Then increment the sequence number.
  If you got a datagram whose sequence number is one less than
  what you expected, assume that the sender never got your
  acknowledgment.  Don't process the datagram, but send another
  acknowledgment. }

IF receive_buf.msg_number = expected_msg_number THEN
  BEGIN
    text_len := receive_len - sizeof( datagram_t.msg_number );
    writeln ( 'Received message: ', receive_buf.msg_text : text_len );
    expected_msg_number := expected_msg_number + 1;
  END;

{ Send an acknowledgment for the datagram you just processed,
  or for a datagram whose acknowledgment was lost.
  Send the sequence number for the datagram you are acknowledging. }
```

```
  ack_buf := receive_buf.msg_number;
  ipc_$send( rcv_socket_handle,        { where to send ack          }
             local_socket_handle,      { your handle                }
             ack_buf,                  { the number you're sending  }
             sizeof( ack_buf ),        { length of the number       }
             data_buf,                 { data portion               }
             0,                        { length of data portion     }
             status );

  IF status.all <> status_$ok THEN
    error_$print_name (status, 'Error sending acknowledgment.', 29);

  { When you receive a 'q' then quit running. }

  IF ( text_len = 1 ) AND
     ( receive_buf.msg_text[1] = 'q' ) THEN
     program_exit;

UNTIL false;          { end receive loop }

END.
```

A.16. IPC_CLIENT.PAS

This program uses a socket opened by IPC_SERVER.PAS (see Section A.15). Execute this program from the same directory where you execute the server.

```
PROGRAM ipc_client;

{ This program prompts for input and sends it to an IPC server, using a
  datagram.  The program waits for an acknowledgment before prompting
  for new input.  Execute this program from the same directory where
  IPC_SERVER is running. }


%INCLUDE '/sys/ins/base.ins.pas';
%INCLUDE '/sys/ins/error.ins.pas';
%INCLUDE '/sys/ins/pgm.ins.pas';
%INCLUDE '/sys/ins/ipc.ins.pas';


  CONST

    socket_file         = 'client_handle_file'; { file for own socket handle  }
    server_socket_file = 'server_handle_file';  { file with server's handle   }
    wait_sec = 10;                               { number of seconds to wait   }

  TYPE                                  { user-defined format for datagram   }
    datagram_t =                        { that fits into the 128-byte header }
      RECORD
      msg_number : integer;                    { 2 bytes for sequence no.     }
      msg_text : ARRAY [ 1..126 ] OF char;     { 126 bytes for data           }
      END;

  VAR
    status                : status_$t;
    local_socket_handle   : ipc_$socket_handle_t;{ handle for your socket     }
    server_socket_handle  : ipc_$socket_handle_t;{ handle for server's socket }
    rcv_socket_handle     : ipc_$socket_handle_t;{ handle for received datagram }
    current_msg_number    : integer;    { datagram you are sending           }
    send_buf              : datagram_t; { buffer with datagram to send       }
    send_len              : integer;    { length of datagram you are sending }
    text_len              : integer;    { length of message text             }
    receive_buf           : datagram_t; { buffer in which to receive datagram }
    receive_len           : integer;    { length of received datagram        }
    data_buf              : integer;    { buffer for data portion of datagram }
    data_len              : integer;    { length of data portion of datagram  }
    count                 : integer;
    send_count            : integer;
    wait_count            : integer;

  LABEL
    message_received;
```

```
PROCEDURE program_cleanup;              { Clean-up procedure }

  BEGIN

  ipc_$close( socket_file,              { file containing handle }
              sizeof( socket_file ),    { length of filename     }
              status );
  IF status.all <> status_$ok THEN
    error_$print_name (status, 'Error closing socket.', 21);

    ipc_$delete( socket_file,           { file containing handle }
                 sizeof( socket_file ), { length of filename }
                 status );
    IF status.all <> status_$ok THEN
      error_$print_name (status, 'Error deleting handle file.', 27);

  END;

PROCEDURE program_exit;         { Program exit procedure }

  BEGIN

  program_cleanup;
  pgm_$exit;

  END;


BEGIN       { ipc_client }

{ Create file to contain a socket handle. }

ipc_$create ( socket_file,              { file for handle    }
              sizeof( socket_file ),    { length of filename }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error creating handle file.', 27);
    pgm_$exit;
  END;

{ Open a socket so you can receive datagrams. }

ipc_$open( socket_file,             { file for handle        }
           sizeof( socket_file ),   { length of filename     }
           4,                       { socket depth           }
           local_socket_handle,     { handle for your socket }
           status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error opening socket.', 21);
    program_exit;
  END;
```

```
{ Get the server's socket handle. }

ipc_$resolve( server_socket_file,          { file with server's handle }
              sizeof( server_socket_file ), { length of filename        }
              server_socket_handle,         { server's handle           }
              status );

IF status.all <> status_$ok THEN
  BEGIN
    error_$print_name (status, 'Error resolving socket name.', 28);
    program_exit;
  END;

{ Initialize datagram sequence number. }

current_msg_number := 1;

{ Enter loop to prompt for messages, send datagrams, and wait for
  acknowledgments. }

REPEAT      { begin get message loop }

  { Prompt for message. }

  writeln ( 'Enter a message ( q to quit) : ');
  readln ( send_buf.msg_text );

  { Determine the message length. }

  text_len := sizeof ( send_buf.msg_text );

  WHILE ( send_buf.msg_text[text_len] = ' ' ) AND
        ( text_len > 0 ) DO
    text_len := text_len - 1;

  { Define the sequence number and length for the datagram. }

  send_buf.msg_number := current_msg_number;
  send_len := text_len + sizeof( datagram_t.msg_number );

  { Send the datagram and wait for an acknowledgment. If you don't
    receive the acknowledgment within 10 seconds, then resend the
    datagram.  Try to send up to 5 times. }
```

```
FOR send_count := 1 TO 5 DO             { begin send loop }
  BEGIN
  ipc_$send( server_socket_handle,      { where to send datagram }
             local_socket_handle,       { your handle            }
             send_buf,                  { header you're sending  }
             send_len,                  { length of the  header  }
             data_buf,                  { data portion           }
             0,                         { length of the data     }
             status );

  { If there's an error, try sending again. }

  IF status.all <> status_$ok THEN
    BEGIN
      error_$print_name (status, 'Error sending datagram.', 23);
      NEXT;
    END;

  { If the send completed successfully, wait for an acknowledgment.
    If you get a bad acknowledgment, ignore it and wait for another one.
    Repeat the wait loop up to 3 times,  then resend the datagram. }

  FOR wait_count := 1 TO 3 DO   { begin wait loop }

    BEGIN

    ipc_$wait( local_socket_handle,  { your handle     }
               wait_sec * 4,         { wait 10 seconds }
               status );

    IF status.all <> status_$ok THEN
      BEGIN

      { If the wait timed out, exit from the wait loop and resend the
        datagram. If there was another type of error, display an
        error message and repeat the wait loop. }

      IF status.all = ipc_$timeout THEN
        EXIT;

      error_$print_name (status, 'Error waiting for datagram.', 27);
      NEXT;
      END;

    { If the wait completed successfully, get the datagram and
      verify that it contains a valid acknowledgment. }

    ipc_$rcv( local_socket_handle,        { your socket              }
              sizeof ( ipc_$hdr_info_t ), { maximum size of header   }
              0 ,                         { maximum size of data     }
              rcv_socket_handle,          { where datagram came from }
              receive_buf,                { buffer for header        }
              receive_len,                { length of header         }
              data_buf,                   { buffer for data          }
              data_len,                   { length of data           }
              status );
```

```
        { If there's a receive error, repeat the wait loop. }

        IF status.all <> status_$ok THEN
          BEGIN
          error_$print_name (status, 'Error receiving datagram.', 25);
          NEXT;
          END;

        { If you got a datagram, make sure it came from the server.
          Otherwise, display an error message and repeat the wait loop. }

        IF server_socket_handle <> rcv_socket_handle THEN
          BEGIN
          error_$print_name (status,
                             'Received message from unexpected socket.', 40);
          NEXT;
          END;

        { If the datagram came from the right socket, make sure it's
          the right length.  If it is, then check that the sequence number
          is correct.If either condition is false, then display an error
          message and repeat the wait loop. }

        IF receive_len <> sizeof( datagram_t.msg_number ) OR ELSE
           receive_buf.msg_number <> current_msg_number

        THEN
          BEGIN
            error_$print_name (status, 'Received bad acknowledgment.', 28);
            NEXT;
          END;

        { If you got a good acknowledgment, exit from the wait loop
          and the send loop. }

        GOTO message_received;

        END;            { end of wait loop }

      END;                  { end of send message loop }

{ If you failed after 5 send attempts, display an error message and exit. }

  writeln ('Unable to communicate with foreign socket.');
  program_exit;
```

```
message_received:

  { If the message started with a 'q', exit from the program. Otherwise,
    increment the sequence number and repeat the get message loop. }

  IF ( text_len = 1 ) AND ( send_buf.msg_text[1] = 'q') THEN
    program_exit;

  current_msg_number := current_msg_number + 1;

UNTIL false;          { end get message loop }

END.
```

Index

P

S

U