Interprocess Communication:Programmer's Guide

900 Series HP 3000 Computers

HEWLETT PACKARD

HP Part No. 32650-90019
Printed in U.S.A.   19871101

E1187

Printing History

The following table lists the printings of this document, together with the respective release dates for each edition. The software version indicates the version of the software product at the time this document was issued. Many product releases do not require changes to the document. Therefore, do not expect a one-to-one correspondence between product releases and document editions.

Edition	**Date**	**Software Version**
First Edition	November 1987	A.01.00

List of Effective Pages

The List of Effective Pages gives the date of the current edition, and lists the dates of all changed pages. Unchanged pages are listed as "ORIGINAL". Within the manual, any page changed since the last edition is indicated by printing the date the changes were made on the bottom of the page. Changes are marked with a vertical bar in the margin. If an update is incorporated when an edition is reprinted, these bars and dates remain. No information is incorporated into a reprinting unless it appears as a prior update.

First Edition	November 1987

Preface

The *Interprocess Communication Programmer's Guide* (32650-90019) is written for the experienced MPE/iX programmer. It is a guide to the Interprocess Communication (IPC) features of the MPE/iX operating system, describing what these features are and illustrating how to use them to make your programs more powerful.

This manual forms part of the MPE/iX Programmer's Series documentation. This series consists of the *MPE/iX Intrinsics Reference Manual* (32650-90028) and a set of task-oriented user's guides. Before reading the *Interprocess Communication Programmer's Guide* you should know the information contained in two of these manuals:

- The *Process Management Programmer's Guide* (32650-90023) explains the concept of "process" on MPE/iX and describes the process handling commands and intrinsics available. Read this manual if you are unfamiliar with MPE/iX process handling features.
- The *Accessing Files Programmer's Guide* (32650-90017) discusses the MPE/iX file system and the types of files supported on MPE/iX. Read this manual if you are planning to use the file system features to perform IPC.

Organization of This Manual

Interprocess Communication contains the following chapters:

Chapter 1	**What Is Interprocess Communication?** defines Interprocess Communication (IPC) and describes the advantages of using it.
Chapter 2	**IPC Using Job Control Words and Other Variables** discusses the use of predefined and user-defined variables (including Job Control Words) in performing IPC.
Chapter 3	**IPC Using File System Intrinsics** discusses how to perform IPC using features of the MPE/iX file system, especially intrinsics.
Chapter 4	**Special Cases of IPC** explains how to use file system IPC in certain complex situations, or when special features are necessary.
Chapter 5	**NOWAIT I/O** describes the NOWAIT I/O feature of the file system and the intrinsics you use to set it up.
Chapter 6	**Software Interrupts** discusses the use of another file system feature, software interrupts, and how to write, initialize, and disarm them.
Chapter 7	**IPC Using the MAIL Facility** describes the MAIL facility, used in some systems to perform interprocess communication.
Appendix A	**Features of Intrinsics for Message Files** provides additional information about certain intrinsics used with message files in performing IPC.
Appendix B	**Sample Programs: WAIT I/O** illustrates the use of message files and WAIT I/O to perform interprocess communication.
Appendix C	**Sample Programs: NOWAIT I/O** provides examples showing the use of NOWAIT I/O in interprocess communication.
Appendix D	**Sample Programs: Software Interrupts** contains two sets of programs (one set in SPL and one in Pascal) showing the use of software interrupts in IPC.

How to Use This Manual

If you are unfamiliar with the concept of IPC, or the types of situations in which using it can be helpful, read Chapter 1. Chapter 2 will be helpful if you want to perform simple IPC within a single job or session.

Chapters 3 through 6, and the Appendixes, treat the features of the MPE/iX file system that you can use to perform IPC across job or session boundaries.

Refer to Chapter 7 if the MAIL facility is already in use on your system; this feature is not recommended if you are creating new programs or applications on MPE/iX.

Contents

Figures

1

What is Interprocess Communication?

Interprocess Communication (IPC) is a facility of the MPE/iX operating system that permits multiple processes to pass information among themselves. Large tasks that have been broken into independent processes can use IPC to synchronize their actions and exchange data. There are several ways you can implement IPC on MPE/iX.

Purposes of IPC

IPC is a useful tool for improving the performance of your system in a variety of complex processing situations. For example:

- Programs must sometimes handle conflicting requirements. For example, a program may need to perform lengthy calculations while simultaneously monitoring a transaction that requires frequent service. If this program is broken up into several processes, one process can be dedicated to monitoring the transaction, ensuring that it gets the constant attention it requires. Other processes can perform other CPU-intensive functions. These processes can coordinate their efforts through the use of the IPC facility.
- Some tasks require more processor time than is available on a single machine. These tasks can be divided into several processes and spread across multiple machines in a network. File system IPC can be used to pass data to different processes on remote machines. Dividing processes across machines also takes advantage of the other benefits of distributed data processing.
- Reliability is improved because processes must interface through well-defined IPC records. These interfaces are easily understood and verified. Large data structures become resources that are managed by specialized processes. Other processes request or update the data with a set of special commands passed through IPC. Unauthorized access or unintentional corruption can be closely controlled.
- When tasks are divided into independent processes using file system IPC, testing them becomes easier. Inputs and outputs to the various processes become IPC records and tend to become well-defined commands and responses. The processes can be tested individually. Editors can be used to build the input records. The output can be easily checked or redirected to a terminal or printer.
- Finally, programs can be implemented more quickly. The overall task can be divided into small pieces. Each programmer can work on a separate piece and run it as an independent process. The development of the pieces can occur in parallel. The pieces are modular and fit together through well-defined interfaces using IPC.

Ways to Perform IPC

A simple method of performing IPC within a single job or session is the use of variables and Job Control Words (JCWs). This feature is described in Chapter 2.

The file system intrinsics provide the most powerful method of performing IPC. These intrinsics can be used to communicate between a process and any other process; the processes need not be part of the same process tree or running in the same job or session. Communication takes place through the use of message files. Chapters 3-6 discuss the file system intrinsics used to perform IPC and give examples of how to set up communications between processes.

Some systems use the MAIL facility to communicate between processes in the same process tree. Each process in the tree can use this facility to send information to a parent or child process. The MAIL facility is covered in Chapter 7 of this manual.

2

IPC Using Job Control Words and Other Variables

Processes belonging to the same job or session can communicate with each other through session-level variables. Job Control Words (JCWs) are a subset of session variables. All session variables, including JCWs, reside in the session variable table.

Note Variables are used in the same way in both jobs and sessions. In this chapter, the term "session variable" is used to refer to both job-level and session-level variables.

Types of Variables

Session variables on MPE/iX can be either predefined or user-defined. A user-defined variable is one that you define yourself, and it is valid for the duration of the session in which you defined it. A predefined variable is defined for you by the system when you start a session. Some predefined variables are READ-only; you cannot change their value. (These are sometimes referred to as "system-reserved" variables.) Other predefined variables are READ-WRITE, and you can change their value within a session.

Relationship of JCWs and Variables

A session variable can have an integer, Boolean, or character string value. Commands and intrinsics exist to create, alter, access, and delete these variables.

JCWs are session variables that have been created or last altered by one of the JCW commands or intrinsics. These commands and intrinsics accept only integer values in the range of 0 through 65535. The only differences between a JCW and a standard integer variable are the limited value range of JCWs, and an internal "flag," set by the JCW commands and intrinsics, that identifies a variable as a JCW. (Also, JCW names cannot include the underbar character.)

In general, the process of setting up and using any type of variable for IPC is the same. However, if your program utilizes both JCWs and other types of variables, you need to be aware that there are separate commands and intrinsics for JCWs that do not treat standard variables the same way as JCWs. However, the commands and intrinsics for variables treat variables and JCWs the same way, since JCWs are simply a subset of session-level variables. See the section "Special Considerations" at the end of this chapter for more discussion of this subject.

The following list summarizes the commands and intrinsics used to manipulate JCWs and standard variables. These commands and intrinsics are discussed in this chapter within the context of their use in interprocess communication. For a complete discussion of how to set up, use, change, and delete variables and JCWs, refer to *Command Interpreter Access & Variables Programmer's Guide* (32650-90011).

`:SETVAR`	Adds or alters a variable in the variable table. The value can be Boolean, string, or 32-bit integer.
`:SETJCW`	Adds or alters a JCW in the variable table. The value must be an integer in the range 0 through 65535.
`:SHOWVAR`	Shows specified variable(s) in the variable table, including JCWs.
`:SHOWJCW`	Shows specified JCW(s) in the variable table.
`:DELETEVAR`	Deletes any user-created variable, including JCWs, from the variable table.
`HPCIPUTVAR`	Adds or alters a variable in the variable table. The value can be Boolean, string, or 32-bit integer.
`PUTJCW`	Adds or alters a JCW in the variable table. The value must be an integer in the range 0 through 65535.
`HPCIGETVAR`	Returns the value of the named variable.
`FINDJCW`	Returns the value of the named JCW.
`SETJCW`	Sets the JCW named `JCW` to the value passed.
`GETJCW`	Returns the current value of the JCW named `JCW`.
`HPCIDELETE-VAR`	Deletes an entry from the variable table.

Variables with integer, string, or Boolean type values can be used for interprocess communication within a job or session. Information, values, and status information can be passed between processes by using variables.

Variable Intrinsics

The intrinsics used for manipulating variables are the following:

HPCIPUTVAR

The `HPCIPUTVAR` intrinsic is used to set a variable. The intrinsic call could be:

```
        HPCIPUTVAR(VARNAME,STATUS,KEYWORD,KEYVALUE);
```

where `VARNAME` is a valid MPE/iX variable name. (Variable names can be up to 255 characters. The first character of the name must be an alphabetic letter or the underbar character. Letters, numbers, and underbars are allowed in the name.) The status of the procedure is indicated in the `STATUS` parameter.

The `KEYWORD` parameter indicates the variable type (that is, what is returned in `KEYVALUE`). These are:

1	Integer value
2	String value

3	Boolean variable

HPCIGETVAR

The `HPCIGETVAR` intrinsic scans the variable table for a given variable and returns its value. The intrinsic call could be:

```
HPCIGETVAR(VARNAME,STATUS,KEYWORD,KEYVALUE);
```

where `VARNAME` is a valid MPE/iX variable name. The status of the procedure is returned in `STATUS`. The `KEYWORD` parameter has the same meaning as in `HPCIPUTVAR`; refer to the description above.

HPCIDELETEVAR

You use `HPCIDELETEVAR` to remove a variable from the variable table. The intrinsic call could be:

```
HPCIDELETEVAR (VARNAME,STATUS);
```

where `VARNAME` is a valid MPE/iX variable name. The status of the procedure is returned in `STATUS`.

Note that there is no `DELETE` intrinsic specifically for Job Control Words. Since JCWs are a type of variable, you can use the `HPCIDELETEVAR` intrinsic to delete a JCW.

For more information and full syntax descriptions for the above intrinsics, consult the *MPE/iX Intrinsics Reference Manual* (32650-90028).

Variable Commands

You can also manipulate variables by using various MPE/iX commands. These are the following:

:SETVAR

This command can be used to set or alter a variable (including JCWs) in the variable table. The syntax is as follows:

`:SETVAR` *varname, expression*

The *varname* must be a valid MPE/iX variable name, either of an existing variable or of a new one. A space or comma must precede the *expression* parameter, if you use it. *Expression* can be a string, an integer, a Boolean value, an expression, or the name of another variable. `SETVAR` always evaluates *expression*; certain operators can be used in this parameter, as well as numeric values.

:SHOWVAR

Use this command to display a selected variable or variables from the variable table. This command displays JCWs as well as standard variables.

Entering `:SHOWVAR` without parameters shows you a list of all variables and values that you have set. You can enter a variable name (or more than one, separated by commas) to see the variable's value.

You can also use "wild card" characters such as `?`, `#`, `[m-n]`, and `@`. Entered by itself, `@` causes `:SHOWVAR` to display a list of all variables, both user-created and predefined. Or you can use it in a variable name to signify "zero or more alphanumeric characters."

:DELETEVAR

This command enables you to delete any user-created variable from the variable table. Again, you can delete JCWs as well as standard variables by using this command.

The one required parameter is *varname*, the name of the variable to be deleted. You may enter more than one *varname* at a time, separated by commas. Predefined variables cannot be deleted. The same wild card characters allowed in `:SHOWVAR` are also valid here.

For more information and syntax descriptions for the above commands, refer to the *MPE/iX Commands Reference Manual* (32650-90003).

Job Control Words

Job Control Words (JCWs) are a subset of variables. A JCW is an integer variable in the range 0 through 65535, and contains an internal "flag" (set by one of the JCW commands or intrinsics) that makes it recognizable to the system as a JCW. All the variable intrinsics and commands discussed above can be used to manipulate JCWs; however, the JCW commands and intrinsics will recognize and affect only those variables flagged as JCWs.

It is possible to reclassify JCWs as variables, or vice versa, by using the appropriate intrinsics to change their values. However, not all JCWs and variables can be changed. The JCWs `CIERROR` and `JCW` cannot be reclassified as standard variables, because system processes depend upon the fact that these variables can always be accessed and altered using JCW intrinsics. Also, predefined standard variables such as `HPPATH` cannot be reclassified as JCWs.

For a list of predefined variables and JCWs, refer to the *MPE/iX Commands Reference Manual*, Appendix A.

Using the Predefined JCW for Interprocess Communication

Two processes belonging to the same job or session can communicate with each other through the predefined Job Control Word `JCW`. This word enables a subsystem process to return information to the job or session that initiated the process. This communication mechanism is used by the command executors for `:RUN`, and by various subsystem commands (for example, compilers). However, you may find this control word helpful in other applications.

Using User-Defined JCWs for Interprocess Communication

MPE/iX allows you to establish and manipulate user-defined Job Control Words. This capability overcomes one of the disadvantages of using the system-defined Job Control Word: MPE/iX uses `JCW` for status information, and you cannot be sure that MPE/iX will not modify it, thus possibly changing the information you want to pass in `JCW`. (MPE/iX clears bits (0:2) of `JCW`, but the remaining bits are user-definable.)

A user-defined JCW is a 16-bit word that resides in the session variable table managed by MPE/iX. This table, which also holds the system-defined JCWs, is shared by all processes in a job or session; thus any process of a job can access any JCW in the table.

Job Control Word Intrinsics

The following intrinsics are used to manipulate JCWs. Note that the first intrinsic affects only the predefined Job Control Word called `JCW`; the rest can be used with any JCW in the session variable table.

SETJCW

The `SETJCW` intrinsic is used to set the bits in the predefined Job Control Word `JCW`. A `SETJCW` intrinsic call could be:

```
SETJCW(WORD);
```

where `WORD` is a 16-bit logical value whose bits are set by you. If you set bit (0:1)=1, the system displays the following message when your program terminates, either normally or due to an error:

```
PROGRAM TERMINATED IN AN ERROR STATE (CIERR 976)
```

Bits (1:15) may be set to any pattern.

Note In batch mode, the job is terminated unless the `:CONTINUE` command is used. If you have a JCW of exactly %140000 (bits (0:2) only), the "`CIERR 976`" message is replaced by "`CIERR 989, PROGRAM ABORTED PER USER REQUEST`". Refer to the *MPE/iX Commands Reference Manual* (32650-90003) for a discussion of `:CONTINUE`.

The predefined Job Control Word `JCW` can be read by a process with the `GETJCW` intrinsic. The form of the `GETJCW` intrinsic call is:

```
JCW:=GETJCW
```

The Job Control Word is returned to `JCW`.

For example, consider a job where two processes in the same process tree pass information to each other through `JCW`. In one process, you transmit the contents of the word `PROCLNK` to `JCW`. Process A sets `JCW` to `PROCLNK` as follows:

```
SETJCW(PROCLNK);
```

When process B is executed, it obtains the value of `JCW` through the `GETJCW` intrinsic. In this case, the contents of `JCW` are returned to the word `STORELNK`.

```
        STORELNK:=GETJCW;
```

PUTJCW

The PUTJCW intrinsic is used to establish a user-defined JCW, or to change the value of an existing JCW. This intrinsic scans the variable table for the JCW name. The name of a user-defined JCW must be alphanumeric, must start with a letter, and must be between 1 and 255 characters long. If the JCW name is found, the value of JCW is updated to the value passed by PUTJCW. If not found, the JCW name is added to the table and assigned the value passed with the name. For example, the intrinsic call:

```
        PUTJCW(JCWNAME,JCWVALUE,STATUS);
```

searches the variable table for a name that matches the name contained in JCWNAME (a character array). If the name exists, its value is updated to the value contained in JCWVALUE. If the name is not found, the name is added to the table and assigned the value contained in JCWVALUE.

The STATUS parameter of PUTJCW indicates the status of the intrinsic call and returns an integer value to indicate this status, as follows:

0	Successful execution.
1	Error. JCWNAME is longer than 255 characters.
2	Error. JCWNAME does not start with a letter.
3	Error. The variable table is out of space.
4	Error. Attempted to assign a value to an MPE/iX-defined JCW value mnemonic (OK, WARN, FATAL, or SYSTEM).
5	Error. Cannot assign a value to a system-reserved JCW (for example, PUTJCW(HPMONTH) is illegal).
6	Warning. A standard variable has been reclassified as a JCW. (This happens when a variable set with SETVAR has its value altered with PUTJCW or SETJCW.)
7	Error. Cannot reclassify predefined standard variables as JCWs (for example, PUTJCW (HPPROMPT) is illegal).

FINDJCW

The FINDJCW intrinsic is used to scan the variable table for a JCW and return its value. Thus, the intrinsic call:

```
        FINDJCW(JCWNAME,JCWVALUE,STATUS);
```

searches the variable table for a JCW of the same name as that contained in JCWNAME. If found, its current value is returned in JCWVALUE. If not found, an error is returned in STATUS, and JCWVALUE is returned unchanged.

The STATUS parameter of FINDJCW indicates the status of the intrinsic call and returns an integer value indicating this status as follows:

0	Successful execution.
1	Error. JCWNAME is longer than 255 characters.
2	Error. JCWNAME does not start with a letter.

3	Error. The JCW named in `JCWNAME` does not exist.

For more information and full syntax descriptions for the above intrinsics, refer to the *MPE/iX Intrinsics Reference Manual* (32650-90028).

Job Control Word Commands

Several MPE/iX commands are available that allow you to set, alter, or display JCWs. These are:

:SETJCW

This command adds a JCW to the variable table, or alters the value of an existing variable. The value must fall within the valid range for a JCW, that is, in the range 0 through 65535.

Command syntax is as follows:

`:SETJCW` *jcwname delimiter [+ or -] value*

The *jcwname* parameter can contain the name of a new or existing user-defined or system-defined JCW. You can use `@` to specify all currently defined JCWs.

The *delimiter* can be one or more punctuation characters or spaces, except for `%` or `-`. The *value* parameter must be one of the following:

1. An octal number between 0 and %177777.
2. A decimal number between 0 and 65,535.
3. An MPE/iX-defined JCW value mnemonic, or an offset value of a mnemonic.
4. The name of an existing JCW.

:SHOWJCW

This command displays the specified JCW or JCWs in the variable table.

`:SHOWJCW`, entered without parameters, displays all user-defined and system-defined JCWs currently in effect. The optional parameter *jcwname* must be the name of a valid JCW.

For more information and full syntax descriptions for the above commands, refer to the *MPE/iX Commands Reference Manual* (32650-90003).

Note There is no `DELETE` intrinsic or command that applies specifically to Job Control Words. Since a JCW is a type of variable, you can use the `HPCIDELETEVAR` intrinsic or the `:DELETEVAR` command to delete it.

Special Considerations

When your program uses both JCW and variable commands and intrinsics, there are some considerations you should keep in mind. Remember that a Job Control Word is a type of variable, but not all variables are JCWs. The variable commands and intrinsics operate on all entries in the variable table; they do not distinguish between JCWs and standard variables. On the other hand, JCW commands and intrinsics deal only with variable table entries that are flagged as JCWs.

Displaying JCWs and Variables

When you use `PUTJCW` or `:SETJCW` to set or alter a variable, that variable is "flagged" as a JCW. If you use the `:SHOWJCW` command, all JCWs set or last altered with `PUTJCW` or `:SETJCW` are shown.

When you create a new variable with `HPCIPUTVAR` or `:SETVAR`, that variable is not flagged as a JCW, even if it is within the valid range for a JCW. So if you use `:SHOWJCW`, that variable does not appear. Use the `:SHOWVAR` command, which displays both JCWs and standard variables.

It is possible to reset an existing JCW by using `HPCIPUTVAR` or `:SETVAR`. If you do so, and the new value is still within the valid range for a Job Control Word, then the JCW remains a JCW and you can display it with `:SHOWJCW`.

Warning Messages

If you use `HPCIPUTVAR` or `:SETVAR` to change a JCW to a value that is *outside* the valid JCW range, you see the following warning message:

```
JCW VARIABLE RECLASSIFIED AS A STANDARD VARIABLE.
(CIWARN 8126)
```

This variable is no longer flagged as a JCW, and you must use `:SHOWVAR` to see it. `FINDJCW` will no longer return it; use `HPCIGETVAR` instead.

Likewise, you can use `PUTJCW` or `:SETJCW` to change a variable that was originally set using `HPCIPUTVAR` or `:SETVAR`. If the variable is within the valid range of JCW values, it becomes a JCW, and you see the following warning message:

```
STANDARD VARIABLE RECLASSIFIED AS A JCW VARIABLE.
(CIWARN 8127)
```

You can display this variable using either `:SHOWVAR` or `:SHOWJCW`.

Note that the value passed with the variable name is assigned to the variable; the command or intrinsic did its job. However, the variable is now classified differently.

Additional Discussion

The *Command Interpreter Access & Variables Programmer's Guide* (32650-90011) contains an extensive discussion of variables and JCWs. Refer to that document for further information.

3

IPC Using File System Intrinsics

There are several ways processes can communicate under MPE/iX. The most powerful is the IPC facility provided by the file system.

Characteristics

This type of IPC has several advantages because it is part of the file system. Most functions are performed with standard file system intrinsics that you are already familiar with. The cooperating processes find each other through an agreed-upon file name. Thus they don't have to determine each other's process ID.

The file system IPC facility uses a FIFO queue structure. Sending processes queue multiple messages that are stored until a receiver reads them, even across system shutdowns. Receiving processes are allowed to wait for messages on one or more empty queues. Messages are deleted from the queue as they are read.

The cooperating processes using IPC do not need to be related; that is, they don't need to be part of the same process tree. They can even be running on different machines on the same network.

There are several different ways to perform I/O. The `:FILE` command can be used to redirect the I/O to another disc device (local or remote), or to change the way in which the message file is accessed. The existing file system security features can also be used.

The heart of the file system IPC facility is the "message file." Message files reside partly in memory and partly on disc. MPE/iX uses the memory buffer part as much as possible, to achieve the best performance. The disc portion of the message file is used only as secondary storage in case the memory buffer part overflows. For many users of IPC, MPE/iX never accesses the disc portion of the message file.

Creating a Message File

Message files can be created in several ways. To create a message file with the :BUILD command, use the MSG keyword. For example, to build a message file named PAULA, enter:

```
        :BUILD PAULA; MSG
```

When a user process opens a new file programmatically and indicates that it will be a message file, the HPFOPEN intrinsic creates the new message file. Or HPFOPEN can explicitly reference a :FILE command. Use the MSG keyword in the :FILE command to create a new message file:

```
         :FILE LARS, NEW; MSG
```

When you perform a :LISTF,2 command, message files are identified by an M in the third column of the TYP field. For example:

```
   FILENAME  CODE -----LOGICAL RECORD------ -----SPACE-----
               SIZE   TYP  EOF   LIMIT  R/B  SECTORS  #X  MX

   PAULA       128W   VBM    0   1031     1    258     1   8
```

How To Use IPC—A Simple Case

IPC can be relatively easy to set up. Suppose that a large programming task is to be divided into two processes. One process will interface with the user. This process is referred to as the "supervisor" process. It does some processing tasks itself and offloads others to a "server" process. This process only handles requests from the supervisor and returns the results. The following paragraphs describe how to set up the communications between these two processes.

Note In this chapter, the MPE/iX versions of the file system intrinsics (for example, HPFOPEN) will be discussed. You may also use the MPE V/E versions of these intrinsics (for example, FOPEN) to perform IPC; there is no need to rewrite existing programs, since these intrinsics are compatible with respect to IPC.

Program Structure

Like most other files, message files need to be opened explicitly with the HPFOPEN intrinsic. Two-way communication is needed, so each process opens two message files: a Command File for supervisor-to-server commands and a response file for server-to-supervisor responses.

The supervisor opens the command file with WRITE-ONLY access, and the response file with READ-ONLY access. The server opens the Command File with READ-ONLY access, and the response file with WRITE-ONLY access. The HPFOPEN parameters are similar to any other HPFOPEN these processes would perform on another file. The processes communicate by using file names known to both of them.

Message File Names

To use IPC, processes must reference each other through a known file name. This means that the file must be in a directory that is available to all accessors. In the example in Figure 3-1, either a `:BUILD` must be done previously, or one process must first `HPFOPEN` the file as new and `FCLOSE` it as permanent, to put the file in the permanent file directory. The file can be created as temporary if all accessors are running under the same job or session. If you are concerned about security, create the file with a lockword.

IPC Processing

In addition to `HPFOPEN`, these processes use three other file system intrinsics to perform IPC: `FREAD`, `FWRITE`, and `FCLOSE`. These intrinsics are used exactly as they would be for any other type of file.

When the server starts executing, it performs an `FREAD` on the Command File. The Command File is empty, so the server process blocks in `FREAD` instead of getting an end-of-file condition.

As part of the supervisor's processing, it eventually writes to the Command File. Since the file is not full, there is nothing to block this `FWRITE`, so it completes almost immediately. The supervisor then reads the response file, which, since it is empty, causes the supervisor process to block.

MPE/iX notices that data has been written to the Command File, and that the server process is waiting in `FREAD` for data from that file. MPE/iX moves the data to the address that the server passed in the `FREAD` intrinsic, deletes the record from the Command Message File, and restarts that process.

The server exits from the `FREAD` and processes the command. When it has finished processing the command, the server writes its answer to the response file. At this point the server is finished. It can terminate, or it can issue another `FREAD` on the Command File and start the sequence over again.

MPE/iX moves the response data to the supervisor's buffer, deletes it from the file, and restarts the supervisor at the `FREAD`. The supervisor continues processing, possibly repeating the cycle. Eventually both processes close each of their files as part of terminating.

These processes are illustrated in the diagram shown in Figure 3-1:

```
      SUPERVISOR                     SERVER

       v                            v
     HPFOPEN "command" file         HPFOPEN "command" file
     HPFOPEN "response" file        HPFOPEN "response" file
       .                            v
       .                          FREAD "command" file
.      . (processing)                 |
.      .                              | (blocked)
.      .                              |
. T    v                  IPC         | (fread
. I  FWRITE "command" file -------->  _  completes)
. M    v                            .
. E  FREAD "response" file          .
.        |                          . (processing)
.        | (blocked)                .
v        |                          .
         | (fread          IPC      v
         _  completes) <--------- FWRITE "response" file
       v                            v
     FCLOSE "command" file        FCLOSE "command" file
     FCLOSE "response" file       FCLOSE "response" file
       v                            v
```

Figure 3-1. IPC Processing

End-of-File (EOF) Conditions

"EOF conditions" may prevent an I/O from completing (for example, an `FREAD` from an empty file, or an `FWRITE` to a full file). The I/O intrinsic may return an end-of-file indication, or it may wait until the condition is resolved (a record becomes available for the `FREAD`, or space in the file becomes available for the `FWRITE`).

The intrinsic waits if an EOF condition is encountered and either:

- There is an "opposite accessor" to the file, or
- This is the first I/O on this file since it was opened.

The reasoning is as follows:

- An "opposite accessor" is a process that is performing the opposite function on a file, for instance, an `FWRITE` when the process is waiting to perform an `FREAD`. If there is an opposite accessor, it is possible that the I/O request will be satisfied eventually. (If there were no opposite accessors, the process might wait forever.)
- When processes start up, there may be a "race" condition in which it is difficult to predict whether one process's I/O request will be made before another process's `HPFOPEN`. For this reason, the first call to an I/O intrinsic will wait if there is an EOF condition. This gives the opposite accessor time to open the file.
- Finally, a process may always want to wait rather than receive an `EOF`. It can do this by requesting "extended wait." This is discussed in more detail in Chapter 4. If extended wait is explicitly enabled, the process will always wait on an EOF condition.

In Figure 3-1, the server can open the Command File and issue the `FREAD` before the supervisor opens the Command File. The server's `FREAD` on the Command File waits because it is the first I/O after the open. But suppose that, after the first command is processed, the server issues another `FREAD` on the Command File, and the supervisor has terminated unexpectedly. The second `FREAD` will receive an `EOF`, signaling that something is wrong. This `FREAD` receives an `EOF` because the `FREAD` is not the first I/O from the server on the Command File, and there are now no writers accessing the Command File.

An `FCLOSE` by another process can cause an intrinsic that is waiting on an EOF condition to stop waiting and receive an EOF. If the process that is waiting is not using extended wait, and the last process that can resolve the EOF condition does an `FCLOSE` on the file, MPE/iX will wake up the waiting intrinsic and return an EOF condition.

Recovery From Abnormal Terminations

In the event of an abnormal termination of the processes involved in IPC, some unread records may be left in the message file. Some applications are concerned only with current data and do not want to see unprocessed data from a previous run. The simplest solution is to programmatically `:PURGE` and `:BUILD` all message files as part of the initialization processing.

Another possibility requires that the writer always open the file first. (The writer is the process sending messages to the message file.) If the writer's `HPFOPEN` specifies an access type option (item #11) of WRITE access only (item=1), and this is the only process with the file open, the records are automatically purged. If the writer's `HPFOPEN` specifies an Access Type Option of APPEND (item=3), then the old records are kept and the new records added to the end.

Note that if the reader opens the file first, regardless of the writer's access type, the records are kept. (The reader is the process using `FREAD` to receive messages from a message file.) To recover the data cleanly in an application in which the reader opens the file first, use `FLOCK` to prevent the writer from altering the file. Then use `FREAD` to read all the records and either discard or process them.

Sample Programs

For sample programs illustrating the use of message files and WAIT I/O to perform IPC, refer to Appendix B.

4

Special Cases of IPC

File system IPC provides several optional sophisticated features that you may find useful. These are described below.

Multiple Concurrent Readers or Writers

An application may need to use multiple concurrent readers or writers. For example, multiple supervisors may need the services of the same server. The server may not be able to keep up with the requests, and so multiple copies of it may be needed to handle the incoming requests.

Setting up a message file to be accessed by multiple concurrent readers or multiple concurrent writers is not difficult. However, there are some subtleties you should be aware of:

- Since each reader receives its own record when it does an `FREAD`, and there is no "broadcast" facility in IPC, there is no way for all readers to receive copies of the same record.
- If either readers or writers are waiting on an EOF condition, they are awakened in the order in which they called their I/O intrinsics.
- Although a common command queue works well for both multiple readers and multiple writers, it is advisable to have individual response queues. This ensures that a supervisor receives only the responses to its own commands. In this situation, applications often pass the name of the response message file, or the writer's ID, along with the command, so that the file can be opened and the response sent to the correct destination. (The writer's ID is discussed in the "Writer Identification" section in this chapter.)

Preventing Deadlocks

If multiple writers access a message file that has a small file limit, this can cause a deadlock due to the way MPE/iX monitors the writers that open and close the message file. MPE/iX does this by adding some extra records that are normally inaccessible. To leave room for these records, MPE/iX increases the file limit by 2 when the file is first created. This leaves enough room for one open and one close record *per file*.

As writers open the file and begin writing, they each use one record for an "open" record and reserve another for their "close" record. One record is used and the space for another is reserved *per writer*.

If the file has a very small file limit, or there is a large number of writers, it is possible to create a state in which all the space in the file is reserved. Writers are not able to write, because the file “looks” full (no available space). Readers are not able to read, because the file “looks” empty (no real records to be read). A deadlock has occurred.

To prevent such a deadlock, make the file limit larger than the number of writers; there should be room for more records than there will be writers. (You can set the limit when the file is created with the `:BUILD` command or the `HPFOPEN` intrinsic.) Another way to detect a deadlock (that is, using timeouts) is discussed in a later section of this chapter.

Writer Identification

Sometimes a reader needs to know when a new writer opens the file, the records written by this writer, and when the writer closes the file and hence no longer adds records. This allows the reader to keep track of who is sending records. It also helps the reader to manage its resources, perhaps by allocating a data segment when a new writer opens the file, and releasing it when the writer closes the file. IPC provides this feature through the use of `FCONTROL` with a controlcode of 46.

Readers who call `FCONTROL` with a controlcode of 46 receive two additional words at the beginning of each record returned by `FREAD`. The first word contains the record type (0 = data, 1 = open, 2 = close). The second word contains a writer's ID. The writer's ID is a unique identifier assigned by the MPE/iX file system to processes that have opened a message file with WRITE access, and it is used only to associate the open, close, and data writes performed by a specific writer. If additional information is needed (such as process ID number or program name), it can be passed by the writer in the first record, and saved by the reader in a table indexed according to the writer's ID.

Note The writer's ID is always written when a writer does an `FWRITE`. `FCONTROL` with a controlcode of 46 only makes it visible to the reader. Writers who call `FCONTROL` with a controlcode of 46 will receive an error. See the *MPE/iX Intrinsics Reference Manual* (32650-90028) for full syntax details on `FCONTROL`.

Extended Wait

If a process calls an intrinsic to perform an I/O and it encounters an EOF condition, it will either return a CCG condition code or wait in the intrinsic until the EOF condition is resolved. (See “EOF Conditions” in Chapter 3.) In some applications, it is useful that the process always wait instead of receiving a CCG. It may be known that another process will eventually open the file and satisfy the blocked I/O request.

MPE/iX lets the process request this extended wait mode through `FCONTROL` with a controlcode of 45. Extended wait always starts out disabled, but both readers and writers can enable or disable it.

If extended wait is explicitly enabled, the process always waits on an EOF condition. If extended wait is explicitly disabled, the process will not wait on an EOF condition unless there is an “opposite accessor” to the file. In the default condition (extended wait is not explicitly enabled or disabled), the process will wait if there is an opposite accessor, or if this is the first I/O to the file.

Another feature of extended wait also concerns opposite accessors. If a process is waiting for action by an opposite accessor, and the last opposite accessor closes the file, this will cause an EOF and “wake up” the waiting process.

Refer to the discussion of “EOF Conditions” in Chapter 3 for more information about this feature. Also, see Example B-2 in Appendix B.

Timeouts

Suppose one process is using extended wait, and, while it is waiting, the only process that could resolve the condition terminates unexpectedly. The waiting process will never wake up and must be aborted. A “timeout” allows the waiting process to detect the situation and terminate gracefully.

MPE/iX allows a process to set a timeout. If, after a specified time, the I/O still has not completed, the I/O intrinsic returns to the process. If there is a complex set of processes and message files, and they get into a state where all the processes are waiting on `FREAD`s and none of them can do an `FWRITE`, timeouts can help detect this deadlock. (An example of a complex set of processes and message files is a case in which there are processes that read from one message file and write to another.)

Timeouts are useful when a process must perform some time-sensitive processing (such as updating a table every *n* seconds) and, therefore, cannot wait for long periods of time. Some tasks are both command and time driven (that is, display status every *n* seconds or when asked) and therefore could use timeouts.

Timeouts are set (in seconds) using the `FCONTROL` intrinsic with a controlcode of 4, and they are valid for both readers and writers. Note that timeouts come into play only on EOF conditions. If `FREAD` or `FWRITE` times out, a CCL condition code is returned, and `FCHECK` returns `FSERR 22, Software Time-Out.`

Timeouts on terminals are valid only for the next I/O. Timeouts on message files stay in effect on every I/O until explicitly turned off. (See Example B-2 in Appendix B.)

Nondestructive Reads

Another potential problem occurs when a server is not able to honor a request in a message file record immediately because of a lack of resources. The requests must be handled in order, and so the server tries to free the resources it requires. Eventually the server comes back and tries again to process the request.

In this case it is useful to have a “nondestructive” read, that is, to look at the request record in the message file without deleting it. A test is made to see if the request can be satisfied. If it can, a regular destructive read is done and the request is processed. If the request cannot be satisfied, it stays on the top of the queue, where it can be read and tested at a later time.

This feature is provided by `FCONTROL` with a controlcode of 47. It is valid for readers only, and it affects only the following `FREAD`. Note that repeated use of nondestructive `FREAD` always reads the same record.

You can copy the message file to another file, if, for example, you want to save it after it has been read. You can do this by accessing it with the Copy Mode option (`HPFOPEN` Item #17, or `FOPEN` *aoption* (3:1)). You should note, however, that accessing a file in Copy Mode means that it does not have any of the special features of message files. For a description of the Copy Mode option, refer to Appendix A of this manual, or the *MPE/iX Intrinsics Reference Manual* (32650-90028).

Forcing Records To Disc

Message files reside primarily in buffers, and they can lose data in the event of a system crash. For applications that must use message files and must be "crash-proof," MPE/iX provides a way to force the data to disc.

This is done by using `FCONTROL` with a controlcode of 6. Either readers or writers can call this intrinsic. It forces all buffers to be written to disc and the disc file label to be updated. This causes several disc I/Os and, therefore, takes a relatively long time. It must be done after each `FWRITE` to be most effective, and even then there is a "window" between the `FWRITE` and the `FCONTROL` during which a crash would cause the record to be lost.

Remember that IPC is designed to be a fast, efficient means of passing messages between processes. If the task the application is performing is really event logging, and it must be highly crash-resistant, you should consider using "circular files" or other files with `FSETMODE`. (`FSETMODE` is ignored for message files.) For more information about `FSETMODE`, refer to *Accessing Files* (32650-90017) and the *MPE/iX Intrinsics Reference Manual* (32650-90028).

5

NOWAIT I/O

Sometimes a programmer wants an application to read or write a record, but does not want it to wait for I/O to complete. For such an application, waiting is wasting time when it could be doing other processing. Timeouts do not adequately address this problem. The programmer wants this application to start an I/O, continue processing immediately, and check periodically to see if the I/O has finished.

MPE/iX provides a way to solve this problem with NOWAIT I/O. This feature is requested by enabling the `NOWAIT I/O` option (item #16) in `HPFOPEN`.

When using NOWAIT I/O, the process must make at least two intrinsic calls to perform the I/O, one to start it and one to finish it. MPE/iX still handles the file in the same way. But instead of waiting for the I/O to complete, MPE/iX returns control to the application so that the application can do some useful processing.

NOWAIT I/O has been available to users of standard files for a long time, but to use it on standard files requires Privileged Mode. On standard files the mechanics of NOWAIT I/O prevent MPE/iX from protecting a process from corrupting its own stack. However, because Message files work differently, NOWAIT I/O on Message files does *not* require Privileged Mode.

NOWAIT I/O Intrinsics

To perform a NOWAIT I/O, the `FREAD` or `FWRITE` intrinsic must be called to initiate the transfer. These intrinsics return immediately, and no data is transferred yet. The return value for `FREAD` is set to zero and is not needed.

To complete the transfer, either `IODONTWAIT` or `IOWAIT` must be called. `IODONTWAIT` tests whether the I/O has finished. If it has, the intrinsic returns a condition code of CCE and the file number as the return value. If the I/O has not completed, CCE and a zero return value are passed back. If `IOWAIT` is called, it waits until the I/O has finished, like a normal WAIT I/O `FREAD` or `FWRITE`.

Only one NOWAIT I/O can be outstanding against a file by a particular accessor at a time; however, an accessor can have NOWAIT I/Os outstanding against several files at the same time. These I/Os can be completed by a “generalized” `IODONTWAIT` or `IOWAIT` (the file number parameter is zero or is omitted). In this case, these intrinsics report on the first I/O to complete, returning the file number for that file. If the call to one of these intrinsics is in a loop, then that one call can be used to complete all the NOWAIT I/Os.

Aborting NOWAIT I/O

Occasionally, after a process has started a NOWAIT I/O with `FREAD` or `FWRITE`, something occurs that causes completion of that I/O to be no longer needed. Perhaps the process is "shutting down" and does not want to wait for the I/O (that is, to issue `IOWAIT` or `IODONTWAIT`).

MPE/iX lets the process abort NOWAIT I/Os that have not yet completed by using `FCONTROL` with a controlcode of 43. A condition code of CCE is returned if the I/O was aborted; in this case, nothing more needs to be done. CCG is returned if the I/O has already completed; in this case, `IODONTWAIT` or `IOWAIT` must be called to clear it. CCL and `FSERR 79`, `No NOWAIT I/O pending for special file,` are returned if there was nothing to abort.

Limitations

Currently, MPE/iX does not support NOWAIT I/O to Message files across a network. In most cases this is not an important limitation, because it is rare that both readers and writers to the same message file need to use NOWAIT I/O. If the file is made local to the accessor that needs NOWAIT I/O, the other accessor can then do WAIT I/O across the network.

Examples

Appendix C contains two sample programs illustrating the use of NOWAIT I/O in IPC. (One program is in COBOL, and the other is in FORTRAN.)

6

Software Interrupts

NOWAIT I/O requires an application to "poll" to see if the requested I/O has completed. Each time the check is made, there is some overhead, whether the I/O has completed or not.

The application is faced with a difficult trade-off. The more often the application polls, the greater the overhead, and the poorer its overall performance becomes. If it polls less frequently, this increases the delay between when the I/O can complete and when the application completes it, thus reducing performance.

One solution to this dilemma is to use software interrupts. When software interrupts are enabled, MPE/iX signals the application when to complete the I/O. There is no need for repeated polling; the application completes the I/O only when signaled, so the I/O always completes on the first try.

Software interrupts are a special case of NOWAIT I/O. The difference is that MPE/iX interrupts the process when the I/O can be completed; the process does not need to poll to determine whether the I/O can be completed.

Most of the discussion about NOWAIT I/O also applies to software interrupts. Like NOWAIT I/O, a call to `IOWAIT` or `IODONTWAIT` is needed to complete an I/O request.

Example—Use of Software Interrupts

Software interrupts are most often used to handle high-priority requests while the process is doing low-priority time-consuming tasks.

For example, suppose an application is copying a large file across a network. The copy may take a long time (up to several hours). During this time, the application wants to see and respond to high-priority commands written to its command message file (for example, requests for the number of records copied so far, or that the copy stop immediately).

To accomplish this, the application posts a "software interrupt" `FREAD` against its command message file. The `FREAD` just signals to MPE/iX that this application wants to know about any new commands written to this message file. No data is transferred at the time the `FREAD` is called. Software interrupt `FREAD`s never wait; they return to the application immediately.

The application then starts to perform the copy. Unlike NOWAIT I/O, the application does not have to poll the command message file repeatedly to see if data has been written there. MPE/iX signals the application (with a software interrupt) that data has been written to the command message file and that the application should complete the I/O.

In our example, if a high-priority command is written to the message file, MPE/iX immediately causes a software interrupt. The part of the application performing the copy is stopped, and MPE/iX forces the execution of the application's interrupt handling procedure. There the application completes the read by calling `IOWAIT` or `IODONTWAIT`, and processes the command. When the interrupt handling procedure completes, the part of the application performing the copy resumes automatically at the statement where it left off.

Software Interrupt Intrinsics

Three intrinsics are specific to software interrupts: `FCONTROL` with a controlcode of 48, `FINTSTATE`, and `FINTEXIT`.

`FCONTROL` with a controlcode of 48 arms software interrupts for a particular file. It is also used by the application to tell MPE/iX the address of the application's interrupt handler.

Note The *param* parameter of `FCONTROL` is used to pass the interrupt handler's procedure label (*plabel*). Native Mode *plabels* are 32 bits long, while Compatibility Mode *plabels* are 16 bits. Therefore, the interrupt handler and the `FCONTROL` call to it must be compiled in the same mode. Arming software interrupts in cross-mode programming can be unpredictable, and you should avoid this whenever possible.

`FINTSTATE` is used by the application to enable or disable software interrupts for all files with interrupts armed by this process. `FINTSTATE` also returns a code which shows the status of the interrupts prior to the time it was invoked; it returns a 0 if software interrupts were disabled and a 1 if software interrupts were enabled.

`FINTEXIT` is used to return from the interrupt handler and leave software interrupts enabled or disabled.

The following section explains how to set up and use software interrupts.

Software Interrupt Initialization

As with most other files, the message file must be explicitly opened. Software interrupts are usually used when reading from the file, but there is nothing to stop a process from using them when writing. After the call to `HPFOPEN` the application can perform normal WAIT or NOWAIT I/O on the file.

When software interrupt operation begins, the process calls `FCONTROL` with a controlcode of 48, passing the *plabel* of its interrupt handler. `FCONTROL` with a controlcode of 48 returns the previous value of the *plabel*. If that value is zero, this means that software interrupts were not armed. If the program is using software interrupts on multiple files, `FCONTROL` with a controlcode of 48 must be called for each file. Each file can have its own interrupt handler, or more than one file can share the same interrupt handler. When a message file has been opened and `FCONTROL` has been called for the file with a controlcode of 48, that message file is said to have software interrupts "armed."

At this point, the process can start the I/O (in this example, assume it is `FREAD`). `FCONTROL` with a controlcode of 48 overrides the `HPFOPEN` NOWAIT I/O option setting. Regardless of the setting of this bit, an `IODONTWAIT` or `IOWAIT` is required to complete any I/O started after the `FCONTROL`, controlcode 48, software interrupt. A call to `IODONTWAIT` or `IOWAIT` before the interrupt occurs does not complete the I/O. A CCL is returned, and `FCHECK` returns the message, `FSERR 79--NO NOWAIT I/O PENDING FOR SPECIFIED FILE.`

Note If the `FREAD` intrinsic was called before `FCONTROL` with a controlcode of 48 was invoked, it would have been handled as a normal, non-software interrupt `FREAD`.

MPE/iX starts out with software interrupts disabled. If the `FREAD` is satisfied, the software interrupt is postponed until interrupts are explicitly enabled. The process uses `FINTSTATE` at this point to enable software interrupts for all "armed" files opened by this process. This includes all files currently armed, and any yet to be armed by this process, until `FINTSTATE` is called to disable interrupts. The call to `FINTSTATE` can occur anywhere in this sequence, but the other intrinsic calls should be made in the order given.

Interrupt Handler

The interrupt handler is a special procedure of the process, devoted to completing the I/O request after an interrupt occurs. This procedure is never called explicitly. Instead, MPE/iX invokes the interrupt handler when a software interrupt occurs, possibly from the middle of a statement. The procedure declaration can be either without parameters, or have a single `INTEGER` parameter into which MPE/iX puts the file number.

The file number parameter of the software interrupt handler is useful if there are several files sharing the same software interrupt handler procedure and some require special handling. For example, each file may have a different buffer address passed to `IOWAIT`.

MPE/iX automatically disables software interrupts and CONTROL-Y traps when it jumps to the interrupt handler. A call to `IODONTWAIT` or `IOWAIT` is needed to finish the I/O request. The I/O request can be completed immediately, so `IOWAIT` and `IODONTWAIT` will work the same way. (`IOWAIT` always completes immediately.)

Check the condition code (a good idea after any intrinsic call). CCG means the interrupt occurred because of an End-Of-File condition. If a CCE is returned, the process has a record. It can be processed here, or a flag can be set to indicate that the record has been received and should be handled during the normal non-interrupt processing.

Often, at this point, the next software interrupt read is set up by performing another `FREAD`, if the process wants to continue to see new records written to this message file. The `FREAD` does not have to be done here and could be performed elsewhere during normal processing. The process may not want to issue any `FREAD` at all, if the command just received was to stop immediately and terminate.

The last statement in the interrupt handler must be a call to `FINTEXIT`. This allows the process to pick up where it left off when the interrupt occurred, enable CONTROL-Y traps, and optionally leave software interrupts enabled or disabled. If this call is omitted, the process may never get back to where it was before the interrupt. Exiting with software interrupts enabled is usual, but the process may leave them disabled if the record needs special processing and it does not want any additional interrupts until it is completed. At that time it needs to call `FINTSTATE` to enable interrupts.

Main Line Code

For the most part, the “main line” code of a process does not need to be concerned with the I/O to message files using software interrupts. As long as interrupts are enabled, they can occur anywhere in user code. If one occurs during an MPE/iX intrinsic, the interrupt is postponed until you exit the intrinsic and re-enter the process' code.

There are some exceptions. Interrupts can occur during a “generalized” `IOWAIT`, during an `IOWAIT` on another message file not using software interrupts, or during a `PAUSE`.

The use of software interrupts introduces the possibility of a problem that applications normally do not have to think about. Some code is sensitive to interrupts. The problem usually occurs with data that is altered by both the interrupt handler and the main line code.

For example, suppose the main line code decrements a counter and the interrupt handler increments the same counter. The main line code loads the old value and subtracts one from it. Before it is stored back, an interrupt occurs. The interrupt handler loads the old value, increments it, and stores the new value back. The main line code resumes, storing its new value on top of the interrupt handler's new value, and the increment is lost.

One solution is to protect sensitive code by using `FINTSTATE (FALSE)` to disable interrupts before the operations and `FINTSTATE (TRUE)` to enable interrupts afterwards.

Disarming Software Interrupts

It is possible to shut down software interrupt operation and resume normal WAIT or NOWAIT I/O on the message file. If there was an I/O posted against the file (that is, a software interrupt `FREAD` or `FWRITE` that has not yet caused a software interrupt and, therefore, has not been completed by an `IOWAIT` or `IODONTWAIT`), you need to use `FCONTROL` with a controlcode of 43 to abort it, just as in NOWAIT I/O. If software interrupts were disabled with `FINTSTATE`, the I/O completed, and the interrupt postponed, then `FCONTROL` with a controlcode of 43 returns a CCG. This means that the I/O is too far along to be aborted. Interrupts need to be enabled to let the interrupt handler finish the request. Take care to ensure that the interrupt handler does not start another I/O, as this can cause a loop. (See the sample program at the end of this chapter for a suggested way to handle this.) Using `FCONTROL` with a controlcode of 48, but passing a zero instead of the *plabel*, disarms the interrupt routines for the file. `FCONTROL` with a controlcode of 48 will return a CCL if an I/O is pending.

Restrictions

There are some additional limitations. Currently, software interrupts are not available directly from COBOL or on remote files. If the process contains privileged code, the interrupt handler must be privileged to handle interrupts from it, or else an `ABORT 22, Invalid stack marker,` will occur.

If the code you want to interrupt is privileged, then your interrupt handling procedure must also be executing in Privileged Mode. If the code to be interrupted is non-privileged, then your interrupt handler can be either privileged or non-privileged.

The following restrictions apply to Compatibility Mode code only. In CM, your interrupt handler may reside in any system Segment Library. The hierarchy of Segment Libraries is as follows:

PROG = User program segments
GSL = Group segment libraries
PSL = Public segment libraries
SSL = System segment library

A routine on one level can call routines on that level or below. However, this distinction does not apply to software interrupts. So your interrupt handler can reside in any library and interrupt code in any other library.

There is one exception: code in the SSL (System Segment Library) cannot be interrupted. The reason is that MPE/iX routines cannot be interrupted, and the system cannot distinguish between the MPE/iX routines and the other routines residing in the SSL.

Sample Program—Use of Software Interrupts

The following sample program illustrates software interrupts and the use of the intrinsics discussed earlier in this chapter. It contains four sections: an interrupt handler procedure, a section that arms software interrupts and passes the *plabel* of this interrupt handler, the main processing loop, and a section to disarm software interrupts.

You should note several things about this program. First, the "main processing loop" is just a "placeholder" for purposes of illustration; in a real IPC situation, this section of the program would perform actual work. Second, note the use of a "flag" in the disarm section of the program to prevent the interrupt handler from starting another read of the message file when interrupts are re-enabled after being postponed. If you don't use a flag of this sort, your program may go into a continuous loop in this situation. For more examples, refer to Appendix D.

```
$uslinit$
$standard_level 'HP_PASCAL'$
{
   SOFTWARE INTERRUPT FRAGMENT

   Program including a software interrupt handler, software
   interrupt set-up, and software interrupt shutdown.

}
```

```
program intfrag1(input,output);

   type
      int = -32768..32767;
      rec = packed array [1..80] of char;

   var
      error,msg_file_num,plabel,dummy : int;
      msg_file_name : packed array [1..8] of char;
      done,need_another_read : boolean;
      msg_rec : rec;
      length : real;

function fopen : int; intrinsic;
procedure quit; intrinsic;
procedure fcheck; intrinsic;
procedure fintstate; intrinsic;
procedure fintexit; intrinsic;
procedure iowait; intrinsic;
procedure fread; intrinsic;
procedure fcontrol; intrinsic;
procedure fclose; intrinsic;
procedure pause; intrinsic;

{  INTERRUPT HANDLER ROUTINE  }

procedure inthandler(local_msg_file_num : int);
   begin

      {  Complete the read on the message file  }
      iowait(local_msg_file_num,msg_rec);
      if ccode <> 2 then
         begin
            fcheck(msg_file_num,error);
            quit(error);
         end;

      {  Perform any processing on the incoming record.  }
      writeln(msg_rec);

      {  Restart the read on the message file if needed.  }
      if need_another_read then
         fread(local_msg_file_num,msg_rec,-80);

      {  Re-enable interrupts when handler routine exits.}
      {  Same effect as FINTEXIT(-1).                    }
      fintexit;
   end;

begin
```

```
   need_another_read := true;

   {  INTERRUPT INITIALIZATION  }

   {  Open the file with FOPTION = old, ascii, }
   {   and AOPTION = read only }

   msg_file_name := 'MSGFILE1';
   msg_file_num := fopen(msg_file_name,5,0);
   if ccode <> 2 then
      begin
         fcheck(msg_file_num,error);
         quit(error);
      end;

   {  Arm software interrupts and pass interrupt handler}
   {  procedure address for THIS file.                  }
   plabel := waddress(inthandler);
   fcontrol(msg_file_num,48,plabel);
   if ccode <> 2 then
      begin
         fcheck(msg_file_num,error);
         quit(error);
      end;

   {Enable software interrupts for ALL files with software}
   {interrupts armed by this program.  (-1 = enable)      }
   fintstate(-1);

{ Start first read on message file.  This read acts }
{ like NOWAIT read (no data returned until IOWAIT called) }
{ because software interrupts are armed on this file. }
   fread(msg_file_num,msg_rec,-80);
   if ccode <> 2 then
      begin
         fcheck(msg_file_num,error);
         quit(error);
      end;

   {  MAIN PROCESSING OF PROGRAM  }

   done := false;
   length := 1.0;
   repeat
      begin
         writeln('Another pass through the main processing
          loop.');
         pause(length);
         length := length + 1.0;
         if length = 60.0 then done := true;
      end
   until (done);

   {  DISARMING SOFTWARE INTERRUPTS  }

   done := false;
   repeat
      begin

         { Request MPE to disarm software interrupts.  }
         plabel := 0;
         fcontrol(msg_file_num,48,plabel);

         if ccode = 1 then  { CCL }
            begin

{  MPE could not disarm software interrupts because an  }
{  I/O was started and never completed.  Abort the I/O. }
               fcontrol(msg_file_num,43,dummy);

               if ccode = 0 then  { CCG }

   { The I/O has progressed too far to abort.  The only }
   { way this can happen is if software interrupts were }
   { disabled and the interrupt postponed.  Re-enable   }
   { interrupts.  The interrupt will occur immediately. }
   { A flag is set to prevent the interrupt handler     }
   { from starting yet another read.                    }
                  begin
                     need_another_read := false;
                     fintstate(-1);
                  end;
            end
         else  {Software interrupts successfully disarmed.}
            done := true;
      end
   until (done);
    { We will never go through this loop more than twice.}

{  Software interrupts are turned off for this file.  We  }
{  could do standard FREADs on the file at this point.    }

   fclose(msg_file_num,0,0);
   if ccode <> 2 then
      begin
         fcheck(msg_file_num,error);
         quit(error);
      end;
end.
```

7

Interprocess Communication Via the MAIL Facility

A number of MPE V/E-based systems use the MAIL facility to direct the communication of information between processes. This information transfer, however, is restricted to upward or downward paths through the process tree structure, so that any process can communicate only with its parent or children. Between any parent/child pair, only one such transfer is allowed at any particular time. The MAIL facility should not be confused with message files, which can be used to transfer information between unrelated processes.

Restrictions

While the MAIL facility is supported on MPE/iX, it is not as effective as other types of IPC, for several reasons:

- Communication can take place only between processes in the same process tree. Communication between processes in different trees is not possible using MAIL.
- Each pair of processes requires a separate "mailbox" to transfer messages between them. This is a less efficient use of system storage than in other forms of IPC.
- Only upward and downward (parent/child) communication is possible. Communication between sibling processes is not.

For these reasons, it is preferable to use other forms of IPC when generating new applications on MPE/iX. Use variables for simple IPC within jobs and sessions; use the file system features to perform IPC between different jobs and sessions in a network.

Definition of Mail

Information transferred between processes is referred to as "mail." It is sent from one process to another through an intermediate storage area called a "mailbox." At any given time, a mailbox can contain only one item of mail (a "message"). For any process, there are two sets of mailboxes:

- The mailbox used for communication between the process and its parent; each process has one of these.
- The set of mailboxes used for communication between the process and its children; each process has one of these mailboxes for each of its children.

Even though there are two sets of mailboxes, there is only one mailbox between any two processes.

Mail Transfer Process

The transfer of mail is based upon a transaction between the sending and receiving processes that involves the following steps:

1. The sending process tests the mailbox to determine its status, that is, whether it is empty, contains a message, or is being used by the receiving process (optional).
2. The sending process transmits the mail to the mailbox. The message transferred is an array of halfwords in the sending process stack, defined by starting location and word count. The smallest message allowed is a single word. MPE/iX automatically performs a bounds check to ensure that the array specified actually falls within the limits of the process stack.
3. The receiving process may test the mailbox to determine its status (optional).
4. If the mailbox contains a message, the receiving process collects this mail. MPE/iX performs another bounds check to validate the address given for the stack of the receiving process. If the mail is not collected before new mail is sent, any additional mail from the sending process will overwrite it.

Testing Mailbox Status

A process can determine the status of the mailbox used by its parent or child with the `MAIL` intrinsic. If the mailbox contains mail that is awaiting collection by this process, the length of the message (in halfwords) is returned to the calling process. This enables the calling process to initialize its stack in preparation for receipt of the message.

For example, to test the status of the mailbox associated with one of its child processes, a process could use the following intrinsic call:

```
   STATCOUNT:=MAIL(CHILDPIN,MCOUNT);
```

`CHILDPIN` contains the Process Identification Number (PIN) of the child process. An integer count signifying the length, in words, of the incoming message will be returned to the word `MCOUNT`. The status returned to `STATCOUNT` will be one of the following values:

Status	**Meaning**
0	The mailbox is empty.
1	The mailbox contains previous outgoing mail from this calling process that has not yet been collected by the destination process.
2	The mailbox contains incoming mail awaiting collection by this calling process. The length of the mail is returned in `MCOUNT`.
3	An error occurred because an invalid `PIN` was specified or a bounds check failed.
4	The mailbox is temporarily inaccessible because other intrinsics are using it to prepare or analyze mail.

Sending Mail

A process sends mail to its parent or child with the `SENDMAIL` intrinsic. If the mailbox for the receiving process contains a message sent previously by the calling process but not collected by the receiving process, the action taken depends on the *waitflag* parameter specified in `SENDMAIL`.

For example, to send mail to its parent, the following intrinsic call could be used:

```
   STAT:=SENDMAIL(0,3,LOCAT,WAITSTAT);
```

The parameters specified are:

pin	0, specifying that the mail is to be sent to the parent process.
count	3, specifying that the length of the message is three halfwords.
location	LOCAT, an unsigned integer array with the address of the buffer in the stack containing the message to be sent.
waitflag	WAITSTAT, a 16-bit unsigned integer by value. If bit (15:1)=0, any mail sent previously will be overwritten. If bit (15:1)=1, the intrinsic will wait until the receiving process collects the previous mail before sending the current mail.

The status returned to `STAT` is one of the following values:

Status	**Meaning**
0	The mail was transmitted successfully. The mailbox contained no previous mail.
1	The mail was transmitted successfully. The mailbox contained previously sent mail that was overwritten by the new mail, or contained previous incoming/outgoing mail that was cleared.
2	The mail was not transmitted successfully because the mailbox contained incoming mail to be collected by the sending process (regardless of the *waitflag* parameter setting). This means that the mailbox currently contains mail sent to the would-be sending process, and the sending process must collect this mail before it can send any itself.
3	An error occurred because an illegal *pin* was specified or a bounds check failed.
4	An illegal wait request would have produced a deadlock. This means that mail has been sent to the would-be sending process, and the sender is waiting for a reply.
5	The request was rejected because the count specified in the *count* parameter exceeded the mailbox size allowed by the system. The maximum size cannot exceed the maximum DST size for the system.
6	The request was rejected because storage resources for the mail data segment were not available.

Receiving (Collecting) Mail

A process collects mail transmitted from its parent or a child with the `RECEIVEMAIL` intrinsic. If the mailbox for the receiving process is empty, the action taken depends on the *waitflag* parameter specified in `RECEIVEMAIL`.

To collect a message from a child process, the following intrinsic call could be used:

```
STAT:=RECEIVEMAIL(CHILDPIN,MDATA,WAITSTAT);
```

The parameters specified are:

pin	`CHILDPIN`, which contains the Process Identification Number of the child process (0 for parent process).
location	`MDATA`, an array in the stack in which the incoming mail will be stored.
waitflag	`WAITSTAT`, a 16-bit unsigned integer value. If bit (15:1)=1, the intrinsic will wait until the incoming mail is ready for collection. If bit (15:1)=0, the intrinsic will return to the calling process immediately.

One of the following status codes is returned to `STAT`:

Status	**Meaning**
0	The mailbox was empty (and `WAITSTAT` bit (15:1)=0).
1	No message was collected because the mailbox contained outgoing mail from the receiving process.
2	The message was collected successfully.
3	An error occurred because of an illegal *pin* or a bounds check failed.
4	The request was rejected because *waitflag* specified that the receiving process should wait for mail if the mailbox is empty, but the other process sharing the mailbox is already suspended, waiting for mail. If both processes were blocked, neither could activate the other, and they could be deadlocked.

Avoiding Deadlocks

Simultaneous use of mail-transmission, process-suspension, and `RIN`-locking intrinsics throughout a process structure can result in a deadlock if the intrinsic calls are not synchronized properly. Be aware of the following:

1. In a multi-process job/session, whenever a process is suspended (through the `SUSPEND` intrinsic, or blocked, or when locking a RIN or receiving mail), MPE/iX does not determine whether all other processes in the process tree are suspended or blocked. Avoid this situation.
2. An attempt by a process to lock a global RIN succeeds only if both of the following conditions are met:
   - No other process within the job/session currently has locked this RIN. A global RIN cannot be used as a local RIN, because deadlock within the same job/session can occur.

- The calling process currently has no other global RIN locked for itself. This could otherwise result in deadlock between two jobs/sessions.

A

Features of Intrinsics for Message Files

Several intrinsics have features that apply specifically to message files. Most of these features are found in `HPFOPEN` and `FCONTROL`, but other intrinsics are used slightly differently for message files.

Intrinsics Not Allowed for Message Files

Certain intrinsics are not allowed for message files. These are:

- `FPOINT`
- `FREADDIR`
- `FREADSEEK`
- `FSPACE`
- `FUPDATE`
- `FWRITEDIR`
- `FDELETE`

The `FSETMODE` intrinsic is permitted, but ignored.

Intrinsics Exclusive to IPC

The following intrinsics are specific to IPC and are not used for any other purpose:

- `FINTSTATE`
- `FINTEXIT`

A Note On Syntax

In some of the following intrinsics, a parameter contains more than one piece of information within a word. When this is the case, data fields are described in the format (*n:m*), where *n* is the first bit of the field and *m* is the number of consecutive bits in the field. For example, the `FOPTION` field for File Type occupies bits (2:3), or bits 2, 3 and 4.

Parameters not discussed in the following descriptions retain their normal value range and defaults. For more information about these intrinsics, refer to the *MPE/iX Intrinsics Reference Manual* (32650-90028).

HPFOPEN

Note The following discussion also applies to the `FOPEN` intrinsic, which is the equivalent within MPE V/E of `HPFOPEN`. The `FOPEN` parameter name follows the `HPFOPEN` parameter name. For more information on `FOPEN`, refer to the *MPE/iX Intrinsics Reference Manual* (32650-90028).

Item #6, record format option; `FOPTION` (8:2)

Message files are always formatted internally as variable-length record files (option=1). If fixed-length records are requested (option=0), the file can appear fixed when it is opened. There is no difference for a writer; for a reader, the portion of the target area that exceeds the record is filled with blanks (for an ASCII file) or zeros (for a binary file). Files accessed in copy mode are always formatted as variable. (See item #17 for more information on copy mode.)

Item #11, access type option; `AOPTION` (12:4)

The type of access allowed for this file is specified by the following values:

0 - READ-only access. A process that has opened a file with this access type is a “reader.” (`FWRITE` calls will not be allowed.)

1 - WRITE-only access. If this is the only current open of the file, any data written in the file prior to the current `HPFOPEN` request is deleted. If this is not the only current open of the file, the file is treated as if Access Type 3 has been specified. A process that has opened a file with this access type is a “writer.” (`FREAD` calls will not be allowed.)

2 - WRITE-SAVE access. The file system sets this to APPEND access for message files.

3 - APPEND-only access. The `FREAD` intrinsic cannot reference this file. Calls to `FWRITE` will add data to the end of the file. A process that has opened a file with this access type is a “writer.”

For a file accessed in copy mode, any access type other than READ-only access (option = 0) is changed to WRITE-only access (option = 1) by the file system.

Item #13, exclusive option; `AOPTION` (8:2)

The values for this option are the same as for any other file. However, the significance of each value is different:

User	**Value**	**Meaning**
EXCLUSIVE	1	One reader, one writer
SEMI	2	One reader, multiple writers
SHARE	3	Multiple readers and writers
Default	0	Same as 1

In copy mode, the value is always 1.

Item #14, multiaccess mode; `AOPTION` (5:2)

This feature permits processes located in different jobs or sessions to open the same file.

0 - No multiaccess. The file system changes this value to 2 to allow global multiaccess.

1 - Only intrajob multiaccess allowed; this is the same as specifying the `MULTI` option in a `:FILE` command.

2 - Interjob multiaccess allowed; this is the same as specifying the `GMULTI` option in a `:FILE` command.

In copy mode, the value is always 0 (no multiaccess).

Item #15, multirecord option; `AOPTION` (11:1)

For message files, the file system sets this bit to 0, except in copy mode. This option is available only if you also select the inhibit buffering option, Item #46.

Item #17, copy mode option; `AOPTION` (3:1)

This feature permits a message file to be treated as a standard sequential file, so that it can be copied by logical record or physical block to another file.

0 - The file is accessed in its own mode; that is, a message (`MSG`) file is treated as a message file.

1 - The file is to be treated as a standard (`STD`), sequential file with variable-length records. For message files, this allows nondestructive reading of an old message file at either the logical record or the physical block level. Only block level access is permitted if the file is being written in message-file format. This prevents incorrectly formatted data from being written to the message file while that file is unprotected. In order to access a message file in copy mode, a process must have EXCLUSIVE access to the file.

Item #22, volume class option, Item #23, volume name option; `DEVICE` field

These fields are relevant only if this is a new file. If you specify volume class field, you must specify a disc; specification of any device other than a disc opens the device, and the file is no longer a message file.

Item #35, filesize option; `FILESIZE` field

For message files, the number of records is rounded up to completely fill the last block and to make the last extent the same size as the other extents. Two additional records are included for the open and close records. Because of spare tracks or remapped tracks, the logical size is usually smaller than the physical size.

Item #44, numbuffers option; NUMBUFFERS field	The number of buffers you wish to allocate to the file. Value must be between 2 and 31; default is 2. IPC needs at least two buffers; if fewer than two are specified, the field is changed to two. Also, IPC never uses more buffers than there are blocks in the file. If more are specified, the lower number is used (except in copy mode).
Item #46, inhibit buffering option; AOPTION (7:1)	For message files, the file system sets this value to 0. Readers may open a message file with NOBUF if they are in copy mode. This option determines whether the reader will be accessing the file record by record or block by block: 0 - read by logical record 1 - read by physical block Writers must open message files with NOBUF if they are in copy mode; they access the file block by block.

FCONTROL

A few control codes deal specifically with message files. Those not mentioned here are invalid when message files are being used.

Code	Param	Description
2		Complete all I/O; ignored in the case of message files.
4	Integer	Set timeout interval. Indicates that a timeout interval is to be applied to a file. In the case of message files, **param** specifies the length of time (in seconds) that a process waits when reading from an empty file or writing to a full one. The timeout remains enabled until it is explicitly cancelled. (Refer to Chapter 4.)
6		Write End-Of-File. For message files, this is used to verify the state of the file by writing the file label and buffer area to disc. This ensures that the message file can survive system crashes. No EOF is written. (Refer to Chapter 4.)
43		Abort NOWAIT I/O. For message files, CCG is returned when an outstanding I/O operation has completed. You must issue an `IOWAIT` call to finish the request. (Refer to Chapter 5.)
45	1 or 0	Enable/disable extended wait. For message files, a **param** value of 1 enables extended wait. This permits a reader to wait on an empty file that is not currently opened by any writer, or a writer to wait on a full file that has no reader. This parameter remains in effect until you issue an `FCONTROL` call with a controlcode of 45 and a **param** value of 0.
46	1 or 0	Enable/disable reading the writer's ID. For message files, a **param** value of 1 enables reading the writer's ID. Each record read has a one-word header. The first 16 bits indicate the type of record with the following codes: 0 - data record 1 - open record 2 - close record The second 16 bits contain the writer's ID number. If the record is a data record, the data follows the header. Open and close records contain no more information. If the **param** value is 0, reading the writer's ID is disabled. Only data is read to the reader's target area. The open and close records are skipped and deleted by the file system when they come to the head of the message queue, and the one-word header is transparent to the reader. (Refer to Chapter 4.)

Code	Param	Description
47	1 or 0	Nondestructive read. If the **param** value is 1, the next `FREAD` by this reader will not delete the record. Subsequent `FREAD` calls are unaffected. If the **param** value is 0, the next `FREAD` by this reader deletes the record. (Refer to Chapter 4.)
48	*plabel*	Arm/disarm software interrupts. The **param** parameter must pass the external label (*plabel*) of your interrupt procedure. If the value of *param* is 0, the interrupt mechanism is disabled for this file. Also, if `AOPTIONS` (4:1) was set to 0, option 48 resets it to 1. Be sure to use `IOWAIT` or `IODONTWAIT` if you use controlcode 48. (Refer to Chapter 6.)

Note Native Mode *plabel*s are 32 bits long, while Compatibility Mode *plabel*s are 16 bits. Therefore, the interrupt handler and the `FCONTROL` call to it must be compiled in the same mode. Arming software interrupts in cross-mode programming can be unpredictable, and you should avoid this whenever possible.

FCHECK

One error message is returned only when using IPC:

```
151 CURRENT RECORD WAS LAST RECORD WRITTEN BEFORE SYSTEM
CRASHED.
```

This error is returned when for some reason (usually a system failure, or running out of disc space), the system was unable to close the file cleanly. (An `FREAD` of the record returns a CCL, and you can then use `FCHECK` to read the message.)

FGETINFO

The values returned for *recsize* and *EOF* differ if `FCONTROL` with a controlcode of 46 is in effect. The *recsize* value indicates the size of your data records, including the 4-byte header. The number of records returned in *EOF* includes open and close records as well as the data records.

Note The *recsize* is supported to ensure backward compatibility with MPE V/E-based systems. If a file's record size exceeds MPE V/E limits, a zero is returned. Refer to the discussion of `FGETINFO` in the *MPE/iX Intrinsics Reference Manual* (32650-90028).

FFILEINFO

Three values for *itemvalue* are specifically for use with IPC:

Item #	**Type**	**Description**
34	I16	The current number of writers.
35	I16	The current number of readers.
49	I16	Software interrupt *plabel*. A value of zero implies that software interrupts are not being used.

B

Sample Programs: WAIT I/O

This appendix contains a group of sample programs illustrating the use of File System features to perform Interprocess Communication. Included are two simple COBOL programs (Examples B-1 and B-2) that show the use of WAIT I/O for interprocess communications. For sample programs showing more complex forms of IPC, refer to Appendixes C and D.

Example B-1.

```
*
*   MSGWRTR
*
*   Compiled with COBOLII.
*
*   This program reads records from a terminal and writes
*   the data to a message file, whose FILE STATUS is displayed.
*   The message file must be built as follows:
*
*   BUILD MSGFILE1;REC=-80,,F,ASCII;DISC=nnn;MSG
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    MSGWRTR.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT WRITE-FILE ASSIGN TO "MSGFILE1"
     STATUS IS MSG-STAT.
*
 DATA DIVISION.
 FILE SECTION.
 FD WRITE-FILE.
 01   OUT-REC                  PIC X(80).
 WORKING-STORAGE SECTION.
 01   TERM-REC.
   02   END-REC                PIC X(2).
   02   REST-REC               PIC X(76).
 01   DONE                     PIC X.
   88 FINISHED                                 VALUE IS "T".
 01   MSG-STAT                 PIC X(2).
*
 PROCEDURE DIVISION.
*
 100-START-OF-PROGRAM.
     OPEN OUTPUT WRITE-FILE.
     DISPLAY MSG-STAT.
     MOVE "F" TO DONE.
     PERFORM 200-GET-LINE UNTIL FINISHED.
     CLOSE WRITE-FILE.
     DISPLAY MSG-STAT.
     STOP RUN.
*
 200-GET-LINE.
     MOVE SPACES TO TERM-REC.
     ACCEPT TERM-REC.
     IF END-REC = "//" THEN
       MOVE "T" TO DONE
     ELSE
       WRITE OUT-REC FROM TERM-REC
       DISPLAY MSG-STAT
       IF MSG-STAT NOT = "00" THEN
*        Error during write or file is full, stop writing.
         MOVE "T" TO DONE.
```

Example B-2.

```
*
*   MSGREADR
*
*   Compiled with COBOLII.
*
*   This program reads records from the message file and processes them.
*   It uses standard wait I/O because no other processing
*   can be done while waiting for the record and wait I/O is simpler to
*   use than no-wait I/O.  Extended wait is used so this program
*   will not get an (EOF) error if the file is empty and the program
*   writing to it terminates.  A 30-second timeout is used so that this
*   program will not wait forever if the writer never comes back.
*
 IDENTIFICATION DIVISION.
 PROGRAM-ID.    MSGREADR.
*
 ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
   SELECT READ-FILE ASSIGN TO "MSGFILE1"
     STATUS IS MSG-STAT.
*
 DATA DIVISION.
 FILE SECTION.
 FD READ-FILE.
 01   IN-REC                  PIC X(80).
 WORKING-STORAGE SECTION.
 01   TERM-REC                PIC X(78).
 01   DONE                    PIC X.
   88 FINISHED                               VALUE IS "T".
   88 NOTFINISHED                              VALUE IS "F".
 01   MSG-STAT                PIC X(2).
 01   PARM                    PIC S9(4) COMP.
*
 PROCEDURE DIVISION.
*
 100-START-OF-PROGRAM.
     OPEN INPUT READ-FILE.
     DISPLAY MSG-STAT.
*
*    Set up extended waits on read-file.
*
*    The read will wait for a record to be written instead of
*    returning an End-Of-File condition.
*
*
     MOVE 1 TO PARM.
     CALL INTRINSIC "FCONTROL" USING READ-FILE 45 PARM.
*
     MOVE "F" TO DONE.
     PERFORM 200-GET-LINE UNTIL FINISHED.
     CLOSE READ-FILE.
     DISPLAY MSG-STAT.
     STOP RUN.
*
 200-GET-LINE.
     MOVE SPACES TO TERM-REC.
*
*    Set up 30-second timeout.  We actually need to set the timeout only
*    once for message files, but we set it here for each read in case
*    message file timeouts are changed to work like terminal timeouts,
*    which are valid only for the next I/O.
*
*    Because extended waits were set up, we will wait forever on an
*    empty message file. However, for esthetic reasons we don't want
*    to wait forever. Neatness counts, so we set the read to fail
*    if no data is in the message file after 30 seconds.
*
     MOVE 30 TO PARM.
     CALL INTRINSIC "FCONTROL" USING READ-FILE 4 PARM.
*
     READ READ-FILE INTO TERM-REC.
     IF MSG-STAT = "00" THEN
       PERFORM 300-WRITEREC
     ELSE
*      Error or End-Of-File on the message file.
       DISPLAY MSG-STAT
       MOVE "T" TO DONE.
*
 300-WRITEREC.
*
```

```
009100*    Process (in this case display) the record received from the
009200*    message file.
009300*
009400     DISPLAY TERM-REC.
```

C

Sample Programs: NOWAIT I/O

This appendix contains two programs (Example C-1 in COBOL, and Example C-2 in FORTRAN) which illustrate the use of NOWAIT I/O in Interprocess Communication. Software interrupts are not used in these examples; see Appendix D for sample programs illustrating their use.

Example C-1.

```
 001000*
 001100*   NOWAITRD
 001200*
 001300*   Compiled with COBOLII.
 001400*
 001500*   This program has a background task that it does in a loop.  After
 001600*   each pass through the loop it checks a message file to see if a
 001700*   special request has been made.  The check for the special request
 001800*   is made only at the completion of a pass through the loop, for two
 001900*   reasons.  First, the time it takes to make a pass through the loop
 002000*   is not too long for the special request to wait to be read.  Second,
 002100*   the special request may require the use of some of data structures
 002200*   used by the background task, and those data structures may be in an
 002300*   inconsistent state part way into a pass through the loop.  The
 002400*   message file is checked for records containing the special requests
 002500*   using NOWAIT FREADs.  Standard (wait) FREADs were not used because
 002600*   they would have caused this program to wait if the message file did
 002700*   not contain any records (and there was another program with this
 002800*   file open for write access), when what we want is to continue doing
 002900*   the background task in the loop.  Software interrupts were not used
 003000*   because they would try to receive a special request anywhere in the
 003100*   loop, and would have added complexity to provide features that this
 003200*   program does not need.
 003300*

003400 IDENTIFICATION DIVISION.
003500 PROGRAM-ID.   NOWAITRD.
003600*
003700 ENVIRONMENT DIVISION.
003800 CONFIGURATION SECTION.
003900 SOURCE-COMPUTER.  HP-SYSTEM.
004000 OBJECT-COMPUTER.  HP-SYSTEM.
004100 SPECIAL-NAMES.
004200   CONDITION-CODE IS CC.
004300*
004400 DATA DIVISION.
```

```
004500 WORKING-STORAGE SECTION.
004600 01   IN-REC           PIC X(80).
004700 01   TERM-REC         PIC X(80).
004800 01   BACK-GROUND-MSG  PIC X(17)     VALUE "BACKGROUND WORK  ".
004900 01   ERROR-MSG        PIC X(17)     VALUE "UNEXPECTED ERROR ".
005000 01   IN-FILE-NAME     PIC X(9)      VALUE "MSGFILE1 ".
005100 01   DONE             PIC X.
005200   88 FINISHED                       VALUE IS "T".
005300   88 NOTFINISHED                    VALUE IS "F".
005400 01   PARM             PIC S9(4) COMP.
005500 01   LNGTH            PIC S9(4) COMP.
005600 01   IN-FILE          PIC S9(4) COMP.
005700 01   LOOP-COUNTER     PIC S9(4) COMP.
005800 01   MSG-FLAG         PIC S9(4) COMP.
005900*
006000 PROCEDURE DIVISION.
006100*
006200 100-START-OF-PROGRAM.
006300     MOVE 0 TO LOOP-COUNTER.
006400     CALL INTRINSIC "FOPEN" USING IN-FILE-NAME %5 %4000
006500                            GIVING IN-FILE.
006600     IF CC NOT = 0 THEN
006700       PERFORM 900-ERROR-CONDITION.
006800*
006900*    Set up extended waits on read-file.
007000*
007100*    This will cause IODONTWAIT to indicate that the record is still
007200*    unavailable rather than returning an End-Of-File error if the
007300*    writer program terminates.
007400*
007500     MOVE 1 TO PARM.
007600     CALL INTRINSIC "FCONTROL" USING IN-FILE 45 PARM.
007700     IF CC NOT = 0 THEN
007800       PERFORM 900-ERROR-CONDITION.
007900*
008000*    Start the first read on the message file.
008100*
008200     PERFORM 600-START-NEXT-READ.
008300*
008400     MOVE "F" TO DONE.
008500     PERFORM 200-PROCESSING-LOOP UNTIL FINISHED.
008600*
008700*    Abort the outstanding read (PARM is ignored) and close the msg file.
008800*
008900     CALL INTRINSIC "FCONTROL" USING IN-FILE 43 PARM.
009000     IF CC NOT = 0 THEN
009100       PERFORM 900-ERROR-CONDITION.
009200     CALL INTRINSIC "FCLOSE" USING IN-FILE 0 0.
009300     IF CC NOT = 0 THEN
009400       PERFORM 900-ERROR-CONDITION.
009500     STOP RUN.
```

```
*
 200-PROCESSING-LOOP.
*
*    Each pass through this loop we do one iteration of the "background
*    task" and then we test to see if a message has come in.
*
     PERFORM 300-BACKGROUND-TASK.
     PERFORM 400-CHECK-FOR-MSG.
*
 300-BACKGROUND-TASK.
*
*    This could be any background processing, but in our case it is
*    just a display.
*
     PERFORM 700-CPU-WASTER 10000 TIMES.
     ADD 1 TO LOOP-COUNTER.
     IF LOOP-COUNTER = 100 THEN
       MOVE "T" TO DONE.
     DISPLAY BACK-GROUND-MSG.
*

 400-CHECK-FOR-MSG.
*
*    Call IODONTWAIT to see if a record has been written to the
*    message file.  Whether or not a message is there, IODONTWAIT
*    will always return immediately.
*
     CALL INTRINSIC "IODONTWAIT" USING IN-FILE IN-REC LNGTH
                                 GIVING MSG-FLAG.
     IF CC NOT = 0 THEN
       PERFORM 900-ERROR-CONDITION.
*
*    MSG-FLAG will be non-zero (= file number) if a message was received.
*    If a message was received, handle it and re-start the next read on
*    the message file.
*
     IF MSG-FLAG NOT = 0 THEN
       PERFORM 500-HANDLE-MSG
       PERFORM 600-START-NEXT-READ.
*
 500-HANDLE-MSG.
*
*    Do any processing that is required to handle the incoming message.
*
     MOVE IN-REC TO TERM-REC.
     DISPLAY TERM-REC.
*
 600-START-NEXT-READ.
*
*    Start the NOWAIT FREAD.  It will be completed by IODONTWAIT.  Note
*    that NOWAIT FREADs on message files do not require Priv Mode.
```

```
014600*
014700     MOVE -80 TO LNGTH.
014800     CALL INTRINSIC "FREAD" USING IN-FILE IN-REC LNGTH.
014900     IF CC NOT = 0 THEN
015000       PERFORM 900-ERROR-CONDITION.
015100*
015200 700-CPU-WASTER.
015300*
015400*    This is here just to burn up time.  Real work should be done here.
015500*    DO NOT put a "CPU waster" like this in a real program.
015600*
015700     MOVE SPACES TO TERM-REC.
015800*
015900 900-ERROR-CONDITION.
016000     DISPLAY ERROR-MSG.
016100     CALL INTRINSIC "PRINTFILEINFO" USING IN-FILE.
016200     STOP RUN.
```

Example C-2.

```
$CONTROL USLINIT
$STANDARD_LEVEL SYSTEM
C
       PROGRAM NOWAITREAD
C
C      Compiled with FORTRAN 77.
C
C      This program reads messages from both a terminal and a message
C      file, and processes them.  When not processing a message, the
C      program just waits for the next message.  This program uses
C      NOWAIT I/O because it allows it to start FREADs on both the
C      terminal and the message file, and then wait in a single
C      "IOWAIT(0,..." statement for whichever FREAD is finished first.
C
       INTEGER*2        fnum,fnumterm,fnuminfile,fnumoutfile
       INTEGER*2        tcount,length,condcode
       LOGICAL          buf(40)
       LOGICAL          eof
       SYSTEM INTRINSIC GETPRIVMODE,GETUSERMODE,FOPEN,FREAD,IOWAIT
       SYSTEM INTRINSIC FWRITE,FCLOSE,PRINTFILEINFO
C
C
C      Priv Mode needed to open the terminal for NOWAIT I/O.  Must
C      also PREP with PM.
C      foption = $STDIN, ascii, old. aoption = no wait I/O, read access.
C
       CALL GETPRIVMODE
       fnumterm = FOPEN(  , 45B, 4000B)
       IF (CCODE() .NE. 0) THEN
          CALL PRINTFILEINFO( fnumterm )
          STOP ' Error occured during terminal FOPEN '
       END IF
```

```
      CALL GETUSERMODE
C
C     Open the input message file.
C     foption = ascii, old.   aoption = no wait I/O, read access.
C
      fnuminfile = FOPEN( "MSGFILE1", 5B, 4000B)
      IF (CCODE() .NE. 0) THEN
         CALL PRINTFILEINFO( fnuminfile )
         STOP ' Error occured during input message file FOPEN '
      END IF
C
C     Open the output message file.
C     foption = ascii, old.   aoption = write access.
C
      fnumoutfile = FOPEN( "MSGFILE2", 5B, 1B)
      IF (CCODE() .NE. 0) THEN
         CALL PRINTFILEINFO( fnumoutfile )
         STOP ' Error occured during output message file FOPEN '
      END IF
C
C     Start the read on the terminal.  No Wait FREADs always return
C     a length of 0.  The real data length is returned by IOWAIT.
C
      tcount = -80
      length = FREAD( fnumterm, ibuf, tcount)
      IF (CCODE() .NE. 0) THEN
         CALL PRINTFILEINFO( fnumterm )
         STOP ' Error occured during the terminal FREAD '
      END IF
C
C     Start the read on the message file.  No Wait FREADs always return
C     a length of 0.  The real data length is returned by IOWAIT.
C
      tcount =  -80
      length = FREAD( fnuminfile, buf, tcount)
      IF (CCODE() .NE. 0) THEN
         CALL PRINTFILEINFO( fnuminfile )
         STOP ' Error occured during the message file FREAD '
      END IF
C
      eof = .FALSE.
      DO WHILE (.NOT. eof)
C
C        An IOWAIT with file-number = 0 will complete whichever
C        FREAD is ready to be finished first.
C
         fnum = IOWAIT( 0, buf, tcount)
         condcode = CCODE()
         IF (condcode .EQ. -1) THEN    ! Error
            CALL PRINTFILEINFO( fnum )
```

```
             STOP ' Error occured during IOWAIT '
          END IF
          IF (condcode .EQ. 1) THEN     ! EOF
             eof = .TRUE.
          END IF
          IF (condcode .EQ. 0) THEN
C
C            Process the message that came in;  in this case write it to
C            "message file 2".  FREAD and FWRITE want byte lengths to be
C            negative.  IOWAIT returns a positive byte length.  We have
C            to change the sign.
C
             tcount = - tcount
             CALL FWRITE( fnumoutfile, buf, tcount, 0)
             IF (CCODE() .NE. 0) THEN
                CALL PRINTFILEINFO( fnumoutfile )
                STOP ' Error occured during FWRITE '
             END IF

C
C            Re-start the FREAD that the IOWAIT just completed.
C
             tcount = -80
             length = FREAD( fnum, buf, tcount)
             IF (CCODE() .NE. 0) THEN
                CALL PRINTFILEINFO( fnum )
                STOP ' Error occured during FREAD '
             END IF
          END IF
       ENDDO
C
C      Time to shut down.
C      Call FCONTROL-43 to abort the no wait read that is pending
C      against the terminal and the message file.  If CCG is returned,
C      then the abort could not complete, and an IOWAIT must be called
C      to clear the I/O.  CCE means aborted OK.  CCG means nothing to
C      abort, which is OK here because the read is not restarted if
C      there was an error.
C
       CALL FCONTROL( fnumterm, 43, 0)
       IF (CCODE() .EQ. 1) THEN
          fnum = IOWAIT( fnumterm, buf, tcount)
          IF (CCODE() .NE. 0) THEN
             CALL PRINTFILEINFO( fnumterm )
             STOP ' Error occured during terminal FCONTROL/IOWAIT '
          END IF
       END IF
       CALL FCONTROL( fnuminfile, 43, 0)
       IF (CCODE() .EQ. 1) THEN
          fnum = IOWAIT( fnuminfile, buf, tcount)
          IF (CCODE() .NE. 0) THEN
```

```
            CALL PRINTFILEINFO( fnuminfile )
            STOP ' Error occured during message file FCONTROL/IOWAIT '
         END IF
      END IF
C
      CALL FCLOSE( fnumterm, 0, 0)
      CALL FCLOSE( fnuminfile, 0, 0)
      CALL FCLOSE( fnumoutfile, 0, 0)
C
      STOP 'Successful'
      END
```

D

Sample Programs: Software Interrupts

This appendix contains a group of sample programs illustrating a fairly complex, but very typical, use of message files and soft interrupts. There are two groups of programs, one in SPL and another in Pascal.

SPL Examples

Examples D-1 through D-3 are a group of SPL programs showing the use of software interrupts and message files in Interprocess Communication. A group of sample Pascal programs follows.

Example D-1.

```
begin
<<
   MON

   This program opens message file MONTOLOG and writes a data record
   every 2 seconds. The data record is the dateline plus the 2-word
   return from the CLOCK intrinsic.
>>
   integer i,file'num,error;
   double timer;
   integer array timer1(*)=timer;
   logical array outdata(0:16);
   integer sequence;
   byte array file'name(0:9):="MONTOLOG  ";
   byte array msg1(0:9):="Opened OK ";
   real pause'val;
   intrinsic print,dateline,pause,fopen,fwrite,quit,fcheck,clock;

<<
   Open message file.
>>
   file'num:=fopen(file'name,5,193);
   if <> then
      begin
         fcheck(file'num,error);
         quit(error);
      end
   else print(msg1,-10,0);
   pause'val:=2.0;
   for i:=1 until 100 do
```

```
      begin
         sequence:=i;
         timer:=clock;
         dateline(outdata);
         outdata(14):=sequence;
         outdata(15):=timer1(0);
         outdata(16):=timer1(1);
         fwrite(file'num,outdata,-34,0);
         if <> then
            begin
               fcheck(file'num,error);
               quit(error);
            end;
         pause(pause'val);
      end;
end.
```

Example D-2.

```
$control uslinit
<<

   LOG

   SPL program to demonstrate soft interrupts in a process
   handling environment.

   This program opens $STDIN nowait. It opens a message file
   (MONTOLOG) and enables soft interrupts. It then opens a log
   file (LOGFILE) to write progress messages from the interrupt
   handler.
>>

begin
   integer i,lth,addr,parm,error,file'num'0:=0;
   integer log'file'num,file'num,file'num'1;
   integer plabel;
   logical array buff(0:39),buff1(0:39);
   byte array file'name(0:9):="MONTOLOG  ";
   byte array file'name'1(0:9):="TERM      ";
   byte array log'file'name(0:9):="LOGFILE   ";
   logical array stat1(0:8):="Opened terminal OK";
   logical array stat2(0:8):="Opened MSGFILE OK ";
   logical array stat3(0:8):="Opened LOGFILE OK ";

   intrinsic dateline,pause,fopen,quit,fcheck,getprivmode,
             fintexit,fintstate,getusermode,iowait,iodontwait,
             fread,fcontrol,clock,ascii,print,fclose,fwrite;

   procedure getasc(num,temp);
      BYTE ARRAY num;
      INTEGER temp;
      begin
         integer lth;

         lth:=ascii(temp,10,num);
         if lth=1 then
            begin
               num(1):=num(0);
               num(0):="0";
            end;
      end; <<getasc>>

<<
    INTERRUPT HANDLER ROUTINE

      When an interrupt occurs on the message file (filenum), we
      get the time of the interrupt using the CLOCK intrinsic.
      The time the record was written into the message file
      is written as part of the record in the message file.  We
      extract that time as well and write both times into our
      log file.
>>
   procedure inthandler(filenum);
      VALUE filenum;
      integer filenum;

      begin
      logical array outdata(0:16);
<<
   Since we must extract the bytes for the CLOCK intrinsic,
   we must equivalence an integer to the double integer returned
   by CLOCK. Could have used a byte array here, just didn't choose
   to.
>>
      double timestamp;
      integer array time1(*)=timestamp;

      integer temp;
      byte array msg1(0:29),msg2(0:29),num(0:5);


<<
      Get time of interrupt.
>>
      timestamp:=clock;
<<
      Complete I/O from message file.
>>
      iodontwait(filenum,outdata);
      move msg1:="Time file written=   :  :  :   ";
      move msg2:="Time interrupted =   :  :  :   ";
<<
```

```
        In this section, we convert the proper clock bits to times
        which can be printed as part of the above messages. Long
        and involved, but that's a problem with SPL.
  >>
  <<
        First we must extract the timestamp from the message file
        record.
  >>

  <<
        Extract hour.
  >>
              temp:=outdata(15).(0:8);
              getasc(num,temp);
              move msg1(19):=num,(2);
  <<
        Extract minute.
  >>
              temp:=outdata(15).(8:8);
              getasc(num,temp);
              move msg1(22):=num,(2);
  <<
        Extract second.
  >>
              temp:=outdata(16).(0:8);
              getasc(num,temp);
              move msg1(25):=num,(2);
  <<
        Extract tenth-of-second.
  >>
              temp:=outdata(16).(8:8);
              getasc(num,temp);
              move msg1(28):=num,(2);
  <<
     END of processing for TIME FILE WRITTEN message.
  >>
        fwrite(log'file'num,msg1,-30,0);
        fwrite(log'file'num,outdata,-28,0);
  <<
     Begin processing TIME OF INTERRUPT message.
  >>
  <<
        Extract hour.
  >>
              temp:=time1(0).(0:8);
              getasc(num,temp);
              move msg2(19):=num,(2);
  <<
        Extract minute.
  >>
              temp:=time1(0).(8:8);
```

```
            getasc(num,temp);
            move msg2(22):=num,(2);
<<
      Extract second.
>>
            temp:=time1(1).(0:8);
            getasc(num,temp);
            move msg2(25):=num,(2);

<<
      Extract tenth-of-second.
>>
            temp:=time1(1).(8:8);
            getasc(num,temp);
            move msg2(28):=num,(2);
<<
   END of processing for TIME OF INTERRUPT message.
>>
      fwrite(log'file'num,msg2,-30,0);
<<
   Re-post the message file read.
>>
      fread(filenum,buff,-34);
<<
   Re-enable interrupts when the handler routine exits.
   Same effect as FINTEXIT(TRUE).
>>
      fintexit;
      end;

<<
   MAINLINE
>>
   fintstate(1);
<<
   OPEN $STDIN with NOWAIT I/O.
>>
   getprivmode;
   file'num'1:=fopen(file'name'1,36,2048);
   if <> then
      begin
         getusermode;
         fcheck(file'num'1,error);
         quit(error);
      end
      else print(stat1,-18,0);
   getusermode;
<<
   OPEN MONTOLOG as a message file.
>>
   file'num:=fopen(file'name,5,192);
```

```
      if<> then
         begin
            fcheck(file'num,error);
            quit(error);
         end
         else print(stat2,-18,0);
   <<
      OPEN LOGFILE as an OLD ASCII file.

      LOGFILE is VERY IMPORTANT.  If the messages in the interrupt
      routine are posted to the terminal (via PRINT), then we will
      have to wait for the FREAD on the terminal to complete before
      the write will complete.  The IOWAIT can be interrupted when done
      as shown below, but the PRINT cannot be interrupted!  This will
      keep us from processing further interrupts until the status
      messages from the first interrupt have been completed.

      By writing to a file, we avoid this problem.
   >>
      log'file'num:=fopen(log'file'name,5,1);
         if <> then
            begin
               fcheck(log'file'num,error);
               quit(error);
            end
         else print(stat3,-18,0);
   <<
      Set up FCONTROL to enable SOFT INTERRUPTS.
   >>
      plabel:=@inthandler;
      fcontrol(file'num,48,plabel);
   <<
      Set up EXTENDED WAIT. If we don't do this, the read of the
      message file will interrupt with FSERR 0 when empty.
   >>
      parm:=1;
      fcontrol(file'num,45,parm);
   <<
      Post read against message file.
   >>
      fread(file'num,buff,-34);
      for i:=1 until 100 do
         begin
            fread(file'num'1,buff1,-80);
   <<
      The only way we will interrupt an IOWAIT is if we post a GENERAL
      IOWAIT. In other words, we use a 0 for the file number.

      In our case here, we don't care to get the return since we know
      which file we posted a read on.
```

```
>>
         iowait(0,,lth);
         print(buff1,-80,0);
      end;
   fclose(log'file'num,1,0);
end.
```

Example D-3.

```
$control uslinit
<<
   PARENT

   This program is the parent process for MON and LOG.

>>
begin
   integer array nums(0:5),items(0:5);
   integer pin,pin1,error,error1;
   byte array prog'1(0:9):="MON       ";
   byte array prog'2(0:9):="LOG       ";
   logical array msg1(0:9):="Process MON created ";
   intrinsic print,createprocess,quit,activate;

   nums(0):=3;
   items(0):=1;
   nums(1):=0;
   items(1):=0;
   createprocess(error,pin,prog'1,nums,items);
   if error <>0 then quit(error);
   activate(pin);
   print(msg1,-20,0);
   nums(1):=10;
   items(1):=2;
   nums(2):=0;
   items(2):=0;
   createprocess(error1,pin1,prog'2,nums,items);
   if error1<>0 then quit(error1);
end.
```

Pascal Examples

The following group of programs (Examples D-4 through D-6) show the use of message files and soft interrupts for Interprocess Communication. For a simpler example, see the sample Pascal program in Chapter 6.

Example D-4.

```
$uslinit$
program mon(input,output);
{
   MON

   This program opens message file MONTOLOG and writes a data record
   every 2 seconds. The data record is the dateline plus the 2-word
   return from the CLOCK intrinsic.
}

type
   int = -32768..32767;
   timer = packed record
         hour : 0..255;
         min : 0..255;
         sec : 0..255;
         tenth : 0..255;
      end;
   data_rec = record
            dataline : packed array [1..28] of char;
            sequence : int;
            time : timer;
          end;

var
   I : int;
   outdata : data_rec;
   pause_val : real;
   file_num : int;
   file_name : packed array [1..10] of char;
   error : int;

procedure dateline; intrinsic;
procedure pause; intrinsic;
function fopen:int; intrinsic;
procedure fwrite;intrinsic;
procedure quit; intrinsic;
procedure fcheck;intrinsic;
function clock:timer;external;
begin
{
   Open message file.
}
   file_name:='montolog  ';
   file_num := fopen(file_name,5,193);
   if ccode <> 2 then
      begin
         fcheck(file_num,error);
         quit(error);
      end
      else writeln('Opened OK');
   pause_val := 2.0;
   for i:=1 to 100 do
      begin
         outdata.sequence := i;
         dateline(outdata.dataline);
         outdata.time:=clock;
         fwrite(file_num,outdata,-34,0);
         if ccode<>2 then
            begin
               fcheck(file_num,error);
               quit(error);
            end;
         pause(pause_val);
      end;
end.
```

Example D-5.

```
$uslinit$
$standard_level 'HP3000'$
{
   LOG

   Pascal program to demonstrate soft interrupts in a process
   handling environment.

   This program opens $STDIN nowait.  It opens a message file
   (MONTOLOG) and enables soft interrupts.  It then opens a log
   file (LOGFILE) to write progress messages from the interrupt
   handler.
}
program log(input,output);

type
   int = -32768..32767;

{
   TIMER is a record type which corresponds to the return from
   the CLOCK intrinsic.
}
   timer = packed record
         hour : 0..255;
         min : 0..255;
         sec : 0..255;
         tenth : 0..255;
```

```
      end;

{
   DATA_REC is a record type which corresponds to the record
   written by the MON program.
}
   data_rec = record
            dataline : packed array [1..28] of char;
            sequence : int;
            time : timer;
          end;

var
   fle : text;
   i,lth,addr,parm : int;
   file_num_0,file_num,file_num_1 : int;
   logfile,file_name,file_name_1 : packed array [1..10] of char;
   buff,buff1 : packed array [1..80] of char;
   error : int;

procedure dateline; intrinsic;
function fopen:int; intrinsic;
procedure fwrite;intrinsic;
procedure quit; intrinsic;
procedure fcheck;intrinsic;
procedure getprivmode;intrinsic;
procedure fintexit;intrinsic;
procedure fintstate;intrinsic;
procedure getusermode;intrinsic;
procedure iowait;intrinsic;
procedure iodontwait;intrinsic;
procedure fread;intrinsic;
procedure fcontrol;intrinsic;
function clock : timer; external;
{
   INTERRUPT HANDLER ROUTINE

   When an interrupt occurs on the message file (filenum), we
   get the time of the interrupt using the CLOCK intrinsic.
   The time the record was written into the message file
   is written as part of the record in the message file.  We
   extract that time as well and write both times into our
   log file.
}
procedure inthandler(filenum:int);
   var
      timestamp : timer;
      buffer : data_rec;
   begin
{
      Get time of interrupt.
```

```
}
      timestamp:=clock;
{
      Complete I/O from message file.
}
      iodontwait(filenum,buffer);
      writeln(fle,buffer.dataline,'  ',buffer.sequence);
      writeln(fle,'Time file written= ',buffer.time.hour,':',
              buffer.time.min,':',buffer.time.sec,':',
              buffer.time.tenth);
      writeln(fle,'Time interrupted= ',timestamp.hour,':',
              timestamp.min,':',timestamp.sec,':',
              timestamp.tenth);
{
      Restart the message file read.
}
      fread(filenum,buff,-34);

{
      Re-enable interrupts when the handler routine exits. Same
      as FINTEXIT(1).  Interesting point--Pascal Boolean values
      are different from SPL logical values.  Pascal TRUE <> SPL TRUE.
}
      fintexit;
   end;

begin
{
   Open LOGFILE as text file.

   LOGFILE is VERY IMPORTANT.  Soft interrupts may interrupt a
   GENERAL NOWAIT I/O as posted below.  However, they may not
   interrupt any other I/O in progress.  If we WRITELN to the
   terminal in the interrupt handler routine, that WRITELN will
   have to wait for the read I/O on the terminal to complete.  Since
   that wait is in the interrupt handler routine, it can't be
   further interrupted.  The bottom line here is that interrupts
   will not be processed except after the terminal I/O completes
   and before a new I/O is started.

   To avoid this, we write to LOGFILE.
}
   logfile:='LOGFILE   ';
   rewrite(fle,logfile,'NOCCTL');
{
   Set up interrupts for this program.

   Again, as stated above in the interrupt handler routine, SPL
   TRUE (which is how the intrinsic is defined) = %000001.  Pascal
   TRUE, for a Boolean variable, is %000400.  Therefore, we cannot
   use a Boolean variable here.
```

```
}
   fintstate(1);
{
   Open the terminal for nowait I/O.
}
   file_name_1 := 'TERM       ';
   getprivmode;
   file_num_1 := fopen(file_name_1,36,2048);
   if ccode <> 2 then
      begin
         getusermode;
         fcheck(file_num_1,error);
         quit(error);
      end
      else writeln('Opened terminal OK');
   getusermode;

{
   Open the message file.
}
   file_name:='montolog  ';
   file_num := fopen(file_name,5,192);
   if ccode <> 2 then
      begin
         fcheck(file_num,error);
         quit(error);
      end
      else writeln('Opened MSGFILE OK');
{
   Set up soft interrupts.
}
   addr := waddress(inthandler);
   fcontrol(file_num,48,addr);
{
   Set up EXTENDED WAIT.  If we don't do this, the read of the
   message file will continually interrupt with FSERR 0 when empty.
}
   parm := 1;
   fcontrol(file_num,45,parm);
{
   Post read against message file.
}
   fread(file_num,buff,-34);
   for i := 1 to 100 do
      begin
         fread(file_num_1,buff1,-80);
{
      The only way we will interrupt an IOWAIT is if we post a
      GENERAL IOWAIT (using 0 for the file number).

      In this case, we don't care to get the return since we know
      which file had a read posted against it.
}
         file_num_0:=0;
         iowait(file_num_0,buff1,lth);
      end;
   close(fle,'SAVE');
end.
```

Example D-6.

```
$uslinit$
{
   PARENT

   This program is the parent process for MON and LOG.
}
program daddy(input,output);
type
   int = -32768..32767;
var
   prog_1,prog_2 : packed array[1..10] of char;
   pin,pin1,error,error1 : int;
   nums,items : array[1..5] of int;

procedure createprocess;intrinsic;
procedure quit;intrinsic;
procedure activate;intrinsic;

begin
   prog_1 := 'mon       ';
   prog_2 := 'log       ';
   nums[1]:=3;
   items[1]:=1;
   nums[2]:=0;
   items[2]:=0;
   createprocess(error,pin,prog_1,nums,items);
   if error<>0 then quit(error);
   activate(pin);
   writeln('Process MON created');
   nums[2]:=10;
   items[2]:=2;
   nums[3]:=0;
   items[3]:=0;
   createprocess(error1,pin1,prog_2,nums,items);
   if error<>0 then quit(error);
end.
```

Index