TEXAS INSTRUMENTS

9900

Device Independent File I/O

MICROPROCESSOR SERIES™

IMPORTANT NOTICES

Texas Instruments reserves the right to make changes at any time to improve design and to supply the best possible product for the spectrum of users.

Information contained in these publications is believed to be accurate and reliable. However, no responsibility is assumed for its use. No license is granted by implication or otherwise under any patent or patent right of Texas Instruments or others.

TABLE OF CONTENTS

SECTION I. OVERVIEW

SECTION II. DEVICE INDEPENDENT FILE I/O SUBSYSTEM STANDARDS

SECTION III. FILE I/O DECODER ROUTINES

SECTION IV: THE INTERPROCESS COMMUNICATION SUBSYSTEM

SECTION V: ENCODE AND DECODE ROUTINES

SECTION VI: CONFIGURING AN APPLICATION TO INCLUDE DIF I/O ROUTINES

APPENDICES

LIST OF FIGURES

LIST OF TABLES

PREFACE

This manual documents the user procedures for the 9900 Device Independent File I/O Package. The manual is organized as follows:

Section I provides a product overview of the Device Independent File I/O Package.

Section II provides an explanation of the concepts implemented in this package.

Section III describes the File I/O Decoder, lists the entry points into the decoder, and presents a Pascal and assembly language calling sequence for each.

Section IV describes the Interprocess Communciation (IPC) I/O Subsystem, defining the routines that comprise this subsystem and presenting Pascal and assembly language calling sequences for these routines.

Section V describes the Encode and Decode Routines included in this package. Each routine is documented along with its assembly language calling sequence.

Section VI describes configuration of applications containing DIF I/O routines. Initialization as well as configuration information is presented.

Appendix A describes the Operator Interface (T02) I/O Subsystem demonstrating for the Pascal user how to write his own I/O Subsystem.

Appendix B provides pictures of data structures used during system initialization of DIF I/O applications.

Appendix C defines the status codes returned by the File I/O Decoder.

Appendix D defines the Dummy I/O Subsystem included in this package.

Appendix E describes the file attributes that are defined during access of File I/O Decoder routines.

Appendix F presents listings of global declaration files for use by the Pascal user in working with this package.

The following publications offer informational support to this document and to users of this product.

MP375	AMPLUS Software System User's Manual
MP357	Microprocessor Pascal System User's Manual
MP385	Microprocessor Pascal Executive User's Manual
946250-9701 thru 9706	Model 990 Computer DX10 Operating System Reference Manual
949617-9701	Model 990 Computer Link Editor Reference Manual
943441-9701	Model 990 Computer TMS9900 Microprocessor Assembly Language Programmer's Guide
MP373	Realtime Executive User's Manual

SECTION I

OVERVIEW

1.1 GENERAL

Texas Instruments Device Independent File (DIF) I/O User's Manual documents a collection of routines providing I/O services for the 9900 family of microprocessors. This software enables Pascal and assembly language users to perform device independent I/O utilizing a consistent file interface. Through the use of these routines, requests for file services are automatically translated into calls to individual subsystems dedicated to managing data resident on various devices such as CRTs, card readers, printers, etc.

The DIF I/O package is especially useful when the performance of input and output operations is required on any of several devices and the specific device is not identifiable until run-time. Use of the DIF I/O package also enables the user to access a variety of software subsystems using a single interface.

In addition to documenting software supplied by Texas Instruments, this manual presents standards that a user must follow to produce his own subsystems with the capability to interface with this package and be accessed in a device independent manner.

1.2 SOFTWARE

The software routines described in the Device Independent File I/O Package User's Manual are listed below. The Pascal versions of these routines are provided in the Microprocesor Pascal Executive contained in the Microprocessor Pascal System (TMSW753P and 754P). For the assembly language user, these routines are provided in the Device Independent File I/O Package (TMSW360D)

- The File I/O Decoder providing a device-independent interface between the user and a variety of I/O devices (floppy diskette, tape reader, printer, CRT, etc.).
- The Interprocess Communication (IPC) I/O Subsystem implementing file-level message passing between processes.
- The Operator Interface I/O Subsystem (T02) implementing data communication with terminals connected to a 9902 asynchronous communications controller.

- Encode and Decode routines performing data conversion from the internal representation to printable format and conversely from printable format to internal representation. (These routines are used transparently in the Microprocessor Pascal Executive. However, the assembly language user calls these routines directly.)

1.3 HOST AND TARGET SYSTEMS

Development capabilities for applications using DIF I/O routines are provided by the following host systems:

- Single-User FS990/4 or /10 floppy disk minicomputer with operating system software provided by the AMPLUS Software System and development system software provided by AMPLUS or the Microprocessor Pascal System.

- Multi-user DS990/10 or /12 hard disc minicomputer with operating system software provided by the DX Operating System and development system software provided by the DX system or the Microprocessor Pascal System.

Target systems include TMS9900 and TM990 Microprocessor systems.

Device independent routines and data structures support I/O in the target system. I/O in the host system is provided for in the host system's operating system.

1.4 DIF I/O PACKAGE AS A COMPONENT

The DIF I/O is a member of Texas Instruments' 9900 series of component software. This series contains a variety of individual software products that can be separately purchased and combined with an application to produce a powerful software product. Because of the modularity of the routines comprising this product (as well as other Texas Instrument component software), the load module produced to run on a target will include only those "pieces" of the component that are required by the application. In this way, memory requirements are kept to the minimum.

The Realtime Executive (see Subsection 1.5 below) acts as a software link for an application written in one of several languages and utilizing various "components". Figure 1-1 pictures this concept. The Device Independent File I/O Package component is present along with two I/O Subsystem components.

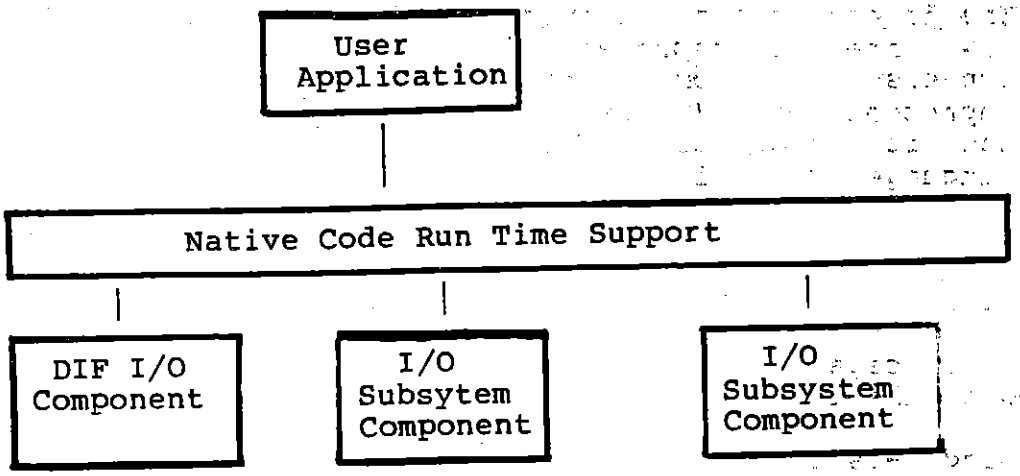


FIGURE 1-1. COMPONENT SOFTWARE REPRESENTATION

1.5 REQUIRED EXECUTIVE ENVIRONMENT

Execution of the routines described in this document requires native code run-time support of an executive. (The executive provides control of the software execution in a computer system including control of CPU usage, memory usage, routine calling conventions, data structures, etc.) Run-time support is provided for the assembly language programmer by Texas Instruments' Realtime Executive (TMSW330R) and for the Pascal user by the Microprocessor Pascal Executive (contained in the Microprocessor Pascal System - TMSW753P and 754P). These executives are described in the the Realtime Executive User's Manual (MP373) and in the Microprocessor Pascal Executive User's Manual (MP385) respectively.

SECTION II

DEVICE INDEPENDENT FILE I/O SUBSYSTEM STANDARDS

2.1 GENERAL

This section of the manual discusses the concept of device independent I/O as it is implemented by this software and the standards which, when applied, allow the user to write his own subsystem to interface with this software.

2.2 TERMINOLOGY

Device Independent I/O refers to a mode of implementing input/output requests on target devices without naming the specific target device in the procedure call (a requirement when the appropriate device is not identified until run time). As listed in Subsection 1.2, the routines translating device independent I/O requests into calls for a particular device operation are contained in the File I/O Decoder. A subsystem accessible to the File I/O Decoder and performing target I/O operations on the device is an I/O Subsystem. The logical connection between the CPU and the physical means of controlling a device is called a Port. In most I/O Subsystems, the port identifies a particular device controller such as a TM990/303 Floppy Disk Controller. Each device controlled on the target is called a Node. The Node associated with the Floppy Disk Controller is a floppy disk. I/O operations are executed sequentially in the order in which they are received. If, when a command is received, the calling process is suspended until all previously issued commands are completed, then Executed I/O is performed. Initiated I/O refers to instances when the calling process is reactivated before the command is completed.

2.3 I/O SUBSYSTEM RATIONALE

The I/O Subsytem Standards provide for the standardization of an I/O interface in the user application. This standarization enables the user application to:

1) Realize a general file interface for I/O requests to various devices supported on the target system. Devices are treated as files. Requests for I/O services on the device are made as requests for file services. In this way, the Pascal user can use Pascal READLN and WRITELN statements to perform device I/O. The assembly language user can access the File I/O Decoder directly to perform device I/O.

2) Implement initiated I/O, a means of servicing I/O requests without blocking the requesting process.

3) Activate multiple instances of the Interface Handler (I/F Handler) manipulating the device on the target.

4) Add various Interface Handlers and establish the means to communicate with them. The user can commuinicate with as many devices (of varying types) as he requires.

5) Access I/O service routines at varying levels of logical orgainzation (from the logical file level to the physical device level).

2.4 I/O MODEL

An "I/O Model" adhering to the requirements of the I/O Subsystem standards presented in this document is pictured in Figure 2-1 below. This model is comprised of the following components:

- The File I/O Decoder translating user I/O requests to procedure calls for specific device services.
- Various I/O subsystems, each implementing the procedure calls from the File I/O Decoder for its specific device. An I/O Subsystem may be one of the component software packages that can be obtained from Texas Instruments (the File Manager and the Operator Interface I/O Subsystem are two examples), or can be written by the user to conform to the standards presented in this document.
- Channels created to handle the communication from the I/O Subsystem to the Interface Handler software.
- The Interface Handler manipulating the device.

The components described above are ordered from a logical to a physical interface. This ordering traces the flow of program control when device independent I/O is performed. The application generates I/O requests by invoking the File I/O Decoder . The Decoder selects the appropriate I/O Subsystem (File Manager, Operator Interface I/O packages, etc.) and passes control to that subsystem. The subsystem selects the appropriate device (floppy, printer, etc.) passing control to the associated Interface Handler. Movement of control between the levels is transparent (invisible) to the user.

The user may choose the level at which he requires the request to be executed. He may call the File I/O Decoder, the individual I/O Subsystem, or even the appropriate interface handler from his application. However, it is only via the File I/O Decoder that calls to I/O routines can be performed without regard to a specific target device.

Entering the I/O model below the File I/O Decoder Level (i.e., at the I/O Subsystem Level, or at the Interface Handler level) requires the

user to understand the requirements associated with that lower level and to structure his code to meet those requirements.

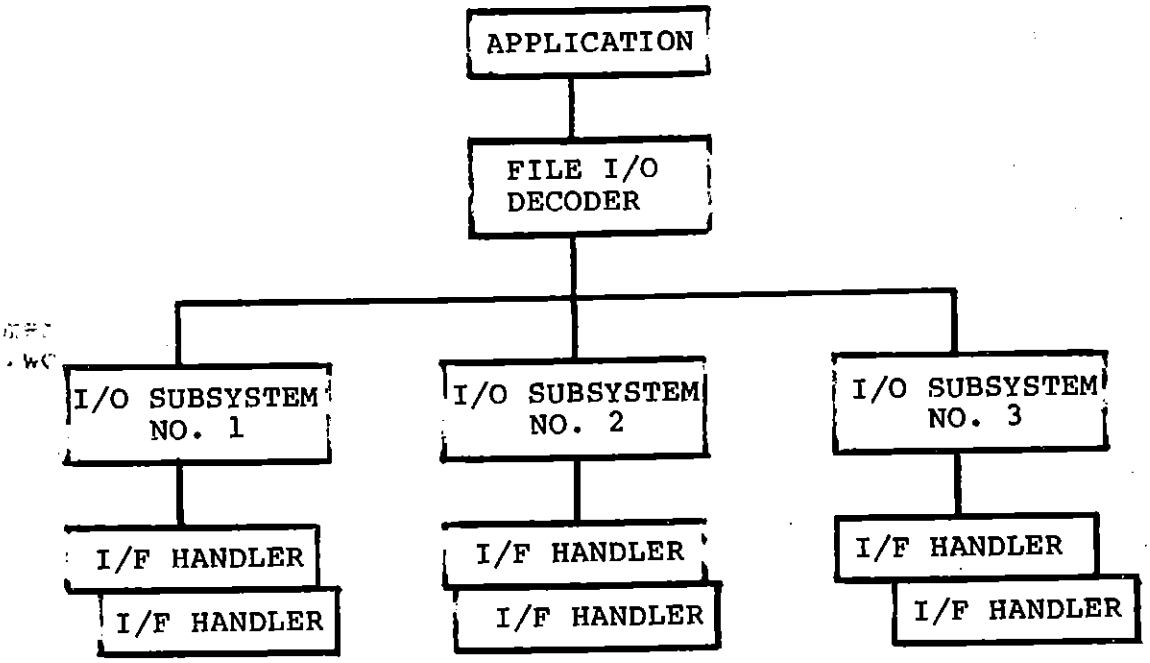


FIGURE 2-1. I/O MODEL

NOTE: The above figure depicts more than one Interface Handler attached to each I/O Subsystem. In actuality, an I/O Subsystem can start a single Interface Handler several times to service different devices of the same type (e.g., the File Manager interfacing with several floppy disk controllers) or different Interface Handlers attached to different devices (e.g., the File Manager interfacing with a floppy disk controller and to a bubble memory controller).

The remainder of this subsection describes each of the above components in detail.

2.4.1 FILE I/O DECODER

The File I/O Decoder provides the software capability for device independent I/O. The Pascal programmer uses such Pascal statements as SETNAME, RESET, REWRITE, READ, and WRITE to perform file I/O. These "Pascal primitives" in turn automatically access the File I/O Decoder without any further user interface. The assembly language user application accesses the File I/O Decoder directly via file operation entry points (the Pascal user can also access these entry points directly). Each entry point corresponds to a device operation supported on the target and is callable as a separate file service

routine. The Pascal and assembly language calling sequences for the File I/O Decoder routines are presented in Section III. Among the entry points (callable routines) present are:

- D$INIT — Initializes the File I/O Decoder and supported I/O Subsystems.
- D$CONNECT — Connects a file pathname with an associated I/O Subsystem and ultimately the physical device or node controlled by that subsystem.
- D$CREATE — Creates a file.
- D$OPEN — Opens a file for access.
- D$READ — Initiates reading of a file.
- D$WRITE — Initiates writing to (or updating) a file.
- D$RDWAIT — Reads a file; but delays return of control to calling routine until read operation is completed.
- D$WRWAIT — Writes a file, but delays return of control to calling routine until write operation is completed.
- D$POSITION — Resets the internal subsystem's pointer to point at the requested record.
- D$STATUS — Checks for status of oldest I/O request on the specified file at the File I/O Decoder Level.
- D$DSTATUS — Checks for status of oldest I/O request on the specified file at the subsystem level.
- D$ABORTIO — Aborts all I/O requests outstanding on a file.
- D$CLOSE — Closes a file disallowing further access until re-opened.
- D$DELETE — Deletes a file.
- D$DISCONNECT — Severs the connection between a file pathname and physical device as controlled by an I/O Subsystem.

The operations requested via these entry points are viewed by the user on the File I/O Decoder level. At the point that these routines are called, the user need not associate the requested operation with a particular device; the File I/O Decoder makes that association for him. In reality, each file service accesses an entry point into a device dependent service managed by the appropriate I/O Subsystem.

Two entry points in the File I/O Decoder bear special mentioning: D$CONNECT and D$DISCONNECT.

D$CONNECT must be the first File I/O Decoder service requested (after initialization) and is called to connect a specified file pathname (passed to D$CONNECT as a calling parameter) with the appropriate I/O Subsystem (and ultimately the device itself). At "connect" time, the File I/O Decoder will invoke each subsystem supported on the target in succession until some subsystem recognizes the pathname parameter passed with the call. Such recognition is made because of the meaning attributed to the nodes making up the file pathname. For example, The File Manager I/O Subsystem recognizes the first node in the pathname as the name of a volume it controls provided that volume has been previously installed via the File Manager's Install Volume command.

NOTE: The possibility exists that a single file pathname can be claimed by more than one I/O Subsystem. For this reason, the order in which individual I/O Subsystems are polled can be critical.

When the pathname is recognized, internal data structures are created and maintained to sustain the connection between the file and the physical node or device, enabling the user to perform subsequent operations on the file. When no subsystem claims the pathname, an error condition is signalled.

A call to D$DISCONNECT is performed to sever the association between the file and the physical device. At "disconnect" time, the memory allocated for the data structures used to link the file to the physical node are returned to the System heap.

The internal data structures referred to above are illustrated in Appendix B. Each routine accessed in the File I/O Decoder is discussed in detail in Section III.

2.4.2 I/O Subsystems

An I/O Subsystem is a collection of procedures managing a logically similar set of I/O resources. Accessed through the File I/O Decoder, these resources are made available to file level users in a consistant manner invisible to the user. Examples of I/O Subsystems created by Texas Instruments are the 9900 File Manager (TMSW340F), the Interprocess Communication (IPC) Subsystem, and the Operator Interface (T02) I/O Subsystem.

The File I/O Decoder accesses I/O Subsystem routines via entry points present in the I/O Subsystem. These entry points correspond to entry points present in the File I/O Decoder (i.e., the D$routines described above in Subsection 2.4.1). The I/O Subsystem entry points are formed by attaching a prefix (unique to the particular subsystem) to the generic names of the file services. For example, READ is FM$READ in the File Manager and T02$READ in the Operator Interface I/O Subsystem.

Because of its purpose, a particular I/O Subsystem may not support every file service for which it posseses entry points. For example, DELETE does not make sense in an I/O Subsystem managing a line printer. These entry points are connected to stub (or dummy) routines.

If the user wishes, he can bypass the File I/O Decoder and can directly access the I/O Subsystem entry points. The I/O Subsystem reaction when each particular entry point is accessed (i.e., each file request is made) is general to all subsystems (with the exception of those file services that are not supported in a particular subsystem as mentioned above). Thus the parameterization for a given entry point type is the same across all subsystems.

2.4.3 Channels

Channels can be conceptualized as data structures over which messages (data) can be sent and received. In the context of the I/O Subsystem Standards, channels are initialized to handle message passing between two processes: the I/O Subsystem Manager and the appropriate Interface Handler. The tasks that are executed to initialize the channel, construct the message, and synchronize the message transfer are all performed transparently to the user.

It is possible for the user himself to create and pass messages to a selected Interface Handler. However, to do this, the user must identify the channel associated with the selected Interface Handler and construct the message according to the requirements of the Interface Handler.

Information on the routines that implement interprocess communication via channels using native code run-time support is presented in the Realtime Executive's User's Manual (MP373). Channel routines are documented for Microprocessor Pascal Executive users in the Microprocessor Pascal Executive User's Manual.

2.4.4 Interface Handler

The Interface Handler provides the lowest level of interface with the actual physical device. The handler enables requests for logical services made in the user's code to be translated to requests for physical services on the actual device.

By entering the I/O system at the File I/O Decoder or at the I/O Subsystem levels, the user need not be concerned with the requirements for accessing the Interface Handler. However, it is possible for the user to call the Interface Handler directly or invoke the Interface Handler via messages sent across channels as described in Subsection 2.4.3. The user should refer to the user's manual for the specific I/O Subsystem when directly accessing the Interface Handler.

SECTION III

FILE I/O DECODER ROUTINES

3.1 GENERAL

This section documents the entry points into the File I/O Decoder. The individual routines associated with these entry points are examined and the parameters passed when each routine is accessed are defined. Pascal and assembly language calling sequences are presented for each. The assembly language programmer must be familiar with the assembly language programming standards and the conventions governing register usage documented in the Realtime Executive User's Manual (MP373) to understand the assembly language code and register usage.

Prior to the description of these entry points, general information on user interface with the File I/O Decoder and parameter passing is presented.

3.2 USER INTERFACE WITH THE FILE I/O DECODER

As established previously, the user application interfaces with the File I/O Decoder to perform device independent I/O on the target system. The means by which the user achieves this interface is described below for Pascal and assembly language users.

- The Pascal User merely uses Pascal statements supported in the Microprocessor Pascal System to perform file I/O operations. These Pascal statements, RESET, REWRITE, READ, READLN, WRITE, and WRITELN, in turn invoke File I/O Decoder entry points invisibly to the user. Information on these Pascal statements is contained in the Microprocessor Pascal System User's Manual (MP351). The Pascal user can also invoke the File I/O Decoder entry points directly as demonstrated in the calling sequences below.

- The assembly language user must invoke the File I/O Decoder entry points directly in order to perform device independent I/O. The assembly language calling sequences are presented below in Subsection 3.4.

3.3 PARAMETER PASSING

In order to understand the calling sequences presented below, the user should be aware of the conventions governing the way parameters are passed. These conventions apply to both Pascal and assembly language users as decribed below.

- When parameters are passed by reference, the address of the data required by the called routine is passed. The address is passed in one word. Parameters passed by reference can be changed by the called procedure. These parameters include variables.

- When parameters are passed by value, the actual data required by the called routine is passed. This data can be passed as one or two words (e.g., Long Integers are passed as two words).

- Pointers are passed by value. However, the value passed is an address of some data (or data structure).

- Records and arrays are passed by reference (though the Pascal calling sequence may indicate that they are passed by value). In other words, the address of the record/array and not the data structure itself is passed to the called procedure.

Detailed information concerning parameter passing is presented in the Microprocessor Pascal System User's Manual (MP351) for the Pascal user and the Realtime Executive User's Manual (MP373) for the assembly language user.

3.4 FILE I/O DECODER ENTRY POINTS

The File I/O Decoder entry points are discussed below in detail. Because of system conventions, the order in which several specific routines may be invoked is fixed (e.g., the call to Connect must precede any other call to a file service, a file must be opened before accessed, etc.). This order is reflected in the descriptions below.

In addition to the entry points presented here, other entry points are present for internal use. As such, these entry points will not be called by the user and are thus not documented below.

NOTE: In writing his code to access these various entry points, the user must be careful that the sharing of variables among processes is synchronized (one method of achieving this synchronization is through the use of semaphores). FIDs can be shared among processes within scope; however the restrictions of the specific I/O Subsystem invoked must be considered in any such attempt.

3.4.1 System Initialization (D$INIT)

Initialization of the File I/O Decoder and all I/O Subsystems with which it is linked occurs automatically at power-up time. The Ghost$ procedure present in the native code run-time support contains a call to the D$INIT routine in the File I/O Decoder. As a result of D$INIT

being accessed, each of the supported I/O Subsystems are entered at their respective initialization entry points. In this manner, system initialization takes place transparently to the user.

In fact, invoking the D$INIT entry point leads to initiating the devices, characteristics tables, and configuration data associated with the File I/O Decoder and with each supported subsystem. Appendix B presents pictures of the various data structures. Section VI describes how a system is configured.

Pascal Calling Sequence:

```
PROCEDURE D$INIT
```

Assembly Language Calling Sequence:

```
DATA  CALL$
DATA  D$INIT
```

3.4.2 Connecting the File to an I/O Subsystem (D$CONNECT)

D$CONNECT must be the first file service requested and is called to connect a specified file pathname with the appropriate I/O Subsystem. At "CONNECT" time, the file decoder will invoke each subsystem supported on the target in succession until some subsystem recognizes the pathname parameter passed with the call. If no subsystem recognizes the pathname, an error condition is signaled (the naming conventions applicable to the file pathname are specific to the I/O Subsystem being invoked). When the file is recognized, internal data structures are created and maintained to sustain the connection between the file and the physical node (or device). Also, the File Identifier (FID) is initialized. The FID connects the user with the associated I/O Subsystem enabling the user to perform subsequent file operations on the file.

NOTE: The Pascal user must do a type override to enable the pathname pointer to point to a buffer of sufficient size to hold the pathname.

After connect, legal file requests are D$CREATE, D$OPEN, D$DELETE, and D$DISCONNECT.

The calling parameters for D$CONNECT are defined below.

Parameter	Definition	Limits	Input/Output
Pathname Pointer	Pointer to buffer containing pathname of file to be serviced. This pathname is subsystem dependent.	Pointer: Word address. Buffer: Character array, length of which depends on the subsystem. The pathname is left-justified in the buffer.	Input
Number of Characters	Number of characters contained in the pathname.	Integer	Input
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Output

Pascal Calling Sequence:

```
PROCEDURE D$CONNECT(     Pathname    :  DUMMY_BUFFER_PTR;
                         No_of_char  :  INTEGER;
                   VAR My_fid        :  FID );
```

Assembly Language Calling Sequence:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

Pathname Pointer	at 0
Number of Characters	at 2
My Fid	at 4

```
MOV    *LF,*SP+              PASSING PATHNAME POINTER
MOV    @2(LF),*SP+           PASSING NUMBER OF CHARACTERS
MOV    LF,*SP                PASSING FILE IDENTIFIER
A      @FOUR(CODE),*SP+
DATA   CALL$
DATA   D$CONNECT
```

where the following sequence is in the user's prologue:

```
MOD   EQU   $                MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD          PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
```

3.4.3 D$CREATE

The entry point D$CREATE creates a file by invoking the appropriate I/O Subsystem. The attributes of the file that is created are defined by the calling parameters passed to the D$CREATE command. The meaning of these attributes are discussed in detail in Appendix E.

The parameterization for D$CREATE pertains to I/O Subsystems managing multifile devices. These parameters are ignored by I/O Subsystems not supporting multifile devices. The Create function leaves the file in a closed condition so that the only legal operations are D$OPEN, D$DELETE, and D$DISCONNECT.

The calling parameters passed to D$CREATE are described below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Password List Pointer	Pointer to record structure containing Creator and User Passwords	Pointer: Word address; Record Fields: Creator Password: character array; User Password: character array.	Input
Protection Code	Record defining access protection. This record has four fields each defining the level of access protection for a separate access activity. These fields are: Read, Write, Modify, and Execute. The levels of protection assigned are specified as #1 Any Access; #2 User Password; #3 Creator Password; and #4 No Access. See Appendix E for details.	Read (Bits 0-3): #1 thru #4; Write (Bits 4-7): #1 thru #4; Modify (Bits 8-11): #1 thru #4; Exec. (Bits 12-15): #1 thru #4.	Input

File Type	A record containing four fields defining the physical and logical organization of the file. These file attributes are discussed in Appendix E.	File Type (Bits 0-3): Contig.=1; Non-contig.= 2; Record Type (Bits 4-7): Free Len.=1; Var. Len.=2; Fixed Len.=3; Usage (Bits 8-11): Data File=1; Compression (Bits 12-15): Uncompress.=1; Compress.=2.	Input
Logical Record Length	Length in bytes of the records contained in a file. For Fixed Length record files, the actual record length is used. For Variable Length record files, the maximum length is used. For Free Region record files, a record length of 1 byte is used.	Any positive integer.	Input
Primary Allocation	The minimum storage space (number of records) to be allocated to the file (represents maximum number in a contiguous file). Default is indicated by blank or zero. Free Length default = 800; Non-Free Length Default = 50.	Any non-negative Long Integer (two words).	Input
Secondary Allocation	The increment (number of records) by which a non-contiguous file is allowed to grow per expansion (up to 16 expansion steps are allowed for non-contiguous files. Default is indicated by blank or zero. Default = Primary Allocation.	Any non-negative Long Integer (two words).	Input

Pascal Calling Sequence:

```
PROCEDURE D$CREATE (   My_fid          : FID;
                       Pass_code_list: PASSWORD_LIST_PTR;
                       Protect         : PROT;
                       FT              : FILE_TYPE
                       Log_rec_len     : INTEGER;
                       Pa_log_rec      : LONGINT;
                       Sa_log_rec      : LONGINT);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

My Fid	at 0
Passcode List Pointer	at 2
Protection Code	at 4
File Type	at 6
Logical Record Length	at 8
Primary Allocation	at 10
Secondary Allocation	at 14

```
        MOV    *LF,*SP+            PASSING MY FID
        MOV    @2(LF),*SP+         PASSING PASS CODE LIST POINTER
        MOV    LF,*SP              PASSING ACCESS PROTECTION ADDRESS
        A      @FOUR(CODE),*SP+
        MOV    LF,*SP              PASSING FILE TYPE ADDRESS
        A      @SIX(CODE),*SP+
        MOV    @8(LF),*SP+         PASSING LOGICAL RECORD LENGTH
        MOV    @10(LF),*SP+        PASSING PRIMARY ALLOCATION (2 WORDS)
        MOV    @12(LF),*SP+
        MOV    @14(LF),*SP+        PASSING SECONDARY ALLOCATION (2 WORDS)
        MOV    @16(LF),*SP+
        DATA   CALL$
        DATA   D$CREATE
```

Where the following sequence appears in the user's prologue:

```
MOD     EQU    $                   MOD LABELS BEGINNING OF LOCAL DATA
        DATA   PRO-MOD             PRO LABELS BEGINNING OF EXECUTABLE CODE
          .
          .
FOUR    EQU    $-MOD
        DATA   4
SIX     EQU    $-MOD
        DATA   6
```

3.4.4 D$OPEN

D$OPEN is called to prepare a file for reading, writing, or both depending on the access type specified by the Access Type parameter

passed to this procedure. If the passwords passed to this command do not match the passwords required for the access type desired as identifed when the file was created, the Open will fail. If the Open is successful, the type of of the file opened along with its number of logical records and record length (if appropriate) are returned to the user. Once D$OPEN has been called, all file services with the exception of D$CONNECT, D$CREATE, D$DELETE, and D$DISCONNECT are allowed until the file is closed.

The parameters passed to the D$OPEN command are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Password Pointer	Pointer to an array containing the Creator or User Password.	Pointer: Word address; Array: An array four characters in length.	Input
Access Type	Indication of the type of access in which I/O is executed. Access type remains effective until the file is closed. Access types are defined in Appendix E.	Byte Relative = 1; Sequential = 2; Direct = 3.	Input
Access Privilege	Relationship between user and the file which defines the user's activity and precludes other user access. The user can specify True or False for each of five fields: Exclusion, Read, Write, Execute or Extend. For more information, see Appendix E. (Note: this record is packed to use bits 0,1,2,3, and 15 of the word.)	Exclusion, Read, Write, Execute, and Extend: False = 0; True = 1.	Input
File Type	Record to which the File Type defined when the file was created is returned.	Pointer: Word Address.	Output

Logical Record Length	Integer to which the logical record length defined when file was created is returned.	Integer.	Output
Number of Logical Records	Integer to which number of logical records in the file is returned. In files of variable length records, an end of file record is present and counted as an extra record.	Integer.	Output

Pascal Calling Sequence:

```
PROCEDURE D$OPEN (    My_fid                : FID
                      Passwords             : PASSWORD_PTR;
                      Access_type           : FILE_ACCESS_MODE;
                      Access_priv           : FILE_ACCESS_PRIVILEGE;
                  VAR Ft                    : FILE_TYPE;
                  VAR Logical_rec_length    : INTEGER;
                  VAR Number_log_rec        : LONGINT);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records passed by address):

My Fid	at 0
Password Pointer	at 2
Access Type	at 4
Access Privilege	at 6
File Type	at 8
Logical Record Length	at 10
Number of Logical Records	at 12

```
MOV   *LF,*SP +              PASSING MY FID
MOV   @2(LF),*SP+            PASSING PASSWORD POINTER
MOV   @4(LF),*SP+            PASSING ACCESS TYPE
MOV   LF,*SP                 PASSING ACCESS PRIVILEGE ADDRESS
A     @SIX(CODE),*SP +
MOV   LF,*SP                 PASSING FILE TYPE ADDRESS
A     @EIGHT(CODE),*SP+
MOV   LF,*SP                 PASSING LOGICAL RECORD LENGTH ADDRESS
A     @TEN(CODE),*SP+
MOV   LF,*SP                 PASSING NUM. OF LOG RECORDS ADDRESS
A     @TWELVE(CODE),*SP+
DATA  CALL$
DATA  D$OPEN
```

where the following sequence appears in the user's prologue:

```
MOD     EQU   $                 MOD LABELS BEGINNING OF LOCAL DATA
        DATA  PRO-MOD           PRO LABELS BEGINNING OF EXECUTABLE CODE
          .
          .
SIX     EQU   $-MOD
        DATA  6
EIGHT   EQU   $-MOD
        DATA  8
TEN     EQU   $-MOD
        DATA  10
TWELVE  EQU   $-MOD
        DATA  12
```

3.4.5 D$READ

The call to D$READ initiates a read operation on a file. The calling process begins the read and can (if appropriate) continue to execute without delaying until the read operation is complete.

D$READ provides for record organized data transfer. If the file is not organized into logical records, the buffer will be filled in physical record length increments until no further physical record can be held. Data transfer begins at the current file position. At the end of the read operation, the file position is left pointing at the next available unit of data (record or byte, as appropriate).

The variable parameter Count is not set until I/O is complete. To be sure that this parameter is set correctly, the user should call D$WAIT after the call to D$READ.

The parameters passed to D$READ are described below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Buffer Pointer	Pointer to an array in RAM into which the data read is transerred.	Pointer: Word Address Array: Ram-resident data area large enough to accomodate the number of bytes in the read.	Input

Read Count	Number of bytes to be read; input lesser of the number requested and the logical record length.	Positive Integer	Input
Count	Integer to which the actual number of bytes read is transferred.	Integer.	Output

Pascal Calling Sequence:

```
PROCEDURE D$READ (    My_fid        : FID;
                      Buffer        : DUMMY_BUFFER_PTR;
                      Read_count    : INTEGER;
                  VAR Count         : INTEGER);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

Fid	at 0
Buffer Pointer	at 2
Read Count	at 4
Count	at 6

```
MOV   *LF,*SP+          PASSING FID
MOV   @2(LF),*SP+       PASSING BUFFER POINTER
MOV   @4(LF),*SP+       PASSING READ COUNT
MOV   LF,*SP            PASSING COUNT
A     @SIX(CODE),*SP+
DATA  CALL$
DATA  D$READ
```

where the following sequence appears in the user's prologue:

```
MOD   EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
SIX   EQU   $-MOD
      DATA  6
```

3.4.6 D$WRITE

The call to D$WRITE initiates a write operation on a file. The calling process begins the write and can (if appropriate) continue to execute without delaying until the write operation is complete.

D$WRITE provides for record organized data transfer. If the file is not organized into logical records, the buffer will be filled in

physical record length increments until no further physical record can be held. At the end of the write operation, the file is positioned to be ready to store the next unit of data (record or byte as appropriate). Type of access permited (byte-relative, sequential, or direct) is a function of the device itself and the access type for which the file was opened (see D$OPEN above). Access protection and privilege are functions of the parameters passed to the D$CREATE and D$OPEN routines respectively.

The parameters passed to D$WRITE are described below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Buffer Pointer	Pointer to an array in RAM from which data is to be transferred.	Pointer: Word Address Array: Ram-resident array size of which large enough to accomodate data transfer (as specified in Write Count below).	Input
Write Count	The number of bytes to be written.	Positive Integer	Input

Pascal Calling Sequence:

```
PROCEDURE D$WRITE (     My_fid          : FID;
                        Buffer          : DUMMY_BUFFER_PTR;
                        Write_count     : INTEGER);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
        My Fid                          at 0
        Buffer Pointer                  at 2
        Write Count                     at 4


MOV     *LF,*SP+            PASSING MY FID
MOV     @2(LF),*SP+         PASSING BUFFER POINTER
MOV     @4(LF),*SP+         PASSING WRITE COUNT
DATA    CALL$
DATA    D$WRITE
```

3.4.7 D$RDWAIT

D$RDWAIT is called to execute (as oppossed to initiate) a read operation. Unlike the action of D$READ, the calling process begins the read and is then suspended until the read is completed. In other words, a call to D$RDWAIT results in the same action as a call to D$READ followed by a call to D$WAIT. The parameterization of D$RDWAIT is the same as for D$READ.

3.4.8 D$WRWAIT

D$WRWAIT is called to execute (as oppossed to initiate) a write operation. Unlike the action of D$WRITE, the calling process begins the write and is then suspended until the write is completed. In other words, a call to D$WRWAIT results in the same action as a call to D$WRITE followed by a call to D$WAIT. The parameterization of D$WRWAIT is the same as for D$WRITE.

3.4.9 D$POSITION

D$POSITION is called to move the file position forward or backward prior to the next I/O attempt. The Relative parameter specifies whether the change in file position will be relative to the current file position or absolute (relative to the start of the file). If this boolean parameter is passed as True, the Record parameter indicates the number of records by which the file position will change from the current position (the sign of this value indicates whether movement is forward or backward). If Relative is passed as False, the Record parameter specifies the absolute record number at which the file will be positioned. An error condition results if an attempt is made to position the file beyond the end of file mark or prior to the beginning of file mark. Errors may also occur if the device does not support record number.

The parameters passed to D$POSITION are described below.

Parameter	Definition	Limits	Input/Output
File	Value returned by	Integer	Input

Identifier	D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.		
Relative	Boolean parameter by which user specifies if repositioning of file will take place in relation to current file position (true) or absolute. See explanation above.	False = 0; True = 1.	Input
Record Number	Depending on value of Relative parameter, the number of records the file will be moved from the current position, or the record number at which the file will be newly positioned. See explanation above.	Long Integer (two words)	Input

Pascal Calling Sequence:

```
PROCEDURE D$POSITION(   My_fid      : FID;
                        Relative    : BOOLEAN;
                        Record No.  : LONGINT);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

My fid	at 0
Relative	at 2
Record No.	at 4

```
MOV    *LF,*SP+          PASSING MY FID
MOV    @2(LF),*SP+       PASSING RELATIVE
MOV    @4(LF),*SP+       PASSING FIRST WORD OF RECORD NO.
MOV    @6(LF),*SP+       PASSING SECOND WORD OF RECORD NO.
DATA   CALL$
DATA   D$POSI
```

3.4.10 D$WAIT

By calling D$WAIT, a user requires the process to wait for the completion of the I/O service he previously initiated for that file (FID). Regardless, a wait will automatically occur on a user's initiate I/O request if he has a current request outstanding on that same file (FID). The calling parameters passed to D$WAIT are described below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input

Pascal Calling Sequence:

```
PROCEDURE D$WAIT (    My_fid        : FID);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

FID at 0

```
MOV    *LF,*SP+          PASSING FID
DATA   CALL$
DATA   D$WAIT
```

3.4.11 D$STATUS

The function D$STATUS can be called once a file has been CONNECTED to check on the current status (success or failure) of the user's oldest outstanding request on a file (the oldest request on the FID). In the File Identifier Record itself (see Appendix B for illustration), a status field is present to capture status information. This function enables the user to inspect this information. Appendix C details the various status messages and provides some suggestions for corrective actions when appropriate.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling I/O Subsystem to associate a specific file with a specific user.	Integer	Input

Pascal Calling Sequence:

```
FUNCTION D$STATUS (My_Fid : FID):INTEGER;
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements in the calling procedure's local frame.

My Fid	at 0
Result will be saved	at 2

```
MOV  *LF, *SP+      PASSING FID
DATA CALL$
DATA D$STAT
MOV *SP,@2(LF)      SAVING RESULT
```

3.4.12 D$DSTATUS

D$DSTATUS enables the user to examine status information on the I/O Subsystem level (compare to D$STATUS which returns a status message at the FID level). Because the meanings of the status messages returned by D$DSTATUS are I/O Subsystem dependent, the user must refer to the user's manual dedicated to the specific I/O Subsystem for message definitions and corrective actions. The only parameter passed to D$DSTATUS is the FID.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input

Pascal Calling Sequence:

```
FUNCTION D$DSTATUS (My_Fid : FID): INTEGER
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements in the calling procedure's local frame.

```
My Fid                          at 0
Result will be saved            at 2
```

```
MOV  *LF, *SP+      PASSING FID
DATA CALL$
DATA D$DSTA
MOV *SP,@2(LF)      SAVING RESULT
```

3.4.13 D$VALID

D$VALID is a Boolean function that may be called to check for valid state transitions for the FID. State refers to the FID condition such as Connected, Created, Open for Access, etc (refer to Appendix C for information regarding valid state changes). The parameters passed to D$VALID are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Opcode	Operation attempted on the FID.	$check = 0; $open = 1; $read = 2; $write = 3; $close = 4; $disconnect = 5; $create = 6; $delete = 7; $position = 8.	Input

Pascal Calling Sequence:

```
FUNCTION D$VALID( My_fid      : FID;
                  Op          : FID_OPERATION):BOOLEAN;
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
        My Fid                          at 0
        Fid Operation                   at 2
        Result will be saved            at 4

MOV     *LF,*SP+            PASSING FID POINTER
MOV     @2(LF),*SP+         PASSING FID OPERATION
DATA    CALL$
DATA    D$VALI
MOV     *SP,@4(LF)          SAVING RESULT
```

3.4.14 D$ABORTIO

D$ABORTIO is called to abort all outstanding read/write operations a user has requested on a file (i.e, all outstanding read/write requests on a FID). The only parameter passed to D$ABORTIO is the FID.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input

Pascal Calling Sequence:

```
PROCEDURE D$ABORTIO(My_fid : FID);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
        My Fid                          at 0

MOV     *LF,*SP+            PASSING MY FID
DATA    CALL$
DATA    D$ABOR
```

3.4.15 D$CLOSE

D$CLOSE is accessed to enable the user to close the file (FID) when no more outstanding I/O requests are present. The Close EOF parameter (a BOOLEAN parameter) enables the file to be closed with the End-of-File mark being placed at the current file position (if response is True). Otherwise (False), the End-of-File mark is left unchanged (i.e., the

last record in the file remains the same). The user who closes with End of File must be careful to avoid inadvertently placing an EOF prior to the last record. This is especially true if the user had written to the file using random access. The user who had opened a file for reading only should always close without EOF to avoid an error.

After a call to D$CLOSE, the only valid file requests are D$OPEN, D$DELETE, or D$DISCONNECT.

The parameters passed to D$CLOSE are described below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Close End of File	BOOLEAN parameter by which user specifies whether he wishes to close with EOF. A true response means that an EOF mark should follow the last record accessed. False indicates that EOF remains unchanged.	False = 0; True = 1.	Input

Pascal Calling Sequence:

```
PROCEDURE D$CLOSE  (     My_fid          : FID;
                         Close_With_EOF  : BOOLEAN  );
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
My Fid                          at 0
Close With EOF                  at 2
```

```
MOV     *LF,*SP+            PASSING FID
MOV     @2(LF),*SP+         PASSING CLOSE WITH EOF
DATA    CALL$
DATA    D$CLOSE
```

3.4.16 D$DELETE

D$DELETE is called to delete a file that has been closed for access, thus preventing all further requests except for D$DISCONNECT or D$CREATE. The calling parameters passed to D$DELETE are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to associate a specific file with a specific user.	Integer	Input
Password Pointer	Pointer to a data structure containing the creator or user password. Required password specified in Modify field of Protection record defined when file was created.	Pointer: Word Address. Passwords: Character arrays, each four characters in length (either Creator or User).	Input

Pascal Calling Sequence:

```
PROCEDURE   D$DELETE ( My_Fid  :  FID;
                       Pass    :  PW_PTR);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

My Fid	at 0
Password Pointer	at 2

```
MOV     *LF,*SP+            PASSING FID
MOV     @2(LF),*SP+         PASSING PASSING POINTER
DATA    CALL$
DATA    D$DELE
```

3.4.17 D$DISCONNECT

The last operation performed on the file (FID) by a user is D$DISCONNECT. The call to D$DISCONNECT severs the connection between the file and the physical node (or device) on the target. As a result of this procedure, the memory allocated to hold the File Identifier Record (FID) is returned to the heap.

The only parameter passed to D$DISCONNECT, the File Identifier is passed by address on input.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by D$CONNECT enabling the I/O Subsystem to file with a specific user.	Integer	Input

Pascal Calling Sequence:

```
PROCEDURE D$DISCONNECT  (VAR My_fid : FID);
```

Assembly Language Calling Sequence:

```
MOV   *LF,*SP+    PASSING FID
DATA  CALL$
DATA  D$DISC
```

3.4.18 D$TERM

At process termination, the user may call D$TERM to deallocate memory resources holding the remaining file data structures (including all FIDs currently outstanding). D$TERM provides the means to terminate all user connections with I/O Subsystems using one line of code. Thus, the call to D$TERM makes calls to D$DISCONNECT for individual subsystems unnecessary. D$TERM has no calling parameters.

SECTION IV

THE INTERPROCESS COMMUNICATION SUBSYSTEM

4.1 GENERAL

This section of the manual presents a specific I/O Subsystem, the Interprocess Communication (IPC) Subsystem. This subsystem is supplied to the Pascal user in the Microprocessor Pascal Executive (MPX) and provided for the assembly language user in the Device Independent File I/O Package. The purpose of this subsystem is to implement file-level communication between processes within a 9900 target system. This is accomplished by passing data from process to process via channels using in-memory buffers.

Interprocess Communication (IPC) Subsystem routines can be invoked via the File I/O Decoder or directly by the user. Subsection 4.3 below describes the former mode; Subsection 4.4 the latter.

Interface with the IPC Subsystem is consistent with the standards governing interface with all other I/O Subsystems (as described in this manual) that may be accessed by the File I/O Decoder. This is true even though the IPC Subsystem is concerned with processes and not devices.

4.2 IMPLEMENTATION OF THE IPC SUBSYSTEM

Before describing the methods of invoking the IPC Subsystem, it is necessary to discuss its implementation.

I/O between processes is managed through files. An IPC file may be thought of as a path of information between processes. A port can be viewed as providing a sending and receiving process with access to the path. Each process can be viewed as a node. Ports are defined for input and output by the invocation of special IPC routines.

Information is passed over the path in the form of messages (the structure of messages along with other IPC data structures are presented in Appendix B)

A file (comprising a message path) is implemented by a data structure called the Pathname Record The Pathname Record contains data structures required to synchronize and control the message flow between processes. Message flow is actually implemented over channels The Channel Record is used to implement channels. Both the Pathname Record and the Channel Record are system global data structures.

Ports are implemented by the File Identifier (FID), the File Variable Record, and the Message Record. The latter data structures are local to the user's implementation of the IPC I/O Subsystem. The IPC data structures are illustrated and described in Appendix B.

4.3 IPC ACCESS VIA THE FILE I/O DECODER

Like other I/O Subsystems, the IPC Subsystem has a general set of entry points corresponding to the file service entry points present in the File I/O Decoder (File I/O Decoder entry points have been described in detail in Section III of this manual). By way of these I/O Subsystem entry points, the device independent file services requested in the File I/O Decoder are translated into service requests of the IPC Subsystem. When accessing the IPC Subsystem in this way, the user is oblivious of its entry points and their calling sequences. I/O Subsystem initalization takes place via the File I/O Decoder entry point for initialization and all access to IPC is transparent to the user.

4.4 DIRECT USER ACCESS OF IPC SUBSYSTEM ENTRY POINTS

The paragraphs that follow describe how Pascal and assembly language users may directly access the entry points into the IPC Subsystem. Each entry point definiton is presented in terms of its meaning within this particular I/O Subsystem. However, the entry points into this I/O Subsystem are general to all I/O Subsystems. Thus, the parameter definitons and calling sequences presented below have a general relevance to all I/O Subsystems.

For the definitions of the file service routines (accessed via the File I/O Decoder) corresponding to these I/O Subsystem entry points, refer to Section III of this manual.

The entry point names for this subsystem are formed by adding the prefix "IPC$" to the generic name of the file serivce (connect, open, read, etc.). The rules and conventions governing parameter passing in the calling sequences presented below are the same as those described in Subsection 3.1.

4.4.1 IPC$INIT

IPC$INIT is called to initialize the data structures used within the IPC Subsystem and set up synchronization and system access to all files. This routine initializes a mutual exclusion guard for the list of all pathnames connected to the IPC Subsystem and links the subsystem record (see below) with the list of all subsystem records. IPC$INIT must be called once prior to the calling of any other IPC routines.

The parameters passed to IPC$INIT are defined below..

Parameter	Definition	Limits	Input/Output
Service Directory	Pointer to the Service Directory for the IPC Subsystem.	Integer. (Additional detail can be found following the calling sequences.)	Input
Port Constants Record Pointer	Pointer to the Port Constants Record. Not used by this subsystem.	Ignored by this subsystem.	Ignored
Subsystem Record	Pointer to Subsystem related data structures. This pointer must be supplied in each call to the IPC CONNECT routine.	Integer.	Output

Pascal Calling Sequence:

```
PROCEDURE IPC$INIT(    Service   : Service_directory_ptr;
                       Port_cons : Port_constants_ptr;
                   VAR Subsys    : Subsystem_ptr);
```

Assembly Language Calling Sequence:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

Service directory Pointer at 0
Port Constants Pointer at 2
Subsystem Record Pointer at 4

```
MOV   *LF,*SP+              PASSING SERVICE DIRECTORY POINTER
MOV   @2(LF),*SP+           PASSING PORT CONSTANTS RECORD POINTER
MOV   LF,*SP                PASSING SUBSYSTEM RECORD POINTER
A     @FOUR(CODE),*SP+
DATA  CALL$
DATA  IPC$IN
```

where the following sequence appears in the user's code:

```
MOD   EQU   $              MOD LABLES BEGINNING OF LOCAL DATA
      DATA PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR EQU   $-MOD
      DATA 4
```

Service Directory, the data structure passed as the first parameter to this procedure, is supplied in the IPC Subsystem. The Pascal user can declare the procedure containing this directory (IPC$SD) as EXTERNAL and pass IPC$SD's location as the Service Directory parameter. This is accomplished by making Service Directory an Integer via a type override and setting the parameter Service Directory to the location of IPC$SD:

```
SERVICE::INTEGER := LOCATION (IPC$SD)
```

The assembly language programmer REF's IPC$SD and sets the Service Directory to the location of IPC$SD using a DATA instruction.

4.4.2 IPC$CONNECT

IPC$CONNECT searches the list of all pathnames connected to the IPC Subsystem to determine if there is a path corresponding to the input parameter Pathname (this parameter is passed by address even though this is not indicated in the Pascal calling sequence). If not, a path is constructed by allocating a pathname record and associated message channel.

NOTE: IPC accepts any pathname. Since the I/O decoder presents pathnames to I/O subsystems for connection in the order in which the subsystems are enumerated in the I/O subsystem directory present in CONFIG, the entry for IPC must be last if other subsystems are to have a chance to accept the pathname.

Once a pathname record is obtained, IPC$CONNECT creates local data structures defining a path and returns the identifier of the associated file as the FID parameter which must be used to make subsequent I/O service requests.

The parameters passed to IPC$CONNECT are described below.

Parameter	Definition	Limits	Input/Output
Subsystem Record	Pointer to Subsystem related data structures. This pointer is supplied by the IPC$INIT routine.	Integer.	Input

Pathname	Memory data structure holding file pathname specifying name of path over which messages will be sent.	Character array large enough to accomodate pathname being passed. (Pascal user will need to do a type override.)	Input
Length	Character length of the above pathname.	Integer.	Input
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Output

Pascal Calling Sequence:

```
PROCEDURE IPC$CONNECT(     Sub       :  SUBSYSTEM_PTR;
                       VAR Pathname  :  DUMMY_BUFFER;
                           Length    :  INTEGER;
                       VAR F         :  FID );
```

Assembly Language Calling Sequence:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

Subsystem Record Pointer	at 0
Pathname Address	at 2
Length	at 4
FID	at 6

```
MOV    *LF,*SP+          PASSING SUBSYSTEM RECORD POINTER
MOV    LF,*SP            PASSING PATHNAME ADDRESS
INCT   *SP+
MOV    @4(LF),*SP+       PASSING LENGTH
MOV    LF,*SP            PASSING FID VARIABLE
A      @SIX(CODE),*SP+
DATA   CALL$
DATA   IPC$CO
```

where the following sequence appears in the user's code:

```
MOD   EQU   $            MOD LABLES BEGINNING OF LOCAL DATA
      DATA  PRO-MOD      PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
SIX   EQU   $-MOD
      DATA  6
```

4.4.3 IPC$CREATE

Routine IPC$CREATE must be called by a process connected to a path in order to define the characteristics of that path.

The parameters File Type and Logical Record Length (see below) define characteristics of the file serving as the path. The parameters Password, Protect, Primary Allocation, and Secondary Allocation are included for compatibility with the "Create" service routines of other I/O subsystems but are ignored by the IPC Subsystem.

The parameters passed to IPC$CREATE are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Input
Password Pointer	Pointer to record structure containing Creator and User Passwords.	Ignored by this subsystem.	Ignored
Protection Code	Record defining access protection.	Ignored by this subsystem.	Ignored
File Type	A record containing four fields defining the physical and logical organization of the file. These file attributes are discussed in Appendix E.	File Type (Bits 0-3): Contig.=1; Non-contig.= 2; Record Type (Bits 4-7): Free Len.=1; Var. Len.=2; Fixed Len.=3; Usage (Bits 8-11): Data File=1; Compression (Bits 12-15): Uncompress.=1; Compress.=2.	Input
Logical Record Length	The actual (fixed length) or maximum allowable (variable length) byte length of the records in the file.	Any positive integer.	Input
Primary Allocation	Minimum storage space in the file.	This two-word parameter is ignored.	Ignored

Secondary Allocation	Incremental storage space in the file.	This two-word parameter is ignored.	Ignored

Pascal Calling Sequence:

```
PROCEDURE IPC$CREATE (   My_fid          : FID;
                         Pass_code_list  : PASSWORD_LIST_POINTER;
                         Protect         : PROT;
                         Ft              : FILE_TYPE;
                         Log_rec_len     : INTEGER;
                         Pa_log_rec      : LONGINT;
                         Sa_log_rec      : LONGINT);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

My Fid	at 0
Passcode List Pointer	at 2
Protection Code	at 4
File Type	at 6
Logical Record Length	at 8
Primary Allocation	at 10
Secondary Allocation	at 14

```
        MOV     *LF,*SP+            PASSING FID
        CLR     *SP+                PASS CODE PARAMETER IGNORED
        CLR     *SP+                ACCESS PROTECTION PARAMETER IGNORED
        MOV     LF,*SP              PASSING FILE TYPE ADDRESS
        A       @SIX(CODE),*SP+
        MOV     @8(LF),*SP+         PASSING LOGICAL RECORD LENGTH
        MOV     @10(LF),*SP+        PASSING PRIMARY ALLOCATION (2 WORDS)
        MOV     @12(LF),*SP+
        MOV     @14(LF),*SP+        PASSING SECONDARY ALLOCATION (2 WORDS)
        MOV     @16(LF),*SP+
        DATA    CALL$
        DATA    IPC$CR
```

Where the following sequence appears in the prologue of the application

```
MOD     EQU     $                   MOD LABELS BEGINNING OF LOCAL DATA
        DATA    PRO-MOD             PRO LABELS BEGINNING OF EXECUTABLE CODE
          .
          .
SIX     EQU     $-MOD
        DATA    6
```

4.4.4 IPC$OPEN

After the characteristics of a path have been defined, each process that is connected to the path must call IPC$OPEN to open the port for communication.

The Privilege parameter specifies whether the port will be a producer (writer) or a consumer (reader). Note that a port cannot be both. The Password and Access Type parameters are not used by the IPC Subsystem. The parameters File Type and Logical Record Length are returned with the values that were specified when IPC$CREATE was called. The number of Records parameter is set to 0.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Input
Password Pointer	Pointer to an array containing Creator or User Password.	Ignored by this subsystem.	Ignored
Access Type	Indication of the type of access by which I/O is performed.	Ignored by this subsystem.	Ignored
Access Privilege	This parameter defines whether port will be used for input or output. Port cannot be used for both.	Read Access (Bit 1): False = 0; True = 1; Write Access (Bit 2): False = 0; True = 1; Execute Access (Bit 3): Must be False.	Input
File Type	Variable to which File Type record, defined when file was created, is returned.	Defined above IPC$CREATE.	Output
Logical Record Length	Integer to which Logical Record Length, defined when file was created, is returned.	Integer.	Output
Number of Logical Records	Long Integer to which the number of logical records is returned. This number is always zero.	Long Integer.	Output

Pascal Calling Sequence:

```
PROCEDURE IPC$OPEN (    My_fid              : FID;
                        Password            : PASSWORD_PTR;
                        Access_type         : FILE_ACCESS_MODE;
                        Access_priv         : FILE_ACCESS_PRIVILEGE;
                    VAR FT                  : FILE_TYPE;
                    VAR Logical_rec_length  : INTEGER;
                    VAR Number_log_rec      : LONGINT);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records passed by address):

My Fid	at 0
Password Pointer	at 2
Access Type	at 4
Access Privilege	at 6
File Type	at 8
Logical Record Length	at 10
Number of Logical Records	at 12

```
MOV   *LF,*SP +          PASSING FID
CLR   *SP+               PASSWORD POINTER IS IGNORED
CLR   *SP+               ACCESS TYPE PARAMETER IS IGNORED
MOV   LF,*SP             PASSING ACCESS PRIVILEGE ADDRESS
A     @SIX(CODE),*SP +
MOV   LF,*SP             PASSING FILE TYPE ADDRESS
A     @EIGHT(CODE),*SP+
MOV   LF,*SP             PASSING LOGICAL RECORD LENGTH ADDRESS
A     @TEN(CODE),*SP+
MOV   LF,*SP             PASSING NUM. OF LOG RECORDS ADDRESS
A     @TWELVE(CODE),*SP+
DATA  CALL$
CALL  IPC$OP
```

where the following sequence appears in the user's code:

```
MOD     EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
        DATA  PRO-MOD        PRO LABELS BEGINNING OF EXECUTABLE CODE
          .
          .
SIX     EQU   $-MOD
        DATA  6
EIGHT   EQU   $-MOD
        DATA  8
TEN     EQU   $-MOD
        DATA  10
TWELVE  EQU   $-MOD
        DATA  12
```

4.4.5 IPC$WRITE

Following the call to IPC$OPEN, a process can begin transmitting or receiving data from the file depending upon its I/O mode. If a process opens a file as a writer, it can transmit data to the file by calling IPC$WRITE.

NOTE: The Pascal user can do a type override to enable the buffer pointer to point to a buffer of sufficent size to accomodate the data transfer.

The parameters passed to IPC$WRITE are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Output
Buffer Pointer	Pointer to a character array in RAM from which data is written.	Pointer: Word Address Array: Ram-resident data area containing data for transfer.	Input
Write Count	The number of bytes to be written.	Positive Integer	Input

Pascal Calling Sequence:

```
PROCEDURE IPC$WRITE(    My_fid        : FID;
                        Buffer        : DUMMY_BUFFER_PTR;
                        Write_count   : INTEGER);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
        My Fid                          at 0
        Buffer Pointer                  at 2
        Write Count                     at 4


    MOV   *LF,*SP+          PASSING FID
    MOV   @2(LF),*SP+       PASSING BUFFER POINTER
    MOV   @4(LF),*SP+       PASSING WRITE COUNT
    DATA  CALL$
    DATA  IPC$WR
```

4.4.6 IPC$READ

If a process opens a file as a reader, it can receive data from the file by calling IPC$READ.

The count parameter (below) is not set until I/O is complete. To be sure that this parameter is set correctly, the user should call IPC$WAIT after the call to IPC$READ. The parameters passed to IPC$READ are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Input
Buffer Pointer	Pointer to a character array in RAM into which the data read is transferred.	Pointer: Word Address Array: Ram-resident data area large enough to accomodate the number of characters read.	Input
Read Count	Number of bytes to be read.	Positive Integer	Input
Count	Number of bytes that were actually transferred.	Positive Integer	Output

Pascal Calling Sequence:

```
PROCEDURE IPC$READ ( My_Fid          : FID;
                     Buffer          : DUMMY_BUFFER_PTR;
                     Read_count      : INTEGER;
                VAR Count            : INTEGER);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

My Fid	at 0
Buffer Pointer	at 2
Read Count	at 4
Count	at 6

```
MOV   *LF,*SP+          PASSING FID
MOV   @2(LF),*SP+       PASSING BUFFER POINTER
MOV   @4(LF),*SP+       PASSING READ COUNT
MOV   LF,*SP            PASSING COUNT
A     @SIX(CODE),*SP+
DATA  CALL$
DATA  IPC$RE
```

where the following sequence appears in the user's code:

```
MOD   EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
SIX   EQU   $-MOD
      DATA  6
```

4.4.7 IPC$WAIT

Following a call to either IPC$READ or IPC$WRITE, a process should call IPC$WAIT. This routine will wait for the completion of all outstanding requests on the specified FID. Until this routine has been called, a process cannot be sure that all of the data has been transmitted or received.

The parameters passed to IPC$WAIT are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Input

Pascal Calling Sequence:

```
PROCEDURE IPC$WAIT (    My_fid        : FID);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame:

My Fid at 0

```
MOV   *LF,*SP+          PASSING FID
DATA  CALL$
DATA  IPC$WA
```

4.4.8 IPC$CLOSE

Once all data has been written or no more data is to be read from a file, IPC$CLOSE should be called to mark the file as having stopped data communication. IPC$CLOSE shuts down a port without disassociating it from the path to which it is connected. The parameter Close With EOF (see below) is not used by the IPC Subsystem. If a port is closed, it may be reopened with the same or different I/O characteristics (see IPC$OPEN above).

The parameters passed to IPC$CLOSE are defined below.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer.	Input
Close With End of File	Boolean parameter by which user specifies whether or not he wishes to close with the same end of file.	Ignored by this subsystem.	Ignored

Pascal Calling Sequence:

```
PROCEDURE IPC$CLOSE (    My_fid          : FID;
                         Close_With_EOF  : BOOLEAN);
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame (records are passed by address):

```
        My Fid                          at 0
        Close With EOF                  at 2


    MOV   *LF,*SP+          PASSING FID
    CLR   *SP+              CLOSE WITH EOF PARAMETER IGNORED
    DATA  CALL$
    DATA  IPC$CL
```

4.4.9 IPC$DISCONNECT

When a process no longer needs access to a particular path (and the associated file is closed), IPC$DISCONNECT should be called to deallocate the data structures associated with the file and disconnect the process from the path. If no more processes remain connected to the path, the data structures associated with the path will also be deallocated.

The only parameter passed to IPC$DISCONNECT is the FID passed by address on input.

Parameter	Definition	Limits	Input/Output
File Identifier	Value returned by IPC$-CONNECT enabling the IPC Subsystem to relate a specific file to a specific user.	Integer (set to nil on return to caller).	Input

```
PROCEDURE IPC$DISCONNECT ( VAR My_Fid : FID );
```

Assembly Language Calling Sequence:

Assume: The parameters are stored at the following displacements into the calling procedure's local frame:

```
        My Fid                          at 0

    MOV   *LF,*SP+          PASSING FID
    DATA  CALL$
    DATA  IPC$DI
```

4.5 IPC SYNCHRONIZATION

This subsection details special interactions among processes calling certain IPC routines.

4.5.1 IPC$CREATE/IPC$OPEN Interaction

A path for communicating among processes is established by the first process to call IPC$CONNECT with a given pathname. Each process must call IPC$OPEN to begin communicating over the path. All processes calling IPC$OPEN for a path are suspended until some process calls IPC$CREATE to specify the characteristics of the path; all calls to IPC$CREATE for a pathname that already exists will be ignored.

4.5.2 IPC$OPEN/IPC$CLOSE Interaction

Before a file (FID) can be reopened, it must be closed. Following a close operation on a file, it can be reopened in either read or write mode. It is possible for a process to read or write from a file, close the file, and reopen it in a write or read mode.

Suppose a path has several producers and consumers connected to it and is in an open state. If all producers close, then the consumers will receive an end-of-file status once all buffered data has been consumed. To clear this end-of-file status, each consumer must acknowledge its receipt by entering a closed state. Any process attempting to open a file connected to a path at end-of-transmission will be suspended until all consumers have closed.

If all consumers connected to a path enter a closed state, producers are not required to close. In general, they will become suspended due unconsumed buffers of data and will not be able to proceed until a consumer opens and begins processing buffers.

If a file that is consuming data is closed before reaching end-of-file, it is possible that some transmissions will be discarded. This will occur if the last producer connected to a path has closed after transmitting data and the last consumer decides to close without processing the data. In such a situation IPC$CLOSE must assume that no other consumer will connect to the path and as a consequence must discard all unreceived data so buffers can be reclaimed.

4.6 USE OF DUMMY SUBSYSTEM ENTRY POINTS

The following file service requests are not meaningful in the IPC Subsystem:

- ABORTIO
- DELETE
- POSITION
- STATUS

Dummy or "NO-OP" routines are provided for these services to conform to I/O Subsystem interface requirements. The dummy entry points are placed in the I/O Service Directory in place of the entry points for the above services.

Refer to Appendix D for more information on implementation of the Dummy Subsystem entry points.

SECTION V

ENCODE AND DECODE ROUTINES

5.1 GENERAL

Encode and decode routines are supplied in the Device Independent File I/O Package to enable the assembly language user to perform data conversions from the internal representation to printable format and conversely from the printable format to internal representation (these routines are executed transparently for the Pascal user in the Microprocessor Pascal Executive). This capability is useful when primitive data is to be printed or data is to be input via a keyboard.

For each of the routines described, the calling parameters generally fit into a convenient template (with some exceptions that are noted in the individual procedure descriptions). This template is presented below. The user should refer to this template to understand the procedure definitions and calling sequences.

Parameter	Definition	Limits	Input/Output
String Pointer	Pointer to the output string array (Encode) and the input string array Decode).	Integer	Input
String Length	The number of bytes in the 'String' parameter.	Integer	Input
Index	The starting position of output field (for Encode routines) or input field (for Decode routines). Upon return, this field will contain the position of the character following the output field (Encode) or following the input field (Decode). This facilitates encoding or decoding multiple numbers into one string. This parameter is always passed by address.	Integer	Input/Output

Status	An integer containing a status message. Upon return, this integer contains a zero if the encoding or decoding was successful; a non-zero status indicates that the input parameters are contradictory or the result will not fit into the specified output field. This parameter is always passed by address.	Integer	Output
Input Data or Result	The address of the data being encoded or the result of a decode (passed by address)	Integer	Input
Width	The width in bytes of the output field in Encode and of the input field in Decode.	Integer	Input

5.2 ENCODE ROUTINES

Encoding is the process of converting from an internal format to a character string. Routines accomplishing this are defined below.

5.2.1 Encoding an Integer (ENC$IN)

ENC$IN is used to convert from an integer to character format. One additional parameter is passed to ENC$IN: Hex which is a Boolean value. If Hex is True, a hexadecimal value is generated. When Hex is False, a decimal results.

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Input Data Address	at 8
Width	at 10
Hex	at 12

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING STRING LENGTH
      MOV   LF,*SP                PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING INPUT DATA ADDRESS
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING WIDTH
      MOV   @12(LF),*SP+          PASSING HEX
      DATA  CALL$
      DATA  ENC$IN
```

where the following sequence is in the user's code:

```
MOD   EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.2.2 Encoding a Longint (ENC$LO)

ENC$LO is called to convert from an extended integer to character format. As in ENC$IN (above) a Hex parameter is passed to ENC$LO indicating if the result is to be hex (parameter value is True) or decimal (parameter value is false).

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

```
          String Pointer          at 0
          String Length           at 2
          Index                   at 4
          Status                  at 6
          Input Data Address      at 8
          Width                   at 10
          Hex                     at 12

    MOV   *LF,*SP+              PASSING STRING  POINTER
    MOV   @2(LF),*SP+           PASSING MAXIMUM NUMBER
    MOV   LF,*SP                PASSING INDEX
    A     @FOUR(LF),*SP+
    MOV   LF,*SP                PASSING STATUS
    A     @SIX(CODE),*SP+
    MOV   LF,*SP                PASSING INPUT DATA ADDRESS
    A     @EIGHT(CODE),*SP+
    MOV   @10(LF),*SP+          PASSING WIDTH
    MOV   @12(LF),*SP+          PASSING HEX
    DATA  CALL$
    DATA  ENC$LO
```

where the following sequence is in the user's prologue:

```
MOD   EQU   $               MOD LABLES BEGINNING OF LOCAL DATA
      DATA PRO-MOD          PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU  $-MOD
      DATA 4
SIX   EQU  $-MOD
      DATA 6
EIGHT EQU  $-MOD
      DATA 8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.2.3 Encoding Boolean (ENC$BO)

The ENC$BO routine is called to convert from the internal Boolean to character format. If the byte width of the output field is less than five, then TRUE is encoded as "T" and FALSE as "F"; otherwise, TRUE and FALSE are spelled out.

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Input Data Address	at 8
Width	at 10

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING STRING LENGTH
      MOV   LF,*SP                PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING INPUT DATA ADDRESS
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING WIDTH
      DATA  CALL$
      DATA  ENC$BO
```

where the following sequence is in the user's prologue:

```
MOD   EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD        PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.2.4 Encoding a Character (ENC$CR)

ENC$CR is called to store a single character (padded with blanks on the left) in a string. The character is right justified in the output field.

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Input Data Address	at 8
Width	at 10

```
MOV    *LF,*SP+              PASSING STRING POINTER
MOV    @2(LF),*SP+           PASSING STRING LENGTH
MOV    LF,*SP                PASSING INDEX
A      @FOUR(LF),*SP+
MOV    LF,*SP                PASSING STATUS
A      @SIX(CODE),*SP+
MOV    LF,*SP                PASSING INPUT DATA ADDRESS
A      @EIGHT(CODE),*SP+
MOV    @10(LF),*SP+          PASSING WIDTH
DATA   CALL$
DATA   ENC$CR
```

where the following sequence is in the prologue of the user's code:

```
MOD   EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD        PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.2.5 Encoding a String (ENC$ST)

The routine ENC$ST is called to store a character string in a field within another character string. One additional parameter passed to this routine is the width in bytes of the input field. The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Input Data Address	at 8
Input Width	at 10
Output Width	at 12

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING STRING LENGTH
      MOV   LF,*SP                PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING INPUT DATA ADDRESS
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING INPUT WIDTH
      MOV   @12(LF),*SP+          PASSING OUTPUT WIDTH
      DATA  CALL$
      DATA  ENC$ST
```

where the following sequence is in the user's prologue:

```
MOD   EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD        PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus

one byte exceeds String Length.

5.2.6 Encoding a Real (ENC$RE)

ENC$RE is called to convert from the internal representation of a real to its corresponding character format. One additional parameter is passed to this routine. This parameter, F, represents the number of digits falling to the right of the decimal. If F < 0, then the output is in floating point format. To generate output in fixed point format, set F to the number of digits to the right of the decimal point (i.e., number of decimal places). Otherwise, set F less than 0 for floating point format.

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Input Data Address	at 8
Output Width	at 10
F	at 12

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING STRING LENGTH
      MOV   LF,*SP                PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING INPUT DATA
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING OUTPUT WIDTH
      MOV   @12(LF),*SP+          PASSING F
      DATA  CALL$
      DATA  ENC$RE
```

where the following sequence is in the user's prologue:

```
MOD   EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
      DATA PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU  $-MOD
      DATA 4
SIX   EQU  $-MOD
      DATA 6
EIGHT EQU  $-MOD
      DATA 8
```

EXCEPTIONS AND CONDITIONS:

Any error in the parameters will result in status being set to 1. In this event, the output field will be set to all asterisks ('***...').

5.3 DECODE ROUTINES

Decoding is the process of converting from a character string to an internal format.

5.3.1 Decoding an Integer (DEC$IN)

DEC$IN is called to convert a field in a character string to an integer. If the number is preceded by a '#', it is interpreted as a hexadecimal number, otherwise decimal is assumed. The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Input Width	at 10

```
      MOV    *LF,*SP+              PASSING STRING POINTER
      MOV    @2(LF),*SP+           PASSING STRING LENGTH
      MOV    LF,*SP                PASSING INDEX
      A      @FOUR(LF),*SP+
      MOV    LF,*SP                PASSING STATUS
      A      @SIX(CODE),*SP+
      MOV    LF,*SP                PASSING RESULT ADDRESS
      A      @EIGHT(CODE),*SP+
      MOV    @10(LF),*SP+          PASSING INPUT WIDTH
      DATA   CALL$
      DATA   DEC$IN
```

where the following sequence is in the user's code:

```
MOD   EQU   $              MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD        PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

(3) Incomplete Data. An example is a plus sign without digits following.

(4) Invalid character in field. This happens when a non-numeric character is found in a number.

(5) Data value too large. This occurs when a number is too large to be stored in the given variable (e.g., 32768 is an integer).

5.3.2 Decoding a Longint (DEC$LO)

The procedure DEC$LO is called to convert a field in a character string to an extended integer. If the number is preceded by a '#', it is interpreted as a hexadecimal number, otherwise decimal is assumed. The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Input Width	at 10

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING STRING LENGTH
      MOV   LF,*SP                PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING ADDRESS OF RESULT
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING INPUT WIDTH
      DATA  CALL$
      DATA  DEC$LO
```

where the following sequence is in the user's code:

```
MOD   EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

(3) Incomplete Data. An example is a plus sign without digits following.

(4) Invalid character in field. This happens when a non-numeric character is found in a number.

(5) Data value too large. This occurs when a number is too large to be stored in the given variable (e.g., decoding a number larger than #7FFFFFFF).

5.3.3 Decoding Boolean (DEC$BO)

The procedure DEC$BO is called to convert a field in a Boolean character string to an integer. Valid boolean strings are 'T', 'TRUE', 'F', and 'FALSE'. No conversion of lower to upper case is done. The assembly language calling sequence follows.

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Input Width	at 10

```
      MOV   *LF,*SP+              PASSING STRING POINTER
      MOV   @2(LF),*SP+           PASSING MAXIMUM NUMBER
      MOV   LF,*SP                PASSING NUMBER
      A     @FOUR(LF),*SP+
      MOV   LF,*SP                PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP                PASSING ADDRESS OF RESULT
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+          PASSING INPUT WIDTH
      DATA  CALL$
      DATA  DEC$BO
```

where the following sequence is in the user's code:

```
MOD   EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

(3) Invalid character in field. This happens when an invalid separator is found.

5.3.4 Decoding a Character (DEC$CH)

The procedure DEC$CH is called to convert a field in a character

string to an integer. In passing the byte width of the input field (W), if W > 0, the first non-blank character in the next W characters is returned. If the field is all blanks, a blank is returned. If W = 0, a blank is returned. If W < 0, the field width is assumed to be 1 (i.e. the next character is returned, blank or not).

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Input Width	at 10

```
      MOV   *LF,*SP+            PASSING STRING POINTER
      MOV   @2(LF),*SP+         PASSING STRING LENGTH
      MOV   LF,*SP              PASSING INDEX
      A     @FOUR(LF),*SP+
      MOV   LF,*SP              PASSING STATUS
      A     @SIX(CODE),*SP+
      MOV   LF,*SP              PASSING ADDRESS OF RESULT
      A     @EIGHT(CODE),*SP+
      MOV   @10(LF),*SP+        PASSING INPUT WIDTH
      DATA  CALL$
      DATA  DEC$CH
```

where the following sequence is in the user's code:

```
MOD   EQU   $             MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD       PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
EIGHT EQU   $-MOD
      DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.3.5 Decoding a String (DEC$ST)

DEC$ST is called to move a field in a character string to another character string. One additional parameter is passed to DEC$ST: the length of the result string.

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Output Length	at 10
Input Field Width	at 12

```
MOV    *LF,*SP+              PASSING STRING POINTER
MOV    @2(LF),*SP+           PASSING STRING LENGTH
MOV    LF,*SP                PASSING INDEX
A      @FOUR(LF),*SP+
MOV    LF,*SP                PASSING STATUS
A      @SIX(CODE),*SP+
MOV    LF,*SP                PASSING RESULT ADDRESS
A      @EIGHT(CODE),*SP+
MOV    @10(LF),*SP+          PASSING OUTPUT LENGTH
MOV    @12(LF),*SP+          PASSING INPUT WIDTH
DATA   CALL$
DATA   DEC$ST
```

where the following sequence is in the user's code:

```
MOD    EQU   $               MOD LABELS BEGINNING OF LOCAL DATA
       DATA  PRO-MOD         PRO LABELS BEGINNING OF EXECUTABLE CODE
         .
         .
FOUR   EQU   $-MOD
       DATA  4
SIX    EQU   $-MOD
       DATA  6
EIGHT  EQU   $-MOD
       DATA  8
```

EXCEPTIONS AND CONDITIONS:

Possible errors resulting (by error code):

(1) Bad parameter passed to routine. An example is the Index parameter exceeding the parameter for String Length.

(2) Field width too large. This occurs when Index plus Width minus one byte exceeds String Length.

5.3.6 Decoding a Real (DEC$RE)

The DEC$RE routine is called to convert from a character string providing the printable representation of a Real number to its internal floating point format.

The assembly language calling sequence follows:

Assume: The parameters for this procedure are stored at the following displacements into the calling procedure's local frame:

String Pointer	at 0
String Length	at 2
Index	at 4
Status	at 6
Address of Result	at 8
Width	at 10

```
MOV    *LF,*SP+              PASSING STRING POINTER
MOV    @2(LF),*SP+           PASSING STRING LENGTH
MOV    LF,*SP                PASSING INDEX
A      @FOUR(LF),*SP+
MOV    LF,*SP                PASSING STATUS
A      @SIX(CODE),*SP+
MOV    LF,*SP                PASSING RESULT ADDRESS
A      @EIGHT(CODE),*SP+
MOV    @10(LF),*SP+          PASSING WIDTH
DATA   CALL$
DATA   DEC$RE
```

where the following sequence is in the user's code:

```
MOD   EQU   $                MOD LABELS BEGINNING OF LOCAL DATA
      DATA  PRO-MOD          PRO LABELS BEGINNING OF EXECUTABLE CODE
        .
        .
FOUR  EQU   $-MOD
      DATA  4
SIX   EQU   $-MOD
      DATA  6
```

EXCEPTIONS AND CONDITIONS:

If the input parameters are contradictory, the status will be set to one.

If the field specified is not contained in the array (i.e., the field width is too large) the status is set to two.

If the field does not contain a valid real number, the status is set to three.

SECTION VI

CONFIGURING AN APPLICATION TO INCLUDE DIF I/O ROUTINES

6.1 GENERAL

The paragraphs that follow provide information on initializing and configuring an application containing the File I/O Decoder and various I/O Subsystems. The main points presented include a description of system initialization, detail on the various object modules used to build the target application, and an overview of the link editing process. The default version of the GHOST$ process, a sample CONFIG module, and an example Link Edit Control File are also presented.

The link editor present in the user's development system provides the means for generating the target application (or load module). The user specifies a link edit control file as input for the link editor. The link editor resolves all of the application's external references via the libraries specified in the link edit control file.

Detailed information regarding configuring a load module for native code execution is presented in the Realtime Executive User's Manual (MP373) for the assembly language user and the Microprocessor Pascal Executive User's Manual (MP385) for the Pascal user.

6.2 INITIALIZATION

Initialization of applications configured with one or more I/O Subsystems and the File I/O Decoder takes place automatically at power up time. The GHOST$ process supplied by the run-time support contains a call to D$INIT, the entry point for initialization present in the File I/O Decoder. In turn, each of the I/O Subsystems present on the target system is initialized via D$INIT. If the user wishes to activate the File I/O Decoder and the suported I/O Subsystems directly from his application, he can remove the call to D$INIT from GHOST$.

In addition, GHOST$ contains a call to MSG$INIT which identifies the name of the device acting as the destination of the standard procedure MESSAGE. The statement:

```
MESSAGE( 'Execution begins.' );
```

can be inserted after the START statement in the user application to signal the "Operator" (specified in the default version of GHOST$) that execution has begun. To implement this call, the node name "Operator" must be present in a Port Contstants Record associated with some I/O Subsystem on the target ("Operator" is the node name assigned to a device in a Port Constants Record in the Operator Interface I/O Subsystem--see Appendix A). If this node name is not so specified, its reference must be removed from GHOST$. Note also that IPC Subsystem will always claim the name 'Operator' if that name is not claimed by

any other subsystem.

For most applications the default version of GHOST$ will be adequate. If certain initialization must be performed for a class of applications (e.g. special devices must be initialized), it is appropriate that it be performed in the ghost procedure so it need not be repeated in each application. If it is known that the File I/O Decoder will not be used, then a slight savings in code space can be made by removing the calls to D$INIT in GHOST$. If the standard procedure MESSAGE will not be used, the call to MSG$INIT can also be removed from GHOST$. If the File I/O Decoder is not specified at link edit time, D$INIT will be resolved by a "dummy" routine (present in the Rx Sequential Library RXOBJ) that performs no processing.

The default version of GHOST$ is displayed below (the source for GHOST$ is written in assembly language).

```
system ghost$system;

const
  dont_care = 2;

type
  dummy_buffer = packed array[ 1..dont_care ] of char;

procedure d$init; external;

procedure msg$init( var pathname: dummy_buffer; length: integer );
  external;

program systm$; external;

procedure ghost$;
var
  pathname: packed array[ 1..8 ] of char;
begin
  d$init;
  pathname := 'OPERATOR';
  msg$init( pathname::dummy_buffer, size( pathname ) );
  start systm$;
end { ghost$ };

begin
  {$ nullbody }
end.
```

FIGURE 6-1. DEFAULT VERSION OF PROCEDURE GHOST$

6.3 CONFIGURATION MODULES

The object modules required to configure an application with the File I/O Decoder and the I/O Subsystems provided in MPX and in the Device

Independent File I/O Package are described below.

6.3.1 DIF I/O Routines

The File I/O Decoder, the Operator Interface I/O Subsystem, the Interprocess Communication I/O Subsystem, and the Encode and Decode routines are packaged as sequential libraries as described below. These libraries are supplied for the Pascal user in the Microprocessor Pascal Executive and for the assembly language user in the Device Independent File I/O Package.

- D$OBJ containing object modules to support the File I/O Decoder level of device-independent I/O (described in Section III). This library also contains the dummy I/O Subsystem (described in Appendix D) used when a specific file service is not supported on the target.

- IPC$OBJ containing object modules comprising the Interprocess Communication I/O Subsystem (described in Section IV). IPC$OBJ uses routines from the libraries C$OBJ (supplied in MPX or Rx) and D$OBJ. Because IPC$OBJ accepts any pathname passed at Connect time, it should be the last I/O Subsystem referenced in the I/O Subsystem Service Directory (IODIR) specified in CONFIG (see Subsection 6.3.3 below).

- T02$OBJ containing object modules comprising the Operator Interface I/O Subsystem (described in Appendix A). These routines support communication with a variety of terminals connected to a 9902 interface. Routines from the libraries C$OBJ and D$OBJ are also required by the T02$OBJ library.

- DE$OBJ containing object modules that implement Decode and Encode routines (described in Section V).

NOTE: The run-time support library MPP$OBJ providing data types routines is supplied to the assembly language programmer in the DIF I/O package. The Pascal user can find this library in the Microprocessor Pascal Executive.

6.3.2 The Executive Library

The libraries providing native code run-time support consist of the sequential library RX$OBJ and the random library RX$LIB, each containing miscellaneous Rx routines; and the sequential libraries COBJ, CLKOBJ, and MPP$OBJ containing channel routines, clock routines, and Data Types routines respectively. With the exception of CLK$OBJ, the above run-time support libraries are required in most aplications utilizing DIF I/O software. The Microprocessor Pascal Executive supplies these libraries to the Pascal user. The assembly language user obtains all native code libraries except for the library

MPP$OBJ from the Realtime Executive (Rx). MPP$OBJ is supplied to the assembly language programmer in the DIF I/O package.

6.3.3 CONFIG

CONFIG is provided in the native code run-time support (MPX and RX). The default version of this module must be customized to fit the user's application. Information regarding this module is contained in the Microprocessor Pascal Executive User's Manual and in the Realtime Executive User's Manual. The Information below describes data required in CONFIG when the load module will contain DIF I/O routines.

6.3.3.1 Specification of the I/O Service Directory. The default version of CONFIG provides for the specification of the I/O Service Directory used during system initialization at power up time. In this directory, the user specifies the address of an I/O Subsystem Directory and the address of an initial Port Constants Record for each I/O Subsystem supported on the target. The I/O Subsystem Directory contains the entry points for the routines making up the I/O Subsystem. These entry points are standard from I/O Subsystem to I/O Subsystem (see Subsection 2.4.2 for information on the derivation of entry point names and Appendix B for a picture of the I/O Subsystem Directory). The Port Constants Record contains fixed data describing the port associated with an I/O Subsystem. The port provides the logical connection between the I/O Subsystem and the CPU; in many I/O Subsystems the port is associated with some device controller (see Appendix B for more information regarding the Port Constants Record). While it is not required, the Port Constants Record(s) and the Node Constants Record it points to can be conveniently placed in the CONFIG module.

The end of the I/O Directory is marked by a null entry.

Sample code used in the I/O Directory in CONFIG is presented below. In the example, entries are present for two I/O Subsystems: the Operator Interface (T02) I/O Subsystem described in Appendix A and the Interprocess Communication (IPC) I/O Subsystem described in Section IV. Note that the pointer to the Port Constants Record for the Interprocess Communication I/O Subsystem is set to nil. The sample CONFIG module below also contains the Port Constants Record and Node Constants Records required by the Operator Interface I/O Subsystem.

```
IODIR   EQU   $                 I/O DIRECTORY
*
        REF   T02$SD,
        DATA  T02$SD,T02$PC     T02 SERVICE DIR. AND PORT CONSTANTS REC.
*
        REF   IPC$SD            IPC SERVICE DIRECTORY
        DATA  IPC$SD,0
*
        DATA  0                 LIST TERMINATOR
```

6.3.3.2 Example CONFIG. An example CONFIG is presented below. Two I/O Subsystems are specified in the I/O Service Directory: the Operator Interface I/O Subsystem and the Interprocess Communication I/O Subsystem.

```
        IDT  'CONFIG'              SPECIFY CONFIGURATION
* REVISION: 08/01/80  1.00        ORIGINAL FOR RX 2.0
* ROUTINE LIST: CONFIG, IWP$0 .. IWP$15, BAD$WP,
*                $RAMTB, $RESTA, $LREX, $SYSCR,
*                $DEFAU, $FILL, $STKSZ, $BOOTP,
*                $IODIR, DB$WP
* COPY MODULES:
*       NONE.
* MACRO DEFINITIONS:
*       NONE.
* EXTERNAL ROUTINES:
*       NONE.
* EXTERNAL DATA:
        PSEG
* MODULE CONSTANTS:
IWPSZ   EQU  24                  EXAMPLE SIZE OF AN INTERRUPT
*                                  WORKSPACE (R4-R15)
LOWRAM  EQU  >8000               LOW BOUNDARY OF RAM
* MODULE VARIABLES:
*
        DORG LOWRAM
*
        DEF  IWP$0,IWP$1,IWP$2,IWP$3
        DEF  IWP$4,IWP$5,IWP$6,IWP$7
        DEF  IWP$8,IWP$9,IWP$10,IWP$11
        DEF  IWP$12,IWP$13,IWP$14,IWP$15
        DEF  BAD$WP,DB$WP
IWP$0   BSS  32
IWP$1   BSS  32
DB$WP   EQU  IWP$1
IWP$2   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$3   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$4   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$5   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$6   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$7   EQU  $-32+IWPSZ
        BSS  IWPSZ
IWP$8   EQU  $-32+IWPSZ
        BSS  IWPSZ
```

FIGURE 6-2. CONFIG (Sheet 1 of 5)

```
IWP$9   EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$10  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$11  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$12  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$13  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$14  EQU   $-32+IWPSZ
        BSS   IWPSZ
IWP$15  EQU   $-32+IWPSZ
        BSS   IWPSZ
BAD$WP  BSS   32
*
LOWHP   EQU   $
*
        RORG
        TITL  'CONFIG:          SPECIFY CONFIGURATION'
        PAGE
* ABSTRACT:
*       SPECIFY CERTAIN SYSTEM PARAMETERS, THE RAM
*       CONFIGURATION, AND THE I/O SUBSYSTEM
*       DIRECTORY.
* CALLING SEQUENCE:
*       NONE.
* EXCEPTIONS AND CONDITIONS:
*       NONE.
* LOCAL DATA:
*       NONE.
* ENTRY POINT:
*       NONE.
*****************************************************************
* ADDRESS OF THE "BLWP" VECTOR FOR RESTARTS; USE "0" FOR
* LEVEL 0 INTERRUPT, ">FFFC" FOR THE "LREX" VECTOR, OR
* THE ADDRESS OF A USER-DEFINED VECTOR.
*****************************************************************
        DEF   $RESTA
$RESTA  DATA  0
```

FIGURE 6-2. CONFIG (Sheet 2 of 5)

```
**************************************************************
* ADDRESS OF THE "BLWP" VECTOR FOR THE "LREX" INSTRUCTION;
* USE "0" IF THERE IS TO BE NO "LREX" VECTOR OR IF HIGH
* MEMORY IS ROM.
**************************************************************
       DEF  $LREX
$LREX  DATA 0
**************************************************************
* ADDRESS OF THE USER-DEFINED ROUTINE TO BE INVOKED IN CASE
* OF A SYSTEM CRASH; USE "0" FOR THE SYSTEM DEFAULT WHICH
* IS TO MASK INTERRUPTS AND IDLE THE PROCESSOR.
**************************************************************
       DEF  $SYSCR
$SYSCR DATA 0
**************************************************************
* ADDRESS OF THE MPP ROUTINE TO BE INVOKED IF AN EXCEPTION
* OCCURS BUT NO EXCEPTION HANDLER HAS BEEN SPECIFIED; USE
* "0" FOR THE SYSTEM DEFAULT WHICH IS A "NO EXCEPTION
* HANDLER" SYSTEM CRASH.
**************************************************************
       DEF  $DEFAU
$DEFAU DATA 0
**************************************************************
* THIS IS THE VALUE WITH WHICH THE HEAP WILL BE
* INITIALIZED AT POWER-UP.
**************************************************************
       DEF  $FILL
$FILL  JMP  $
**************************************************************
* THIS IS THE DEFAULT STACK SIZE (IN WORDS) THAT IS USED
* IF A "STACKSIZE" CONCURRENT PARAMETER IS NOT SPECIFIED.
**************************************************************
       DEF  $STKSZ
$STKSZ DATA >100
**************************************************************
* THE PARAMETER LIST FOR THE CALL TO "S$PRCS" TO START THE
* "BOOT" PROGRAM.
**************************************************************
       DEF  $BOOTP
$BOOTP DATA >0000            FRAME SIZE
       DATA >0000            LEXICAL NESTING LEVEL
       DATA >0000            PRIORITY
       DATA >0100            STACK SIZE
       DATA >0000            HEAP SIZE
```

FIGURE 6-2. CONFIG (Sheet 3 of 5)

```
*************************************************************
* ADDRESS OF THE "RAM TABLE," THE TABLE THAT DESCRIBES THE
* REGIONS OF READ-WRITE MEMORY TO BE COLLECTED INTO THE
* HEAP.
*************************************************************
       DEF  $RAMTB
$RAMTB DATA RAMTB
*************************************************************
* ADDRESS OF THE DIRECTORY OF I/O SUBSYSTEMS.
*************************************************************
       DEF  $IODIR
$IODIR DATA IODIR
*************************************************************
* THE FOLLOWING TABLE IS A LIST OF "LENGTH_IN_BYTES,
* STARTING_ADDRESS" PAIRS THAT DEFINE THE RAM TO BE USED
* BY THE EXECUTIVE; A WORD OF "0" TERMINATES THE LIST.
* THE RAM REGIONS MUST BE IN ASCENDING ORDER AND MUST NOT
* OVERLAP.
*************************************************************
RAMTB  DATA >FFFE-LOWHP,LOWHP
       DATA 0                    LIST TERMINATOR
*************************************************************
* THE FOLLOWING TABLE IS A LIST OF "SERVICE_DIRECTORY,
* PORT_CONSTANTS" PAIRS THAT DEFINE THE I/O SUBSYSTEM TO
* BE INITIALIZED WHEN ROUTINE "D$INIT" IS CALLED;
* A WORD OF "0" TERMINATES THE LIST.
*************************************************************
IODIR  EQU  $
*
       REF  T02$SD,T02$PC
       DATA T02$SD,T02$PC    T02 SERVICE DIR. AND PORT CONSTANTS REC.
*
       REF  IPC$SD           IPC SERVICE DIRECTORY
       DATA IPC$SD,0
*
       DATA 0                LIST TERMINATOR
*
```

FIGURE 6-2. CONFIG (Sheet 4 of 5)

```
*****************************************************************
* THE FOLLOWING IS A PORT CONSTANTS RECORD FOR THE
* OPERATOR INTERFACE I/O SUBSYSTEM
*****************************************************************

  T02$PC EQU  $
  *
         DATA 0         LINK
         DATA 4         INTERRUPT LEVEL
         DATA >0080     CRU ADDRESS
         DATA 0         BAUD RATE; 0 => ADJUSTABLE
         DATA 0         HEAP SIZE
         DATA 0         INTERFACE HANDLER
         DATA NAME0     PORT NAME
         DATA LNGTH0    PORT NAME LENGTH
         DATA NODE1     NODE HEADER POINTER
  *
  NODE1  DATA NODE2     LINK
         DATA 0         NODE TYPE
         DATA NAME1     NODE NAME
         DATA LNGTH1    NODE NAME LENGTH
         DATA >8001     OPTIONS = ( ECHO, CR/LF AFTER WRITE )
  *
  NODE2  DATA 0         LINK
         DATA 0         NODE TYPE
         DATA NAME2     NODE NAME
         DATA LNGTH2    NODE NAME LENGTH
         DATA >A001     OPTIONS = ( ECHO, CR/LF AFTER READ,
  *                                       CR/LF AFTER  WRITE )
  *
  NAME0  TEXT '9902 AT >080'
  LNGTH0 EQU  $-NAME0
         EVEN
  *
  NAME1  TEXT 'OPERATOR'
  LNGTH1 EQU  $-NAME1
         EVEN
  *
  NAME2  TEXT 'VDT'
  LNGTH2 EQU  $-NAME2
         EVEN
  END
```

FIGURE 6-2. CONFIG (Sheet 5 of 5)

6.4 LINK EDITING

Link editing enables the user to link together the user application, the File I/O Decoder, the desired I/O Subsystems, and required run-time support. The link editor in the user's development system provides the necessary software tools to carry out the configuration process. The link editor requires as input a link edit control file. The paragraphs that follow describe the link editor and link edit control files.

6.4.1 Link Editor

For information on initializing and executing the link editor, refer to the Model 990 Computer Link Editor Reference Manual (949617-9701) or to the 9900 AMPLUS Software System User's Manual (MP375).

6.4.2 Link Edit Control File

The user must create a link edit control file to input to the Link Editor. This file is generated using the text editor and is specified when the link editor is brought up. The link edit control file defines which modules are to be linked into the load module and in which order they are to be linked.

A sample link edit control file is presented below. Detailed information concerning the format and instructions used can be found in the user manuals for the respective link editors.

NOTE: The file names used below are merely examples. The actual file names used may change depending on their user-assigned locations.

```
TASK      SAMPLE         !PROGRAM NAME
LIBRARY   MPX.RX$LIB     !RX RANDOM LIBRARY
INCLUDE   (RXKERNEL)     !RX KERNEL
INCLUDE   <CONFIG>       !USER'S CONFIG
INCLUDE   VOL1.APPL      !USER'S APPLICATION
SEARCH                   !RESOLVE ALL REFERENCES TO HERE
FIND      MPX.DE$OBJ     !ENCODE AND DECODE
FIND      MPX.T02$OBJ    !OPERATOR INTERFACE
FIND      MPX.IPC$OBJ    !INTERPROCESS COMMUNICATION
FIND      MPX.D$OBJ      !FILE I/O DECODER
FIND      MPX.C$OBJ      !CHANNEL ROUTINES
FIND      MPX.MPP$OBJ    !DATA TYPES FROM MPX OR DIF I/O
FIND      MPX.RX$OBJ     !RX SEQUENTIAL LIBRARY
END
```

6-3. SAMPLE LINK EDIT CONTROL FILE

The above example can either be used with a DX or AMPLUS development system. If AMPLUS is used, the drive location (e.g., DS01 or DS02) can be substituted for the volume name.

If the user does not place the tables required by an I/O Subsystem in the CONFIG module, he must create a separate module to contain these records (these records include the Port Constants Records and the Node Constants Record, as well as any other required data. Should he do this, the user must "Include" the name of this module in his Link Edit Control File.

NOTE: Both the IPC and TO2 subsystem sequential object libraries contain service directories. The names of these service directory modules are REF'ed in the example Config (above). These modules will be automatically included in the load module when thelink editor encounters the appropriate REF in the CONFIG module.

APPENDIX A

IMPLEMENTING THE OPERATOR INTERFACE I/O SUBSYSTEM

A.1 GENERAL

This section presents a detailed example of the application of the tools introduced in this manual. Routines are developed to permit interrupt-driven interactions with most terminals that can be connected to a 9902 asynchronous communications controller. The approach presented in building this subsystem may be adopted by a user in the construction of his own subsystem. Subsection A.2 describes fundamental routines that permits low-level interface to a terminal; they manipulate an abstract representation of a terminal (a device record) and may be called from and execute within the user's application process. Subsection A.3 describes an interface handler, a separate process that is implemented with the routines of Subsection A.2. It executes concurrently with user processes and accepts requests for service via message channels. An I/O subsystem is constructed around the interface handler in Subsection A.4 to provide a media-independent collection of I/O services. These services are based on an abtraction called a file ID and are implemented through commands sent to the interface handler.

The software in this section is discussed in terms of excerpts from the source text that is delivered in library MPX.T02$LIB.

A.2 INTERFACE VIA EMBEDDED ROUTINES

This section describes routines with which the user can perform direct I/O to a 9902 at the character or logical record level. The 9902 provides three concurrent functions: transmission and reception of a character via a serial interface and interval timing. In this application the 9902 will be configured to interrupt the host 9900 processor whenever one of these functions completes; since the same interrupt is used for each function, interrupt demultiplexing must be provided. The routines of this package are

H02$RATE	Initialize the 9902 including optional measurement of the transmission rate
H02$OPEN	Allocate and initialize the device descriptor
H02$WAIT	Wait for and demultiplex an interrupt
H02$IN	Read a character
H02$OUT	Write a character
H02$GET	Read a logical record
H02$PUT	Write a logical record

and will be discussed in the following subsections.

Access to each 9902 is made through 32 bits of the CRU address

space. The input bits that will be used are

```
const
  receiver_interrupt = 16;
  transmitter_interrupt = 17;
  timer_interrupt = 19;
```

Each of these bits is set to one when the corresponding function completes. The output bits are

```
const
  request_to_send_on = 16;
  receiver_interrupt_enable = 18;
  transmitter_interrupt_enable = 19;
  timer_interrupt_enable = 20;
```

"Request to send" is used to activate the transmitter. Writing a zero or one to the other bits disables or enables, respectively, interrupts at the completion of the corresponding function; in either case a pending interrupt is cleared. The eight CRU bits beginning at displacement zero are used to move data to and from the 9902. (More information on the 9902 can be found in the TMS9902 Asynchronous Communications Controller Data Manual.)

The interface to a particular 9902 will be represented by the device descriptor shown in Figure A-1. Variable BASE contains the CRU base of the particular 9902 that is associated with the record; RATE is the baud rate at which (both) transmission and reception occur. CHARACTER_SENT and TIMER_ELAPSED are flags that are set by the interrupt demultiplexer to indicate the completion of the corresponding function. KEYBOARD_BUFFER, NEXT_IN, NEXT_OUT, and NUMBER_OF_CHARACTERS comprise a circular buffer into which characters are placed as they are received from the 9902; if the buffer is full when a character arrives, the flag CHARACTER_LOST is set. ATTENTION is the semaphore to which the 9902 interrupt is connected.

```
const
  circular_buffer_size = 16;

type
  device_9902 = record
    base, rate: integer;
    character_sent, timer_elapsed: boolean;
    character_lost: boolean;
    attention: semaphore;
    keyboard_buffer:
      packed array [ 1..circular_buffer_size ] of char;
    next_in, next_out, number_of_characters: integer;
    end;
  device_ptr = @device_9902;
```

FIGURE A-1. 9902 DEVICE DESCRIPTOR

A.2.1 Procedure H02$RATE

This procedure has calling sequence

```
procedure h02$rate( base: integer; var rate: integer )
```

and is used to initialize the 9902 at CRU base BASE. If RATE is not zero and has an acceptable value (110, 300, 600, 1200, 2400, 4800, 9600, or 19200 baud), then both the transmitter and receiver are initialized for that communication rate. Otherwise, the start bit of the first character that is entered is timed, and the transmission rate is calculated; the least significant bit of the first character must be "1" (e.g., a carriage return). The interval timer is set to 16.32 milliseconds.

A.2.2 Procedure H02$OPEN

The first routine in this package that is called must be H02$OPEN (Figure A-2), the routine that initializes the 9902 interface. The device record for the 9902 is allocated and initialized. The parameter LEVEL is used to associate the semaphore ATTENTION with the appropriate interrupt level and enable that interrupt through a TMS9901 programmable systems interface (which is assumed to be at CRU base #0100, as is the case for the TM990 family of computer boards).

```
procedure h02$open( base, level, rate: integer;
                    var d: device_ptr );
const
  base_9901 = #100;
begin
  new( d );
  with dv = d@ do begin
    dv.base := base;
    h02$rate( base, rate );
    dv.rate := rate;
    dv.character_sent := true;
    dv.timer_elapsed := false;
    dv.character_lost := false;
    initsemaphore( dv.attention, 0 );
    externalevent( dv.attention, level );
    dv.next_in := 1;
    dv.next_out := 1;
    dv.number_of_characters := 0;
    crubase( base_9901 );
    sbz( 0 );
    crubase( base_9901 + 2*level );
    sbo( 0 );
    crubase( base );
    sbo( receiver_interrupt_enable );
    end;
end { h02$open };
```

FIGURE A-2. PROCEDURE H02$OPEN

A.2.3 Procedure H02$WAIT

This procedure (Figure A-3) waits until an interrupt is generated by the 9902 which is specified by the device record that is its parameter. The CRU bits TRANSMITTER_INTERRUPT, RECEIVER_INTERRUPT, and TIMER_INTERRUPT are examined to determine which interrupts have occurred. The response for transmitter and timer interrupts is to clear the interrupt and set the appropriate flag to be examined by the caller of H02$WAIT. If a receiver interrupt occurs, a character is read from the 9902 and stored in the keyboard buffer; NUMBER_OF_CHARACTERS is incremented to indicate that keyboard characters are available. (The keyboard is buffered so characters can be entered while output is taking place.)

```
procedure h02$wait( d: device_ptr );
var
  ch: char;
begin
  with d@ do begin
    crubase( base );
    wait( attention );
    if tb( transmitter_interrupt ) then begin
      sbz( transmitter_interrupt_enable );
      character_sent := true;
      end;
    if tb( receiver_interrupt ) then begin
      sbz( receiver_interrupt_enable );
      if number_of_characters < circular_buffer_size then begin
        stcr( 8, ch::integer );
        keyboard_buffer[ next_in ] := ch;
        if next_in = circular_buffer_size then
          next_in := 0;
        next_in := next_in + 1;
        number_of_characters := number_of_characters + 1;
        end
      else
        character_lost := true;
      sbo( receiver_interrupt_enable );
      end;
    if tb( timer_interrupt ) then begin
      sbz( timer_interrupt_enable );
      timer_elapsed := true;
      end;
    end;
end { h02$wait };
```

FIGURE A-3. PROCEDURE H02$WAIT

A.2.4 Procedure H02$IN

This function (Figure A-4) returns the next character from the keyboard buffer. Note that H02$WAIT is called if NUMBER_OF_CHARACTERS is zero since the calling program must wait until a character arrives.

```
function h02$in( d: device_ptr ): char ;
begin
  with d@ do begin
    while number_of_characters = 0 do
      h02$wait( d );
    h02$in := keyboard_buffer[ next_out ];
    if next_out = circular_buffer_size then
      next_out := 0;
    next_out := next_out + 1;
    number_of_characters := number_of_characters - 1;
    end;
end { h02$in };
```

Figure A-4. PROCEDURE H02$IN

A.2.5 Procedure H02$OUT

This procedure (Figure A-5) sends a character to 9902 for transmission. If the transmission of the last character has not completed, H02$WAIT is called until the TRANSMITTER INTERRUPT occurs. After the character has been sent to the 9902, the transmission rate is examined. If it is 1200 baud, the output device is assumed to be a mechanical printer and delays are inserted to compensate for movement of the print mechanism of a TI Model 733 terminal. That is, characters are accepted at 1200 baud but printed at 300 baud; a carriage return requires as much time as 23 characters at 1200 baud. If the transmission rate is less than 1200 baud, then a delay is inserted only for a carriage return. Thus the delays per carriage return and per 1200 baud character are the number of 16.32 millisecond intervals required to transmit 23 and 3 characters, respectively, at 1200 baud. (Note that, since the interval timer is free-running, the delay loop begins at 0, not 1, to ensure that the proper number of full intervals is delayed.)

```
const
  delay_per_cr = 12;
"              = 23 char. delay per cr at 1200 baud
"                * 10 bits per character
"                div 1200 bits per sec
"                div .01632 seconds per interval
"                + 1 to round up

  delay_per_1200_baud_character  = 2;
"                                = 3 char. delay per 1200 baud char.
"                                  * 10 bits per character
"                                  div 1200 bits per second
"                                  div .01632 seconds per interval
"                                  + 1 to round up

procedure h02$out( d: device_ptr; ch: char );
var
  delay: integer;
begin
  with d@ do begin
    crubase( base );
    while not character_sent do
      h02$wait( d );
    character_sent := false;
    sbo( request_to_send_on );
    ldcr( 8, ch::integer );
    sbz( request_to_send_on );
    sbo( transmitter_interrupt_enable );
    if rate <= 1200 then
 delay_block: begin
      if ch = cr then
        delay := delay_per_cr
      else
        if rate = 1200 then
          delay := delay_per_1200_baud_character
        else escape delay_block;
      while not character_sent do
        h02$wait( d );
      for i := 0 to delay do begin
        sbo( timer_interrupt_enable );
        while not timer_elapsed do
        h02$wait( d );
        timer_elapsed := false;
        end;
      end;
    end;
end { h02$out };
```

FIGURE A-5. PROCEDURE H02$OUT

A.2.6 Procedure H02$GET

This procedure (Figure A-6) permits a logical record (terminated by

a carriage return) to be read from the 9902. The option parameter OPTIONS controls whether the record will be echoed as it is entered and whether a carriage return / line feed sequence will be emitted before the record, after the record, or not at all. If the first character that is entered is a DC3 (control-S), then "-1" is returned as the number of characters read (COUNT) to indicate that end-of-file has occurred. Otherwise, up to MAX_LENGTH characters are read into the buffer B. The back space (control-H) may be used to edit a line as it is entered.

```
type
  option_record = packed record
    echo_while_reading: boolean;
    cr_lf_before_read: boolean;
    cr_lf_after_read: boolean;
    cr_lf_before_write: boolean;
    cr_lf_after_write: boolean;
    end;

procedure h02$get( d: device_ptr; b: dummy_buffer_ptr;
                   max_length: integer;
                   options: option_record;
                   var count: integer );
```

FIGURE A-6. PROCEDURE H02$GET (SHEET 1 OF 2)

```
var
  ch: char;
  i: integer;
  echo: boolean;
begin
  with d@ do begin
    ch := h02$in( d );
    if ch = dc3 then count := -1
    else begin
      echo := options.echo_while_reading;
      if echo and options.cr_lf_before_read then begin
        h02$out( d, cr );
        h02$out( d, lf );
        end;
      i := 0;
loop: while true do begin
        i := i + 1;
        if ch = bs then begin
          if echo then h02$out( d, lf );
          repeat
            if i > 1 then begin
              i := i - 1;
              if echo then h02$out( d, bs );
              end;
            ch := h02$in( d );
          until ch <> bs;
          end;
        if ch = cr then begin
          count := i-1;
          escape loop;
          end
        else begin
          if echo then h02$out( d, ch );
          b@[ i ] := ch;
          if i < max_length then ch := h02$in( d )
          else begin
            count := i;
            escape loop;
            end;
          end;
        end;
      if echo and options.cr_lf_after_read then begin
        h02$out( d, cr );
        h02$out( d, lf );
        end;
      end;
    end;
end { h02$get };
```

FIGURE A-6. PROCEDURE H02$GET (SHEET 2 of 2)

A.2.7 Procedure H02$PUT

This procedure (Figure A-7) writes a record to the 9902 interface with carriage control as specified by the option parameter OPTIONS.

```
procedure h02$put( d: device_ptr; b: dummy_buffer_ptr;
                   count: integer; options: option_record );
var
  cmd: command_ptr;
begin
  with d@ do begin
    if options.cr_lf_before_write then begin
      h02$out( d, cr );
      h02$out( d, lf );
      end;
    for i := 1 to count do
      h02$out( d, b@[ i ] );
    if options.cr_lf_after_write then begin
      h02$out( d, cr );
      h02$out( d, lf );
      end;
    end;
end { h02$put };
```

FIGURE A-7. PROCEDURE H02$PUT

A.2.8 An Example

Figure A-8 shows the skeleton of an operator communications program that communicates with the user by reading a command from the same line on which a prompt has been written. The 9902 is configured for the primary port of a TM990/101 board with CRU base of #080 and interrupt level 4; the interface routines will measure the transmission rate. Note that the operator program must have a priority that is consistent with the interrupt level of the 9902 since the program will be waiting on an interrupt semaphore.

```
system example;

type
  buffer = packed array[ 1..80 ] of char;
  buffer_ptr = @buffer;

  device_ptr = @device_ptr;

  option_record = packed record
    echo_while_reading: boolean;
    cr_lf_before_read: boolean;
    cr_lf_after_read: boolean;
    cr_lf_before_write: boolean;
    cr_lf_after_write: boolean;
    end;


procedure h02$open( base, level, rate: integer;
                    var d: device_ptr );
  external;
procedure h02$get( d: device_ptr; b: buffer_ptr;
                   max_length: integer;
                   options: option_record;
                   var count: integer );
  external;
procedure h02$put( d: device_ptr; b: buffer_ptr;
                   count: integer; options: option_record );
  external;
```

FIGURE A-8 AN EXAMPLE (SHEET 1 OF 2)

```
program operator;
var
  d: device_ptr;
  count: integer;
  input_buffer, output_buffer: buffer_ptr;
  options: option_record;
begin
{# priority = 4 }
  h02$open( #080, 4, 0, d );
  new( input_buffer );
  new( output_buffer );
  with options do begin
    echo_while_reading := true;
    cr_lf_before_read := false;
    cr_lf_after_read := false;
    cr_lf_before_write := true;
    cr_lf_after_write := false;
    end;
  while true do begin
    { Fill the output_buffer with a prompt. }
    h02$put( d, output_buffer, 80, options );
    h02$get( d, input_buffer, 80, options, count );
    { Process the command in input_buffer.}
    end;
end { operator };

begin
  { ... }
end { example }.
```

FIGURE A-8. AN EXAMPLE (SHEET 2 OF 2)

A.3 INTERFACE VIA MESSAGE CHANNELS

The previous section presents routines that may be executed within the user's process to communicate with a serial device. The primary advantage of this level of interface is that there is a minimum data space overhead. It is particularly appropriate for applications that have a single user process. If several processes require access to a device, it is desirable to produce an interface handler, a process that services a queue of requests from application processes and communicates with the device that it controls. With this approach it is possible for application processes to overlap computation with input and output.

Figure A-9 shows H02$HANDLER, an interface handler constructed from the routines described in Subsection A.2. BASE, LEVEL, and RATE are parameters to H02$HANDLER which are required by H02$OPEN to initialize a 9902. KEYBOARD and PRINTER are message channels upon which input and output requests, respectively, will arrive. Each request to the handler is a record of type COMMAND. BUFF is a pointer to a buffer of size LENGTH characters. COUNT is set by the

user to indicate the number of characters to be sent and is set by the handler to indicate the number of characters received. OPTIONS is the option record described in Section A.2.6.

The calls to the routine C$NOTIFY cause the device interrupt semaphore to be signaled whenever a command is sent to the KEYBOARD or PRINTER channel, thus simulating an interrupt. Note that the code for H02$WAIT in Section A.2.3 waits for any signal to ATTENTION, whether it be generated by an interrupt or by a software signal. The processing in the handler occurs within an infinite loop that is traversed whenever there is a state change (signal to ATTENTION). If a keyboard or printer command is not pending, C$CRECEIVE is called to accept a command if one has arrived. If a character has been entered (the keyboard buffer is not empty) and a keyboard command is present, the H02$GET is called to input a logical record, and C$ACKNOWLEDGE is called to signal the requestor that his command has been processed. If there is no keyboard activity and a printer command is present, then H02$PUT is called to output a record. If neither keyboard nor printer activity is pending, H02$WAIT is called to await a change of state.

Note that a printer command is not processed unless there is no keyboard activity pending. If a series of printer commands is queued for output and a character is entered at the keyboard, printer activity will be suppended at the end of the current record until the keyboard record has been completely entered. Since H02$GET is not called unless the printer is idle, there is no need to synchronize access to the printer in order to echo keyboard characters. With this implementation characters will not be echoed unless a keyboard request is pending. Note also that H02$WAIT is not called following the processing of a keyboard or printer command unless all tests for pending activity fail. These tests must all be made since it is possible that a change of state (e.g., arrival of a keyboard command) occurred when H02$WAIT had been called by a routine (e.g., H02$OUT) that could not recognize the simulus.

```
type
  command = record
    buff: dummy_buffer_ptr;
    length, count: integer;
    options: option_record;
    end;
  command_ptr = @command;

program h02$handler( base, level, rate: integer;
                     keyboard, printer: cid );
var
  d: device_ptr;
  keyboard_cmd, printer_cmd: command_ptr;
begin
{# stacksize = 100; priority = level }
  h02$open( base, level, rate, d );
  with d@ do begin
    keyboard_cmd := nil;
```

```
      printer_cmd := nil;
      c$notify( keyboard, attention );
      c$notify( printer, attention );
      while true do begin
        if keyboard_cmd = nil then
          c$creceive( keyboard, keyboard_cmd );
        if printer_cmd = nil then
          c$creceive( printer, printer_cmd );
        if number_of_characters > 0 and keyboard_cmd <> nil then begin
          h02$get( d, keyboard_cmd@.buff, keyboard_cmd@.length,
                   keyboard_cmd@.options, keyboard_cmd@.count );
          c$acknowledge( keyboard_cmd );
          keyboard_cmd := nil;
          end
        else
          if printer_cmd <> nil then begin
            h02$put( d, printer_cmd@.buff, printer_cmd@.count,
                     printer_cmd@.options );
            c$acknowledge( printer_cmd );
            printer_cmd := nil;
            end
          else h02$wait( d );
        end;
      end;
  end { h02$handler };
```

Figure A-9. AN INTERFACE HANDLER

A.4 INTERFACE VIA FILE I/O SUBSYSTEM

In this section a file I/O subsystem conforming to the conventions of Section 6.3 is constructed that permits media-independent communication with the interface handler H02$HANDLER that was described in Section 7.3. Of the services that must be provided by an I/O subsystem (Subsection 2.4.2), only INIT, CONNECT, READ, WRITE, WAIT, and DISCONNECT require special versions; the remaining services are provided by entries of the "dummy" subsystem (documented in Appendix D).

Figure A-10 presents the data structures that are device dependent. The command record is the same as that described in Subsection A.3. T02$FID_VARIABLES_RECORD contains the variable data that are associated with each file that is connected to the subsystem. COMMAND is a pointer to the command record that is used to request services for the file. READ_LENGTH_PTR is used to remember the address of the parameter ACTUAL_LENGTH that must be set when a read request completes. OPTIONS contains the formatting options for the file (Subsection A.2.6). T02$NODE_HEADER_RECORD specifies one pathname that will be serviced by this subsystem and the format options for that file. T02$PORT_CONSTANTS_RECORD is standard with the exception of the usage of the I/O address double word; in this application it contains the CRU base and transmission rate of the port. The two fields in T02$PORT_VARIABLES_RECORD contain the

message channel IDs of the keyboard and printer (input and output devices) of the port. (Computing WAITING as #7FFF + 1 yields a one-word constant with value #8000; entering #8000 directly results in a LONGINT constant.)

```
const
  ok = #0000;
  waiting = #7FFF + 1 { = #8000 };

type
  command_record = packed record
    buffer: dummy_buffer_ptr;
    length: integer;
    count: integer;
    options: option_record;
    end;
  command_ptr = @command_record;

  t02$fid_variables_record = record
    { subsystem dependent structure }
    command: command_ptr;
    read_length_ptr: @integer;
    options: option_record;
    end;

  t02$node_header_record = record
    link: node_header_ptr;
    node_type: integer;
    node_name: dummy_buffer_ptr;
    node_name_length: integer;
    { subsystem dependent fields }
    options: option_record;
    end;

  t02$port_constants_record = record
    link: port_constants_ptr;
    interrupt_level: integer;
    base, rate: integer;
    heap_size: integer;
    interface_handler: address;
    port_name: dummy_buffer_ptr;
    port_name_length: integer;
    node_header: node_header_ptr;
    { subsystem dependent fields }
    end;

  t02$port_variables_record = record
    { subsystem dependent structure }
    keyboard: cid;
    printer: cid;
    end;
```

FIGURE A-10. SUBSYSTEM DEPENDENT DATA TYPES

A.4.1 Procedure T02$INIT

This procedure (Figure A-11) initializes the 9902 terminal subsystem. The port variable record is allocated from the heap and is initialized with (unique) channel IDs for the keyboard and printer devices. The interface handler program H02$HANDLER is activated using parameters from the port constants record.

```
procedure t02$init(     serv:      service_directory_ptr;
                        port_cons: port_constants_ptr;
                    var sub:       subsystem_ptr );
var
  port_vars: port_variables_ptr;
begin
  new( port_vars );
  with port_vars@ do begin { initialize port variables }
    c$init( 0, keyboard );
    c$init( 0, printer );
    start h02$handler( port_cons@.base,
                       port_cons@.interrupt_level,
                       port_cons@.rate,
                       keyboard,
                       printer );
    end;
  d$subsystem( serv, port_cons, port_vars, sub );
end { t02$init };
```

FIGURE A-11. PROCEDURE T02$INIT

A.4.2 Procedure T02$CONNECT

This procedure (Figure A-12) is called to determine if a pathname corresponds to a node of this subsystem; utility function EQ$NAMES is used to compare pathnames with node names. If the pathname is recognized, then a file ID variable record is created, and D$FID is called to allocate and initialize a file ID record.

```
procedure t02$connect(     sub:      subsystem_ptr;
                       var pathname: dummy_buffer;
                           length:   integer;
                       var f:        fid );
var
  found: boolean;
  node: node_header_ptr;
  fid_vars: fid_variables_ptr;

function eq$names( var pathname1: dummy_buffer; length1: integer;
                       pathname2: dummy_buffer_ptr; length2: integer ):
  boolean; external;

begin
  node := sub@.port_constants@.node_header;
search:
  repeat
    found := eq$names( pathname, length,
                       node@.node_name,
                       node@.node_name_length );
    if found then escape search
    else node := node@.link;
  until node = nil;
  if found then begin
    new( fid_vars );
    with fid_vars@ do begin { initialize fid_vars }
      c$allocate( size(command_record), command );
      read_length_ptr := nil;
      options := node@.options;
      end;
    d$fid( sub, fid_vars, f );
    f@.status := ok;
    f@.state := closed;
    end
  else f := nil;
end { t02$connect };
```

FIGURE A-12. PROCEDURE T02$CONNECT

A.4.3 Procedure T02$READ

This procedure (Figure A-13) initiates input from the 9902 by sending a keyboard command to the interface handler. Since this procedure does not wait for the input to complete, the address of ACTUAL_LENGTH is saved in the FID variable record so the result can be returned by T02$WAIT.

```
procedure t02$read(   f:              fid;
                      b:              dummy_buffer_ptr;
                      max_length:     integer;
                  var actual_length: integer );
begin
  if d$valid( f, $read ) then begin
    with vars = f@.fid_variables@, cmd = vars.command@ do begin
      cmd.buffer := b;
      cmd.length := max_length;
      cmd.count := 0;
      cmd.options := vars.options;
      c$send( f@.subsystem@.port_variables@.keyboard, vars.command );
      vars.read_length_ptr::address := location( actual_length );
      f@.status := waiting;
      end;
    end;
end { t02$read };
```

Figure A-13. PROCEDURE T02$READ

A.4.4 Procedure T02$WRITE

This procedure (Figure 7-14) initiates output to the 9902 by sending a printer command to the interface handler.

```
procedure t02$write( f:      fid;
                     b:      dummy_buffer_ptr;
                     length: integer );
begin
  if d$valid( f, $write ) then begin
    with vars = f@.fid_variables@, cmd = vars.command@ do begin
      cmd.buffer := b;
      cmd.length := 80;
      cmd.count := length;
      cmd.options := vars.options;
      c$send( f@.subsystem@.port_variables@.printer, vars.command );
      vars.read_length_ptr := nil;
      f@.status := waiting;
      end;
    end;
end { t02$write };
```

FIGURE A-14. PROCEDURE T02$WRITE

A.4.5 Procedure T02$WAIT

This procedure (Figure A-15) waits for a keyboard or printer request to be completed. If READ_LENGTH_PTR is not NIL, then the command was to the keyboard, and the number of characters that were transferred is returned as the result ACTUAL_LENTGH of T2$READ.

```
procedure t02$wait{ f: fid };
begin
  if f@.status < ok then
    with f@.fid_variables@ do begin
      c$wait( command );
      f@.status := ok;
      if read_length_ptr <> nil then begin
        if command@.count < 0 then f@.status := eof_encountered;
        read_length_ptr@ := command@.count;
        read_length_ptr := nil;
        end;
      end;
end { t02$wait };
```

FIGURE A-15. PROCEDURE T02$WAIT

A.4.6 Procedure T02$DISCONNECT

This procedure (Figure A-16) disconnects a file ID by deallocating the command and file ID varaiables records and calling D$RELEASE the release the file ID record.

```
procedure t02$disconnect{ var f: fid };
begin
  if d$valid( f, $disconnect ) then begin
    c$dispose( f@.fid_variables@.command );
    dispose( f@.fid_variables );
    d$fidrelease( f );
    end;
end { t02$disconnect };
```

FIGURE A-16. PROCEDURE T02$DISCONNECT

A.4.7 Module T02$SD

This module (Figure A-17) declares the service directory for the 9902 terminal interface subsystem. Services OPEN, CLOSE, DSTATUS, ABORTIO, CREATE, DELETE, and POSITION are provided by the "dummy" subsystem (Appendix D).

```
        REF   T02$IN,T02$CO,DUM$OP,T02$RE,T02$WR,DUM$CL
        REF   DUM$DS,T02$DI,DUM$AB,DUM$CR,DUM$DE,DUM$PO,T02$WA

T02$SD  EQU   $         *****ENTRY*****
        DEF   T02$SD
*
        DATA  DIRSZ,T02$IN,T02$CO,DUM$OP,T02$RE,T02$WR,DUM$CL
        DATA  DUM$DS,T02$DI,DUM$AB,DUM$CR,DUM$DE,DUM$PO,T02$WA
```

FIGURE A-17. MODULE T02$SD

A.4.8 Module T02$PC

This module (Figure A-18) is a sample port constants structure for the 9902 subsystem. The port is the primary 9902 on a TM990/101 board (CRU base >0080 and interrupt level 4) and the transmission rate will be determined by the first character received. Two nodes are provided, OPERATOR and VDT; they differ only with respect to the carriage control options: OPERATOR permits a prompt to be written to the line from which keyboard input will be read. (OPERATOR is the node to which the default ghost procedure directs the output of the standard procedure MESSAGE.)

```
T02$PC  EQU   $          *****ENTRY*****
        DEF   T02$PC
*
        DATA  0          LINK
        DATA  4          INTERRUPT LEVEL
        DATA  >0080      CRU ADDRESS
        DATA  0          BAUD RATE; 0 => ADJUSTABLE
        DATA  0          HEAP SIZE (CURRENTLY UNUSED)
        DATA  0          INTERFACE HANDLER
        DATA  NAME0      PORT NAME
        DATA  LNGTH0     PORT NAME LENGTH
        DATA  NODE1      NODE HEADER POINTER.  CANNOT BE NIL.
*
NODE1   DATA  NODE2      LINK
        DATA  0          NODE TYPE
        DATA  NAME1      NODE NAME
        DATA  LNGTH1     NODE NAME LENGTH
        DATA  >9000      OPTIONS = ( ECHO, CR/LF BEFORE WRITE )
*
NODE2   DATA  0          LINK
        DATA  0          NODE TYPE
        DATA  NAME2      NODE NAME
        DATA  LNGTH2     NODE NAME LENGTH
        DATA  >D000      OPTIONS = ( ECHO, CR/LF BEFORE READ,
*                                    CR/LF BEFORE WRITE )
*
NAME0   TEXT  '9902 AT >080'
LNGTH0  EQU   $-NAME0
        EVEN
*
NAME1   TEXT  'OPERATOR'
LNGTH1  EQU   $-NAME1
        EVEN
*
NAME2   TEXT  'VDT'
LNGTH2  EQU   $-NAME2
        EVEN
```

FIGURE A-18. MODULE T02$PC

APPENDIX B

INITIALIZATION DATA STRUCTURES

B.1 GENERAL

The information that follows presents the data structures included in the configuration of systems with I/O Subsystem components and used in their initialization. Also presented in this appendix are those data structures associated with the IPC Subsystem described in Section IV.

Prior to presenting this material, the requirements affecting the way in which initialization must work are listed.

B.2 INITIALIZATION REQUIRMENTS

Listed below are the Device Independent File I/O initialization requirements. These requirements affect the data structures used by the device independent file I/O routines.

- I/O Subsystems are members of TI's family of component software (see Subsection 1.4 for a description of TI component software). As such, they may not be bound to a specific system configuration until power up. Therefore the data structures that define the system configuration must be part of the initialization code.

- The call to system initialization can occur at any of the various levels of entry into the I/O model (File I/O Decoder level, I/O Subsystem level, interface handler level). Therefore, the data must be structured into hierarchical levels.

- Many users will require the configuration code to be specified in ROM. Therefore, the data must be partitioned into ROM (for constant and default values), and RAM (for variable or dynamic values).

B.3 INITIALIZATION DATA STRUCTURES

The data structures required for system initialization are described in the subsections that follow. These data structures are constructed and organized to meet the initialization requirments listed above in Subsection B.2.

B.3.1 I/O Service Directory (D$IODIR).

The I/O Service Directory (D$IODIR) is the top-level data structure required for system initialization. This directory contains a two-word

entry for each of the I/O Subsystems in the target system that can be accessed via the File I/O Decoder. The end of this directory is marked by a null entry.

Each word in the two-word entry contains a pointer to a data structure containing information required by its associated I/O Subsystem. The first word points to the service directory for the I/O Subsystem. The second word points to a Port Constants Record (for the first I/O port managed by that subsystem). Information on these data structures is presented in Subsections B.3.2 and B.3.3 respectively.

Figure B-1 depicts the I/O Service Directory.

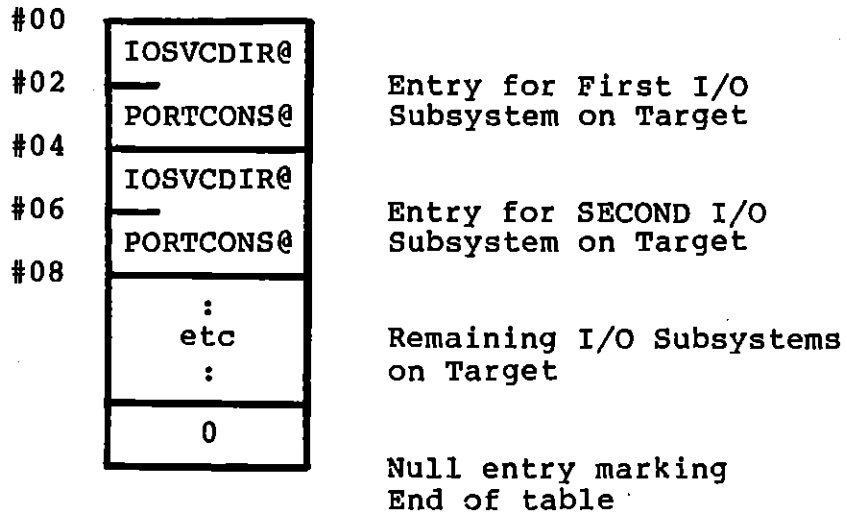


FIGURE B-1. I/O SERVICE DIRECTORY

B.3.2 I/O Subsystem Service Directory (IOSVCDR).

As stated above, the first word in each of the two-word entries making up the I/O Service Directory is a pointer to the I/O Subsystem Service Directory (IOSVCDR). The I/O Subsystem Service Directory contains the entry points to the specific procedures within the I/O Subsystem which must be invoked to perform the file level services requested via the File I/O Decoder (i.e., via calls to the D$routines listed in Subsection 2.3.1). As previously stated, these I/O Subsystem entry points are formed by attaching a prefix (unique to the particular subsystem) to the generic names of the file services (connect, open, read, write, etc.).

The I/O Subsystem Service Directory is normally packaged in one of the libraries supplied with an I/O Subsystem. The directory itself is pulled into the load module at link ;edit ;time

The first entry in the I/O Subsystem Service Directory specifies the

length of the directory. This entry is required because the user has the capability of adding additional entry points as warrented by the I/O Subsystem.

Figure B-2 depicts the I/O Subsystem Service Directory. The entries listed below are the minimum entries that each such directory must contain. (As noted previously, even if a particular I/O Subsystem does not contain a procedure to implement a file level service requested via the File I/O Decoder, it still contains a corresponding entry point. However, in this case, the entry point is associated with a dummy routine. Note that the order of the entry points in the table is fixed and specified by the I/O standards, as shown in the figure below.

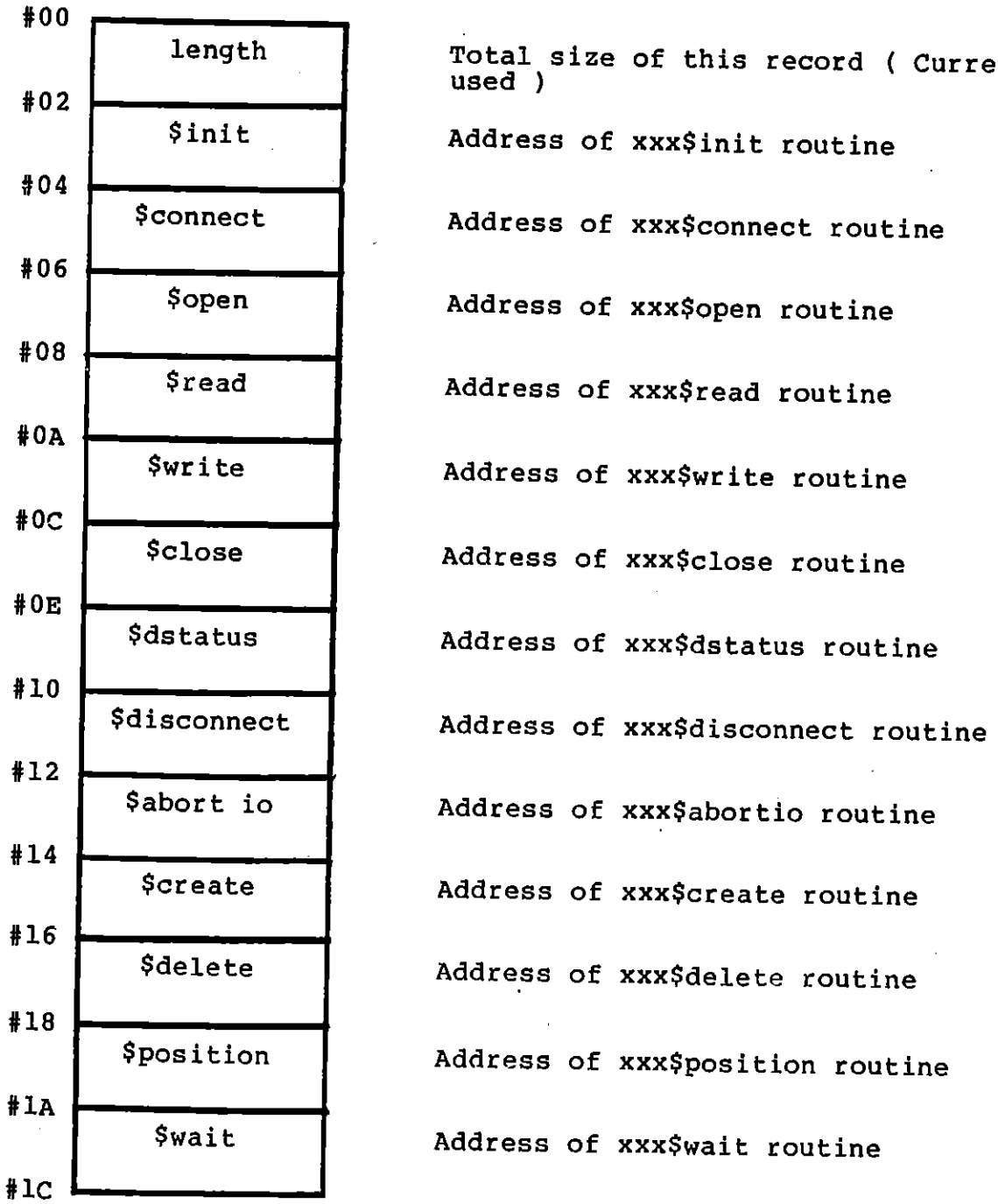


FIGURE B-2. I/O SUBSYSTEM SERVICE DIRECTORY

B.3.3 Port Constant Record (PORTCONS)

The second word in each two-word entry contained in the I/O Service Directory contains a pointer to a Port Contstants Record (PORTCONS). A Port Constants Record contains constant information describing the physical characteristics of an I/O port associated with a given I/O Subsystem (port refers to the connection between the system and the I/O device or node). The Port Constants Record may contain only fixed information because in many cases, Port Constant information will be accessed from read-only memory during normal program execution. The user can build his Port Constants Record in the CONFIG module or can place it in a separate module. If he chooses the latter, the user will have to "include" this module in his link edit control file.

More than one port may be accessible to an I/O Subsystem; a separate Port Constants Record exists for each port. All the Port Constant Records associated with a single subystem are joined together in a forward linked list. The pointer present in the I/O Directory begins the list. Each Port Constants Record in turn points to the Port Constants Record for another associated Port. The last Port Constants Record in the list contains a null pointer.

The required format of the Port Constants Record is displayed below in Figure B-3 (note that the last part of this structure is reserved for I/O Subsystem-dependent information).

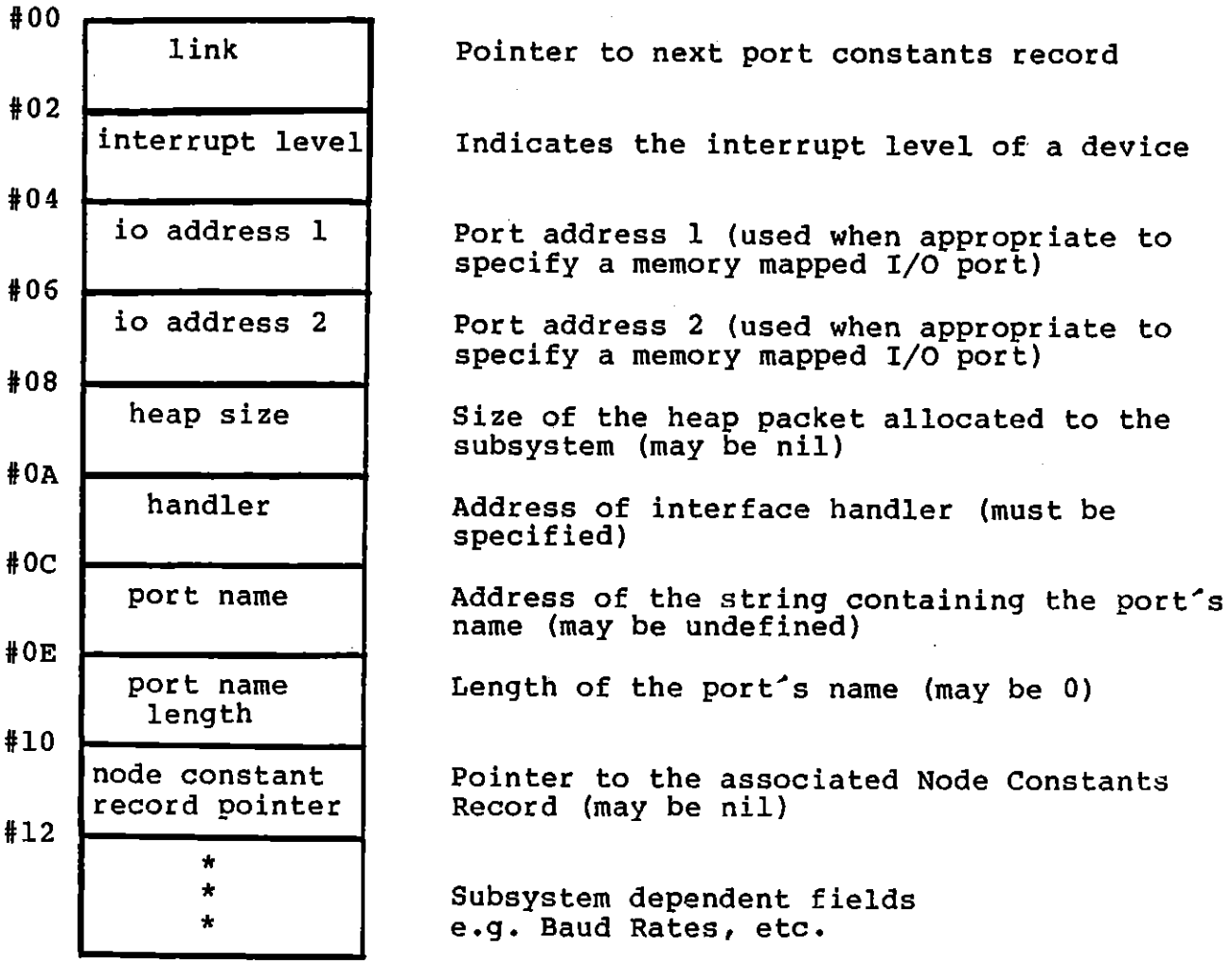


FIGURE B-3. PORT CONSTANTS RECORD

B.3.4 Node Constants Record

The Node Constants Record (also called Node Header Record) provides a desciption of a terminal node accessible through a port. In essence, node refers to the actual physical device (contrast with file which is a logical entity). Each terminal node accessible through the port has a separate Node Constants Record associated with it. A forward link list connects all of the Node Constants Records associated with a given port.

The format of the Node Header Record is presented below in Figure B-4. As for the Port Constants Record, the Node Constants Record must contain only fixed information. As such, the user should build this record in ROM.

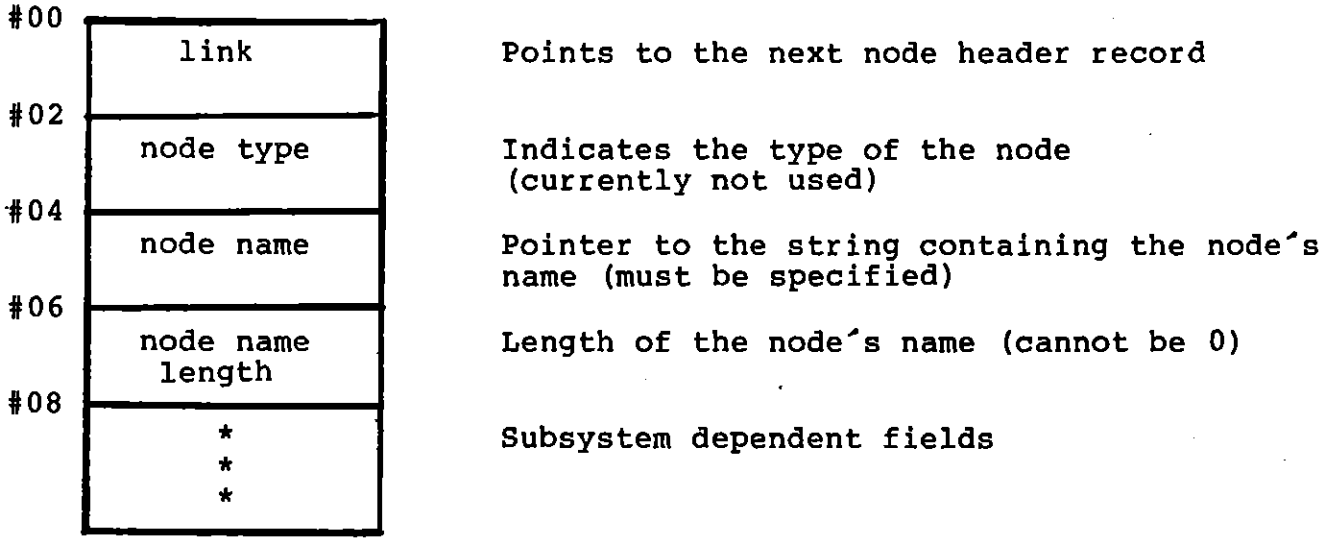


FIGURE B-4. NODE CONSTANTS RECORD

B.3.5 File Identification Record (FIDRCD).

When an I/O Subsystem begins execution, it calls a procedure to allocate memory for and link in a File Identification Record (FIDRCD). The File Identification Record thus created provides for the association of a file (passed to the I/O Subsystem by the Connect procedures) with the device controlled by that I/O Subsystem. In addition, The FIDRCD provides for the return of I/O "status" and file "state" information and associates the specific user with the specific file. Subseqeunt to the "Connect", the FIDRCD is used to identify the file being manipulated to the I/O Subsystem. All FIDRCDs associated with a single process are connected in a forward link list. The pointer on the last FIDRCD is set to nil. The format of the FIDRCD is fixed as displayed below in Figure B-5.

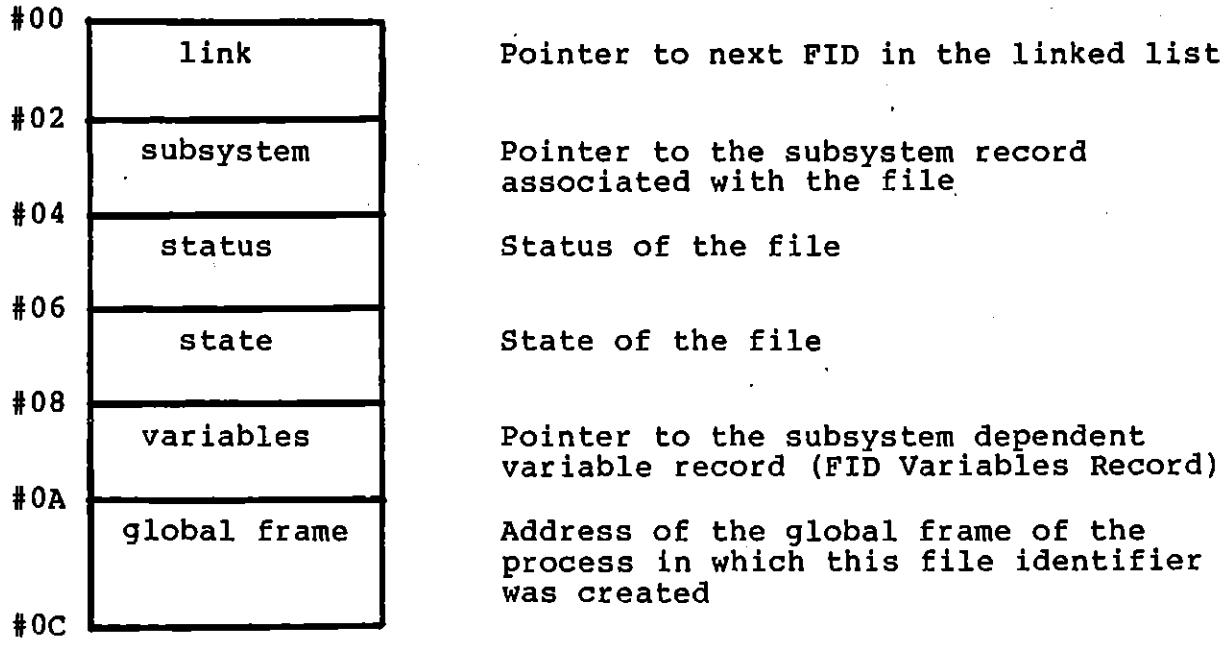


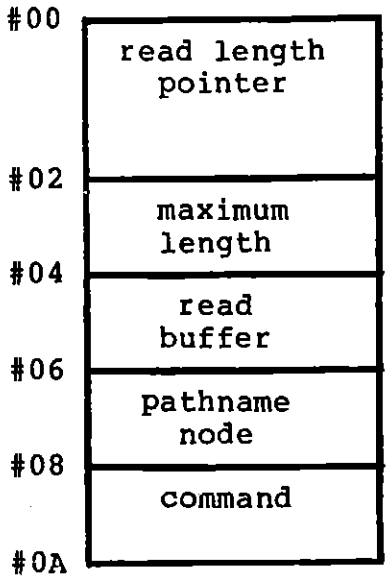
FIGURE B-5. FILE IDENTIFICATION RECORD

B.4 IPC I/O SUBSYSTEM DATA STRUCTURES

These data structures are used exclusively to implement the interprocess communication (IPC) I/O subsystem. The following data structures allow data to be transferred via messages passed through channels.

B.4.1 IPC FID Variables Record

This record is accessed through a FID record. It contains the addresses of parameters used to read data, the file's message buffer, and a pointer to the pathname record.



Address of the word into which the number of characters transferred is to be stored at the completion of a read request

Indicates the maximum number of characters that can be read into a buffer

Address of the buffer into which data are to be read

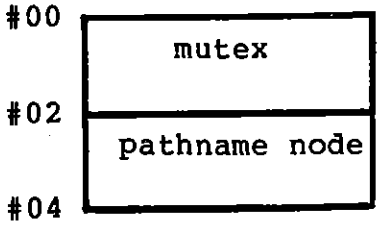
Address of the unique pathname record for the file

Address of the message record used to transmit data

FIGURE B-7. IPC FID VARIABLES RECORD

B.4.2 IPC Port Variables Record

This record is accessed through an IPC-subsystem record. It points to a linked list of pathname records, each containing the unique characteristics of a particular file.



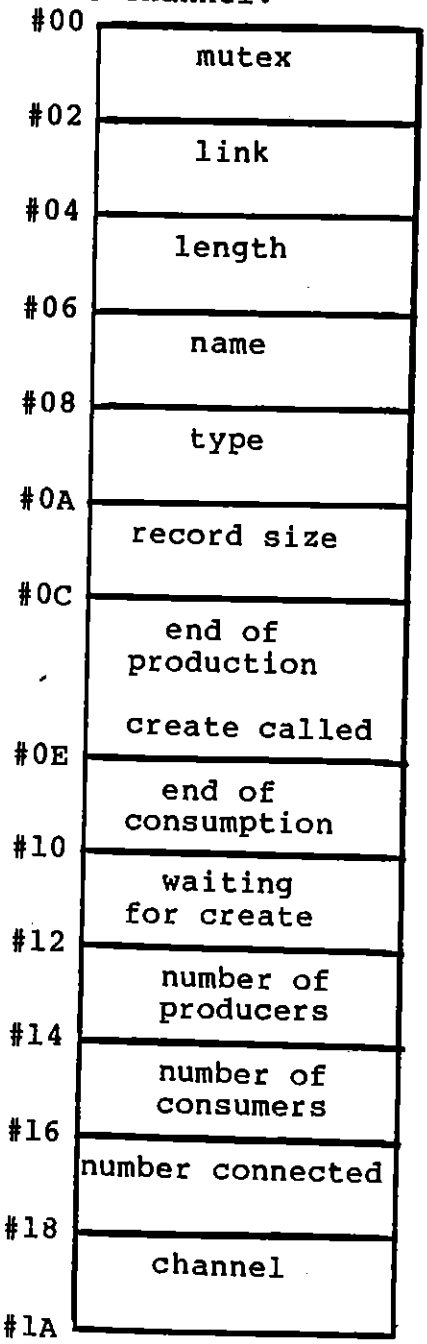
Address of the semaphore used to ensure mutual exclusion when accessing the list

Address of the first pathname in the list

FIGURE B-8. IPC PORT VARIABLES RECORD

B.4.3 Pathname Record

This record is accessed through either the pathname node field of the process IPC FID variable record or the IPC port variables record. The pathname record contains characteristics unique to a given file. Also contained are values used to access and synchronize access to the file's channel.



Address of the semaphore used to ensure exclusive access to the pathname record

Address of next pathname in linked list

Number of characters in the file's name

Address of the string containing the file's name

Packed record defining file format, record format, file usage, and file compression

Maximum number of characters in a logical record

Boolean indicates if all producers have closed on a channel

Boolean indicates file creation

Points to a semaphore used to synchronize the closing of producers

Points to a semaphore used to synchronize the creation of a file

The number of processes writing to a file

The number of processes reading a file

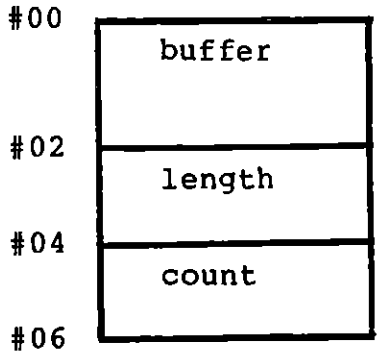
The number of processes connected to a file

Address of the unique channel associated with this pathname

FIGURE B-9. IPC PATHNAME RECORD

B.4.4 Message Record

Interprocess communication data is transmitted through a message record.



Address of the data to be sent via IPC or of the buffer into which data is to be received

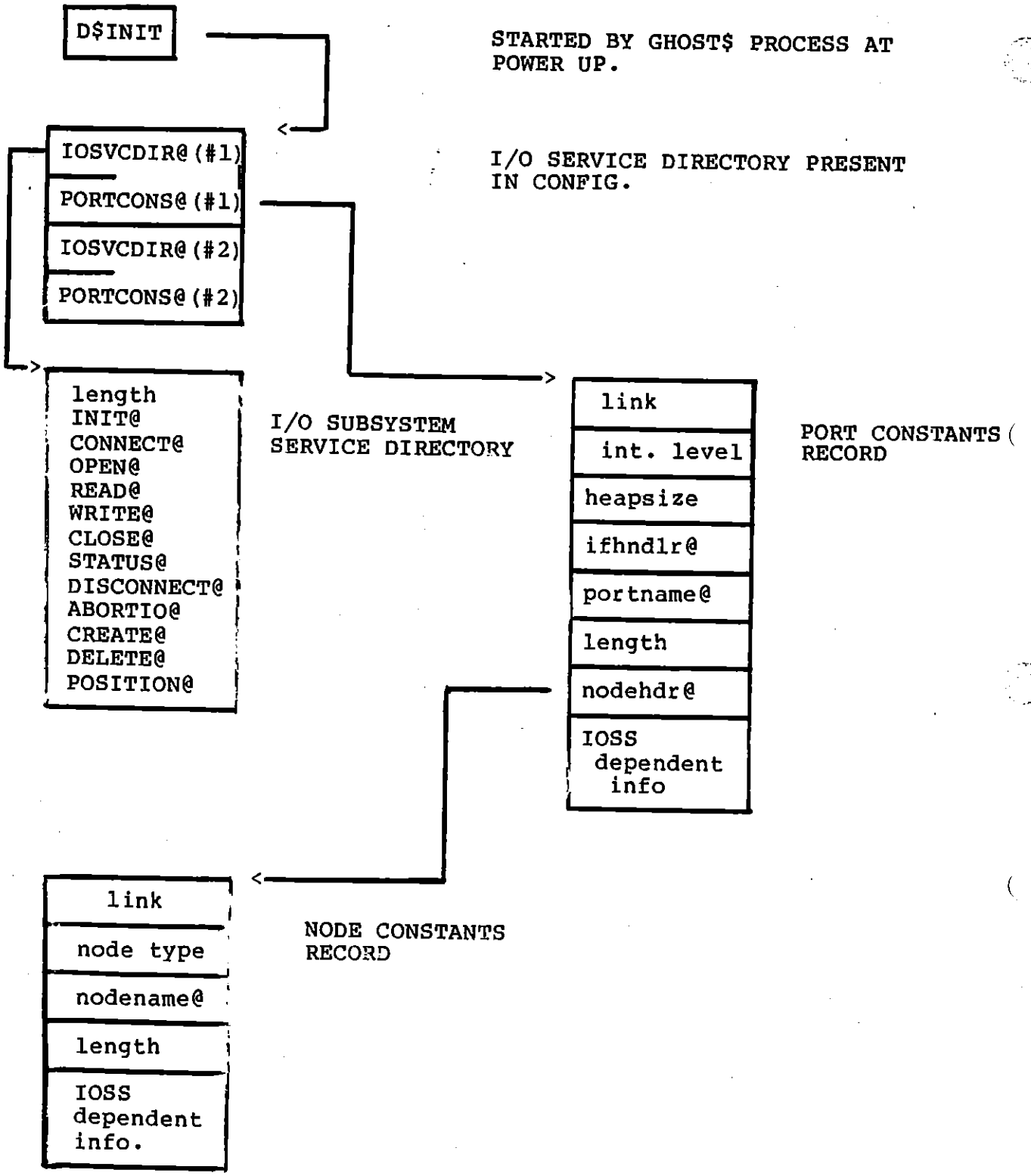
Number of total bytes in the message field

Number of bytes actually sent in the message field

FIGURE B-10. IPC MESSAGE RECORD

B.5 INITIALIZATION OVERVIEW

The figure that follows presents an overview of system initialization via D$INIT. The purpose of this illustration is to tie together many of the above data structures.


D$INIT
STARTED BY GHOST$ PROCESS AT POWER UP.
IOSVCDIR@(#1)
PORTCONS@(#1)
IOSVCDIR@(#2)
PORTCONS@(#2)
I/O SERVICE DIRECTORY PRESENT IN CONFIG.
length
INIT@
CONNECT@
OPEN@
READ@
WRITE@
CLOSE@
STATUS@
DISCONNECT@
ABORTIO@
CREATE@
DELETE@
POSITION@
I/O SUBSYSTEM SERVICE DIRECTORY
link
int. level
heapsize
ifhndlr@
portname@
length
nodehdr@
IOSS dependent info
PORT CONSTANTS RECORD
link
node type
nodename@
length
IOSS dependent info.
NODE CONSTANTS RECORD

APPENDIX C

STATUS AND ERROR MESSAGES

C.1 GENERAL

This section presents error messages generated during execution of the File I/O Decoder. These error messages are at the level of the File Identifier Record (FID) denoting error information relative to operations on the FID. Three categories of messages are described: "Status" information captured in the STATUS field of the FID record and returned by the function D$STATUS, "State" information captured in the STATE field of the FID and returned by a call to the function D$VALID, and process information which may be examined in the Process Descriptor Record.

Error messages generated during execution of the individual I/O Subsystems are unique to each I/O Subsystem. In many instances, a user who is returned a FID level error message will need to inspect error messages returned by the specific I/O Subsystem. Two ways of obtaining this error information are available. If more than one I/O Subsystem is operating on the target and the user is unaware of the particular I/O Subsystem to be accessed, a call to D$DSTATUS should be made. If the appropriate I/O Subsystem is known, a call to the Status function in that subsystem can be made. Information on the Status messages at the I/O Subsystem level can be obtained from the user manuals dedicated to the various I/O Subsystems.

C.2 STATUS

The current status maintained in the FID record field "Status" is returned to the user when the File I/O Decoder function D$STATUS is called. This status information is subsystem independent, defining the success or failure of the oldest outstanding request on the FID.

In general, a status value of 0 indicates that no error condition currently exists and no activity is in progress. A non-zero value indicates the current status of an outstanding request or the existence of an exception condition. Values for the following conditions are standarized (device/subsystem independent):

CODE	CONDITION
0000	Idle or last request complete. No exception condition.
8xxx	Request in progress.
0101	End of File encountered.

0102	End of information encountered. No more information is present on the medium.
0103	End of medium encountered.

NOTE: The 01xx messages defined above are not exclusive but rather may occur together.

02x0	File error condition as follows:
1	Unsuccessful open
2	Unsuccessful read
3	Unsuccessful write
4	Unsuccessful close
5	Unsuccessful disconnect
6	Unsuccessful create
7	Unsuccessful delete
8	Unsuccessful position
9	Unsuccessful abortio

The user may need to examine the error messages generated at the I/O Subsystem level to determine the cause of the 02x0 messages above.

04x0	Physical data link error on last request as defined by subclassifications 1 through 9 above.
08xy	Illegal state change: x = 0..6 and defines present state of FID; y = 0..9 and defines operation on FID that was requested but failed (y values specify same conditions as x values above). For additional information on State, refer to Subsection C.3 below.

C.3 VALID STATE CHANGES

State information defines the condition of the FID (e.g., Connected, Open, Closed, etc.). By calling the File I/O Decoder function D$VALID, the user may check for valid state changes. The function returns a value of True or False based upon the attempted operation on the FID (the call to D$VALID is described in Subsection 3.12.13). If False is returned, a call to D$STATUS should be performed in order to check for the specific error (#08xy will be returned as described above).

Valid State changes are presented in table format below.

TABLE C-1. STATE TABLE FOR FILE I/O DECODER

State of FID	Operation on FID							
	Conn	Create	Open	Read	Posit	Write	Close	Delete	Disc
0. Initial	1								
1 Conn/Close		2	4,5,6					3	0
2 Created			4,5,6					3	0
3 Deleted		2							0
4 Open for Rd				4	4		1		
5 Open for Wt						5	1		
6 Open for R/W				6	6	6	1		

In the above table, the FID states are listed vertically and numbered 0 thru 6. Operations that can be attempted on the FID are indicated to the right of each FID state. The numbers to the right of each FID state identify the subsequent state of the FID (as idenitified by the vertical numbers) after the corresponding FID operation is performed. For example, After the FID is connected the FID state changes from 0 (Initial) to 1 (Connect/Close). After the connected FID is Opened, the FID state is 4, 5, or 6.

NOTE: the Initial state (assigned 0) is not a true state but rather exists merely for documentation purposes. Prior to connection, the FID does not exist. Also, after disconnection, the FID does not exist.

C.4 RUN-TIME SUPPORT ERROR MESSAGES

The following error messages concerned with the File I/O Decoder and I/O Subsystems are generated by the RTS during program execution. These errors are captured in the process record of the active process.

I/O Decoder Errors - Class Code = B

1 empty file identifier list
2 file identifier not found
3 file identifier not released

Interprocess Communication Errors - Class Code = C

1 no heap for pathname record
2 no heap for name field
3 no heap for file variable record
4 no heap for port variables

APPENDIX D

IMPLEMENTATION OF DUMMY I/O SUBSYSTEMS

D.1 GENERAL

The Dummy I/O Subsystem is comprised of a set of dummy routines that serve two purposes:

1) Certain services are not meaningful in some subsystems due to the nature of the medium associated with the subsystem. Often the corresponding service routine in the dummy subsystem may be used to provide a "dummy" or "no-op" routine that conforms to I/O subsystem interface requirements. This substitution is made in the routine list of the I/O Service Directory (Appendix B) of the subsystem. For example, IPC$DELETE is not implemented in the IPC subsystem (Section IV) since there is no physical media to be deleted. If a process references a delete routine through the I/O Service Directory, then the dummy subsystem routine DUM$DELETE will be invoked.

2) The Dummy Subsystem also allows producer and consumer processes to create and access specific files on which most file operations are suppressed. Such a system is useful to define "dummy files" in processes. In these systems, the messages written to dummy files are consumed but not passed along to other processes. Processes which try to read from such files receive end-of-file status. This subsystem can be used to "dummy" out file access in user applications.

Routines comprising the dummy subsystem may be invoked as all other I/O Subsystem routines (via the File I/O Decoder). Thus, the calling sequence for each dummy routine is compatible with the calling sequence for the corresponding routine found in any other subsystem. Also, the definition of the use of the parameters in the dummy routines is consistent with their definition in all other corresponding subsystem routines.

The data structures (records) used in the dummy subsystem contain the same fields as illustrated in Appendix B (describing all I/O Subsystem data structures).

D.2 Dummy Routine Descriptions

The routines comprising the dummy subsystem are described below. The routines are considered alphabetically; no calling sequences are presented as they are the same as for other I/O Subsystems. The Pascal source for each routine is supplied with ;the product.

D.2.1 Procedure DUM$ABORTIO

This routine will call a wait routine through the I/O Decoder. The I/O Decoder will then reference the appropriate subsystem's wait routine as configured. If the FID is a dummy subsystem file, then the Decoder will call DUM$WAIT, which does nothing.

D.2.2 Procedure DUM$CLOSE

This routine checks that a file is in a valid state for a call to a close routine. If the file is in a valid state, then a wait routine is called through the I/O Decoder to insure that all impending I/O operations have completed. DUM$CLOSE then updates the status and state of the file and returns. The parameter "with_eof" is not used in this system.

D.2.3 Procedure DUM$CONNECT

This routine searches the port node records associated with the dummy subsystem to determine if the pathname parameters of this routine is one that should be connected to the dummy subsystem. If it is, a file identifier record is constructed and returned to the caller.

D.2.4 Procedure DUM$CREATE

This routine checks that a file is in a valid state for a call to a create routine. If the state is valid, then routine will update the status and state of the file. DUM$CREATE performs no create operation on the file but was included to provide the dummy subsystem with a create routine which has a calling sequence conforming with the create call of other I/O subsystems (the only meaningful parameter is the FID).

D.2.5 Procedure DUM$DELETE

This routine checks that a file is in a valid state for a call to a delete routine. If the state is valid, then routine will update the status and state of the file. DUM$DELETE performs no delete operation on the file but was included to provide the dummy subsystem with a delete routine which has a calling sequence conforming with the delete call of other I/O subsystems.

D.2.6 Procedure DUM$DISCONNECT

This routine checks that a file is in a valid state for a call to a disconnect routine and then deallocates those data structures that are associated with a file identifier record. Note that this routine can be used by other subsystems if they allocate "fid_variables" using NEW

and if the fid variable record contains no substructures that require special processing to be deallocated.

D.2.7 Function DUM$DSTATUS

This function always returns a normal status for the file. DUM$DSTATUS performs no status check on the file but was included to provide the dummy subsystem with a status routine which has a calling sequence conforming with the status call of other I/O subsystems.

D.2.8 Procedure DUM$INIT

This routine initializes the subsystem record.

D.2.9 Procedure DUM$OPEN

This routine checks that a file is in a valid state for a call to an open routine. If the state is valid, then the file is opened for reading, writing, or both as specified in the parameter "priv". The parameters "ft", "logical_record_length" and "number_of_records" are then initialized to be compatible with a file of variable length records with typical length of 80 bytes.

D.2.10 Procedure DUM$POSITION

This routine checks that a file is in a valid state for a call to a position routine. If the state is valid, then the routine will update the status of the file. DUM$POSITION performs no position operation on the file but was included to provide the dummy subsystem with a position routine which has a calling sequence conforming with the position call of other I/O subsystems.

D.2.11 Procedure DUM$READ

This routine checks that a file is in a valid state for a call to a read routine. If the state is valid, then the routine will set the read parameter "Actual Length" to a default value, and update the status of the file. No actual read will occur and an end-of-file status will always be returned.

D.2.12 Procedure DUM$WAIT

This routine performs no wait operation on a file. DUM$WAIT performs no wait operation on the file but was included to provide the dummy subsystem with a wait routine which has a calling sequence conforming with the wait call of other I/O subsystems.

D.2.13 Procedure DUM$WRITE

This routine checks that a file is in a valid state for a call to a write routine. If the state is valid, then the routine will update the status of the file. No message is written to a file.

APPENDIX E

FILE ATTRIBUTES FOR USE IN CALLING FILE SERVICE ROUTINES

E.1 GENERAL

This appendix contains a discussion of the file attributes described by the parameters passed to the File I/O Decoder entry points D$CREATE and D$OPEN (detailed in Section III). These entry points are accessed (whether directly or via the Pascal primitves) to create and open a file for access.

The file attributes discussed below include file format, record format, file compression, access type, access privilege and protection, and file names.

NOTE: Some parameters passed at the file level to define these attributes are ignored at the I/O Subsystem level because they are meaningless to the node being controlled. (e.g., D$CREATE's parameter Protection Code is ignored by the IPC Subsytem's Create routine).

E.2 FILE FORMAT

File format describes the physical organization of a file and is specified at the time the file is created. Contiguous files have a fixed file extent. Non-contiguous files have no such fixed length and are allowed to grow dynamically up to a maximum of 16 secondary allocations. Contiguous and non-contiguous files and how they are allocated to bulk memory storage are described below.

E.2.1 Contiguous Files

When a contiguous file is created, the user specifies the maximum number of records to be written in the file and the length in bytes of each logical record. The number of records is multiplied by logical record length to arrive at file extent.

E.2.2 Non-Contiguous Files

When a non-contiguous file is created, the user specifies a logical record length, an initial number of records to be written in the file, and an incremental number of records to expand the file. The primary allocation is the product of the primanry number of records and the logical record length both rounded up to the nearest multiple of AUs. The secondary allocation is the product of the incremental number of records and the logical record length.

The non-contiguous file format allows files to grow dynamically. Non-contiguous files can be created small and can be allowed to grow as needed.

E.3 RECORD FORMATS

Record format describes the logical organization of the file and is specified by the user at the time of file creation. Three record formats are defined:

- Variable length
- Fixed length
- Free length

Each of these formats is examined below.

E.3.1 Variable Length

Variable length records making up a file are logical data structures in which the individual record length is not fixed. A variable length record format provides an economical way to use file space in applications that must record data structures of unpredictable lengths. Since the length of the records is a variable, the length of each individual record must be recorded along with the record data itself. Record headers and trailers are used to contain length information and provide record boundaries. The format of variable length records supports file compression as discussed in Subsection E.4.

E.3.2 Fixed Length

A file of fixed length records consists of copies of logical data structures all of which are exactly the same length. Fixed length records facilitate random access; the command to change a file's access position functions more easily on files of fixed length records.

E.3.3 Free Length

The free region format allows the user to conceptually subdivide an unformatted array of bytes into a structure of his own choosing. In a file of free region records, the logical record length is one.

E.4 FILE COMPRESSION

File compression is achieved via the surpression of nulls in the recording of data contained in files with variable length records. Information stored in the record header and trailer results in the automatic restoration of nulls when the file's records are accessed. Because of the necessity of header and trailer information, and because of the varying record lengths that can result from file compression, file compression cannot be used on files of fixed length (or free length) records.

E.5 ACCESS METHODS

Access methods pertain to the action of recording and retrieving data in a file. Access via the D$READ and D$WRITE routines takes place sequentially. Records are processed in order of their increasing record numbers on the medium. Random access (processing of a record regardless of its location) can occur by using the D$POSITION command to reposition the file to the desired location. Because of the nature of variable and fixed length records, random access proceeds slower on the former.

E.6 ACCESS PRIVILEGE

Access privilege is requested at the time the file is opened for access and establishes a relationship between a user and a data file. This relationship specifies what activities the user can perform on the file (read, write, execute, and extend as described below) and whether or not any other user will be allowed to access the file. The user specifies True or False as defined below.

Exclusion	By specifying True, the user gains exclusive access to the file or is denied access if the file is being accessed by another user.
Reading	By specifying True, the user requests the capability to read. Whether or not others may access depends on the "Exclusion" field above.
Writing	By specifying True, the user requests the capability to write. If the user will be writing to a variable length record file, he must specify True to the "Exclusion" field.
Execute	By specifying True, the user requests the capability to execute a program file. This option is for possible use by operating systems and is not currently supported.
Extend	By specifying True, the user wishes to extend the file.

For each of the above access activities listed, the appropriate password is also required to allow user access as described in the following subsection.

E.7 ACCESS PROTECTION

At create time, the user can assign a level of password protection to the access activities that can be performed on a file. The parameter Protection code defines this protection. For each of four access activities, read, write, modify, and execute, a level of password proctection can be assigned as follows:

#1 Unrestricted Access
#2 User Password needed to access
#3 Creator password needed to access
#4 No access

where user and creator passwords are defined at the time the file is opened. It is possible to access a file with level 2 protection by using a Creator Password, since Creator access is a special form of User access.

E.8 FILE NAME

A special file attribute defined when the file is first connected to an I/O Subsystem is the file pathname. While this name is I/O Subsystem denpendent, the general naming convention is as follows:

1) A pathname consists of one or more node names.

2) Each node name consists of up to eight valid ASCII characters; a character may be an upper-case alphabetic, a numeral, or a dollar sign ('$').

3) The first character in each node name must be alphabetic.

4) Each node name is separated from other nodes by an ASCII period.

APPENDIX F

GLOBAL DECLARATION FILES FOR DIF I/O PACKAGE

F.1 GENERAL

Global Declaration Files for the Pascal programmer using DIF I/O routines are listed below. Two Global Declaration Files are presented. The first is used for Pascal applications requiring the File I/O Decoder. The second is used for Pascal applications utilizing the Interprocess Communication I/O Subsystem.

F.2 FILE I/O DECODER GLOBAL DECLARATION FILE

```
program d$declarations;

const

  dont_care = 2;

{ protection type }
  any_access = 1;
  user_password = 2;
  creator_password = 3;
  no_access = 4;
  max_protection_type = 4;

{ file format }
  contiguous = 1;
  noncontiguous = 2;
  max_file_format = 2;

{ record format }
  free_length = 1;
  variable_length = 2;
  fixed_length = 3;
  max_record_format = 3;

{ file usage }
  data = 1;
  directory = 2;
  allocation_map = 3;
  max_file_usage = 3;

{ file compression }
  uncompressed = 1;
  compressed = 2;
  max_file_compression = 2;

{ file access mode }
  byte_relative = 1;
```

```
  sequential = 2;
  direct = 3;
  max_file_access_mode = 3;

type

  address = integer;
  byte = 0..#FF;
  dummy_index_range = 1..dont_care;
  dummy_buffer = packed array[ dummy_index_range ] of char;
  dummy_buffer_ptr = @dummy_buffer;
  fid = @fid;
  hex_digit = 0..#F;

  file_access_mode = 1..max_file_access_mode;

  file_access_privilege = packed record
    exclusive_access: boolean;
    read_access: boolean;
    write_access: boolean;
    execute_access: boolean;
    extend_access: boolean;
    end;

  file_type = packed record
    file_format: hex_digit;
    record_format: hex_digit;
    file_usage: hex_digit;
    file_compression: hex_digit;
    end;

  password_list = record
    { to be determined }
    end;
  password_list_ptr = @password_list;

  protection = packed record
    read_protect: hex_digit;
    write_protect: hex_digit;
    modify_protect: hex_digit;
    execute_protect: hex_digit;
    end;
```

```
{*****************************************************************}

procedure d$abortio( f: fid ); external;

procedure d$close( f: fid; with_eof: boolean ); external;

procedure d$connect( var pathname: dummy_buffer;
                         length:   integer;
                     var f:        fid ); external;

procedure d$create( f:                    fid;
                    passwords:            password_list_ptr;
                    protect:              protection;
                    ft:                   file_type;
                    logical_record_length: integer;
                    initial_allocation:   longint;
                    extension_allocation: longint ); external;

procedure d$delete( f: fid );  external;

procedure d$disconnect( var f: fid );  external;

procedure d$init;  external;

procedure d$open(     f:                    fid;
                      passwords:            password_list_ptr;
                      mode:                 file_access_mode;
                      priv:                 file_access_privilege;
                  var ft:                   file_type;
                  var logical_record_length: integer;
                  var number_of_records:    longint ); external;

procedure d$position( f:        fid;
                      relative: boolean;
                      number:   longint ); external;

procedure d$rdwait(     f:             fid;
                    var b:             dummy_buffer;
                        max_length:    integer;
                    var actual_length: integer ); external;

procedure d$read(     f:             fid;
                  var b:             dummy_buffer;
                      max_length:    integer;
                  var actual_length: integer ); external;

function d$status( f: fid ): integer; external;

procedure d$wait( f: fid ); external;

procedure d$write(     f:      fid;
                   var b:      dummy_buffer;
                       length: integer ); external;
```

```
procedure d$wrwait(     f:      fid;
                   var b:       dummy_buffer;
                       length: integer ); external;

{*****************************************************************}

begin
  {$ nullbody }
end.
```

F.3 GLOBAL DECLARATION FILE FOR INTERPROCESS COMMUNICATION SUBSYSTEM

```
{$ statmap, map }
 program dummy$subsystem;

 const

  bytes_per_word = 2;
  dont_care = 2;

{ protection type }
  any_access = 1;
  user_password = 2;
  creator_password = 3;
  no_access = 4;
  max_protection_type = 4;

{ file format }
  contiguous = 1;
  noncontiguous = 2;
  max_file_format = 2;

{ record format }
  free_length = 1;
  variable_length = 2;
  fixed_length = 3;
  max_record_format = 3;

{ file usage }
  data = 1;
  directory = 2;
  allocation_map = 3;
  max_file_usage = 3;

{ file compression }
  uncompressed = 1;
  compressed = 2;
  max_file_compression = 2;

{ file access mode }
  byte_relative = 1;
  sequential = 2;
```

```
    direct = 3;
    max_file_access_mode = 3;

  { fid state }
    closed = 1;
    created = 2;
    deleted = 3;
    open_for_reading = 4;
    open_for_writing = 5;
    open_for_both = 6;

  { fid operations }
    $open = 1;
    $read = 2;
    $write = 3;
    $close = 4;
    $disconnect = 5;
    $create = 6;
    $delete = 7;
    $position = 8;
    max_fid_operation = 8;

    $abortio = 9;

    eof_encountered = #0101;

    file_error = #0200;
    unsuccessful_open = file_error + #10 * $open;
    unsuccessful_read = file_error + #10 * $read;
    unsuccessful_write = file_error + #10 * $write;
    unsuccessful_close = file_error + #10 * $close;
    unsuccessful_disconnect = file_error + #10 * $disconnect;
    unsuccessful_create = file_error + #10 * $create;
    unsuccessful_delete = file_error + #10 * $delete;
    unsuccessful_position = file_error + #10 * $position;
    unsuccessful_abortio = file_error + #10 * $abortio;

    fid_illegal_operation_error = #0800 { + fid operation };

    normal = 0;
    waiting = #8000 ;

    ipc$err = 12;
    no_heap_for_pathname_record       = 1;
    no_heap_for_name_field            = 2;
    no_heap_for_file_variable_record  = 3;
    no_heap_for_port_variables_record = 4;

  type
     rt_type = ( none$$, err$, f$$, hp$$, in$$, p$$, c$$, sm$$,
                 ipc$$, ct$$, iod$$ );
    address = integer;
    byte = 0..#FF;
    dummy_index_range = 1..dont_care;
```

```
dummy_buffer = packed array[ dummy_index_range ] of char;
dummy_buffer_ptr= @dummy_buffer;
fid_operation = 1..max_fid_operation;
hex_digit = 0..#F;

file_access_mode = 1..max_file_access_mode;

fid = @fid_record;
fid_variables_ptr = @ipc$fid_variables_record;
port_constants_ptr = @ipc$port_constants_record;
port_variables_ptr = @ipc$port_variables_record;
service_directory_ptr = @service_directory_record;
subsystem_ptr = @subsystem_ptr;

memptr = @memptr;
hp$ = @hp$;
byte_length = 0 .. 32767;

semaphorestate = ( awaited, zero, signaled );

cid = @cid;
command_ptr = @command_record;
passcode_ptr = @passcode_ptr;
pathname_node_ptr = @pathname_node;
parameters_ptr = @ipc$parameters;

file_access_privilege = packed record
  exclusive_access: boolean;
  read_access: boolean;
  write_access: boolean;
  execute_access: boolean;
  extend_access: boolean;
  end;

file_type = packed record
  file_format: hex_digit;
  record_format: hex_digit;
  file_usage: hex_digit;
  file_compression: hex_digit;
  end;

password = packed array [1..4] of char;

password_ptr = @password;

password_list = record
  user_password: password;
  creator_password: password;
  end;

password_list_ptr = @password_list;

protection = packed record
  read_protect: hex_digit;
```

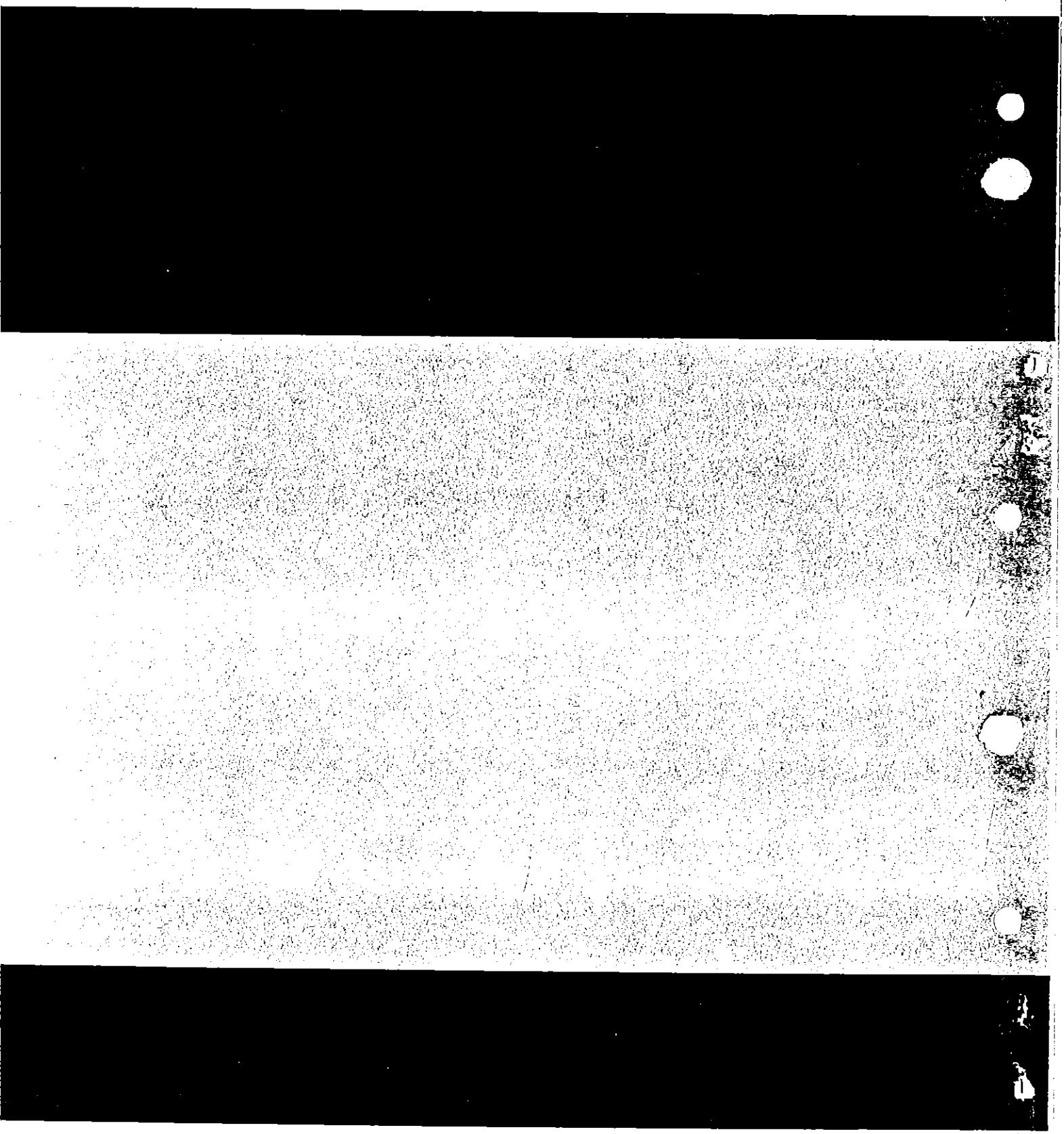
```
  write_protect: hex_digit;
  modify_protect: hex_digit;
  execute_protect: hex_digit;
  end;

fid_record = record
  link: fid;
  subsystem: subsystem_ptr;
  status: integer;
  state: integer;
  fid_variables: fid_variables_ptr;
  global_frame: address;
  end;

service_directory_record = record
  length: integer;
  $init: address;
  $connect: address;
  $open: address;
  $read: address;
  $write: address;
  $close: address;
  $status: address;
  $disconnect: address;
  $abortio: address;
  $create: address;
  $delete: address;
  $position: address;
  $wait: address;
  end;


ipc$port_constants_record = record
  { unused by ipc$ }
  end;

ipc$port_variables_record = record
  mutex: semaphore;
  pathname_node: pathname_node_ptr;
  end;

command_record = record
  buffer: dummy_buffer_ptr;
  length: integer;
  count: integer;
  end;

ipc$fid_variables_record = record
  read_length_ptr: @integer;
  read_maxlength: integer;
  read_buffer_ptr: dummy_buffer_ptr;
  pathname_node: pathname_node_ptr;
  command: command_ptr;
  end;
```

```
  pathname_node = packed record
    node_mutex: semaphore;
    link: pathname_node_ptr;
    length: integer;
    name: dummy_buffer_ptr;
    ft: file_type;
    logical_record_length: integer;
    end_of_production, create_called: boolean;
    end_of_consumption, waiting_for_create: semaphore;
    number_of_producers, number_of_consumers,
    number_connected: integer;
    channel: cid;
    end;

  ipc$parameters = record
    base: integer;
    level: integer;
    rate: integer;
    length: integer;
    name: dummy_buffer;
    end;

common
  $msg$: fid;


procedure d$fid(     sub:       subsystem_ptr;
                     fid_vars:  fid_variables_ptr;
                var  f:         fid ); external;

procedure d$fidrelease( var f: fid ); external;

procedure d$subsystem(     serv:      service_directory_ptr;
                           port_cons: port_constants_ptr;
                           port_vars: port_variables_ptr;
                      var  sub:       subsystem_ptr ); external;

function d$valid( f:  fid;
                  op: fid_operation ): boolean; external;

procedure ipc$close( f: fid; close_with_eof: boolean ); forward;

procedure ipc$connect(     sub:      subsystem_ptr;
                      var  pathname: dummy_buffer;
                           length:   integer;
                      var  f:        fid ); forward;

procedure ipc$create( f:                    fid;
                      passwords:            password_list_ptr;
                      protect:              protection;
                      ft:                   file_type;
                      logical_record_length: integer;
                      initial_allocation:   longint;
```

```
                        extension_allocation:  longint ); forward;

procedure ipc$disconnect( var f: fid );  forward;

procedure ipc$init(     serv:      service_directory_ptr;
                        port_cons: port_constants_ptr;
                    var sub:       subsystem_ptr ); forward;

procedure ipc$open(     f:                        fid;
                        password:                 password_ptr;
                        mode:                     file_access_mode;
                        priv:                     file_access_privilege;
                    var ft:                       file_type;
                    var logical_record_length:    integer;
                    var number_of_records:        longint ); forward;

procedure ipc$read(   f:             fid;
                      b:             dummy_buffer_ptr;
                      max_length:    integer;
                  var actual_length: integer ); forward;

procedure ipc$wait( f: fid ); forward;

procedure ipc$write( f:      fid;
                     b:      dummy_buffer_ptr;
                     length: integer ); forward;

procedure cwait( s: semaphore; var received: boolean ); external;
function cksemaphore( s: semaphore ): boolean; external;
procedure initsemaphore( var s: semaphore; count: integer ); external;
procedure signal( s: semaphore ); external;
function semastate( sema: semaphore ): semaphorestate; external;
procedure termsemaphore( var s: semaphore ); external;
procedure wait( s: semaphore ); external;
procedure waitsignal( waitfor, signalthe : semaphore ); external;

procedure exception( classcode, reasoncode: integer ); external;
procedure rt$enter( typ: rt_type; abstract_object: address ); external;
procedure rt$exit; external;
function eq$names(var name1: dummy_buffer; length1: integer;
        name2: dummy_buffer_ptr; length2: integer ): boolean; external;

procedure hp$free(heap: hp$; var ptr: memptr); forward;
procedure hp$new(heap: hp$; var ptr: memptr; length: byte_length);
  forward;
function hp$system: hp$; forward;

procedure c$acknowledge( cmd: command_ptr ); external;
procedure c$allocate( msg_size: integer; var cmd: command_ptr );
  external;
procedure c$creceive( c: cid; var cmd: command_ptr ); external;
procedure c$dispose( var cmd: command_ptr ); external;
procedure c$init( i: integer; var c: cid ); external;
procedure c$receive( c: cid; var cmd: command_ptr ); external;
```

```
procedure c$send( c: cid; cmd: command_ptr ); external;
procedure c$term( var c: cid ); external;
procedure c$wait( cmd: command_ptr ); external;

begin
 {$ nullbody }
end.
```

INDEX



TEXAS INSTRUMENTS
INCORPORATED

Post Office Box 1443 / Houston, Texas 77001
Semiconductor Group

MP 386

Printed in U.S.A.