INTELLEC® SERIES III MICROCOMPUTER DEVELOPMENT SYSTEM PROGRAMMER'S REFERENCE MANUAL

Order Number: 121618-002

PRINT HISTORY

Edition	Software	Print Date	Reason
Second	ISIS-II V4.1 ISIS-III (D) V1.0	3/81	Added information concerning Network Development System-I (NDS-I).

BXP	Intel	Library Manager	Plug-A-Bubble
CREDIT	intel	MCS	PROMPT
i	Intelevision	Megachassis	Promware
ICE	Intellec	Micromainframe	RMX/80
iCS	iRMX	Micromap	System 2000
im	iSBC	Multibus	UPI
Insite	iSBX	Multimodule	µScope

PREFACE

This manual provides you with complete information about the Series III from the programming viewpoint. This includes pertinent factors in the Series III hardware environment, plus discussions of the two operating systems with their built-in requirements and capabilities, notably the system service routines.

These routines are Intel program modules, supplied with the system, that your program can call to perform various input/output functions on your Intellec Microcomputer Development System.

Because that system contains two separate operating environments for program execution, Intel provides two sets of system service routines:

- For 8086-based user systems, described in Chapter 2
- For 8080/8085-based user systems, working under ISIS-II, described in Chapter 3

To run programs in either of these environments requires that you have used the appropriate tools. For example, to execute a PL/M program on the 8085 processor, you need to have compiled and linked it using PL/M-80 and LINK rather than PL/M-86 and LINK86. Similarly, to execute on the 8086 chip, your program needs to have passed through the latter two tools. (Appendix H discusses the issue of linking modules compiled under the 8080-based PL/M-86 with those compiled under the 8086-based translator.)

Required Software

The Series III is an Intellec Microcomputer Development System that provides support for both development and execution of programs using either the 8086 chip or the 8080/8085 chip. The Series III contains both hardware and software beyond earlier versions of the Intellec system.

Software

- A new 8086-resident monitor, DEBUG-86, used during bootstrap (initial cold startup) and available for later program analysis and modification.
- A new 8086-resident nucleus, providing new built-in system service functions for your 8086-based programs, including hard disk access.
- A new system command, RUN, which provides the communication and processing between the 8080/8085 processor and the resident 8086 processor board, RPB-86. RUN loads programs into the RPB-86 environment and determines when to pass control to them.

Other Reference Manuals to Use

• *Intellec Series III Console Operating Instructions*	121609
• *PL/M-86 User's Guide for 8086-Based Development Systems*	121636
• *8086/8087/8088 Macro Assembly Language Reference Manual for 8086-Based Development Systems*	121627

Reader's Guide: How to Use this Manual

The manual is a reference tool enabling you to find the information needed to design and write programs that run on the Series III. To develop a full understanding requires study of the other manuals listed above, notably the *Intellec Series III Console Operating Instructions*.

Parallel Structure

The topics in Chapters 2 and 3 have a parallel structure in that similar functions are presented in similar order. This parallelism is reflected in table 1-3. It shows most of the Series III system service routines, with their closest ISIS-II equivalents, in the order of usage by a hypothetical program.

Functional Groupings

Although alphabetical listings of the system services appear in Chapter 1 and the Index, the central chapters of the manual use functional groupings to present capabilities related to each general topic.

For example, as indicated in the Table of Contents, the routines to allocate or free memory appear in the same section as the routine for finding out the size of an allocated segment, all under the heading of Memory Management.

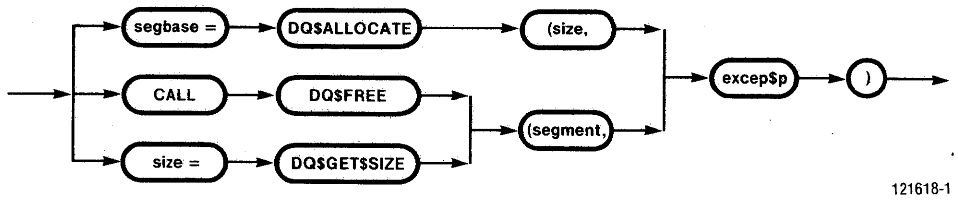
Similarly, the thirteen routines for file management appear under that heading, but separated into four groups of related sub-functions.

Although the Index is enough to find any individual description, these groupings may aid both initial study and later reference to routines related by function.

"Railroad-Chart" Notation

In Chapters 2 and 3 the system service routines are discussed in functional groupings such as File Management Routines, Exception Routines, or Monitor I/O Interface Routines.

At the beginning of each such group discussion there is a chart showing how the routines of that group are invoked, as functions or procedures. For example:



Memory Management Routines

The full name of each system routine is shown in all capital letters, as is the word CALL. These must appear exactly as shown.

Names shown in lower-case italic letters indicate variables or parameters you must supply when you invoke the routine. However, you may substitute any names you choose for these lower-case items.

The example chart indicates that DQ$FREE is invoked as a procedure by a CALL, while DQ$ALLOCATE and DQ$GET$SiZE are invoked as functions. The word returned by DQ$ALLOCATE is a segment base, and the word returned by DQ$GET$SIZE is the size of a block of memory. The parameters used by DQ$FREE and DQ$GET$SIZE are the same, whereas DQ$ALLOCATE has a different first parameter. All three invocations have an exception pointer as the final (second) parameter. The detailed discussion of each of these routines appears in Chapter 2, but the general interpretation of these charts is the same throughout this manual.

CONTENTS

CONTENTS

CONTENTS (Cont'd.)

TABLES

CHAPTER 1
SERIES III INTRODUCTION

Operating System Considerations

An operating system is a group of programs that provide the functional (as opposed to physical) environment in which your programs do their work. As an analogy, the operating system is not unlike the elevators, mail chutes, telephone systems, and other machines inside a modern office building: they greatly facilitate efficient production and allocation of resources.

Needed Capabilities

Usual capabilities of an operating system include the management of devices attached to the hardware of the system. These include the console input and output devices and auxiliary memory devices such as flexible or hard disk drives. Typically, the operating system also recognizes commands to invoke the execution of programs such as language translators, or the programs you develop using them.

Desirable Features

The use of interrupts and the management of overlays and error conditions extend the range of functions (and recoveries) available to your programs. Interrupts allow orderly communication to additional devices or coprocessors attached to the original hardware. Overlays permit the design and use of a program larger than the memory size, by partitioning it into parts whose processing is mutually exclusive and which can fit in the memory available. These are linked by calls that cause memory to be reused by the other sections of the program. Service routines designed to handle exceptions allow early detection and handling of unwanted conditions arising during execution.

Functions of the Series III and ISIS-II Operating Systems

The Series III and ISIS-II operating systems include the above capabilities plus several other functions, such as command scanning and dynamic file or device manipulation (i.e., under program control during execution).

Command scanning permits your program to pick up options specified on the line invoking program execution, or to treat specially formatted files as if they were input at the console.

Dynamic file control means that during execution you can maintain a list of twelve files or devices used by your program (e.g., written or read), but restrict physical access to a smaller group of six actually in use at any one time. The existence of this list can make input/output operations more efficient.

The Series III operating system also provides memory management and an interactive symbolic debugging aid.

Memory management during execution allows you to allocate memory for specific processes as they arise, and free those blocks when they are no longer needed.

The debugging tool is named DEBUG-86. Its interactive language is similar to those of Intel's ICE-86 or ICE-88 emulators. Using it, you can insert breakpoints into your program, execute it until some predefined condition is encountered, and halt so that you may examine the state of processor registers or of variables in your program.

How you use DEBUG-86 is fully described in the *Intellec Series III Console Operating Instructions*.

Program Development Cycle

The cycle begins with an idea and ends with a fully checked-out program that performs the desired work in an acceptable manner.

The idea evolves into a design and, ultimately, into program specifications. The program specifications are split up into smaller groups of functions, each to be performed by a single module of code.

As work progresses, problems may arise. These may be due to unforeseen gaps or complications within the base design, or may reflect difficulties in implementing it. Modules may require expansion, modification, or integration with other modules. Some functions may merge or be abandoned. Communication of parameters may change.

To minimize such changes (and the redesign, rework, and relearning resulting from them), four guidelines are suggested:

1. Extremely clear and specific goals for the program, written down and explicitly agreed upon by the designers, implementers, and users
2. Isolation into separate modules of every non-trivial function of the system or program, including the isolation of difficult design decisions
3. Full and clearly understandable documentation for every module, including liberal comments in the code
4. Clearly written standards for implementation of modules, including conventions for naming and passing parameters

The closer you approach these ideals, the farther you get from unexpected, costly changes.

The modules defined by the process above then form the units of actual programming work, separately specified at the detail level. They are coded, translated, and tested, both individually and in logical groups.

As these groups are tested and combined with other checked-out groups, the complete program approaches final integration. The final tests explore every major option defined by the original program specifications, using input data that reflects the ultimate usage as closely as possible.

At each stage of individual and multi-module testing, the debugging functions provided in the operating system aid in isolating the source of unexpected results. Under the Series III operating system, DEBUG-86 permits use of symbolic names for debugging output, reference to instructions by line number, access to processor registers and flags, and alternate execution modes with or without the use of breakpoints. Under the ISIS-II operating system, the Monitor debugger or an In-Circuit Emulator provide similar functions.

Specific System Services for Each Target Environment

When you are developing a program to run on the Series III, you must choose one of the two environments provided for program execution: the 8086-based or the 8080/8085-based. Each has similar built-in facilities to aid the development and testing of your program products, including standard system services (e.g., input/output) callable from your programs. This frees you from rewriting routines already embedded in the operating system, and provides a standard interface for all modules or systems you develop. However, the interface for each operating system environment is unique: the calls and parameters differ.

For the 8086-Based Environment

For the 8086-based environment, you will develop the modules using the languages which run on the 8086 and produce code that works on the 8086, e.g., the resident ASM-86 or PL/M-86 translators. (Earlier versions of these translators ran on the 8080/8085 chip but produced code to run on the 8086 chip. In some cases, modules compiled or assembled with these prior versions of the translators can be used unchanged. Appendix H clarifies the circumstances in which this is workable.)

Over two dozen system service routines are available in the 8086-based environment. They enable you to use the capabilities of the Series III operating system to manage the resources of that environment.

A conceptual introduction to their expected parameters and results appears early in Chapter 2. A full discussion of each parameter used also appears with the sample PL/M-86 declarations for these external procedures. Discussion of each routine ends with syntax examples and the list of exception conditions that can occur during the routine's execution. Brief, combined usage examples appear after these discussions.

Whenever you write a module that uses one of these service routines, you simply declare it as an external procedure. LINK86 then provides the correct address to the resident system program. You specify the library appropriate to the PL/M-86 model of segmentation you programmed for: SMALL.LIB, COMPAC.LIB, or LARGE.LIB.

After linking and locating groups of modules that will work together in your final system, you can test them alone or together. The DEBUG-86 feature described in the *Intellec Series III Console Operating Instructions* can aid the process of isolating and correcting defects as they become apparent.

For the 8085-Based Environment

For the 8085-based environment, under the ISIS-II operating system, you will use the standard 8085-based versions of these same translators to build your modules. These modules will then run under ISIS-II on the 8085. In this environment, they may call upon ISIS routines for a variety of input/output services which already exist as part of the ISIS-II facilities. Many of these operate similarly to those of the 8086-based operating system, and some are unique to ISIS-II.

There are 15 ISIS-II routines and 13 Monitor routines available. Their conceptual introduction appears in Chapter 3, followed by a detailed discussion of each routine, its parameters, and sample program usage.

As under Series III, whenever you write a module that uses one of these service routines, you simply declare it as an external procedure. When the module is processed by LINK, the correct address to the resident system program is provided. The DEBUG feature of the Monitor can aid the program analysis and correction process.

Built-In Service Routines

The parameters appropriate to each service routine tell what the desired effect is, and include the address of a word to be filled by the system indicating whether the desired operation finished successfully. Your call should normally be followed by code that tests this word, permitting error recovery, alternate processing, or exit, depending on the operation's results.

These parameters are described in the next two chapters. Each chapter presents a uniform (though not identical) format for its system service routines, based on the two different environments and operating systems.

To help you understand what services are available, the names of all 8086-based system routines appear in each of three separate tables below, ordered

a. Alphabetically (table 1-1)
b. In groups by function (table 1-2)
c. By sequence of use in a hypothetical program. Table 1-3 also shows their nearest functional equivalent under ISIS-II. (Some procedures used under the Monitor of ISIS-II have no direct counterpart in the 8086-based environment.)

All 8086-based procedures begin with DQ$.

Table 1-1. Alphabetical List of Service Routines Available in the Series III Operating System

DQ$ALLOCATE	DQ$GET$SYSTEM$ID
DQ$ATTACH	DQ$GET$TIME
DQ$CHANGE$EXTENSION	DQ$OPEN
DQ$CLOSE	DQ$OVERLAY
DQ$CREATE	DQ$READ
DQ$DECODE$EXCEPTION	DQ$RENAME
DQ$DELETE	DQ$SEEK
DQ$DETACH	DQ$SPECIAL
DQ$EXIT	DQ$SWITCH$BUFFER
DQ$FREE	DQ$TRAP$CC
DQ$GET$ARGUMENT	DQ$TRAP$EXCEPTION
DQ$GET$CONNECTION$STATUS	DQ$TRUNCATE
DQ$GET$EXCEPTION$HANDLER	DQ$WRITE
DQ$GET$SIZE	

Table 1-2. Service Routines by Functional Groups

UTILITY and INPUT SCANNING

DQ$GET$SYSTEM$ID
DQ$GET$TIME
DQ$GET$ARGUMENT
DQ$SWITCH$BUFFER

MEMORY MANAGEMENT

DQ$ALLOCATE
DQ$FREE
DQ$GET$SIZE

Table 1-2. Service Routines by Functional Groups (Cont'd.)

FILE MANAGEMENT

Program Connection and File Existence

DQ$ATTACH
DQ$CREATE
DQ$DELETE
DQ$DETACH
DQ$GET$CONNECTION$STATUS

Naming

DQ$RENAME
DQ$CHANGE$EXTENSION

Program Usage

DQ$OPEN
DQ$SEEK
DQ$READ
DQ$WRITE
DQ$TRUNCATE
DQ$CLOSE
DQ$SPECIAL

Program Control

DQ$OVERLAY
DQ$EXIT

EXCEPTION HANDLING

DQ$DECODE$EXCEPTION
DQ$TRAP$CC
DQ$TRAP$EXCEPTION
DQ$GET$EXCEPTION$HANDLER

Table 1-3. Hypothetical Steps in Program Execution and Service Routines Relevant to Each Step

	Service Routine Names	
	For Use In 8086 Environment	**For Use In 8080 Environment (Some names repeat because routine is multi-function)**
1. finds out date and system i.d. for logging/ reporting purposes	DQGETTIME DQGETSYSTEM$ID	none
2. allocates a memory work area for intermediate calculations	DQ$ALLOCATE DQ$GET$SIZE	none
3. determines whether console input is transparent or line-edited	DQ$SPECIAL	none: console is always line-edited
4. rescans the last command (at first, the one invoking this program)	DQGETARGUMENT DQ$SWITCH$BUFFER	RESCAN
5. asks user to enter needed data or parameters at the console	DQ$WRITE DQ$READ	WRITE READ
6. writes to a file	DQ$WRITE	WRITE

Table 1-3. Hypothetical Steps in Program Execution and Service Routines Relevant to Each Step (Cont'd.)

	Service Routine Names	
	For Use In 8086 Environment	For Use In 8080 Environment (Some names repeat because routine is multi-function)
7. loads overlay to process next phase or user response	DQ$OVERLAY	LOAD
8. checks to see if required files are on-line; gets status, including file pointer position	DQ$ATTACH/DQ$OPEN DQGETCONNECTION$STATUS	SEEK/OPEN
9. creates files as needed for program reads/writes	DQ$CREATE/DQ$OPEN	OPEN
10. opens the files for reads/writes	DQ$OPEN	OPEN
11. reads file(s) or seeks to desired position in file	DQ$READ DQ$SEEK	READ SEEK
12. calculates	user supplied	none
13. frees memory work areas no longer needed	DQ$FREE	none
14. closes and/or deletes files	DQ$CLOSE DQ$DELETE	CLOSE DELETE
15. writes new or old file(s)	DQ$WRITE	WRITE
16. renames certain files or changes extension on filename string	DQ$RENAME DQ$CHANGE$EXTENSION	RENAME none
17. truncates and/or closes files no longer needed	DQ$TRUNCATE DQ$CLOSE	 CLOSE
18. detaches files not currently needed	DQ$DETACH	none
19. repeats as needed from (1.)	none	none
20. naturally, errors or exceptions or unwanted conditions can occur at each of the above steps. Thus an implicit step after any of them is the detection/handling of such conditions.	DQ$TRAP$EXCEPTION DQ$DECODE$EXCEPTION DQ$TRAP$CC DQGETEXCEPTION$HANDLER	ERROR
21. exits when job is complete or cannot continue	DQ$EXIT	EXIT

CHAPTER 2
SERIES III OPERATING SYSTEM AND THE 8086-BASED ENVIRONMENT

This chapter discusses each system service routine available in the 8086-based environment of the Series III, as introduced in Chapter 1. These routines provide a variety of capabilities to programs running on the Series III, without the need for user development and checkout, since they are part of the operating system.

Conceptual Consideratons

The system service routines embody a variety of expectations as to their usage, constituting a model of the way programs interact with files, the console, and each other. These expectations are directly reflected in the parameters you must supply when calling these routines. The following are some of the key concepts underlying these parameters.

Command Tail Arguments

An 8086-based program, e.g., named PROGRM, can be invoked by typing RUN PROGRM, or simply PROGRM if the system is already in the RUN mode. (RUN is discussed in the *Intellec Series III Console Operating Instructions*.) PROGRM may have options that can be specified on the invocation line. If so, the remainder of that line (after RUN) is called a "command tail," including any continuation lines.

This command tail is accessible to PROGRM via the system service routine named DQ$GET$ARGUMENT. You call this routine to get each option in the command tail. The first parameter of this call tells the system where to put the next option found, i.e., the address of the name you declared in PROGRM as the string to receive these options.

Successive calls to DQ$GET$ARGUMENT return successive options, each separated by some delimiting character such as a blank or parenthesis. (The details of how to use this routine appear later in this chapter.) The concept of the command tail is basic to the discussion of that routine. The existence of this capability can influence design of programs.

Memory Management

The memory management routines keep track of which memory areas are in use and which are free to be allocated to new uses.

Free space memory management is handled by the service routines DQ$ALLOCATE, DQ$FREE, and DQ$GETSIZE. The extent of the memory managed by the free space manager is determined by the type of object module that is loaded when RUN is invoked, i.e., absolute or relocatable.

When an absolute object module is loaded into memory, the lower limit of the initial free space pool is set **after** the load, to the highest address of the module. An absolute program should be linked/located at 7800H to maximize the amount of free space available to it. Files saved by the ICE SAVE command lack the object records required to keep track of the free space and therefore should not use the free space manager. Thus, relocatable modules are the preferred form.

There are two types of relocatable object module: position-independent-code (PIC) or load-time-locatable (LTL). In PIC modules, there are no segment register changes. The code can work wherever it is ultimately loaded. LTL modules contain special records to resolve program references that do require segment register changes, e.g., an intersegment jump.

When a relocatable object module (PIC or LTL) is loaded, the lower limit of the free space pool is set **before** the load. Memory required to load the segments is then allocated from the initial free space pool by the free space manager.

The top of memory is determined by the first 32k memory block that is not present. In a 128K Series III with a 20-bit IPB board, the top of memory is 1FFFFH (128K). In a Series III with a Model 800, the top of memory is 1F7FF (126K). The difference arises because the memory occupied by the Monitor responds to all multibus accesses to locations between nF800 and nFFFF, and thus the corresponding RAM locations are not usable. In other words, because the Model 800 uses 16-bit addressing, it masks the upper 2K of each 64K block of offboard memory.

A request for memory (i.e., invoking DQ$ALLOCATE) will return the lowest-addressed segment that satisfies the request. When a segment is freed, it is automatically combined with adjacent free memory to form the largest contiguous area possible.

Connections

The operating system will maintain a list of twelve devices or files that your program may use during its execution, i.e., a list of "connections." A connection is a word, named by you, filled by the system service routines DQ$ATTACH or DQ$CREATE. (Only six connections may be open at once, although multiple opens of a single device count for only one.)

You then use this word to specify that file or device whenever you need to perform any operation on it. For files which already exist, DQ$ATTACH and DQ$DETACH can add or delete connections. New files get connected by your use of DQ$CREATE.

For example, when your program performs console input and console output, the connections for :CI: and :CO: must be on this list. The list permits efficient specification and manipulation of devices or files during execution.

Only objects on this list can be opened or closed, read or written. You use the connection rather than the actual device or file name. During execution, your program may perform these functions on multiple files, but only six files and devices total may be open at one time (not counting :CO:).

Some Series III service routines include an 'internal' open as part of their operation (and their descriptions later in this chapter will say so). When you use such a routine, you may need to close another file or device temporarily so as not to exceed the limit of six open files-plus-devices at once. However, you may open a physical device more than once, with this counting as only one open 'file.'

Before a file can be read or written (by DQ$READ or DQ$WRITE), it must not only be connected but opened (by DQ$OPEN). When the activity to that file is completed, it can be closed (by DQ$CLOSE). These four routines can be used only with connections established earlier.

Output devices are created; input devices are attached. For example, workfiles, defined below, and console output :CO: must be created, not attached. Console input is the opposite—:CI: must be attached rather than created.

Buffers

Buffers are areas reserved for expediting disk input/output. A request to buffer a device will be ignored except in the case of :CO: having been redirected to a disk file.

If a series of read operations can be interspersed with calculation, more efficient operation results. This is the case when the data being read in is not used immediately. Similarly, if a sequence of write operations can be interspersed with calculation, efficiency rises.

When you open a file, buffers are allocated according to your specifications. Your program will read (only), write (only), or update (both). When a file is opened for write, the use of the buffers begins with the first call to DQ$WRITE. When opened for read or update, the use of the buffers begins with the call to DQ$OPEN. When a file is open for update, the most efficient usage is to cluster the reads (interspersed with calculations) separately from the writes.

Further, you indicate the optimal number of buffers for the type of usage you see for that file. For seldom-used files, one buffer is best. If the input/output is desired during the actual DQ$READ or DQ$WRITE, as in the case of the console, zero is the correct specification. In all other cases, double buffering is appropriate.

Workfiles

Many programs, e.g., language translators, need temporary files to store intermediate results before final integration and processing can proceed. These files, by their very nature, are of no interest afterward.

Series III recognizes files of this class by the name :WORK:. Such files may be created any number of times. Each time DQ$CREATE is invoked with a pointer to the string "6, :WORK:", a new connection is returned representing a new workfile. The files will be named 001.TMP, 002.TMP, up to 999.TMP.

You can choose what drive will be used for these files with the workfiles command described in the *Intellec Series III Console Operating Instructions.* If you make no choice, drive 1 is used as the default, so that the full pathname for workfiles is :F1:001.TMP, etc. If you issue the command

```
WORK :F3:
```

from the console, this choice is saved in the system file RUN.MAC, and future workfile names would be :F3: 001.TMP, etc.

This can also be done from a program by invoking DQ$CREATE with a pointer to a pathname consisting of only a drive-name part. For example:

```
WRK = DQ$CREATE (@ (4, ':F3:'), @ ERR);
```

Workfiles must be created, opened, closed, and detached like any other new file. However, they are automatically deleted when they are detached. Permanent files on a shared disk must not be named 001.TMP (up to 999.TMP) since they will be deleted as soon as any program creates a workfile (or that many workfiles).

Exception-Conditions and Exception-Handling

Exceptions, or errors, are detected when an indicated operation cannot be completed in a standard manner. The Series III operating system classifies errors as either avoidable or unavoidable. Every system service routine except EXIT returns an error-code through a pointer referred to as *excep$p* in the descriptions later in this chapter. Your programs can test this code to see whether the operation you called for completed successfully. Appendix C contains the standard error names and values.

An error-code of zero means things went well, i.e., as expected. For example, when you call DQ$OPEN to prepare a file for input/output, you supply a connection number. A zero error-code would be returned if the connection representing that file was successfully opened.

A non-zero error-code indicates an inappropriate event during the routine's execution. For example, a write operation would fail if the connection representing the desired file indicated that the file had already been closed or detached. A non-zero error code would be returned. Your program should always check for this.

Unavoidable Errors

These errors generally arise from environmental conditions outside the control of a program. Examples include insufficient memory for a requested operation, or an expected file not found. These errors always return a non-zero value. Often, appropriate program action can be taken. In other instances, no remedial possibilities exist until, for example, the correct disk is found and inserted or the available memory is increased by adjusting other program functions.

Avoidable Errors

These are typically caused by coding errors such as inappropriate parameters or unusable numeric results. Hardware-detected errors also fall into this category. They include division by zero (interrupt 0), overflow (interrupt 4), and interrupts generated by the 8087 Numeric Data Processor. These cause the system's default excepton handler to be executed. (See also Appendix C.)

However, you may establish your own routine to handle hardware-detected exceptions by using the system routine DQ$TRAP$EXCEPTION and supplying a pointer to your exception-handler. (The state of the stack when your routine gets control is discussed under DQ$TRAP$EXCEPTION.)

One special type of exception is defined into the system: When a control-C is typed at the physical console input device, the system cancels the current job. You can program your own response to a control-C, and cause the system to use it by providing a pointer to your private routine, via the system routine DQ$TRAP$CC.

Data Types and Register Convention

The descriptions of the system service routines that follow later in this chapter assume data types largely similar to those of PL/M-86. Your calls to system service routines must supply parameters meeting these specifications.

BOOLEAN	A BYTE object taking the values TRUE (0FFH) or FALSE (00H). The BOOLEAN specification assumes the following literal definition (in PL/M-86 terms):

```
DECLARE BOOLEAN LITERALLY 'BYTE' ;
```

BYTE	Equivalent to PL/M-86
CONNECTION	A token representing a connection to a file or device. The CONNECTION specification assumes the following literal definition (in PL/M-86 terms): `DECLARE CONNECTION LITERALLY 'WORD' ;`
DWORD	Four byte unsigned integer.
POINTER	Equivalent to PL/M-86. Two bytes in the SMALL model of segmentation, four bytes in all others.
STRING	A sequence of bytes, the first of which contains the number of bytes following in the sequence, i.e., not including the length byte. A length of zero indicates the null string.
TOKEN	A word containing the means of locating an operating-system-object. The TOKEN specification assumes the following literal definition (in PL/M-86 terms): `DECLARE TOKEN LITERALLY 'WORD' ;`
WORD	Equivalent to PL/M-86.

The operating system follows the conventions for interfacing PL/M-86 programs with assembly language programs in preserving only registers CS, DS, SS, IP, SP, and BP on a call. Other registers and flags may be used by the operating system routines and, upon return to your program, have no predefined value.

External Procedure Definitions for Series III System Service Routines

Introduction

Any module using a service routine must first declare it as an external procedure and later link the final object module with the appropriate interface library. This section shows appropriate PL/M-86 declarations for all Series III routines, with syntax and usage examples. Appendix D provides a sample of similar declarations using 8086 assembly language.

Instead of an alphabetic listing, this section presents the routines in four categories:

- Utility/command parsing
- Memory management
- File management
- Exception handling

File management, has four subclasses:

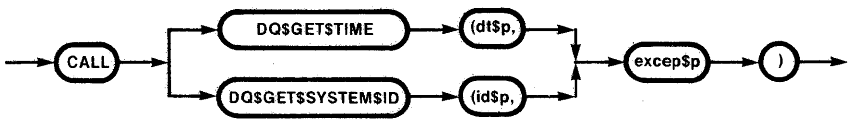
- Program connection to files
- Existence and naming of files
- Program usage of files
- Program control (overlays, exit)

The presentation of each category or subclass begins with a syntax/usage chart of the routines in it. These charts emphasize the commonality of many routines, partly by using a standard naming convention for parameters. These parameter names also clarify a natural order to the use of many service routines. For example, before you can validly invoke a procedure that *uses* a connection number (e.g., DQ$OPEN), you must first invoke one of the two routines that *provides* such a number: DQ$ATTACH or DQ$CREATE.

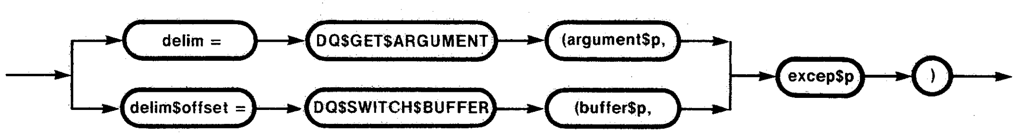
Appendix I lists all the routines and parameters in alphabetic order.

Utility and Command Parsing Service Routines

Two utility and three command-parsing routines are described in this section. The charts below illustrate how you invoke them.



Time and I.D. Utility Calls



Input-Line-Scanning Control Routines

Lower case entries indicate names, values, or pointers that you must create. Upper case words must be used exactly as shown. Where an equal sign appears, the service routine invoked on the right returns a byte or word value. This number, possibly modified if it is part of an expression, is then stored as the value of the variable to the left of the equal sign.

The routines shown are discussed below.

```
DQ$GET$TIME:  PROCEDURE (dt$p, excep$p) EXTERNAL;
        DECLARE dt$p     POINTER,
                excep$p POINTER  ;
        END  ;
```

Series III does not maintain a time log. However, this routine will return the system date, which can be set by the DATE command.

dt$p must be a pointer to a user structure of the form

```
DECLARE
DT STRUCTURE (  DATE (8) BYTE, TIME (8) BYTE) ;
```

DATE has the form MM/DD/YY for month, day, year. TIME is returned as 8 blanks.

Exception Codes Returned: EOK, EPTR

```
DQ$GET$SYSTEM$ID:  PROCEDURE (id$p, excep$p) EXTERNAL;
       DECLARE id$p      POINTER,
               excep$p   POINTER   ;
       END  ;
```

This routine returns a string identifying the operating system. *id$p* must point to a 21-byte buffer you define in your program.

The message returned is

```
    SERIES-III
```

Exception Codes Returned: EOK, EPTR

```
DQ$GET$ARGUMENT:  PROCEDURE (argument$p, excep$p) BYTE EXTERNAL;
       DECLARE argument$p    POINTER,
               excep$p       POINTER   ;
       END  ;
```

argument$p points to an 81-byte area you have declared to receive, as a string, an argument from the command tail (see discussion earlier in this chapter).

This typed procedure is used as a function (i.e., on the right side of an equal sign in a PL/M-86 assignment statement). The variable left of the equal sign is filled with the delimiter found by DQ$GET$ARGUMENT, as shown in the examples below. If the exception code is zero, all went well. The possible delimiters include

,) . (= # ! $ % ' \

~ + - & |] [> <; or DEL (RUB OUT)

or have a hexadecimal value of 0 to 20H.

This function returns the arguments in the command line or user-supplied buffer. Each successive call returns the next argument.

The command line is pre-scanned at the time your program is invoked. At that time the sytem makes the following changes to the command line before it is saved in a system buffer:

1. Each continuation line sequence is converted to a blank. A continuation line sequence begins with a "&" and ends with the line terminator.
2. A comment is removed entirely. A comment begins with a ";" and ends with the character preceding the line terminator.
3. Any "RUN" or "DEBUG" commands preceding the pathname are removed.
4. If the pathname command entered is an abbreviation, it is replaced by the correct pathname, e.g., PLM86 is replaced by PLM86.86. (See Appendix A.)

The following rules apply to the arguments and delimiters returned by DQ$GET$ARGUMENT:

1. Lower case alphabetic characters are converted to upper case, except inside strings.
2. Multiple adjacent blanks separating two arguments are treated as one blank. One or more blanks adjacent to any other delimiter are ignored. A tab is treated like a blank and returned as a blank.
3. Strings enclosed within a pair of matching quotes are considered literals and not scanned for interpretation. The enclosing quotes are not returned as part of the argument. Quotes may be included inside a quoted string by using quotes of the other type or doubling the quote.

The DQ$SWITCH$BUFFER routine (described next) may be used to get arguments from a user-supplied buffer. The command-line pre-scan is not performed on this buffer.

The following examples illustrate the argument returned by calls to DQ$GET$ARGUMENT:

A) `RUN PLM86 LINKER.PLM PRINT(:LP:) NOLIST`

ARGUMENT	DELIMITER
8,'PLM86.86'	' '
10,'LINKER.PLM'	' '
5,'PRINT'	'('
4,':LP:'	')'
6,'NOLIST'	CR

B)
```
RUN PLM86 MODULE.SRC PRINT(:F1:THISIS.IT)OPTIMIZE(0)
TITLE ('MY MODULE')
```

ARGUMENT	DELIMITER
8,'PLM86.86'	' '
10,'MODULE.SRC'	' '
5,'PRINT'	'('
13,':F1:THISIS.IT'	')'
8,'OPTIMIZE'	'('
1,'0'	')'
5,TITLE	'('
9,MY MODULE	')'
0	CR

C)
```
LINK :F4:X.OBJ,LLIB(MODL),SYSTEM.LIB,PUBLICS &
 (:F3:FUNNY.LIB(MOD1)) MAP; my link command
```

ARGUMENT	DELIMITER
7,'LINK.86'	' '
9,':F4:X.OBJ'	','
4,'LLIB'	'('
4,'MODL'	')'
0	','
10,'SYSTEM.LIB'	','
7,'PUBLICS'	'('
13,':F3:FUNNY.LIB'	'('
4,'MOD1'	')'
0	')'
3,'MAP'	CR

Usage Example:

```
DECLARE DELIM_SCAN BYTE, ERR WORD;
DECLARE NEXT_ARG STRUCTURE
            (LENGTH BYTE, ARG (80) BYTE);
DELIM_SCAN=DQ$GET$ARGUMENT
            (@NEXT_ARG.LENGTH, @ERR);
```

Exception Codes Returned: `E$OK, E$STRING$BUF, E$PTR`

```
DQ$SWITCH$BUFFER:  PROCEDURE (buffer$p, excep$p) WORD EXTERNAL;
        DECLARE buffer$p    POINTER,
                excep$p     POINTER  ;
        END  ;
```

This routine is used as a function.

The first time it is invoked, it returns zero after switching the scan pointer to the buffer whose address you supplied in the invocation. Thereafter, it returns the offset [from the start of the buffer] of the first character past the last delimiter returned by DQ$GET$ARGUMENT.

This routine should not be called until the command tail has been completely handled, as there is no way to switch back.

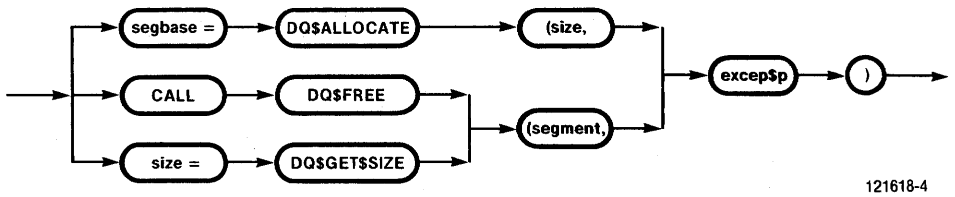
Example:

```
DECLARE NEXT_COUNT WORD, ERR WORD;
NEXT_COUNT=DQ$SWITCH$BUFFER (@ARGLIST, @ERR);
```

Exception Codes Returned: E$OK

Memory Management Routines

The chart below shows how you invoke the three memory management service routines.



Memory Management Routines

Lower case entries indicate names, values, or pointers that you must create. Upper case words must be used exactly as shown. Where an equal sign appears, the service routine invoked on the right returns a byte or word value. This number, possibly modified if it is part of an expression, is then stored as the value of the variable to the left of the equal sign.

The routines shown are discussed below.

Memory management routines can only be used to allocate and free blocks of memory and to get the size of any loaded or allocated segment.

```
DQ$ALLOCATE:  PROCEDURE (size, excep$p) TOKEN EXTERNAL;
        DECLARE size WORD,
                excep$p  POINTER   ;
        END  ;
```

size is the number of bytes of memory desired. A size of zero means a request for 64k bytes. If enough memory is available, this function returns a TOKEN representing the base of the acquired memory block, that is, the segment part of the pointer to the acquired area. (The offset of this pointer is zero.) If the operation fails, this TOKEN will be 0FFFFH.

DQ$ALLOCATE is used as a function, i.e., on the right of an equal sign in a PL/M-86 assignment statement. The variable left of the equal sign is filled with the connection number.

Example:

```
PAY_REC_STRUC_BASE = DQ$ALLOCATE(12,@ERROR_PAY) ;
IF (PAY_REC_STRUC_BASE  =  0FFFFH) THEN
            CALL MY_ERRCHK (PAY_REC_STRUC_BASE, LESS_MEM) ;
IF (ERROR_PAY <> 0) THEN
            CALL MY_ERRCHK (ERROR_PAY, MEM_ERR) ;
.....
.....
```

Exception Codes Returned: `E$OK, E$MEM`

```
DQ$FREE:  PROCEDURE (segment, excep$p) EXTERNAL;
       DECLARE segment    TOKEN,
               excep$p    POINTER    ;
       END  ;
```

segment is a TOKEN representing a memory segment acquired earlier from DQ$ALLOCATE. The indicated segment is freed. This should be done at the end of the task that allocated the segment, or whenever it will no longer be needed.

Example:

```
   CALL DQ$FREE (PAY_REC_STRUC_BASE,@ERROR_LESS)
```

Exception Codes Returned: `E$OK, E$EXIST`

```
DQ$GET$SIZE:  PROCEDURE (segbase, excep$p) WORD EXTERNAL;
       DECLARE segbase   TOKEN,
               excep$p   POINTER    ;
       END  ;
```

segbase is a TOKEN for a memory segment.

This function returns the size of the segment in bytes; zero means 64k bytes. This segment must have been previously allocated with the DQ$ALLOCATE routine.

If your program is relocatable, then the loader uses DQ$ALLOCATE to get the memory it requires from the memory manager. When you link your program, you can specify an "expanding segment", meaning that the size of the segment will be determined when the program is loaded. The size will depend upon the amount of memory available.

Relocatable PL/M-86 programs compiled under the SMALL model of segmentation can use an expanding data segment whose size can be determined with a statement of the form

```
   SIZE = DQ$GET$SIZE (STACKBASE, @EXCEP) ;
```

In SMALL model programs, the stack segment base and the data segment base are the same. Thus the above PL/M-86 statement passes the token representing the data segment base and obtains the data segment size.

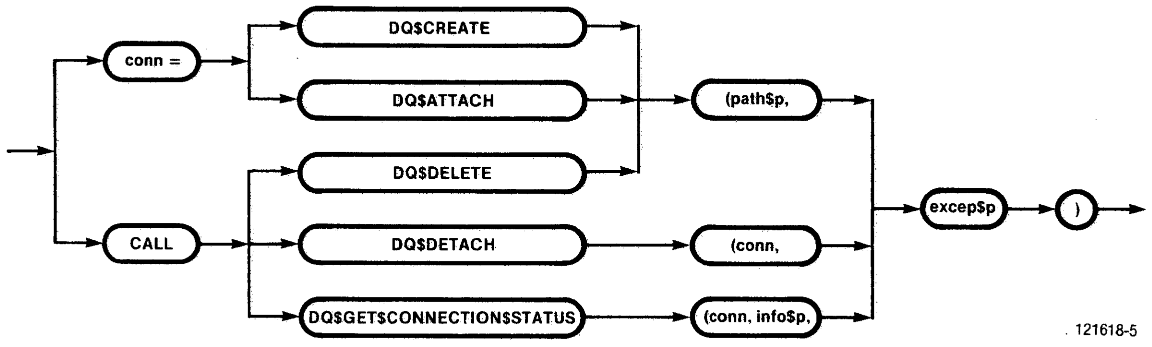
Example:

```
DECLARE ARRAY_SIZE WORD;
ARRAY_SIZE = DQ$GET$SIZE (ARRAY_BASE, @ERR);
```

Exception Codes Returned: `E$OK, E$EXIST`

Program Connection and File Existence Routines

The chart below shows how you invoke the routines controlling file existence and the connections between programs and files or devices. Output devices are created and input devices are attached; e.g., the console input device (:CI:) must be attached; the console output device (:CO:) must be created. Workfiles must be created, not attached. A file may be deleted only if no connection is currently established.



File/Program Connection Routines

Lower case entries indicate names, values, or pointers that you must create. Upper case words must be used exactly as shown. Where an equal sign appears, the service routine invoked on the right returns a byte or word value. This number, possibly modified if it is part of an expression, is then stored as the value of the variable to the left of the equal sign.

The routines shown are discussed below.

```
DQ$ATTACH:  PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
        DECLARE path$p   POINTER,
                excep$p  POINTER;
        END;
```

path$p points to a string containing a pathname. This string must begin with a number, telling how many characters will be given within the quotes that follow. This is the standard format for strings in the Series III operating system. (It differs from the format used under the ISIS-II operating system, where a string is not required to contain its length as the first element.) Examples appear below.

Only input devices, as well as disk files that are not workfiles, can be specified via the pathname. Attempting to attach :CO: (or :LP:), or to create :CI:, will cause an E$SUPPORT exception condition. (However, you may DQ$ATTACH, DQ$CREATE, or DQ$SEEK the Byte Bucket, :BB:.)

The console input device must be attached only via the :CI: pathname, and the console output device must be created only via the :CO: pathname. For example, assume the console input device is already assigned to the video input (:VI:). Then this device must not be attached via any other name: an attempt to attach :VI: is an error.

Also, an attempt to establish a connection on a currently executing user file containing overlays will cause an exception condition.

A maximum of twelve connections can be maintained by Series III. Only one connection can be established on a particular disk file, as there is only one current file pointer per file. However, multiple connections to physical (e.g., :LP:) and logical devices (e.g., :BB:, :CI:) are allowed.

DQ$ATTACH operates as a function, i.e., a typed procedure. It returns, in the variable to the left of the equal sign, a connection to an existing file. If the named file does not already exist, the operation will fail and return a non-zero error code at the address pointed to by *excep$p*. Similarly, if the file already has a connection established, the operation will fail and return a non-zero error code. DQ$ATTACH internally opens the file to check whether it exists and then closes it.

Examples:

```
DECLARE TAX_CONNECTION WORD, ERR WORD;
TAX_CONNECTION = DQ$ATTACH (@(14,':F1:FEDTAX.JUN') , @ERR);
```

Exception Codes Returned: `E$CONTEXT, E$PTR, E$FNEXIST, E$OK, E$SHARE, E$SIX, E$SYNTAX`

```
DQ$CREATE:  PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
       DECLARE path$p   POINTER,
               excep$p  POINTER;
       END;
```

path$p points to a string containing a pathname.

DQ$CREATE is a typed procedure, returning a connection to a new file. If a file of the same name exists and is not connected, it is deleted and a new file with this name is created. If such a file is write-protected or has the format attribute set, or already has a connection, this function will fail. A non-zero error code will then be returned in the location pointed to by *excep$p*. DQ$CREATE internally opens the file to check whether it exists or not and then closes it. This can impact the limit of six open connections.

Example:

```
DECLARE NEW_CONNECTION WORD, ERR WORD;
NEW_CONNECTION = DQ$CREATE ( @(10,'NEWTAX.AUG') , @ERR) ;

     /* A connection number will be created for the named*/
     /* file and stored in NEW_CONNECTION.              */
```

Exception Codes Returned: `E$CONTEXT, E$SHARE, E$FACCESS, E$OK, E$PTR, E$SIX, E$SYNTAX, E$SPACE`

```
DQ$DELETE:  PROCEDURE (path$p, excep$p) EXTERNAL;
       DECLARE path$p   POINTER,
               excep$p  POINTER;
       END;
```

path$p points to a string containing a pathname.

File deletion is a physical operation rather than a logical operation. The deletion actually occurs when this procedure is invoked, and an error will result if a connection is already established on the file to be deleted. This means the file must be detached if this current execution attached or created it earlier.

In addition, the supplied pathname must contain a file-name part, and the file to be deleted must not have the write-protect or format attributes set. It may not be the file to which the console has been assigned, nor a file which contains user overlays expected for use in the current execution. (In a name like :F1:MYPROG, the characters MYPROG constitute the file-name part. The characters :F1: constitute the device-name part. See also the *Intellec Series III Console Operating Instructions*.)

If these conditions are not met, this operation will fail and a non-zero error code will be returned in the location pointed to by *excep$p*.

Example:

```
CALL DQ$DELETE ( FILE$PTR, @ERR ) ;
     /* The file pointed to by FILE$PTR will be deleted, */
     /* assuming it meets the above conditions.          */
```

Exception Codes Returned: E$CONTEXT, E$FACCESS, E$SHARE, E$OK,
EPTR, ESUPPORT, E$SYNTAX, E$FNEXIST

```
DQ$DETACH:  PROCEDURE (conn, excep$p) EXTERNAL;
        DECLARE conn      CONNECTION,
                excep$p   POINTER;
        END;
```

conn is a token representing the connection to be detached, i.e., removed from the current list of attached devices-or-files.

DQ$DETACH breaks the connection created by DQ$ATTACH or DQ$CREATE. If the connection is open, it is closed before being detached.

Example:

```
CALL DQ$DETACH ( PAY_FILE_CONNECTION, @ERR);

     /* The file whose connection is PAY_FILE_CONNECTION */
     /* will be closed and removed from the list of      */
     /* connected/attached files, i.e., it will be detached. */
```

Exception Codes Returned: E$EXIST, E$OK, E$PARAM

```
DQ$GET$CONNECTION$STATUS:  PROCEDURE (conn, info$p, excep$p) EXTERNAL;
        DECLARE conn      CONNECTION,
                info$p    POINTER,
                excep$p   POINTER;
        END;
```

conn is a connection established earlier via attach or create. The parameter *info$p* points to a structure you have declared to receive the connection data found by this routine. For example:

```
   DECLARE INFO STRUCTURE (
        OPEN            BOOLEAN,
        ACCESS          BYTE,
        SEEK            BYTE,
        FILE$PTR$LOW    WORD,
        FILE$PTR$HIGH   WORD);
```

These fields bear the following interpretations:

`OPEN`

True if connection is open, otherwise false.

`ACCESS`

Access privileges of the connection. The rights are granted if the corresponding bit is on.

Bit	Access
0	delete
1	read
2	write
3	update

`SEEK`

Types of seek supported:

Bit	Seek Types
0	seek forward
1	seek backward

`FILE$TR$HIGH, FILE$PTR$LOW`

Current position of file pointer interpreted as a four-byte unsigned integer representing the number of bytes from the beginning of the file. This field is undefined if file is not open or if seek backward is not supported.

When the connection you specified is established on a physical or logical device, the access value returned depends on the nature of the device. For example, access privilege of the line printer is write. For a disk file with the write protect or format attributes set, access is read; for workfiles, access is read, write, and update. All other disk files have access privileges of delete, read, write, and update.

Physical devices and the console do not support any type of seek. For the byte bucket (:BB:), the returned file pointer is 0.

Example:

```
CALL DQ$GET$CONNECTION$STATUS (INVENTORY_CONN,
        @FILE_STATUS, @ERR);
```

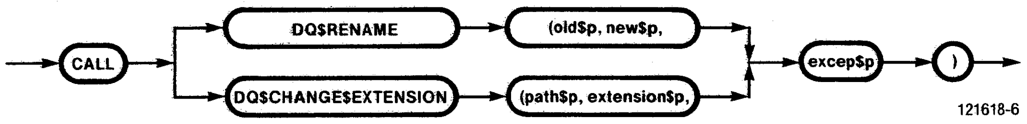
Exception Codes Returned: `E$EXIST, E$PARAM, E$OK, E$SHARE, E$PTR, E$SIX, E$FNEXIST`

NOTE

FILE$PTR$HIGH and FILE$PTR$LOW have the same meaning as the parameters in seek, named HIGH$OFFSET and LOW$OFFSET. The high word in each case has the high order bits of the four-byte integer.

File Naming Routines

The chart below shows how you invoke the system service routines that alter file names.



File Naming Routines

Lowercase entries above represent pointers you must create, indicating the locations for the pathname strings (and exception code) used in these routines. Uppercase words must be used as shown. The routines shown are discussed below.

```
DQ$RENAME:   PROCEDURE (old$p, new$p excep$p) EXTERNAL;
        DECLARE old$p    POINTER,
                new$p    POINTER,
                excep$p  POINTER;
        END;
```

old$p and *new$p* are pointers to the strings containing the existing pathname and the new pathname, respectively. There must be a filename part and the device parts must be identical, i.e., cross-volume renames are not supported. (In a name like :F1:MYPROG, the characters MYPROG constitute the file-name part. The characters :F1: constitute the device-name part. See also the *Intellec Series III Console Operating Instructions*.)

An exception condition occurs if a file with the new name already exists or if the file to be renamed has the write-protect or format attributes set. Renaming a file on which a connection is established is valid. It is not necessary that such a connection be established.

Example:

```
CALL DQ$RENAME(@FILE_PTR(3),@(9,'TERMS.NOV'),@ERR);
```

Exception Codes Returned: E$CROSSFS, E$FACCESS, E$FEXIST, E$FNEXIST, EOK, EPTR, E$SUPPORT, E$SYNTAX, E$SHARE, E$CONTEXT

```
DQ$CHANGE$EXTENSION:  PROCEDURE (path$p, extension$p, excep$p) EXTERNAL;
        DECLARE path$p        POINTER,
                extension$p   POINTER,
                excep$p       POINTER;
        END;
```

path$p points to a string containing the pathname to be changed. *extension$p* points to a three-character extension that is to become the extension in the pathname. These characters may not be delimiters. This procedure changes only the specified string and performs no file operations whatever.

DQ$CHANGE$EXTENSION replaces any existing extension on the file name with the supplied extension, e.g., :F4:FILE.SRC can be changed to :F4:FILE.OBJ or :F4:FILE.LST. If the first character addressed by *extension$p* is a blank, the file name will have any prior extension deleted, including the trailing period. (Trailing blanks are allowed, i.e., the third character or both the second and third characters of the new extension may be blanks.)

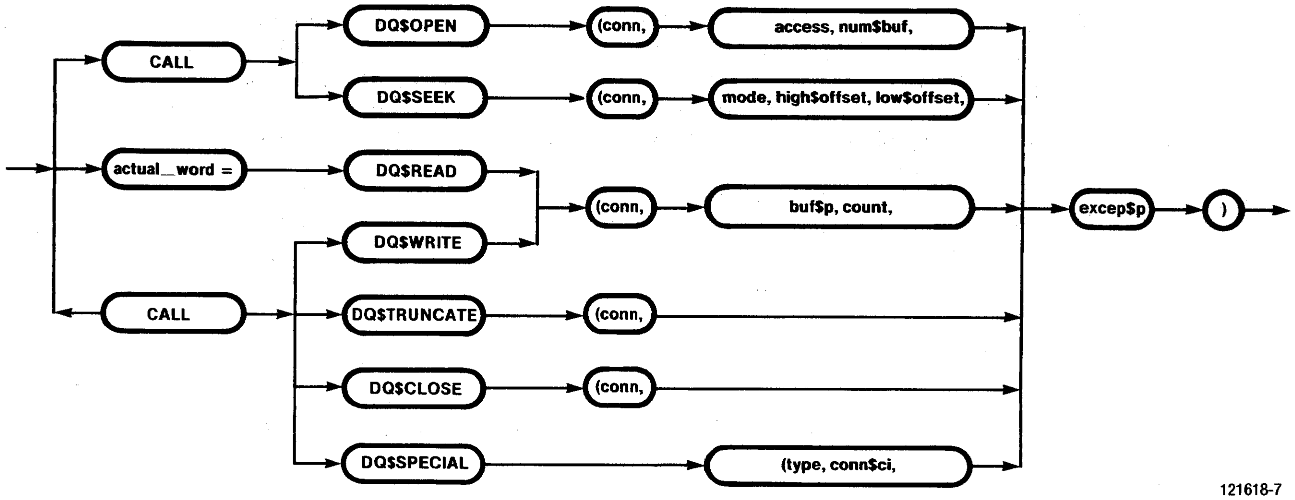
Examples:

```
CALL DQ$CHANGE$EXTENSION (@(8,'TASK.QRY'),@('ANS'),@ERR$P);
/*Filename string will be changed from TASK.QRY to TASK.ANS*/

CALL DQ$CHANGE$EXTENSION ( FILE$PTR, @('OBJ'), EXCEP$P) ;
        /* This will change the extension on the filename */
        /* pointed to by FILE$PTR to be .OBJ               */
```

Exception Codes Returned: `E$OK, E$STRING$BUF, E$PTR, E$SYNTAX`

File Usage Routines

The system service subroutines shown below provide the means for dynamic file usage during execution. Only the read routine is a function (typed procedure). All require a connection number established previously by DQ$ATTACH or DQ$CREATE.



File Usage Routines

Lower case entries indicate names, values, or pointers that you must create. Upper case words must be used exactly as shown. Where an equal sign appears, the service routine invoked on the right returns a byte or word value. This number, possibly modified if it is part of an expression, is then stored as the value of the variable to the left of the equal sign.

The routines shown are discussed below.

```
DQ$OPEN:  PROCEDURE (conn, access, num$buf, excep$p) EXTERNAL;
        DECLARE conn      CONNECTION,
                access    BYTE,
                num$buf   BYTE,
                excep$p   POINTER;
        END;
```

conn represents a connection established earlier via attach or create.

access specifies the type of access desired:

Value	Type
1	means read access only
2	means write access only
3	means update (both read and write)

num$buf indicates the optimal number of buffers, and should be 0, 1, or 2. Zero means that no buffering should occur—physical I/O should occur during a DQ$READ or DQ$WRITE. For seldom-used files, *num$buf* should be 1. In all other cases it should be 2 for double buffering. If program computation cannot be interspersed as described earlier in this chapter, *num$buf* should be 0 to maximize performance.

Files can be opened 'externally' using this routine or 'internally' as part of the operation of other system service routines. The limit of six open files-and-devices includes those opened 'internally' by the system.

Since it is possible to establish multiple connections on the same device, multiple opens of such a device are also permitted. The device still counts as only one open device on the list of six. The console input device counts toward this limit but the console output device does not.

The situations which do an 'internal' open include DQ$ATTACH, DQ$CREATE, DQ$TRUNCATE, DQ$OVERLAY, and entry to DEBUG-86. All but the last are explained under their entries in this chapter.

Entering DEBUG-86 internally opens :CI:.

If *access* is write or update, the file represented by the connection must not have the write-protect or format attributes set. The file pointer is set to 0, i.e., the beginning of the file. If the next access to this file is write, writing begins at the first byte, destroying earlier contents. To append, you must first read or seek to the end.

The use of DQ$OPEN must not violate physical limitations; e.g., the line printer must not be opened for read or update.

Example:

```
CALL DQ$OPEN (EMPLOYEE_CONN, 3, 2, @ERR);
```

Exception Codes Returned: E$EXIST, E$FACCESS, EOK, EPARAM, E$OPEN, E$SIX, E$SHARE, E$FNEXIST

```
DQ$SEEK:  PROCEDURE (conn, mode, high$offset, low$offset, excep$p) EXTERNAL;
        DECLARE conn          CONNECTION
                mode          BYTE
                low$offset    WORD,
                high$offset   WORD,
                excep$p       POINTER;
        END;
```

conn represents a currently open connection established earlier via attach or create.

mode indicates the type of seek required.

Value	Type
1	Move file pointer *back* by offset.
2	Set pointer *to* offset.
3	Move file pointer *forward* by offset.
4	Move file pointer to *end of file minus* offset.

When combined, *high$offset* and *low$offset* form a four-byte unsigned integer representing the number of bytes to move the file pointer. (Several of these parameters and interpretations differ slightly from those of ISIS-II; e.g., the offset interpretation for a SEEK under ISIS-II is block number and byte number—see Chapter 3.) If the seek goes beyond the end of file, a file opened for write or update will be extended with nulls; for a file opened for read, such a seek will position the pointer to the end of file. If a seek would move the pointer to before the start of the file, the pointer is set to the beginning of the file. Seeks are invalid on connections to physical devices or the console.

Example:

```
CALL DQ$SEEK (IONS_CONN, 3, 22, 248, @ERR);
```

This call does a seek from current position forward by an offset of 22 * 2^{16} + 248 bytes, on the file whose connection is IONS__CONN.

Exception Codes Returned: E$EXIST, E$NOPEN, EOK, EPARAM, E$SUPPORT

NOTE

high$offset and *low$offset* have the same meaning as the parameters in DQ$GET$CONNECTION$STATUS named *file$ptr$high* and *file$ptr$low*. The high word in each case has the high order bits of the four-byte integer. In the invocation of DQ$SEEK, *high$offset* is pushed onto the stack prior to *low$offset*.

```
DQ$READ:  PROCEDURE (conn, buf$p, count, excep$p) WORD EXTERNAL;
        DECLARE conn      CONNECTION,
                buf$p     POINTER,
                count     WORD,
                excep$p   POINTER;
        END;
```

conn represents an open connection established earlier via attach or create. *buf$p* points to a buffer area, at least *count* bytes long, that you have declared to receive the data read.

This routine is used as a function, i.e., a typed procedure returning the number of bytes actually transferred. This number will equal *count* unless an error occurred or an end of file was encountered.

count bytes are read from the current location of the file pointer and placed in your buffer. If the procedure returns a value of zero and an exception code of E$OK, end of file was encountered.

If your buffer is not long enough to receive the number of bytes requested, this routine will over-write the memory locations which follow the buffer.

DQ$READ will not recognize control C and control D as having any special meaning if the console has been assigned to a disk or device other than the cold start console. (See also DQ$TRAP$CC.)

Example:

```
DECLARE ACTUAL WORD ENTRIES (256) BYTE, ERR WORD;
ACTUAL = DQ$READ (JOURNAL_CONN, @ENTRIES(0), 256,
@ERR);
```

Exception Codes Returned: E$EXIST, E$NOPEN, EOK, EOWRITE, E$PARAM, E$PTR

```
DQ$WRITE:  PROCEDURE (conn, buf$p, count, excep$p) EXTERNAL;
      DECLARE  conn     CONNECTION,
               buf$p    POINTER,
               count    WORD,
               excep$p  POINTER;
      END;
```

conn represents an open connection established earlier via attach or create. Access must be write or update.

buf$p points to the start of the data to be written out.

count is the number of bytes to be written.

If the count exceeds the remaining length of your buffer, the contents of memory locations following the buffer will be written to the device or file represented by the connection you supplied.

Writing begins at the current value of the filepointer, which is 0 if no prior reads, seeks, or writes to this file have occurred.

A write to a pre-existing file you opened will destroy earlier contents unless the filepointer is first positioned (by a seek) at or after the end of file. A subsequent close, however, does *not* truncate the file: the original extent (or the new extent enlarged by seeks or writes) is maintained.

Example:

```
CALL DQ$WRITE (INVENTORY_CONN,@PHYSICAL,256, @ERR);
```

Exception Codes Returned: E$EXIST, E$NOPEN, EOK, EOREAD, E$PARAM, E$PTR

```
DQ$TRUNCATE:  PROCEDURE (conn, excep$p) EXTERNAL;
      DECLARE conn     WORD   ,
              excep$p  POINTER;
      END;
```

conn represents a connection established earlier via attach or create and currently open for write or update. This routine truncates the file represented by *conn* at the current file pointer and frees all previously allocated disk space beyond that pointer value. (If the pointer is at or past the end of file, truncation has no effect.) During the truncate operation, a workfile is opened and remains open until truncate completes.

Example:

```
CALL DQ$TRUNCATE (INTERIM_CONN, @ERR);
```

Exception Codes Returned: EOK, EEXIST, E$NOPEN, E$OWRITE, E$OREAD, E$SHARE, ESIX, ESPACE, E$PARAM

```
DQ$CLOSE:  PROCEDURE (conn, excep$p) EXTERNAL;
       DECLARE conn     CONNECTION,
               excep$p  POINTER;
       END;
```

conn represents a connection established earlier using attach or create, and opened via OPEN.

DQ$CLOSE waits for completion of input/output operations (if any) taking place on the file, ensures that output buffers are empty, and frees buffers. Once closed, a connection may be either re-opened or detached. Close does *not* truncate the file: the original extent (or the new extent enlarged by seeks or writes) is maintained.

Your programs must attach :CI:, create :CO:, open, close, and detach them. However, console output does not count toward the limit of six open files.

Example:

```
   CALL DQ$CLOSE (TAX_CONNECTION, @ERR);
           /* The file whose connection number is in    */
           /* TAX_CONNECTION will be closed.            */
```

Exception Codes Returned: E$EXIST, E$NOPEN, E$PARAM, E$OK

```
DQ$SPECIAL:  PROCEDURE (type, parameter$p, excep$p) EXTERNAL;
       DECLARE type          BYTE,
               parameter$p   POINTER,
               excep$p       POINTER;
       END;
```

This procedure is relevant only to console input. This differs from ISIS-II—see Chapter 3.

If *type* is one, subsequent console input is transparent, i.e., not line-edited. If *type* is two, subsequent console input will be line-edited. The initial default, when a job begins, is two. A *type* of three, is identical to a *type* of 1 except that a DQ$READ of :CI: will only return the single character already in the system buffer.

parameter$p must point to a connection representing a DQ$ATTACH of :CI:.

:CI: can be assigned to a disk file prior to initiating RUN, e.g., by a SUBMIT system call. If so, this routine returns E$SUPPORT if type one or type three is specified, even if :CI: is temporarily restored to the cold start console via control E.

A call to this routine changing :CI: from line-edited to transparent causes all characters currently in the physical console buffer to be discarded.

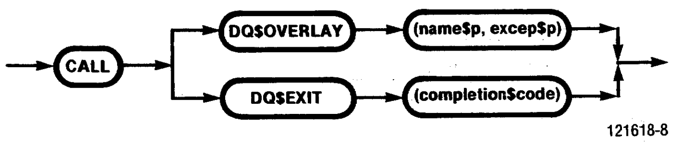
Example:

```
   CALL DQ$SPECIAL (1,@CI_CONN, @ERR);
```

Exception Codes Returned: EOK, EEXIST, E$PARAM, E$PTR, E$SUPPORT

Program Control Routines

The chart below shows how you invoke the exit and overlay routines.



Program Control Routines

Lower case entries above represent pointers or values you must create indicating the locations for the pathname strings (and exception and completion codes) used in these routines. The services named above are discussed below.

```
DQ$EXIT:   PROCEDURE (completion$code) EXTERNAL;
        DECLARE completion$code WORD;
        END;
```

Exit terminates a job. All files are closed and all resources are freed. If the RUN program was in control, it then prompts for another command.

The *completion$code* is intended to indicate whether termination was normal. Its values and interpretation are to be supplied by user programs if desired. This code is not referenced by the Series III operating system. (There is no exception pointer argument because this routine can never generate an exception.)

Example:

```
CALL DQ$EXIT (COMPL);
```

Exception Codes Returned: none

```
DQ$OVERLAY:   PROCEDURE (name$p, excep$p) EXTERNAL;
        DECLARE name$p    POINTER,
                excep$p   POINTER;
        END;
```

This routine causes the loading of the overlay whose name is the string that *name$p* points to. Only one level of overlays is allowed, and this routine may be called only from the root (non-overlaid) phase.

You define the overlay name with the LINK86 OVERLAY control. The name need not be restricted to ISIS-II filename conventions but it must not exceed 40 characters. The string used in the call must match the name used in LINK86.

(See the *iAPX 86, 88 Family Utilities User's Guide for 8086-Based Development Systems* for a full discussion of overlays.)

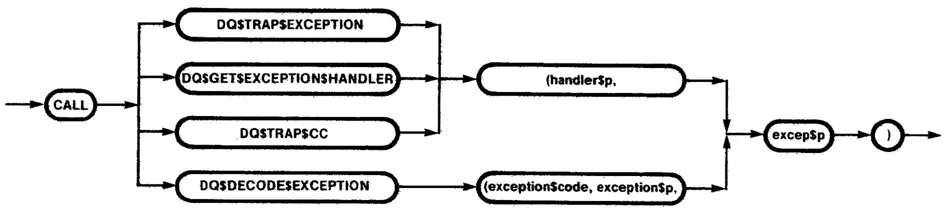
Example:

```
CALL DQ$OVERLAY (@(10'MYPROG.OV2'), @ERR);
```

Exception Codes Returned: EOK, EEXIST, E$PARAM, E$SIX, E$SYNTAX, E$UNSAT, E$ADDRESS, E$BAD$FILE

Exception Handling Routines

These four routines enable you to specify the routines you want called to handle exceptions of various types, and to decode the exception numbers returned by executing programs. The chart below shows how you invoke these system services.



Exception Handling Routines

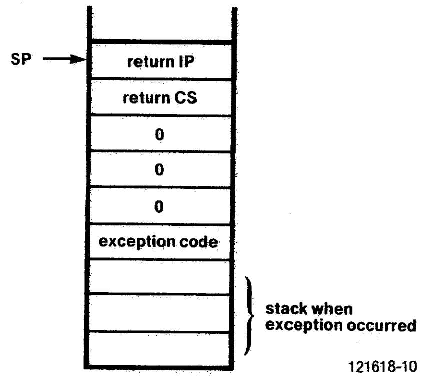
Lower case entries above represent pointers or values you must create, indicating the locations for the pathname strings (and exception and completion codes) used in these routines. The services named above are discussed below.

```
DQ$TRAP$EXCEPTION: PROCEDURE (handler$p, excep$p) EXTERNAL;
        DECLARE handler$p POINTER,
                excep$p   POINTER;
        END;
```

handler$p is the address of a four-byte area containing a long pointer to the entry point of your exception handler. (Programs compiled under the SMALL model of segmentation have no access to CS, and thus cannot create the long pointer directly.)

Hardware-detected exceptions will cause the exception handler to be executed. The state of the stack upon entry is as if the instruction pushed four words and then executed a long call to the exception handler. The first word pushed is the condition code. The next three words are reserved for future use by the operating system and numeric data processor.

Upon entry to the exception handler the stack looks like this:



See Appendix C for exception code values and descriptions.

The default system action is to display an error message, close files, and terminate program execution. The message format is

```
*** EXCEPTION nnnnH error message
    CS:IP = xxxx:yyyy
```

Example:

```
EXCEP_ROUT=DQ$TRAP$EXCEPTION (@USER_HANDLER, @ERR);
```

Exception Codes Returned: `E$OK`

```
DQ$GET$EXCEPTION$HANDLER: PROCEDURE (handler$p, excep$p) EXTERNAL;
         DECLARE handler$p  POINTER,
                 excep$p    POINTER;
         END;
```

handler$p must point to a four-byte area, which the system fills with a long pointer to the current avoidable-exception handler. A four-byte pointer is always returned, even if called from a program compiled under the SMALL model of segmentation.

This pointer will be the address specified in the last call of DQ$TRAP$EXCEPTION, if it has been called.

Example:

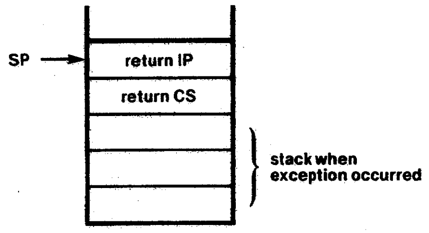
```
CALL DQ$GET$EXCEPTION$HANDLER (@WHICH_HANDLER,@ERR);
```

Exception Codes Returned: `E$OK, E$PTR`

```
DQ$TRAP$CC:  PROCEDURE (handler$p, excep$p) EXTERNAL;
         DECLARE handler$p POINTER,
                 excep$p   POINTER;
         END;
```

Whenever control-C is pressed at the physical console device, the system executes the default handler or your specially coded routine whose address is specified as *handler$p* via this system service call.

When your control-C routine receives control, the registers and flags are the same as the interrupted program. The stack looks like this:



Write the control-C routine in assembly language. The program must save the 8086 processor flags and registers and load the DS register with the control-C routine's data segment value. Before returning to the interrupted program, the control-C routine must restore the registers and flags and then execute a long return.

The default control-C handler closes files and terminates program execution. If the key was pressed while the system was performing a system service routine (other than a DQ$READ of :CI:), that routine is completed and the control-C routine is not executed until just before the system returns to the calling program. If a DQ$READ of :CI: is being serviced, the control-C handler is executed immediately and the DQ$READ returns an actual count of zero to the calling program.

(If :CI: has been redirected to another device or a disk file, e.g., under SUBMIT, the special meanings of control-C and control-D will not be recognized from the SUBMIT file. They will be treated as ordinary characters unless they come from the cold start console. However, if a control-E in the file temporarily redirects :CI: to the cold start console, then these characters temporarily lose their special meanings even from that console.)

Example:

```
CALL DQ$TRAP$CC ( SPECIAL_C_PTR,@ERR_C);
```

Exception Codes Returned: E$OK

```
DQ$DECODE$EXCEPTION:  PROCEDURE (exception$code, message$p, excep$p) EXTERNAL;
        DECLARE exception$code WORD,
                message$p      POINTER,
                excep$p        POINTER;
        END;
```

exception$code is a word containing an exception code. *message$p* is a pointer to the 81-byte area you declared to receive the error message decoded by the operating system. The first byte of the message is the length of the string. The word whose address is *excep$p* will contain zero unless an unexpected problem in decoding causes a non-zero code to be returned.

The routine returns a string containing

```
EXCEPTION nnnnH message
```

where nnnn is the exception code value and message is the exception message. If the *exception$code* you supply as a parameter is not a recognized system error number, then a question mark (?) appears instead of the message.

Example:

```
CALL DQ$CODE$EXCEPTION (ERRNO, @ERRMESS(0), @ERR);
```

Exception Codes Returned: E$OK

Usage Examples

Of the more than two dozen system service routines, most are invoked by CALL statements. A few are used as functions, returning a byte or word you can use in later statements. This section illustrates the usage of some of the more commonly invoked routines. (Some examples show more than one.)

Appendix I gives brief definitions for every parameter name used in the *general* procedure declaration for each routine.

The usage examples, however, declare variable names you might invent as actual parameters when you design programs to use the system services. Most of the examples are not complete programs or procedures, but merely indications of correct order and reasonable declarations, usage, etc.

Example of Allocate and Free

This example illustrates the use of DQ$ALLOCATE and DQ$FREE for a block of 1000 bytes. The base of this block, if allocation is successful, is returned by DQ$ALLOCATE and assigned to BASE_PARTS.SEGB. This word is the segment part of the pointer giving the address where the block begins. (The offset part of this address is always zero.)

```
PL/M-86 COMPILER    ALLOCFREEEXAMPLE                                                    PAGE   1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE ALLOCFREEEXAMPLE
OBJECT MODULE PLACED IN :F1:ALFREE.OBJ
COMPILER INVOKED BY:  :F2:PLM86 :F1:ALFREE.SRC


   1             ALLOC$FREE$EXAMPLE: DO;

   2    1        DQ$ALLOCATE: PROCEDURE (SIZE,EXCEPT$P) WORD EXTERNAL;
   3    2           DECLARE SIZE WORD, EXCEPT$P POINTER;
   4    2        END DQ$ALLOCATE;

   5    1        DQ$FREE: PROCEDURE (SEGMENT,EXCEPT$P) EXTERNAL;
   6    2           DECLARE SEGMENT WORD, EXCEPT$P POINTER;
   7    2        END DQ$FREE;


   8    1        DECLARE BASE$NAME POINTER;
   9    1        DECLARE (LIM,BYTES$WANTED,ERR) WORD;
  10    1        DECLARE ALLO LITERALLY '1', FRE LITERALLY '2';
  11    1        DECLARE BASE$PARTS STRUCTURE (OFFSETB WORD,SEGB WORD)
                                                 AT (@BASE$NAME) INITIAL (0,0);

  12    1        MY$ERR$CHK: PROCEDURE (SERVICE$CODE,FIXUP) EXTERNAL;
  13    2           DECLARE SERVICE$CODE BYTE, FIXUP WORD;
  14    2        END MY$ERR$CHK;

                 /* OTHER CODE HERE */

  15    1        LIM, BYTES$WANTED = 1000;
  16    1        BASE$PARTS.OFFSETB = 0;
  17    1        BASE$PARTS.SEGB = DQ$ALLOCATE (BYTES$WANTED,@ERR);
  18    1        IF ERR <> 0 THEN CALL MY$ERR$CHK (ALLO,LIM);

                 /* OTHER CODE HERE */
  20    1        CALL DQ$FREE (BASE$PARTS.SEGB,@ERR);
  21    1        IF ERR <> 0 THEN CALL MY$ERR$CHK (FRE,BASE$PARTS.SEGB);

                 /* MORE CODE HERE */

  23    1        END ALLOC$FREE$EXAMPLE;


MODULE INFORMATION:

     CODE AREA SIZE     = 0066H     102D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0008H       8D
     MAXIMUM STACK SIZE = 0006H       6D
     36 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

If allocation is unsuccessful, a non-zero error number will be placed in the location named "ERR". In these examples, the statement after each use of a system service routine checks the contents of ERR, and executes a user-written error recover routine if the error code is non-zero.

After the code using those 1000 bytes completes its task, a call to DQ$FREE releases the allocated block. If this operation does not complete normally, a non-zero error code is put into ERR. This is again tested by the next statement, which executes an error recovery routine if ERR is non-zero.

Example of GET$SIZE

This example shows a procedure to initialize an array that is based on a pointer supplied in its call. The segment word of this pointer is used by DQ$GET$SIZE to find out how large the array is, in bytes. If the service routine cannot complete successfully, it fills ERR with a non-zero error number. This causes the next statement in this example to execute your error-recovery routine with a code telling which service failed and where.

```
PL/M-86 COMPILER    GETSIZEEXAMPLE                                                      PAGE   1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE GETSIZEEXAMPLE
OBJECT MODULE PLACED IN :F1:GETSIZ.OBJ
COMPILER INVOKED BY:  :F2:PLM86 :F1:GETSIZ.SRC


   1             GET$SIZE$EXAMPLE: DO;

   2    1        DQ$GET$SIZE:  PROCEDURE (SEGMENT,EXCEPT$P) WORD EXTERNAL;
   3    2                      DECLARE SEGMENT WORD, EXCEPT$P POINTER;
   4    2                      END DQ$GET$SIZE;
```

```
  5   1        MYSERRSCHK: PROCEDURE (SERVICESCODE,CAUSE) EXTERNAL;
  6   2                    DECLARE SERVICESCODE BYTE, CAUSE WORD;
  7   2                    END MYSERRSCHK;

  8   1        ARRAYSINITIALIZE: PROCEDURE (BASESNAME);

  9   2        DECLARE BASESNAME POINTER;
 10   2        DECLARE (ARRAYSNAME BASED BASESNAME) (1) WORD;
 11   2        DECLARE SEGSOFFSET (2) WORD AT (@BASESNAME);
 12   2        DECLARE (LIMIT,ERR,I) WORD;
 13   2        DECLARE GET LITERALLY '3';

 14   2        LIMIT = DQSGETSSIZE (SEGSOFFSET(1),@ERR)/2;
 15   2        IF ERR <> 0 THEN CALL MYSERRSCHK (GET,SEGSOFFSET (1));
 17   2        DO I = 1 TO LIMIT;
 18   3          ARRAYSNAME (I) = 0;
 19   3        END;
 20   2        RETURN;

 21   2        END ARRAYSINITIALIZE;

 22   1        END GETSSIZESEXAMPLE;


MODULE INFORMATION:

     CODE AREA SIZE     = 0058H      88D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0006H       6D
     MAXIMUM STACK SIZE = 000AH      10D
     28 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Example of ATTACH, CREATE, OPEN, READ, WRITE, CLOSE, and DETACH

This example sets up a hypothetical set of files, connections, and codes to illustrate the proper order of input/output operations.

The first declarations include codes for

1. Error recovery, e.g., ATTA, CRE
2. Subscripts used with file pointers in creating connections, e.g., PAY, FTAX
3. A file size (PAYSIZE) and a synonym for the type "WORD" to identify names that will be connections

The hypothetical files are

1. Payroll data,
2. Tax data for California and Federal returns
3. Data on terminated employees.

Only the files for payroll and terminations are explicitly handled in the example.

There are several points illustrated:

1. Before service routines can be used in input/output operations, the appropriate pointers, buffers, connections, and pathnames must be declared and initialized.
2. Before files can be read or written, they must be opened via the DQ$OPEN procedure. Prior to opening, they must be attached via the DQ$ATTACH function (for existing files) or created via the DQ$CREATE function (for new files).
3. When processing for a given file is over, the file should be closed via the DQ$CLOSE procedure. If it will not be used again during this execution it may be detached via the DQ$DETACH procedure. This closes the file and removes its name from the list of connected files.
4. Error checking is advisable after any service routine use.

```
PL/M-86 COMPILER    READWRITEEXAMPLE                                                    PAGE   1


ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE READWRITEEXAMPLE
OBJECT MODULE PLACED IN :F1:REAWRI.OBJ
COMPILER INVOKED BY:  :F2:PLM86 :F1:REAWRI.SRC



   1             READ$WRITE$EXAMPLE: DO;

   2    1        DQ$ATTACH:  PROCEDURE (PATH$P, STATUS) WORD EXTERNAL;
   3    2                     DECLARE (PATH$P, STATUS) POINTER;
   4    2                     END DQ$ATTACH;

   5    1        DQ$CLOSE:  PROCEDURE (AFTN, STATUS) EXTERNAL;
   6    2                    DECLARE AFTN WORD, STATUS POINTER;
   7    2                    END DQ$CLOSE;

   8    1        DQ$CREATE:  PROCEDURE (PATH$P, STATUS) WORD EXTERNAL;
   9    2                     DECLARE (PATH$P, STATUS) POINTER;
  10    2                     END DQ$CREATE;

  11    1        DQ$DETACH:  PROCEDURE (CONNECTION, EXCEPT$P) EXTERNAL;
  12    2                     DECLARE CONNECTION WORD, EXCEPT$P POINTER;
  13    2                     END DQ$DETACH;

  14    1        DQ$OPEN:  PROCEDURE (AFTN, MODE, NUM$BUF, STATUS) EXTERNAL;
  15    2                  DECLARE AFTN WORD, STATUS POINTER;
  16    2                  DECLARE (MODE, NUM$BUF) BYTE;
  17    2                  END DQ$OPEN;

  18    1        DQ$READ:  PROCEDURE (AFTN, BUFFER, COUNT, STATUS) ADDRESS EXTERNAL;
  19    2                  DECLARE (AFTN, COUNT) WORD;
  20    2                  DECLARE (BUFFER, STATUS) POINTER;
  21    2                  END DQ$READ;

  22    1        DQ$WRITE:  PROCEDURE (AFTN, BUFFER, COUNT, STATUS) EXTERNAL;
  23    2                   DECLARE (AFTN, COUNT) WORD;
  24    2                   DECLARE (BUFFER, STATUS) POINTER;
  25    2                   END DQ$WRITE;
  26    1        MY$ERR$CHK:  PROCEDURE (SERVICE$CODE,CAUSE) EXTERNAL;
  27    2                     DECLARE SERVICE$CODE BYTE, CAUSE WORD;
  28    2                     END MY$ERR$CHK;


  29    1        DECLARE (ATTA,CRE,OPE,CLO,WRI,REA,DET) WORD EXTERNAL;

  30    1        DECLARE   READ         LITERALLY  '1',
                           WRITE        LITERALLY  '2',
                           READ$WRITE   LITERALLY  '3',
                           PAY          LITERALLY  '0',
                           CTAX         LITERALLY  '1',
                           FTAX         LITERALLY  '2',
                           TERM         LITERALLY  '3',
                           PAY$SIZE     LITERALLY  '128',
                           CONNECTION   LITERALLY  'WORD';

  31    1        DECLARE REC$SIZE (5) WORD INITIAL (PAY$SIZE);



PL/M-86 COMPILER    READWRITEEXAMPLE                                                    PAGE   2


  32    1        DECLARE FILE$PTRS (2) POINTER;
  33    1        DECLARE (PAY$READ$P,EMPL$TERM) POINTER,
                         PAY$BUF (PAY$SIZE) BYTE AT (@PAY$READ$P),
                         (GOT,COUNT) WORD INITIAL (0,0);
  34    1        DECLARE (PAY$FILE,CTAX$FILE,TERM$FILE) CONNECTION;
  35    1        DECLARE FILE$NAME$1 (*) BYTE AT (@FILE$PTRS (PAY))
                                         INITIAL (11,':F1:PAYROLL');
  36    1        DECLARE FILE$NAME$4 (*) BYTE AT (@FILE$PTRS (TERM))
                                         INITIAL (6,'TRMN8D');
  37    1        DECLARE (I,ERR) WORD;

  38    1        PAY$FILE = DQ$ATTACH (FILE$PTRS(PAY),@ERR);
  39    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (ATTA,'PA');
  41    1        TERM$FILE = DQ$CREATE (FILE$PTRS(TERM),@ERR);
  42    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (CRE,'TR');
  44    1        CALL DQ$OPEN (TERM$FILE,WRITE,2,@ERR);
  45    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (OPE,'TR');
  47    1        COUNT = REC$SIZE(PAY);
  48    1        CALL DQ$OPEN (PAY$FILE,READ,2,@ERR);
  49    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (OPE,'PA');
  51    1        GOT = DQ$READ (PAY$FILE,PAY$READ$P,COUNT,@ERR);
  52    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (REA,'PA');

  54    1        PAY1: DO I = 1 TO GOT;
                       /* ASSUME EMPL$TERM FILLED SOMEWHERE IN THIS DO */
  55    2              CALL DQ$WRITE (TERM$FILE,@EMPL$TERM,PAYSIZE,@ERR);
  56    2                 IF ERR <> 0 THEN CALL MY$ERR$CHK (WRI,'TR');
  58    2              END PAY1;

  59    1        CALL DQ$CLOSE (TERM$FILE,@ERR);
  60    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (CLO,'TR');
  62    1        CALL DQ$CLOSE (PAY$FILE,@ERR);
  63    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (CLO,'PA');
  65    1        CALL DQ$DETACH (TERM$FILE,@ERR);
  66    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (DET,'TR');
  68    1        CALL DQ$DETACH (PAY$FILE,@ERR);
  69    1           IF ERR <> 0 THEN CALL MY$ERR$CHK (DET,'PA');

                 /* ADDITONAL CODE HERE */

  71    1        END READ$WRITE$EXAMPLE;
```

```
MODULE INFORMATION:

     CODE AREA SIZE      = 019EH     414D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0020H      32D
     MAXIMUM STACK SIZE = 000AH      10D
     92 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Example Using GET$CONNECTIONS$STATUS, SEEK, WRITE, TRUNCATE, CHANGE$EXTENSION

(To be complete, this example would have to include several of the declarations of the prior example.) It shows a hypothetical typed procedure that

1. Checks the access privileges of the file whose connection is sent as a parameter (presumably write or update)
2. Seeks to a desired file position using a mode 4 seek, which goes to the end of file minus a number of bytes, defined by another parameter
3. Writes over the file at that point using data from a buffer address supplied in the call
4. Truncates the file to eliminate any old data left later in the file beyond what this write replaced
5. Changes the extension of the file (a pointer to the name is a parameter) to reflect the date of these changes. (Error processing is not shown.)
6. If all goes well, the procedure returns a zero. Errors along the way cause returns of non-zero codes.

```
PL/M-86 COMPILER    GETCONNEXAMPLE                                                   PAGE   1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE GETCONNEXAMPLE
OBJECT MODULE PLACED IN :F1:LATEST.OBJ
COMPILER INVOKED BY:  :F2:PLM86 :F1:LATEST.SRC


  1            GET$CONN$EXAMPLE: DO;

  2   1        DQ$CHANGE$EXTENSION: PROCEDURE (PATH$P,EXTENSION$P,EXCEPT$P) EXTERNAL;
  3   2                             DECLARE (PATH$P,EXTENSION$P,EXCEPT$P) POINTER;
  4   2                             END DQ$CHANGE$EXTENSION;

  5   1        DQ$GET$CONNECTION$STATUS: PROCEDURE (CONNECTION,INFO$P,EXCEPT$P) EXTERNAL;
  6   2                                  DECLARE CONNECTION WORD;
  7   2                                  DECLARE (INFO$P,EXCEPT$P) POINTER;
  8   2                                  END DQ$GET$CONNECTION$STATUS;

  9   1        DQ$GET$TIME: PROCEDURE (DT$P,EXCEPT$P) EXTERNAL;
 10   2                     DECLARE (DT$P,EXCEPT$P) POINTER;
 11   2                     END DQ$GET$TIME;

 12   1        DQ$SEEK: PROCEDURE (CONNECTION,MODE,HIGH$OFFSET,LOW$OFFSET,EXCEPT$P) EXTERNAL;
 13   2                 DECLARE (CONNECTION,HIGH$OFFSET,LOW$OFFSET) WORD;
 14   2                 DECLARE EXCEPT$P POINTER;
 15   2                 DECLARE MODE BYTE;
 16   2                 END DQ$SEEK;

 17   1        DQ$TRUNCATE: PROCEDURE (CONNECTION,EXCEPT$P) EXTERNAL;
 18   2                     DECLARE CONNECTION WORD, EXCEPT$P POINTER;
 19   2                     END DQ$TRUNCATE;

 20   1        DQ$WRITE: PROCEDURE (CONNECTION,BUF$P,COUNT,EXCEPT$P) EXTERNAL;
 21   2                  DECLARE (CONNECTION,COUNT) WORD;
 22   2                  DECLARE (BUF$P,EXCEPT$P) POINTER;
 23   2                  END DQ$WRITE;

 24   1        LATEST$SUMMARIES: PROCEDURE (LEDGER$CONN,
                                            LEDGER$P,
                                            MTD$COUNT,
                                            NEW$MTD$P,
                                            NEW$COUNT,
                                            NEW$EXT$P) WORD;

 25   2        DECLARE (LEDGER$CONN,MTD$COUNT,NEW$COUNT,ERR) WORD;
 26   2        DECLARE (NEW$MTD$P,NEW$EXT$P,LEDGER$P) POINTER;
 27   2        DECLARE FILE$DATA STRUCTURE (OPEN BYTE,ACCESS BYTE,SEEK BYTE,LOW$OFF WORD,
                                            HIGH$OFF WORD);
 28   2        DECLARE DATER STRUCTURE (DATE (8) BYTE, TIME (8) BYTE);
 29   2        CALL DQ$GET$CONNECTION$STATUS (LEDGER$CONN,@FILE$DATA,@ERR);
 30   2          IF ERR <> 0 THEN RETURN 1;  /* NEED STATUS TO OPERATE */
 32   2        IF FILE$DATA.OPEN THEN        /* SAFELY; FILE MUST BE OPEN */
 33   2          DO;
 34   3            IF FILE$DATA.ACCESS < 4 THEN RETURN 2;
                   /* ACCESS MUST BE WRITE OR UPDATE */
 36   3            IF NOT FILE$DATA.SEEK THEN RETURN 3;
                   /* SEEK FORWARD MUST BE SUPPORTED */


PL/M-86 COMPILER    GETCONNEXAMPLE                                                   PAGE   2

 38   3            CALL DQ$SEEK (LEDGER$CONN,4,0,MTD$COUNT,@ERR);
                   /* TO ENDFILE LESS COUNT */
 39   3              IF ERR <> 0 THEN RETURN 4;
 41   3            CALL DQ$WRITE (LEDGER$CONN,@NEWS$MTD$P,NEWS$COUNT,@ERR);
                   /* REPLACE SUMMARIES WITH LATEST */
 42   3              IF ERR <> 0 THEN RETURN 5;
 44   3            CALL DQ$TRUNCATE (LEDGER$CONN,@ERR);
                   /* END THE FILE AFTER CURRENT DATA */
 45   3              IF ERR <> 0 THEN RETURN 6;
 47   3            CALL DQ$GET$TIME (@DATER,@ERR);
 48   3              IF ERR <> 0 THEN RETURN 7;
 50   3            DATER.DATE (2) = ' ';
 51   3            CALL DQ$CHANGE$EXTENSION (@LEDGER$P,@DATER.DATE(0),@ERR);
 52   3              IF ERR <> 0 THEN RETURN 8;
 54   3            RETURN 0;
 55   3          END;
                ELSE
 56   2          RETURN 9;

 57   2        END LATEST$SUMMARIES;

 58   1        END GET$CONN$EXAMPLE;



MODULE INFORMATION:

     CODE AREA SIZE     = 010AH     266D
     CONSTANT AREA SIZE = 0000H       0D
     VARIABLE AREA SIZE = 0019H      25D
     MAXIMUM STACK SIZE = 001AH      26D
     73 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

Echo Program Example

This program example enables the user to type in characters from a console. The console output device then echoes back the same characters that were typed in.

ERR$CHK is a procedure whereby a check is made after each service routine to determine if the routine has executed properly. If an error is found, the routine DQ$DECODE$EXCEPTION returns a string of characters containing a formatted error message. This error message, in turn, is displayed to the console output device before an exit from the program is performed.

BEGIN is the start of the program whereby a sign-on message is displayed to the user.

AGAIN is the main loop of the program, where the writing and reading is done. If the first character typed in by the user is an 'E', an exit will be performed.

```
PL/M-86 COMPILER    ECHOPGM                                                          PAGE   1

ISIS-II PL/M-86 V2.0 COMPILATION OF MODULE ECHOPGM
OBJECT MODULE PLACED IN :F1:EPG.OBJ
COMPILER INVOKED BY:  :F2:PLM86 :F1:EPG.SRC


  1             ECHO$PGM: DO;

  2   1         DQ$ATTACH:  PROCEDURE (PATH$P, STATUS) WORD EXTERNAL;
  3   2                     DECLARE (PATH$P, STATUS) POINTER;
  4   2                     END DQ$ATTACH;

  5   1         DQ$CLOSE:  PROCEDURE (AFTN, STATUS) EXTERNAL;
  6   2                    DECLARE AFTN WORD, STATUS POINTER;
  7   2                    END DQ$CLOSE;
  8   1      DQ$CREATE:  PROCEDURE (PATH$P, STATUS) WORD EXTERNAL;
  9   2                  DECLARE (PATH$P, STATUS) POINTER;
 10   2                  END DQ$CREATE;

 11   1      DQ$DECODE$EXCEPTION:  PROCEDURE (ERRNUM, EXCEPTION$P, STATUS) EXTERNAL;
 12   2                            DECLARE ERRNUM WORD;
 13   2                            DECLARE (EXCEPTION$P, STATUS) POINTER;
 14   2                            END DQ$DECODE$EXCEPTION;

 15   1      DQ$DETACH:  PROCEDURE (CONNECTION, EXCEPT$P) EXTERNAL;
 16   2                  DECLARE CONNECTION WORD, EXCEPT$P POINTER;
 17   2                  END DQ$DETACH;

 18   1      DQ$EXIT:  PROCEDURE (COMPLETION$CODE) EXTERNAL;
 19   2                DECLARE COMPLETION$CODE WORD;
 20   2                END DQ$EXIT;

 21   1      DQ$OPEN:  PROCEDURE (AFTN, MODE, NUM$BUF, STATUS) EXTERNAL;
 22   2                DECLARE AFTN WORD, STATUS POINTER;
 23   2                DECLARE (MODE, NUM$BUF) BYTE;
 24   2                END DQ$OPEN;

 25   1      DQ$READ:  PROCEDURE (AFTN, BUFFER, COUNT, STATUS) ADDRESS EXTERNAL;
 26   2                DECLARE (AFTN, COUNT) WORD;
 27   2                DECLARE (BUFFER, STATUS) POINTER;
 28   2                END DQ$READ;

 29   1      DQ$WRITE:  PROCEDURE (AFTN, BUFFER, COUNT, STATUS) EXTERNAL;
 30   2                 DECLARE (AFTN, COUNT) WORD;
 31   2                 DECLARE (BUFFER, STATUS) POINTER;
 32   2                 END DQ$WRITE;


 33   1      DECLARE (STATUS,ACTUAL) ADDRESS;
 34   1      DECLARE STATUS$$PTR (80) BYTE;
 35   1      DECLARE BUFFER (128) BYTE;
 36   1      DECLARE CR$LF (2) BYTE DATA (0DH,0AH);
 37   1      DECLARE SIGN$ON$MSG (*) BYTE INITIAL
             ('THIS TEST PROGRAM ECHOES THE CHARACTERS YOU HAVE TYPED IN...',0DH,0AH,
              'TO EXIT TYPE IN "E" <CR>',0DH,0AH,0AH);
 38   1      DECLARE (R$CONN,W$CONN) ADDRESS PUBLIC;
```

```
PL/M-86 COMPILER    ECHOPGM                                                      PAGE   2


 39   1      ERR$CHK: PROCEDURE PUBLIC;
 40   2        IF STATUS <> 0 THEN
 41   2          DO;
 42   3            CALL DQ$DECODE$EXCEPTION (STATUS,@STATUS$$PTR,@STATUS);
 43   3            CALL DQ$WRITE (W$CONN,@STATUS$$PTR(1),STATUS$$PTR(0),@STATUS);
 44   3            CALL DQ$WRITE (W$CONN,@CRLF,2,@STATUS);
 45   3            CALL DQ$EXIT (0);
 46   3          END;
 47   2      END ERR$CHK;

 48   1      W$CONN = DQ$CREATE (@(4,':CO:'),@STATUS);
 49   1      CALL ERR$CHK;
 50   1      R$CONN = DQ$ATTACH (@(4,':CI:'),@STATUS);
 51   1      CALL ERR$CHK;

 52   1      CALL DQ$OPEN (W$CONN,2,0,@STATUS);
 53   1      CALL ERR$CHK;
 54   1      CALL DQ$OPEN (R$CONN,1,0,@STATUS);
 55   1      CALL ERR$CHK;

 56   1      BEGIN: /* WRITE TO THE CONSOLE THE SIGN-ON MESSAGE FOR THE USER */
             CALL DQ$WRITE (W$CONN,@CR$LF,2,@STATUS);
 57   1      CALL DQ$WRITE (W$CONN,@CR$LF,2,@STATUS);
 58   1      CALL ERR$CHK;
 59   1      CALL DQ$WRITE (W$CONN,@SIGN$ON$MSG,LENGTH(SIGN$ON$MSG),@STATUS);
 60   1      CALL ERR$CHK;

 61   1      AGAIN: /* BEGIN ECHO LOOP */
             CALL DQ$WRITE (W$CONN,@CR$LF,2,@STATUS);
 62   1      CALL ERR$CHK;
 63   1      CALL DQ$WRITE (W$CONN,@('?  '),3,@STATUS);
 64   1      CALL ERR$CHK;
 65   1      ACTUAL = DQ$READ (R$CONN,@BUFFER,128,@STATUS);
 66   1      CALL ERR$CHK;
 67   1      IF BUFFER(0) = 'E' THEN
 68   1        DO;
 69   2          CALL DQ$CLOSE (W$CONN,@STATUS);
 70   2          CALL ERR$CHK;
 71   2          CALL DQ$CLOSE (R$CONN,@STATUS);
 72   2          CALL ERR$CHK;
 73   2          CALL DQ$DETACH (W$CONN,@STATUS);
 74   2          CALL ERR$CHK;
 75   2          CALL DQ$DETACH (R$CONN,@STATUS);
 76   2          CALL ERR$CHK;
 77   2          CALL DQ$EXIT (0);
 78   2        END;
             ELSE
 79   1        DO;
 80   2          CALL DQ$WRITE (W$CONN,@('=> '),3,@STATUS);
 81   2          CALL ERR$CHK;
 82   2          CALL DQ$WRITE (W$CONN,@BUFFER,ACTUAL,@STATUS);
 83   2          CALL ERR$CHK;
 84   2          GOTO AGAIN;
 85   2        END;

 86   1      END ECHO$PGM;
```

```
PL/M-86 COMPILER    ECHOPGM                                                    PAGE   3



MODULE INFORMATION:

     CODE AREA SIZE     = 01ACH     428D
     CONSTANT AREA SIZE = 0012H      18D
     VARIABLE AREA SIZE = 0131H     305D
     MAXIMUM STACK SIZE = 000EH      14D
     109 LINES READ
     0 PROGRAM ERROR(S)

END OF PL/M-86 COMPILATION
```

CHAPTER 3
ISIS-II AND MONITOR SERVICE ROUTINES

The routines available under ISIS-II and the Monitor are listed alphabetically in table 3-1.

As shown in table 1-3, many are similar to those available under Series III.

Under ISIS-II, there are 15 system service routines: 7 in file management, 3 in maintenance of disk directories, 3 for console control, and 2 for transfer of execution control.

Under the Monitor, there are 13 additional services: 7 input/output (I/O) interface routines, 3 status checks, and 3 services relating to I/O device configuration. The Monitor is described more fully in the *Intellec Series III Console Operating Instructions*.

The routines have been grouped by function to make it easy to see the features they have in common. Table 3-1 indicates in what group you can find the discussion of each routine. (M means monitor routine.)

Table 3-1. Index from Routine to Discussion

ISIS-II and MONITOR Routines	Discussed in Group Named
ATTRIB	Disk Directory Maintenance
CI	(M) Console Input
CLOSE	File Management
CO	(M) Console Output
CONSOL	Console Control
CSTS	(M) Console Input Status
DELETE	Disk Directory Maintenance
ERROR	Console Control
EXIT	Execution Control
IOCHK	(M) Configuration Check
IODEF	(M) User-Defined Devices
IOSET	(M) Configuration Set
LO	(M) List Output
LOAD	Execution Control
MEMCK	(M) RAM Size Check
OPEN	File Management
PO	(M) Punch Output
READ	File Management
RENAME	Disk Directory Maintenance
RESCAN	File Management
RI	(M) Reader Input
SEEK	File Management
SPATH	File Management
UI	(M) UPP Input (Series II only)
UO	(M) UPP Output (Series II only)
UPPS	(M) UP Status (Series II only)
WHOCON	Console Control
WRITE	File Management

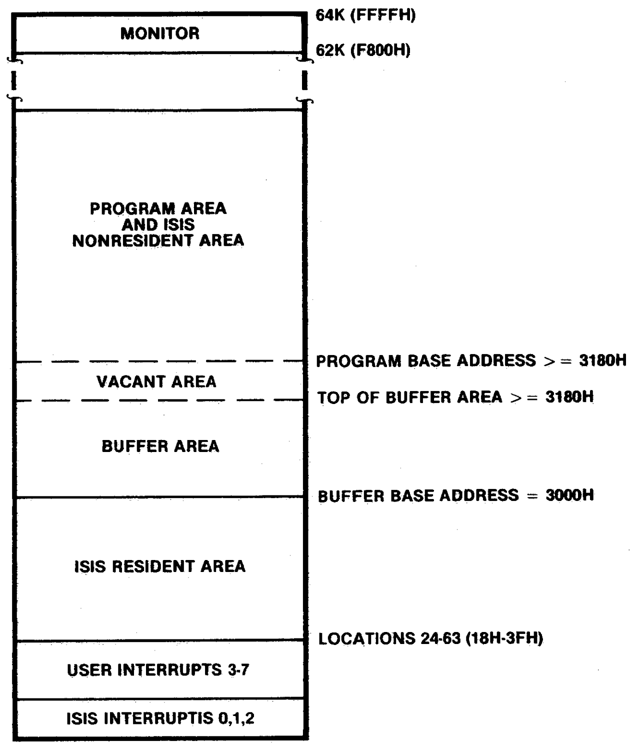
For effective use of system resources, e.g., memory and disk files, you must know the parameters expected by these routines and the assumptions that validate them. Further, your choice of where in memory to put your program must take into consideration how memory is organized under ISIS-II.

This organization is discussed in the next section, service parameters and assumptions in the one following, and line-edited input in the next, before the detailed discussions of each routine.

The UI, UO, and UPPS routines do not apply to the Model 800. See the UPP control commands (Compare, Program PROM, and Transfer PROM) described in the *Intellec 800 Microcomputer Development System Operator's Manual* (9800129).

Memory Organization and Allocation

The organization of Intellec memory under ISIS-II is shown in the following illustration:


MONITOR
64K (FFFFH)
62K (F800H)
PROGRAM AREA AND ISIS NONRESIDENT AREA
PROGRAM BASE ADDRESS >= 3180H
VACANT AREA
TOP OF BUFFER AREA >= 3180H
BUFFER AREA
BUFFER BASE ADDRESS = 3000H
ISIS RESIDENT AREA
LOCATIONS 24-63 (18H-3FH)
USER INTERRUPTS 3-7
ISIS INTERRUPTIS 0,1,2

Interrupt Vectors

Interrupts 0 through 2 are reserved for ISIS-II and the Monitor. Interrupts 3 through 7 are available for use within your program. However, other Intel software products can also use interrupts 3 through 7. When using interrupt, you must be sure to avoid conflicts with Intel software. These locations, 24 through 63, are the only locations below 12k (3000H) that can be loaded with user code. Loading other locations below 12k is not allowed.

The Kernel

The ISIS-II resident area is reserved for the kernel, that part of ISIS-II that is always resident in RAM memory. It may be viewed as a collection of subroutines. Although the kernel is protected from a program load operation, it is not protected from an executing program, which may accidentally destroy the integrity of ISIS-II by writing into this area. This will cause subsequent errors when system services are requested.

Input and Output to Files

User programs perform input/output (I/O) by making calls to the kernel, i.e., system calls. All I/O occurs to or from "files" and is status-driven rather than interrupt-driven. Interrupts 0, 1, and 2 are reserved for ISIS-II and must not be masked or altered by your programs.

A "file" is an abstraction of an I/O device, considered to be a collection of information, usually in machine-readable form. A file can be formally defined as a sequence of 8-bit values called "bytes."

ISIS-II usually places no semantic interpretation on the byte values of a file, with the exception of lined files (see below). Programmers, programs, and devices frequently interpret the bytes as representing ASCII values, and thereby characters.

Programs receive information by "reading" from an "input file" and transmit information by "writing" to an "output file."

A major purpose of ISIS-II is to implement files (called "disk files") on diskettes/platters.

"Diskette" is the recording medium for floppies while "platter" is the recording medium for the hard disk; "drive" is the mechanism on which the medium is mounted; "disk" is the drive together with a mounted diskette/platter. Where the meaning is unambiguous, "disk" is often used in place of "diskette" and "platter."

Every disk file is identified by a name unique on its diskette/platter, which has 2 parts: a filename and an optional extension. A disk file's filename is a sequence of from 1 to 6 ASCII characters; an extension is a sequence of from 1 to 3 ASCII characters. To facilitate name specification within command strings, these ASCII characters are constrained to be letters and/or digits.

For every non-disk device supported by ISIS-II, there are one or more associated files, each identified by a name consisting of a pair of ASCII characters between colons (see Appendix G for a complete list). Disk drives also have names, which are used as prefixes to filenames, specifying where the file resides.

No file can exist on more than 1 physical device. In particular, a disk file must reside entirely on one diskette/platter.

Three files (:BB:, :CI: and :CO:) deserve special mention:

ISIS-II supports a virtual input/output device known as a "Byte Bucket (:BB:). This device acts as an infinite sink for bytes when written to, and a file of zero length when read from. Multiple opening of :BB: is allowed; each open returns a different connection number (AFTN). (See below.)

ISIS-II supports a virtual console known as "the Console," which is implemented as 2 files, an input file (:CI:) and an output file (:CO:). These 2 files are always "open." :CI: is always a "lined file," :CO: is its associated "echo file." (Quoted terms are discussed later in this section.) Each is a pseudonym for some file corresponding to an actual physical device. After a cold start of ISIS-II (see *Intellec Series III Console Operating Instructions*), :CI: and :CO: will reference either the teletype (:TI: and :TO:) or the video terminal (:VI: and :VO:), which will be called the "cold start Console." User programs may change the definition of the two halves of the Console from one physical device to another.

Whenever an end of file is encountered on :CI:, then both :CI: and :CO: are automatically redefined to the cold start Console.

It is always from the current Console that the Command Line Interpreter (CLI) obtains its command lines. Briefly, each command contains a generalized keyword which specifies either a user program or CLI command. If the former, CLI causes that program to be loaded and run; otherwise CLI performs the indicated command and reads another command line.

Buffers

The buffer area is used by ISIS-II for input/output buffers of 128 bytes each. One permanent buffer is used by ISIS-II for console input/output. Other buffers are allocated and deallocated dynamically for you by ISIS-II according to the input/output requirements of your program. These requirements come from your explicit system calls to ISIS-II or are derived from your source code by the translator you used (e.g., PL/M or assembler).

The minimum size of the buffer area is 384 bytes, allowing for three buffers including the ISIS-II permanent buffer. If your program requires more than two buffers, the buffer area increases at the expense of the vacant area.

Computing Program Base Address

Program Area

The program area is above the buffer area. It is used alternately by your programs and ISIS-II.

Nonresident ISIS-II routines (all commands except DEBUG) run in the program area. These include the command interpreter, the editor, assembler, compiler, linker, locater, and library manager. Whenever you communicate with ISIS-II via console commands, you are using the nonresident command interpreter running in the program area. These nonresident ISIS-II routines may use all available RAM for buffers.

Therefore, if your programs must be permanently resident, they should be placed in ROM, or in RAM that is physically not next to the first contiguous block of RAM.

Monitor Area

The Monitor occupies the top 2k of memory. Addresses F800H to FFFFH are shadowed by the Monitor ROM. If your memory configuration encompasses those addresses, they cannot be written. The Monitor also uses the top 320 bytes of contiguous RAM for its workspace.

The first 32k of memory must be RAM. Above 32k, memory can be any combination of ROM and RAM. The Monitor MEMCK routine can be called if a program needs to know the highest available location of contiguous RAM (below the Monitor workspace).

Base Address of Your Program

You determine your program's base address by commands to LOCATE (or by an ORG statement in 8080/8085 assembly language absolute programs).

Before deciding the base address, you must determine the maximum area required for buffers under ISIS-II. The number of buffers varies during execution, but the buffer area must be large enough for the maximum number of buffers allocated simultaneously.

If you locate the base address of your program below 3180H (or allocate fewer than 3 buffers), an error message is generated. If you allocate more than 19 buffers, an error message is generated.

The program base address can be calculated using the following formula:

```
12,288 + (128 * N)
```

where N is the maximum number of buffers required simultaneously by your program. Use the following rules to determine N:

1. Each open disk file requires two buffers until the file is closed.
2. An open line-edited file (e.g., :CI:) requires one buffer until the file is closed. For a disk file, this buffer is in addition to the two required in rule 1.
3. ISIS-II processing of system calls requires two buffers.
4. If a CONSOL system call assigns the console input and output to a disk file, three buffers are required for the console input file and two more for the console output file. These buffers are required until the program exits or otherwise terminates execution. If you plan to have a program called by a system command in a SUBMIT file, you must also allow for these buffers in determining where in memory to put the origin point of that program.

Example 1:

A program that has no system calls, does not assign the console to a disk file, and is not called by a command in a disk file, needs three buffers. Therefore, it can have a base address of 3180H. If the program is changed to open one disk file (other than :CI:), it needs five buffers, and the base address must be 3280H:

```
3000H + (128 * 5) = 3280H ; 3000H = 12288 decimal
```

If this same program were to be called from a SUBMIT file and then defined the console output device also as a disk file, four more buffers would be needed, requiring an address of 3480H:

```
3000H + (128 * 9) = 3480H
```

Example 2:

Suppose a program opens a line-edited disk file (3 buffers) and an echo file on disk (2 buffers). Assuming console input is open but not a disk file, three buffers are required for console and system use. The program origin point is calculated as follows:

```
3000H + (128 * 8) = 3400H
```

NOTE

If you want to write a program independent of the type of device used for data transfers and also independent of how it is called (from the console or from a SUBMIT file), you should allow for the maximum number of buffers it might need. This means that for any open non-lined file you would allow two buffers, and three buffers for each lined file, whether or not it is a disk file. You would also allow a total of seven buffers to handle console input and console output, whether or not they are disk files, and two for system processing of system calls.

General Parameter Discussion

The parameters to system service routines presume certain uses. If you understand the intended usage, you will readily see the reasons for the parameters and how to specify them to achieve your purposes.

The easiest one to understand, used by every routine, is *status$p*: This address is filled with a non-zero error code if the service operation could not complete its task normally. (Appendix G gives these codes and their meanings.)

Arguments

To invoke the execution of a program, you can type its name at the console, e.g., PROGRM. PROGRM may have options that can be specified on that invocation line. If so, the remainder of that line (after the program name) is called a "command tail."

This command tail is accessible to PROGRM via ISIS-II system calls to READ or RESCAN, as described later in this chapter. These routines can handle, in similar fashion, any file that you create to be read as if it were the console. These are called "lined" or line-edited files and are discussed in the next section. The PROGRM using them can then alter its mode of operation depending on the options specified with each invocation, adding flexibility to the user-interface of your programs.

Connections

ISIS-II maintains a list of twelve devices or files that your program may use during its execution, i.e., a list of "connections." A connection is a word, named and declared by you, filled by the system service routine OPEN.

You then use this word to specify that file or device whenever you need to perform any operation on it, i.e., to read, write, seek, rescan, or close it.

The list is also called an Active File Table, and the entries on it Active File Table Numbers, or AFTNs. Only objects on this list can be used for input/output operations. Such operations are accomplished using the connection rather than the actual device or file name. During execution, your program may perform these functions on multiple files, but only six may be open at one time (not counting console input :CI: and console output :CO:). When I/O actions for a given file are complete or the file will not be needed for the next phase of program activity, it can be closed to make room for other files that may be needed sooner.

Function References

Several Monitor routines are used as functions; i.e., they are invoked as part of an expression or a parameter list rather than being called. The charts later in this chapter show the necessary usage by the placement of terms such as "*byte$p*" or "*byte$out*" or "*prom$addr*" to indicate names or addresses you must supply.

Input/Output Parameters

In order to perform a READ or a WRITE, several questions must be answered unambiguously:

1. How many bytes are to be transferred?
2. To (or from) what file?
3. From (or to) what memory locations?

In the descriptions that follow, (1) is usually supplied as the parameter *count*, and (3) as the parameter *buffer$p*, the address of the locations to be read from or written into.

Question (2) naturally requires an ISIS-II pathname, e.g., :F1:YOURDA.TA1 , which can be up to 14 characters long in the format shown. This format for a string differs from the one used under the Series III operating system in two respects: it does not begin with a count of the characters to follow, and it must end with a non-valid pathname character. ISIS-II interprets the first non-valid pathname character encountered as marking the end of the string. Thus the last character before the non-valid character is the last character in the string.

Rather than require you to give the name of each file every time you do any input/output, ISIS-II maintains the table of active files mentioned above. One call to OPEN establishes the full name as an entry in this table and returns to you a connection number for your use in all further references to this file while it is in use.

Terms

If you have used earlier ISIS systems, you will notice a change in the terms describing some parameters for system calls. The parameters have not changed. The new terms highlight similarities between ISIS-II and Series III in both concept and usage.

In several cases the change is simply appending the characters "*$p*" to each term which is actually used as an address rather than as the value stored at that address. In the PL/M calls to the routines, you simply use the dot operator to provide the address of the variable you declared for use in these routines.

In other cases, there is a new name. For example, *conn* is used instead of AFTN to represent a connection to a file (formerly called an active file table number, as above). The pointer to this connection, giving its address, is called *conn$p* where it was formerly named AFTNPTR.

The syntax charts introducing each functional grouping of commands show the placement of each parameter, and the examples given with each individual command illustrate actual declaration and usage.

Line-Edited Input Files

ISIS-II provides a special way of reading ASCII files, called line-editing. Line-editing was designed for (but is not restricted to) the case of a human user, prone to err, typing characters at a keyboard. The rubout key and control characters discussed below allow you to correct mistakes and then transmit a perfect line by typing a carriage return (to which a line feed is appended automatically).

You tell ISIS-II that a file is to be line-edited by supplying a parameter in the OPEN system call. This call must also specify an echo file you opened earlier, because every line-edited file must have an associated file to which ISIS-II sends an echo of the input. If no echo is desired, the byte bucket (:BB:) can be opened as the echo file.

Terminating a Line

While a line is being physically entered from an input device, it is accumulated by ISIS-II in a 122-character line-editing buffer. When an editing character (described below) is entered, ISIS-II changes the contents of the buffer. No data is transferred to the requesting program until the line is terminated in one of three ways:

- A line feed is entered (automatically appended to every carriage return)
- An escape is entered
- A non-editing character is entered as the 122nd character

Reading From the Line-Edit Buffer

When the line has been terminated, the next (or pending) READ system call transfers the specified number of bytes of data from the line-editing buffer to the requesting program's buffer. When the number of bytes entered from a lined READ is greater than the number requested, ISIS-II keeps track of what characters have been read from the line-editing buffer. The remaining bytes are returned in response to subsequent READs.

For example, if the line-editing buffer contains 100 characters and you issue a READ system call with a *count* of 50, the first 50 characters are transferred to the program's buffer. The next READ system call transfers characters starting at the 51st character. The term MARKER is used in later discussions of this chapter to represent the position of the next byte to be processed.

If the READ system call requests 100 bytes and the line-editing buffer contains 50, only 50 bytes are transferred.

A READ call returns bytes from only one logical line at a time. This means only up to 122 "uncancelled" characters. Thus READ's of line-edited files often transfer fewer bytes than requested by *count*.

A READ system call returns no characters from a logical line until the line has been input in its entirety. Thus, during physical input, the logical line is accumulated in an internal buffer; no information in the buffer is transferred to the reading program until the termination character (normally a LF) is seen. Therefore ISIS-II has the opportunity to modify buffer contents conditionally on values entering the buffer. This is the mechanism of line editing which permits the manipulations described below.

When all the characters in the line-editing buffer have been read, the buffer pointer is positioned after the last character; the buffer is not yet cleared. In fact, the RESCAN system call can be used to reposition the buffer pointer to the beginning of the line-editing buffer so subsequent READs can reread the contents.

When the buffer has been completely read, with the pointer after the last character, a new READ system call will transfer new input from the line-edited file into the line-editing buffer. When the line is terminated, the number of characters requested by READ are transferred to the program.

Editing Characters

The following characters are used to edit the input of a line-edited file. Control characters are entered by holding down the control key (CTRL) while the character is typed.

RUBOUT	Pressing RUBOUT deletes the preceding character from the buffer.
CONTROL P	A CONTROL P causes the next character typed to be entered literally in the line-editing buffer. Use control P when you want an editing character or terminating character to be entered in the buffer rather than to cause its usual editing or terminating function.
CONTROL R	A CONTROL R causes a carriage-return/line-feed to be sent to the console output device, followed by the current undeleted contents of the buffer. It has no other effects.

CONTROL X	CONTROL X causes the entire contents of the buffer to be deleted, including itself. It is echoed as a #, carriage return, line feed.
CONTROL Z	CONTROL Z is the only way to indicate end-of-file from a keyboard input device. It acts like control X except that it has no echo and it causes the READ system call to return immediately without transferring any characters, thus simulating an end-of-file. If more characters are entered after the control Z, they are entered in the line-editing buffer and can be read by a subsequent READ system call.

Reading a Command Line

Reading a command line from the console input device is a special case of reading a line-edited file.

When a command is entered at the console, it is collected by ISIS-II in the line-editing buffer for :CI: and is not available to the command interpreter (a nonresident ISIS-II routine) until it is terminated. The command interpreter reads only the command name and then calls the program with that name, leaving the line-editing buffer pointer positioned after the command name. The loaded program can issue a READ which transfers data starting with the first parameter; or the program can issue a RESCAN to position the pointer to the beginning of the buffer so it can also read the command name.

For example, suppose the following command has been entered:

```
—TYPE :F1:PROGA.SRC(CR—LF)
```

The line-editing buffer for :CI: contains 20 characters as follows (the CR means carriage return, LF means line feed):

T	Y	P	E		:	F	1	:	P	R	O	G	A	.	S	R	C	CR	LF
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20

When the TYPE program is loaded, the buffer pointer is at the fifth character (the space following TYPE). A READ call starts transferring characters at the fifth character. New input from :CI: to the line-edit buffer does not happen until the buffer pointer is moved to the end of the buffer (after the 20th character) and a READ call is issued.

Remember that when control is passed to the loaded program, the buffer pointer is positioned after the command name, not at the end of the buffer. This means that if no parameters are passed, the first READ from :CI: returns the carriage-return/line-feed left from the command line. For example, suppose the following command has been entered:

```
—PROGA.BIN(CR—LF)
```

and the line-editing buffer contains 11 characters as follows:

P	R	O	G	A	.	B	I	N	CR	LF
1	2	3	4	5	6	7	8	9	10	11

When PROGA.BIN is loaded, the pointer is at the carriage return. If subsequent input is expected from the console input device, an extra READ must first be issued to clear the buffer of the carriage return-line feed.

If the program does not read from :CI:, the remaining carriage-return/line-feed is cleared by ISIS-II from the buffer before a new command is read by the command interpreter (e.g., after PROGA.BIN exits).

Summary of System Calls

The ISIS-II and Monitor services that can be called by your program include the following:

- Input/output operations for the disk and the standard Intellec peripherals, except the Universal PROM Programmer (OPEN, CLOSE, READ, WRITE, SEEK, RESCAN, SPATH)
- Disk directory maintenance (ATTRIB, DELETE, RENAME)
- Console device assignment and error message output (CONSOL, WHOCON, ERROR)
- Program loading and execution and return to the supervisor (LOAD, EXIT)
- Monitor I/O routines for control of peripheral devices (CI, CO, RI, PO, LO, UI, UO)
- Monitor status routines for peripheral devices (CSTS, UPPS, IOCHK, IOSET, MEMCK)
- Monitor routine to extend I/O system to user written drivers (IODEF)

The interface to these services is a call to ISIS-II that specifies the services desired and the address of the parameter list the supervisor is to access. The specific calling sequences are described with the call descriptions. Note that an ISIS-II routine does not use your stack. Your stack depth is not affected by the call. A call to ISIS-II destroys the contents of the CPU registers.

The system calls are described in terms of their operation and the parameters your program must supply.

To clarify the effect of certain system calls on your files, two integer quantities, LENGTH and MARKER, are associated with each file in this description. LENGTH is the number of bytes in the file. MARKER is the number of bytes already read or written in the file (that is, it acts as a file pointer).

System Call Syntax and Usage

Many of the ISIS-II system calls have names and functions similar to those of the ISIS-II commands discussed in the *Intellec Series III Console Operating Instructions Manual*. This is true because use of ISIS-II by another program is essentially the same as your use of ISIS-II when seated at the console.

The ISIS-II system routines can be called from your PL/M or Assembler Language programs. If your program does make an ISIS-II system call, you must remember to link your object program with SYSTEM.LIB using the LINK program.

SYSTEM.LIB is a library file supplied with the ISIS-II system disk. It contains the procedures necessary to interface your programs containing ISIS-II system calls with the ISIS-II system.

PL/M Calls

Your PL/M program can interface to ISIS by performing calls to procedures in SYSTEM.LIB. Your program must include external procedure declarations so the proper procedures from SYSTEM.LIB will be included with your program by LINK. These external procedure declarations may be declared as type address, but may also be values as well as addresses of values.

Assembly Language Calls

The interface between the 8080/8085 Assembler Language program and ISIS is accomplished by calling a single ISIS entry point (labeled ISIS) and passing two parameters. The first parameter is a number that identifies the system call; the second is the address of a control block that contains the additional parameters required by the system call. The first parameter is passed in register C, and the address of the control block is passed in the register pair DE. The entry point must be defined in your program as an external:

```
EXTRN ISIS
```

The ISIS entry point is defined in a routine in SYSTEM.LIB that must be included in your program. Use LINK, specifying the name of your program followed by the name SYSTEM.LIB. See the *MCS-80/85 Utilities User's Guide* for more information on LINK.

System call identifying numbers can be defined in EQUATE statements before they are referenced in your program. This allows you to reference these routines symbolically. Only the specific system calls needed by your program need be defined. Table 3-2 lists the identifying numbers for the system calls.

Table 3-2. System Call Identifiers

SYSTEM CALL	IDENTIFIER
OPEN	0
CLOSE	1
DELETE	2
READ	3
WRITE	4
SEEK	5
LOAD	6
RENAME	7
CONSOL	8
EXIT	9
ATTRIB	10
RESCAN	11
ERROR	12
WHOCON	13
SPATH	14

File Input/Output Calls

Seven system calls are available to your program for controlling file input/output. These subroutines let you open files for read or write operations, move the pointer in an open file, and close the files when you're finished.

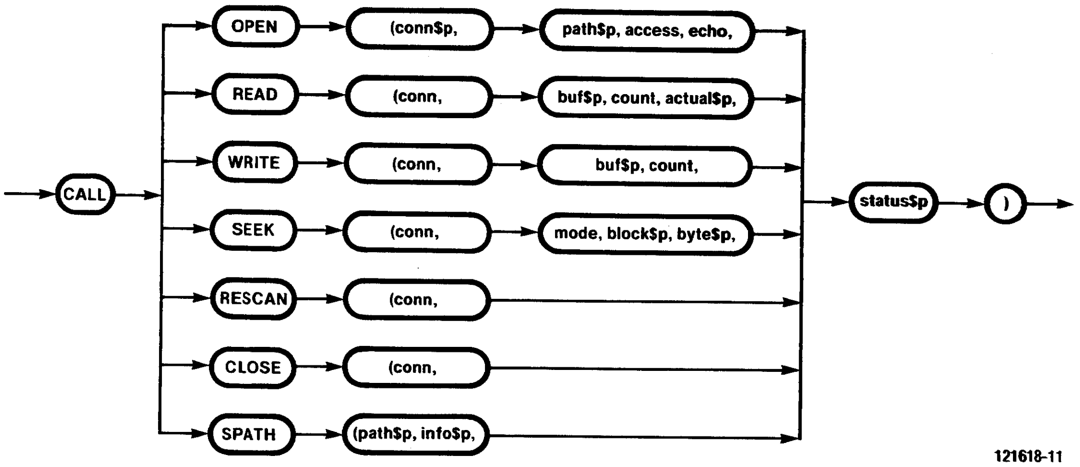
These services of the supervisor enable you to transfer variable-length blocks of data between standard peripheral devices and a memory buffer area in your program. In addition to the data transfer buffer in your program area, the disk supervisor requires two 128-byte buffers for each open disk file. This buffer is located in the buffer area described in the memory layout in Memory Organization and Allocation. These calls establish and maintain the MARKER and LENGTH quantities associated with each file in use.

System Call Cautions

ISIS-II references files by a connection number (AFTN, or active file table number). Be careful not to confuse this number with the PL/M construction .AFTN. The period (.) specifies the address of the memory location where the AFTN is stored.

To reduce this potential confusion, both the syntax charts and the sample PL/M declarations in the examples refer to the connection number as *conn*, and to the address as *conn$p*. Then in the CALLs of the PL/M examples, *periods* precede the names when *addresses* are to be supplied.

Similarly, the charts and declarations show *path$p* to represent the *address* of a memory location containing the string naming a device or file. The CALL then shows a period before the variable you declared to hold that string.



ISIS-II File Management Routines

Good programming practice suggests *status* checks when making system calls. This is similar to the use of *excep$p* under Series III in Chapter 2. Refer to the TYPE program in Appendix E for an example of how status checks are used.

OPEN - Initialize File for Input/Output Operations

The OPEN call initializes ISIS tables and allocates buffers that are required for input/output processing of the specified file. If the specified file is a punch device (:HP: or :TP:), 12 inches of leader (ASCII nulls) are punched.

You must pass five parameters in the OPEN call:

1. ***conn$p***, an address of a two byte field in which ISIS will store the connection number (AFTN) of the file that is opened. Your program will use this value for other calls relative to this file. :CI: and :CO: are always open and have the AFTNs 1 and 0 permanently assigned. Excluding :CI: and :CO:, you can only have six files open at any one time. Be careful not to confuse AFTN with the PL/M construction .AFTN. The period prefacing .AFTN signifies the location in memory of AFTN.
2. ***path$p***, the address of the ASCII string containing the name of the file to be opened. The ASCII string can contain leading space characters but no embedded space characters. It must be terminated by a character other than a letter, digit, colon (:), or period (.). A space can be used.
3. ***access***, a value indicating the access mode for which the file is being opened.

   A value of 1 specifies that the file is open only for input to the system: READ. MARKER is set to 0 and LENGTH is unchanged. If the file is nonexistent, a nonfatal error occurs.

   A value of 2 specifies that the file is open only for output from the system: WRITE. MARKER and LENGTH are set to 0. If the file is nonexistent, a disk file is created with the filename specified at location *path$p*, and all attributes of the new file are reset (0). If it already exists, information in the file will be over-written. Specifying a file whose format or write-protect attributes are set causes a nonfatal error.

   A value of 3 specifies that the file is open for update: READ and WRITE. MARKER is set to 0. LENGTH is unchanged for existing files and set to 0 for new files. If the file is nonexistent, a new file is created with the filename specified at location *path$p*, and all attributes are reset (0).

   Opening a file for an access mode that is not physically possible causes a non-fatal error, e.g., opening :HP: (high-speed paper tape punch) for input or :LP: (line printer) for update.
4. ***echo***, the AFTN of the echo file if the file is to be opened for line editing. The echo file must be previously opened for output (access=2). The AFTN of the echo file is passed in the least significant byte of the field. If this field contains 0, no line editing is done. To specify an AFTN of 0 for :CO:, a nonzero value must be in the most significant byte and zero in the least significant byte. For example, FF00H specifies the AFTN for the :CO: device.
5. ***status$p***, the address of a memory location for the return of nonfatal error numbers.

Nonfatal error codes returned: 3,4,5,9,12,13,14,22,23,25,28

Fatal error codes returned: 1,7,24,30,33

PL/M OPEN Call Example

```
OPEN:
  PROCEDURE (conn$p, path$p, access, echo, status$p) EXTERNAL;
    DECLARE (conn$p, path$p, access, echo, status$p) ADDRESS;
  END OPEN;
.
.
.
```

```
    DECLARE AFT$IN ADDRESS;
    DECLARE FILENAME(15) BYTE DATA (':F1:MYPROG.SRC ');
    DECLARE STATUS ADDRESS;
  .
  .
  .
  CALL OPEN(.AFT$IN,.FILENAME,1,0,.STATUS);
  IF STATUS <> 0 THEN ...
```

Assembly Language OPEN Call Example

```
            EXTRN       ISIS          ;LINK TO ISIS ENTRY POINT
   OPEN     EQU         0             ;SYSTEM CALL IDENTIFIER

            MVI         C,OPEN        ;LOAD IDENTIFIER
            LXI         D,OBLK        ;ADDRESS OF PARAMETERS
                                      ;BLOCK
            CALL        ISIS
            LDA         OSTAT         ;TEST ERROR STATUS
            ORA         A
            JNZ         EXCEPT        ;BRANCH TO EXCEPTION
                                      ;ROUTINE
.
.
.
OBLK:                                 ;PARAMETER BLOCK FOR OPEN
            DW          OAFT          ;POINTER TO AFTN
            DW          OFILE         ;POINTER TO FILENAME
ACCESS:     DW          1             ;ACCESS, READ = 1, WRITE = 2,
                                      ;UPDATE = 3
ECHO:       DW          0             ;IF ECHO <> 0,
                                      ;ECHO = AFTN OF
                                      ;ECHO OUTPUT FILE
            DW          OSTAT         ;POINTER TO STATUS
;
OAFT:       DS          2             ;AFTN (RETURNED)
OSTAT:      DS          2             ;STATUS (RETURNED)
OFILE:      DB          ':F0:FILE.EXT ' ;FILE TO BE OPENED
```

READ - Transfer Data from File to Memory

The READ call transfers data from an open file to a memory location specified by the calling program. See also "Line-Edited Input Files" for additional information about such files.

You must pass five parameters in the READ call:

1. *conn*, the connection number (AFTN) of a file that is open for input or update. This connection was returned by a preceding OPEN call or is 1 for :CI:.
2. *buf$p*, the address of a buffer you have declared to receive the data read from the open file. The buffer must be at least as long as the count described below. If the buffer is too short, memory locations following the buffer will be overwritten.
3. *count*, the number of bytes to be transferred from the file to the buffer.
4. *actual$p*, the address of a memory location you have declared to receive from ISIS the actual number of bytes successfully transferred. The same number is added to MARKER. The actual number of bytes transferred is never more than the number specified in the *count* parameter above.

For line-edited files, the actual number of bytes is never more than the number of bytes in the line-edit buffer. When a file is not line edited, the number of bytes is equal either to count or to LENGTH minus MARKER, whichever is fewer.

actual = 0 is the only reliable way of detecting end-of-file (for both lined and non-lined files).

5. *status$p*, the address of a memory location you declared for ISIS to store nonfatal error numbers.

Nonfatal error codes returned: 2, 8

Fatal error codes returned: 24, 30, 33

PL/M READ Call Example

```
READ:
   PROCEDURE(conn, buf$p, count, actual$p, status$p)EXTERNAL;
      DECLARE (conn, buf$p, count, actual$p, status$p) ADDRESS;
   END READ;
.
.
.
DECLARE AFT$IN ADDRESS;
DECLARE BUFFER(128) BYTE;
DECLARE ACTUAL ADDRESS;
DECLARE STATUS ADDRESS;
.
.
.
CALL READ (AFT$IN,.BUFFER,128,.ACTUAL,.STATUS);
IF STATUS < > 0 THEN ...
```

Assembly Language READ Call Example

```
               EXTRN      ISIS         ;LINK TO ISIS ENTRY POINT
READ           EQU        3            ;SYSTEM CALL IDENTIFIER

               MVI        C,READ       ;LOAD IDENTIFIER
               LXI        D,RBLK       ;ADDRESS OF PARAMETER
                                       ;BLOCK
               CALL       ISIS
               LDA        RSTAT        ;TEST ERROR STATUS
               ORA        A
               JNZ        EXCEPT       ;BRANCH TO EXCEPTION
                                       ;ROUTINE
.
.
.
RBLK:                                  ;PARAMETER BLOCK FOR READ
RAFT:          DS         2            ;FILE AFTN
               DW         IBUF         ;ADDRESS OF INPUT BUFFER
RCNT:          DW         128          ;LENGTH OF READ REQUESTED
               DW         ACTUAL       ;POINTER TO ACTUAL
               DW         RSTAT        ;POINTER TO STATUS
;
ACTUAL:        DS         2            ;COUNT OF BYTES READ
                                       ;(RETURNED)
```

```
RSTAT:      DS      2       ;STATUS (RETURNED)
IBUF:       DS      128     ;INPUT BUFFER
;
```

WRITE - Transfer Data from Memory to File

The WRITE call transfers data from a specified location in memory called a buffer to an open file. You must pass four parameters in the WRITE call:

1. *conn*, the connection number (AFTN) of a file open for output or update. This connection was returned by a preceding OPEN call or is 0 for :CO:.
2. *buf$p*, the address of the memory buffer from which data is to be transferred. This address may be in the form

   .(string literal)

   The period causes the use of the address where the string literal (given inside the parentheses) is stored. See example below.
3. *count*, the number of bytes to be transferred from the buffer to the output file. The value of the count is added to MARKER. If this results in MARKER being greater than LENGTH, then LENGTH is set equal to MARKER. The number of bytes actually transferred by WRITE is exactly equal to count. Thus if the buffer length is less than count, memory locations following the buffer are written to the file.
4. *status$p*, the address of the memory location for the return of nonfatal error numbers.

Nonfatal error codes returned: 2,6

Fatal error codes returned: 7,24,30,33

PL/M WRITE Call Example

```
WRITE:
    PROCEDURE (conn, buf$p, count, status$p) EXTERNAL;
        DECLARE (conn, buf$p, count, status$p) ADDRESS;
    END WRITE;
.
.
.
DECLARE AFT$OUT ADDRESS;
DECLARE BUFFER(128) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL WRITE (0,.('this is an example of string literal', 0DH,0AH),38, .STATUS);
CALL WRITE (AFT$OUT,.BUFFER,128,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language WRITE Call Example

```
         EXTRN     ISIS        ;LINK TO ISIS ENTRY POINT
WRITE    EQU       4           ;SYSTEM CALL IDENTIFIER

         MVI       C,WRITE     ;LOAD IDENTIFIER
         LXI       D,WBLK      ;ADDRESS OF PARAMETER
                               ;BLOCK
         CALL      ISIS
         LDA       WSTAT       ;TEST ERROR STATUS
         ORA       A
         JNZ       EXCEPT      ;BRANCH TO EXCEPTION
                               ;ROUTINE
.
.
.
WBLK:                          ;PARAMETER BLOCK FOR
                               ;WRITE
WAFT:    DS        2           ;FILE AFTN
         DW        OBUF        ;ADDRESS OF OUTPUT BUFFER
WCNT:    DW        128
         DW        WSTAT       ;POINTER TO STATUS
;
WSTAT:   DS        2           ;STATUS (RETURNED)
OBUF:    DS        128         ;OUTPUT BUFFER
;
```

SEEK - Position Disk File Marker

The SEEK call allows your program to determine or to change the value of the MARKER associated with a disk file open for read or update. SEEK cannot be used with a file open for write only. The MARKER value can be changed in four ways: moved forward, moved backward, moved to a specific location, or moved to the end of the file. A nonfatal error occurs if SEEK is issued for a file opened for output.

You must pass five parameters in the SEEK call as discussed below. The mode and file pointer parameters differ from those of Series III.

1. *conn*, the connection number (AFTN) of a file on a random access device opened for update or input. This connection was returned by a preceding OPEN call.

2. *mode*, a value from 0 through 4 that indicates what action should be performed on the MARKER. The block and byte parameters (see below) are used either to represent the current MARKER position or to calculate the desired offset, depending on "mode." A detailed discussion follows this parameter list.

3. *block$p*, the address of a memory location containing a 2-byte value used for the block number. A block is equivalent to 128 bytes, the same as a sector on the disk.

4. *byte$p*, the address of a memory location containing a 2-byte value used for the byte number. The byte number may be greater than 128.

5 *status$p*, the address of a memory location you declared to receive nonfatal error numbers from ISIS.

Detailed Discussion of Mode Values

Return Marker Location: Mode=0

Under this mode, the system returns a pair of block and byte values (at *block$p* and *byte$p*) that signify the current position of the marker. For example, if the marker is just beyond the first block of the file, the system might return the numbers 1 and 0 in the addresses assigned to block and byte, respectively. It might also return the numbers 0 and 128, which point to the same byte in the file. The value of MARKER is given by the following equation:

```
MARKER = 128 * (block number) + byte number
```

Move Marker Backward: Mode=1

If the mode value is 1, the marker is moved backward toward the beginning of the file. The block and byte parameters determine how many bytes the marker is moved back; for example, if block is equal to 0 and byte is equal to 382, the marker is moved backward 382 bytes. To define an offset of N, use block and byte values such that

```
N = 128 * (block number) + byte number
```

If N is greater than MARKER, the prescribed action would place the marker before the beginning of the file. MARKER is then set to 0 (beginning of file) and a nonfatal error occurs.

Move Marker to Specific Location: Mode=2

In this mode, the marker is moved to a specific position in the file. The block and byte parameters define the position; for example, if block is equal to 27 and byte is equal to 63, the marker will be moved to block 27, byte 63. Similarly, if both block and byte are equal to 0, the marker is moved to the beginning of the file. If the file is open for update and the prescribed action would place the marker beyond the end of the file, ASCII nulls (0000H) are added to the file to extend the file to the marker. (Thus, LENGTH becomes equal to MARKER.)

Move Marker Forward: Mode=3

In this mode, the marker is moved forward toward the end of the file. The block and byte parameters define the offset N according to the following equation:

```
N + 128 * (block number) + byte number
```

If the file is open for update and the specified action would place the marker beyond the end of the file, ASCII nulls (0000H) are added to the file to extend the file to the marker. (Thus, LENGTH becomes equal to MARKER.)

If the extension of a file by the SEEK operation causes an overflow on the disk, a fatal error is reported, either during the execution of the SEEK call or when a program tries to write into the extended area of the file. This error can become evident at any time in the life of the file.

If an attempt is made to extend a file that is open only for input, the marker is set to the former end-of-file and a nonfatal error occurs.

Move Marker to End of File

If the mode value is 4, the marker is moved to the end of the file. Block and byte parameters are ignored.

NOTE

For a file opened for update, MARKER manipulations can allocate more memory than LENGTH requires, i.e., than are subsequently written with data. A DIR will show the allocated file locations as in use. You can actually allocate more storage than exists as data on disk. Data can still be written to these locations.

Nonfatal error codes returned: 2,19,20,27,31,35

Fatal error codes returned: 7,24,30,33

PL/M Seek Call Example

```
SEEK:
  PROCEDURE (conn, mode, block$p, byte$p, status$p) EXTERNAL;
      DECLARE (conn, mode, block$p, byte$p, status$p) ADDRESS;
  END SEEK;
.
.
.
DECLARE AFT$IN ADDRESS;
DECLARE BLOCKNO ADDRESS;
DECLARE BYTENO ADDRESS;
DECLARE STATUS ADDRESS;
.
.
.
CALL SEEK (AFT$IN,0,.BLOCKNO,.BYTENO,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language SEEK Call Example

```
                EXTRN      ISIS          ;LINK TO ISIS ENTRY POINT
SEEK            EQU        5             ;SYSTEM CALL IDENTIFIER

                MVI        C,SEEK        ;LOAD IDENTIFIER
                LXI        D,SBLK        ;ADDRESS OF PARAMETER
                                         ;BLOCK
                CALL       ISIS
                LDA        SSTAT         ;TEST ERROR STATUS
                ORA        A
                JNZ        EXCEPT        ;BRANCH TO EXCEPTION
                                         ;ROUTINE
.
.
.
SBLK:                                    ;PARAMETER BLOCK FOR SEEK
SAFT:           DS         2             ;AFTN FROM OPEN
MODE:           DS         2             ;TYPE OF SEEK
                DW         BLKS          ;POINTER TO BLKS
                DW          NBYTE       ;POINTER TO NBYTE
                DW          SSTAT       ;POINTER TO STATUS
;
BLKS:           DS          2           ;NUMBER OF SECTORS TO SKIP
NBYTE:          DS          2           ;NUMBER OF BYTES TO SKIP
SSTAT:          DS          2           ;STATUS (RETURNED)
;
```

RESCAN - Position MARKER to Beginning of Line

The RESCAN call is used on line-edited files only. It allows your program to move the MARKER to the beginning of a logical line that has already been read. Thus the next READ call starts at the beginning of the last logical line read. This line is not re-echoed (output to the echo file) because it has already been input from the keyboard and echoed. Thus the subsequent READ does not input from a file but only reads from a buffer in memory.

You must pass two parameters in the RESCAN call:

1. *conn*, the connection number (AFTN) of a file opened for line-edited input (with echo file AFTN specified) by a preceding OPEN call
2. *status$p*, the address of a memory location for the return of nonfatal error numbers

Nonfatal error codes returned: 2,21

Fatal error codes returned: 33

PL/M RESCAN Call Example

```
RESCAN:
  PROCEDURE (conn, status$p) EXTERNAL;
     DECLARE (conn, status$p) ADDRESS;
  END RESCAN;
.
.
.
DECLARE AFT$IN ADDRESS;
DECLARE STATUS ADDRESS;
.
.
.
CALL RESCAN (AFT$IN,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language RESCAN Call Example

```
                EXTRN       ISIS        ;LINK TO ISIS ENTRY POINT
RESCAN          EQU         11          ;SYSTEM CALL IDENTIFIER

                MVI         C,RESCAN    ;LOAD IDENTIFIER
                LXI         D,IBLK      ;ADDRESS OF PARAMETER
                                        ;BLOCK
                CALL        ISIS
                LDA         ISTAT       ;TEST ERROR STATUS
                ORA         A
                JNZ         EXCEPT      ;BRANCH TO EXCEPTION
                                        ;ROUTINE
    .
    .
    .
IBLK:                                   ;PARAMETER BLOCK FOR
                                        ;RESCAN
IAFT:           DS          2           ;AFTN FROM OPEN
                DW          ISTAT       ;POINTER TO STATUS
;
ISTAT:          DS          2           ;STATUS (RETURNED)
;
```

CLOSE - Terminate Input/Output Operations on a File

The CLOSE call removes a file from the system input/output tables and releases the buffers allocated for it by OPEN. You should close each file in use when input/output processing is complete. If the file closed is a paper tape punch device (:HP: or :TP:), 12 inches of trailer (ASCII null characters) are punched.

You must pass two parameters in the CLOSE call:

1. *conn*, the connection number (AFTN) of the file to be closed. This connection was returned by a preceding OPEN call.
2. *status$p*, the address of a memory location for the return of nonfatal error numbers.

Nonfatal error codes returned: 2

Fatal error codes returned: 33

PL/M CLOSE Call Example

```
CLOSE:
  PROCEDURE (conn, status$p) EXTERNAL;
      DECLARE (conn, status$p) ADDRESS;
  END CLOSE;
.
.
.
DECLARE AFT$IN ADDRESS;
DECLARE STATUS ADDRESS;
.
.
.
CALL CLOSE (AFT$IN,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language CLOSE Call Example

```
                EXTRN       ISIS        ;LINK TO ISIS ENTRY POINT
CLOSE           EQU         1           ;SYSTEM CALL IDENTIFIER

                MVI         C,CLOSE     ;LOAD IDENTIFIER
                LXI         D,CBLK      ;ADDRESS OF PARAMETER
                                        ;BLOCK
```

```
               CALL      ISIS
               LDA       CSTAT          ;TEST ERROR STATUS
               ORA       A
               JNZ       EXCEPT         ;BRANCH TO EXCEPTION
                                        ;ROUTINE
;
CBLK:                                   ;PARAMETER BLOCK FOR
                                        ;CLOSE
CAFT:          DS        2              ;FILE AFTN
               DW        CSTAT          ;POINTER TO STATUS
;
CSTAT:         DS        2              ;STATUS (RETURNED)
;
```

SPATH - Obtain File Information

The SPATH call allows your program to obtain information relating to a specified file. The information returned by this call includes the device number, file name and extension, device type, and if a disk file, the drive type.

You pass three parameters in the SPATH call:

1. *path$p*, the address of an ASCII string containing the name of the file for which information is requested. The string can contain leading spaces but no embedded spaces. It must be terminated by a character other than a letter, digit, colon (:), or period (.). A space can be used.
2. *info$p*, the address of a 12-byte memory location in which the system will return the information. After the call is completed, the buffer will contain the following information:

   Byte 0 - Device number

   Bytes 1 through 6 - File name

   Bytes 7 through 9 - File name extension

   Byte 10 - Device type

   Byte 11 - Drive type

3. *status$p*, the address of a memory location for the return of a nonfatal error number.

Nonfatal error codes returned: 4,5,23,28

Fatal error codes returned: 33

The possible values for the contents of info$p device number are:

0 - disk drive 0
1 - disk drive 1
2 - disk drive 2
3 - disk drive 3
4 - disk drive 4
5 - disk drive 5
6 - teletype input
7 - teletype output
8 - CRT input
9 - CRT output
10 - user console input
11 - user console output

12 - teletype paper tape reader
13 - high speed paper tape reader
14 - user reader 1
15 - user reader 2
16 - teletype paper tape punch (teletype)
17 - high speed paper tape punch
18 - user punch 1
19 - user punch 2
20 - line printer
21 - user list 1
22 - byte bucket (a pseudo input/output device)
23 - console input
24 - console output
25 - disk drive 6
26 - disk drive 7
27 - disk drive 8
28 - disk drive 9

The file name and extension are the ISIS file name, e.g., SAMPLE.SRC without the period.

The device type specifies the type of peripheral with which the file is associated. The possible values for this field are

0 - sequential input device
1 - sequential output device
2 - sequential input/output device
3 - random access input/output device

The drive type field specifies the type of drive controller if the device type field is 3. If the device type is anything except 3, the drive type is undefined. The possible values for a device type of 3 are

0 - controller not present
1 - two-board double density
2 - two-board single density
3 - integrated single density
4 - two-board hard disk

PL/M SPATH Call Example

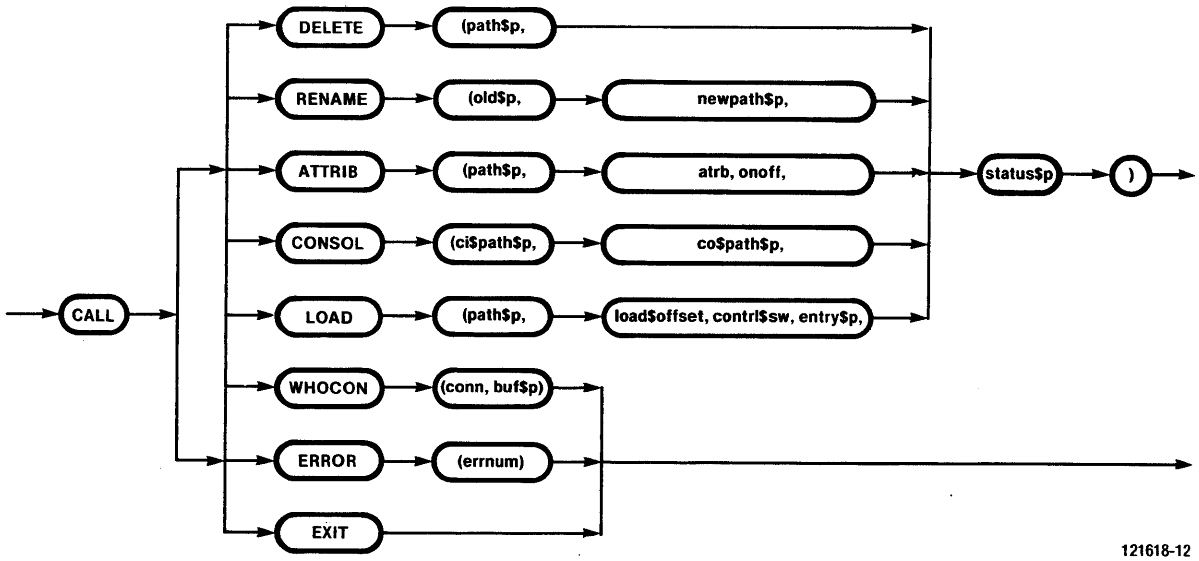
```
SPATH:
  PROCEDURE (path$p, info$p, status$p) EXTERNAL;
      DECLARE (path$p, info$p, status$p) ADDRESS;
  END SPATH;
.
.
.
DECLARE FILENAM(15) BYTE;
DECLARE FILINF STRUCTURE (DEVICE$NO BYTE,
                          FILENAME (6) BYTE,
                          FILE$EXT (3) BYTE,
                          DEVICE$TYPE BYTE,
                          DRIVE$TYPE BYTE);
DECLARE STATUS ADDRESS;
.
.
.
CALL SPATH (.FILENAM,.FILINF,.STATUS);
IF STATUS < > 0 THEN ...
.
.
```

Assembly Language SPATH Call Example

```
                EXTRN     ISIS          ;LINK TO ISIS ENTRY POINT
SPATH           EQU       14            ;SYSTEM CALL IDENTIFIER
;
                MVI       C,SPATH       ;LOAD IDENTIFIER
                LXI       D,SBLK        ;LOAD PARAM ADDR
                CALL      ISIS
                LDA       SSTAT         ;TEST ERROR STATUS
                ORA       A
                JNZ       EXCEPT        ;BRANCH TO EXCEPTION
                                        ;ROUTINE
.
.
.
SBLK:                                   ;PARAMETER BLOCK FOR
                                        ;SPATH
                DW        FILEN         ;POINTER TO FILE NAME
                DW        BUFIN         ;POINTER TO BUFFER
                DW        SSTAT         ;POINTER TO STATUS
;
FILEN:          DS        15            ;FILE NAME FIELD
BUFIN:          DS        12            ;BUFFER FOR DATA
SSTAT:          DS        2             ;STATUS (RETURNED)
;
```

Disk Directory, Console, and Program Execution Control

The chart below shows the form of calls to the routines discussed in the following pages. Each parameter is explained where used.



Commands for Control of Disk Directories, Console, and Program Execution

Disk Directory Maintenance

Three system calls are available to your program for changing information in the disk directory. These calls allow you to delete a disk file, rename a disk file, and change the attributes of a disk file.

DELETE - Delete a File from the Disk Directory

The DELETE call removes a specified file from its disk. The file must not be open. The disk space allocated to the file is released. The space can then be reused for another file.

You must pass two parameters in the DELETE call:

1. *path$p*, the address of an ASCII string that specifies the name of the file to be deleted. The file to be deleted must not be open. The string can contain leading space characters but no embedded spaces. It must be terminated by a character other than a letter, digit, colon (:), or period (.). You can use a space.
2. *status$p*, the address of a memory location for the return of a nonfatal error number.

Nonfatal error codes returned: 4,5,13,14,17,23,28,32

Fatal error codes returned: 1,24,30,33

PL/M DELETE Call Example

```
DELETE:
  PROCEDURE (path$p, status$p) EXTERNAL;
      DECLARE (path$p, status$p) ADDRESS;
  END DELETE;
.
.
.
DECLARE FILENAM(20) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL DELETE (.FILENAM,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language DELETE Call Example

```
            EXTRN      ISIS         ;LINK TO ISIS ENTRY POINT
DELETE      EQU        2            ;SYSTEM CALL IDENTIFIER
;
            MVI        C,DELETE     ;LOAD IDENTIFIER
            LXI        D,DBLK       ;ADDRESS OF PARAMETER
                                    ;BLOCK
            CALL       ISIS
            LDA        DSTAT        ;TEST ERROR STATUS
            ORA        A
            JNZ        EXCEPT       ;BRANCH TO EXCEPTION
                                    ;ROUTINE
.
.
.
DBLK:                               ;PARAMETER BLOCK FOR
                                    ;DELETE
            DW         DFILE        ;POINTER TO FILENAME
            DW         DSTAT        ;POINTER TO STATUS
;
```

```
DSTAT:      DS      2           ;STATUS (RETURNED)
DFILE:      DB      'FILE.EXT'  ;NAME OF FILE TO BE DELETED
;
```

RENAME - Change Disk Filename

The RENAME call allows your program to change the name of a disk file.

You must pass three parameters in the RENAME call:

1. *old$p*, the address of an ASCII string that contains the old file name. The string can contain leading spaces but no embedded spaces. It must be terminated by a character other than a letter, digit, colon (:), or period (.). You can use a space.
2. *newpath$p*, the address of an ASCII string that contains the new file name. The string must obey the rules above. The device portion of the name must be the same as that in old name.
3. *status$p*, the address of a memory location for the return of a nonfatal error number.

Nonfatal error codes returned: 4,5,10,11,13,17,23,28

Fatal error codes returned: 1,24,30,33

PL/M RENAME Example

```
RENAME:
  PROCEDURE (old$p, newpath$p, status$pEXTERNAL;
     DECLARE (old$p, newpath$p, status$p) ADDRESS;
  END RENAME;
.
.
.
DECLARE OFILE(20) BYTE;
DECLARE NFILE(20) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL RENAME (.OFILE,.NFILE,.STATUS);
IF STATUS < > 0 THEN ...
.
.
.
```

Assembly Language RENAME Call Example

```
            EXTRN   ISIS        ;LINK TO ISIS ENTRY POINT
RENAME      EQU     7           ;SYSTEM CALL IDENTIFIER
;
            MVI     C,RENAME    ;LOAD IDENTIFIER
            LXI     D,NBLK      ;ADDRESS OF PARAMETER
                                ;BLOCK
            CALL    ISIS
            LDA     NSTAT       ;TEST ERROR STATUS
            ORA     A
            JNZ     EXCEPT      ;BRANCH TO EXCEPTION
                                ;ROUTINE
.
.
.
```

```
NBLK:                                   ;PARAMETER BLOCK FOR
                                        ;RENAME
                DW          FILE2       ;POINTER TO OLD FILENAME
                DW          FILE1       ;POINTER TO NEW FILENAME
                DW          NSTAT       ;POINTER TO STATUS
;
NSTAT:          DS          2           ;STATUS (RETURNED)
FILE1:          DB          'FILE.NEW ' ;NEW NAME OF FILE
FILE2:          DB          'FILE.OLD ' ;OLD NAME OF FILE
;
```

ATTRIB - Change the Attributes of a Disk File

The ATTRIB call allows your program to change an attribute of a disk file.

You must pass four parameters in the ATTRIB call:

1. *path$p*, the address of an ASCII string containing the name of the file whose attribute is to be changed. The string can contain leading space characters but no embedded spaces. It must be terminated by a character other than a letter, digit, colon (:), or period (.). A space can be used.
2. *atrb*, a number indicating which attribute is to be changed:

   0 - invisible attribute

   1 - system attribute

   2 - write protect attribute

   3 - format attribute
3. *onoff*, a value indicating whether the attribute is to be set (turned on) or reset (turned off). The value is stored in the low order bit of the low order byte. A value of 1 specifies that the attribute be set and a value of 0 specifies that it be reset.
4. *status$p*, the address of a memory location for the return of a nonfatal error number.

Nonfatal error codes returned: 4,5,13,23,26,28

Fatal error codes returned: 1,24,30,33

PL/M ATTRIB Call Example

```
ATTRIB:
  PROCEDURE (path$p, atrb, onoff, status$p) EXTERNAL;
     DECLARE (path$p, atrb, onoff, status$p) ADDRESS;
  END ATTRIB;
.
.
.
DECLARE FILE(15) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL ATTRIB (.FILE,2,0,.STATUS);
IF STATUS < > 0 THEN ...
.
.
.
```

Assembly Language ATTRIB Call Example

```
                EXTRN       ISIS          ;LINK TO ISIS ENTRY POINT
ATTRIB          EQU         10            ;SYSTEM CALL IDENTIFIER
;
                MVI         C,ATTRIB      ;LOAD IDENTIFIER
                LXI         D,ABLK        ;LOAD PARAM ADDR
                CALL        ISIS
                LDA         ASTAT         ;TEST ERROR STATUS
                ORA         A
                JNZ         EXCEPT        ;BRANCH TO EXCEPTION
                                          ;ROUTINE
;
ABLK:                                     ;PARAMETER BLOCK FOR
                                          ;ATTRIB
                DW          FILEN         ;POINTER TO FILE NAME
                DW          2             ;ATTRIBUTE IDENTIFIER
                DW          0             ;SET/RESET SWITCH
                DW          ASTAT         ;POINTER TO STATUS
;
FILEN:          DS          15            ;FILE NAME FIELD
ASTAT:          DS          2             ;STATUS (RETURNED)
;
```

Console Reassignment and Error Message Output

Three system calls are available to your program for system console control. They allow you to change the device used as the system console, to determine which device is the current console, and, if needed, to send an error message to the console.

CONSOL - Change Console Device

The CONSOL call allows your program to change the console input and output device names (:CI: and :CO:) to refer to devices other than the initial system console.

You must pass three parameters in a CONSOL call:

1. *ci$path$p*, the address of an ASCII string containing the name of the file to be used for system console input. The string can have leading spaces but no embedded spaces. It must end with a character other than a letter, digit, colon (:), or period (.). You can use a space. Before opening the new file, the file is closed unless it happens to be :CI: which is always open. If the specified file cannot be opened, a fatal error occurs.
2. *co$path$p*, the address of an ASCII string containing the name of the file to be used for system console output. The name must follow the rules above. Before opening the new file, the current output file is closed unless it happens to be :CO: which is never closed. If the specified file cannot be opened, a fatal error occurs.
3. *status$p*, the address of a memory location for the return of a nonfatal error number.

No nonfatal error codes are returned; all errors are fatal: 1,4,5,12, 13,14,22,23,24,28,30,33

PL/M CONSOL Call Example

```
CONSOL:
  PROCEDURE (ci$path$p, co$path$p, status$p) EXTERNAL;
      DECLARE (ci$path$p, co$path$p, status$p) ADDRESS;
  END CONSOLE;
.
.
.
DECLARE INFILE(6) BYTE;
DECLARE OUTFILE(6) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL CONSOL (.INFILE,.OUTFILE,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language CONSOL Call Example

```
            EXTRN      ISIS          ;LINK TO ISIS ENTRY POINT
CONSOL      EQU        8             ;SYSTEM CALL IDENTIFIER
;
            MVI        C,CONSOL      ;LOAD IDENTIFIER
            LXI        D,CBLK        ;LOAD PARAM ADDR
            CALL       ISIS
            LDA        CSTAT         ;TEST ERROR STATUS
            ORA        A
            JNZ        EXCEPT        ;BRANCH TO EXCEPTION
                                     ;ROUTINE
;
CBLK:                                ;PARAMETER BLOCK FOR
                                     ;CONSOL
            DW         INFILE        ;POINTER TO FILE NAME
            DW         OTFILE        ;POINTER TO FILE NAME
            DW         CSTAT         ;POINTER TO STATUS
;
INFILE:     DS         15            ;INPUT FILE NAME
OTFILE:     DS         15            ;OUTPUT FILE NAME
CSTAT:      DS         2             ;STATUS (RETURNED)
;
```

WHOCON - Determine File Assigned as System Console

The WHOCON call allows your program to determine what file is assigned as the current system input console or output console.

You must pass three parameters in the WHOCON call:

1. *conn*, a value that indicates whether the input or output file (:CI: or :CO:) name is to be returned. A value of 0 specifies output and a value of 1 specifies input.
2. *buf$p*, the address of a 15-byte buffer reserved by your program for the return of the name of the file assigned to :CI: or :CO:. The name is returned as an ASCII string terminated by a space.
3. (Assembly language only) The address of a memory location for return of an error number.

No nonfatal error codes are returned. Fatal 33 may occur.

PL/M WHOCON Call Example

```
WHOCON:
  PROCEDURE (conn, buf$p) EXTERNAL;
      DECLARE (conn, buf$p) ADDRESS;
  END WHOCON;
.
.
.
DECLARE BUFF$IN(15) BYTE;
.
.
.
CALL WHOCON (1,.BUFF$IN);
.
.
.
```

Assembly Language WHOCON Call Example

```
            EXTRN     ISIS
WHOCON      EQU       13            ;CALL IDENTIFIER
;
            MVI       C,WHOCON      ;LOAD IDENTIFIER
            LXI       D,WBLK        ;LOAD PARAM ADDR
            CALL      ISIS
;
WBLK:
AFTN:       DS        2             ;AFTN FOR IN OR OUT
            DW        BUFIN         ;POINTER TO BUFFER
            DW        STATUS        ;POINTER TO STATUS RETURN
;
BUFIN:      DS        15            ;BUFFER FOR RETURN
                                    ;FILE NAME
STATUS:     DS        2             ;STATUS RETURN
;
```

ERROR - Output Error Message on System Console

The ERROR call enables your program to send an error message to the initial system console.

You must pass two parameters in the ERROR call:

1. *errnum*, the error number to output to the console. The error number must be in the low order eight bits of the parameter. Only the numbers 101 through 199 inclusive should be used for user programs (under ISIS-II); the other numbers (0-100 and 200-255) are reserved for system programs. The system displays the error in the following format: ERROR nnn, USER PC mmmm where nnn is the error number specified in the call and mmmm is the return address in the calling program.
2. (Assembly language only) The address of a memory location for return of an error number.

No nonfatal error codes are returned. Fatal 33 may occur.

PL/M ERROR Call Example

```
ERROR:
  PROCEDURE (errnum) EXTERNAL;
      DECLARE (errnum) ADDRESS;
  END ERROR;
.
.
.
DECLARE ENUM ADDRESS;
.
.
.
CALL ERROR (ENUM);
.
.
.
```

Assembly Language ERROR Call Example

```
             EXTRN      ISIS
ERROR        EQU        12           ;CALL IDENTIFIER
;
             MVI        C,ERROR      ;LOAD IDENTIFIER
             LXI        D,EBLK       ;LOAD PARAM ADDR
             CALL       ISIS
;
EBLK:
ERNUM:       DS         2            ;ERROR NUMBER FIELD
             DW         STATUS       ;ISIS-II WANTS TO RETURN A
STATUS:      DS         2            ;STATUS, SO PUT IT HERE
```

Program Execution

Your program can transfer control to another program by using the LOAD system call, or to ISIS by using the EXIT system call. The LOAD call loads another program and then transfers control to it, to the Monitor, or back to the calling program. The EXIT call is used to terminate processing and return to ISIS.

LOAD - Load a File of Executable Code and Transfer Control

The LOAD call allows your program to load a LOCATED or absolute object file. After the file is loaded, control is passed to the loaded program, the calling program, or to the Monitor depending on the value of a parameter.

A parameter list of five variables must be passed with the LOAD call:

1. *path$p*, the address of an ASCII string containing the name of the file to be loaded. The string can contain leading spaces but no embedded spaces. It must be terminated by a character other than a letter, digit, colon (:), or a period (.). You can use a space.
2. *load$offset*, a bias value to be added to the load address, causing the program to be loaded at the adjusted address. The use of the bias does not mean that the program is relocatable. Usually the code cannot be executed at the biased address. For most applications, the bias will be zero.
3. *control$sw*, a value indicating where control is transferred after the load. A value of zero returns control to the calling program. The debug toggle is unchanged (see also the *Intellec Series III Console Operating Instructions*).

   A value of 1 transfers control to the loaded program. The debug toggle is reset. If the program is not a main program, its entry point is zero, which causes control to vector through location zero to the Monitor.

A value of 2 transfers control to the Monitor. The debug toggle is set. The Monitor Execute (G) command can be used to start the program.

4. *entry$p*, the address of a memory location for the return of the loaded program entry point address when the control value is zero. The entry point is obtained from the loaded program. A zero is returned if the program is not a main program.
5. *status$p*, the address of a memory location for the return of a nonfatal error number.

Nonfatal error codes returned: 3,4,5,12,13,22,23,28,34

Fatal error codes returned: 1,15,16,24,30,33

PL/M LOAD Call Example

```
LOAD:
  PROCEDURE (path$p, load$offset, control$sw, entry$p, status$p) EXTERNAL;
      DECLARE (path$p, load$offset, control$sw, entry$p, status$p) ADDRESS;
  END LOAD;
.
.
.
DECLARE FILNAM(15) BYTE;
DECLARE ENTRY ADDRESS;
DECLARE STATUS ADDRESS;
.
.
.
CALL LOAD (.FILNAM,0,1,.ENTRY,.STATUS);
IF STATUS <> 0 THEN ...
.
.
.
```

Assembly Language LOAD Call Example

```
                EXTRN     ISIS
LOAD            EQU       6             ;CALL IDENTIFIER
;
                MVI       C,LOAD        ;LOAD IDENTIFIER
                LXI       D,LBLK        ;LOAD PARAM ADDR
                CALL      ISIS
                LDA       LSTAT         ;TEST ERROR STATUS
                ORA       A
                JNZ       EXCEPT        ;BRANCH IF ERROR
;
LBLK:
                DW        FILNAM        ;POINTER TO FILE NAME
BIAS:           DS        2             ;BIAS FIELD
SWITCH:         DS        2             ;CONTROL SWITCH
                DW        ENAD          ;POINTER TO ENTRY ADDRESS
                DW        LSTAT         ;POINTER TO STATUS
;
FILNAM:         DS        15            ;FILE NAME FIELD
ENAD:           DS        2             ;ENTRY POINT ADDR (RETURN)
LSTAT:          DS        2             ;STATUS (RETURNED)
;
```

EXIT - Terminate Program and Return to ISIS-II

The EXIT call terminates execution and returns to ISIS-II. All open files are closed, with the exception of :CO: and :CI:. The current system console assignment is not changed.

You pass no parameters in a PL/M call to EXIT. In an assembly language call, one parameter is passed: the address of a memory location for return of an error number.

No error codes are returned

PL/M EXIT Call Example

```
EXIT:
  PROCEDURE EXTERNAL;
  END EXIT;
.
.
.
CALL EXIT;
.
.
.
```

Assembly Language EXIT Call Example

```
          EXTRN     ISIS
EXIT      EQU       9          ;CALL IDENTIFIER
;
          MVI       C,EXIT     ;LOAD IDENTIFIER
          LXI       D,EBLK     ;LOAD PARAM ADDR
          CALL      ISIS
;
EBLK:
          DW        ESTAT      ;POINTER TO STATUS
;
ESTAT:    DS        2          ;STATUS FIELD
;
```

Monitor I/O Interface Routines

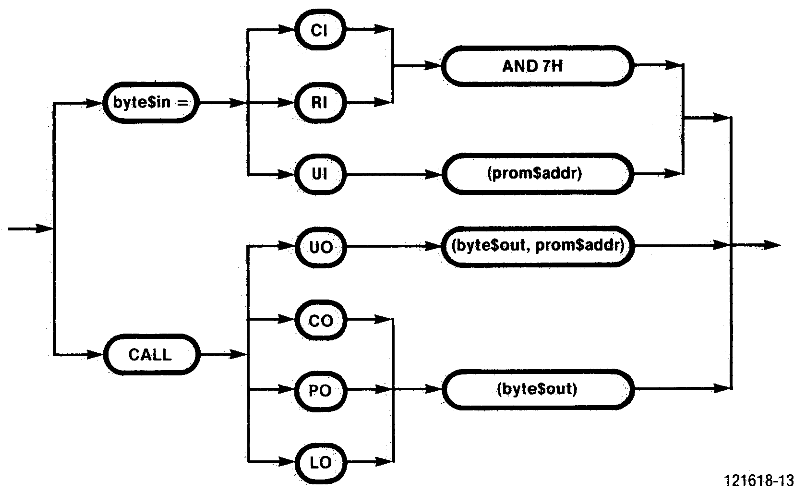
The Monitor contains the following I/O interface routines:

- Console Input, which reads a character entered at the system console.
- Console Output, which writes a character to the system console.
- Reader Input, which reads a character from the system reader device.
- Punch Output, which writes a character to the system punch device.
- List Output, which writes a character to the system list device.
- UPP Input, which reads a byte from the Universal PROM programmer. (Series II only.)
- UPP Output, which writes a byte to the Universal PROM programmer. (Series II only.)

These routines are available as ISIS-II calls. The following sections describe how to use each of these routines, how and where information is passed to them, how and where information is returned, and an example of each.

NOTE

Assembly language calls to Monitor I/O routines change the contents of the registers. If the contents of a register must be preserved, you should save them before calling the Monitor and then restore them after return from the Monitor I/O routine.



Monitor Routine Usage in PL/M

Table 3-2. Monitor Routine Usage

Procedure or Function	Name of Monitor Routine	Performs	PL/M Results	Assembly Language Results
F	CI	Console Input	fills *byte$p*	fills Reg. A
			with character from console input device	
P	CO	Console Output	sends *byte$out*	sends byte from Reg. C
			to console output device	
F	RI	Reader Input	fills *byte$p*	fills Reg. A
			with character from paper tape reader device	
P	PO	Punch Output	sends *byte$out*	sends byte from Reg. C
			to paper tape punch device	
P	LO	List Output	sends *byte$out*	sends byte from Reg. C
			to system list device	

Procedure or Function	Name of Monitor Routine	Performs	PL/M Results	Assembly Language Results
F	UI	Universal PROM Programmer Input	fills *byte$p*	fills Reg. A
			with 8 bits from the PROM address supplied as the contents of	
			prom$addr in the call	Register pair BC
P	UO	Universal PROM Programmer Output	sends *byte$out*	sends byte from Reg. C
			to the PROM address supplied as the contents of	
			prom$addr in the call	the most-significant byte in the D register and the least-significant byte in the E register

CI - Console Input Routine

The Console Input routine reads a character entered at the Intellec Console input device and returns it as a byte variable (if called from PL/M) or in the A-register (if called from the assembler). No parameters are passed to the routine. The routine, once called, loops until a character is input at the console device. This character is not echoed on the Console Output device.

The name of the Console Input routine in SYSTEM.LIB is CI.

PL/M CI Call Example

This example routine reads a string of characters from the Console device. The routine terminates when a carriage return is detected or when the number of characters specified by BUFSIZ has been read. If a carriage return is detected, the DONE code is executed, and if the buffer is filled, the OVFL code is executed.

```
CI: PROCEDURE BYTE EXTERNAL;        /*ENTRY POINT INTO SYSTEM.LIB*/
  END CI;

DECLARE BUFSIZ LITERALLY '122';     /*BUFFER SIZE*/
DECLARE BUFFER(BUFSIZ) BYTE;        /*BUFFER FOR STORING CHARACTERS*/
DECLARE INDEX BYTE;                 /*INDEX INTO BUFFER*/
DECLARE CR LITERALLY '0DH';         /*CARRIAGE RETURN*/

INDEX = 0;
BUFFER(INDEX) = CI AND 7FH;         /*READ IN CHARACTER AND STRIP OFF*/
                                    /*PARITY BIT*/

DO WHILE BUFFER(INDEX) <> CR;
  IF INDEX < LAST (BUFFER);
    DO;
      INDEX = INDEX + 1;
      BUFFER(INDEX) = CI AND 7FH;   /*CONTINUE READING UNTIL A CARRIAGE*/
                                    /*RETURN HAS BEEN INPUT OR THE*/
                                    /*BUFFER IS FULL*/
      END;
    ELSE
      DO;
        /*OVFL CODE*/
        END;
  END;
  /*DONE CODE*/
```

Assembly Language CI Call Example

```
                EXTRN    CI          ;ENTRY POINT INTO SYSTEM.LIB
                                     ;FOR CI
BUFSIZ          EQU      122         ;BUFFER SIZE
CR              EQU      0DH         ;CARRIAGE RETURN
BUFFER:         DS       BUFSIZ      ;BUFFER
;
                LXI      H,BUFFER    ;HL POINT TO BEGINNING OF
                                     ;BUFFER
                MVI      D,BUFSIZ    ;SET UP BUFFER SIZE COUNTER
LOOP:
                CALL     CI          ;GET CHARACTER
                ANI      7FH         ;STRIP OFF PARITY
                MOV      M,A         ;STORE IT IN BUFFER
                CPI      CR          ;IS IT A CARRIAGE RETURN
                JZ       DONE        ;IF IT IS, JUMP TO THE DONE
                                     ;CODE
                INX      H           ;OTHERWISE, MOVE THE
                                     ;BUFFER POINTER
                DCR      D           ;DECREASE CHARACTER
                                     ;COUNT
                JZ       OVFL        ;IF BUFFER FULL, JUMP TO THE
                                     ;OVFL CODE
                JMP      LOOP        ;GET THE NEXT CHARACTER
DONE:
                                     ;DONE CODE

OVFL:
                                     ;OVFL CODE
```

CO - Console Output Routine

The Console Output routine takes a single character and transmits it to the system console output device. The character is passed as a byte parameter if called from PL/M or passed in the C-register if called from the assembler.

The name of the Console Output routine in SYSTEM.LIB is CO.

PL/M CO Call Example

This example uses the Console Output routine to output a string of characters to the Console device. The routine terminates after a carriage return is detected in the output string and is transmitted to the Console device. In this simple example there is no check to see if the buffer has been exhausted.

```
CO: PROCEDURE (CHAR) EXTERNAL;      /*ENTRY POINT INTO SYSTEM.LIB*/
  DECLARE CHAR BYTE;
  END CO;
```

```
DECLARE BUFFER(122) BYTE;           /*BUFFER CONTAINING STRING TO BE*/
                                    /*OUTPUT*/
DECLARE INDEX BYTE;                 /*INDEX INTO BUFFER*/
DECLARE CR LITERALLY '0DH';         /*CARRIAGE RETURN*/

INDEX = 0;
CALL CO(BUFFER(INDEX));             /*OUTPUT THE FIRST CHARACTER*/
DO WHILE BUFFER(INDEX) < > CR;
  INDEX = INDEX + 1;
  CALL CO(BUFFER(INDEX));           /*CONTINUE OUTPUTTING UNTIL A*/
                                    /*CARRIAGE RETURN HAS BEEN OUTPUT*/
END;
```

Assembly Language CO Call Example

```
            EXTRN       CO          ;ENTRY POINT INTO SYSTEM.LIB
                                    ;FOR CO
CR          EQU         0DH         ;CARRIAGE RETURN
BUFFER:     DS          122         ;BUFFER CONTAINING OUTPUT
                                    ;STRING
;
            LXI         H,BUFFER    ;HL CONTAIN ADDRESS OF
                                    ;BUFFER
LOOP:
            MOV         C,M         ;GET CHARACTER FROM
                                    ;BUFFER
            CALL        CO          ;OUTPUT THE CHARACTER TO
                                    ;THE CONSOLE
            MVI         A,CR
            CMP         M           ;IS IT A CARRIAGE RETURN?
            JZ          EXIT        ;GO TO EXIT IF IT IS
            INX         H           ;INCREMENT BUFFER POINTER
            JMP         LOOP        ;OUTPUT NEXT CHARACTER
EXIT:
```

RI - Reader Input Routine

The Reader Input routine reads a single character from the system Reader device and returns it as a byte value (if called from PL/M) or in the A-register (if called from the assembler). If a character is not read within 250 milliseconds, an end-of-file condition is simulated and a value of zero is returned with the 8080 carry condition code set to 1. Thus the condition of the carry bit specifies whether or not valid data was returned. Your program must handle the end-of-file character when it is read.

The name of the Reader Input routine in SYSTEM.LIB is RI.

PL/M RI Call Example

The following example uses the Reader Input routine to read a string of characters from paper tape and store them in a buffer. The operation is terminated when the reader runs out of tape or a control Z (1AH) character is read (used here as an end-of-file character). In this simple example there is no check made for overflowing the buffer area.

```
RI: PROCEDURE BYTE EXTERNAL;        /*ENTRY POINT INTO SYSTEM.LIB*/
  END RI;
```

```
DECLARE BUFFER$PTR ADDRESS;
DECLARE BUFFER BASED BUFFER$PTR BYTE;
DECLARE ENDFILE LITERALLY '1AH';   /*END OF FILE CONDITION*/
DECLARE TEMP BYTE;                 /*TEMPORARY VARIABLE IN CASE*/
                                   /*READ RESULTS IN END OF FILE*/
                                   /*CONDITION*/
DECLARE ESCAPE BYTE;               /*BOOLEAN TO DECIDE IF*/
                                   /*SHOULD EXIT ROUTINE*/
DECLARE TRUE LITERALLY '0FFH';
DECLARE FALSE LITERALLY '00H';

BUFFER$PTR = .MEMORY;              /*INITIALIZE THE BUFFER*/
                                   /*POINTER*/
TEMP = RI;                         /*READ IN FIRST CHARACTER*/
IF CARRY THEN ESCAPE = TRUE;
    ELSE ESCAPE = FALSE;
DO WHILE NOT ESCAPE;
  BUFFER = TEMP AND 7FH;           /*STORE THE CHARACTER AFTER*/
                                   /*STRIPPING OFF PARITY BIT*/
  IF BUFFER <> END$FILE THEN
    DO
      BUFFER$PTR = BUFFER$PTR + 1;
      TEMP = RI;                   /*CONTINUE READING IN THE NEXT*/
                                   /*CHARACTER*/
      IF CARRY THEN ESCAPE = TRUE;
    END;
END;
```

Assembly Language RI Call Example

```
            EXTRN       RI          ;ENTRY POINT INTO SYSTEM.LIB
                                    ;FOR RI
EOF         EQU         1AH         ;END OF FILE CONDITION
                                    ;
            LXI         H,BUFFER    ;HL POINTS TO BEGINNING OF
                                    ;BUFFER
LOOP:
            CALL        RI          ;GET A CHARACTER
            JC          EXIT        ;EXIT IF CARRY BIT SET (I.E. 250
                                    ;MS. TIME-OUT)
            ANI         7FH         ;STRIP OFF PARITY BIT
            MOV         M,A         ;STORE IT IN THE BUFFER
            CPI         EOF         ;IS IT AN EOF CHARACTER?
            JZ          EXIT        ;EXIT IF IT IS
            INX         H           ;OTHERWISE, MOVE THE
                                    ;BUFFER POINTER
            JMP         LOOP        ;GET THE NEXT CHARACTER
EXIT:
                                    ;EXIT CODE

BUFFER:     DS          1           ;EXPANDABLE BUFFER
```

PO - Punch Output Routine

The Punch Output routine takes a single character and transmits it to the System Punch device. The character is passed as a byte parameter if called from PL/M or passed in the C-register if called from assembler.

The name of the Punch Output routine in SYSTEM.LIB is PO.

PL/M PO Call Example

This example uses the Punch Output routine to output a string of characters to the Punch device. The routine terminates after a Control Z(1AH) is detected in the output string and is transmitted to the Punch device. In this simple example there is no check to see if the buffer has been exhausted.

```
PO: PROCEDURE (CHAR) EXTERNAL;          /* ENTRY POINT INTO SYSTEM.LIB*/
  DECLARE CHAR BYTE;
  END PO;

DECLARE BUFFER(122) BYTE;               /* BUFFER CONTAINING STRING TO */
                                        /* BE OUTPUT */
DECLARE INDEX BYTE;                     /* INDEX INTO BUFFER */
DECLARE END$FILE LITERALLY'1AH';        /* END OF FILE*/

INDEX = 0;
CALL PO(BUFFER(INDEX));                 /* OUTPUT THE FIRST CHARACTER */
DO WHILE BUFFER(INDEX) < > END$FILE;
  INDEX = INDEX + 1;
  CALL PO(BUFFER(INDEX));               /* CONTINUE TO OUTPUT UNTIL */
                                        /* AN END-OF-FILE HAS BEEN PUNCHED */
END;
```

Assembly Language PO Call Example

```
            EXTRN     PO          ;ENTRY POINT INTO SYSTEM.LIB FOR PO
EOF         EQU       1AH         ;CONTROL/Z
BUFFER:     DS        122         ;BUFFER CONTAINING OUTPUT STRING
;
            LXI       H,BUFFER    ;HL CONTAINS ADDRESS OF BUFFER
LOOP:
            MOV       C,M         ;GET CHARACTER FROM BUFFER
            CALL      PO          ;OUTPUT THE CHARACTER TO THE
                                  ;PUNCH
            MVI       A,EOF       ;LOAD THE EOF CHARACTER INTO THE
                                  ;A-REG
            CMP       M           ;IS IT AN END-OF-FILE?
            JZ        EXIT        ;GO TO EXIT IF IT IS
            INX       H           ;INCREMENT BUFFER POINTER
            JMP       LOOP        ;OUTPUT NEXT CHARACTER
EXIT:
```

LO - List Output Routine

The List Output routine takes a single character and transmits it to the system list device. The character is passed as a byte parameter if called from PL/M or passed in the C-register if called from an assembly language program.

The name of the List Output routine in SYSTEM.LIB is LO.

PL/M LO Call Example

This example uses the List Output routine to output a string of characters to the list device. The routine terminates after an ETX character (03H) is detected in the output string and is transmitted to the list device. In this simple example there is no check to see if the buffer has been exhausted.

```
LO: PROCEDURE(BUFF) EXTERNAL;        /* ENTRY POINT INTO SYSTEM.LIB */
  DECLARE CHAR BYTE;
  END LO;
```

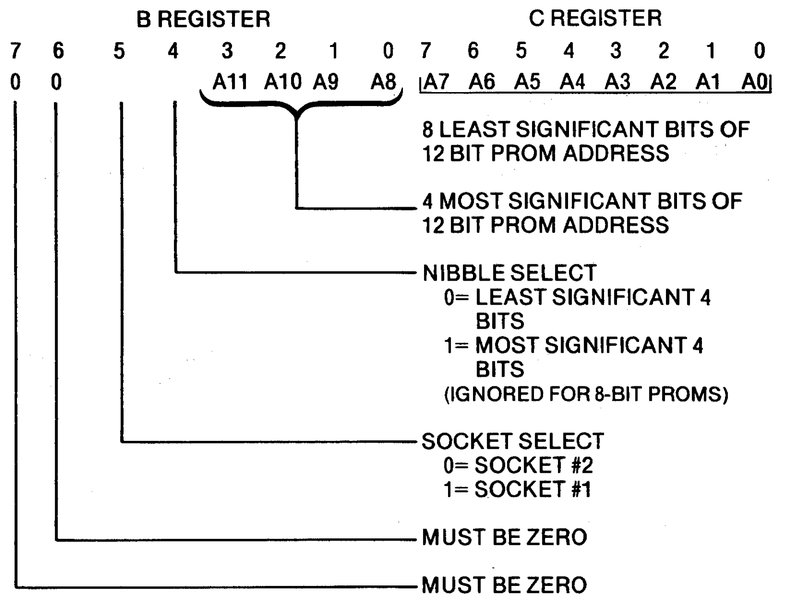
```
DECLARE BUFFER(122) BYTE;          /* BUFFER CONTAINING STRING TO */
                                   /* BE OUTPUT */
DECLARE INDEX BYTE;                /* INDEX INTO BUFFER */
DECLARE ETX LITERALLY'03H';        /* TERMINAL CHARACTER */

INDEX = 0;
CALL LO(BUFFER(INDEX));            /* OUTPUT THE FIRST CHARACTER */
DO WHILE BUFFER(INDEX) < > ETX;
  INDEX = INDEX + 1;
  CALL LO(BUFFER(INDEX));          /* CONTINUE TO OUTPUT UNTIL AN ETX */
                                   /* HAS BEEN TRANSMITTED */

END;
```

Assembly Language LO Call Example

```
           EXTRN   LO          ;ENTRY POINT INTO SYSTEM.LIB FOR LO
ETX        EQU     03H         ;TERMINATING CHARACTER
BUFFER:    DS      122         ;BUFFER CONTAINING OUTPUT STRING
;
           LXI     H,BUFFER    ;HL CONTAINS ADDRESS OF BUFFER
LOOP:
           MOV     C,M         ;GET CHARACTER FROM BUFFER
           CALL    LO          ;OUTPUT THE CHARACTER TO THE LIST
           MVI     A,ETX       ;LOAD EXT CHARACTER INTO A-REG
           CMP     M           ;IS IT AN END OF FILE?
           JZ      EXIT        ;BRANCH TO EXIT IF IT IS
           INX     H           ;INCREMENT BUFFER POINTER
           JMP     LOOP        ;OUTPUT NEXT CHARACTER
EXIT:
```

UI - Universal PROM Programmer Input Routine (Series II Only)

This routine reads eight bits of data from the Universal PROM Programmer (UPP) and returns it as a byte value (if called from PL/M) or in the A-register (if called from the Assembler). You send the PROM address of the desired data as an address parameter (if calling from PL/M) or load it into register pair BC (if calling from an assembly language program), as shown below:


B REGISTER
C REGISTER
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
0 0 A11 A10 A9 A8 A7 A6 A5 A4 A3 A2 A1 A0
8 LEAST SIGNIFICANT BITS OF 12 BIT PROM ADDRESS
4 MOST SIGNIFICANT BITS OF 12 BIT PROM ADDRESS
NIBBLE SELECT
0= LEAST SIGNIFICANT 4 BITS
1= MOST SIGNIFICANT 4 BITS
(IGNORED FOR 8-BIT PROMS)
SOCKET SELECT
0= SOCKET #2
1= SOCKET #1
MUST BE ZERO
MUST BE ZERO

Before calling the UI routine, you should call the Universal PROM Programmer Status routine (described later in this chapter) to ascertain the current status of the UPP. If an error occurs in the process of reading from the UPP, the content of the A-register is meaningless. Thus, you should follow the call to the UPP input routine with another call to the UPP status routine.

The name of the Universal PROM Programmer Input routine in SYSTEM.LIB is UI.

PL/M UI Call Example

The following PL/M procedure reads the first LENGTH locations from a PROM in socket 1 into the user buffer pointed to by BUFFER$ADDRESS. If it encounters any UPP error, it will stop immediately and return the nonzero value of the UPP status byte; otherwise, it will return a value of zero after LENGTH locations have been read.

```
UI: PROCEDURE(PROM$ADDR) BYTE EXTERNAL;/* RETURNS DATA IN PROM */
  DECLARE PROM$ADDR ADDRESS;              /* AT PROM$ADDR */
  END UI;
UPPS: PROCEDURE BYTE EXTERNAL;            /* RETURNS UPP STATUS */
  END UPPS;
  READ$PROM$TO$BUFFER: PROCEDURE (BUFFER$ADDRRESS,LENGTH) BYTE;
  DECLARE (BUFFER$ADDRESS,LENGTH) ADDRESS;
 DECLARE (BUFFER BASED BUFFER$ADDRESS)(1) BYTE;
 DECLARE SOCKET$1 LITERALLY'2000H';      /* SELECT SOCKET 1 MASK */
 DECLARE SOCKET$2 LITERALLY'0000H';      /* SELECT SOCKET 2 MASK */
 DECLARE BUSY LITERALLY'01H';            /* UPP BUSY STATUS */
 DECLARE COMPLETE LITERALLY'02H';        /* OPERATION COMPLETE STATUS */
 DECLARE STATUS BYTE;                    /* SAVE FOR STATUS */
 DECLARE I ADDRESS;
 DO I = 0 TO LENGTH - 1;
   DO WHILE ((UPPS AND BUSY) < > 0);     /* WAIT FOR UPP READY */
   END;
   BUFFER(I) = UI(I OR SOCKET$1);        /* READ DATA */
   IF (STATUS: = UPPS) < > COMPLETE
   THEN                                  /* CHECK STATUS */
     RETURN STATUS;
 END;
   RETURN 0;
END READ$PROM$TO$BUFFER;
```

Assembly Language UI Call Example

The following assembly language procedure implements the same function as the preceding PL/M example. Note that the status value is returned in the A-register.

```
            EXTRN     UI        ;ROUTINE TO READ FROM DATA
            EXTRN     UPPS      ;ROUTINE TO READ UPP STATUS
BUSY        EQU       01H       ;UPP BUSY CODE
COMPLT      EQU       02H       ;UPP OPERATION COMPLETE CODE
SOC1        EQU       2000H     ;SOCKET 1 SELECT MASK
SOC2        EQU       0000H     ;SOCKET 2 SELECT MASK
BUFAD:      DW        0         ;SAVE FROM BUFFER POSITION
LENGTH:     DW        0         ;SAVE FOR COUNT
PROMAD:     DW        0         ;SAVE FOR PROM ADDRESS OR'D WITH
                                ;'SOC1'
RPTB:       MOV       H,B       ;STORE 'BUFFER$ADDRESS'
            MOV       L,C
            SHLD      BUFAD
```

```
                LXI     H,SOC1      ;SET UP PROMAD TO 'SOC1' TO GET
                                    ;'SOC1 or ADDR'
                SHLD    PROMAD
                XCHG                ;STORE 'LENGTH'
                SHLD    LENGTH
RPTB1:          MOV     A,H         ;CHECK IF NO LOCATIONS LEFT TO
                                    ;READ
                ORA     L
                RZ                  ;RETURN WITH RESULT ZERO IF DONE
RPTB2:          CALL    UPPS        ;CHECK FOR 'UPP NOT BUSY' STATUS
                ANI     BUSY
                JNZ     RPTB2       ;LOOP WHILE BUSY
                LHLD    PROMAD      ;SET UP PROM ADDRESS
                MOV     B,H         ;UI EXPECTS ADDRESS IN BC
                MOV     C,L
                INX     H           ;INCREMENT POSITION FOR NEXT TIME
                SHLD    PROMAD      ;STORE IT BACK
                CALL    UI          ;READ PROM LOCATION
                LHLD    BUFAD       ;READ CURRENT BUFFER POSITION
                MOV     M,A         ;STORE PROM VALUE
                INX     H           ;INCREMENT POSITION FOR NEXT TIME
                SHLD    BUFAD
                CALL    UPPS        ;CHECK IF OPERATION OK
                CPI     COMPLT
                RNZ                 ;RETURN WITH BAD STATUS IF NOT
                                    ;COMPLETE
                LHLD    LENGTH      ;CHECK LOOP COUNTER
                DCX     H           ;DECREMENT COUNT
                SHLD    LENGTH
                JMP     RPTB1       ;LOOP
```

UO - Universal PROM Programmer Output Routine (Series II Only)

This routine transfers eight bits of data as a byte parameter (if called from PL/M) or from the C-register (if called from the Assembler) to the UPP. You pass the PROM address (to be programmed) as an address parameter if calling from PL/M, or with the most significant byte in the D-register and the least signficant byte in the E-register, if calling from an assembly language program. See the Universal PROM Programmer Input Routine (UI) for a description of the address bits.

Before calling the Universal PROM Programmer Output routine, you should call the Universal PROM Programmer Status routine (described later in this chapter) to ascertain the current status of the UPP. You should also follow the call to the UPP output routine with another call to the UPP status routine to determine the success of the operation.

The name of the Universal PROM Programmer output routine in SYSTEM.LIB is UO.

PL/M UO Call Example

The following PL/M procedure programs the first LENGTH locations (of a PROM in socket 2) from the user buffer pointed to by BUFFER$ADDRESS. If it encounters any UPP errors, it will stop immediately and return the nonzero value of the UPP status byte; otherwise, it will return a value of zero after LENGTH locations have been programmed.

```
UO; PROCEDURE(PROM$DATA,PROM$ADDR) EXTERNAL;  /* PROGRAM PROM LOCATION */
  DECLARE PROM$DATA BYTE;                    /* 'PROM$ADDR' WITH DATA */
  DECLARE PROM$ADDR ADDRESS;                 /* 'PROM$DATA' */
END UO;
UPPS: PROCEDURE BYTE EXTERNAL;               /* RETURNS UPP STATUS */
END UPPS;
PROGRAM$PROM$FROM$BUFFER: PROCEDURE(BUFFER$ADDRESS,LENGTH) BYTE;
  DECLARE (BUFFER$ADDRESS,LENGTH) ADDRESS;
  DECLARE (BUFFER BASED BUFFER$ADDRESS)(1) BYTE;
  DECLARE SOCKET$1 LITERALLY'2000H';         /* SELECT SOCKET 1 MASK */
  DECLARE SOCKET$2 LITERALLY'0000H';         /* SELECT SOCKET 2 MASK */
  DECLARE BUSY LITERALLY'01H';               /* UPP BUSY STATUS */
  DECLARE COMPLETE LITERALLY'02H';           /* OPERATION COMPLETE STATUS */
  DECLARE STATUS BYTE;                       /* SAVE FOR STATUS */
  DECLARE I ADDRESS;
  DO I = 0 TO LENGTH - 1;
    DO WHILE ((UPPS AND BUSY) < >,0);        /* WAIT FOR UPP READY */
    END;
    CALL UO(BUFFER(I), I OR SOCKET$1);       /* PROGRAM LOCATION */
    IF (STATUS: = UPPS) < > COMPLETE
      THEN                                   /* CHECK STATUS */
      RETURN STATUS;
  END;
  RETURN 0;
END PROGRAM$PROM$FROM$BUFFER;
```

Assembly Language UO Call Example

The following assembly language procedure implements the same function as the preceding PL/M example. Note that the status value is returned in the A-register.

```
            EXTRN     UO        ;ROUTINE TO WRITE PROM DATA
            EXTRN     UPPS      ;ROUTINE TO READ UPP STATUS
BUSY        EQU       01H       ;UPP BUSY CODE
COMPLT      EQU       02H       ;UPP OPERATION COMPLETE CODE
SOC1        EQU       2000H     ;SOCKET 1 SELECT MASK
SOC2        EQU       0000H     ;SOCKET 2 SELECT MASK
BUFAD:      DW        0         ;SAVE FOR BUFFER POSITION
LENGTH:     DW        0         ;SAVE FOR COUNT
PROMAD:     DW        0         ;SAVE FOR PROM ADDRESS OR'D
                                ;WITH 'SOC1'
PPFB:       MOV       H,B       ;STORE 'BUFFER$ADDRESS'
            MOV       L,C
            SHLD      BUFAD
            LXI       H,SOC1    ;SET UP PROMAD TO 'SOC1' TO GET
                                ;'SOC1 OR ADDR'
            SHLD      PROMAD
            XCHG                ;STORE 'LENGTH'
            SHLD      LENGTH
PPFB1:      MOV       A,H       ;CHECK IF NO LOCATIONS LEFT
                                ;TO READ
            ORA       L
            RZ                  ;RETURN WITH RESULT ZERO IF DONE
PPFB2:      CALL      UPPS      ;CHECK FOR 'NOT BUSY' STATUS
            ANI       BUSY
            JNZ       PPFB2     ;LOOP WHILE BUSY
            LHLD      PROMAD    ;SET UP PROM ADDRESS
            MOV       B,H       ;UO EXPECTS ADDRESS IN BC
            MOV       C,L
```

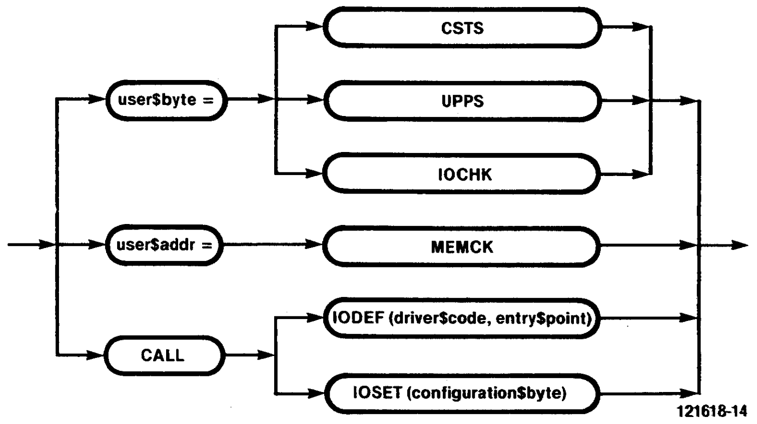
```
INX       H         ;INCREMENT POSITION FOR NEXT TIME
SHLD      PROMAD    ;STORE IT BACK
LHLD      BUFAD     ;READ CURRENT BUFFER POSITION
MOV       A,M       ;READ PROM VALUE
INX       H         ;INCREMENT POSITION FOR NEXT TIME
SHLD      BUFAD
CALL      UO        ;PROGRAM VALUE INTO PROM
CALL      UPPS      ;CHECK IF OPERATION OK
CPI       COMPLT
RNZ                 ;RETURN WITH BAD STATUS IF NOT
                    ;COMPLETE
LHLD      LENGTH    ;CHECK LOOP COUNTER
DCX       H         ;DECREMENT COUNT
SHLD      LENGTH
JMP       PPFB1     ;LOOP
```

System Status Routines

The Monitor contains the following system status routines:

- Console input status, which determines if a character is ready for input from the Console input device.
- Universal PROM programmer status, which reads an eight-bit status byte from the Universal PROM Programmer.
- Define I/O drivers, which links non-standard I/O devices to the Monitor.
- System I/O configuration status, which returns an eight bit byte describing the current I/O assignments.
- Set I/O configuration, which changes the current I/O assignments.
- RAM memory status, which returns the highest RAM address available to the user.

The following sections describe how to use each of these routines, how and where information is passed to them, how and where information is returned, and an example of each.



Status Routine Usage (in PL/M)

Table 3-3. System Status Routine Usage

Procedure or Function	Name of Status Routine	Reports On	Returns	
			PL/M	Assembly Language
F	CSTS	Console Input Status	00H if no key, 0FFH if any key pressed	The values shown at left are returned in Reg. A
F	UPPS	Universal PROM Programmer Status	Eight-bit value[1]	Eight-bit value in A-register
P	IODEF	Existence and location of user-written programs (I/O drivers) for non-standard devices	Nothing, IODEF tells the Monitor what routines are to be used.	
F	IOCHK	I/O Configuration	Eight-bit value describing the currently configured devices[1]	Value returned in the A-register
P	IOSET	New System Configuration	Eight-bit value is sent, setting new I/O assignments	Value is sent in the C-register
F	MEMCK	Highest address of contiguous memory available to user programs	The highest (address (a 16-bit value)	Address value returned in the H and L registers

[1]Bit meanings in routine description

CSTS - Console Input Status Routine

The Console Input Status routine tests the Console device to determine if a character is ready for input. If this routine is called from PL/M, it returns a value of 00H if no key has been pressed since the last call to the Console Input Routine (CI) or a value of 0FFH if a key has been pressed. If this routine is called from an assembly language program, then the 00H or 0FFH value will be returned in the A-Register.

The name of the Console Input Status routine in SYSTEM.LIB is CSTS.

PL/M CSTS Call Example

The following example tests the Console Input Device during a Console Output operation so that the operator has the facility to signal that the output operation be terminated. A Control C character (03H) entered at the Console Input Device will signal this termination.

```
CSTS: PROCEDURE BYTE EXTERNAL;          /* ENTRY POINT INTO SYSTEM.LIB FOR */
                                        /* CSTS */
  END CSTS;
CI: PROCEDURE BYTE EXTERNAL;            /* ENTRY POINT INTO SYSTEM.LIB FOR CI */
  END CI;

DECLARE CTLC LITERALLY '03H';           /* CONTROL/C SIGNALS TERMINATE */
                                        /* OPERATION */
```

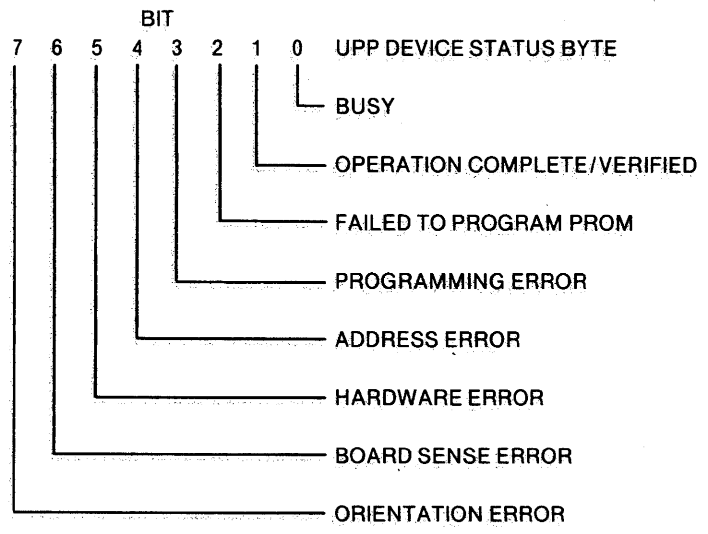
```
IF CSTS THEN
  DO;                                   /* A KEY HAS BEEN PRESSED */
    IF CI AND 7FH = CTLC THEN
      DO;

                                        /* CONTROL/C RECEIVED. TERMINATE */
                                        /* OUTPUT OPERATION. */


      END;
END;
```

Assembly Language CSTS Call Example

```
                EXTRN     CSTS          ;ENTRY POINT INTO SYSTEM.LIB FOR
                                        ;CSTS
                EXTRN     CI            ;ENTRY POINT INTO SYSTEM.LIB FOR CI
    CTLC        EQU       03H           ;CONTROL/C SIGNALS TERMINATE
                                        ;OUTPUT

                CALL      CSTS          ;GET CONSOLE STATUS
                RRC                     ;ROTATE TO CARRY FLAG
                JNC       CONT          ;NO CHARACTER, CONTINUE OUTPUT
                                        ;OPERATION
                CALL      CI            ;THERE IS A CHARACTER, GET IT
                ANI       7FH
                CPI       CTLC          ;IS IT A CONTROL/C
                JZ        TERM          ;IF YES, BRANCH TO TERMINATE CODE
    CONT:
    ;
                                        ;CODE TO CONTINUE OUTPUT
                                        ;OPERATION


    TERM:
                                        ;CODE TO TERMINATE OUTPUT
                                        ;OPERATION
```

UPPS - Universal PROM Programmer Status Routine (Series II Only)

The Universal PROM Programmer Status routine returns an eight-bit status byte as a byte value (if called from PL/M) or in the A-register (if called from an assembly language program). The meaning of the bits in the status byte are



If the UPP is not present or is timed-out, the value returned is 0FFH.

For additional information concerning the meaning of the status bits, see the *Universal PROM Programmer Hardware Reference Manual.*

The name of the Universal PROM Programmer Status routine in SYSTEM.LIB is UPPS.

See the description of UI and UO for examples of the use of UPPS.

IODEF - I/O Definition Routine

You can write I/O drivers for non-standard I/O devices and make them a part of the Monitor. By making your drivers a part of the Monitor, they become accessible to other Intellec programs.

This section does not describe how to write an I/O driver for a non-standard device. It only describes how to link the driver, once it is written, to the Monitor. See the specific device hardware manual for information on writing your own driver.

The Monitor has facilities to handle eight user-written drivers. Each driver is assigned to one of the following driver-codes:

0	User-defined Console input
1	User-defined Console output
2	User-defined Reader 1
3	User-defined Reader 2
4	User-defined Punch 1
5	User-defined Punch 2
6	User-defined List device
7	User-defined Console status routine

Only one program can be assigned to each of these codes at any one time. You can change and swap programs, but you cannot have two assigned to the same code.

These codes correspond to the number assignments available in the I/O configuration assignment command and routine.

When you write your own driver for a Console device you have to supply three routines: one for input, one for output, and one to check the Console status.

To link your driver to the Monitor, you must call IODEF and pass it two parameters. If you use a PL/M call, pass the function code (from above) as a byte value and the entry point address of the driver as an address parameter. If you use an assembly language call, the function code is passed in the C-registers and the entry address in the D and E registers (most-significant bits of D and least- significant bits of E).

In defining your own driver you must be careful to locate it in an area of RAM which will not be overwritten by ISIS-II or other programs. If your driver is located entirely in ROM below you will not have this problem.

The name of the I/O Definition routine in SYSTEM.LIB is IODEF.

The external declaration for IODEF in PL/M necessary to link it with SYSTEM.LIB is

```
IODEF: PROCEDURE(driver$code, entry$point) EXTERNAL;
  DECLARE driver$code BYTE;
  DECLARE driver$entry$point ADDRESS;
  END IODEF;
```

The external declaration for IODEF in Assembly Language is EXTRN IODEF.

IOCHK - Check System I/O Configuration Routine

The Check System I/O Configuration routine returns a value which describes the current assignment of physical devices to the logical system devices (Console, Reader, Punch, and List). It is returned as a byte value (if called from PL/M) or in the A-Register (if called from the assembler). This value is divided into four 2-bit fields:

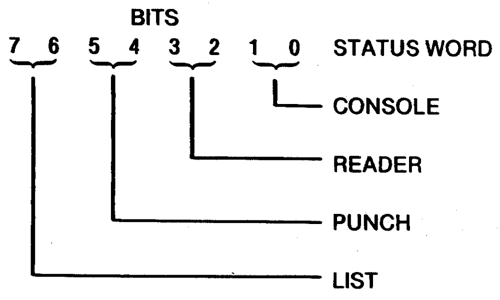


Table 3-4 shows the meaning of the possible values in each field:

Table 3-4: IOCHK Configuration Values

VALUE	CONSOLE	READER	PUNCH	LIST
00	TTY	TTY	TTY	TTY
01	CRT	H.S.READER	H.S.PUNCH	CRT
10	BATCH	U.D.	U.D.	LINE PRINTER
11	U.D.	U.D.	U.D.	U.D.

U.D. - User-defined device using user written routines.

The following are lists of the mask values you must use to check for specific system devices and types of physical devices assigned to them. The mask values are shown in hexadecimal and binary representation.

Masks to check for system device:

CONSOLE	03H	00000011B
READER	0CH	00001100B
PUNCH	30H	00110000B
LIST	C0H	11000000B

Masks to check for physical device codes:

CONSOLE	TTY	00H	00000000B	PUNCH	TTY	00H	00000000B
	CRT	01H	00000001B		PUNCH	10H	00010000B
	BATCH	02H	00000010B		User 1	20H	00100000B
	User	03H	00000011B		User 2	30H	00110000B
READER	TTY	00H	00000000B	LIST	TTY	00H	00000000B
	PUNCH	04H	00000100B		CRT	40H	01000000B
	User 1	08H	00001000B		PRINTER	80H	10000000B
	User 2	0CH	00001100B		User	C0H	11000000B

NOTE

When first loaded, the Monitor is initially configured to give the console and list devices the codes of the first device operated during initialization, and to assign the code for TTY to all other devices.

The name of the check system I/O configuration routine in SYSTEM.LIB is IOCHK.

PL/M IOCHK Call Example

This example checks which device is assigned as the system punch. If the high speed stand-alone device is being used, the program can go ahead and punch a tape because this type of device is presumed to be turned on and ready. However, if another device (e.g., TTY) is assigned as the punch device, a message must be sent to the operator to turn the punch on.

```
IOCHK: PROCEDURE BYTE EXTERNAL;          /* ENTRY POINT INTO SYSTEM.LIB */
  END IOCHK;

DECLARE DEVMSK LITERALLY'00110000B';     /* MASK TO ISOLATE PUNCH ASSIGNMENT */
DECLARE TYPE LITERALLY '00010000B';      /* MASK FOR HIGH SPEED PUNCH DEVICE */

IF (IOCHK AND DEVMSK) < > TYPE THEN
  DO;
                                         /* PUNCH DEVICE IS NOT */
                                         /* HIGH SPEED PUNCH, SO SEND */
                                         /* MESSAGE TO THE OPERATOR */
END;
```

Assembly Language IOCHK Call Example

```
            EXTRN     IOCHK        ;ENTRY POINT INTO SYSTEM.LIB FOR
                                   ;IOCHK
DEVMSK      EQU       00110000B    ;MASK TO ISOLATE PUNCH DEVICE
                                   ;ASSIGNMENT
TYPE        EQU       00010000B    ;MASK FOR HIGH SPEED PUNCH
;
            CALL      IOCHK        ;GET THE STATUS BYTE
            ANI       DEVMSK       ;MASK ALL BUT THE PUNCH
                                   ;ASSIGNMENT
            CPI       TYPE         ;YES, BRANCH TO PUNCH CODE
            JZ        CONT         ;OTHERWISE, EXECUTE CODE TO SEND
                                   ;MESSAGE TO THE OPERATOR

;
CONT:
```

IOSET - Set System I/O Configuration Routine

This routine modifies the system I/O configuration assignments. The new configuration is passed to the routine as a byte parameter (if called from PL/M) or in the C-register (if called from an assembly language routine). Refer to the description of the IOCHK routine for a specification of this configuration byte parameter.

The name of the Set System I/O Configuration routine in SYSTEM.LIB is IOSET.

PL/M IOSET Call Example

The following PL/M sequence changes the Console device to be the CRT:

```
IOSET: PROCEDURE(CONFIG) EXTERNAL;      /* ENTRY POINT INTO SYSTEM.LIB */
                                        /* FOR IOSET */
  DECLARE CONFIG BYTE;
  END IOSET;
IOCHK: PROCEDURE BYTE EXTERNAL;         /* ENTRY POINT INTO SYSTEM.LIB */
                                        /* FOR IOCHK */
  END IOCHK;

DECLARE DEVMSK LITERALLY '00000011B'; /*MASK TO ISOLATE CONSOLE ASSIGNMENT*/
DECLARE NEWDEV LITERALLY '00000001B'; /*MASK TO ASSIGN CRT TO CONSOLE*/

CALL IOSET((IOCHK AND (NOT DEVMSK)) OR NEWDEV);
```

Assembly Language IOSET Call Example

```
               EXTRN    IOSET        ;ENTRY POINT INTO SYSTEM.LIB FOR
                                     ;IOSET
               EXTRN    IOCHK        ;ENTRY POINT INTO SYSTEM.LIB FOR
                                     ;IOCHK
     DEVMSK    EQU      00000011B    ;MASK TO ISOLATE CONSOLE
                                     ;ASSIGNMENT
     NEWDEV    EQU      00000001B    ;MASK TO ASSIGN CRT TO CONSOLE

               CALL     IOCHK        ;GET THE CURRENT I/O STATUS
               ANI      NOT DEVMSK   ;CLEAR THE CURRENT CONSOLE
                                     ;ASSIGNMENT
               ORI      NEWDEV       ;ASSIGN THE CRT TO THE CONSOLE
                                     ;DEVICE
               MOV      C,A
               CALL     IOSET        ;SET THE NEW ASSIGNMENT
```

MEMCK - Check RAM Size Routine

The Check RAM Size routine returns the highest memory address of contiguous memory available to the user. This address is the highest address available after the Monitor has reserved its own memory (320 bytes) at the top of contiguous RAM. This value is returned as an address value (if called from PL/M) or in the H and L registers (if called from an assembly language routine).

The name of the Check RAM Size routine in SYSTEM.LIB is MEMCK.

In a system containing 64k of RAM, the user top of memory is 0FFFFH minus 2k for the Monitor PROM and minus 320 bytes for Monitor RAM space, or 0F6C0H. In a 32k system the top of memory is 7EC0H, since the 2k for the monitor is still located between 62 and 64k.

The BOOT program determines the last 256 byte page of RAM by checking the first byte of each page starting from the beginning of memory. When the first non-RAM location is encountered, the previous byte of memory is considered to be the last byte of RAM in the system. (A 2k region is skipped from E800H to EFFFH. The BOOT ROM resides at these locations.)

This test allows the user to add additional ROMs to the system or to use memory-mapped I/O boards in the system as long as the locations used begin at the start of a 256 byte page of memory. It is always best to use the last page(s) of physical memory for these purposes to preserve RAM for use by ISIS.

Example: If you use the 16 contiguous memory-mapped locations from F700H to F70FH for a peripheral controller, the top of user memory then becomes F700H-140H, or F5C0A.

PL/M MEMCK Call Example

The following example obtains the highest address of contiguous memory available.

```
MEMCK: PROCEDURE ADDRESS EXTERNAL;    /* ENTRY POINT INTO SYSTEM.LIB */
  END MEMCK;

DECLARE MADR ADDRESS;   /* ADDRESS TO CONTAIN VALUE RETURNED BY MEMCK */

MADR = MEMCK;
```

Assembly Language MEMCK Call Example

```
                EXTRN       MEMCK       ;ENTRY POINT INTO SYSTEM.LIB FOR
                                        ;MEMCK
     MADR:      DS          2           ;CONTAINS VALUE RETURNED BY
                                        ;MEMCK
     ;
                CALL        MEMCK
                SHLD        MADR        ;STORE ADDRESS IN MADR
```

Interrupt Processing

Interrupt processing is controlled by logic on the processor board. It provides an eight-level priority interrupt structure using an Interrupt Mask Register (the I-register), and a "current operating level" indicator, which keeps track of the level of interrupt currently serviced. The Interrupt Mask Register is set by a program or from the Console device. You select which interrupts will be acknowledged at any time.

NOTE

Interrupts 0, 1, and 2 are reserved for internal use and must not be referenced by the user.

Priority of Interrupts

There are eight levels of interrupts, numbered 0 through 7. The levels correspond to the eight Interrupt switches and lights on the front panel. Interrupt 0 has the highest priority and interrupt 7, the lowest. An interrupt is not serviced until all higher priority interrupts are serviced. An interrupt of level 4 that is currently being serviced can be interrupted to service an interrupt of level 3, 2, 1, or 0. It cannot be interrupted to service one of level 5, 6, or 7, nor can it interrupted by another level 4.

The Interrupt Mask Register

The Intellec Interrupt Mask Register (I-register) determines which interrupts are accepted by the system. The Interrupt Mask Register contains eight bits, each of which corresponds to an interrupt level:

BITS	7	6	5	4	3	2	1	0
INTERRUPT LEVELS	7	6	5	4	3	2	1	0

A "1" bit in the Interrupt Mask Register prevents the corresponding interrupt from being serviced. A "0" bit allows the interrupt to be serviced. For example, the Intellec Monitor sets the Interrupt Mask Register to 0FEH (11111110B) which blocks all interrupts except interrupt 0.

The Interrupt Mask Register can be set programmatically by writing the desired value to Port 0FCH. For example:

```
MVI   A,0F0H                    (OUTPUT(0FCH)=0F0H  ; in PL/M)
OUT   0FCH
```

sets the Interrupt Mask Register to 11110000B, blocking interrupts 4 through 7 and allowing interrupts 0 through 3.

A program can also read the current value of the Interrupt Mask Register from Port 0FCH. For example:

```
IN    0FCH                      (INPUT$MASK=INPUT (0FCH); in PL/M)
```

places the current value of the Interrupt Mask Register into the A-register (or INPUT$MASK, for the PL/M example).

Interrupt Mask Register Initialization

The Interrupt Mask Register is initialized by the Monitor when the system is turned on and when the RESET button is pressed. At both of these times it is set to 0FEH (11111110B); only interrupt 0 is allowed. If ISIS-II is boot loaded, the mask register is set to 0FCH (11111100B); interrupts 0 and 1 are allowed.

Interrupt Acceptance

When an interrupt occurs, the Interrupt Mask Register is checked to see if an interrupt of that level is permitted. If it is not, no further action is taken, but the interrupt is not cleared and remains pending. If the interrupt is permitted, the "current operating level" is checked to see if another interrupt of equal or higher priority is being serviced. If so, the new interrupt remains pending until the value of the "current operating level" is less than the priority of the new interrupt.

When the new interrupt can be serviced, all Multibus interrupts are locked out, while an RST instruction to the appropriate interrupt address (see the following table) is generated and the "current operating level" is set to the new value. The interrupt lock-out is then removed.

The addresses called when an interrupt is accepted are

INTERRUPT LEVEL	ADDRESS
0	0000H
1	0008H
2	0010H
3	0018H
4	0020H
5	0028H
6	0030H
7	0038H

Interrupt Removal

The program servicing an interrupt must do two things: transmit a signal to the interrupting device, telling it to remove the interrupt signal it generated initially; and restore the "current operating level" maintained by the system. The former action is device-dependent. The latter is accomplished by writing a value of 20H to port 0FDH. *This must be done with interrupts disabled.* (If the code permits another interrupt to be serviced while this is being done, a stack overflow could result.) The following is a sample sequence in assembly language for doing this:

```
MVI         A,20H
OUT         0FDH
POP         PSW         ;RESTORE A-REGISTER AND FLAGS
EI                      ;ENABLE INTERRUPTS
```

The following is a sample PL/M sequence for restoring the current operating level:

```
DISABLE;                    /*DISABLE INTERRUPTS*/
OUTPUT(0FDH) = 20H;         /*RESTORE THE INTERRUPT LOGIC*/
ENABLE;                     /*ALLOW INTERRUPTS*/
```

The following example shows the skeleton of the code necessary to service an interrupt at level 5:

```
            ASEG                    ;VECTOR GOES AT ABSOLUTE
                                    ;LOCATION
            ORG         40          ;RST ADDRESS FOR INTERRUPT 5
            JMP         INT5
            CSEG                    ;PUT CODE IN RELOCATABLE CODE
                                    :SEGMENT
INT5:
            EI                      ;ROUTINE CAN BE INTERRUPTED
            PUSH        PSW         ;SAVE
            PUSH        B           ;   REGISTERS
            PUSH        D           ;
            PUSH        H           ;
            .
            .
            .

    ;<code to service interrupt and remove signal>

            .
            .
            .
```

```
POP     H       ;RESTORE
POP     D       ;    REGISTERS
POP     B       ;
DI              ;CRITICAL SECTION:
                ;                    DISABLE INTERRUPTS
MVI     A,20H   ;RESTORE CURRENT OPERATING LEVEL
OUT     0FDH
POP     PSW     ;RESTORE A REG AND FLAGS
EI              ;PERMIT INTERRUPTS AFTER NEXT
                ; INSTRUCTION
RET             ;THE RETURN MUST IMMEDIATELY
                ;FOLLOW THE EI TO MAKE SURE
                ;IT IS EXECUTED BEFORE ANOTHER
                ;INTERRUPT OCCURS
```

Here is another example, in PL/M, that also discusses issues in ISIS-II which must be confronted by programmers preparing interrupt handlers:

```
INTERRUPT$EXAMPLE;
DO;

/*********************************************************************

  THIS EXAMPLE APPLIES WHEN EXECUTING WITH ISIS-II. IT SHOWS
  HOW AN INTERRUPT SERVICE PROCEDURE IS WRITTEN IN PL/M AND WHAT
  SHOULD BE DONE IN THE MAIN PROGRAM TO PREPARE FOR SERVICING AN
  INTERRUPT.
  THE FOLLOWING CHARACTERISTICS OF ISIS ARE SIGNIFICANT WHEN WRITING
  INTERRUPT PROGRAMS:

     1.  ISIS USES AN INTERNAL (ITS OWN) STACK AND NOT THE
         USER'S STACK FOR ITS OPERATION.

     2.  ISIS DEPENDS ON DISK COMPLETION INTERRUPTS (INTERRUPT 2)
         FOR ITS CORRECT OPERATION.

  THESE CHARACTERISTICS IMPOSE THE FOLLOWING REQUIREMENTS ON A
  USER-WRITTEN INTERRUPT PROGRAM:

     1.  IT MUST SAVE THE ISIS STACK AND LOAD ITS OWN STACK.
         ISIS PROVIDES 10 BYTES OF ITS STACK TO ALLOW THE
         SWAPPING OF STACKS.

     2.  IT MUST RESTORE THE ISIS STACK WHEN ITS PROCESSING IS
         COMPLETE.

     3.  IT MUST SAVE/RESTORE THE STATE OF CPU REGISTERS
         (AUTOMATIC WITH PL/M).

     4.  IT SHOULD NOT USE PERIPHERAL DEVICES WHICH ARE MANAGED
         BY ISIS. CONFLICTS CAN ARISE WHEN BOTH ISIS AND A USER
         PROGRAM ATTEMPT TO COMMUNICATE WITH THE SAME DEVICE SUCH
         AS THE CONSOLE (CRT,TTY). ISIS IS GENERALLY POLLING THE
         DEVICE (E.G., WAITING FOR INPUT), AND THE USER-PROGRAM IN
         THIS CASE IS USING INTERRUPTS WITH THE SAME DEVICE.

     5.  IT MUST LEAVE THE INTERRUPT SYSTEM ENABLED WHEN CALLING
         ISIS ROUTINES WHICH ACCESS THE DISK (E.G., OPEN, READ,
         SEEK, ETC.)

     6.  IT SHOULD USE THE INTERRUPT MASK TO ENABLE/DISABLE THE
         DESIRED INTERRUPT(S).

     7.  IT SHOULD NOT CALL ISIS ROUTINES UNLESS IT IS A CERTAINTY
         THAT NO OTHER PART OF THE ENTIRE USER-PROGRAM CAN BE
         INTERRUPTED IN THE MIDDLE OF USING AN ISIS ROUTINE. ISIS
         IS NOT REENTRANT! IF ANY PART OF THE PROGRAM USES ISIS
         AND CAN BE INTERRUPTED, THEN THIS INTERRUPT PROCEDURE
         MUST NOT CALL ISIS ROUTINES.

*********************************************************************/
```

```
USER$INTERRUPT:
  PROCEDURE INTERRUPT 5 PUBLIC;
  DECLARE ISIS$STACK$PTR ADDRESS;

  ISIS$STACK$PTR = STACK$PTR ;     /* SAVE ISIS STACK */
  STACK$PTR = 0F6C0H ;             /* SETUP USER STACK TO HIGH MEMORY */

/* INSERT USER CODE TO SERVICE THE
     DEVICE GENERATING INTERRUPT 5  */

  STACK$PTR = ISIS$STACK$PTR ;     /* RESTORE ISIS STACK */
  OUTPUT(0FDH) = 020H ;            /* OUTPUT EOI (END OF INTERRUPT)
                                     SIGNAL TO SYSTEM INTERRUPT
                                     CONTROLLER (8259) */


END USER$INTERRUPT;


/***********************************/
/*                                 */
/*     START OF MAIN PROGRAM       */
/*                                 */
/***********************************/


/*  EXAMPLE OF USER SERVICING INTERRUPTS GENERATED ON LEVEL 5 */

OUTPUT (0FCH) = INPUT(0FCH) AND 0DFH ;  /* READ INTERRUPT MASK AND */
                                        /* UNMASK INTERRUPT 5, LEAVING */
                                        /* REMAINING INTERRUPTS UNCHANGED */
ENABLE ;                          /* ENSURE INTERRUPTS ENABLED */

/* INSERT USER CODE TO START THE */
/*   DEVICE THAT GENERATES AN    */
/*   INTERRUPT 5                 */

OUTPUT (0FCH) = INPUT(0FCH) OR 020H ;   /* WHEN INTERRUPTS ARE NOT TO BE    */
                                        /* SERVICED ANYMORE, READ INTERRUPT */
                                        /* MASK AND RESTORE ORIGINAL MASK   */

END ;

EOF
```

APPENDIX A
SERIES III NOTES

1. When a program is invoked under RUN, i.e., under Series III in the 8086-based environment, its extension is presumed to be ".86" if none is provided. If the program name is followed by a period only, the name is used with no extension.

   Examples:

```
                    Program Invoked
RUN MYPROG.ABS      MYPROG.ABS
RUN MYPROG          MYPROG.86
RUN MYPROG.         MYPROG
```

These rules also apply to programs loaded via the console command LOAD.

2. The Series III loader does *not* fully support the following:

   a. *shared* overlays in the LARGE mode of segmentation. The overlays are spread out in memory rather than sharing space as much as possible. See the *iAPX 86, 88 Family Utilities User's Guide for 8086-Based Development Systems*.

   b. *multiple* expanding segments. The intention of the expanding segment is to make available all of the memory remaining after all other segments are successfully located. This is correctly done when the expanding segment is last and the only one of its kind. However, when earlier segments request a maximum-desired length, it is granted. Consequently later expanding memory segments may be allocated less than their maximum-desired space, and will abort the load if their minimum-required specification cannot be met.

APPENDIX B
SERIES III INTERRUPT DISCUSSION

There are three sets of interrupts at work in the Series III operating system:

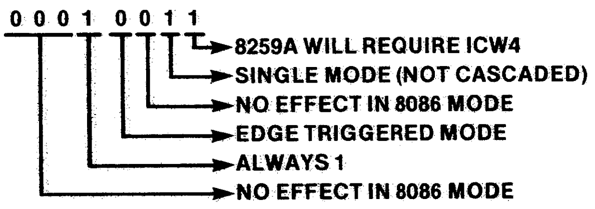
1. The set of interrupts fielded by the operating system, which includes "divide by zero" and "overflow". These are accessible to the user if he writes an exception handler for that purpose and incorporates it into the environment under the operating system by using the DQ$TRAP$EXCEPTION system call.
2. The set of hardware interrupts which interrupt the Series III software. These use the interrupt vectors numbered 56 through 63, located at addresses 224 through 252 decimal, representing levels 0 through 7 of the master interrupt controller. This is an 8259A located on the RPB-86. It is initialized by a built-in sequence as follows:

   a. Output PIC PORT A = 13H; (ICW1)
   b. PIC PORT B = 38H; (ICW2)
   c. PIC PORT B = 0DH; (ICW4)
   d. PIC PORT B = FEH; (interrupt mask)

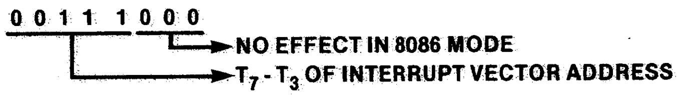
   PIC PORT A is port 0C0H, as discussed in the description of the 8259A in Intel's 1980 Component Data Catalog.

The last instruction is the initial setting of the interrupt mask, enabling interrupt levels 0 and 1 on the 8259A which vector through interrupt vectors 56 and 57 in 8086 memory.

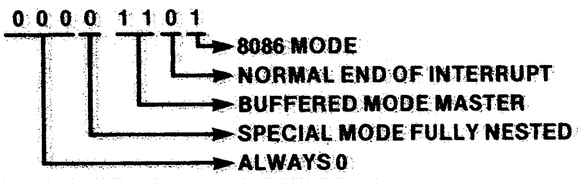
a. **(ICW1)**



b. **(ICW2)**



c. **(ICW4)**



d. The initial interrupt mask is set to 0FCH to allow chip interrupts 0 and 1 to generate interrupts to the RPB-86. The 8253 Programmable timer chip counter 1 is connected to level 0 (corresponding to an interrupt through vector 56) and is used by the operating system.

3. The set of interrupt vectors available for user software include 64 through 183. Under the Series III operating system, users should not use interrupt vectors outside this range except by writing exception handlers as described in item 1 above. To do so is likely to conflict with system software in unpredictable and undesirable ways. Any reprogramming of the 8259A requires EXTREME caution.

APPENDIX C SERIES III EXCEPTION CODES

1. Exceptions Returned from System Calls:

E$OK (0000H)

E$CONTEXT (0101H)

The routine was called in an illegal context. More specifically, this includes an attempt to

- Attach or create a file when 12 connections already existed
- Attach, create, or delete a file to which the console was assigned
- Attach, create, or delete the user file containing overlays
- Attach, create, delete, or rename a file at the Network Manager when the user is not logged on. (Applies to NDS-I Workstation only.)

E$CROSSFS (0102H)

The operation attempted an illegal cross volume rename.

E$EXIST (0103H)

The specified token or connection did not exist, or the specified overlay did not exist.

E$FACCESS (0026H)

A deletion, rename, or destructive creation of a write-protect or format file was attempted; or the mode of open did not agree with the file attributes or device characteristics.

E$FEXIST (0020H)

The specified file exists when it is not expected to exist.

E$FNEXIST (0021H)

The specified file does not exist when it is expected to exist. The deletion, rename, or attachment of a workfile will also cause the exception.

E$MEM (0002H)

Insufficient memory for requested operation.

E$NOPEN (0104H)

The operation attempted to close, read, write, or seek a connection which wasn't opened.

E$OPEN (0105H)

The operation attempted to open a connection which was already opened.

E$OREAD (0106H)

A write operation was attempted on a connection opened for read.

E$OWRITE (0107H)

A read operation was attempted on a connection opened for write.

E$PARAM (0108H)

An argument had an illegal value. This is usually the result of a bounds check (e.g., 0 ≤ connection token < 12).

E$PTR (0109H)

A pointer argument was illegal. If this was the *excep$p* argument, the operating system aborts the job and prints an error message to the cold-start console.

E$SHARE (0028H)

An attempt was made to delete, rename, open, destructively create, or attach to a file on which a connection was already established. At an NDS-I Workstation, this may mean that the file is currently open by another user.

E$SIX (010AH)

An attempt was made to open a seventh connection.

E$SPACE (0029H)

The operation attempted to add a directory entry to a full directory.

E$STRING$BUF (0081H)

The string is over 45 characters in length (DQ$CHANGE$EXTENSION) or the argument is over 80 characters in length (DQ$GET$ARGUMENT).

E$SUPPORT (23H)

One of the following operations was attempted:

- The deletion or rename of a physical or logical device
- The seeking of a physical device or the console
- A DQ$SPECIAL with a connection which was not established on :CI:
- A DQ$SPECIAL with type one or type three when :CI: has been assigned to a disk file.

E$SYNTAX (010CH)

An illegal ISIS pathname was specified. This includes device-name-parts not supported by Series III, or an illegal overlay name.

E$UNSAT (010EH)

The overlay contains unresolved external symbols. Load completed.

E$ADDRESS (010FH)

The overlay loaded contained addresses in the operating system area. The load was not completed.

E$BAD$FILE (0110H)

The file containing the overlay is not a valid object file.

2. Hardware-Detected Conditions

E$ZERO$DIVIDE (8000H)
A divide by zero was attempted.

E$OVERFLOW (8001H)
An overflow occurred.

E$8087 (8007H)
An 8087 error occurred.

APPENDIX D SAMPLE ASSEMBLY LANGUAGE USAGE OF SERIES III SERVICES

Detailed discussions of the language features used below appear in the *8086/8087/8088 Macro Assembly Language Reference Manual for 8086- Based Development Systems*.

```
SOURCE


NAME UDI_EXAMPLES

;THE FOLLOWING ILLUSTRATES THE CALLING CONVENTIONS FOR THE SYSTEMS
;SERVICE ROUTINES. THE PARAMETERS ARE PUSHED ONTO THE STACK
;PRIOR TO THE CALL. THESE EXAMPLES ASSUME THE SMALL MODEL
;OF COMPILATION. FOR LARGE, YOU WOULD CHANGE THE ATTRIBUTES
;OF THE PROCEDURES TO "FAR", AND PRIOR TO THE PUSH OF EACH
;PARAMETER YOU WOULD HAVE TO "PUSH DS".


ASSUME CS:CODE, DS:DATA

DATA SEGMENT PUBLIC

  FILE_NAME  DB   9,'AFILE.SRC'     ;A FILE NAME STRING


  EXCEPT     DW   0                 ;WHERE EXCEPTION CODES ARE RETURNED
  CONN       DW   0                 ;FILE CONNECTION STORED HERE
  ACTUAL     DW   0                 ;NUMBER OF BYTES FROM I/O CALL

DATA ENDS

CODE SEGMENT PUBLIC

  ;THE EXTERNAL DECLARATIONS FOR THE SYSTEMS PROCEDURES

  EXTRN  DQATTACH:NEAR, DQOPEN:NEAR, DQREAD:NEAR, DQCLOSE:NEAR

  ATTACH_FILE  PROC   NEAR

               MOV    AX, OFFSET FILE_NAME  ;GET POINTER TO FILE NAME
               PUSH   AX
               MOV    AX, OFFSET EXCEPT     ;GET POINTER TO EXCEPT
               PUSH   AX
               CALL   DQATTACH              ;CALL SERVICE ROUTINE
               MOV    CONN, AX              ;CONNECTION RETURNED IN AX
               RET

  ATTACH_FILE  ENDP

  ;THE FOLLOWING PROCEDURE HAS TWO PARAMETERS WHICH ARE PASSED
  ;ON THE STACK TO OPEN_FILE. THEY ARE DEFINED BELOW.

  ACCESS       EQU    WORD PTR [BP + 4]
  NUMBUF       EQU    WORD PTR [BP + 6]

  OPEN_FILE    PROC   NEAR


               PUSH   CONN                 ;GET CONNECTION TOKEN
               PUSH   ACCESS               ;GET ACCESS TYPE
               PUSH   NUMBUF               ;GET NUMBER OF BUFFERS
               MOV    AX, OFFSET EXCEPT    ;GET POINTER TO EXCEPT
               PUSH   AX
               CALL   DQOPEN
               RET
  OPEN_FILE     ENDP

  ;THE FOLLOWING PROCEDURE HAS TWO PARAMETERS WHICH ARE PASSED
  ;ON THE STACK TO READ_FILE. THEY ARE DEFINED BELOW.

  BUFFERP       EQU     WORD PTR [BP + 4]
  COUNT         EQU     WORD PTR [BP + 6]

  READ_FILE     PROC    NEAR

                PUSH    CONN                    ;GET CONNECTION TOKEN
                PUSH    BUFFERP                 ;GET POINTER TO BUFFER
                PUSH    COUNT                   ;GET COUNT OF BYTES
                MOV     AX, OFFSET EXCEPT       ;GET POINTER TO EXCEPT
                PUSH    AX
                CALL    DQREAD
                MOV     ACTUAL, AX              ;NUMBER BYTES READ IN AX
                RET

  READ_FILE     ENDP

  CLOSE_FILE    PROC    NEAR

                PUSH    CONN                    ;GET CONNECTION TOKEN
                MOV     AX, OFFSET EXCEPT       ;GET POINTER TO EXCEPT
                PUSH    AX
                CALL    DQCLOSE
                RET

  CLOSE_FILE    ENDP

CODE ENDS

END
```

APPENDIX E
TYPE—A SAMPLE ISIS-II PROGRAM

This appendix describes two programs—one written in PL/M, one written in 8080/8085 Assembly Language, with identical functions. Both programs allow you to type a file to the :CO: device by specifying:

```
TYPE filename
```

rather then

```
COPY filename to :CO:
```

The PL/M program must be compiled and linked with SYSTEM.LIB and PLM80.LIB and located to an actual memory location before it can be executed. The assembly language program must be assembled, linked to SYSTEM.LIB and located to an actual memory location before it can be executed.

PL/M Version of TYPE

```
TYPE:
    DO;
        DECLARE BUFFER(128) BYTE;
        DECLARE ACTUAL$COUNT ADDRESS;
        DECLARE STATUS ADDRESS;
        DECLARE AFT$IN ADDRESS;
        DECLARE READ$ACCESS LITERALLY '1';

OPEN:
    PROCEDURE (AFT,FILE,ACCESS,MODE,STATUS) EXTERNAL;
        DECLARE (AFT,FILE,ACCESS,MODE,STATUS) ADDRESS;
    END OPEN;

CLOSE:
    PROCEDURE (AFT,STATUS) EXTERNAL;
        DECLARE (AFT,STATUS) ADDRESS;
    END CLOSE;

READ:
    PROCEDURE (AFT,BUFFER,COUNT,ACTUAL,STATUS) EXTERNAL;
        DECLARE (AFT,BUFFER,COUNT,ACTUAL,STATUS) ADDRESS;
    END READ;

WRITE:
    PROCEDURE (AFT,BUFFER,COUNT,STATUS) EXTERNAL;
        DECLARE (AFT,BUFFER,COUNT,STATUS) ADDRESS;
    END WRITE;

EXIT:
    PROCEDURE EXTERNAL;
    END EXIT;

ERROR:
    PROCEDURE (ERRNUM) EXTERNAL;
        DECLARE (ERRNUM) ADDRESS;
    END ERROR;
    /*
      READ THE CONSOLE FILE TO GET THE PARAMETER STRING.
      FOR THIS EXAMPLE, THE COMMAND ENTERED IS

          TYPE   ASM.LST(CR)(LF)

      AT THIS POINT, THE CONSOLE INPUT BUFFER CONTAINS

          ASM.LST(CR)(LF)

    */
    CALL READ(1,.BUFFER,128,.ACTUAL$COUNT,.STATUS);
    CALL OPEN(.AFT$IN,.BUFFER,READ$ACCESS,0,.STATUS);
    IF STATUS > 0 THEN CALL ERROR(STATUS);
    /*
      THE FILE ASM.LST IS NOW OPEN FOR INPUT.
    */
    ACTUAL$COUNT = 1;
    DO WHILE ACTUAL$COUNT < > 0;
      CALL READ(.AFT$IN, .BUFFER, 128, .ACTUAL$COUNT, .STATUS);
      IF STATUS > 0 THEN CALL ERROR(STATUS);
      CALL WRITE(0, .BUFFER, ACTUAL$COUNT, .STATUS);
      IF STATUS > 0 THEN CALL ERROR(STATUS);
    END;

    CALL WRITE (0,.('COPY COMPLETED', 0DH, 0AH), 16, .STATUS)
    CALL CLOSE(AFT$IN, .STATUS);
    IF STATUS > 0 THEN CALL ERROR(STATUS);
    CALL EXIT;

END;
```

8080/8085 Assembly Language Version of TYPE

```
;           Sample Program
;
OPEN        EQU         0
CLOSE       EQU         1
READ        EQU         3
WRITE       EQU         4
EXIT        EQU         9
ERROR       EQU         12
;
            EXTRN       ISIS
;
            CSEG                    ;BEGINNING OF CODE SEGMENT
BEGIN:
            LXI         SP,STCKA+4
            MVI         C,READ              ; READ THE CONSOLE
            LXI         D,RBLK
            CALL        ISIS
            LDA         STATUS
            ORA         A
            JNZ         ERR
```

```
;
        MVI     C,OPEN          ; OPEN THE INPUT FILE
        LXI     D,OBLK
        CALL    ISIS
        LDA     STATUS
        ORA     A
        JNZ     ERR
        LHLD    AFT
        SHLD    CAFT
;
LOOP:
        MVI     C,READ          ; READ THE INPUT FILE
        LXI     D,RBLK
        CALL    ISIS
        LDA     STATUS
        ORA     A
        JNZ     ERR
        LHLD    ACTUAL
        MOV     A,H
        ORA     L
        JZ      DONE
        MVI     C,WRITE         ; WRITE TO THE CONSOLE
        LXI     D,WBLK
        CALL    ISIS
        LDA     STATUS
        ORA     A
        JNZ     ERR
        JMP     LOOP
DONE:
        MVI     C,CLOSE         ; CLOSE THE INPUT FILE
        LXI     D,CBLK
        CALL    ISIS
        MVI     C,EXIT          ; NORMAL EXIT
        LXI     D,XBLK
        CALL    ISIS
;
ERR:
        MVI     C,ERROR         ; ERROR MESSAGE
        LXI     D,EBLK
        CALL    ISIS
        MVI     C,EXIT          ; ERROR EXIT
        LXI     D,XBLK
        CALL    ISIS
;
        DSEG            ; BEGINNING OF DATA SEGMENT
OBLK:
        DW      AFT
        DW      BUFFER
        DW      1               ; READ ACCESS
        DW      0               ; NO ECHO
        DW      STATUS
;
CBLK:
CAFT:   DS      2
        DW      STATUS
```

```
;
RBLK:
AFT:        DW        1
            DW        BUFFER
            DW        128
            DW        ACTUAL
            DW        STATUS
;
WBLK:
            DW        0
            DW        BUFFER
ACTUAL:     DS        2
            DW        STATUS
;
XBLK:
            DW        STATUS
;
EBLK:
STATUS:     DS        2
            DW        STATUS
;
BUFFER:     DS        128
;
STCKA:      DS        4
;
END         BEGIN
```

APPENDIX F
ASCII CODES

Table F-1. ASCII Code List

Decimal	Octal	Hexadecimal	Character
0	000	00	NUL
1	001	01	SOH
2	002	02	STX
3	003	03	ETX
4	004	04	EOT
5	005	05	ENQ
6	006	06	ACK
7	007	07	BEL
8	010	08	BS
9	011	09	HT
10	012	0A	LF
11	013	0B	VT
12	014	0C	FF
13	015	0D	CR
14	016	0E	SO
15	017	0F	SI
16	020	10	DLE
17	021	11	DC1
18	022	12	DC2
19	023	13	DC3
20	024	14	DC4
21	025	15	NAK
22	026	16	SYN
23	027	17	ETB
24	030	18	CAN
25	031	19	EM
26	032	1A	SUB
27	033	1B	ESC
28	034	1C	FS
29	035	1D	GS
30	036	1E	RS
31	037	1F	US
32	040	20	SP
33	041	21	!
34	042	22	‘‘
35	043	23	#
36	044	24	$
37	045	25	%
38	046	26	&
39	047	27	‘
40	050	28	(
41	051	29	)
42	052	2A	*
43	053	2B	+
44	054	2C	,
45	055	2D	–
46	056	2E	.
47	057	2F	/
48	060	30	0
49	061	31	1
50	062	32	2
51	063	33	3
52	064	34	4
53	065	35	5
54	066	36	6
55	067	37	7
56	070	38	8
57	071	39	9
58	072	3A	:
59	073	3B	;
60	074	3C	<

Table F-1. ASCII Code List (Cont'd.)

Decimal	Octal	Hexadecimal	Character
61	075	3D	=
62	076	3E	>
63	077	3F	?
64	100	40	@
65	101	41	A
66	102	42	B
67	103	43	C
68	104	44	D
69	105	45	E
70	106	46	F
71	107	47	G
72	110	48	H
73	111	49	I
74	112	4A	J
75	113	4B	K
76	114	4C	L
77	115	4D	M
78	116	4E	N
79	117	4F	O
80	120	50	P
81	121	51	Q
82	122	52	R
83	123	53	S
84	124	54	T
85	125	55	U
86	126	56	V
87	127	57	W
88	130	58	X
89	131	59	Y
90	132	5A	Z
91	133	5B	[
92	134	5C	\
93	135	5D	]
94	136	5E	∧
95	137	5F	—
96	140	60	'
97	141	61	a
98	142	62	b
99	143	63	c
100	144	64	d
101	145	65	e
102	146	66	f
103	147	67	g
104	150	68	h
105	151	69	i
106	152	6A	j
107	153	6B	k
108	154	6C	l
109	155	6D	m
110	156	6E	n
111	157	6F	o
112	160	70	p
113	161	71	q
114	162	72	r
115	163	73	s
116	164	74	t
117	165	75	u
118	166	76	v
119	167	77	w
120	170	78	x
121	171	79	y
122	172	7A	z
123	173	7B	{
124	174	7C	|
125	175	7D	}
126	176	7E	~
127	177	7F	DEL

Table F-2. ASCII Code Definition

Abbreviation	Meaning	Decimal Code
NUL	NULL Character	0
SOH	Start of Heading	1
STX	Start of Text	2
ETX	End of Text	3
EOT	End of Transmission	4
ENQ	Enquiry	5
ACK	Acknowledge	6
BEL	Bell	7
BS	Backspace	8
HT	Horizontal Tabulation	9
LF	Line Feed	10
VT	Vertical Tabulation	11
FF	Form Feed	12
CR	Carriage Return	13
SO	Shift Out	14
SI	Shift In	15
DLE	Data Link Escape	16
DC1	Device Control 1	17
DC2	Device Control 2	18
DC3	Device Control 3	19
DC4	Device Control 4	20
NAK	Negative Acknowledge	21
SYN	Synchronous Idle	22
ETB	End of Transmission Block	23
CAN	Cancel	24
EM	End of Medium	25
SUB	Substitute	26
ESC	Escape	27
FS	File Separator	28
GS	Group Separator	29
RS	Record Separator	30
US	Unit Separator	31
SP	Space	32
DEL	Delete	127

APPENDIX G ISIS-II ERROR CODES

The following manual is referred to in this appendix as "reference 1":

Intellec Series III Microcomputer Development System Console Operating Instructions

This appendix provides a listing of error codes and/or messages issued by ISIS-II. Reference 1 provides such a listing for RUN, DEBUG-86, and some nonresident system routines. Other nonresident system routine error messages, such as those for link and locate, are listed in the *iAPX Family Utilities User's Guide for 8086-Based Development Systems* and the *MCS-80/85 Utilities User's Guide for 8080/8085-Based Development Systems*.

Error message numbers are allocated as follows:

- 1-99 inclusive — ISIS-II resident routines (8080/8085 mode)
- 100-119 inclusive — RUN command (8086 mode)
- 120-199 inclusive — DEBUG-86 (8086 mode)
- 200-255 inclusive — nonresident system routines (8080/8085 mode)

ISIS-II Error Routines (8080/8085 Mode)

Errors encountered by ISIS-II are either fatal or nonfatal. In the following lists fatal errors are noted as such. The other errors are generally nonfatal unless they are issued by the CONSOL system call (see tables G-1 and G-2).

A nonfatal error immediately halts processing and permits your program to take a recovery path of your choosing. The error number is returned to your program.

If an error occurs when you are entering a console command, the error is echoed followed by an error message. For example, the following input results in the error message shown:

```
-COPY :PR:CREDIT TO :F1:<CR>
:PR:CREDIT, UNRECOGNIZED DEVICE NAME
```

A fatal error immediately halts processing but does not permit recovery. Control returns to ISIS-II which overlays some user program area with nonresident ISIS-II files, and displays the following error message:

```
ERROR nnn USER PC mmmm
```

where nnn is the error number and mmmm is the contents of the program counter when the error occurred.

In general, after displaying an error message, the system displays the ISIS-II prompt character (a hyphen) and waits for you to enter the corrected input.

The action taken in response to fatal errors depends on the setting of an internal system switch called the debug toggle. That switch indicates whether control is to return to ISIS-II (debug=0) or the Monitor (debug=1) when an error occurs.

Table G-1. Nonfatal Error Numbers Returned by System Calls

OPEN	3, 4, 5, 9, 12, 13, 14, 22, 23, 25, 28.
READ	2, 8.
WRITE	2, 6.
SEEK	2, 19, 20, 27, 31, 35.
RESCAN	2, 21.
CLOSE	2.
DELETE	4, 5, 13, 14, 17, 23, 28, 32.
RENAME	4, 5, 10, 11, 13, 17, 23, 28.
ATTRIB	4, 5, 13, 23, 26, 28.
CONSOL	None; all errors are fatal.
WHOCON	None.
ERROR	None.
LOAD	3, 4, 5, 12, 13, 22, 23, 28, 34.
EXIT	None.
SPATH	4, 5, 23, 28.

Table G-2. Fatal Errors Issued by System Calls

OPEN	1, 7, 24, 30, 33.
READ	24, 30, 33.
WRITE	7, 24, 30, 33.
SEEK	7, 24, 30, 33.
RESCAN	33.
CLOSE	33.
DELETE	1, 24, 30, 33.
RENAME	1, 24, 30, 33.
ATTRIB	1, 24, 30, 33.
CONSOL	1, 4, 5, 12, 13, 14, 22, 23, 24, 28, 30, 33.
WHOCON	33.
ERROR	33.
LOAD	1, 15, 16, 24, 30, 33.
SPATH	33.

Any of the following actions sets the debug toggle to one and transfers control to the Monitor:

- Pressing interrupt switch 0 while a program is running.
- Executing program load with the DEBUG switch specified in the command line.
- Executing a LOAD system call with a transfer value of 2.

Any of the following actions sets the debug toggle to zero, performs the operation listed, then transfers control to ISIS-II:

- Pressing interrupt switch 1 while a program is running. This action terminates processing.
- Executing an EXIT system call. This action terminates a program.
- Executing a LOAD system call with a transfer value of 1. This action loads an absolute object file.
- Executing a Monitor G8 command. This action exits the Monitor.

If the debug toggle is zero when a fatal error occurs, the following occur:

- All open files are closed in their current state, including :CI: and :CO:.
- The initial system console device is opened as :CI: and :CO:.
- A fresh copy of ISIS-II is read in from the disk, and ISIS-II prompts for a command with a hyphen (-).

If the debug toggle is set to one when a fatal error occurs, the following occur:

- All open files are left open.
- Control passes to the Monitor.
- Monitor prompts for a command with a period (.).

At this point Monitor commands can be used to examine registers and memory to try to determine the cause of the error. However, the program should not be restarted with a simple Monitor G command, because the ISIS-II restart address has not been saved. DO NOT RESET THE SYSTEM AT THIS POINT. A G8 command should be used instead so all files are closed. Rebooting does not close files.

NOTE

Although programs cannot be loaded in the ISIS-II area, the ISIS-II area is not protected from a running program. If a program should happen to destroy parts of ISIS-II, subsequent system calls may not operate correctly and input/output may destroy areas on your disk. This would happen mainly when an undebugged program is running. ISIS-II can always be restored by bootstrapping from a good system disk.

ISIS-II error messages codes are:

1. Fatal error. The memory area from 3000H to program origin is used for input/output buffers. Too few buffers were allocated to meet the current request in addition to earlier requests.
2. Illegal AFTN argument. The number supplied as an AFTN (active file table number) is inappropriate. Perhaps your program closed a file prematurely and then tried to read it.
3. Fatal error. AFT (Active File Table) is full. At most, six files may be active at one time. You must close one of your open files before a file can successfully be opened.
4. Incorrectly specified filename. You have possibly entered too many characters for filename, as in OLDFILE.1 (the maximum is six characters before the period, three after). Filename conventions are described in Chapter 3 of Reference 1.
5. Unrecognized device name. You have entered an incorrect device name, as in :PR: for the line printer :LP:. Check the device names in Chapter 3 of Reference 1.
6. Attempt to write to input device. An attempt has been made to write to an input device. You can only write to an output device, such as a line printer (:LP:). See Chapter 3 of Reference 1 for information on devices.
7. Fatal error. The disk is full. Check that you have specified the intended disk.
8. Attempt to read from output device. Some devices, like the line printer (:LP:), are output only and cannot be read. The current operation either should not be a READ or needs to use a different device name. See Chapter 3 of Reference 1 for devices.
9. Disk directory is full. There is no room on the target disk's directory to add an additional filename. The limit is 200 entries for flexible disks and 992 entries for hard platters.
10. Pathname is not on same disk. A system call was attempted (RENAME) that requires two pathnames on the same device but the specified pathnames did not specify the same device.
11. File already exists. A filename identical to the one just used was found. Perhaps a different drive was intended, or a different spelling of the filename.

12. File is already open. Only console input (:CI:) and console output (:CO:) may be opened multiple times. If the spelling of the filename is correct, a flaw may exist in the program logic. For example, an earlier module may be using the file too soon or there may be an unintended loop.
13. No such file. The specified filename could not be found in the directory on the disk in the drive indicated by your command. A different drive or disk may have the file. For example, a console request to load a RUN file with a default extension of .86.
14. Write-protected file encountered. The intended operation (e.g., WRITE, RENAME, DELETE) could not be done because the specified file has the write-protect or format attribute set.
15. Fatal error. ISIS overwrite. The system detected an attempt to write into the area reserved for the ISIS resident files, i.e., below 3000H. Such an operation would create unpredictable results and is disallowed.
16. Fatal error. Bad load format. This error was possibly caused by a source-language file. Files to be loaded for 8080/8085 execution must be in absolute object module format.
17. Not a disk file. An attempt was made to reference a disk file on a wrong device type, with an improper pathname, such as :HP:FILE2 instead of :Fn:FILE2. File accessing conventions are described in Chapter 3 of Reference 1.
18. Illegal ISIS commands. This error results when an ISIS system call is made with an illegal command number. For RUN programs, the version number of RUN may not be compatible with the version number of ISIS.
19. Attempted seek on non-disk file. Seeks on physical devices other than disk drives are invalid (:BB: is an exception and is valid).
20. Attempted back seek too far. The seek attempted to go beyond the beginning of the file; MARKER is set to zero.
21. Can't rescan. The file was not opened for line-editing.
22. Illegal access mode to open. Only 1, 2, and 3 are valid, meaning input (read), output (write), or update (both read and write).
23. Missing filename. The system expected a filename, but one was not supplied.
24. Fatal error. Disk input/output hardware error. When error number 24 occurs, an additional message is displayed:

```
STATUS=00nn
D=x T=yyy S=zzz
```

where x represents the drive number, yyy the track address, zzz the sector address, and where nn has the following meanings:

For flexible disks:

01	Deleted record
02	Data field CRC error
03	Invalid address mark
04	Seek error
08	Address error
0A	ID field CRC error
0E	No address mark
0F	Incorrect data address mark
10	Data overrun or data underrun
20	Attempt to write on Write Protect
40	Drive has indicated a Write error
80	Drive not ready

For hard disks:

01	ID field miscompare
02	Data field CRC error
04	Seek error
08	Bad sector address
0A	ID field CRC error
0B	Protocol violations
0C	Bad track address
0E	No ID address mark or sector not found
0F	Bad data field address mark
10	Format error
20	Attempt to write on write-protected drive
40	Drive has indicated a write error
80	Drive not ready

25. Illegal echo file. An echo file must have an active file table number (AFTN) between 0 an 255, and must already be opened for output. Check that these conditions are met.
26. Illegal attribute identifier. This error refers to the second parameter to the ATTRIB system call routine. Check that you have specified a valid parameter. Only 0, 1, 2, or 3 is valid, meaning the invisible, system, write-protect, or format attributes, respectively.
27. Illegal seek command. An unsupported mode for the specified device was used in a seek command.
28. Missing extension. An expected file extension was not supplied.
29. Fatal error. Premature EOF. An unexpected end of file was encountered from the console.
30. Fatal error. Drive specified was not ready.
31. Can't seek on write only file. Seeks can be executed only on read or update files.
32. Can't delete open file. You need to close the file before attempting to delete it. Verify the pathname.
33. Fatal error. Illegal system call parameter. A parameter was specified in a system call which is meant to be used as a pointer to a memory area intended for the receipt of data; however, ISIS found that this pointer was pointing to the memory space which ISIS occupies. ISIS will not allow a user to write into its memory space.
34. Fatal error. The return switch in a LOAD system call was not 0, 1 or 2, the only valid values.
35. Seek past EOF. An attempt was made to extend a file opened for input by seeking past end-of-file.

APPENDIX H OBJECT MODULE RELOCATION AND LINKAGE

Development projects which have libraries of checked-out modules will naturally prefer to avoid retranslating their source, i.e., recompiling or reassembling each such module.

Figure H-1 presents the valid possibilities for combining object modules created by resident and cross-product translators or relocation-and-linkage packages. ("Cross-product" here means software packages that execute on an 8080/8085-based system but create code to run on an 8086-based system.)

As long as LINK and LOCATE are used in sequence, object or absolute modules developed using cross-product translators can be combined with new object modules developed using resident translators. The following text identifies the permitted and prohibited possibilities:

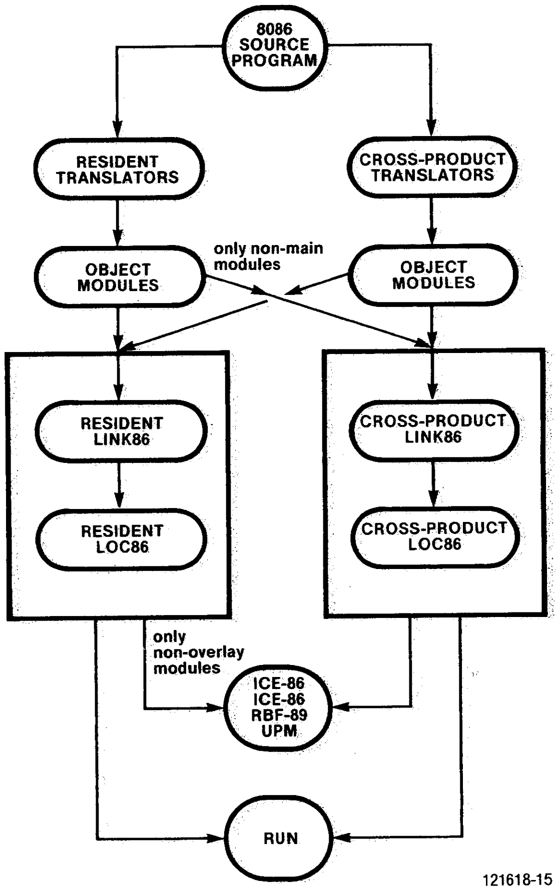
PERMITTED

1. The resident R&L86 will successfully process any object or absolute module produced by any Intel cross-product or resident product.
2. The cross-product R&L package will successfully process any non-main, object module produced by resident translators.
3. Current absolute loaders (e.g., those named in figure H-1) will successfully process non-overlay modules produced by resident products as well as the output of cross-products.
4. Modules which are Position-Independent Code (PIC) or Load-Time Locatable (LTL) do not need LOC86 and can be RUN directly after being processed by LINK86.

PROHIBITED

1. The cross-product R&L package cannot process:
   a. Main object modules from resident translators.
   b. Modules produced by resident R&L86.
2. Current absolute loaders cannot process overlay modules produced by resident R&L86.

See the *iAPX 86, 88 Family Utilities User's Guide* for complete details on Intel's R&L packages.



Use of Relocation and Linkage Packages

APPENDIX I
TABLES OF PARAMETERS AND SYSTEM SERVICE ROUTINES

Although the discussion of each routine includes explanation of the parameters, table I-1 lists them in alphabetic order for study or reference. In many cases the same parameter name is used in the discussions of both operating systems. For this reason, the routine names appear only once, without DQ$, e.g., OPEN, SEEK rather than DQ$OPEN, DQ$SEEK, OPEN, SEEK.

Table I-2 repeats the alphabetical table from Chapter 1 as an index to the descriptions of the Series III operating system service routines, which appear in Chapter 2. The external PL/M-86 declarations for these routines follow table I-2 in alphabetical order.

The external PL/M-86 declarations for the ISIS-II operating system service routines follow table I-3 in alphabetical order.

Table I-1. Alphabetical Parameter Definitions

Parameter Name	Routines Using This Parameter and Brief Definition of Parameter
(2) access	OPEN, SEEK A number telling how you plan to use the file, e.g., read, or write, or both
arg$p	GET$ARGUMENT Pointer to the 81-byte area you have declared to receive the argument from a line-edited input source
block$no	SEEK (ISIS-II only) A number telling what (or how many) block of a disk file are being referred to
(1) buf$p	READ, WRITE, SWITCH$BUFFER Pointer to the area you have declared for reading from (or writing to) a file; for read or write, it should be at least COUNT bytes long or unintended results will occur
byte$no	SEEK (ISIS-II only) A number telling what (or how many) bytes of a disk file block are being referred to
compl$cod	EXIT A word telling the success of program completion and termination
(1) conn	DETACH, GET$CONNECTION$STATUS, OPEN, SEEK, READ, WRITE, TRUNCATE, CLOSE, SPECIAL Connection to a file or device, established earlier via attach or create
(1) count	READ, WRITE The number of bytes you want read or written
delim	GET$ARGUMENT A byte filled by the routine with the delimiter ending the current argument

Table I-1. Alphabetical Parameter Definitions (Cont'd.)

Parameter Name	Routines Using This Parameter and Brief Definition of Parameter
dt$p	GET$TIME Pointer to the structure you set up for date and time
(1) excep$p	ALL ROUTINES BUT EXIT Pointer to the word you have declared to receive the exception value
exception$cod	DECODE$EXCEPTION A word into which you have placed an exception code
exception$p	DECODE$EXCEPTION Pointer to the 81-byte area you have declared for receiving the formatted message corresponding to excep$cod
extension$p	CHANGE$EXTENSION Pointer to the extension as you wish it to be
file$ptr$low or high	Same as low$offset, high$offset
handler$p	TRAP$EXCEPTION, GET$EXCEPTION$HANDLER, TRAP$CC Pointer to the entry-point of your routine to handle current exceptions (or Control C)
high$offset	SEEK, GET$CONNECTION$STATUS Most significant 2-byts of 4-byte integer whose value is the position of the file pointer, i.e., the number of bytes from the beginning of the file
id$p	GET$SYSTEM$ID Pointer to the location where you want the system-name or sign-on put
(2) info$p	GET$CONNECTION$STATUS, SPATH (ISIS-II) Pointer to the pathname whose connection you need to know
low$offset	SEEK, GET$CONNECTION$STATUS Least significant 2-bytes of 4-byte integer position of file pointer—see high$offset
mode	SEEK Value representing direction and type of seek operation
(2) name$p	OVERLAY Pointer to the pathname of the overlay to be loaded next
(1) new$p	RENAME Pointer to the pathname as you wish to have it
(1) num$buf	OPEN Number of buffers to be used for I/O to file being opened
(1) old$p	RENAME Pointer to the pathname as it is now
(2) path$p	CREATE, DELETE, ATTACH, CHANGE$EXTENSION Pointer to the pathname you wish to use

Table I-1. Alphabetical Parameter Definitions (Cont'd.)

Parameter Name	Routines Using This Parameter and Brief Definition of Parameter
segbase	FREE, GET$SIZE The word containing the base of a block of bytes
size	ALLOCATE The number of bytes you want to use
status$p	ISIS-II Synonym for excep$p
type	SPECIAL A value determining whether console input should be line-edited or transparent

(1) same parameter used with same meaning by ISIS-II routines

(2) same parameter name discussed in Chapter 3 on ISIS-II, but full meaning includes differences, discussed under the routines where it is used

Table I-2. Alphabetical List of Series III Service Routines

```
ALLOCATE
ATTACH
CHANGE$EXTENSION
CLOSE
CREATE
DECODE$EXCEPTION
DELETE
DETACH
EXIT
FREE
GET$ARGUMENT
GET$CONNECTION$STATUS
GET$EXCEPTION$HANDLER
GET$SIZE
GET$SYSTEM$ID
GET$TIME
OPEN
OVERLAY
READ
RENAME
SEEK
SPECIAL
SWITCH$BUFFER
TRAP$CC
TRAP$EXCEPTION
TRUNCATE
WRITE
```

```
DQ$ALLOCATE:   PROCEDURE (size, excep$p) TOKEN EXTERNAL;
       DECLARE size       WORD,
               excep$p    POINTER ;
       END ;
```

```
DQ$ATTACH:  PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
       DECLARE path$p  POINTER,
               excep$p POINTER ;
       END ;
```

Table I-2. Alphabetical List of Series III Service Routines (Cont'd.)

```
DQ$CHANGE$EXTENSION:  PROCEDURE (path$p, extension$p, excep$p) EXTERNAL;
        DECLARE path$p        POINTER,
                extension$p   POINTER,
                excep$p       POINTER ;
        END ;


DQ$CLOSE:  PROCEDURE (conn, excep$p) EXTERNAL;
        DECLARE conn      CONNECTION,
                excep$p   POINTER ;
        END ;


DQ$CREATE:  PROCEDURE (path$p, excep$p) CONNECTION EXTERNAL;
        DECLARE path$p    POINTER,
                excep$p   POINTER ;
        END ;


DQ$DECODE$EXCEPTION:  PROCEDURE (exception$code, message$p, excep$p) EXTERNAL;
        DECLARE exception$code WORD,
                message$p      POINTER,
                excep$p        POINTER ;
        END ;


DQ$DELETE:  PROCEDURE (path$p, excep$p) EXTERNAL;
        DECLARE path$p    POINTER,
                excep$p   POINTER ;
        END ;


DQ$DETACH:  PROCEDURE (conn, excep$p) EXTERNAL;
        DECLARE conn      CONNECTION,
                excep$p   POINTER ;
        END ;


DQ$EXIT:  PROCEDURE (completion$code) EXTERNAL;
        DECLARE completion$code WORD ;
        END ;


DQ$FREE:  PROCEDURE (segment, excep$p) EXTERNAL;
        DECLARE segment     TOKEN,
                excep$p     POINTER ;
        END ;


DQ$GET$ARGUMENT:  PROCEDURE (argument$p, excep$p) BYTE EXTERNAL;
        DECLARE argument$p    POINTER,
                excep$p       POINTER ;
        END ;
```

Table I-2. Alphabetical List of Series III Service Routines (Cont'd.)

```
DQ$GET$CONNECTION$STATUS:  PROCEDURE (conn, info$p, excep$p) EXTERNAL;
        DECLARE conn     CONNECTION,
                info$p   POINTER,
                excep$p  POINTER ;
        END ;


DQ$GET$EXCEPTION$HANDLER:  PROCEDURE (handler$p, excep$p) EXTERNAL;
        DECLARE handler$p  POINTER,
                excep$p    POINTER ;
        END ;


DQ$GET$SIZE:  PROCEDURE (segbase, excep$p) WORD EXTERNAL;
        DECLARE segbase     TOKEN,
                excep$p     POINTER ;
        END ;


DQ$GET$SYSTEM$ID:  PROCEDURE (id$p, excep$p) EXTERNAL;
        DECLARE id$p     POINTER,
                excep$p  POINTER ;
        END ;


DQ$GET$TIME:  PROCEDURE (dt$p, excep$p) EXTERNAL;
        DECLARE dt$p     POINTER,
                excep$p  POINTER ;
        END ;


DQ$OPEN:  PROCEDURE (conn, access, num$buf, excep$p) EXTERNAL;
        DECLARE conn     CONNECTION,
                access   BYTE,
                num$buf  BYTE,
                excep$p  POINTER ;
        END ;


DQ$OVERLAY:  PROCEDURE (name$p, excep$p) EXTERNAL;
        DECLARE name$p   POINTER,
                excep$p  POINTER ;
        END ;


DQ$READ:  PROCEDURE (conn, buf$p, count, excep$p) WORD EXTERNAL;
        DECLARE conn     CONNECTION,
                buf$p    POINTER,
                count    WORD,
                excep$p  POINTER ;
        END ;


DQ$RENAME:  PROCEDURE (old$p, new$p excep$p) EXTERNAL;
        DECLARE old$p    POINTER,
                new$p    POINTER,
                excep$p  POINTER ;
        END ;
```

Table I-2. Alphabetical List of Series III Service Routines (Cont'd.)

```
DQ$SEEK:  PROCEDURE (conn, mode, high$offset, low$offset, excep$p) EXTERNAL;
       DECLARE conn        CONNECTION
               mode        BYTE,
               low$offset  WORD,
               high$offset WORD,
               excep$p     POINTER ;
       END ;


DQ$SPECIAL:  PROCEDURE (type, parameter$p, excep$p) EXTERNAL;
       DECLARE type           BYTE,
               parameter$p    POINTER,
               excep$p        POINTER ;
       END ;


DQ$SWITCH$BUFFER:  PROCEDURE (buffer$p, excep$p) WORD EXTERNAL;
       DECLARE buffer$p  POINTER,
               excep$p   POINTER ;
       END ;


DQ$TRAP$CC:  PROCEDURE (handler$p, excep$p) EXTERNAL;
       DECLARE handler$p POINTER,
               excep$p   POINTER ;
       END ;


DQ$TRAP$EXCEPTION:  PROCEDURE (handler$p, excep$p) EXTERNAL;
       DECLARE handler$p POINTER,
               excep$p   POINTER ;
       END ;


DQ$TRUNCATE:  PROCEDURE (conn, excep$p) EXTERNAL;
       DECLARE conn      WORD,
               excep$p   POINTER ;
       END ;


DQ$WRITE:  PROCEDURE (conn, buf$p, count, excep$p) EXTERNAL;
       DECLARE conn      CONNECTION,
               buf$p     POINTER,
               count     WORD,
               excep$p   POINTER ;
       END ;
```

Table I-3. Index from Routine to Discussion

ISIS-II and MONITOR Routines	Discussed in Group Named
ATTRIB	Disk Directory Maintenance
CI	(M) Console Input
CLOSE	File Management
CO	(M) Console Output
CONSOL	Console Control
CSTS	(M) Console Input Status
DELETE	Disk Directory Maintenance
ERROR	Console Control
EXIT	Execution Control
IOCHK	(M) Configuration Check
IODEF	(M) User-Defined Devices
IOSET	(M) Configuration Set
LO	(M) List Output
LOAD	Execution Control
MEMCK	(M) RAM Size Check
OPEN	File Management
PO	(M) Punch Output
READ	File Management
RENAME	Disk Directory Maintenance
RESCAN	File Management
RI	(M) Reader Input
SEEK	File Management
SPATH	File Management
UI	(M) UPP Input
UO	(M) UPP Output
UPPS	(M) UP Status
WHOCON	Console Control
WRITE	File Management

```
ATTRIB:
  PROCEDURE (path$p, atrb, onoff, status$p) EXTERNAL;
     DECLARE (path$p, atrb, onoff, status$p) ADDRESS;
  END ATTRIB;


CI: PROCEDURE BYTE EXTERNAL;              /*ENTRY POINT INTO SYSTEM.LIB*/
  END CI;


CLOSE:
  PROCEDURE (conn, status$p) EXTERNAL;
     DECLARE (conn, status$p) ADDRESS;
  END CLOSE;


CO: PROCEDURE (CHAR) EXTERNAL;            /*ENTRY POINT INTO SYSTEM.LIB*/
  DECLARE CHAR BYTE;
  END CO;


CONSOL:
  PROCEDURE (ci$path$p, co$path$p, status$p) EXTERNAL;
     DECLARE (ci$path$p, co$path$p, status$p) ADDRESS;
  END CONSOLE;


CSTS: PROCEDURE BYTE EXTERNAL;            /* ENTRY POINT INTO SYSTEM.LIB FOR */
                                          /* CSTS */
  END CSTS;
```

Table I-3. Index from Routine to Discussion (Cont'd.)

```
DELETE:
  PROCEDURE (path$p, status$p) EXTERNAL;
      DECLARE (path$p, status$p) ADDRESS;
  END DELETE;


ERROR:
  PROCEDURE (errnum) EXTERNAL;
      DECLARE (errnum) ADDRESS;
  END ERROR;


EXIT:
  PROCEDURE EXTERNAL;
  END EXIT;


IOCHK: PROCEDURE BYTE EXTERNAL;        /* ENTRY POINT INTO SYSTEM.LIB */
  END IOCHK;


IODEF: PROCEDURE(driver$code, entry$point) EXTERNAL;
  DECLARE driver$code BYTE;
  DECLARE driver$entry$point ADDRESS;
  END IODEF;


IOSET: PROCEDURE(CONFIG) EXTERNAL;     /* ENTRY POINT INTO SYSTEM.LIB */
                                       /* FOR IOSET */
  DECLARE CONFIG BYTE;
  END IOSET;


LO: PROCEDURE(BUFF) EXTERNAL;          /* ENTRY POINT INTO SYSTEM.LIB */
  DECLARE CHAR BYTE;
  END LO;


LOAD:
  PROCEDURE (path$p, load$offset, control$sw, entry$p, status$p) EXTERNAL;
      DECLARE (path$p, load$offset, control$sw, entry$p, status$p) ADDRESS;
  END LOAD;


MEMCK: PROCEDURE ADDRESS EXTERNAL;    /* ENTRY POINT INTO SYSTEM.LIB */
  END MEMCK;


OPEN:
  PROCEDURE (conn$p, path$p, access, echo, status$p) EXTERNAL;
    DECLARE (conn$p, path$p, access, echo, status$p) ADDRESS;
  END OPEN;


PO: PROCEDURE (CHAR) EXTERNAL;         /* ENTRY POINT INTO SYSTEM.LIB*/
  DECLARE CHAR BYTE;
  END PO;
```

Table I-3. Index from Routine to Discussion (Cont'd.)

```
READ:
    PROCEDURE(conn, buf$p, count, actual$p, status$p)EXTERNAL;
        DECLARE (conn, buf$p, count, actual$p, status$p) ADDRESS;
    END READ;


RENAME:
  PROCEDURE (old$p, newpath$p, status$pEXTERNAL;
      DECLARE (old$p, newpath$p, status$p) ADDRESS;
  END RENAME;


RESCAN:
  PROCEDURE (conn, status$p) EXTERNAL;
      DECLARE (conn, status$p) ADDRESS;
  END RESCAN;


RI: PROCEDURE BYTE EXTERNAL;               /*ENTRY POINT INTO SYSTEM.LIB*/
  END RI;


SEEK:
  PROCEDURE (conn, mode, block$p, byte$p, status$p) EXTERNAL;
      DECLARE (conn, mode, block$p, byte$p, status$p) ADDRESS;
  END SEEK;


SPATH:
  PROCEDURE (path$p, info$p, status$p) EXTERNAL;
      DECLARE (path$p, info$p, status$p) ADDRESS;
  END SPATH;


UI: PROCEDURE(PROM$ADDR) BYTE EXTERNAL;      /* RETURNS DATA IN PROM */
  DECLARE PROM$ADDR ADDRESS;                 /* AT PROM$ADDR */
  END UI;


UO; PROCEDURE(PROM$DATA,PROM$ADDR) EXTERNAL;  /* PROGRAM PROM LOCATION */
  DECLARE PROM$DATA BYTE;                     /* 'PROM$ADDR' WITH DATA */
  DECLARE PROM$ADDR ADDRESS;                  /* 'PROM$DATA' */
END UO;


UPPS: PROCEDURE BYTE EXTERNAL;             /* RETURNS UPP STATUS */
  END UPPS;


WRITE:
    PROCEDURE (conn, buf$p, count, status$p) EXTERNAL;
        DECLARE (conn, buf$p, count, status$p) ADDRESS;
    END WRITE;


WHOCON:
  PROCEDURE (conn, buf$p) EXTERNAL;
      DECLARE (conn, buf$p) ADDRESS;
  END WHOCON;
```

INDEX

This index uses the phrase "for Series III" as an abbreviation for "under the Series III operating system." Similarly, the phrase "for ISIS-II" means "under the ISIS-II operating system."

intel®

REQUEST FOR READER'S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process.

Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

____

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

____

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

____

4. Did you have any difficulty understanding descriptions or wording? Where?

____

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. ____________

NAME ______________________ DATE ____________

TITLE ______________________

COMPANY NAME/DEPARTMENT ______________________

ADDRESS ______________________

CITY ____________ STATE ____________ ZIP CODE ____________

Please check here if you require a written reply. ☐

WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.

NO POSTAGE
NECESSARY
IF MAILED
IN U.S.A.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051

intel®

INTEL CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) **987-8080**

Printed in U.S.A.