ISIS-II USER'S GUIDE

Order Number: 9800306-06

intel®

ISIS-II USER'S GUIDE

Order Number: 9800306-06

Copyright © 1976, 1977, 1978, 1979, 1980, 1981 Intel Corporation
Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95051

Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's software license, or as defined in ASPR 7-104.9(a)(9).

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products:

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

and the combination of ICE, iCS, iRMX, iSBC, iSBX, MCS, or RMX and a numerical suffix.

PRINTED IN U.S.A./A343/581/20K CP

PREFACE

This manual describes, and defines the use of, the Intel Systems Implementation Supervisor (ISIS-II). ISIS-II is the operating system for the Intellec and Intellec Series II microcomputer development systems.

The information in this manual supports the system designer who is using a microprocessor or related Intel products in a program-controlled system.

This manual contains seven chapters and five appendices:

- "Chapter 1. Introduction," which describes the general capabilities of ISIS-II.
- "Chapter 2. Getting Started with ISIS-II," which describes some of the programs of ISIS-II and the step-by-step procedures for loading and executing ISIS-II on the Intellec development systems.
- "Chapter 3. File Creation and Management," which describes and shows examples of the ISIS-II commands for entering, storing, and managing program and data files.
- "Chapter 4. Working with Program Modules," which describes and shows examples of the ISIS-II commands for storing, identifying, and manipulating user-written program modules.
- "Chapter 5. Use of ISIS-II and the Monitor by other Programs," which describes and shows examples of ISIS-II system calls that can be included in user-written programs to use ISIS-II and Monitor capabilities without direct intervention by the user from the system console.
- "Chapter 6. The Intellec Monitor," which describes and shows examples of the Monitor commands for program debugging.
- "Chapter 7. Interrupt Processing," which describes and shows examples of using the Intellec interrupt functions.
- "Appendix A. Hexadecimal Paper Tape Format," which describes the paper tape format used by the Monitor.
- "Appendix B. Hexadecimal-Decimal Conversion," which is an aid for doing hexadecimal to decimal and decimal to hexadecimal conversions.
- "Appendix C. Error Messages," which is a listing of the error messages issued by ISIS-II and the ISIS-II commands.
- "Appendix D. ISIS-II Sample Programs—TYPE," which lists two versions of TYPE, in PL/M and in 8080/8085 Assembly Language.
- "Appendix E. ASCII Codes," which shows ASCII codes, their meanings, and their values.

Related Publications

For more information on the Intellec Series II microcomputer development system see the following manuals:

- *A Guide to Intellec Microcomputer Development Systems,* 9800558, which introduces microcomputer development systems in general and the Intellec systems in particular. It presents an overview of the purpose and use of development systems.
- *Intellec Series II Model 210 User's Guide,* 9800557, which describes the use of the facilities of the Intellec Series II ROM-based Microcomputer Development System.

- *Intellec Series II Installation Manual,* 9800559, which contains the installation information for all models.
- *Intellec Series II Hardware Interface Manual,* 9800555, which describes the use of peripheral I/O devices with the Intellec Series II.
- *Intellec Series II Schematic Drawing,* 9800554, which contains the schematic diagrams for Intellec Series II Microcomputer Development Systems.

For more information on the Intellec Microcomputer Development System see the following manuals:

- *Intellec MDS Operator's Manual,* 9800129, which describes how to operate the basic system.
- *Intellec MDS Hardware Reference Manual,* 9800132, which describes the principles of operation of the system.
- *MDS-DOS Hardware Reference Manual,* 9800212, and the *Intellec Double Density Diskette Operating System Hardware Reference Manual,* 9800422, which describes the physical installation of the flexible disk drives supported by ISIS-II for the Intellec Microcomputer Development System.
- *MDS-740 Hard Disk Subsystem Operating and Checkout,* 9800943, which describes the day-to-day operation procedures for the hard disk drive.

Additionally, you may need some of the following manuals. These manuals are available from the Intel Corporation or your supplier.

- *PL/M-80 Programming Manual,* 9800286.
- *PL/M-80 Compiler Operator's Manual,* 9800300.
- *PL/M-86 Programming Manual,* 9800466.
- *PL/M-86 Compiler Operator's Manual,* 9800478.
- *FORTRAN-80 Programming Manual,* 9800481.
- *ISIS-II FORTRAN-80 Compiler Operator's Manual,* 9800480.
- *8080/8085 Assembly Language Programming Manual,* 9800301.
- *ISIS-II 8080/8085 Macro Assembler Operator's Manual,* 9800292.
- *MCS-86 Assembly Language Reference Manual,* 9800640.
- *MCS-86 Assembler Operating Instructions for ISIS-II Users,* 9800641.
- *BASIC-80 Reference Manual,* 9800758.
- *ISIS-II CREDIT (CRT-Based Text Editor) User's Guide,* 9800902
- *MCS-48 and UPI-41 Assembly Language Programming and Operator's Manual,* 9800255.

Notational Conventions

The following conventions are used to show syntax in this manual:

UPPERCASE	Information in uppercase must be entered as shown. It can be entered in uppercase or lowercase.
<lowercase>	Fields in lowercase indicate variable information. They are enclosed in angle brackets (< >) to show field limitation because some formats do not need a delimiting space or punctuation. The angle brackets should not be entered.
[ ]	Brackets indicate optional fields.

...	Ellipses indicate that a field may be repeated.
{ }	Braces indicate a choice. One of the items within the braces must be picked unless the field is also surrounded by brackets, in which case it is optional.
|	A vertical bar indicates choice of syntactic elements on either side.
punctuation	Punctuation other than ellipses, braces and brackets must be entered as shown. For example, the punctuation shown in the following command must be entered:

```
SUBMIT PLM80(PROGA,SRC,'9 SEPT 78')
```

CONTENTS

CHAPTER 1 INTRODUCTION

PAGE

The Intellec Systems 1-1
The ISIS-II Disk Operating System 1-1
ISIS-II Functions 1-2
Disk Files 1-2
Modular Programming 1-3

CHAPTER 2 GETTING STARTED WITH ISIS-II

System Composition 2-1
ISIS-II Disks 2-1
Memory 2-2
The Monitor 2-2
I/O Interface 2-2
Editing 2-3
Code Format Translations 2-3
Module Management 2-3
I/O Device Management 2-3
Device/File Name Format 2-3
System Designated Device Names 2-4
Device/File Accessing 2-5
System Start-Up 2-5
Flexible Disk Care 2-6
Hard Disk Care 2-6
Start-Up Procedure 2-7
Flexible Disk Start-Up 2-7
Intellec Series II Microcomputer Development Systems 2-7
Intellec Microcomputer Development Systems . 2-8
Hard Disk with Flexible Disk Start-Up 2-8
Hard Disk Start-Up 2-8
With a System Disk in :F0: 2-8
With a Non-System Disk in :F0: 2-9

CHAPTER 3 FILE CREATION AND MANAGEMENT

Console User Aids 3-1
Line Editing 3-1
The System Console 3-2
Intellec Interrupt Switches 3-2
Error Processing and Debugging 3-3
Disk Recording Characteristics 3-5
File Character Coding 3-5
Disk Organization 3-5
Disk Directory Content 3-6
Filename 3-6
Blocks 3-6
Length 3-6
Attributes 3-6
Wild Card File Names 3-7
File Copying 3-8
Using ISIS-II With a Single Drive System 3-8

PAGE

Using ISIS-II Commands 3-8
Command Syntax 3-10
Disk Maintenance Commands 3-10
IDISK - Disk Formatting Command 3-10
FORMAT - Disk Formatting Command 3-12
FIXMAP - Map Bad Sectors on Hard Disk 3-14
FIXMAP Commands 3-15
Mark Command 3-16
Free Command 3-17
List Command 3-18
Count Command 3-19
Record Command 3-19
Quit Command 3-20
Exit Command 3-20
FIXMAP Errors 3-21
Program Execution Commands 3-22
Filename - Direct Program Execution 3-23
DEBUG - Program Execution Under Monitor .. 3-23
SUBMIT - Non-Interactive Program Execution . 3-24
File Control Commands 3-26
DIR - Disk Directory Listing 3-26
COPY - Copy a File 3-28
HDCOPY - Copy Hard Disk Tracks 3-32
DELETE - Delete a Disk File 3-34
RENAME - Rename a Disk File 3-36
ATTRIB - Change/Display Disk File Attributes . 3-37
Code Conversion Commands 3-38
BINOBJ - Convert Binary to Absolute Object Module 3-38
HEXOBJ - Convert Hexadecimal Code to Absolute Object Code 3-39
OBJHEX - Convert ISIS-II Absolute Object Code to Hexadecimal 3-39
Using the Text Editor 3-40
The Editor Functions 3-40
Command Entry 3-41
Aborting Commands 3-41
Correcting Typing Errors 3-42
Formatting Characters 3-42
Carriage Return and Line Feed Characters 3-42
TAB Characters 3-42
Activating the Editor 3-42
Editor Commands 3-43
The Text Pointer 3-43
B - Beginning of Text Command 3-44
B Command Examples 3-44
Z - End of Text Command 3-44
Z Command Examples 3-45
L - Line Command 3-45
L Command Examples 3-45
C - Character Command 3-45
C Command Examples 3-46

CONTENTS (Cont'd.)

	PAGE
F - Find Command	3-46
F Command Examples	3-47
Text Commands	3-47
I - Insert Command	3-47
I Command Examples	3-47
S - Substitute Text Command	3-48
S Command Examples	3-49
D - Delete Command	3-49
D Command Examples	3-49
K - Kill Line Command	3-49
K Command Examples	3-50
Typing a File	3-50
T - Type Command	3-50
T Command Examples	3-51
Terminating a Session and Saving Your File	3-51
E - Exit Command	3-51
E Command Example	3-52
Q - Quit Command	3-52
Q Command Example	3-52
W - Write Command	3-52
W Command Example	3-52
Reading Data from Disk	3-53
A - Append Command	3-53
A Command Examples	3-53
Determining Memory Space Available	3-53
M - Memory Command	3-53
M Command Examples	3-54
Command String Iterations	3-54

CHAPTER 4
WORKING WITH PROGRAM MODULES

Microprocessor Memory Allocation	4-1
Program Segments	4-2
Code Segment	4-2
Data Segment	4-2
Stack Segment	4-3
Memory Segment	4-3
Common Segments	4-3
Absolute Information	4-3
Modular Program Development	4-3
Faster Program Development	4-4
Use of Different Source Language	4-4
Shared Subprograms	4-4
Easier Debugging and Program Modification	4-4
Mechanics of Relocation and Linkage	4-4
Relative Addressing	4-4
Intrasegment References	4-5
Intersegment References	4-5
External References and Public Symbols	4-5
Use of Libraries	4-6
LINK Command	4-7
Link Map	4-8
Order of Modules in the Output File	4-9
LOCATE Command	4-10
LOCATE Control Descriptions	4-11
MAP	4-11
COLUMNS (number)	4-12
PRINT (file)	4-12
SYMBOLS	4-12
LINES	4-13
PUBLICS	4-13
PURGE	4-13
ORDER(segment sequence)	4-13
NAME(name)	4-14
RESTART0	4-14
START(address)	4-15
STACKSIZE(value)	4-15
How LOCATE Locates Segments	4-15
Locating with the Default Order	4-15
Locating with the Default and ORDER Control	4-16
Locating with the Default, ORDER Control, and Specific Addresses	4-16
LIB Command	4-17
Continuation Lines	4-18
CREATE - Create a Library File	4-18
ADD - Add Modules to a Library File	4-18
DELETE - Delete Modules from a Library File	4-18
LIST - List Library Modules and Their Public Symbols	4-19
EXIT - Return to ISIS-II	4-19
Program Overlays and Linked Loading	4-20
Memory Pages and the H and L Registers	4-21

CHAPTER 5
USE OF ISIS-II BY OTHER PROGRAMS

Memory Organization and Allocation	5-1
Line-Edited Input Files	5-3
Terminating a Line	5-3
Reading from the Line-Edit Buffer	5-3
Editing Characters	5-4
Reading a Command Line	5-4
Summary of System Calls	5-5
System Call Syntax and Usage	5-5
PL/M Calls	5-6
Assembler Language Calls	5-6
File Input/Output Calls	5-7
System Calls Cautions	5-7
OPEN - Initialize File for Input/Output Operations	5-8

CONTENTS (Cont'd.)

PAGE

READ - Transfer Data from File to Memory..... 5-9
WRITE - Transfer Data from Memory to File... 5-10
SEEK - Position Disk File Marker............. 5-12
RESCAN - Position Marker to Beginning of Line........................ 5-14
CLOSE - Terminate Input/Output Operations on a File........................ 5-15
SPATH - Obtain File Information............ 5-16
Disk Directory Maintenance................... 5-18
DELETE - Delete a File from the Disk Directory........................ 5-18
RENAME - Change Disk Filename............ 5-19
ATTRIB - Change the Attribute of a Disk File........................ 5-20
Console Reassignment and Error Message Output. 5-22
CONSOL - Change Console Device........... 5-22
WHOCON - Determine File Assigned as System Console........................ 5-23
ERROR - Output Error Message on System Console........................ 5-24
Program Execution........................ 5-25
LOAD - Load a File of Executable Code and Transfer Control........................ 5-25
EXIT - Terminate Program and Return to ISIS-II........................ 5-26
Monitor I/O Interface Routines................ 5-27
CI - Console Input Routine.................. 5-28
CO - Console Output Routine................ 5-29
RI - Reader Input Routine................... 5-30
PO - Punch Output Routine.................. 5-31
LO - List Output Routine.................... 5-32
UI - Universal PROM Programmer Input Routine........................ 5-33
UO - Universal PROM Programmer Output Routine........................ 5-35
System Status Routines........................ 5-37
CSTS - Console Input Status Routine.......... 5-37
UPPS - Universal PROM Programmer Status Routine........................ 5-38
IODEF - I/O Definition Routine.............. 5-39
IOCHK - Check System I/O Configuration Routine........................ 5-40
IOSET - Set System I/O Configuration Routine. 5-42
MEMCHK - Check RAM Size Routine........ 5-42

CHAPTER 6
THE INTELLEC MONITOR
Command Entry........................ 6-1
Entry Errors........................ 6-2
Command Categories........................ 6-3
Monitor I/O Configuration Commands.......... 6-3
A - Assign Command........................ 6-4

PAGE

Q - Query Command........................ 6-5
Memory Control Commands.................... 6-6
D - Display Command........................ 6-6
F - Fill Command........................ 6-7
M - Move Command........................ 6-7
S - Substitute Command...................... 6-8
Register Command........................ 6-9
X - Register Command (Display Form)......... 6-9
X - Register Command (Modify Form)........ 6-10
Paper Tape I/O Commands.................... 6-11
R - Read Command........................ 6-11
W - Write Command........................ 6-12
E - End-of-File Command.................. 6-12
N - Null Command........................ 6-13
Execute Command........................ 6-14
G - Execute Command..................... 6-14
Utility Command........................ 6-16
H - Hexadecimal Command.................. 6-16

CHAPTER 7
INTERRUPT PROCESSING
Priority of Interrupts........................ 7-1
The Interrupt Mask Register.................... 7-1
Interrupt Mask Register Initialization............ 7-2
Interrupt Acceptance........................ 7-2
Interrupt Removal........................ 7-2

APPENDIX A
HEXADECIMAL PAPER TAPE FORMAT

APPENDIX B
HEXADECIMAL-DECIMAL CONVERSION

APPENDIX C
ERROR MESSAGES
Numbered ISIS-II Error Messages................. C-1
Link Error Messages........................ C-4
Fatal Error Messages........................ C-4
Non-Fatal Error Messages...................... C-5
Locate Error Messages........................ C-5
Fatal Error Messages........................ C-5
Non-Fatal Error Messages...................... C-7
Lib Error Messages........................ C-7
Editor Error Messages........................ C-9

APPENDIX D
ISIS-II SAMPLE PROGRAMS'TYPE

APPENDIX E
ASCII CODES

INDEX

TABLES

TABLE	TITLE	PAGE
2-1	ISIS-II Supported Configurations	2-1
C-1	Nonfatal Error Numbers Returned by System Calls	C-3
C-2	Fatal Errors Issued by System Calls	C-3
E-1	ASCII Code List	E-1
E-2	ASCII Code Definition	E-2

ILLUSTRATIONS

FIGURE	TITLE	PAGE
A-1	Paper Tape Record Format	A-1

CHAPTER 1
INTRODUCTION

This chapter is for the user who has little or no prior programming and computer system experience. However, the information is beneficial to anyone who wishes to familiarize herself or himself with the terms and concepts pertinent to ISIS-II.

The Intellec Systems

ISIS-II is an operating system for the Intellec development systems:
- Intellec Series II Microcomputer Development Systems
- Intellec Microcomputer Development System

ISIS-II supports all models of Intellec systems with one or more single- or double-density disk drives and at least 32K of RAM.

The Intellec systems support both software and hardware program development. In conjunction with ISIS-II, they provide the environment for software development, testing and debugging. You can execute your assembly language code on the development system during the logic debugging phase (simulating any specialized I/O used in your product).

When you add the appropriate In-Circuit Emulator you can execute and debug your code for all In-Circuit Emulator supported processor families.

ISIS-II supports the following peripheral devices:
- Universal PROM programmer
- Line printer
- Paper tape reader
- Paper tape punch
- TTY
- CRT terminals
- Flexible disk drives
- Hard disk drives

Additionally, you can attach non-standard I/O devices for which you have written drivers. This flexibility, along with the functions provided in the ISIS-II operating system, makes your Intellec system a complete microcomputer development system.

The ISIS-II Disk Operating System

An operating system is a group of programs that manages the resources of a system and frees you to concentrate on other tasks. It provides a simple command language that allows you to state what you want done and what device or file you want it done to. For example, a disk drive and a line printer have very little in common. The disk drive is a random access input/output device and the line printer is a sequential output-only device. The operating system lets you copy data to either one with the same command.

The computer has to execute thousands of instructions to carry out each command. It has to read the command, determine what it means, get the data from one location, copy it to another, and tell you that it is done. All you have to specify is what you want done; in this case, copying a file. You don't have to worry about the details.

The Intellec development systems contain a Monitor in ROM (Read Only Memory), which provides basic system control facilities in the form of commands (that you enter at the console) and system calls (that you code in your programs) to access memory and I/O devices.

The ISIS-II operating system uses the Monitor also, rather than duplicate functions that already exist there. In fact, all input and output operations, except for disk operations, are handled by the Monitor.

ISIS-II Functions

ISIS-II provides the functions to manage your programs and data and assemble your assembly language code. You can add the PL/M and FORTRAN compilers and other assemblers to ISIS-II.

To manage your programs and data ISIS-II provides:

- A text editor to create and change files. The text editor automatically keeps a copy of the unchanged version of a file when you make changes. The editor has commands to handle single characters and large numbers of lines.
- A function to delete whole files that are no longer needed
- Functions to convert program object code to and from hexadecimal formats
- A function to copy files
- A function to change the name of files
- A function to work with file attributes
- Functions to combine program parts into complete functional systems
- A feature to list all files, their length, and their attributes
- A feature to invoke system calls under pseudo-batch operation

Disk Files

The programs that make up ISIS-II are contained in disk files, on hard disk drives or flexible disk drives. When the system is reset with an ISIS-II system disk in the appropriate drive the operating system initializes and takes control of the system. Unless your flexible or hard disk has been formatted previously, you must format it for ISIS-II before it may be used.

Only the essential ISIS-II files are loaded. Programs to perform specific functions remain on disk until you enter a command that calls them. Then the required program or programs are loaded into memory and executed. This technique gives you the full capabilities of the operating system and lets you reserve the majority of memory space for your work. After the command program has completed its functions, the memory it was using is again available.

Your programs and data are also stored on disks. Each file on a disk has a name. ISIS-II program files come with names assigned; you name each file you create. ISIS-II keeps a directory of all the files on a disk. To access a file, you don't have to know its position on the disk, only its name. If you forget the name, there is a directory command (DIR) that lists the names of all files on a specified disk.

Any disk used by ISIS-II must be formatted by ISIS-II. Once this is done you can select or create files on the disk. There are four basic types of files:

- *Format files* that contain information about the disk itself. The directory is a format file. ISIS-II rejects disks that do not have the required format files.
- *System files* that contain the basic ISIS-II system programs and command programs. A disk that contains the minimum files essential for ISIS-II operation is called a system disk. A disk that does not have these files is called a non-system disk.
- *User-written* program files
- *Data files* that are used by your programs or by ISIS-II

Modular Programming

A modular approach to program design with ISIS-II is similar to a modular approach to designing a multifunction hardware system. In either case, you can determine the functions required and draw a block-diagram of the proposed system. Each block or module would then be handled as a discrete element whose inputs and outputs are fully defined in a specification.

For example, to produce a program for a traffic light controller, you might want a module to count vehicles approaching from a given direction, another module to determine the time of day; another for the time of year; one or two more modules to control the frequency and duration of the red, yellow, and green lights; and a final module to detect and allow passage of emergency vehicles.

Each of these modules can be written, tested, debugged, and modified independently with ISIS-II. You can also write different modules in different programming languages. For example, you might write the routines that do the calculations in FORTRAN because of its mathematical capabilities, input and output routines in PL/M because of the ease with which it handles port input and output, and assembly language for supervisor routines that handle the bit manipulation that may be required.

Finally, when all the modules are tested and debugged, you can combine them into a system with the ISIS-II LINK command. This LINK capability, combined with the other ISIS-II functions, makes modular programming possible. You don't have to write a single large program that is difficult to test and debug. You write small program segments and combine them into a complete program after they are individually debugged.

While the hardware portion of the system is being designed, you may not know what RAM and ROM addresses you will have for your code. You can write code that can be assigned fixed addresses at a later time, using the relocatable assemblers and compilers. The ISIS-II LOCATE command provides this facility. If testing of the final product finds problems that require the programs and data to be moved, you can do it with the LOCATE command rather than making extensive code changes.

The ISIS-II LIB command lets you create and change libraries of commonly used programs and subroutines. You can then include these programs and subroutines in your programs just by specifying their names. When you LINK your programs, you specify the library along with your programs. Only those programs that you use are included with your code.

During the development you can use the appropriate Intel In-Circuit Emulators to test your programs and hardware and simulate hardware configurations.

The Intellec system also interfaces to your Universal PROM Programmer to load your program in PROMs (Programmable Read Only Memory) or EPROMs (Erasable Programmable Read Only Memory).

CHAPTER 2
GETTING STARTED WITH ISIS-II

This chapter describes the various programs that are called by or associated with ISIS-II, and how these programs are used to manage system resources. It also defines the preliminary steps you must take to load and execute ISIS-II.

System Composition

The primary function of ISIS-II is to coordinate the hardware and software elements essential to development of your programs. The following sections define the ISIS-II associated hardware and software elements. Additional elements, such as In-Circuit Emulators, can be added to the system, but such additions do not alter the functions or operation of ISIS-II as defined in this manual.

ISIS-II Disks

ISIS-II, as well as utility programs and user programs, resides on flexible disks or hard disk platters. ISIS-II supports up to ten drives, including single- and double-density flexible disk drives and hard disk drives. Table 2-1 describes the ISIS-II supported configurations of disk drives.

Table 2-1. ISIS-II Supported Configurations

Configuration	Drive Numbers									
	0	1	2	3	4	5	6	7	8	9
H + D	H-F	H-R	*	*	D	D	(D)	(D)		
H + S	H-F	H-R	*	*	S	S				
H + IS	H-F	H-R	*	*	IS	*				
H + D + S	H-F	H-R	*	*	D	D	(D)	(D)	S	S
H + D + IS	H-F	H-R	*	*	D	D	(D)	(D)	IS	*
H + S + IS	H-F	H-R	*	*	S	S			IS	*
D	D	D	(D)	(D)						
S	S	S	(S)	(S)						
IS	IS	*								
D + S	D	D	(D)	(D)	S	S				
D + IS	D	D	(D)	(D)	IS	*				
S + IS	S	S	(S)	(S)	IS	*				
ID	ID	*								
ID + D	ID	*	D	D						
H + ID	H-F	H-R	*	*	ID	*				
H + ID + D	H-F	H-R	*	*	ID	*	D	D		

H = Hard disk
F = fixed platter of hard disk
R = removable platter of hard disk
D = Double density flexible disk
S = Single density flexible disk
IS = Integrated single density flexible disk
ID = Integrated double density flexible disk
* = Not available
Parentheses ( ) indicate optional drives within the particular configuration.

ISIS-II Flexible Disks

The flexible disks used by ISIS-II contain 77 tracks each. A single-density flexible disk has 26 sectors per track, and a double-density flexible disk has 52 sectors per track, each sector containing 128 bytes. Once a flexible disk has been formatted for one density, it must be reformatted to be used in a drive of another density.

ISIS-II Hard Disk Platters

The hard disk platters used by ISIS-II contain 400 tracks on each of two surfaces. Each track has 36 sectors of 128 bytes. This data format is translated into 200 logical tracks, with 144 logical sectors per logical track. There are two platters per hard disk drive, one fixed, and one removable. The fixed hard disk platter resides in drive 0; the removable hard disk cartridge resides in drive 1.

Memory

ISIS-II requires a minimum of 32K of random access memory (RAM). The first 12K (starting at location 0) is reserved for use by the portion of ISIS-II that must remain in memory during operation. An additional 320 bytes at the top of contiguous RAM is reserved for the Monitor. The remaining RAM is shared by the program being developed, ISIS-II programs loaded from disk, and data you use or generate during program development.

The Monitor

The Monitor handles the initiation of the system when you first start up. (The start-up procedure is described at the end of this chapter.)

In a system without hard disk drives, the Monitor checks drive :F0: to see if power is on and if a system flexible disk is loaded. If ISIS-II is on the flexible disk, it is loaded into memory and control is passed from the Monitor to ISIS-II. If there is no drive :F0:, or if it isn't ready, or if ISIS-II isn't on the disk, control is retained by the Monitor.

If you have a hard disk drive attached, the Monitor seeks a system flexible disk in drive :F4:.

After ISIS-II is loaded, you can access the Monitor functions through the DEBUG command (described in Chapter 3).

I/O Interface

The I/O interface to all standard peripheral devices except disk drives (console, printer, paper-tape reader and punch) is handled by the Monitor. When an input or output operation to these devices is needed, ISIS-II calls the appropriate Monitor routine. When the operation is completed, the Monitor routine returns control to ISIS-II.

ISIS-II handles the disk I/O itself. The routines in the Monitor are for the slower devices that accept or send data on a byte-by-byte basis.

Editing

Editing in the ISIS-II system is entering and modifying data. You can name a disk file, enter program source code, and save it for future use. After the data is entered and saved on disk, you can use it as input to the assemblers or compilers. If the program has errors, you can correct them and assemble or compile again.

Editing in ISIS-II is done with the Text Editor described in Chapter 3, or the optional editor CREDIT described in the *ISIS-II CREDIT* (CRT-Based Text Editor) *User's Guide*.

Code Format Translations

ISIS-II can convert object module format programs to or from the hexadecimal object format or from the absolute binary format. This capability is provided primarily for users of earlier versions of ISIS.

Module Management

Under ISIS-II, management of software modules or routines that comprise your program is module management. ISIS-II assigns absolute addresses to relocatable modules, combines independently developed modules, and creates a library of modules that are used by more than one program.

I/O Device Management

A primary function of ISIS-II is managing I/O devices. ISIS-II provides a number of aids that greatly simplify your management tasks. The ISIS-II aids are:

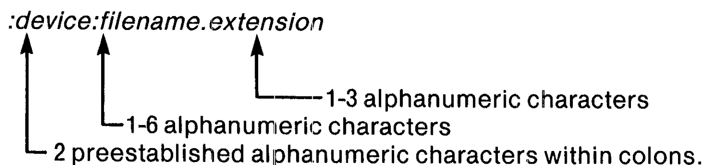
- A simple but uniform method of identifying each device and each disk file.
- Preestablished names for devices.
- Specifying a device or file for a series of subsequent operations.

Device/File Name Format

ISIS-II identifies each possible I/O device by means of a unique name. With disks, the name is expanded to identify the file that is to receive or provide data. The file name may be further expanded to allow for those instances when the file is to exist in several forms. For example, you may wish to retain a file in both its original and modified forms. You may also wish to use the same basic name for a program after it has been assembled.

The general format for naming devices and files is:

:device:filename.extension



:device: is a two character designation for a device. The valid device designations are listed in the next section. The device designation is always enclosed in colons. *:device:* is used alone when an actual device is referenced instead of a file that resides on a device.

filename is the one-to six-character name you create for a file. The characters can be alphabetic or numeric. Files can be referenced by the file name alone only if they reside on a disk loaded in drive :F0:. That is, if *:device:* is not specified, :F0: is the default.

.extension is a one-to three-character name you can create for files. An *.extension* is not required when a file is created. But if *.extension* is specified, it must always be used when referencing the file. A common use of *.extension* is to distinguish between different files associated with a single program.

For example:

:F1:PROGA.SRC	— for source code
:F1:PROGA.LST	— for the listing of the assembly
:F1:PROGA.OBJ	— for the object code
:F1:PROGA.LNK	— for the linked object code
:F1:PROGA	— for the code located at absolute addresses for execution

Note that all these files have the same *filename* and are distinguished only by *.extension.*

If the disk that contains these files was moved to drive :F2:, the names would become :F2:PROGA.SRC, :F2:PROGA.LST, etc. No matter where the disk is loaded, *filename* and *.extension* stay the same but *:device:* is dependent on the physical location.

System Designated Device Names

Filenames and extensions are assigned by you. However, device names are established by the system and include the following:

:F0: through :F9:	Disks (refer to Table 2-1 for supported configurations)
:TI:	Teletypewriter keyboard
:TO:	Teletypewriter printer
:TP:	Teletypewriter punch
:TR:	Teletypewriter reader
:VI:	Video terminal keyboard
:VO:	Video terminal screen
:HP:	High-speed paper tape punch
:HR:	High speed paper tape reader
:LP:	Line printer

In addition to the preceding, ISIS-II also assigns the following device names to certain generic devices that do not exist in their own right. These are device names associated with whatever devices are being used as the system console:

:CI:	Console input
:CO:	Console output

System operations are directly tied to the console. You can establish any input device as the console input device and any output device as the console output device. The means for doing this is the CONSOL system call as defined in Chapter 5.

If you are using an Intellec Series II Microcomputer Development System, the keyboard and screen of the terminal are :CI: and :CO: respectively.

ISIS-II also assigns the following device name to a nonexistent device:

:BB:	Byte bucket

The byte bucket is nonexistent, but is treated as a real device by ISIS-II commands. The byte bucket receives data that you wish discarded.

In some applications you may want to use I/O devices other than those available with the standard system configuration. This requires that you write your own I/O driver routines and define the nonstandard device with the Monitor I/O definition routine. ISIS-II identifies nonstandard devices via the following device names:

:R1:	Paper tape reader 1
:R2:	Paper tape reader 2
:P1:	Paper tape punch 1
:P2:	Paper tape punch 2
:L1:	Line printer 1
:I1:	Console input device
:O1:	Console output device

NOTE

To become active as the system console, nonstandard console devices must be assigned to :CI: and :CO: via the CONSOL system call.

Device/File Accessing

The assignment of names to devices and files saves time, since you gain access to these system resources without concern over the addresses that would otherwise be required. The system maintains a record of the device addresses established when the system is configured and of the file addresses established when files are created. Names are converted to true addresses during processing sequences.

Creating a file puts information about the file into system tables. When a file is deleted, its entries are deleted from the system tables.

When a file is accessed by ISIS-II in response to a console command such as COPY, ISIS-II opens and closes the file. When a file is accessed by a program, it must use the OPEN and CLOSE system calls.

Important points to remember are:

- ISIS-II commands are concerned with data transfers to or from a device.
- Accessing any device, file, or program involves a common I/O process.
- The name of the device, file, or program is included as a parameter of the current command or a parameter of a previous command.

System Start-Up

The step-by-step procedure used to load the ISIS-II program from the system disk is simple. However, before proceeding with this start-up procedure, you should be aware of several factors that will affect your use of disks.

Flexible Disk Care

Flexible disks are a cost-effective and convenient medium for the storage of bulk data. However, proper care of the flexible disks is required to ensure continued trouble-free reading and writing of flexible disk files.

Physical handling of a flexible disk requires care to avoid damage to the recording surface and to prevent flexible disk deformation. Specific precautions include:

- Return the flexible disk to its envelope when not in use.
- Do not touch the recording surface.
- Do not smoke when handling the flexible disk.
- Do not clean the recording surface.
- Do not bend the flexible disk or deform its edges by using paper clips or other mechanical devices.
- Do not use pencil or ball point pen on the flexible disk label: use felt tip pen.

The operating and storage environment must be compatible with the materials of the flexible disk. The environment of the flexible disk should meet the following criteria:

- No noticeable dirt, dust, or chemical fumes in the immediate area.
- Temperature between 50 F° (10 C°) and 125 F° (52 C°).
- Relative humidity between 8 and 80 percent.
- Wet bulb temperature of 85 F° (30 C°) maximum.
- No direct sunlight on flexible disk surface for prolonged periods.
- No magnetic field.

Loading a flexible disk into a flexible disk drive and removing it from a drive requires a few precautions to avoid damage and to ensure proper operation. These precautions include:

- Do not insert or remove a flexible disk unless power is applied to both the system and the flexible disk drive.
- Do not open the flexible disk drive door unless the DRIVE light is off (push Intellec RESET switch and hold down, if necessary, to disengage drive).
- Do not attempt loading of single-density flexible disks from double-density drive or vice versa.
- Insert flexible disk with read/write access slot first.
- Close door of the drive after flexible disk insertion.
- Do not remove flexible disk unless the last output to console was a hyphen (ISIS-II is ready to receive a command).
- Do not attempt to write to a flexible disk unless it has been formatted as defined in Chapter 3.

Hard Disk Care

The hard disk drive assembly (that is, the hard disk drive with removable cartridge installed) is *extremely* sensitive to contaminants on the hard disk platter surface. The head does not make contact with the disk platter, but rides about 1.14 microns above it. An average human hair is about 100 microns in diameter, and a typical smoke particle is about 6.35 microns. If either of these objects or such "invisible" contaminants as fingerprints or dust particles comes between the disk drive head and the disk platter surface, the contact will usually destroy both the head and the disk.

Follow these procedures and precautions when using a hard disk drive:

- The operating environment should meet the criteria listed above for flexible disk care, with the following additions.
- Clean any dust or dirt from the cartridge cover and drive chassis. Use a lint-free cloth.
- Allow nothing to touch the disk surface.
- Inspect the disk surfaces periodically. If it is visibly dirty, or scratched, it must be serviced. Contact your Intel service representative.
- Keep liquids away from the hard disk drive.
- Do not smoke in the hard disk drive area.
- Before using a disk cartridge that has just been brought into a new operating environment, allow at least one hour to let the cartridge temperature stabilize.

For further information, refer to the *MDS-740 Hard Disk Subsystem Operation and Checkout Manual,* 9800943. (In that manual the Intellec Microcomputer Development System is referred to as the Intellec 800.)

Start-Up Procedure

ISIS-II can be run on the Intellec Series II Microcomputer Development Systems or the Intellec Microcomputer Development Systems. The start-up procedures for the two systems differ because of hardware differences, but the operating system functions the same in both systems.

The following can damage or modify the contents of a disk:

Removal or application of power with the disk installed and the drive engaged.

Pressing the top of the BOOT switch (selecting the Monitor bootstrap) during a disk I/O operation.

Removal of a disk while not at the ISIS-II command level (last console output is a hyphen).

Failing to wait for INTERRUPT 2 light to go on and then off before pressing the bottom of the BOOT switch during start-up.

NOTE

If the console indicates that the Monitor is in control after the ISIS-II start-up procedure, check for a non-system disk in drive :F0:, an incorrectly installed disk, or disconnected drive.

Flexible Disk Start-Up

Intellec Series II Microcomputer Development Systems

The following procedure defines system start-up from power application through loading and execution of ISIS-II for the Intellec Series II Microcomputer Development Systems:

1. Apply power to the system and external flexible disk drive unit (if attached).

2. Insert the ISIS-II system flexible disk in drive :F0: (write protect slot first). Push the flexible disk into the slot until it latches.
3. Close the drive door.
4. Press the RESET button.
5. Observe that the following appears on the console screen to indicate the readiness of ISIS-II to accept commands: ISIS-II, Vm.n where Vm.n is the version number.

Intellec Microcomputer Development System

The following procedure defines system start-up from power application through loading and execution for the Intellec Microcomputer Development System.

1. Apply power to system, disk drive, and console.
2. Insert ISIS-II system disk in drive :F0: (write protect slot first).
3. Close drive door.
4. Press top of Intellec BOOT switch.
5. Press top of RESET switch.
6. Observe that INTERRUPT 2 light goes on to indicate loading of ISIS-II.

Be sure INTERRUPT 2 is on before proceeding.

7. Press space bar of video terminal or teletypewriter to select console.
8. Observe that INTERRUPT 2 light goes off to indicate receipt of space bar entry.
9. Press bottom of BOOT switch to execute ISIS-II.
10. Observe that the following appears at the selected console to indicate readiness of ISIS-II to accept commands: ISIS-II,Vm.n where Vm.n is version number.

Hard Disk with Flexible Disk Start-Up

Hard Disk Startup

When you start up a system that includes a hard disk, or when you press the RESET switch on an Intellec Series II Microcomputer Development System that includes a hard disk, ISIS-II is loaded from a flexible disk in drive :F4:.

After the system is initialized with ISIS-II system files from :F4:, a fatal error, software reboot, or interrupt 1 will cause the Intellec Series II Microcomputer Development System to refer to the hard disk platter in drive :F0: for ISIS-II system files. If the RESET switch is pressed, the processor refers to the system flexible disk in :F4:. Therefore, after the system is initialized, you can remove the system disk from :F4: until RESET is again pressed.

With a System Disk in :F0:

In an environment with both hard disk drives and flexible disk drives, and a system hard disk in drive :F0:, follow these procedures:

1. Apply power to Intellec Series II Microcomputer Development System or Intellec Microcomputer Development System.
2. Apply power to hard and flexible disk drives :F0: and :F4:, respectively.

3. Place a system flexible disk in :F4:.

   Press START switch on hard disk drive; wait for READY light to come on.

4. Press RESET on an Intellec Series II, or if using an Intellec Microcomputer Development System:

   a. Set BOOT to ON

   b. Press RESET

   c. Press space bar

   d. Set BOOT to OFF

5. The message ISIS-II, Vm.n is displayed, and the system is ready to accept commands.

With a Non-System Disk in :F0:

This environment lacks a system hard disk in :F0:. The user typically would build a system hard disk on :F0: after invoking ISIS-II. Follow these procedures:

1. Apply power to Intellec Series II Microcomputer Development System or Intellec Microcomputer Development System.
2. Apply power to the hard and flexible disk drives :F0: and :F4:, respectively. Make sure the hard disk drive does *not* indicate READY.
3. Insert a system flexible disk in :F4:.
4. Press RESET on an Intellec Series II, or if using an Intellec Microcomputer Development System:

   a. Set BOOT to ON

   b. Press RESET

   c. Press space bar

   d. Set BOOT to OFF

5. Press START switch on hard disk drive; wait for READY light to come on.
6. Type:

   :F4:FORMAT :F0:SYSTEM.DSK S FROM 4

   this formats the hard disk platter in :F0: as a system disk.

7. Repeat step 4. The hard disk in :F0: is now the system disk.

When turning off the system, be sure the hard disk READY light is OFF before turning off the Intellec Series II Microcomputer Development System or Intellec Microcomputer Development System. If the READY light is on, press and release the START/STOP switch.

CHAPTER 3
FILE CREATION AND MANAGEMENT

This chapter describes how to use ISIS-II to create, revise, and manage files. In most cases the files will be program files generated through the use of an assembler or compiler. However, ISIS-II can also create data or program files through the use of Text Editor commands that are designed to add, delete and replace characters or lines as displayed on the system console.

The chapter begins with an explanation of various aids that are available to the console user. These aids are operating hints or simply descriptions of system characteristics that should be understood before attempting more complex tasks. The chapter then goes on to explain the use of ISIS-II commands that allow you to create, revise, or delete files and/or references to files within a file directory. The final subject matter explains use of the ISIS-II Text Editor. The CREDIT CRT- Based Text Editor is described in the *ISIS-II CREDIT (CRT-Based Text Editor) User's Guide,* 9800902.

Console User Aids

The explanation of console user aids is essentially an introduction to the use of ISIS-II and its subordinate programs. Many of the items covered are delineated to the extent necessary to fully define their impact on major ISIS-II functions. Other items are introduced at this point and receive further coverage elsewhere.

Line Editing

There is no direct hardware data link between the console's keyboard and CRT; rather, each keyboard input to the computer is echoed back, by software, to the console for display (or printing). This technique assures you that the intended character has been entered without error. The entered (input) characters and the echoed (output) characters are stored in separate buffers until a carriage return is typed or a maximum of 122 characters have been entered. In either case, the input (line editing) buffer content is forwarded to its intended destination via one or more system generated READ commands. Prior to line termination, you can revise or delete the buffer contents by using special non-printable 'editing' characters and character combinations. These 'editing' characters are not normally stored in the line editing buffer but rather provide control over the buffer contents. The editing characters are:

RUBOUT — Deletes the preceding character from the line editing buffer. The deleted character also disappears from the CRT. On systems using a teletypewriter, RUBOUT echoes the deleted character to the teletypewriter. Repeated usage is allowed.

CONTROL/P — Used before another editing character (including CONTROL/P) to allow entry of the editing character into the line editing buffer.

CONTROL/R — Causes the display of the current contents of the line editing buffer.

CONTROL/X — Deletes the entire contents of the line editing buffer. Causes the display of a number sign (#) followed by a carriage return and line feed.

CONTROL/Z Causes an end-of-file to be input to the system. There is no change to the display. The contents of the line editing buffer are deleted.

NOTE

The following characters are not editing characters, but they affect terminal output:

CONTROL/S Stops terminal output and delays program execution.

CONTROL/Q Resumes terminal output after the CONTROL/S command is given.

The line-editing facility can be applied to files other than the console input device. There is a fuller description of line-editing in the section called Line-Edited Input Files, in Chapter 5.

The System Console

The device used as the console in the bootstrap procedure is known as the 'initial system console.' The console can be changed to another device by a program executing a CONSOL system call (see Chapter 5) or by using the SUBMIT command. The 'current console' is the device currently serving as console, which may or may not be the same as the initial system console.

The console, whatever device it is assigned to, is always the source of system commands. The SUBMIT command directs ISIS-II to take commands from a disk file. The SUBMIT file can return control to the initial system console by means of a control/E. The console can also use control/E to return control to the SUBMIT file.

When control/E is input to ISIS-II as part of the command line either from the SUBMIT file or the console keyboard, control/E is echoed but is not entered in the input buffer. To enter a control/E into the input buffer and subsequently into a SUBMIT file, the control/E must be preceded by control/P, in which case control/E is entered as a literal.

Under ISIS-II the files :CI: and :CO: are pseudonyms for the devices serving as console input and output. The :CI: file is always the source of system commands. The :CO: file receives console output such as the echo of a command. These two files are always open. However, it is not an error for a program to issue an OPEN system call for either of these files. Neither :CI: or :CO: count as one of the six files allowed open simultaneously by ISIS-II.

After system bootstrap in an Intellec Microcomputer Development System, the files :CI: and :CO: correspond to either :VI: and :VO: or :TI: and :TO:, depending on which device the space bar was typed during the bootstrap procedure. In the Intellec Series II Microcomputer Development System, :CI: and :CO: are automatically assigned to :VI: and :VO: (the integral console devices). The files :CI: and :CO: can be reassigned to other files by programs issuing the CONSOL system call.

When an end-of-file is encountered in :CI:, :CI: and :CO: are closed, ISIS-II is reloaded, and the initial system console is reopened as :CI: and :CO:.

Intellec Interrupt Switches

INTERRUPT switches 0 and 1 on the Intellec front panel are directly related to ISIS-II operation. Both switches terminate processing. However, INTERRUPT 0 returns control to the Monitor whereas INTERRUPT 1 permits ISIS-II to retain control. INTERRUPT 1 can be used at any time to terminate processing. When interrupt switch 1 is pressed, the following occur:

- All open files are closed in their current state.
- The initial system console becomes the current console.
- A fresh copy of ISIS-II is read in from the system disk, and ISIS-II prompts for a command (hyphen).

Interrupt switch 0 can be used to stop processing and pass control to the Monitor. Processing can be restarted at the point of interruption by entering a Monitor G command (execution starts at the location specified by the contents of the program counter.)

If you want to terminate processing and return to ISIS-II command level from the Monitor, use the G8 Monitor command to close all open files, and restart ISIS-II, which prompts for a command. This is equivalent to pressing interrupt switch 1 except that the Monitor masks all interrupts except interrupt 0, making interrupt switch 1 inoperative when the Monitor is in control.

In summary, to pass control from ISIS-II to the Monitor do one of the following:

- Press the interrupt 0 switch.
- Enter the DEBUG command (described later in this chapter).
- Execute a LOAD system call with a transfer value of 2.

To pass control from the MONITOR to ISIS-II enter a Monitor G8 command.

Error Processing and Debugging

Errors encountered by ISIS-II are displayed as decimal numbers and are either fatal or nonfatal. The significance of each error number and identification of fatal errors are provided in Appendix C. If the error encountered is nonfatal, processing is immediately terminated and the error number is returned to the user program.

A fatal error causes a message of the type

```
ERROR nnn USER PC mmmm
```

to be output to the initial system console, where "nnn" is the error number and "mmmm" is the contents of the program counter when the error occurred. Control then returns to ISIS-II or the Monitor depending on the setting of the debug toggle.

If the error number is 24 (disk input/output error), an additional message of the type

```
STATUS = 00nn
D=x T=yyy S=zzz
```

is output to the console, where "x" represents the drive number, "yyy" the track address, and "zzz" the sector address, and where "nn" identifies the type of input/output error that occurred as explained in Appendix C.

When error 24 occurs, the disk surface may be damaged, in which case you may wish to retain as much of the disk data as possible. This may be done by copying the files to a good disk with the COPY command.

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.

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 a DEBUG command.
- 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 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.

Disk Recording Characteristics

The following paragraphs contain information you need to properly use ISIS-II commands and understand the prompts and error messages.

File Character Coding

Although all data in files is some sequence of binary zeros and ones, these bits are interpreted in different ways according to the type of file. The editor creates files of ASCII-coded text. The data in source programs is interpreted as ASCII-coded text by ASM80, PLM80, or FORT80. When ASM80, PLM80, and FORT80 translate source programs, they produce object programs containing the actual machine instructions and auxiliary information used by the linker, locater, and loader.

The kind of data in a file determines what can be done with the file. An ASCII file can be sent to the line printer; others will print, but the output won't be intelligible. Object programs cannot be executed unless they have absolute addresses assigned by the locater. Even then they may not execute properly if they refer to external locations not present in the object program (see Chapter 4).

It is important to know the kind of data in a file so you can know what can be done with the file. You can use the extension of the filename to reflect the type of data in the file. For example, source programs could have the extension .SRC (or .ASM, .PLM, or .FOR to reflect the language used). The object code produced by the translators could have the extension .OBJ; in fact, if you do not supply a name for the output file, the translators use the name of the input file with the extension .OBJ.

Many ISIS-II programs in addition to the translators assign a specific extension on the input file if you do not supply it. One program, SUBMIT, assumes the extension .CSD on the input file if you do not supply it. This saves you time entering commands but can lead to misunderstandings if you are not aware of what the program does. Other ISIS-II programs create temporary working files with the extension TMP. Some examples of these are EDIT.TMP, LOCATE.TMP, LINK.TMP, and LIB.TMP.

Disk Organization

Flexible disks contain 77 tracks. Tracks of single density flexible disks contain 26 blocks of 128 bytes each. Tracks of double density flexible disks contain 52 blocks of 128 bytes each. Hard disks have 800 tracks, each track having 36 sectors of 128 bytes each. Disks can be either system or non-system.

A system disk contains ISIS-II software necessary for operating the system. A non-system disk contains only the information necessary for maintaining the directory of files on that disk, leaving more space for data than on a system disk. Disks must be formatted before they can be used in the system. See the FORMAT and IDISK commands described later in this chapter.

ISIS-II treats a file as a string of bytes. Space is allocated to a file in complete blocks even though the last block in the file may be only partially used. In other words, a block is not shared by two files. When a file is deleted, the blocks it occupied are released for reassignment by ISIS-II.

Disk Directory Content

Each flexible disk directory has space for 200 entries. Each hard disk directory has space for 992 entries. This means that a flexible disk can contain 200 files, or a hard disk 992, if the combined size of the files does not exceed the capacity of the disk.

A directory entry contains identifying information about a file. For example, it includes the following items:

- Filename
- Number of blocks allocated to the file
- Number of bytes in the file (length)
- Attributes

ISIS-II provides a DIR command that displays some of the information contained in a disk directory. The DIR command is described later in this chapter.

Filename

Filename within the directory includes *.extension*. The *:device:* is not stored in the directory as it only specifies which drive the disk is mounted in during the current ISIS-II execution.

Blocks

The number of blocks required by a file includes blocks for data and blocks for pointers that are used by the system to specify the location of files. For every 62 blocks of data, another block is required by the system for pointers. The total number of data and pointer blocks required by a file of N bytes can be calculated by the following formula (any result containing a decimal fraction is rounded to the next higher integer value).

[63*]N/128[/62]

For example, a file of 9000 bytes requires

[63*]9000/128[/62]=
[63*]70.31[/62]=
[63*71/62]=
[72.15]=
73 blocks required

Length

The number of bytes comprising a file is its length. Length increases as a file is written and can be affected by other programs containing the ISIS-II system calls OPEN and SEEK. Refer to Chapter 5 for further information on these system calls.

Attributes

There are four attributes associated with each file that may be set or reset (turned on or off) by the ATTRIB command or the ATTRIB system call. The attributes are:

- Invisible
- Write-protect
- Format
- System

When a file is created, all attributes are reset. That is, the file does not possess any of the attributes listed above. This applies to a file created by the COPY command, also. (Attributes can be copied with the COPY command by specifying the C switch.)

Invisible: Files with the invisible attribute set are not listed by the DIR command unless the I switch is used. Most system files possess the invisible attribute. It is advisable to make invisible files write-protected also.

Write-protect: Files with the write-protect attribute set cannot be opened for update or for output and cannot be deleted or renamed with ISIS-II commands from the console. Write-protected files can be overwritten by the IDISK and FORMAT commands. Write-protected files cannot be opened for output or update with the OPEN system call (see Chapter 5).

Format: What applies to write-protected files also applies to files with the format attribute set. In addition, files with the format attribute are created on a new disk when it is formatted by the FORMAT or IDISK commands. The following files have the format attribute: ISIS.DIR, ISIS.MAP, ISIS-T0, ISIS.LAB, ISIS.BIN, and ISIS.CL1, plus ISIS-BAD if a hard disk. A system disk must have all of these files. A non-system disk must have ISIS.DIR, ISIS.MAP, ISIS.T0, and ISIS.LAB, plus ISIS.BAD if a hard disk. You should not assign the format attribute to any other file, nor remove the format attribute from a file.

System: Files with the system attribute set are copied to the disk being formatted when the S switch is specified with the FORMAT command. When you use the IDISK command, you must copy the files with the COPY command. This gives you the option of choosing the files for the new system disk.

Wild Card File Names

Some ISIS-II commands allow you to specify filenames using a wild card construct. This means you can use an asterisk (*) or a question mark (?) to replace some or all of the characters in a name or extension. These special characters mean match anything when searching a directory for a filename. For example,

name.* - means match any filename with name and any extension or without an extension.

*.extension - means match any filename with extension and any name.

. - means match any filename.

The asterisk can also specify a wild card match for the remainder of the name or extension but not for initial characters. For example,

AB*.HEX - means match any filename with AB as first two characters of the name and HEX as the extension. This example would match: ABC.HEX, ABXYZ.HEX, AB.HEX.

*B.HEX is illegal, since * must follow initial alphanumeric characters.

*.BAK - means match any filename with a .BAK extension. This example would match: A.BAK, AB.BAK, or ABC.BAK.

The question mark specifies a single character for a wild card match. For example,

A?B.HEX - means match any filename with A and B as the first and third characters of a three-character name and HEX as the extension. This example would match: ACB.HEX, AXB.HEX, AMB.HEX.

A??.* - means match any filename with A as first character of a three-character name and any extension.

:device: cannot include a wild card character.

File Copying

Data can be transferred from one file or device to another by using ISIS-II commands. For instance, if you want to transfer the contents of the high-speed paper tape reader to a file on drive :F0:, you load the paper tape reader and use the COPY command to transfer the contents of the paper tape to the disk file.

```
COPY :HR: TO PROG.SRC
```

If you want to display a file on the console, use the COPY command in the following way:

```
COPY FILE.TXT TO :CO:
```

The HDCOPY command performs a track-by-track copy from one hard disk to another hard disk.

The ISIS-II Text Editor transfers files in a special way. The method employed by the Text Editor is defined later in this chapter.

Using ISIS-II With A Single Drive System

On an Intellec Series II Microcomputer Development System containing a single integrated disk drive, ISIS-II utility programs that have the P (Pause) switch allow you to remove the system disk, insert another disk, and carry out an operation.

Since most programs (such as the ISIS-II Text Editor, BASIC-80, and CREDIT) reload ISIS-II system files when you exit from them, be sure to format all disks used on a single drive system as basic system disks, using the IDISK command with the S switch.

Using ISIS-II Commands

The file management capabilities of ISIS-II can be controlled in one of three ways:

- Direct entry of ISIS-II commands at a console keyboard.
- Issuance of ISIS-II system calls by a program.
- Entry of console commands into a file by using the Text Editor. Execution of this file via the SUBMIT command causes ISIS-II to respond as if it were receiving commands directly from you. The advantages are that you need not be present when the submitted job is performed, and the commands do not have to be re-entered each time the job is submitted.

The following text defines only those commands input by you at the console. The definitions include a summary of the preparation of the file created by the command. The use of system calls is described in Chapter 5.

ISIS-II console commands perform four basic tasks:

- Prepare a new disk for use by the system (i.e., format the disk).
- Execute programs.
- Create, delete, and revise files and directories.
- Convert object file formats.

The ISIS-II console commands associated with each of the preceding tasks is identified in the following list and is described in subsequent paragraphs.

Disk Maintenance

IDISK	Format a new disk to a basic system or non-system disk.
FORMAT	Format a new disk and copy files.
FIXMAP	Map bad sectors on a hard disk.

Program Execution

filename	Execute the program named "filename"
DEBUG	Load a program and give control to the Monitor
SUBMIT	Enter the file that contains commands to be executed

File Maintenance

DIR	Output the names of and information about the files listed within the disk directory
COPY	Copy a file from one device to another
HDCOPY	Copy hard disk tracks to another hard disk
DELETE	Remove references to a file from the directory and free disk storage space associated with that file
RENAME	Change the name of a disk file
ATTRIB	Change and/or display the attribute(s) of a disk file

Code Conversion

BINOBJ	Convert a program from binary to object module format
HEXOBJ	Convert a program from hexadecimal to object module format
OBJHEX	Convert a program from object module to hexadecimal format

File Editing

EDIT	Create and modify ISIS-II files. The Text Editor subcommands are described in this chapter.

Program Control

LIB	Create and control program libraries (described in Chapter 4).
LINK	Combine program files and resolve external addressing (described in Chapter 4).
LOCATE	Convert relocatable object to absolute addresses for execution (described in Chapter 4).

Command Syntax

The general syntax of ISIS-II console commands is:

<command> <parameters>

where

<command> is the name of ISIS-II command program.

<parameters> are one or more data required by the command. When more than one parameter is specified, they are separated by commas or blank spaces unless otherwise noted under the individual commands.

A parameter may consist of one or more switches. Switches may be separated by spaces but not by commas.

In some cases a parameter requires additional information. Such information immediately follows the parameter and is enclosed in parentheses as shown in the following example:

SUBMIT PLM80(PROGA, SRC, '9 SEPT 78')

In ISIS-I some commands required a dollar sign ($) before each parameter. In ISIS-II those commands accept the dollar sign but treat it as a blank.

Brackets ([ ]) in the syntax indicate that a parameter is optional. If the option is omitted, default actions are performed by ISIS-II as explained with each command.

In most cases a command is executed when the carriage return is encountered. Any exceptions are noted under the individual commands.

Disk Maintenance Commands

IDISK - Disk Formatting Command

A blank disk must be formatted before it can be used by ISIS-II. There are two types of ISIS-II disks: system and non-system. The type of disk to be formatted is specified by the presence or absence of a switch with the command.

IDISK copies only the files needed for a basic disk (whether system or non-system). A basic disk contains only the files needed to format the disk: ISIS.DIR, ISIS.MAP ISIS.T0, and ISIS.LAB, plus ISIS.BAD if a hard disk. For a basic system disk, IDISK copies two additional files: ISIS.BIN and ISIS.CLI.

If any additional files such as command files are to be on the new disk, they must be copied with the COPY command.

IDISK can be used on single drive and multiple drive disk systems. On single drive systems, you are prompted to remove the system disk and insert the blank disk. When the formatting is completed, you are prompted to insert the original system disk. In systems with only 32K of RAM, two swaps of disks are required.

The syntax of the IDISK command is:

IDISK <device> <label> [<switches>]

where

<device> is the name of the drive containing the disk to be formatted. If you specify :F0:, and no FROM switch, the system assumes a single disk system and prompts for disk swaps as needed. Disk swapping is not required if any drive other than :F0: is specified. A hard disk in :F0: must be formatted as a system disk.

<label> is the name to be given to the disk. The syntax of label is the same as for filename with up to six characters for name and three for extension.

<device><label> must be entered with no intervening space or comma, as in :F1:MYDISK. At least one space must be entered before and after <device> <label>.

<switches> are one or more of the following:

S specifies that the new disk is formatted as a basic system disk. If S is not specified, the disk is formatted as a basic non-system disk.

P specifies that IDISK operates in single drive mode and prompts for output and system disks, pausing to display the prompt messages and to allow changing of disks.

FROM n specifies the disk drive containing the disk files needed for formatting. n is an integer 0-9, specifying drives :F0: through :F9:. If the FROM n switch is not specified, the default is to :F0:. If n is not a valid integer 0-9, the following error message appears on the :CO: device:

```
UNRECOGNIZED SWITCH
```

When used with a hard disk, IDISK verifies each sector. If IDISK cannot read a sector reserved for an ISIS-II file, the message

```
FATAL BAD SPOT AT LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

appears on the console, and ISIS.CLI is reloaded. ttt is the logical track address (in decimal); sss is the logical sector address (in decimal); and nnnn is the hard disk error status (in hexadecimal). If the unreadable sector does not correspond to an ISIS- II file, then the message

```
BAD SPOT AT LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

is displayed. Since ISIS-II allocates hard disk sectors serially, if no mechanism existed to "skip over" bad hard disk sectors, the remaining sectors would remain unallocated and unusable. Instead, ISIS-II checks hard disk sectors for irregularities during FORMAT and IDISK operations. If a bad sector is encountered, it is allocated to ISIS.BAD and hard disk formatting continues.

Example 1: This example formats a new disk in drive :F0: as a basic system disk on a single-drive system. IDISK prompts for the new (output) disk and for the system disk. IDISK gives the disk the name SYS.V1. The COPY command should be used to copy other files on the newly formatted disk. See the COPY command for single drive systems later in this chapter.

```
—IDISK :F0:SYS.V1 S
SYSTEM DISK
LOAD OUTPUT DISK, THEN TYPE (CR)
LOAD SYSTEM DISK, THEN TYPE (CR)
—
```

Example 2: This example formats a new disk in drive :F1: as a basic system disk, and gives the disk the name NSYS.V1. The COPY command copies all other non-format files from the disk in drive :F0: to the disk in drive :F1:.

```
—IDISK :F1:NSYS.V1 S
SYSTEM DISK
—COPY *.* TO :F1: C B
COPIED :F0:ASM80 TO :F1:ASM80
COPIED :F0:ASM80.OV0 TO :F1:ASM80.OV0
COPIED :F0:ASM80.OV1 TO :F1:ASM80.OV1
.
.
.
COPIED :F0:FPAL.LIB TO :F1:FPAL.LIB
COPIED :F0:PLM80.LIB TO :F1:PLM80.LIB
COPIED :F0:SYSTEM.LIB TO :F1:SYSTEM.LIB
—
```

Example 3: This example formats a hard disk platter in drive :F0: as a basic system disk; the basic files needed to format the disk are copied from a system disk in drive :F4:. The COPY command should then be used to copy other files onto the newly formatted disk, as in the previous example.

```
—:F4:IDISK :F0:SYSTEM.DSK S FROM 4
```

FORMAT - Disk Formatting Command

A blank disk must be formatted before it can be used by ISIS-II. FORMAT cannot be used on a single flexible disk drive system. IDISK should be used.

There are two types of ISIS-II disks: system and non-system. A disk is formatted as a system or non-system disk depending on the type of source disk used and on the switches specified in the FORMAT command.

When a system disk is formatted, FORMAT copies other files in addition to the basic format files. When a non-system disk is formatted, however, FORMAT copies only the basic format files: ISIS.DIR, ISIS.MAP, ISIS.T0, and ISIS.LAB, plus ISIS.BAD if a hard disk. Any other files that are to be on the new disk must be copied with the COPY command.

The syntax of the FORMAT command is:

FORMAT <device><label>[<switches>]

where

<device> is the name of the drive containing the disk to be formatted. If you specify :F0: and no FROM switch, or if you specify :F0: and FROM 0, then an error will result. A hard disk in :F0: must be formatted as a system disk.

NOTE

In the previous versions of ISIS-II, <device> defaulted to :F1: in the FORMAT command. This default has been removed. If you have SUBMIT files that use this default, you must change them. You will receive an error message if you try to default <device>.

<label> is the name to be given to the disk. The syntax of <label> is the same as for filename with up to six characters for name and three for extension.

<device><label> must be entered with no intervening space or comma, as in :F1:MYDISK. At least one space must be entered before and after <device> <label>.

<switches> are one or more of the following:

A copies all files to the specified drive. The new disk contains all the files that were on the source disk. If the source disk is a system disk, the new disk becomes a system disk.

S copies the basic format files and all files with the system attribute set. If the source disk is a system disk, the new disk becomes a system disk. (The S switch functions differently under FORMAT than it does under IDISK.)

FROM n specifies the disk drive containing the disk files needed for formatting. n is an integer 0-9, specifying drives :F0: through :F9:. If the FROM n switch is not specified, the default is to :F0:. If n is not a valid integer 0-9, the following error message appears on the :CO: device:

```
UNRECOGNIZED SWITCH
```

When used with a hard disk, FORMAT verifies each sector. If FORMAT cannot read a sector reserved for an ISIS-II file, the message

```
FATAL BAD SPOT AT LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

appears on the console, and ISIS.CLI is reloaded. ttt is the logical track address (in decimal); sss is the logical sector address (in decimal); and nnnn is the hard disk error status (in hexadecimal). If the unreadable sector does not correspond to an ISIS- II file, then the message

```
BAD SPOT AT LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

is displayed. Since ISIS-II allocates hard disk sectors serially, if no mechanism exited to "skip over" bad hard disk sectors, the remaining sectors would remain unallocated. Instead, ISIS-II checks hard disk sectors for irregularities during FORMAT and IDISK operations. If a bad sector is encountered, it is allocated to ISIS.BAD and hard disk formatting continues.

Example 1: This example shows the creation of a duplicate system disk excluding any non-system files on that disk.

```
—FORMAT :F1:IS00AS.SYS S
COPYING SYSTEM FILES
ISIS.T0
ISIS.BIN
ISIS.CLI
ASM80
ASM80.OV0
ASM80.OV1
ASM80.OV2
ASM80.OV3
ASM80.OV4
ASXREF
ATTRIB
BINOBJ
COPY
DELETE
DIR
EDIT
FIXMAP
FORMAT
HDCOPY
HEXOBJ
IDISK
LIB
LINK
LINK.OVL
LOCATE
OBJHEX
RENAME
SUBMIT
FPAL.LIB
PLM80.LIB
SYSTEM.LIB
```

Example 2: This example formats a basic non-system disk on drive :F1:, giving it the name of LIB.V1. System files are not copied.

```
—FORMAT :F1:LIB.V1
NON-SYSTEM DISK
—
```

Example 3: This example formats a hard disk in drive :F0: as a system disk; the files with the system or format attribute set are copied from a system disk in :F4:.

```
—:F4:FORMAT :F0:SYSTEM.DSK S FROM 4
```

FIXMAP - Map Bad Sectors on Hard Disk

Various hardware and software problems can cause a sector on a disk to become bad, or unreliable. The FORMAT and IDISK commands in ISIS-II recognize bad sectors and record the numbers of those sectors to prevent their allocation to files. Fatal errors and disk errors arising during the HDCOPY procedure can also show that sectors are bad; either of the following messages reports that track 137, sector 106 on drive 1 is bad:

```
STATUS=000F
D=1 T=137 S=106

DISK ERROR—UNABLE TO WRITE TO DESTINATION DISK ON DRIVE 1
LOGICAL ADDRESS (137, 106), STATUS=000F
```

The FIXMAP command records the presence of bad sectors reported in either form illustrated above. There is no corresponding command for flexible disks.

The syntax of the FIXMAP command is:

FIXMAP <drive>

where:

<drive> is the number of the hard disk unit on which the command is to operate. Unlike IDISK or FORMAT, FIXMAP requires a drive number, not a drive name, for <drive>.

When the command is entered, it displays the message:

ISIS-II MAP FIXER Vx.y.

where "x.y" is the version number of the FIXMAP program. In some cases, the following message is also displayed:

BAD SECTOR COUNT INCONSISTENCY

This message means that an earlier malfunction caused damage to a required ISIS format file. If you use FIXMAP to change the state (good-bad) of any sector, the inconsistency is resolved, but the disk remains unreliable.

FIXMAP Commands

You direct the operation of FIXMAP by entering commands from the following list. FIXMAP prompts with an asterisk whenever it is ready to accept a command.

Mark <disk address>	Change the known state of a sector from good to bad.
Free <disk address>	Change the known state of a sector from bad to good.
List [<filename>]	List all known bad sectors.
Count	List the number of known bad sectors.
Record	Record changes specified by Mark and Free.
Quit	Exit to ISIS-II without recording changes.
Exit	Record changes and exit to ISIS-II.

A command may be truncated at any point after its first character. For example, M, MA, or MAR may be used to stand for Mark.

When a command calls for a disk address, that address should have the form:

<track> <sector> [T]

where:

<track> is a number from 0 to 199 that specifies the track on which the bad sector exists.

<sector> is a number from 1 to 144 that specifies the number of the bad sector within the track.

[T] is an optional switch indicating that a group of 36 sectors should be processed.

The T switch is appropriate if the STATUS reported in the error message was 0001, 000A, or 000E. (See Appendix C for a description of status.)

If the T switch is present, the <sector> number specifies a group of 36 sectors on <track>:

If <sector> is in the range 1-36, that group of sectors is processed.

If <sector> is in the range 37-72, that group of sectors is processed.

If <sector> is in the range 73-108, that group of sectors is processed.

If <sector> is in the range 109-144, that group of sectors is processed.

Track and sector numbers, and the T switch, if present, should be separated by spaces. The track and sector numbers should be those reported in the error message that identified the bad sector.

Mark Command. The Mark command changes the known state of a sector from good to bad.

The syntax of the Mark command is:

```
MARK <disk address>
```

where:

<disk address> is the track-and-sector address of the sector to be marked as bad.

If the T switch is present, a group of 36 sectors is marked as bad. A sector known to be bad is not allocated to any file.

If the sector specified in the Mark command is not associated with an existing file, the sector is marked as bad. If the T switch is not present, the system displays:

```
SECTOR MARKED
```

If the T switch is present, no message appears when a single sector is marked; instead, when all 36 sectors have been processed, the system displays the message:

```
TRACK PROCESSED
```

If the sector belongs to an existing file, it cannot be marked as bad. Under any of the following conditions, the sector is not marked:

If the sector belongs to one of the required ISIS-II format files, the system displays:

```
(track, sector) REQUIRED BY ISIS-II
```

The system will be unreliable when the questionable disk is in use. You should format a new hard disk and copy your program and data files onto it.

If the sector is already known to be bad, marking the sector is redundant. The system displays:

```
(track, sector) ALREADY MARKED
```

If the sector belongs to a file other than a required format file, the system displays:

```
(track, sector) IN USE
```

If you know the name of the file, exit to ISIS-II and delete the file; then use FIXMAP to mark the bad sector. If you do not know the name of the file, follow this procedure:

1. Exit to ISIS-II, using either the Quit or the Exit command (described below).

2. Give the command

   COPY :Fm:*.* TO :Fn: Q C

   where "Fm" is the device name of the disk being fixed, and "Fn" is the name of another hard disk. Because of the Q switch, a list of copied files will be generated. (See the description of Copy.)

3a. *If the copy is successful,* disk :Fn: contains a usable copy of disk :Fm:. Use FIXMAP to get a list of bad sectors on :Fm:; then use IDISK or FORMAT to reformat that disk, and use FIXMAP to mark any bad sectors missed by the formatting command.

3b. *If an error occurs while the disk is being copied,* write down the last filename displayed by Copy, as well as the track and sector numbers appearing in the error message. Use the Delete command to delete the bad file from disk :Fm:; then use FIXMAP to mark the bad sector. If an error occurs and prevents you from deleting the file, repeat step 2. The file you attempted to delete will not be copied to the new disk. Repeat step 3a.

Example 1: The following example illustrates the use of the Mark command.

```
*MARK 27 83
 SECTOR MARKED
*MARK 27 83
 (27, 83) ALREADY MARKED
```

Free Command. The Free command changes the known state of a sector from bad to good.

The syntax of the Free command is:

```
FREE <disk address>
```

where:

<disk address> is the track-and-sector address of the sector to be freed for allocation.

If the T switch is present, a group of 36 sectors is freed. You might use this command if you had marked a sector by mistake.

If the sector specified in the Free command is known to be bad, it is freed for allocation. If the T switch is not present, the system displays:

```
SECTOR FREED
```

If the T switch is present, no message appears when a single sector is freed; instead, when all 36 sectors have been processed, the system displays the message:

```
TRACK PROCESSED
```

Under either of the following conditions, the sector is not freed:

If a sector is already free for allocation, freeing the sector is redundant. The system displays:

```
(track, sector) ALREADY FREE
```

If the sector is not free because it is in use by a file, the system displays:

```
(track, sector) NOT A BAD SECTOR
```

There is no reason to free a good sector that is part of an existing file.

Example 2: The following example illustrates the use of the Free command. Note that 8 5 T and 8 10 T identify the same group of 36 sectors, i.e., sectors 1-36 on track 8.

```
*FREE 180 51
 SECTOR FREED
*MARK 8 5 T
 TRACK PROCESSED
*FREE 8 10
 SECTOR FREED
*FREE 8 10 T
 (8, 10) ALREADY FREE
 TRACK PROCESSED
```

List Command. The List command writes a list of all known bad sectors on the named file.

The syntax of the List command is:

```
LIST [<filename>]
```

where:

[<filename>] is an optional parameter specifying the listing file.

The listing file may be either an output device or a disk file. It may not reside on the disk being fixed. If no filename is given, the list is printed on the console.

The format of the output is one sector per line, with track and sector numbers separated by a comma. The list includes all sectors marked by FIXMAP, as well as bad sectors found by IDISK and Format.

If there are no known bad sectors, the system displays:

```
NO BAD SECTORS
```

If output is directed to a device other than the console, the following message is displayed after the list is written to the device:

```
LIST WRITTEN
```

If the named file resides on the disk being fixed, the system displays:

```
CANNOT LIST TO TARGET DRIVE
```

Example 3: The following example illustrates the use of the List command. The list is written first to the console, then to a disk file.

```
*LIST
 180, 63
 182, 115
 182, 116
 182, 117
*LIST DISK.FIL
 LIST WRITTEN
```

Count Command. The Count command reports the number of known bad sectors on the disk.

The syntax of the Count command is:

```
COUNT
```

The command displays the following message on the console:

```
xxxxx BAD SECTORS
```

where "xxxxx" is a decimal number, the number of known bad sectors on the disk. A sector that has not been marked, or a sector that has been marked and then freed, is not a known bad sector.

Example 4: The following example illustrates the use of the Count command.

```
*LIST
 180, 63
 182, 115
 182, 116
 182, 117
*COUNT
 4 BAD SECTORS
```

Record Command. The Record command records the changes specified by Mark and Free.

The syntax of the Record command is:

```
RECORD
```

When this command is entered, changes specified by Mark and Free are recorded on the disk.

If you intend to use the Exit command to leave the FIXMAP program, the Record command is unnecessary. (Exit is described below.) If you intend to use the Quit command, the Record command is required; otherwise, none of the marking and freeing specified during the work session—or since the last Record command—will actually take effect.

When the recording is complete, the system displays:

```
CHANGES RECORDED
```

If no sector has been marked or freed during the work session—or since the last Record command—the system displays:

```
NO CHANGES
```

Example 5: The following example illustrates the use of the Record command.

```
*RECORD
 CHANGES RECORDED
*RECORD
 NO CHANGES
```

Quit Command. The Quit command stops the operation of FIXMAP and returns to ISIS-II.

The syntax of the Quit command is:

```
QUIT
```

If the Record command has not been given, changes specified by Mark and Free are not recorded on the disk.

Example 6: The following example illustrates the use of the Quit command. Note that the freeing of sector 12, 86 is not recorded on the disk; therefore, upon reentry to FIXMAP, that sector is still known as bad.

```
*FREE 12 86
 SECTOR FREED
*QUIT
-FIXMAP 1
ISIS-II MAP FIXER Vx.y
*LIST
 12, 86
```

Exit Command. The Exit command records changes and returns to ISIS-II.

The syntax of the Exit command is:

```
EXIT
```

The Exit command is equivalent to the Record command followed by the Quit command: changes specified by Mark and Free are recorded on the disk, and control returns to ISIS-II.

Example 7: The following example illustrates the use of the Exit command. (Compare this example with Example 6, above.)

```
*FREE 12 86
 SECTOR FREED
*EXIT
 CHANGES RECORDED
-FIXMAP 1
ISIS-II MAP FIXER Vx.y
*LIST
 NO BAD SECTORS
```

FIXMAP Errors

The following errors cause immediate termination of FIXMAP and a return to ISIS-II. If execution terminates as a result of one of these errors, work done since the last Record command is not recorded on the disk.

If no hard disk is present, the system displays:

```
USE ON HARD DISK SYSTEM ONLY
```

If no drive number is given in the FIXMAP command, or if an illegal switch is present, the system displays:

```
INVALID SYNTAX
```

If this message appears in response to a command within FIXMAP, it means that the command was typed incorrectly, and it does not terminate the session.

If the specified drive number is greater than 3, the system displays:

```
DRIVE NUMBER OUT OF RANGE
```

(In the maximum configuration of the system, the hard disk drives are numbered 0 and 1.)

If the disk does not exist in the system, is not on-line, or is not properly connected, the system displays:

```
ERROR 30 USER PC xxxx
```

where "xxxx" is a hexadecimal number.

Example 8: The following example illustrates a typical work session with FIXMAP. You invoke the command and begin by getting a list of all bad sectors on the target drive (drive 1, as indicated in the FIXMAP command). The Count command reports that there are eight bad sectors, and the Record command shows that no sectors have been marked or freed during this work session. You free the last 36 sectors on the track containing track 170, sector 113; all sectors except the eight known bad sectors are reported already to be free. You mark track 170, sector 113 as a bad sector, and again list and count the number of bad sectors. This time, the Record command reports that changes have been made. You free the remaining bad sector, list again, and return to ISIS-II.

```
-FIXMAP 1
ISIS-II MAP FIXER V1.0
*LIST
 170, 113
 170, 114
 170, 115
 170, 116
 170, 117
 170, 118
 170, 119
 170, 120
*COUNT
 8 BAD SECTORS
*RECORD
 NO CHANGES
```

```
*FREE 170 113 T
 (170, 109) ALREADY FREE
 (170, 110) ALREADY FREE
 (170, 111) ALREADY FREE
 (170, 112) ALREADY FREE
 (170, 121) ALREADY FREE
 (170, 122) ALREADY FREE
 (170, 123) ALREADY FREE
 (170, 124) ALREADY FREE
 (170, 125) ALREADY FREE
 (170, 126) ALREADY FREE
 (170, 127) ALREADY FREE
 (170, 128) ALREADY FREE
 (170, 129) ALREADY FREE
 (170, 130) ALREADY FREE
 (170, 131) ALREADY FREE
 (170, 132) ALREADY FREE
 (170, 133) ALREADY FREE
 (170, 134) ALREADY FREE
 (170, 135) ALREADY FREE
 (170, 136) ALREADY FREE
 (170, 137) ALREADY FREE
 (170, 138) ALREADY FREE
 (170, 139) ALREADY FREE
 (170, 140) ALREADY FREE
 (170, 141) ALREADY FREE
 (170, 142) ALREADY FREE
 (170, 143) ALREADY FREE
 (170, 144) ALREADY FREE
 TRACK PROCESSED
*MARK 170 113
 SECTOR MARKED
*LIST
 170, 113
*COUNT
 1 BAD SECTOR
*RECORD
 CHANGES RECORDED
*FREE 170 113
 SECTOR FREED
*LIST
 NO BAD SECTORS
*EXIT
 CHANGES RECORDED
```

Program Execution Commands

You can call a program for direct execution in which case you must respond to any queries from the program and to any errors encountered during program execution. You can also call for execution of the program under the Monitor in which case the debugging provisions of the Monitor aid you in identifying and locating program errors. A third method is to submit the program as a job to be handled by the system without any interaction on your part. In this latter case you must prepare a file that interacts witht he program in the same manner as you would during direct execution of the program.

Filename - Direct Program Execution

As ISIS-II commands except DEBUG are actually the names of disk files containing executable programs. Thus the execution of any other program is accomplished in the same manner—simply by entering the name of the file. You may also include parameters with the filename to provide control over the program to be executed. However, the program must be written to accept these parameters and must read the parameters from the line editing buffer. Refer to Chapter 5 for more information on line editing of command lines.

DEBUG - Program Execution Under Monitor

The DEBUG command loads an executable program and passes control to the Monitor, which outputs the contents of the program counter and prompts for a command with a period (.) on the system console. The Monitor G command can then be entered to begin execution of the program. Any errors occurring during execution are handled as described in Chapter 6 of this manual.

The syntax of the DEBUG command is:

DEBUG[<progname>[<parameters>]]

where

<progname> is any ISIS-II command or in broader terms the file name of an executable program. The program must be an absolute object module. If <progname> is omitted, control transfers to the Monitor, but no program is loaded.

<parameters> are the normal parameters of the program to be executed.

A starting address (entry point address) and up to two breakpoint addresses can be specified in the Monitor's G command (see chapter 6). When execution of your program is suspended at the breakpoint address, you can use Monitor commands to inspect and/or change the contents of memory and/or registers and then continue program execution from the point of suspension with another G command.

If an ISIS-II error occurs after control has been passed to the Monitor, any attempt to resume execution with a G command may cause unpredictable results, possibly overwriting ISIS-II or causing spurious input/output to the disk. It is acceptable to examine memory and/or registers to debug an error, but it is not wise to restart the program until control has been returned to ISIS-II.

You can return to ISIS-II from the debug mode and reset the debug toggle in one of the following ways:

- Enter Monitor command G8.
- Execute an EXIT system call in the program being debugged.
- Press Interrupt 1.

Example 1 : This example shows how to execute a program named LIST in debug mode. The source of LIST is not shown, but its load address is 3680H.

```
—DEBUG LIST FILE.TXT
#3680

.G
```

(The LIST program is executed.)

—

Example 2 : This example executes the same program in debug mode, suspends execution at the specified breakpoint address and then returns to ISIS-II with a G8 command instead of letting the program issue an EXIT system call.

```
—DEBUG LIST FILE.TXT
#3680

.G,-36A0
```

(Use Monitor commands to examine registers and memory when the breakpoint 36A0 is reached.)

```
.G8

ISIS-II, Vm.n
—
```

Example 3 : This example allows you to transfer to Monitor control with no program loaded. Return to ISIS-II is by the Monitor G command with no address.

```
—DEBUG
#0008
```

SUBMIT - Non-Interactive Program Execution

The SUBMIT command causes ISIS-II to take its commands from a disk file rather than the console. Before using SUBMIT you create a command sequence definition file with optional formal (or symbolic) parameters. SUBMIT replaces these formal parameters with actual values that you specify in the SUBMIT command. The command sequence definition file thus establishes the order in which the commands are executed, whereas the SUBMIT command determines precisely what command syntax and constants are to be employed.

The syntax of the SUBMIT command is:

SUBMIT <name>[.<extension>][(<parameter0>,<parameter1>,...,<parameter9>)]

where

<name>[.<extension>] specifies the file containing the command sequence definition. If extension is omitted, SUBMIT assumes the file <name>.CSD. The command sequence definition, which must be in a disk file, is the command sequence with formal parameters, that are explained below.

<parameter*n*> specifies the actual values that are to replace the formal parameters in the command sequence definition. The maximum number of parameters is 10. If a parameter is omitted from the SUBMIT list, its position must be indicated by a comma. A parameter is a character string of 31 characters maximum. Any ASCII character from 20H to 7AH is legal, except a comma, space, or right parenthesis. If a parameter contains a comma, space, or right parenthesis, the parameter must be enclosed in quotation marks. To use a quotation mark inside a quoted parameter, use two quotation marks in its place. For example:

'TITLE(''QUOTE ('') SEARCH ROUTINE'')'

is used in the final command as:

TITLE('QUOTE (') SEARCH ROUTINE')

There are two files that SUBMIT uses. One contains the command sequence definition and is created by you with formal parameters. The other contains the command sequence to be executed and is created by SUBMIT with actual parameters instead of formal parameters. The command sequence file has the same name as the command sequence definition file but with the extension CS. The command sequence file should not be modified by the user.

SUBMIT reassigns the console input device to the command sequence file it has created and returns control to ISIS-II, which then executes the commands in the command sequence file. The last command in the command sequence file is a special SUBMIT command (provided automatically by SUBMIT) that restores the console input device to its former device assignment and deletes the command sequence file.

Formal parameters are specified in the command sequence definition by two characters, %n, where n is a digit from 0 through 9. They may appear anywhere in the command sequence definition. The literal character, control-P (↑P), can precede a percent sign (%) that is not to be interpreted as a formal parameter.

Any program that reads its commands from :CI: can be executed noninteractively. The command sequence definition file can also contain commands to the programs being run. If a SUBMIT command is used in a command sequence definition file, it causes another command sequence file to be created. This 'nesting' of SUBMIT commands can be repeated to any depth.

A control/E in a SUBMIT Switches the console input from the command sequence file to the initial system console, allowing interactive processing. A ↑E entered at the initial system console returns control to the command sequence file. If control is *not* returned to the command sequence file, or if an error occurs after a command sequence has started processing, control returns to ISIS-II and the CS file is not deleted.

Any program running under SUBMIT must allow for one buffer in addition to the open files and buffers required by the program itself. See Chapter 4 for information on how to determine the base address of your program.

Example 1 : The following examples show a PL/M compilation executed noninteractively on a 4-drive system. The PLM80 command has only three items that change. Using SUBMIT to enter the command automates the process, saving you keystrokes at the console.

The command sequence definition is in the file PLM80.CSD. See the compiler operator's manual listed in the preface for an explanation of the controls in the PLM80 command. The file PLM80.CSD contains the following:

```
PLM80 :F1:%0.%1 DEBUG XREF PRINT (:F3:%0.LST) DATE(%2)
```

This command sequence definition contains three formal parameters, indicated by %0, %1, and %2. The SUBMIT command used to start the compilation is as follows:

```
SUBMIT PLM80(PROGA,SRC,'9 SEPT 78')
```

The command sequence created by SUBMIT and then executed is as follows:

```
—PLM80 :F1:PROGA.SRC DEBUG XREF PRINT (:F3:PROGA.LST) DATE(9 SEPT 78)

ISIS-II PL/M-80 COMPILER, V3.1
PL/M-80 COMPILATION COMPLETE  0 PROGRAM ERROR(S)

—:F0:SUBMIT RESTORE PLM80.CS(:VI:)
—
```

Example 2 : This example shows a PL/M compilation, a LINK, and a LOCATE executed from a SUBMIT file on a 2 flexible disk drive system. A ↑E is entered in the command sequence definition after the PL/M compilation so the compiler disk can be removed. The operator enters a ↑E when the regular system disk (with LINK and LOCATE) is mounted and processing can continue. The text editor does not echo the ↑E; however, it is echoed when the SUBMIT file is executed.

The file CMPLNK.CSD in drive 1 contains the following command sequence definition:

```
PLM80 %0.%1 DEBUG XREF DATE(%2)
↑E
LINK %0.OBJ,SYSTEM.LIB TO %0.SAT&
PRINT(%0.LNK) MAP
LOCATE %0.SAT PRINT(%0.LOC) MAP
```

The SUBMIT command entered to compile, link, and locate PROGA.SRC follows:

```
SUBMIT :F1:CMPLNK (:F1:PROGA,SRC, '3 OCT 78')
```

The command sequence actually executed is shown as it would be echoed on the console output device:

```
—PLM80 :F1:PROGA.SRC DEBUG XREF DATE(3 OCT 78)

ISIS-II PL/M-80 COMPILER V3.1
PL/M-80 COMPILATION COMPLETE 0 PROGRAM ERROR(S)

—↑E↑E
—LINK :F1:PROGA.OBJ,SYSTEM.LIB TO :F1:PROGA.SAT &
**PRINT(:F1:PROGA.LNK) MAP
—LOCATE :F1:PROGA.SAT PRINT(:F1:PROGA.LOC) MAP
—:F0:SUBMIT RESTORE :F1:CMPLNK.CS(:VI:)
—
```

File Control Commands

The file control commands are those commands that work directly with files and are not concerned with the type of information within the files. This is in contrast with the other ISIS-II console commands that initialize new disks or deal with program files specifically.

The file control commands provide for copying or deletion of files and revision or display of file directory contents. Familiarization with the file control commands is equivalent to learning a primary function of ISIS-II.

DIR - Disk Directory Listing

The DIR command lists one or more directory entries of the disk in a specific drive, sending the list to the console output device or to a list file. The syntax of the DIR command is:

DIR [FOR <file>][TO <listfile>][<switch>]

The positions of these fields are not fixed.

where

<file> is the file or group of files (specified with the wild card construction) whose directory entry is to be listed. If FOR <file> is omitted, the entire directory is listed. If <file> is not a wild card name (that is, does not contain * or ?) it is listed even if it has the invisible attribute.

<listfile> is the name of the file to contain the directory listing. If TO <listfile> is omitted, the listing is sent to the console output device.

<switch> can be one or more of the following:

0-9 Lists the directory of the disk in :F0:, :F1:, :F2:, ... :F9:. If omitted, the directory of the disk in :F0: is listed. More than one drive number can be specified but only the rightmost one has effect. The drive number also overrides any device specification in FOR <file>.

I Lists all files, including files with the invisible attribute set. If omitted, only files with invisible attribute not set are listed.

F Gives fast output, listing only name.ext of files.

O Prints directory in a single column format. The default is double column format.

Z Prints the number of sectors presently used on the specified disk as a fraction of the number of available sectors.

P After loading the command, the system pauses with the message:

```
LOAD SOURCE DISK, TYPE (CR)
```

After the disk is loaded and CR pressed, the requested directory is output to the console output device. The system will then request that the system disk be replaced:

```
LOAD SYSTEM DISK, TYPE (CR)
```

The DIR default is the directory output in two columns with the following headings:

```
DIRECTORY OF name.ext
NAME      .EXT    BLKS  LENGTH ATTR        NAME        .EXT       BLKS LENGTH ATTR
PROGA     .HEX      75  9263      W        SUMS                     51 6357
                                       126
936/2002 BLOCKS USED
```

where

name.ext is the label of the disk volume that is assigned by the FORMAT or IDISK command. It has the same syntax as a filename. Each item listed by DIR is explained in the section "Disk Directory Content" in this chapter. The directory listing shows the number of blocks in use and the total number of blocks within the disk (2002, 4004, or 28800).

Example 1 :

The following example lists two files of a flexible disk on a single density system. The system files, which have the invisible attribute set, are not listed.

```
—DIR
DIRECTORY OF :F0:IS00AB.SYS
NAME      .EXT    BLKS  LENGTH ATTR       NAME       .EXT       BLKS LENGTH ATTR
PROGA     .HEX      75  9263       W      SUMS                    51 6357
                                    126
936/2002 BLOCKS USED
```

Example 2 : This is the same as Example 1 except a fast listing is requested.

```
—DIR F
DIRECTORY OF :F0:IS00AB.SYS
PROGA .HEX              SUMS
936/2002 BLOCKS USED
```

Example 3 : This example requests a directory listing of all format files be sent to the line printer. The format files have the invisible attribute and ISIS.* is a wild card filename, so the I switch must be specified.

```
—DIR I FOR ISIS.* TO :LP:
```

Example 4 : A single-column fast directory listing of the double density flexible disk in drive :F1: is requested by the following command.

```
—DIR 1 F O
DIRECTORY OF DISK :F1:ISIS.V10
TYPE.M80
TYPE.HEX
TYPE
1337/4004 BLOCKS USED
```

COPY - Copy a File

The COPY command copies the contents of a file into another file. The source and destination can be disk files or physical devices. The file from which the copy is made must be an input device and file to which the copy is directed must be an output device. For example, you can copy from the reader to the punch but not from the punch to the reader.

When the source or target is a disk file, the filename must be specified or implied. When the source or target is a physical device, only the device designation need be specified.

Wild card characters can be used in disk filenames. However, the wild card characters cannot be used in device designations (you cannot specify :F*:).

The COPY command supports single disk drive systems. You can copy files from one disk to another using only a single drive. The command prompts for the source, output, and system disks as it needs them. If you specify a copy on a drive with no change in file name, the command assumes you want to swap disks and prompts for the swaps. For example, the command:

```
COPY ABC TO ABC
```

results in prompts to swap disks in drive :F0:. But the commands:

```
COPY ABC TO :F1:ABC
```

and

```
COPY ABC TO DEF
```

do not result in prompts for disk swapping. You can also copy files between different disks on the same drive by specifying the P (pause) switch in the command.

When you punch a file on paper tape, a string of nulls is added to the file. If the punched file is then read back into the system and compared to the original, the nulls will cause a false compare. These nulls may be stripped from the file with the ISIS-II Text Editor or the CREDIT CRT-Based Text Editor. Object files should be converted to hexadecimal format before punching on paper tape.

The syntax of the COPY command is:

COPY <infile1>[,<infile2>,...,<infile*n*>] TO <outfile>[<switches>]

where

<infile> is a file on any device capable of input. The contents of <infile> are not affected by the copy. If more than one <infile> is specified, they are concatenated in the order specified when copied to <outfile>. When concatenating files, you must specify the full name and extension of each file.

Wildcard designations may not be used when concatenating files. The following example gives the error message that will be encountered if this is done.

```
COPY A, BC.* TO D
WILDCARD DELIMITERS DURING CONCATENATE
```

When using the concatenate operation, the destination file name cannot be equal to any source file name. The following example indicates the error message which results if this is attempted.

```
COPY A,B TO B
SOURCE FILE EQUALS OUTPUT FILE ERROR
```

<outfile> is the name of the file to be created or recreated or the name of an output device. <outfile> is specified in the form :device:name.extension. :device: is any system device. If :device: is omitted, and a filename is specified, :device: defaults to :F0:. If :device: is not a disk drive, name and extension are ignored. If <outfile> is an existing disk file and is not write protected, the following message appears on the console:

```
<outfile> FILE ALREADY EXISTS
DELETE?
```

If you respond to the message with 'yes' or just 'y' (followed by a carriage return), COPY deletes the existing file before making the copy. No action is performed if any other response is given.

If <outfile> is write protected, then the following message is output:

```
<outfile>, WRITE PROTECTED
```

Wildcard designations can be used in <outfile> only to the extent they are used in <infile>. That is, they must follow these rules:

- Every position in the <infile> name that contains an * must have a corresponding * in the <outfile> name.
- Every position in the <infile> name that contains a ? must have corresponding ? or * in the <outfile> name.

The COPY command provides a special case for convenience. When copying disk files to a different disk, you can specify as <outfile> a device designation without a filename. In this case, <outfile> will have the same name as <infile>. For example:

```
COPY :F1:ABC.XYZ TO :F2:
```

is the same as specifying:

```
COPY :F1:ABC.XYZ TO :F2:ABC.XYZ
```

This form can be used with wildcard designations in <infile>:

```
COPY :F1:*.* TO :F2:
```

At the end of the listing of files that were copied the message

```
WRITE PROTECTED FILE ENCOUNTERED
```

will be displayed if necessary.

If the rules governing wildcard designations are not followed, an error message will be displayed as shown by the example below.

```
—COPY ABC.* TO D
FILE MASK ERROR
```

COPY command switches allow you to limit wild card copies to system files or non-system files, copy attributes with the file, suppress messages when a file is to replace an existing file, and to receive a prompt that requires a yes response before a copy is performed.

<switches> are one or more of the following:

U U is the update switch, which if used causes the suppression of the "ALREADY EXISTS" message described earlier. <outfile> is opened for update instead of being deleted. The length is not changed unless the copy causes an increase in the size of the file. <outfile> has attributes unchanged unless C is specified, in which case the attributes of <outfile> will be the attributes set before the copy plus the attributes set in the file being copied from. For example, if file XYZ with the I attribute set is copied to the file ABC which has the S attribute set, the final file ABC will have the S and I attributes set.

S S is the system switch, which causes only files with the system attribute to be copied. For example, the command:

```
COPY :F0:*.* TO :F1:*.* S
```

copies only files with the system attribute from drive :F0: to drive :F1:.

N N is the nonsystem switch and causes only files without the system or format attribute to be copied.

P P is the pause switch, which allows files to be copied between two disks on the same drive. The system prompts for disk swaps with the following messages:

```
LOAD SOURCE DISK, THEN TYPE (CR)
LOAD OUTPUT DISK, THEN TYPE (CR)
LOAD SYSTEM DISK, THEN TYPE (CR)
```

Q Q is the query switch, which causes the system to display the following message before a copy is performed: COPY <infile> TO <outfile>? A yes or y response causes the copy to be performed. Any other response causes the copy not to be performed.

C C is the attribute switch, which causes <outfile> to be created with the same attributes as <infile>. If this switch is not specified, <outfile> is created with all attributes reset (off). This switch does not copy the format (F) attribute.

B B is the brief switch, which causes an existing file to be deleted without displaying the "ALREADY EXISTS" prompt. The existing file is deleted and recreated with new data unless the U switch is also specified, in which case, the existing file is overwritten. This switch can significantly improve performance, and should be used, even if it is known that the destination file does not exist.

Example 1: This example copies three files to one, overwriting its contents.

```
—COPY CHAP1,CHAP2,CHAP3 TO BOOK
:F0: BOOK FILE ALREADY EXISTS
DELETE? Y
APPENDED :F0:CHAP1 TO :F0:BOOK
APPENDED :F0:CHAP2 TO :F0:BOOK
APPENDED :F0:CHAP3 TO :F0:BOOK
```

Example 2: Example 1 could have been done in the following way.

```
—COPY CHAP1,CHAP2,CHAP3 TO BOOK U
APPENDED :F0:CHAP1 TO :F0:BOOK
APPENDED :F0:CHAP2 TO :F0:BOOK
APPENDED :F0:CHAP3 TO :F0:BOOK
—
```

Example 3: This example lists a file on the line printer.

```
—COPY BOOK TO :LP:
COPIED :F0:BOOK TO :LP:
—
```

Example 4: This example displays a file on the console output device.

```
—COPY CHAP1 TO :CO:
   (text of CHAP1)
COPIED :F0:CHAP1 TO :CO:
—
```

Example 5: This example copies a file from the disk in :F0: to the disk in :F1:.

```
—COPY PROGA TO :F1:NEWPRG  B
COPIED :F0:PROGA TO :F1:NEWPRG
—
```

Example 6: This example copies system files from one disk to another on drive :F0:

```
—COPY *.* TO *.* P S
LOAD SOURCE DISK, THEN TYPE (CR)
LOAD OUTPUT DISK, THEN TYPE (CR)
COPIED :F0:ASM80 TO :F0:ASM80
.
.
.
LOAD SYSTEM DISK, THEN TYPE (CR)
```

If the files to be copied are quite large (exceeding the size of the available RAM) the LOAD SOURCE and LOAD OUTPUT messages will be displayed more than once. As each file is copied, a COPIED message is displayed. After the last file is copied, the LOAD SYSTEM message is displayed.

Example 7: This example show valid uses of wildcard names with the COPY command:

```
COPY :F1:*.* TO :F2:              (copy all files except those with the
                                   FORMAT attribute)
COPY :F1:A??C TO :F0:D??E
COPY :F1:*.* TO :F3: N            (copy all non-system and
                                   non-format files)
COPY :F1:A????? TO :F0:B*.CPY
```

HDCOPY - Copy Hard Disk Tracks

The HDCOPY command copies the contents of one hard disk to another hard disk on a track-by-track basis. The data transformed are verified during reading of the data into memory. HDCOPY formats the destination disk before writing data to it. The BACKUP switch can be used to backup a removable hard disk platter. The syntax of HDCOPY is:

```
HDCOPY <drive 1> TO <drive2>|BACKUP
```

where

<drive1> is the number of the drive containing the source hard disk.

<drive2> is the number of the drive containing the destination hard disk.

The drive numbers must be 0 or 1. The source and destination cannot be the same drive number, or a fatal error results, and an error message is displayed. If either <drive1> or <drive2> is not a hard disk drive, an error message is displayed:

```
SPECIFIED DRIVES NOT HARD DISK
```

If you specify BACKUP, these actions occur:

- The contents of the disk in drive 1 are copied to the disk in drive 0
- ISIS-II prompts for the backup disk to be placed in drive 1
- The contents of the disk in drive 0 are copied to the disk in drive 1
- ISIS-II prompts for a system disk to be placed in drive 1
- The contents of the disk in drive 1 are copied to the disk in drive 0

If drives 0 and 1 are not hard disk platters, an error message appears.

If a disk error is detected while reading from the source hard disk, the message

```
DISK ERROR—UNABLE TO READ FROM SOURCE DISK ON DRIVE M
     LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

appears on the console device, and a sector of 0FFH is written to the destination disk. If an error is detected when reading from or writing to the destination disk, the message

```
DISK ERROR—UNABLE TO WRITE TO DESTINATION DISK ON DRIVE N
     LOGICAL ADDRESS (ttt, sss), STATUS = nnnn
```

appears on the console device. In both cases, processing continues. You must decide whether or not to use the destination disk or to attempt to make a new copy.

In these error messages, ttt is the logical track address (in decimal); sss is the logical sector address (in decimal); and nnnn is the hard disk error status (in hexadecimal) as described in Appendix C.

As each track is copied to the destination disk, a "T" is printed on the console device. If no errors have been detected during a copy operation, the message

```
VERIFICATION OK
```

appears on the console device when the operation concludes. If errors have been detected, the message

```
BAD SECTORS ENCOUNTERED
```

appears on the console device.

Example 1: A sample HDCOPY command:

```
—HDCOPY 1 TO 0

LOAD DISK(S), THEN TYPE (CR)

DRIVE 1 DISK NAME MYDISK.123 IS SOURCE DISK
DRIVE 0 DISK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
DRIVE 1 DISK NAME MYDISK.123 COPIED TO DRIVE 0
VERIFICATION OK
HARD DISK COPY COMPLETED
—
```

Example 2: A sample HDCOPY command with the BACKUP switch:

```
—HDCOPY BACKUP

LOAD DISK IN DRIVE 1, THEN TYPE (CR)

DRIVE 1 DISK NAME MYDISK.123 IS SOURCE DISK
DRIVE 0 DISK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT TTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
DRIVE 1 DISK NAME MYDISK.123 COPIED TO DRIVE 0

LOAD BACKUP DISK IN DRIVE 1, THEN TYPE (CR)

DRIVE 0 DISK NAME MYDISK.123 IS SOURCE DISK
DRIVE 1 DISK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
DRIVE 0 DISK NAME MYDISK.123 COPIED TO DRIVE 1

LOAD SYSTEM DISK IN DRIVE 1, THEN TYPE (CR)

DRIVE 1 DISK NAME IS00AT.SYS IS SOURCE DISK
DRIVE 0 DISK WILL BE OVER-WRITTEN
OK TO CONTINUE?
y
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
DRIVE 1 DISK NAME IS00AT.SYS COPIED TO DRIVE 0
VERIFICATION OK
HARD DISK COPY COMPLETED
```

DELETE - Delete a Disk File

The DELETE command provides for erasure of specified directory entries. This command effectively removes the specified file or group of files from a disk, making the space it occupied available to ISIS-II for reassignment. A file with the write-protect or format attribute set cannot be deleted. The syntax of the DELETE command is:

```
DELETE <filename1>[Q][,<filename2>[Q]][,<filename3>[Q]][,...] [P]
```

where

<filename> is the name of a file to be deleted. The wild card construction can be used to delete a group of files.

If <filename> is a file with neither the write-protect nor format attribute set, the file is deleted and a confirming message is sent to the console.

If <filename> does not exist, the following message is sent to the console where <filename> is that specified in the DELETE command:

```
filename, NO SUCH FILE
```

If the file cannot be deleted because it has the write-protect or format attributes set, the following message is sent to the console.

```
filename, WRITE PROTECTED
```

Each file name specified in the DELETE command may be followed by a space and a Q in which case the user is asked to confirm the file deletion. This query mode operation is most useful when the wild card construction is employed. You will then receive the following message for each of the files where name conforms to the wild card construct:

```
filename, DELETE?
```

Type a Y or y for a positive answer. Any other response causes DELETE to go on to the next file in the group without deleting the specified file.

If you delete files from a disk other than the system disk in a single drive system, you can specify the P (for pause) switch with the command. Before the deletion is performed, the system displays:

```
LOAD SOURCE DISK, THEN TYPE (CR)
```

When the deletion is completed, the following message is displayed:

```
:Fn: <filename>, DELETED
LOAD SYSTEM DISK, THEN TYPE (CR)
—
```

Example 1: This example deletes three files.

```
—DELETE CHAP?.*
:F0:CHAP1.TXT, DELETED
:F0:CHAP2.LST, DELETED
:F0:CHAP3.SRC, DELETED
—
```

Example 2: This example shows an attempt to delete a write-protected file.

```
—DELETE PROGA.ASM

:F0:PROGA.ASM, WRITE PROTECTED
—
```

Example 3: This example shows the deletion of a file using the P switch.

```
—DELETE PROGB.ASM P
LOAD SOURCE DISK, THEN TYPE (CR)
:F0:PROGB.ASM, DELETED
LOAD SYSTEM DISK, THEN TYPE (CR)
—
```

RENAME - Rename a Disk File

The RENAME command changes the name of a disk file. Only the directory is affected. The syntax of the RENAME command is:

```
RENAME <oldname> TO <newname>
```

where

<oldname> is the name of the existing file whose write-protect or format attribute is not set. If a nonexistent file is specified, an error occurs.

<newname> is the new name to be assigned to the file with <oldname>. An error occurs if the device part of the filename is not the same as that in <oldname>. If a file with <newname> already exists, this message appears on the console:

```
newname, ALREADY EXISTS, DELETE?
```

If you respond to the message with Y or y (followed by a carriage return), RENAME deletes the existing file before renaming. (If the existing file is write-protected, it is not deleted and the file to be renamed is not renamed.) No action is performed and control returns to ISIS-II if a response other than Y or y is given.

RENAME cannot be used on nonsystem disks on a single drive system. To change the name of a nonsystem disk file with only a single drive, use the COPY command to copy the file to a file with the new name, then delete the old file with the DELETE command.

Example 1: The following example changes the name of the file CHAP1.

```
—RENAME CHAP1 TO CHAP.ONE

—
```

Example 2: This example shows an attempt to rename a write-protected file.

```
—RENAME NEWPRG.TXT TO PROGA.TXT

NEWPRG.TXT, WRITE PROTECTED

—
```

Example 3: In this example, the new name is the name of an existent file.

```
—RENAME TEXT.BAK TO TEXT.OLD

TEXT.OLD, ALREADY EXISTS, DELETE? Y

—
```

ATTRIB - Change/Display Disk File Attributes

The ATTRIB command changes and/or displays the specified attributes of a disk file. The syntax of the ATTRIB command is:

ATTRIB <file> [<attriblist>] [Q]

where

<file> is a disk file whose attributes are to be changed. The wild card construction can be used to change and/or display the attributes of a group of files.

<attriblist> is one or more of the following:

I0 or I1	Reset or set the invisible attribute. When set, the file is not listed by the DIR command unless the I switch is specified in the DIR command.
W0 or W1	Reset or set the write-protect attribute. When set, the file cannot be opened for output or update, and cannot be deleted or renamed.
F0 or F1	Reset or set the format attribute. When set, the file is copied to the disk being formatted by the IDISK and FORMAT commands. Removal of the format attribute from system files will cause improper formatting of new system disks. This attribute is reserved for specific system files and should not be assigned to any other file.
S0 or S1	Reset or set the system attribute. When set, the file is copied to the disk being formatted by the FORMAT command when the S switch is used. This file is also copied by the COPY command when the S switch and wild card notation are used.

If two values of the same attribute are specified, for example both I0 and I1, the one rightmost in the command takes precedence.

Q requests query mode operation. Before changing the attributes of a file, ATTRIB sends the following message to the console:

```
filename, MODIFY ATTRIBUTES?
```

Type a 'yes' or just 'y' if you want the file attributes modified. Any other response causes ATTRIB to leave the attributes unchanged for the specified file and to go on to the next file in the group.

If a nonexistent disk file is specified, ATTRIB responds with a message on the console:

```
filename, NO SUCH FILE
```

If a non-disk file is specified, ATTRIB responds with a message on the console:

```
filename, NON-DISK DEVICE
```

where filename is that specified in the command. When attributes for a file have been changed, the current attributes for the file are displayed on the console.

Example 1 : This example changes the write-protect attribute of a group of files:

```
—ATTRIB PROGA.* W1
         FILE              CURRENT ATTRIBUTES
    :F0:PROGA. SRC                W
    :F0:PROGA. OBJ                W
—
```

Example 2 : This example sets the system attribute for the TYPE program so it will be transferred onto new system disks (see FORMAT command).

```
—ATTRIB TYPE S1
       FILE                CURRENT ATTRIBUTES
:F0:TYPE                          S
—
```

Code Conversion Commands

The code conversion commands exist for two reasons:

- To provide compatibility with systems employing hexadecimal object file format.
- To convert programs created under previous versions of ISIS.

The programs called by the code conversion commands convert the character coding of these disk files but do not otherwise alter the information of the original programs. The program files of ISIS-I were created on single density disks and must therefore be input via a single density drive.

BINOBJ - Convert Binary to Absolute Object Module

The BINOBJ command converts the absolute binary format of earlier versions of ISIS to the object module format of ISIS-II. The syntax of the BINOBJ command is:

BINOBJ <binfile> TO <absfile>

where

<binfile> is the file containing the absolute binary format of ISIS versions 1.0 through 1.2.

<absfile> is the file to contain the absolute object module that can be executed under ISIS-II. The starting address (address of first instruction to be executed) is taken from <binfile>. The module name stored with the object module is taken from the name part of the <absfile> filename. The absolute object module produced by BINOBJ does not contain a symbol table.

Example 1 : The following example converts an executable program from ISIS version 1.2 (and earlier) format to an ISIS-II executable absolute object module.

```
—BINOBJ PROGA.BIN TO PROGA.OBJ

—
```

HEXOBJ - Convert Hexadecimal Code to Absolute Object Code

The HEXOBJ command converts object code in hexadecimal format to an absolute object module compatible with ISIS-II. The hexadecimal format is produced by cross-product translators that run on large machines and by assemblers of earlier versions of ISIS. The format of the HEXOBJ command is:

HEXOBJ <hexfile> TO <absfile>[START(addr)]

where

<hexfile> is a file of machine object code in hexadecimal format.

<absfile> is the output file from HEXOBJ containing the absolute object module that can be loaded for execution under ISIS-II. The module name stored with the object module is the same as the name part of the <absfile> filename. The absolute object module produced by HEXOBJ contains a symbol table if symbols were defined.

START(addr) is used to include a starting address (the address of the first instruction to be executed) in the absolute object module. The address can be specified by a hexadecimal, decimal, octal, or binary number followed by a letter indicating the base. The letter is H for hexadecimal, O or Q for octal, B for binary, D or omitted for decimal. Thus the address 4000 hexadecimal is specified as START(4000H).

If START(addr) is omitted, the starting address is taken from the end-of-file record of the hexadecimal format file, which is determined by the END assembly language statement.

If no starting address is specified in any of the above ways, zero is assumed, which is in the ISIS-II area and causes an error 15 on any attempt to load the file.

Example 1 : In this example a hexadecimal format object file on paper tape is converted to an absolute object file on disk.

```
—HEXOBJ :HR: TO SUMS

—
```

Example 2 : This example converts a hex file to absolute object format, recording a starting address in the object module.

```
—HEXOBJ PRIME.HEX TO PRIME.OBJ START(3200H)

—
```

OBJHEX - Convert ISIS-II Absolute Object Code to Hexadecimal

The OBJHEX command converts an ISIS-II absolute object module to hexadecimal format.

The format of the OBJHEX command is:

OBJHEX <absfile> TO <hexfile>

where

<absfile> is the file containing an ISIS-II absolute object module.

<hexfile> is the file to contain the hexadecimal object code converted from the ISIS- II format. The starting address (address of first instruction to be executed) is taken from <absfile>. The hexadecimal object code produced by OBJHEX does not contain a symbol table.

Using The Text Editor

The ISIS-II Text Editor enables you to create and update ASCII text files. The Editor can be used to manipulate text on a line or character basis. One or more characters can be inserted in, deleted from, or changed in a line of text. Insertions and deletions can be made that cover one or more lines. The point of insertion or deletion can be freely selected to be at the beginning or end of the file, beginning or end of a line, or at any point within a line.

The ISIS text editor is a contextual editor. That is, it does not use line numbers but uses the context of the file for editing. For example, to insert some text before a line that contains "LXI QQ", you don't need a line number. You can find the line by issuing a Find command specifying the content of the line.

The usual procedure for creating a new text file is to call the editor, enter text from the system console, perform whatever editing functions are desired, and then output the file to a disk for storage. To edit an existing file, call the editor, input the file from a disk, edit the file, and output the modified version to the disk.

The Editor Functions

The Editor has 16 commands that allow full control over your text file. The commands are divided into the following categories:

- Commands to input and output data and terminate a session
- Commands to manipulate the text pointer
- Commands to modify the text
- Command to type the text
- Command to check available memory

The input, output, and session commands let you access a disk file, make changes, and save the results. You can also terminate a session without saving the work.

The text pointer manipulation commands let you move around in the file to make changes or insertions at the desired place. The text pointer is always pointing to some location in the file. Before you can make a change or insertion, you must move the pointer to where you want it. The text pointer is described in more detail later.

The text modification commands let you insert data, delete data, and change data.

The type command lets you display your file on the Console output device.

The memory check command lets you determine how much space you have left in memory to add data.

Command Entry

When you are using the Editor, an asterisk (*) is displayed at the left margin when the Editor is ready to accept a command. If there is not an asterisk at the left margin either of two conditions exist; 1) you are inserting data with the Insert command in which case your command will be treated as data and entered into the file, or 2) the Editor is still processing the last command you entered.

The commands are all single letter commands and must be followed by two escape (ESC) or alternate mode (ALT MODE) characters, depending on the terminal you are using. In the command syntax statements, these characters are shown as two dollar signs ($$). Thus, a simple Editor command will look like:

```
*B$$
```

(This particular command moves the text pointer to the top of the file.)

Command Parameters

There are two types of parameters that can be included with Editor commands, iteration parameters and data parameters.

Iteration parameters specify how many times the command is to be performed. For example, the Type command (which is t) types one line. To type 10 lines, the command is:

```
*10T$$
```

The iteration parameter always comes before the command code. If it follows the command, it is ignored or assumed to be part of the following command in a string. (Command strings are described later.)

Data parameters specify the data on which the command is to operate. For example, the Substitute command (which is s) replaces one data string with another. To change the data string "CALL LP1" to "JMP ERR1," the command is:

```
*SCALL LP1$JMP ERR1$$
```

The data parameter always comes after the command code. If it precedes the command, the Editor does not know what to do with it and may treat it as a command if the first letter is a valid command code.

The data parameter example also shows another use of the ESC or ALT MODE character, as a separator. It will be used, as shown above, to separate multiple data fields in a command and will be used to separate commands in a command string.

Aborting Commands

A command or command string can be aborted completely, before it is terminated ($$) by entering Control/C. Control/C is entered by pressing the letter c (upper or lower case) while the CTRL key is depressed.

Control/C can also be used to stop execution of a command in progress. For example, if you entered a command to type 100 lines of the file when you meant to type 10 lines, you can stop the typing by entering Control/C. The typing will stop immediately, and the Editor will prompt for the next command.

Correcting Typing Errors

You can correct typing errors in command strings or text by pressing the RUBOUT key once for each character to be deleted and then retype the correct characters. As you press the RUBOUT key, the character that is deleted is displayed on the screen. You can only make corrections of this kind in commands before entering the command terminator ($$).

A line with several corrections made with the RUBOUT key is unreadable. You can type the line as really exists by entering Control/R. Control/R causes the system to type the whole line with the corrections made. For example:

```
Tiihoois aais a kkluuindde.

Control/R

This is a line.
```

When a line has so many errors that you want to start over, you can enter Control/X which cancels that entire line. When Control/X is entered, the system types a crosshatch character (#) and starts at the beginning of the next line.

Formatting Characters

To make your source code listings usable, carriage returns and line feeds are essential and horizonal TABs are helpful.

Carriage Return and Line Feed Characters

When entering data at the system console device, it is not necessary to insert line feed characters. Each time the carriage return key is pressed, the system generates a line feed character and stores it in memory following the carriage return character.

When you use the RUBOUT key to delete a carriage return, you must rubout the carriage return and the line feed. That is, you must press the RUBOUT key twice.

TAB Characters

TAB characters make it easy to line up the columns of a source code listing. You can enter a TAB with Control/I. The TABs defined in the system are every eight character positions, 8, 16, 24, etc. When Control/I is entered the carriage advances to the next TAB position.

(If you are coding in PL/M, please note that PL/M interprets a TAB character as four spaces, not the eight shown on the screen during entry with the Editor.)

Activating The Editor

The Editor is activated with the ISIS-II EDIT command. The EDIT command cannot be executed from a SUBMIT file.

The syntax of the EDIT command is:

EDIT <filename1> [TO <filename2>]

where

<filename1> is either the name of a new file to be created and edited or the name of an existing file to be edited.

<filename2> is the name of the device or disk file to which the edited version of <filename1> is written.

If <filename1> is a disk file and <filename2> is not specified, the action taken by the Editor depends on whether <filename1> is a new or existing file. If it is new, it is created for output. If it exists and is not write-protected, it is opened for input. A temporary file, EDIT.TMP, is created for the output. When the session is ended with the Exit (E) command, the files are closed and <filename1> is renamed "name.BAK" and EDIT.TMP is renamed <filename1>.

If <filename1> is not a disk file, it must be an input file and <filename2> must be specified and must be a disk file. For example,

```
EDIT :HR: TO :F1:ASSY.SRC
```

reads the text from the high speed paper tape reader into the system in response to the Append (A) command. When the editing is complete, the contents of the file are written to a disk in drive :F1: with the name ASSY.SRC.

You can start entering data with the Insert (I) command or you can read an existing file from disk with the Append (A) command. The commands are described in the next section.

Editor Commands

The Editor commands are all single character commands. Many can have iterative and/or data parameters. As you use the commands you will find that you often enter a series of commands one after the other. This takes a lot of extra key strokes (command terminators at the end of each command) and time.

You can enter multiple commands on the same line. When you do this the commands are executed one at a time from left to right, exactly the same way they are executed if you enter each on a separate line.

A single ESC or ALT MODE character can be used at any point between commands as a separator character. When data is included with a command you must separate the data from the next command with an ESC or ALT MODE character. If you don't, the Editor has no way of knowing when the data stops and the next command begins. Without the separator character, everything is considered data. The ESC or ALT MODE character is also used a separator between multiple data fields in a command.

The Text Pointer

The Editor keeps a pointer into the data in a file at all times. The pointer is always pointing between characters. When many of the Editor commands are issued, the action takes place from where the pointer is located. For example, the Type (T) command types the data from the pointer location to the end of a line. The Insert (I) command inserts data into a file at the pointer location.

To update a file you must be able to move the pointer in the file. The Editor has five pointer manipulation commands:

- Beginning of Text (B) which moves the pointer before the first character in the file.
- End of Text (Z) which moves the pointer after the last character in the file.
- Line (L) which moves the pointer before the first character in a line.
- Character (C) which moves the pointer one character in the file.
- Find (F) which moves the pointer after the last character of a data string supplied in the command.

B - Beginning of Text Command

The B command moves the pointer before the first character in the file. When you read a file, the pointer starts at this position.

The B command is useful to:

- Move the pointer to the beginning to type a whole file.
- Establish a starting point when searching a whole file to find a certain character string.
- Inserting text before the existing text.

The format of the B command is:

B$$

where:

B is the command code.

B Command Examples: To move the pointer to the beginning of a file:

```
*B$$
```

If you want to type the first line, you can add the Type (T) command (described later) to the B command:

```
*BT$$
```

The pointer is moved to the beginning and the first line is typed on the Console output device.

Z - End of Text Command

The Z command moves the pointer immediately following the last character in the file. The Z command is useful to add text to the end of a file.

The format of the Z command is:

Z$$

where:

Z is the command code.

Z Command Examples: To move the pointer to the end of the file:

```
*Z$$
```

When the pointer is at the end of the file, there is no data following it. If you issue a command such as Type (T), nothing is typed because there is no data to type.

L - Line Command

The L command moves the pointer before the first character of the next line in the file. You can specify an iterative parameter with the L command to move the pointer several lines. The parameter must precede the command code. You can move the pointer backwards, to the beginning of lines by entering a minus parameter. You can move the pointer to the beginning of the current line by entering a parameter of zero. If no parameter is included, 1 is assumed.

The format of the L command is:

[n]L$$

where:

n specifies the number of lines the pointer is to be moved. A positive n specifies a forward movement and a negative n specifies a backward movement. 0 specifies that the pointer be moved to the beginning of the current line. If n is omitted, 1 is assumed.

L is the command code.

L Command Examples: To move the pointer to the beginning of the next line:

```
*L$$
```

To move the pointer to the beginning of the line five lines back in the file:

```
*-5L$$
```

To move the pointer to the beginning of the current line:

```
*0L$$
```

To move the pointer ahead 12 lines:

```
*12L$$
```

C - Character Command

The C command moves the pointer a specified number of characters forward or backward in the file. You can specify an iterative parameter with the C command. The parameter must precede the command code. A positive number (an unsigned number is assumed to be positive) moves the pointer forward the specified number of characters. A minus number moves the pointer backward. If no parameter is included, 1 is assumed. The parameter can be any number from -65,535 to +65,534. If the beginning or end of the file is encountered before the specified number is reached, the pointer stops at the beginning or end.

The format of the C command is:

[n]C$$

where:

n specifies the number of characters the pointer is to be moved. A positive n specifies a forward movement and a negative n specifies a backward movement. If n is omitted, 1 is assumed.

C is the command code.

C Command Examples: To move the pointer to the next character in the file:

```
*C$$
```

To move the pointer ahead 10 characters:

```
*10C$$
```

To move the pointer backward 100 characters:

```
*-100C$$
```

F - Find Command

The F command searches the file for a character string specified in the command. The search begins at the current location of the pointer and continues until the string is found or the end of the file is encountered.

If the character string is found, the pointer is moved immediately following the last character of the string.

If the character string is not found, the pointer is not moved from its current location and following message is typed:

```
CANNOT FIND "text"
*BREAK*
```

Where text is the character string specified in the command.

All characters must match for a match, including printing and non-printing characters (such as carriage return and line feed).

If the F command is included in a command string, an ESC or ALT MODE character must be used to terminate the character string or the following command will be used as part of the search argument.

The format of the F command is:

Ftext$$

where:

F is the command code.

text is the character string being searched for. The Editor only uses the first 16 characters in the string. Any characters beyond 16 are ignored.

F Command Examples: To find the character string "LOOP1" starting from the current location of the pointer:

```
*FLOOP1$$
```

To find the same character string but start the search from the beginning of the file:

```
*BFLOOP1$$
```

To find the same character string starting from the beginning of the file and then move the pointer to the beginning of the line containing the string:

```
*BFLOOP1$0L$$
```

In the last example, if the ESC or ALT MODE character after the string had not been entered, the Editor would have searched for the string "LOOP10L" and would not have found it.

Text Commands

The Editor contains commands to enter new text into a file, change existing text, delete an entire line, and delete a single character.

All of these commands affect the pointer. The effect on the pointer is described in each command description.

I - Insert Command

The I command inserts text into the file immediately before the pointer. All data following the I command is inserted into the file until a command terminator is entered. This includes carriage returns and line feeds.

The pointer is always immediately following the inserted data.

During initial entry of data into a file, the I command normally goes on for many lines, possibly for the entry of an entire program.

The format of the I command is:

Itext$$

where:

I is the command code.

text is the data to be entered. Any amount of text can be entered, from a single character to an entire file. The Editor keeps accepting data and inserting it into the file until the $$ is encountered.

I Command Examples: To enter a single character into the file where the pointer is located:

```
*IK$$
```

To enter a line to the beginning of the file:

```
*BIThis is the new line<CR>
$$
```

To enter extensive text at the end of an existing file:

```
*ZIThis is the start of the text<CR>
More text<CR>
.
.
.
Last line of text.<CR>
$$
```

S - Substitute Text Command

The S command finds a character string and substitutes another character string for it. The substitution is made only if the search results in an exact match.

After a successful match, the pointer is located immediately following the inserted data.

The S command starts its search from the current location of the pointer and continues through the file until an exact match is found or until the end of the file is encountered.

If a match is not found in the search, the following message is displayed:

```
CANNOT FIND "text"
*BREAK*
```

Where text is the search argument.

There is no limit to the amount of data that can be inserted with the S command. However, the search argument is limited to 16 characters. If more characters are entered for the search argument, only the first 16 are used and only the first 16 are replaced with the new text.

When using the S command, remember that it searches to the end of the file. If you make a typing error in the search argument and that exact string exists in the file, it will be changed by the S command. It is a good idea to type the line just changed until you become experienced with the command.

The format of the S command is:

Sold-text[$new-text]$$

where:

S is the command code.

old-text is the character string to be searched for and to be replaced if found. Only the first 16 characters of old-text is used.

new-text is the character string that is to replace old-text. If new-text is omitted, the character string specified by old-text is deleted.

S Command Examples. To replace the string "JMP" with the string "CALL":

```
*SJMP$CALL$$
```

To make the same substitution but start the search from the beginning of the file:

```
*BSJMP$CALL$$
```

To make the same substitution, searching from the beginning and then typing the entire line in which the change was make:

```
*BSJMP$CALL$0LT$$
```

To delete the first occurance of the string "START" from a file:

```
*SSTART$$
```

In the above example, no replacement data was supplied with the command. Thus, the command found the string and replaced it with nothing (deleted it).

D - Delete Command

The D command deletes the number of characters, starting at the pointer, specified in the command. If the iterative parameter is positive (or unsigned) the deletions are done from the pointer forward. If the parameter is negative, the deletions are done from the pointer backward. If the deletion is performed in a forward direction in the file, the pointer immediately precedes the first character not deleted. If the deletion is performed in a backward direction, the pointer immediately follows the first character not deleted. In other words, the pointer is between the remaining characters on either side of the deletion.

The format of the D command is:

[n]D$$

where:

n specifies the number of characters to be deleted. If n is positive, the deletion is performed in a forward direction and if n is negative the deletion is performed in a backward direction from the pointer. If n is omitted, 1 is assumed.

D is the command code.

D Command Examples. To delete the next 10 characters following the current pointer location:

```
*10D$$
```

To delete the 10 characters preceding the current pointer location:

```
*-10D$$
```

K - Kill Line Command

The K command deletes all the characters in a line beginning at the current location of the pointer. The characters in the line that precede the pointer are not deleted. The K commands also deletes the carriage return and line feed at the end of the line.

An iterative parameter can be specified with the command. If the parameter is positive (or unsigned) the command deletes the remaining portion of the line containing the pointer and and the following lines until the command parameter is met or the end of the file is encountered. If the parameter is negative, the command starts at the pointer and deletes backward for the number of lines specified.

The format of the K command is:

[n]K$$

where:

n specifies the number of lines to be deleted. If n is positive the deletion is performed in a forward direction, and if it is negative the deletion is performed in a backward direction. If n is omitted, 1 is assumed.

K is the command code.

K Command Examples. To delete an entire line, you must make sure the pointer is at the beginning of the line:

```
*0LK$$
```

To delete all the characters in a line from the pointer through the carriage return and line feed characters:

```
*K$$
```

To delete the entire line in which the pointer is located and the following three lines:

```
*0L4K$$
```

Typing a File

You can display your file with the Type command. If you have a teletypewriter terminal, the command types the text.

T - Type Command

The T command displays a line on the Console output device. You can include an iterative parameter to display multiple lines. The parameter can be positive or negative. If the parameter is positive, the Editor starts displaying at the pointer and continues for the specified number of lines. If the parameter is negative, the Editor displays the required number of lines that precede the pointer. A parameter of zero displays from the beginning of a line to the pointer location. The T command without a parameter displays from the pointer through the carriage return and the line feed.

The T command does not move the pointer.

The format of the T command is:

[n]T$$

where:

n specifies the number of lines to be displayed. If n is positive the display starts at the line pointer and proceeds forward in the file. If n is negative, the display starts n lines before the pointer and displays to the pointer. If n is zero, the display starts from the beginning of the current line and proceeds to the pointer. If n is omitted, 1 is assumed.

T is the command code.

T Command Examples. To display a line from the pointer through the end of the line:

```
*T$$
```

To display from the beginning of a line to the location of the pointer:

```
*0T$$
```

To display the entire line no matter where the pointer is located within the line:

```
*0TT$$
```

In the preceding example, the "0T" displays from the beginning of the line to the pointer and the second "T" displays from the pointer to the end of the line.

To move the pointer to the beginning of a line and then display the line:

```
*0LT$$
```

To display the previous line and move the pointer to the beginning of that line:

```
*-1LT$$ or just *-LT$$
```

To display your entire file but you don't know how many lines are in it but know it can't be more than 500:

```
*B500T$$
```

When the end of the file is reached the command stops.

Terminating a Session and Saving Your File

There are two commands to exit the Editor and return to ISIS-II. One command saves your work from the editing session on disk and the other doesn't.

E - Exit Command

The E command saves the entire contents of the file in memory on disk. If the file being edited is not completly read in from the disk, the E command saves the contents of the file in memory and then reads the remainder of the file from the disk and immediately writes it back to the disk. When the entire file is saved, the Editor returns control to ISIS-II.

The format of the E command is:

E$$

where:

E is the command code.

E Command Example. To save the contents of the text file on disk and exit the Editor:

```
*E$$
```

Q - Quit Command

The Q command causes an exit from the Editor and returns control to ISIS-II without saving the data in memory.

The format of the Q command is:

Q$$

where:

Q is the command code.

Q Command Example. To exit the Editor and return to ISIS-II:

```
*Q$$
```

W - Write Command

The W command takes n lines from the beginning of the memory buffer and writes them onto the disk. The lines that are output are deleted from memory. This prevents the duplication of these lines when an E command is issued.

A common usage of the W command is to store part of a very large program while still entering the end portions. Once data has been written out to disk using the W command, it can only be brought back into memory by issuing an EXIT command and then an EDIT command for the file.

The format of the W command is:

[n]W$$

where:

n specifies the number of lines to be output and deleted from memory. If n exceeds the number of lines in the file, the entire file is output. If n is omitted, 1 is assumed.

W is the command code.

W Command Example. To save the first 25 lines of the file in memory on disk:

```
*25W$$
```

Reading Data from Disk

The Editor provides a single command to read data from disk into memory.

A - Append Command

The A command reads a maximum of 50 lines of text from disk into memory. The text is appended to the bottom of the data already in memory (data that was read in with the A command or entered via the Editor commands).

The A command reads 50 lines of text or until:

- The end of the file is reached.
- An end-of-file character (Control/Z) is read. The Control/Z is not read into memory.
- The memory assigned to the Editor is full.
- A form feed character (Control/L) is read. The Control/L is read into memory.

If you have a large file to read into memory, you can issue repeated A comands. An iterative parameter is ignored for the A command.

The format of the A command is:

A$$

where:

A is the command code.

A Command Examples. To read 50 lines of text from disk into memory:

```
*A$$
```

To read 500 lines of text from disk into memory:

```
*AAAAAAAAAA$$
```

Please note that the command "10a" will only read 50 lines of text. The iterative parameter is ignored. (See the section "Command String Iterations" later in this chapter for an alternative way to specify multiple A commands.)

Determining Memory Space Available

The Editor has a command to determine how much memory space is still available for text entry.

M - Memory Command

The M command computes and displays on the system Console output device, the number of bytes of memory still available. The information is supplied in the following message:

```
nnnn-CHARACTER(S) AVAILABLE IN WORKSPACE
```

where nnnn is a decimal integer.

If you are entering a large amount of data, you can check on the amount of storage available and, if it runs short, save the first part of the file on diskette with the W command.

The format of the M command is:

M$$

where:

M is the command code.

M Command Examples. To check on the amount of storage available:

```
*M$$
```

Command String Iterations

You can repeat a command string or a single command any number of times by enclosing the command string in angle brackets (< >), preceded by a number that specifies the number of times the command string is to be performed.

A typical use of this capability is to meet the need to change a character string that exists throughout the file. For example, if you have a program that uses a data field named "X1" and you want to change all occurances of the name to something more meaningful to anyone else reading the code. To change "X1" to "LOOPCNT" through out the file you can use the following:

```
*B100<SX1$LOOPCNT$>$$
```

In this example, note that a B command was used first to move the pointer to the top of the file. Also note that an ESC or ALT MODE character was used to separate LOOPCNT from the final angle bracket. This prevents the Editor from using the angle bracket as part of LOOPCNT.

In the description of the A command it was pointed out that an iterative parameter cannot be used with A command. In an example there, we used a string of 10 A commands to read 500 lines from paper tape. Using this facility, the 10 A commands can be entered as follows:

```
*10<A>$$
```

This is equivalent to:

```
*AAAAAAAAAA$$
```

while:

```
*10A$$
```

is only equivalent to:

```
*A$$
```

Command string iterations can be nested. For example, the command:

```
*100<3C2>5CD<L>$$
```

will go through a 100 line file. On each line it will advance the pointer 3 characters and then, twice, it will advance five more characters and delete a character. The inner string advances the pointer five characters and deletes the sixth character. The outer string advances the pointer three characters before executing the inner command string and then advances the pointer to the next line.

Iteration command strings can be nested eight deep. If more than eight strings are nested, the following error message is issued and the command is not executed:

```
ITERATION STACK FAULT
  *BREAK*
```

CHAPTER 4 WORKING WITH PROGRAM MODULES

The creation of a program module is a fairly straightforward process once you have fully defined the module interfaces. However, the existence of modules would be of little value if you did not have the software tools necessary for manipulation of the modules to build your entire program. Three basic tools are required:

- A linking program that combines many modules into a single module.
- A locating program that assigns absolute memory addresses to a relocatable module.
- A library management program that permits you to add modules to a library whose contents may be accessed when linking.

These capabilities are provided by three commands LINK, LOCATE, and LIB; a standard object code format; two resident compilers, PLM80 and FORT80; and a resident assembler, ASM80.

NOTE

The location and linkage features defined in this chapter are not applicable to the object code of the 8048 microprocessor. For information about MCS-86 object module management, see *MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users*.

The object code format supported by ISIS-II is a relocatable format produced by PLM80, FORT80, and ASM80. ISIS-I formats must be converted to the new format for use with ISIS-II. See the HEXOBJ and BINOBJ command descriptions in Chapter 3.

The LINK program combines files containing object modules of a program into one module in one object file, adjusting the relative addresses in the process. When the modules to be combined reference each other, you must identify these references as public symbols and external symbols thus allowing LINK to satisfy the external references of each module.

The relative addresses assigned to relocatable modules are converted to absolute addresses by the LOCATE program, which produces an absolute object file (or module that can be loaded by ISIS-II for execution, debugging, or memory mapping).

The LIB program creates and maintains libraries of object modules that can be used as building blocks to create new programs via LINK.

The many reasons for relocating a program and for writing programs in modules are discussed in the following text.

Microprocessor Memory Allocation

The microprocessor memory for any given application is generally not of uniform composition. The memory is usually tailored, with RAM (read-write memory) installed for variable data and ROM (read only memory) or PROM (programmable read only memory) installed for program code. Memory chips can be installed in such a way that some addresses have no corresponding physical memory.

This diversity in memory design (as well as programming considerations explained later) requires you to tailor your program, specifying absolute addresses for the actual ROM and RAM locations used for code and data. However, you may not know the absolute addresses of the ROM or RAM in your final application at the time you start to develop the program. At an early stage of development when you are testing your program in the system, your only concern with locating the program is that it doesn't overlap the resident routines of ISIS-II. The system memory is simply a sequence of RAM locations from 0 to 32K, 48K, or 64K. However, as your program development continues and debugging becomes a prime concern, address requirements are much more specific. Code may be in PROM starting at location 0; variable data may begin in the first location of a block of RAM.

The program that had a base address of 4000H for compatibility with ISIS-II resident software and had variable data immediately following code, may need new addresses to meet the requirements of the memory in the final application. In a system that deals only with absolute code, you would have to change the source program to specify new addresses and translate again to get object code with correct addresses. In the relocation and linkage system of ISIS-II, you simply produce a new absolute memory image from the relocatable object modules by using the LOCATE program. LOCATE assigns addresses to place code in ROM or PROM, variable data in RAM, and the stack in another area of RAM.

Program Segments

The LOCATE program allows the user to preassign areas of memory because of the way the language translators (PLM80, FORT80, and ASM80) divide a relocatable object module into segments. The segments are:

- Code
- Data
- Stack
- Memory
- Named and unnamed common segments (FORT80 only)

Each segment starts with a relative address of zero. A base address can be specified for each segment when it is being LOCATEd. The base address is the actual address of the first memory location the segment will occupy. LOCATE adds the base address to each relative address and adjusts address references accordingly.

Code Segment

The code segment is that part of the program destined for ROM because it contains machine instructions and program constants that are never modified during execution. This segment can also be placed in RAM. In assembly language it consists of statements following a CSEG directive.

Data Segment

The data segment is that part of the program that usually requires RAM. It contains variable data and storage for I/O buffers. In assembly language it consists of statements following a DSEG directive.

Stack Segment

The stack segment is for the program stack and must be in RAM. Usually only a module that contains a main routine has a reference to a stack segment. Its length is determined by the compiler for PL/M programs and by the STKLN statement in assembly language. You can also specify the length when the program is made absolute by LOCATE. References to the stack segment are made with STACK (a reserved word in assembly language) and STACKPTR (an identifier in PL/M).

Memory Segment

The memory segment is assigned to RAM memory that is not allocated to code, data, common, or stack segments. References to the memory segment are made with MEMORY (a reserved word in assembly language and an identifier in PL/M). Although the language translators create a memory segment for each relocatable object module, its length is unknown until an absolute module is produced by LOCATE, which uses the Monitor MEMCK routine and the base address of the memory segment to calculate the length of available RAM.

Common Segments

The common segments are used for FORT80 named COMMON areas and the BLANK COMMON area. Common segments usually contain variables, and therefore, are usually placed in RAM.

Absolute Information

In addition to the relocatable code, data, stack, common, and memory segments, an object module can contain information with absolute addresses already assigned. There are three ways this can happen. The ASEG statement in assembly language causes statements following it (until a CSEG or DSEG is encountered) to have absolute addresses. Absolute modules produced by LOCATE can be linked with relocatable modules. PL/M variables declared with the AT attribute produce absolute references.

It is possible to write a self-contained program in assembly language using the ASEG statement. This results in an absolute module that can be executed directly after assembling because the assembler output is a memory image. This approach is possible only in cases where relocation is not needed.

Modular Program Development

Most programs are too lengthy or too complex to code as a straightline program. You can make the job simpler by designing first a main routine that calls separate functions in subprograms. The exchange of parameters with these subprograms can be determined during the design of the main routine. However, the actual coding of the subprograms can be left until later. The final complete program is built from the main routine and the subprograms by the LINK program. The advantages of this approach to program design are summarized in the following paragraphs.

Faster Program Development

Your program can be developed faster if the modular approach is used because small modules are easier to understand and simpler to program. Breaking a program up into functional modules makes it possible to assign the pieces to a team of developers. With the specific tasks well defined, you can concentrate on programming and testing a specific module. You can supply the input required by the module and verify it by examining the output. As the modules are debugged, they can be put in a library file using the LIB program. When all modules are ready, the LINK program is used to combine the modules into one complete module, which can then be assigned absolute memory addresses by LOCATE.

Use of Different Source Languages

The modules that make up the final program need not be translated from the same language. PL/M, FORTRAN, or assembly language can be used, whichever suits the task best. Relocatable modules produced by the compiler, the assembler, or both can be input to LINK to build a program.

Shared Subprograms

When modules of one program have been tested they can be used by other programs. This means that part of the job of future programming is already done. Because the module has relocatable addresses, it can be combined with different programs, having different address requirements each time.

Easier Debugging and Program Modification

It is easier to narrow down the location of a bug when a program is divided into modules. When you have identified the module containing the error, you can concentrate on debugging that module. When a program must be modified, it may mean that only one or two modules must be changed or added. When the new or changed modules have been translated and debugged, a new absolute memory image can be created simply by using LINK and LOCATE rather than retranslating the entire source program.

Mechanics Of Relocation And Linkage

LINK is able to combine modules, adjusting relative addresses, and LOCATE is able to convert relative addresses to absolute addresses because of information put in the object modules by the translators, ASM80, PLM80, and FORT80. The following types of information in the object module are used by LINK and LOCATE:

- Relative addresses of instructions and data.
- A list of address fields in instructions or data that refer to a location in the same segment (intra-segment references).
- A list of address fields in instructions or data that refer to locations in other segments in the same module (inter-segment references).
- A list of address fields in instructions or data that refer to locations not contained in the same module (external references).
- A list of symbols in the module that are declared public or external in the source code.

You should understand external references and declaring symbols public and external to successfully use LINK and LOCATE. However, an understanding of the other topics in this section is not as important. It is provided here to give a complete picture of the mechanics of relocation and linkage.

Relative Addressing

The relative addresses of instructions and data in the code and data segments are assigned by ASM80 and PLM80 when a source module is translated. The addresses are determined by a location counter starting with zero at the beginning and incremented by the number of bytes in each instruction. The addresses are "relative" to the beginning of the segment.

LINK combines input modules to form one output module by combining all code segments into one code segment and all data segments into one data segment. The relative addresses of the first segment remain unchanged, but LINK changes the relative addresses of the segments that follow to reflect their relationship to the beginning of the new segment. In general this means adding the length of the first segment to the relative addresses of the second, adding the combined length of the first two segments to the relative addresses of the third segment, and so on.

LOCATE produces an absolute module from a relocatable one by adding the base address of each segment to each relative address in that segment to get the absolute address. The base address of each segment can be specified in the LOCATE command or left for LOCATE to assign.

Intrasegment References

In addition to relocating load addresses, relative addresses contained in instructions or data items must be adjusted. If the address refers to a location in the same segment, it is called an intrasegment reference. A jump instruction that refers back several instructions to the beginning of a loop is this kind of reference. The value put in the address field by the translator is simply the relative or absolute address of the location referred to. If it is a relative address, it is adjusted by LINK when segments are combined and finalized by LOCATE by adding the base address of the segment.

Intersegment References

When an address in an instruction refers to a location in another segment of the same module, it is called an intersegment reference. An example is an instruction in the code segment that refers to a variable in the data segment.

When LINK combines segments to produce a new object module, intersegment references are changed to reflect the new relative addresses of the locations in the other segments.

LOCATE converts the relative address of an intersegment reference to an absolute address by adding it to the base address of the segment to which reference is made.

External References and Public Symbols

When an address field in an instruction refers to a location not contained in the same module, it is called an external reference. This reference differs from those above because the translator knows nothing about the relative location of this symbol. Therefore, you must declare these symbols external. They then become known as external symbols, which means they are defined in other modules. The instructions that refer to them are external references.

The module containing an external reference is said to be an "unsatisfied" module or is said to contain an "unsatisfied" external reference. LINK combines this module with the module that contains the proper "connector". This connector is a public symbol that matches the external symbol. A public symbol is a symbol declared to be public in the source module and put in the object code with its address by the translator.

When LINK "connects" two modules by matching an external symbol to a public symbol, the value of the public symbol (its relative or absolute address) goes in the address field of the instruction that refers to it. Then LINK removes the external references. It is replaced with an intersegment or intrasegment reference if the public symbol has a relative address. If the public symbol has an absolute address, nothing replaces the external reference because no further address adjustment is required.

If the module that LINK produces contains unsatisfied external names, LINK issues a warning message about each one. This module must be linked again with modules containing the matching public symbols in order to produce a satisfied module. The unsatisfied external name messages from LINK do not indicate that an error exists. In intermediate steps of development before all modules of the program are complete, you can expect LINK to produce these messages. Also, if you have declared a name external but never make a reference to it in your program, LINK produces an unsatisfied external name message even though no unsatisfied external reference exists.

This points up the fact that you should have some way of identifying the state of the object code in a file. Saving the memory maps from a LINK and LOCATE is one way of keeping track; using an extension in the filename that reflects the type of contents is another way.

When a module contains no external references, it is said to be "satisfied". The public symbols are not removed from the object module because they may be needed later if a new module is added that has an external reference to one of the public symbols.

If LOCATE finds an external reference in an object module it is processing, it issues a warning message but continues to produce the absolute module. The absolute module can be executed, perhaps in debug mode with a breakpoint specified to stop processing before the instruction containing the unsatisfied external reference is reached. If that instruction is executed, results are unpredictable because the address in the instruction is undefined.

Use of Libraries

Libraries make your job of building programs from object modules via the LINK program even easier. The library manager program LIB creates and maintains files containing object modules. LIB creates a directory of the modules in each library file to keep track of the modules it contains.

The LINK program treats library files in a special way. If you specify a library file as input to LINK after specifying the modules to be included, LINK searches the library for modules that contain public symbols to match the unresolved external references in the preceding modules. If a module from the library is included but it also has unresolved external references, LINK searches the library again trying to find the module with the public symbols to satisfy the new external references. This process is repeated until a search has been made to satisfy all external references.

The library manager program can give you a list of the modules in a library file, including the public symbols in each module. You may want to keep all the object modules for one program in a library, or you may want to keep modules relating to a specific function in one library file. For example, a system library supplied with ISIS-II is called SYSTEM.LIB and contains the object modules for linking programs to ISIS-II system routines.

Link Command

The ISIS-II program LINK allows you to combine object modules from several input files into one object module in one output file. In the process of combining modules LINK adjusts all addresses so they are relative to the beginning of the segments in the output module. LINK also searches libraries for modules that resolve external references in the modules being combined and includes them in the output module. If any unresolved external references remain in the output module, LINK puts a message in the map that describes the structure of the new module.

The output module must be processed by LOCATE before it can be executed. The LOCATE program assigns absolute memory locations to the object module. The output module can also be used as input to LINK to be combined with other modules into a new and expanded output module.

The LINK program is called into operation by the LINK command. The syntax of the LINK command is:

LINK <inputlist> TO <outputfile> [<controls>]

where

<inputlist> can be either or both of the following two items:

<filename> [(<modname1>,<modname2>,...,<modname*n*>)]

PUBLICS(<filename>,<filename2>,...,<filename*n*>)

In the first item <filename> specifies a file containing object modules or a file containing a library of object modules. If <filename> is not a library file, it is included in the output module. If <filename> is a library file and <modnames> are specified, then only the specified modules are linked into the output module. If <modnames> are omitted and <filename> is a library, only those library modules that satisfy external references in modules already named in the <inputlist> or already included from the library are linked into the output module. In other words, when <modnames> is omitted, only those library modules that satisfy an existing unresolved reference are included.

The second item PUBLICS specifies modules whose absolute public declarations only are to be included in the output module. This allows for linking modules without combining them in the output file so they can be loaded separately. See the section on overlays in this chapter for a description of this capability.

<outputfile> specifies the file to contain the object module resulting from linking the input modules. This file must not also be specified in the input list.

<controls> are one or more keywords that control the operation of LINK. The controls are:

MAP — This control requests that a link map be produced. The link map is sent to the console output device (:CO:) or to the file specified in the PRINT control. The content of the link map is described later in this chapter.

NAME(<modname>) This control specifies the name to be assigned to the output module. The name can be from 1 through 31 characters, each of which must be a letter (A through Z), a digit (0 through 9), a question mark (?), or a commercial at sign (@). However, the first character of the name cannot be be a digit. If NAME(modname) is not specified, the name part of the output file specification is used as the module name.

PRINT(<filename>) This control specifies the file to contain the link map. If omitted, the link map goes to the console output device (:CO:).

If a LINK command is longer than one line on your console (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and the associated parameter list, for example:

```
PRINT&
(:F1:PRTFIL)
```

is a valid use of the continuation character. LINK prompts for the continued line with a double asterisk (**). If necessary, subsequent lines can be continued also.

WARNING

LINK uses a temporary file named LINK.TMP on the disk to which the output is directed. If you have a file by this name on the output disk, it will be destroyed.

"Appendix C: Error Messages" lists the LINK error messages.

Link Map

The link map produced by LINK includes the following information:

- The LINK sign-on message.
- The command used to call LINK (unless map is output to :CO:).
- The length of the relocatable segments in the output module.
- The addresses of absolute information in the output module.
- The names of the input modules.
- Unresolved external names.
- Non-fatal error messages.

The following example shows a link map for the output module contained in the file SHIP.WRK, which was produced from two modules explicitly listed in the input list, a module included from searching a library to resolve external references, and the public symbols of a fourth module.

The REL column in the LINK map specifies the segment relocation type. The following abbreviations are used in the REL column:

B byte relocatable

P page relocatable

I in-page relocatable

A absolute or non-relocatable

```
ISIS-II OBJECT LINKER Vx.y INVOKED BY:
—LINK FILNAM.EXT,PROG.LIB(TRIG),PROG.LIB,&
**PUBLICS(TANSTA.AFL) TO SHIP.WRK NAME(CELESTIGATION) MAP
```

(Non-fatal error messages appear here.)

```
LINK MAP OF MODULE CELESTIGATION
WRITTEN TO FILE :F0:SHIP.WRK
MODULE IS NOT A MAIN MODULE

SEGMENT INFORMATION:
  START      STOP      LENGTH      REL   NAME

                        2345H       P    CODE
  10BCH      10FFH        44H            CODE        *GAP*
  22FEH      22FFH         2H            CODE        *GAP*
                         107H       B    DATA
                          75H       B    /FRED/
                        10F2H       B    //
  0000H      0002H         3H       A    ABSOLUTE
  0040H      0711H       6D2H       A    ABSOLUTE
  0700H      07FFH       100H       A    ABSOLUTE    *OVERLAP*
  FD00H      FD00H         1H       A    ABSOLUTE

INPUT MODULES INCLUDED:
:F0:FILNAM.EXT (NAVPACK)
:F0:PROG .LIB (TRIG)
:F0:PROG .LIB (EXP)
:F0:TANSTA.AFL (MCBRIDE) (PUBLICS)

UNRESOLVED EXTERNAL NAMES:
   COSIGN—REFERENCED IN :F0:FILNAM.EXT(NAVPACK)
```

(Other errors appear here.)

Order of Modules in the Output File

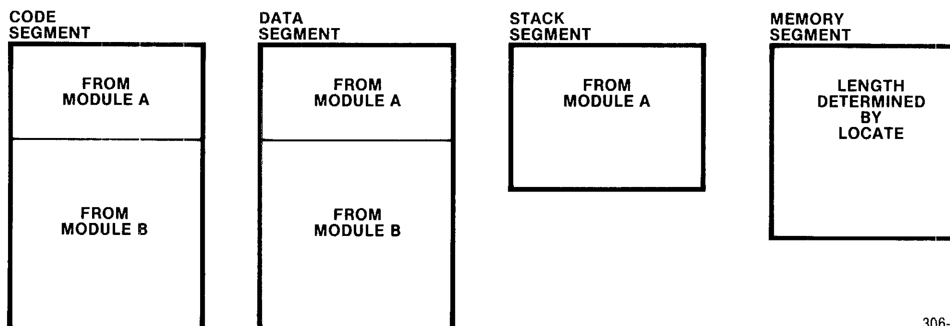
LINK combines modules from the input list by combining all the code segments from the input modules into one code segment, all the data segments into one data segment, and all the stack segments into one stack segment. The length of the memory segment is computed by LOCATE. Any absolute information in the input modules is transferred to the output module with absolute addresses unchanged. If absolute information is in conflict for the same location, a message to that effect is put in the link map.

The order of combining follows the order in which modules are specified in the input list. The first module specified is the first in order in combining. The segments of the second module follow the segments of the first module at the first available location.

Example : The following LINK command and explanation shows how modules are combined.

```
LINK A,B TO C
```

Module A contains code, data, and stack segments. Module B contains code and data segment. They are all byte relocatable. The resulting module C has the following structure



In the process of linking the input modules, external references are satisfied. For instance, if module A has a branch to a point in module B, it is no longer external when A and B are combined into C. If A or B has an external reference to a module in a library file, the library must be specified in the input list *after* the module that refers to it. Suppose the library file RTNS.LIB contained a module that satisfied an external reference in A. Then the input list could be specified A,RTNS.LIB,B or A,B,RTNS.LIB. "Appendix C: Error Messages" lists the LINK error messages.

Locate Command

The LOCATE program takes an input file containing a relocatable object module and produces an output file containing the object module with the relative addresses fixed to absolute locations. The LOCATE program is activated by the LOCATE command. The syntax of the LOCATE command is:

LOCATE <inputfile> [TO <outputfile>] [<controls>]

where:

<inputfile> is the name of the file containing the relocatable object code.

<outputfile> is the name of the file that is to contain the absolute object module. If TO <outputfile> is omitted, it uses the filename portion of <inputfile>. If a file already exists with the same name as specified by <outputfile> (or the defaulted name), it is overwritten with the new data. If <outputfile> is not specified, <inputfile> must consist of a filename and extension because the default name for the output file is the filename (without extension) from <inputfile>.

<controls> specifies one or more keywords that control the operation of LOCATE. The controls and their default values are explained in the following descriptions. The LOCATE controls are:

MAP
COLUMNS(number)
PRINT(file)
SYMBOLS
LINES
PUBLICS
PURGE
ORDER(segment sequence)
CODE(address)
DATA(address)
STACK(address)
MEMORY(address)
/common name/(address)
//(address)
NAME(name)
RESTART0
START(address)
STACKSIZE(value)

If a LOCATE command is longer than one line on your console, (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and the associated parameter list, for example:

```
STACKSIZE &
(40)
```

WARNING

LOCATE uses a temporary file named LOCATE.TMP on the disk to which the output is directed. If you have a file by this name on the output disk it will be destroyed.

"Appendix C: Error Messages" lists the LOCATE error messages.

LOCATE Control Descriptions

MAP

The MAP control specifies that a memory map be printed on the list device. Normally, the printing of the map is suppressed. The map lists the start address for the module, the start and stop addresses for each segment along with their length, and relocation type.

The REL column specifies the relocation type. The following abbreviations are used in the REL column:

B byte relocatable
P page relocatable
I in-page relocatable
A absolute or non-relocatable

When a segment overlaps another segment, a warning, (OVERLAP) is printed in the map. This does not stop the locate function. This may be intentional, as in the sample map. The three byte absolute segment is designated to fit into the data segment.

The following is an example of a memory map:

```
ISIS-II OBJECT LOCATER Vx.y INVOKED BY:
-LOCATE TRIG.REL MAP PRINT(:CO:) NAME(TRIG)

MEMORY MAP OF MODULE TRIG
READ FROM FILE :F0:TRIG.REL
WRITTEN TO FILE :F0:TRIG
MODULE START ADDRESS 3000H

  START     STOP     LENGTH     REL   NAME

  0008H     000AH        3H      A    ABSOLUTE
  3000H     343FH      440H      B    CODE
  3440H     472EH     12EFH      B    DATA
  3630H     3632H        3H      A    ABSOLUTE (MEMORY OVERLAP)
  472FH     475FH       31H      B    STACK
  4760H     F6BFH     AF60H      B    MEMORY
```

WARNING

The length of the MEMORY segment is always computed to be the amount of available memory on the host Intellec development system. If the module is executed on the Intellec system, the MEMORY segment length is correct as specified. If the module is executed on a different system the actual amount of memory depends on the configuration of that system. LOCATE has no knowledge of the configuration of the target system.

COLUMNS(number)

The COLUMNS control specifies whether the symbol table in the list file is to be printed in 1, 2, or 3 columns. The default is 1. This control is ignored unless SYMBOLS, LINES, or PUBLICS is specified.

PRINT(file)

The PRINT control specifies a list file for the LOCATE program output. If the PRINT control is not specified, the output goes to the console output device (:CO:).

SYMBOLS

The SYMBOLS control specifies that a list of local symbols (within a module) and input module names is to be included in the symbol table in the list file. Normally, the printing of the symbol table is suppressed. Local symbols are listed in the table with a type of SYM and module names have the type MOD.

The following is a sample symbol table:

```
SYMBOL TABLE OF MODULE TRIG
READ FROM FILE TRIG.REL
WRITTEN TO FILE TRIG

 VALUE     TYPE     SYMBOL

           MOD      TRIG
 3011H     PUB      SIN
 3015H     SYM      SIO
 3027H     SYM      SI1
 3040H     SYM      TRIGEX
 3063H     PUB      COS
 3102H     PUB      TAN
 3230H     SYM      TANO
 3340H     LIN      272
```

LINES

The LINES control specifies that a list of the line numbers and input module names from the program is to be included in the symbol table in the list file. Line numbers have the type LIN and module names have the type MOD. The VALUE field is blank for module names in the symbol table. Normally, the printing of the symbol table is suppressed.

PUBLICS

The PUBLICS control specifies that a list of symbols declared public is to be included in the symbol table. Public symbols have the type PUB. Normally, the printing of the symbol table is suppressed.

PURGE

The PURGE control specifies that line numbers, local symbols, module names, and public symbols be removed from the output module. Public symbols could be used, if present, to link the absolute module with other modules. Line numbers, local symbols, and module names could be used for debugging. The PURGE control condenses the size of a module that is completely debugged. This results in saving disk space and decreasing load time.

ORDER(segment sequence)

The ORDER control defines the order in which the various segments are assigned memory locations. The list of segments in the segment sequence parameter must be separated with spaces.

If ORDER is not specified, the segments are located in the following order:

```
CODE
STACK
/commons/ (named and unnamed commons in an arbitrary order)
DATA
MEMORY
```

This default order can be changed with the ORDER control. LOCATE determines the addresses at which the segments reside but it is done in the order specified by the control.

A partial list can be specified in the ORDER control. When a partial list is specified, the segments specified are located first in the order specified. The remaining (unspecified) segments follow in the default sequence. For example, the control:

```
ORDER(DATA)
```

results in the sequence: DATA, CODE, STACK, /commons/, MEMORY (assuming all segment types exist in the module.)

The first segment is located 680H bytes above the top of the ISIS-II code (3680H). This allows room for 13 input and output buffers (six open files and :CO: and :CI:). You can also change the order with the CODE, STACK, /name/, //, DATA, and MEMORY controls by specifying a specific address at which each will reside. The section "How LOCATE Locates Segments," following the control descriptions describes how the default order, ORDER control, and specific address controls interact to locate segments.

```
CODE(address)
DATA(address)
STACK(address)
/common name/(address)
//(address)
MEMORY(address)
```

The segment locations can be specified explicitly with the segment controls. The controls are specified with the address. The address can be in decimal, hexadecimal, octal, or binary. The address must begin with a digit and may be followed by a letter specifying the base of number:

Decimal - D or omitted
Hexadecimal - H
Octal - O or Q
Binary - B

The specified addresses of some segments may be changed by LOCATE. If because of the addressing within a segment, it must reside at or between 256 byte multiples of memory, the Locate program will make the adjustment.

The section "How LOCATE Locates Segments," following the control descriptions describes how the default order, ORDER control, and specific address controls interact to locate segments.

NAME(name)

The NAME control specifies a name for the output module. The name can be from 1 through 31 characters, each of which must be a letter (A through Z), a digit (0 through 9), a question mark (?), or a commercial at sign (@). However, the first character of the name cannot be a digit. If NAME is not specified, the name in the header record of the input file is used.

RESTART0

The RESTART0 control places a jump instruction at locations 0, 1, and 2 in the absolute module. The address in the jump instruction is the programs starting address (the address of the first instruction to be executed) taken from the input module or from the START control. You would use the RESTART0 control when

preparing an absolute module for execution in your prototype system, either the standalone system or the system being emulated with the in-circuit emulator. When the RESET signal is input to the CPU, the program counter is set to 0 and execution begins with the jump to the beginning of your program. An absolute module prepared with the RESTART0 control is not compatible with ISIS-II, which does not allow user code to be loaded in locations 0, 1, or 2. The RESTART0 control is ignored if the input module is not a main module.

START(address)

The START control specifies the address of the first instruction in the code segment to be executed. This address overrides the address in the input module. If START is omitted, the address is taken from the input module. The START control is ignored if the input module is not a main module.

STACKSIZE(value)

The STACKSIZE control specifies a value (in bytes) for the size of the stack segment. This value overrides any calculated stack size encountered in the input module.

When debugging a program in an Intellec microcomputer development system, 12 additional bytes of user stack are required beyond that computed by the language translator or LINK. LOCATE adds these 12 bytes if the STACKSIZE parameter is not specified.

How LOCATE Locates Segments

Module segments are normally located sequentially in memory in the order:

CODE segment
STACK segment
/commons/ segments (in an arbitrary order)
DATA segment
MEMORY segment

You can change the order with an ORDER control and with the CODE, STACK, /common/, //, DATA, and MEMORY controls. You can change the order by using the default order in conjunction with the ORDER control and the segment controls specifying an exact address.

Locating With The Default Order

When you use the default order, the CODE segment is located 680H bytes above the top of ISIS-II and the rest of the segments immediately follow in order. Gaps are generated only when required by the relocation type of the segment.

- Byte relocatable (BR) segments are located at the first available byte.
- Page relocatable (PR) segments are located at the first available byte that lies on a page boundry (a multiple of 256 (100H) bytes).
- In-page relocatable (IP) segments are located at the first available byte such that the segment is totally contained within a page.

Locating With the Default and ORDER Control

You can change the order totally with the ORDER control. When you specify all segments in the ORDER control the first segment is located 680H bytes above the top of ISIS-II and the rest of the segments immediately follow in the order specified. Gaps are generated only when required by the relocation type of the segement.

If you don't specify all the segments in the ORDER control:

- First, the segments specified in the ORDER control are located in the order specified.
- Next, the segments not specified in the ORDER control are located immediately following the last segment specified, in the default order.

If you submit the following ORDER control with the LOCATE command:

```
ORDER(DATA STACK)
```

the segments will be located in the following order:

DATA
STACK
CODE
/commons/ (if FORTRAN)
MEMORY

Locating With the Default, ORDER Control, and Specific Addresses

You can change the order with a combination of the default order, an ORDER control, and specific segment location controls. You should not specify a segment in the ORDER control and with a segment location control. When all three forms of locating are used, segments are located according to the following:

- Segments are selected for placement in the order specified by the ORDER control and the default sequence as described in the preceding section.
- The starting address of a segment is either the address following the preceding segment (680H above ISIS-II for the first segment) or the address specified in a location control.

If the LOCATE command is submitted with the following controls:

```
ORDER(DATA STACK) CODE(6000H)
```

the segments are placed in the sequence:

DATA
STACK
CODE
/commons/
MEMORY

The DATA segment is located 680H bytes above the top of ISIS-II and STACK is located immediately following DATA. CODE is located next, but instead of being placed immediately following STACK it is placed at address 6000H as specified by the CODE control. The /commons/ (if any) immediately follow CODE and MEMORY follows the /commons/. In other words, the segments are still located in the sequence specified by the default order and the ORDER control but the locations are modified by the individual segment location controls.

If you are going to locate some segments at specific addresses and let LOCATE place the rest, you should use the ORDER control to modify the default sequence so that segments are located in an order that corresponds with the specific addresses in the controls. If you aren't careful, conflicts can occur. Be sure to specify the MAP control, to verify that segments are placed as intended. Conflicts do not stop the LOCATE function, because there are circumstances where you will want apparent conflicts such as the location of an absolute segment in an internal gap in a segment.

When you want to locate FORTRAN common segments to specific addresses, you should also locate the MEMORY segment to an address above the top of the highest common segment. LOCATE handles the common segments in an arbitrary order. You will not know ahead of time what order the common segments will be handled by the command. If the common segment that you place at low memory is the last one handled by LOCATE, the MEMORY segment will immediately follow it and will conflict with all segments above it.

LIB Command

WARNING

LIB uses a temporary file named LIB.TMP on the disk to which the output is directed. If you have a file by this name on the output disk it will be destroyed.

The ISIS-II LIB program allows you to create specially formatted files to contain libraries of object modules, to maintain these libraries by adding and deleting modules, and to obtain a listing of the modules in a library file. Libraries can be used as input to LINK, which may automatically link modules from the library that satisfy external references in the modules being linked.

The library manager program is called into operation by the LIB command. The syntax of the LIB command is:

LIB

The operation of LIB is controlled by entering commands to indicate which operation LIB is to perform. LIB prompts for commands with an asterisk (*). The commands are:

CREATE
ADD
DELETE
LIST
EXIT

Continuation Lines

If a command to LIB is longer than one line on your console (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and a parameter, for example:

```
DELETE PVT.LIB&
(MOD1)
```

LIB prompts for the continued line with a double asterisk (**). If necessary, subsequent lines can be continued also.

CREATE - Create a Library File

The CREATE command creates an empty library file. You must use the ADD command to add modules to the library file. The syntax of the CREATE command is:

CREATE <filename>

where

<filename> specifies the name to be assigned to the new library file. If a file with that name already exists, an error message is sent to the console and LIB prompts for another command.

ADD - Add Modules to a Library File

The ADD command adds object modules to a library file. The syntax of the ADD command is:

ADD <filename>[(<modname>,...)] [,...] TO <libfile>

where

<filename> can be the name of a library file or the name of a file containing an object module. If a library file is specified, all the object modules contained in it are added to <libfile> unless <modnames> are specified.

<modnames> can be specified only if <filename> is a library file. Only the object modules specified by <modnames> are added to <libfile>.

<libfile> is the library file being modified by the addition of modules in <filename>.

DELETE - Delete Modules from a Library File

The DELETE command deletes modules from a library file. The syntax of the DELETE command is:

DELETE <libfile> (<modname>,...)

where

<modname> specifies the object module to be deleted from <libfile>.

LIST - List Library Modules and Their Public Symbols

The LIST command lists the module directory of the library file. The syntax of the LIST command is:

LIST <libfile>[(<modname>,...)][,...] [TO <listfile>] [PUBLICS]

where

<libfile> is the name of the library file whose entire module directory is to be listed unless <modname> is also specified. In that case, only information about the specified modules is listed.

<listfile> is the name of the file to contain the library listing. If omitted, the directory is listed on the current console output device (:CO:).

PUBLICS specifies that public names in each module are to be listed. If omitted, only the module names are listed.

The format of the listing when public names are requested is:

```
*LIST TEST.LIB PUBLICS

TEST.LIB
|OPEN
|     NOREX
|     ABEX
|REDUCE
|     HEX
|     OCT
|     DATUM
|CLOCK
|    |TIME
|    |LAPSE
|    |CYC
|    |  L---- public names
|    L------- module names
L------------ library name
```

EXIT - Return to ISIS-II

The EXIT command returns control to ISIS-II. When finished with LIB, enter the EXIT command, followed by a carriage return. This terminates the LIB program and returns control to ISIS-II, which prompts for a command with a hyphen (-).

Example : The following example shows the creation of a library file and the entry in the library of two modules. The directory of the library is listed before exiting to ISIS-II.

```
-LIB
ISIS-II LIBRARIAN Vx.y
*CREATE FOO.LIB
*ADD SIN.OBJ,COS.OBJ TO FOO.LIB
*LIST FOO.LIB
  SINE
  COSINE
*EXIT
-
```

"Appendix C: Error Messages" lists the LIB error messages.

Program Overlays And Linked Loading

When a program is larger than the available memory space, it is necessary to link modules without combining them into one module. Thus during execution when part of the program is no longer needed, another part can be loaded in the same area of memory, overlaying the part not needed. Under ISIS-II, programs or parts of programs to be loaded separately must be in separate files. The first load can be done by entering the name of the file as a command. The subsequent loads are done from a program with the LOAD system call or an I/O routine.

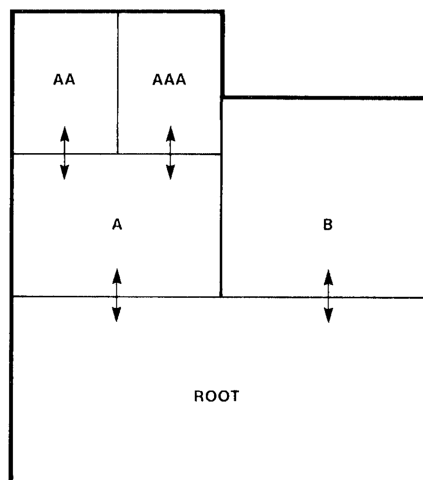
In the typical use of LINK and LOCATE, modules with external references are combined with modules that have matching public symbols to produce a module with no unsatisfied external references. In linking without combining, the external references must still be satisfied; that is, they must know the addresses of their matching public symbols. Using the keyword PUBLICS before a list of modules tells LINK that the modules are not to be combined in the output module but used only to supply the addresses of their public symbols.

In this way the external references of modules listed earlier in the input list are satisfied. For example,

```
LINK A,PUBLICS(B,C) TO A.SAT
```

results in a module A.SAT whose external references to symbols contained in B and C are satisfied. However, modules B and C must be absolute modules because LINK must know the absolute addresses of the public symbols. Therefore the typical use of LINK before LOCATE is reversed. You must LOCATE modules B and C first, creating modules with absolute addresses but perhaps with unsatisfied external references. The module created in this way can be considered a temporary module used only to supply addresses of public symbols needed by other modules.

Consider the following example. A root segment calls segment A and later calls segment AA, then segment AAA overlays AA. A diagram of this overlay structure follows where the vertical lines indicate a division in time (segments A and B are not in memory at the same time) and the horizontal lines indicate a division in memory space.



306-2

All the pieces of this overlay structure must be in separate files because they are loaded separately. The root calls A and B; A calls AA and AAA. If you locate the root first, you can use the memory map produced by LOCATE to determine the base address of A and B. Then locate A and B. Use the memory map produced by LOCATE for A to determine the base address of AA, AAA. The modules produced by LOCATE have absolute memory addresses assigned but external references are unsatisfied. These modules cannot be loaded into memory for execution but can be used with the PUBLICS keyword in the linking process.

Suppose the modules produced by LOCATE were given the extension of TMP. Suppose the root has external references to public symbols in A and B; A has external references to public symbols in the root, AA, and AAA; AA and AAA have external references to public symbols in A; B has external references to public symbols in the root. Then the following LINK commands would produce the satisfied modules ready to execute.

```
LINK ROOT.TMP,PUBLICS(A.TMP,B.TMP) TO ROOT
LINK A.TMP,PUBLICS(ROOT.TMP,AA.TMP,AAA.TMP) TO A
LINK AA.TMP,PUBLICS(A.TMP) TO AA
LINK AAA.TMP,PUBLICS(A.TMP) TO AAA
LINK B.TMP,PUBLICS(ROOT.TMP) TO B
```

The modules ROOT, A, AA, AAA, and B are absolute because of the previous LOCATE operation and fully satisfied because of the above LINK operation. They are connected but not combined.

To verify that all external references are satisfied in the modules produced by LINK, you can LOCATE them again. This time LOCATE should produce no message about unsatisfied external differences.

When you link without combining to produce overlays, you must provide overlay management in the design of your system. That is, before your program makes a reference to an overlay segment, it must make sure that segment is in memory. If not, the segment must be read into memory. When a segment in memory contains new data that must be saved, it must be written out before it is overlaid with another segment. The ability to link without combining provides the hooks for an overlay scheme. The runtime management of overlays must be designed into your software.

Memory Pages and the H and L Registers

Relocation types are provided for programs that reference memory by manipulating the H and L registers independently. (See the *8080/8085 Assembly Language Programming Manual* 9800301, for a description of the HIGH and LOW operators.) You can store data on a page boundary and address elements in it by changing only the L register. If the data does not cross a page boundary, you do not have to change the H register at all.

However caution must be used if the HIGH operator is used on an arbitrary address in relocatable code, you may get an incorrect address because LOCATE assumes the unused portion of the address to be zero. If the unused portion of the address is not

zero and the addition of the low order portion of the segment base address causes a carry into the high order portion, that carry will not be detected when the HIGH operator is used. For example, if HIGH is used on the relocatable address 1234H:

```
HIGH(1234H) = 12H
```

and LOCATE adds a segment base address of 10F0H:

WITH HIGH OPERATOR	WITHOUT HIGH OPERATOR
1200H	1234H
+10F0H	+10F0H
22F0H	2324H
THE HIGH PART OF WHICH IS 22H	THE HIGH PART OF WHICH IS 23H

Because LOCATE has no knowledge of the low order portion of the address there is a chance that the located HIGH address will be off by one. The located address will be correct if there is no carry from the low order portion.

You can avoid this situation by only using the HIGH operator on addresses in segments that you have defined as *page relocatable* .

This circumstance does not exist with the LOW operator. Addresses on which LOW has been used will always be correct.

Savings in memory space and execution time can result from this method, but access to some areas of memory may be lost because of the way LINK and LOCATE act to preserve relocation types.

When LINK combines segments having different relocation types, it follows these rules:

- Byte relocatable segments follow the preceding segment at the next byte. The output segment is byte relocatable only if all input segments are byte relocatable. Otherwise, it is page relocatable.
- Page relocatable segments follow the preceding segment at the first available page boundary. The bytes from the end of the preceding segment to the page boundary, if any, are unused. The output segment is page relocatable.
- Inpage relocatable segments are located at the first location following the preceding segment if they fit within the page. Otherwise, they are located at the next page boundary and the bytes at the end of the previous page are unused. The output segment is inpage relocatable only if it is not more than 256 bytes and all input segments are inpage relocatable. Otherwise, the output segment is page relocatable.

The bytes that are unused in the process of preserving the relocation type of segments are flagged in the LINK memory map by the word 'gap.' These gaps are unused portions of the program; in a sense they are lost.

If LOCATE assigns the base addresses of segments, it does so in a way that preserves the relocation type of the segment. It also issues a message if an inpage relocatable segment is changed to page relocatable. If you specify a base address for a segment with a segment location control that violates the relocation type of the segment, LOCATE places the segment on the next higher page boundary and issues a message.

CHAPTER 5 USE OF ISIS-II AND THE MONITOR BY OTHER PROGRAMS

Writing programs that make use of ISIS-II and Monitor capabilities is the same as writing any other program except that the program includes ISIS-II or Monitor system calls. The placement of your program in memory must take into consideration how memory is organized under ISIS-II.

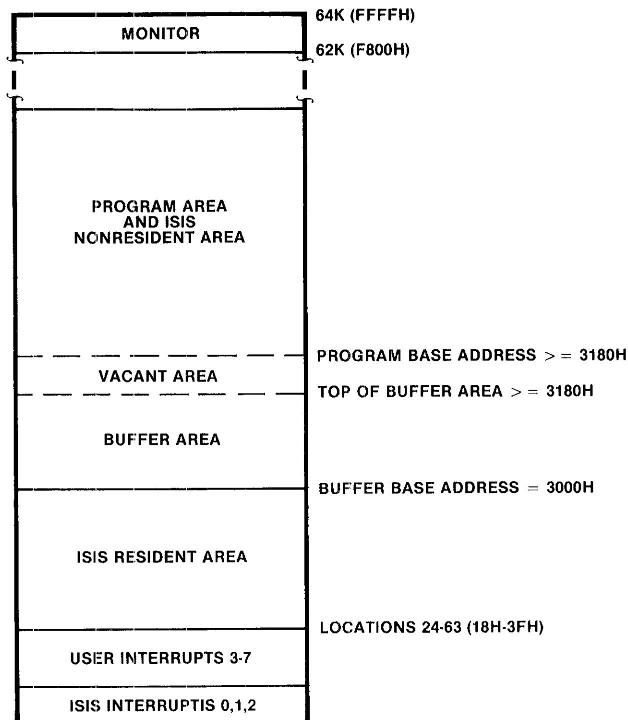
Line-editing is available to ISIS-II programs. It allows you to correct errors at the keyboard so that an error free line can be transmitted to the requesting program.

Memory Organization And Allocation

The organization of Intellec memory under ISIS-II is shown below. Interrupts 0 through 2 are reserved for ISIS-II and the Monitor. Interrupts 3 through 7 are available for use within your program. These locations, 24-63, are the only locations below 12K (3000H) that can be loaded with user code. Loading other locations below 12K is not allowed.

The ISIS-II resident area is reserved for the part of ISIS-II that is always resident in RAM memory. Although the ISIS-II resident area 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 in this area, causing subsequent errors when system services are requested.

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. The minimum size of the buffer area allows for three buffers, including the ISIS-II permanent buffer. If more than two buffers are required by your program, the buffer area increases at the expense of the vacant area.



The program area is above the buffer area. The program base address is determined by the programmer and assigned by LOCATE or entered in an ORG statement of 8080/8085 assembly language absolute programs.

The command interpreter (ISIS.CLI) of ISIS-II, other nonresident ISIS-II routines (all commands except DEBUG), the editor, assembler, compiler, linker, locater, and library manager run in the program area. Whenever you communicate with ISIS-II via console commands, you communicate with the nonresident command interpreter running in the program area.

Nonresident routines of ISIS-II may use all available RAM for buffers. Therefore, user programs that must be permanently resident should be placed in ROM or in RAM that is physically discontiguous from the first contiguous block of RAM.

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 location of contiguous RAM.

The Monitor occupies the top 2K of memory. If your memory configuration encompasses addresses F800H to FFFFH those addresses are shadowed by the Monitor ROM and cannot be written. The Monitor also uses the top 320 bytes of contiguous RAM for its workspace; this area is above the address returned by a call to the MEMCK routine.

The maximum area required for buffers under ISIS-II must be determined before deciding the base address of your program (see Chapter 4). The number of buffers varies dynamically, but the buffer area must be as large as the maximum number of buffers allocated simultaneously.

If you locate the base address of your program below 3180H (or allocate less than 3 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 the 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 including :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. A system call that accesses a disk directory (LOAD, DELETE, RENAME, ATTRIB, CONSOL when it specifies a disk file) requires two buffers during the processing of the call. The buffers are released on return to the calling program.
4. When the CONSOL system call assigns the console input or output device to a disk file, three buffers are required for the console input file and two buffers are required for the console output file. These buffers are required until end-of-file. A program called by a system command in a SUBMIT file must also allow for these buffers in determining its origin point.

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, it needs five buffers; the base address must be 3280H:

3000H + (128*5) = 3280H

If you want to cover the possibility of the program being called from a SUBMIT file where the console output device is also a disk file, four more buffers are needed, requiring an address of 3480H:

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

Example 2: Suppose a program opens a line-edited disk file (3 buffers), an echo file on disk (2 buffers), and also changes the attributes of a disk file with the ATTRIB system call while the files are open (2 buffers). 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 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 file you would allow two buffers whether or not it is a disk file. You would also allow five buffers for the console input and output files, whether or not they are disk files.

Line-Edited Input Files

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

ISIS-II is notified that a file is to be line-edited by a parameter in the OPEN system call. Associated with every line-edited file is a file to which ISIS-II sends an echo of the input. The echo file must be opened before the line-edited file. If no echo is desired, the byte bucket file :BB: can be specified 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. The contents of the buffer are altered by ISIS-II when an editing character is entered. See explanation of editing characters below. 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, a READ system call transfers data from the line-editing buffer to the requesting program's buffer. ISIS-II maintains a pointer to keep track of what characters have been read from the line-editing buffer. For example, if the line-editing buffer contains 100 characters and a READ system call with COUNT=50 is issued, the first 50 characters are transferred to the program's buffer and the buffer pointer is moved to the 51st character. The next READ system call transfers characters starting at the 51st character.

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

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

When the contents of the line-editing buffer have been completely read (the pointer is after the last character), a READ system call transfers 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 entered in the buffer rather than causing its usual editing or terminating function.
CONTROL/R	A CONTROL/R has no effect except its echo, which is carriage return-line feed, followed by the current undeleted contents of the buffer.
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 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		:		1	:	P	R	O	G	A	.	S	C	R	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, a 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.

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, MEMCHK).
- 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 call uses your stack. Your stack must have the depth necessary to handle 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 previously discussed. 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 calls 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 your 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 in 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.

Assembler 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 Chapter 4 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 calls symbolically. Only the specific system calls needed by your program need be defined. The following table lists the identifying numbers for the system calls.

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

Six 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 data transferring services of the supervisor 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 the file being operated upon.

System Calls Cautions

Since some ISIS-II system calls reference monitor routines, you should not mix monitor routine calls with system routine calls, or unpredictable results may occur.

When an interrupt occurs, the location of program execution is saved. If you issue a system call as part of an interrupt service routine, this location information will be lost, with unpredictable results.

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

Good programming practice suggests frequent STATUS checks when making frequent system calls. Refer to the TYPE program in Appendix D 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.

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

- An address of a two byte field in which ISIS will store the active file 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.
- 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.
- 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). A value of 2 specifies that the file is open only for output from the system (WRITE). A value of 3 specifies that the file is open for update, READ and WRITE. When a file is opened for input, MARKER is set to 0 and LENGTH is unchanged. If the file specified for input is nonexistent, a nonfatal error occurs. When a file is opened for output, MARKER and LENGTH are set to 0. If the file specified for output is nonexistent, a disk file is created with the specified filename and all attributes of the new file are reset. Specifying a disk file whose format or write-protect attributes are set causes a nonfatal error. When a file is opened for update, MARKER is set to 0. If the file already exists, LENGTH is unchanged. If the file is nonexistent, a file is created with the specified filename and all attributes reset. LENGTH is set to 0. Opening a file for an access mode that is not physically possible causes a nonfatal error. For example, opening :HP: (high-speed paper tape punch) for input or opening :LP: (line printer) for update causes an error.
- 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.
- The address of a memory location for the return of nonfatal error numbers. The error numbers that OPEN can return are listed in Appendix C.

PL/M OPEN Call Example

```
OPEN:
  PROCEDURE (AFTNPTR,FILE,ACCESS,MODE,STATUS) EXTERNAL;
    DECLARE (AFTNPTR,FILE,ACCESS,MODE,STATUS) 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 specified memory location specified by the calling program. See "Line-Edited Input Files" for information concerning reading of line-edited files.

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

- The AFTN of a file that is open for input or update. The AFTN is returned by a preceding OPEN call or is 1 for :CI:.
- The address of a buffer to contain 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, the memory locations that follow the buffer will be overwritten.
- The number of bytes (count) to be transferred from the file to the buffer.
- The address of a memory location in which ISIS will store 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. If COUNT = 0, then ACTUAL = 0 may or may not indicate end-of-file. End-of-file is best indicated, in the case of line-edited files and COUNT greater than 0, by ACTUAL = 0; in the case of lined files and COUNT greater than 0, it is indicated by ACTUAL less than COUNT.
- The address of a memory location in which ISIS will store nonfatal error numbers. The error numbers returned by the READ call are listed in Appendix C.

PL/M READ Call Example

```
READ:
  PROCEDURE(AFTN,BUFFER,COUNT,ACTUAL,STATUS)EXTERNAL;
      DECLARE (AFTN,BUFFER,COUNT,ACTUAL,STATUS) 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. A parameter list of four variables must be passed with the WRITE call:

- The AFTN of a file open for output or update. The AFTN was returned by a preceding OPEN call or is 0 for :CO:.
- The address of the memory location of the buffer from which data is to be transferred, or a string literal of the format *.string literal*, where the period (.) specifies the contents of the string buffer labeled *.string literal*.

- The number of bytes (count) 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 buffer are written to the file.
- The address of the memory location for the return of nonfatal error numbers. The error numbers returned by WRITE are listed in Appendix C.

PL/M WRITE Call Example

```
WRITE:
    PROCEDURE (AFTN,BUFFER,COUNT,STATUS) EXTERNAL;
        DECLARE (AFTN,BUFFER,COUNT,STATUS) ADDRESS;
    END WRITE;
.
.
.
DECLARE AFT$IN ADDRESS;
DECLARE BUFFER(128) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL WRITE (0,.('this is an example of string literal', 0DH,0AH),38, .STATUS);
CALL WRITE (AFT$IN,.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 find the location of or to change the value of MARKER associated with a disk file open for read or update. The SEEK call can only be used with a file open for update or read. The MARKER 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.

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

- The AFTN of a disk file opened for update or input. The AFTN was returned by a preceding OPEN call.
- A mode value from 0 through 4 that indicates what action should be performed on the MARKER. The block and byte parameters (see below) are used to represent the current MARKER position or to calculate the desired offset.

Return Marker Location

If the mode value is 0, the system returns a pair of block and byte values 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 MODULO 32768) + byte number

Move Marker Backward

If the mode value is 1, the marker is moved backward, toward the beginning of the file. The block and byte parameters determine the offset; 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 MODULO 32768) + byte number

If N is greater than MARKER, i.e., if the prescribed action would place the marker before the beginning of the file, MARKER is set to 0 (beginning of file), and a non-fatal error occurs.

Move Marker to Specific Location

If the mode value is 2, 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 (00HH) are added to the file to extend the file to the marker. (Thus, LENGTH becomes equal to MARKER.)

Move Marker Forward

If the mode value is 3, 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 MODULO 32768) + 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 (00HH) 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 non-fatal 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.

The other three variables that must be passed with the SEEK call are:

- 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.
- The address of a memory location containing a 2-byte value used for the byte number. The byte number may be greater than 128.
- The address of a memory location in which ISIS will store nonfatal error numbers. The error numbers returned by the SEEK call are listed in Appendix C.

NOTE

If MARKER has allocated more memory than LENGTH requires, a DIR will show the allocated location as in use. You can actually allocate more storage than exists on disk in this way. Data can still be written to these locations.

PL/M Seek Call Example

```
SEEK:
  PROCEDURE (AFTN,MODE,BLOCKNO,BYTENO,STATUS) EXTERNAL;
      DECLARE (AFTN,MODE,BLOCKNO,BYTENO,STATUS) 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 echoed (output to the echo file), it has already been input from the keyboard and echoed. Thus the subsequent READ does no input from a file but only reads from a buffer in memory.

A parameter list of two variables must be passed with the RESCAN call:

- The AFTN of a file open for line-edited input (echo file AFTN specified) by a preceding OPEN call.
- The address of a memory location for the return of nonfatal error numbers. The error numbers returned by the RESCAN call are listed in Appendix C.

PL/M RESCAN Call Example

```
RESCAN:
  PROCEDURE (AFTN,STATUS) EXTERNAL;
      DECLARE (AFTN,STATUS) 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. All files should be closed 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.

A parameter list of two variables must be passed with the CLOSE call:

- The AFTN of the file to be closed. The AFTN was returned by a preceding OPEN call.
- The address of a memory location for the return of nonfatal error numbers. The nonfatal error numbers issued by CLOSE are listed in Appendix C.

PL/M CLOSE Call Example

```
CLOSE:
  PROCEDURE (AFTN,STATUS) EXTERNAL;
      DECLARE (AFTN,STATUS) 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.

A parameter list of three variables must be passed with the SPATH call:

- 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.
- 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

The possible values for 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.

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

- The address of a memory location for the return of a nonfatal error number. The nonfatal error numbers issued by the SPATH call are listed in Appendix C.

PL/M SPATH Call Example

```
SPATH:
  PROCEDURE (FILE,BUFFER,STATUS) EXTERNAL;
      DECLARE (FILE,BUFFER,STATUS) ADDRESS;
  END SPATH;
.
.
.
DECLARE FILENAM(15) BYTE;
DECLARE BUF$IN(12) BYTE;
DECLARE STATUS ADDRESS;
.
.
.
CALL SPATH (.FILENAM,.BUF$IN,.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 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 space allocated to the file is released. The space can then be reused for another file.

A parameter list of two variables must be passed with the DELETE call:

- 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.
- The address of a memory location for the return of a nonfatal error number. The error numbers returned by DELETE are listed in Appendix C.

PL/M DELETE Call Example

```
DELETE:
  PROCEDURE (FILE,STATUS) EXTERNAL;
      DECLARE (FILE,STATUS) 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.

A parameter list of three variables must be passed with the RENAME call:

- 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.
- The address of an ASCII string that contains the new 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. The device portion of the name must be the same as that in old name.
- The address of a memory location for the return of a nonfatal error number. The nonfatal error numbers issued by RENAME are listed in Appendix C.

PL/M RENAME Example

```
RENAME:
  PROCEDURE (OLDFILE,NEWFILE,STATUS) EXTERNAL;
      DECLARE (OLDFILE,NEWFILE,STATUS) 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.

A parameter list of four variables must be passed with the ATTRIB call:

- 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.

- An identifier indicating which attribute is to be changed. The identifier can be:
  0 - invisible attribute
  1 - system attribute
  2 - write protect attribute
  3 - format attribute
- 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.
- The address of a memory location for the return of a nonfatal error number. The nonfatal error numbers issued by the ATTRIB call are listed in Appendix C.

PL/M ATTRIB Call Example

```
ATTRIB:
  PROCEDURE (FILE,ATRIBTE,ONOFF,STATUS) EXTERNAL;
      DECLARE (FILE,ATRIBTE,ONOFF,STATUS) 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. These system calls 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 devices (:CI: and :CO:) to devices other than the initial system console.

A parameter list of three variables must be passed with the CONSOL call:

- The address of an ASCII string that contains the name of the file to be used for system console input. 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. 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.

- The address of an ASCII string that contains the name of the file to be used for system console output. 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. 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.

- The address of a memory location for the return of a nonfatal error number. The nonfatal error numbers issued by the CONSOL call are listed in Appendix C.

PL/M CONSOL Call Example

```
CONSOL:
  PROCEDURE (INFILE,OUTFILE,STATUS) EXTERNAL;
     DECLARE (INFILE,OUTFILE,STATUS) 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.

A parameter list of three variables must be passed with the WHOCON call:

- 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.
- 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.
- (Assembly language only) The address of a memory location for return of an error number.

PL/M WHOCON Call Example

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

Assembly Language WHOÇON 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.

A parameter list of two variables must be passed with the ERROR call:

- 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; 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.

- (Assembly language only) The address of a memory location for return of an error number.

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

Two system calls allow your program to transfer control to another program (LOAD) or to ISIS (EXIT). The LOAD call can be used to load another program and then transfer control to it, to the Monitor, or have control returned 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:

- 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.
- A bias value to be added to the load address of the program. The program is 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.
- 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. 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.

NOTE

When control passes to the Monitor, the Monitor saves the CPU registers and system status by pushing these items on the stack belonging to the program that issued the LOAD system call. If the newly loaded program overlays the area formerly occupied by the stack, it will be overwritten when the Monitor pushes items on the stack. Therefore, the stack should be located in an area of memory that will not cause the newly loaded program to be overwritten when the Monitor uses the stack.

- 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.
- The address of a memory location for the return of a nonfatal error number. The error numbers issued by the LOAD call are listed in Appendix C.

PL/M LOAD Call Example

```
LOAD:
  PROCEDURE (FILE,BIAS,SWITCH,ENTRY,STATUS) EXTERNAL;
      DECLARE (FILE,BIAS,SWITCH,ENTRY,STATUS) 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 can be used by your program to terminate execution and return to ISIS-II. All open files are closed, with the exception of :CO: and :CI:. The current system console assignment is not changed.

A parameter list of one variable must be passed with the EXIT call:

- (Assembly language only) The address of a memory location for return of an error number.

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

A call to a Monitor I/O routine from an Assembly language program changes the contents of the registers. If the contents of a register must be saved, you should store them before calling the Monitor, and then restore them after return from the Monitor I/O routine.

CI - Console Input Routine

The Console Input routine is a routine which 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. The 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 is of a routine which 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 7H;            /*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 RETRUN
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 (passed as a byte parameter if called from PL/M or passed in the C-register if called from the assembler) and transmits it to the system console output device.

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. When the reader runs out of tape or a control/Z (1AH) character (used here as an end-of-file character) is read the operation is terminated. 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 '01H';
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 (passed as a byte parameter if called from PL/M or passed in the C-register if called from assembler) and transmits it to the System Punch device.

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 (passed as a byte parameter if called from PL/M or passed in the C-register if called from Assembler) and transmits it to the system list device.

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 (03H) character 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
EXT         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)

The Universal PROM Programmer Input 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). The PROM address of the desired data is passed as an address parameter (if called from PL/M) or is loaded into register pair BC (if called from the Assembler), 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 Universal PROM Programmer Input 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 1 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)

The Universal PROM Programmer Output 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. The PROM address to be programmed is passed as an address parameter (if called from PL/M), or with the most significant byte in the D-register and the least signficant byte in the E-register (if called from the Assembler). 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.

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 the assembler, 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 (03H) character 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 = CTLC THEN
      DO;

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

      END;
END;
```

Assembly Language CSTS Call Example

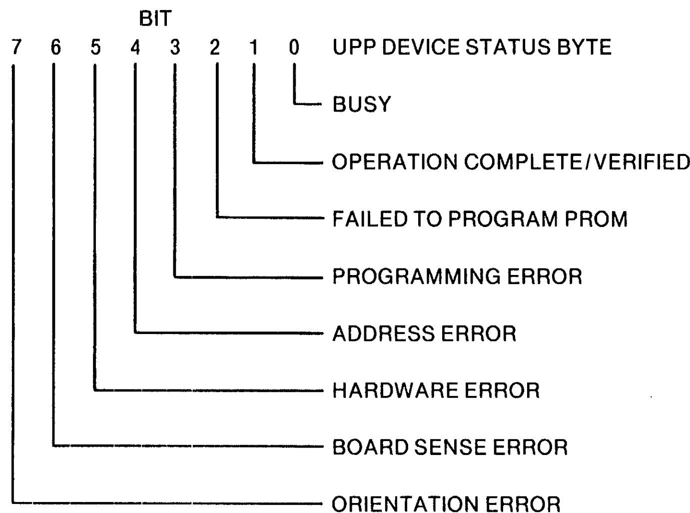
```
          EXTRN   CSTS       ;ENTRY POINT INTO SYSTEM.LIB FOR
                             ;CSTS
          EXTRN   CI         ;ENTRY POINT INTO SYSTEM.LIB FOR CI
CTLC      EQU     03H        ;CONTROL/C SIGNALS TERMINATE
                             ;OUTPUT

          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
          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 the Assembler). 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 IO 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 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 these codes at any time. You can change and swap programs, but cannot have two assigned to the same code.

These codes correspond to the number assignments available in the I/O configuration asignment 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 the I/O Definition routine and pass it the function code, from above, as a byte value (if called from PL/M) or in the C-register (if called from the Assembler). You must also pass the entry point address of the driver as an address parameter, if called from PL/M, or in the D-register (most significant bits) and E-register (least significant bits), if called from the Assembler.

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 you will not have this problem.

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

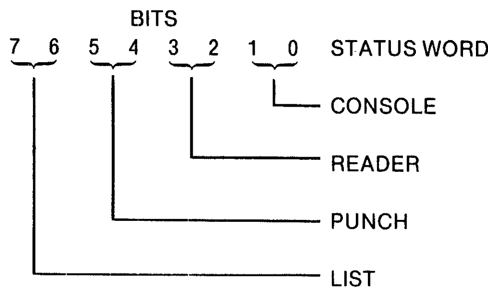
The external declaration for IODEF in PL/M necessary to link it with SYSTEM.LIB is:

```
IODEF: PROCEDURE(CODE,ENTRY$POINT) EXTERNAL;
  DECLARE CODE BYTE;
  DECLARE ENTRY$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:



The following table shows the meaning of the possible values in each field:

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: The Monitor is initially configured when turned on to assign the console and list to the first device operated during initialization and the 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

The Set System I/O Configuration 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 the assembler). 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 the assembler).

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
```

CHAPTER 6
THE INTELLEC® MONITOR

The Monitor is a control program that provides supervisory functions for the Intellec microcomputer development systems. It processes the commands you enter at the console device. The set of available commands provide the following facilities:

- Displaying and modifying memory and processor registers.
- Initiating execution of your programs.
- Inserting breakpoints into your programs before execution.
- Reading hexadecimal data from an external device into memory.
- Writing hexadecimal data from memory to an external device.
- Accessing user written I/O routines.
- Invoking resident diagnostics.

Your communication with the Monitor is through the system console. When you enter the Monitor, the sign-on message and a prompt character are displayed on the system console. The prompt character is a period (.) at the left margin of the system console.

Command Entry

You can enter commands at the console anytime after the prompt character is displayed at the left margin.

All Monitor commands are single alphabetic characters. Some commands have optional parameters and some have required parameters. For example, the X command displays the contents of all the registers. But if you only want to see the contents of a single register, you can specify that register in the command. However, the D command, which displays the contents of memory, requires that beginning and ending memory locations be included in the command.

Normally, commands are ended by pressing the return key on the keyboard. There are exceptions to this and will be fully explained in the individual command descriptions.

Thus the general syntax of the Monitor commands is:

<command>[<parameters>]<CR>]

where:

<command> is the single alphabetic character for the command.

<parameters> are one or more variable data supplied with the command. Parameters can be numeric or alphabetic. When a numeric parameter is called for, it must be entered in hexadecimal form and is limited to four hexadecimal digits (0000H through FFFFH). Larger numbers can be entered but only the four rightmost digits are used by the system. For example, the value 123456H is treated as 3456H by the system.

<CR> is the carriage return key on the keyboard.

Where a comma is shown in the syntax, you can use either a comma or a space unless otherwise noted under the individual commands.

Entry Errors

The Monitor checks for several error conditions:

- Invalid characters
- Address value errors
- Checksum errors

The Monitor checks the validity of each character entered at the Console device. As soon as it encounters an invalid character, it displays a cross-hatch (#) and aborts the command. It displays the prompt character on the next line and waits for more input:

```
.4#
```

4 is rejected because it is not a valid command.

The first character entered must be a valid command, otherwise it is rejected by the Monitor.

When the Monitor is expecting an address in hexadecimal (all addresses are entered in hexadecimal) any character other than 0-9 and A-F is rejected:

```
.D1000,1FFG#
```

G is not a valid hexadecimal digit.

Many commands require two addresses where the first address is lower than the second. If the first address is higher than the second, the operation will be performed on the single address specified as the first address. For example, there is a command that fills memory with a constant value. If you meant to say fill memory from address 900 to address 1000 with FF but entered the addresses in the opposite order:

```
.F1000,900,FF
```

The Monitor would place a FF in address 1000 and do nothing else. No indication that an error occurred is given. You will only find the error when you notice that a single byte was filled instead of 100H bytes.

Addresses are evaluated modulo 65,536. That means that the highest address that will be accepted is FFFF in hexadecimal. If addresses higher than FFFF are entered, only the last four digits will be used and the command will be executed. For example, if in the previous addresssing error example, we had entered 10000 instead of 1000:

```
.F10000,900,FF
```

The command would have been evaluated as:

```
.F0000,900,FF
```

and memory from address 0 through 900 would have been filled with FF. This is not what you wanted, in fact, you probably wiped out data you wanted. As you'll find out later in this chapter, it did wipe out some of the memory that the Monitor itself uses. No indication of this error is given except that the Monitor will not function correctly for some commands without rebooting the system.

When the Monitor detects a checksum error when reading from an input device, such as a paper tape reader, it displays a cross-hatch (#) and destroys all the data from the record in which the error occurred. Subsequent records are not read.

Command Categories

The Monitor commands are divided into seven categories:

- Monitor I/O configuration
- Memory control
- Register control
- Paper tape I/O
- Program execution
- Utility
- Diagnostics (the Z$ command, described in *Intellec Series II Installation Manual*)

Monitor I/O Configuration Commands

The Monitor has four system devices defined:

- Console
- Reader
- Punch
- List

You have the option of selecting the actual peripheral device that will perform the required function. For example, the Console device accepts commands and data and presents error and informational messages. To perform these functions you can assign a teletype console, a CRT console, a paper tape reader (in conjunction with a printer), or some non-standard device (for which you must write the driver program).

There are two Monitor commands with which you can control the I/O configuration:

- Assign (A) to change device assignment.
- Query (Q) to find what devices are currently assigned.

To make I/O device assignments, you must understand the characteristics of the system devices.

The Console is an interactive, character-orientated input and output device. To be used as a Console, a device must have all these characteristics. A teletypewriter and a CRT terminal have all these characteristics. A paper tape punch has all but the input characteristic, therefore it can't be assigned as a Console.

The Reader is a character-oriented input device that transfers data on command and notifies the calling system when no more data is available. A paper tape reader meets these qualifications.

The Punch is a character-oriented output device that accepts a character from the calling system and records it on an external medium. A paper tape punch meets these qualifications.

The List device is a character-oriented output device that accepts a character from the calling program and records it on an external medium in human readable form. A line printer meets these qualifications.

One of four actual devices can be assigned to each of the system devices. The devices for which the Monitor has driver programs are:

- Teletype console with a keyboard, printer, paper tape reader, and punch. This type of device can be assigned to all the system devices.
- CRT devices with a keyboard that are compatible with the Intellec system. This type of device can be assigned to the Console or the List device.
- High speed paper tape reader. This type of device can be assigned to the Reader device.
- High speed paper tape punch. This type of device can be assigned to the Punch device.
- Line printer. This type of device can be assigned to the List device.
- Batch. This is a non-interactive mode in which CONSOLE input is read from the assigned READER device and written to the assigned LIST device. In preparing a command file for BATCH input, the user should enter commands in exactly the same way as if the system were in interactive mode. Each command should end with a carriage return/line feed pair. The period (prompt) character which is generated by the Monitor in interactive mode should not appear as part of the command. Since the Monitor will continue to read from the READER until the CONSOLE is reassigned, the last command in the BATCH command file should reassign the CONSOLE to prevent the Monitor from reading off the end of the tape.

A - Assign Command

You can assign one physical device to a system with the Assign command. The format of the comand is:

A<logical-device>=<physical-device>

where:

A is the Assign command code.

<logical-device> specifies which of the system devices is to to be assigned a <physical-device>. The possible values for <logical-device> are:

C or CONSOLE

R or READER

P or PUNCH

L or LIST

The equal sign (=) must be entered.

<physical-device> specifies which of the physical devices is to be assigned to the <logical-device>. The possible values of <physical-device> for each <logical-device> are:

CONSOLE	T or TTY (teletype terminal)
	C or CRT (Intellec II compatible CRT terminal)
	B or BATCH (batch mode)
	1 (user-defined device for which a user-written program is present)

READER	T or TTY (teletype terminal) P or PTR (high speed paper tape reader) 1 or 2 (user-defined devices for which user-written driver programs are present)
PUNCH	T or TTY (teletype terminal) P or PTP (high speed paper tape punch) 1 or 2 (user-defined devices or which user-written driver programs are present)
LIST	T or TTY (teletype terminal) C or CRT (Intellec II compatible CRT terminal) L or LPT (line printer) 1 (user-defined device for which a user-written driver program is present)

Examples of the Assign Command: To assign a high speed paper tape reader as the system Reader device:

```
.AR=P<CR>
```

or

```
.AREADER=PTR<CR>
```

To assign a CRT terminal as the system Console device:

```
.AC=C<CR>
```

or

```
.ACONSOLE=CRT<CR>
```

Q - Query Command

The Query command displays the current status of the system I/O devices. It displays a list of the system devices and the physical devices assigned to them. The format of the Query command is:

Q

where:

Q is the Query command code. No parameters are allowed with this command.

Example of the Query Command: To list the current assignments of system devices:

```
.Q<CR>
```

The system displays:

```
C=T
R=T
P=T
L=L
```

This response indicates that a teletype terminal is assigned as the Console, Reader, and Punch devices. A line printer is assigned as the *List* device, not the Console device.

Memory Control Commands

There are four Monitor commands with which you can work with the Intellec memory. The commands that only read memory can be used on RAM as well as PROM and ROM. The commands that write to memory can only effectively be used on RAM. If you specify ROM or PROM with these commands, no error indication is given but the write portion of the command is not executed.

The Memory control commands are:

- Display (D) which displays a specified range of memory.
- Fill (F) which overlays a specified range of RAM with a constant value.
- Move (M) which copies the contents of a specified portion of memory into another RAM location.
- Substitute (S) which modifies RAM on a byte-by-byte basis.

D - Display Command

The Display command displays a section of memory formatted into lines of 16 bytes separated by spaces with the address of the first byte at the left margin.

The format of the Display command is:

D<low-address>,<high-address><CR>

where:

D is the Display command code.

<low-address> specifies the beginning of the memory range to be printed. The address must be specified in hexadecimal. <low-address> must be less than or equal to <high-address>. If <low-address> is equal to <high- address>, a single byte is printed. Also, if <low- address> is greater than <high- address>, a single byte is printed.

<high-address> specifies the end of the memory range to be printed. The address must be specified in hexadecimal.

Both parameters of the Display command are required.

If the List device is not ready, the Display command will hang. To continue, make the List device ready, or, if that isn't possible, press the interrupt 0 button on the main chassis.

Example of the Display Command: To print the contents of memory locations 109H through 12AH:

```
D109,12A<CR>
```

The system prints:

```
0109 09 0A 0B 0C 0D 0E 0F
0110 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
0120 01 02 03 04 05 06 07 08 09 0A
```

F - Fill Command

The Fill command writes a specified 1-byte constant into a specified RAM area. If ROM or PROM is specified, no error is issued, and the command continues to completion even though the memory does not change.

The format of the Fill command is:

F<low-address>,<high-address>,<constant><CR>

where:

F is the Fill command code.

<low-address> specifies the beginning of the memory range to be filled with the <constant>. The memory address must be entered in hexadecimal.

<high-address> specifies the last byte of the memory range to be filled with the <constant>. The memory address must be entered in hexadecimal.

<constant> is the byte to written to the specified address range. The <constant> must be entered as a hexadecimal number.

All three parameters of the Fill command are required.

If a character other than 0 through F is entered, the command is terminated and the prompt character (.) is displayed.

Example of the Fill Command: To initialize memory locations 20H through 2FH with 00H:

```
F20,2F,00<CR>
```

M - Move Command

The Move command copies a specified area of memory into an area of RAM. The move is done on a byte by byte basis, that is, the first byte of the specified area is copied to the new location, then the second byte is copied to the location following the first new location, and so forth. The data in the original location is not destroyed. Any data existing at the new location is overlaid.

The format of the Move command is:

M<start-address>,<end-address>,<destination-address><CR>

where:

M is the Move command code.

<start-address> specifies the address of the first byte to be moved. The address must be specified in hexadecimal.

<end-address> specifies the address of the last byte to be moved. The address must be specified in hexadecimal.

<destination-address> specifies the address to which the first byte (start- address) is to be move. Each subsequent byte is moved to a location one higher than the last.

All three parameters are required.

Because the command works on a byte by byte basis, you should be careful when attempting to move a block of data to a location within the block. By the time the command reaches the end of the block, the data will have been overlaid by the first data moved.

Examples of the Move Command: To move the data currently at address 0100H through 0200H to address 0400H through 0500H:

```
M100,200,400<CR>
```

To move the data currently at address 1000H through 1FFFH to address 1500H through 24FFH.

```
M1500,1FFF,1A00<CR>

M1000,14FF,1500<CR>
```

If you tried to do the above example with a single command:

```
M1000,1FFF,1500<CR>
```

the first 500H bytes would be copied as you expected, but the second 500H bytes would be a copy of the first 500H because bytes 1500H through 1FFFH were overlaid by the first 500H bytes.

S - Substitute Command

The Substitute command displays memory locations on an individual basis and gives you the option of modifying each location as it is displayed.

The Substitute command functions differently than most of the Monitor commands. The function of this command is performed before the carriage return (<CR>) is entered. The use of the command is described after the syntax and parameter descriptions. The format of the Substitute command is:

S<address>,[<data-byte>][,[<data-byte>]][...]<CR>

where:

S is the Substitute command code.

<address> specifies a RAM address. The address must be specified in hexadecimal.

<data-byte> specifies a single byte of data in hexadecimal that is to replace the byte currently at the location specified by address. This is an optional parameter. If it is not entered, the byte specified by address is not modified.

The Substitute command functions in the following manner:

1. Enter the command code and the address followed by a comma: S100,
2. The contents of the specified memory location followed by a dash is displayed: S100,FF-
3. You can now:
   - Modify the contents of the address by entering a new byte in hexadecimal: S100,FF-AA
   - Look at the next sequential byte of data by entering a comma: S100,FF-,00-

- End the command and not modify the data byte by pressing the carriage return key: S100,FF-<CR>
- Any combination of the first two and finally ending with a carriage return: S100,FF-AA,00-,11-22<CR>. The example changes the first byte from FFH to AAH, leaves the second byte unchanged, and changes the third byte from 11H to 22H.

Register Command

There is one command to display register contents and to modify register contents. You can display the contents of all the registers, but to modify a register, you must specify a single register with the command.

The registers you can display and modify and their symbols in the Monitor are:

A CPU A register

B CPU B register

C CPU C register

D CPU D register

E CPU E register

F CPU flag byte

H CPU H register

I Intellec interrupt mask

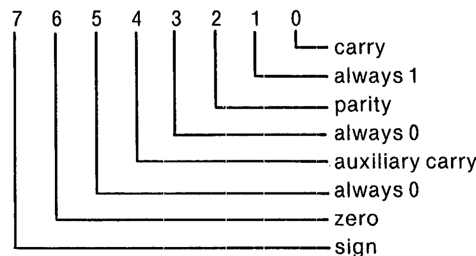
L CPU L register

M CPU H and L registers combined

P CPU program counter

S CPU stack pointer

The F register is packed with the CPU condition flags:



X - Register Command (Display Form)

This form of the Register command displays the contents of all the registers. To modify register contents, use the modify form of the command. The format of the display form of the command is:

X<CR>

where:

X is the Register command code.

No parameters are used with this form of the command.

Example of the Display Form of the Register Command: To display the contents of the Intellec registers:

```
.X<CR>
```

The system displays:

```
A=00 B=78 C=00 D=47 E=11 F=02 H=FC I=FC L=20 M=FC20 P=1024 S=CD10
```

X - Register Command (Modify Form)

The modify form of the Register command allows you to display and optionally change the contents of the registers, one at a time.

This form of the Register command functions differently than most of the Monitor commands. The function of the command is performed before the carriage return (<CR>) is entered. The use of the command is described after the syntax and parameter descriptions. The format of the modify form of the Register command is:

X <register>, [<data>] [,[<data>]] [,...] <CR>

where:

X is the Register command code.

<register> specifies a single register by letter.

<data> specifies one or two bytes of data (depending on the register) to be placed in the register. The data must be entered in hexadecimal.

The modify form of the Register command functions in the following manner:

1. Enter the command code and the register letter:

   ```
   XC
   ```

2. The contents of the specified register followed by a dash is displayed:

   ```
   XC FF-
   ```

3. You can now:

   - Modify the contents of the register by entering new hexadecimal data:

     ```
     XC FF-00
     ```

   - Look at the contents of the next sequential register by entering a comma:

     ```
     XC FF-, EE-
     ```

   - End the command and not modify the register by pressing the carriage return key:

     ```
     XC FF-<CR>
     ```

   - Any combination of the first two and finally ending with a carriage return:

     ```
     XC FF- EE,EE-,02-82<CR>
     ```

     This example changes the C register from FFH to EEH, leaves the D register unchanged, and changes the flag byte from 02H to 82H.

Examples of the Modify Form of the Register Command: To examine but not change the M register:

```
.XM 1234-<CR>
```

To examine and change the M register:

```
.XM 1234-5678<CR>
```

To examine all the registers in sequence and change the D and H registers:

```
.XA 00-,11-,22-,33-FF,44-,02-,55-FF,EC-,66-,FF66-,FC9C- ,E410- <CR>
```

To examine and change the interrupt mask (I) and the L register:

```
.XI FE-FC, 45-44<CR>
```

Paper Tape I/O Commands

The Monitor has four commands to support your usage of paper tape:

- Read (R), which reads data from a paper tape into the Intellec memory.
- Write (W), which writes data from Intellec memory to paper tape.
- End-of-File (E), which writes an end-of-file record to paper tape.
- Null Leader/Trailer (N), which writes null leader and trailer characters to paper tape.

The Intellec reads and writes paper tape in hexadecimal format. This format is described in detail in Appendix A.

R - Read Command

The Read command reads a paper tape in hexadecimal format from the device assigned as the Reader and loads the data into memory at the location specified in the record. A bias value can be specified in the command. The bias is a value that is added to the address specified in the record. The record is then loaded at a location determined by the address plus the bias value.

The format of the Read command is:

R<bias><CR>

where:

R is the Read command code.

<bias> specifies a value (modulo 65,536) to be added to the load address contained in the paper tape record. The data is loaded at the memory location specified by the record address and the bias value. The bias value must be specified in hexadecimal. Normally, a value of 0 is used.

NOTE

The addition of the bias value does not imply that the code is relocatable. In some cases, the code would not be executed at the biased location.

The data read is not changed in any way by the specification of a bias value.

Examples of the Read Command: To read a paper tape into memory:

```
.R0<CR>
```

To read a paper tape into a memory location that is 1000H above the address specified in the tape record:

```
.R1000<CR>
```

W - Write Command

The Write command punches the contents of a specified memory area to the assigned punch device. The memory area is specified by the beginning and ending addresses of the data to be punched.

The Write command does not put an end-of-file record on the paper tape. Thus you can punch non-contiguous areas of memory as a single file. The end-of-file record is written by the End-of-File command (E).

The format of the Write command is:

```
W<start-address>,<end-address><CR>
```

where:

W is the Write command code.

<start-address> specifies the first memory location to be punched onto the tape. <start-address> must be specified in hexadecimal.

<end-address> specifies the last memory location to be punched into the tape. <end-address> must be specified in hexadecimal. <end-address> must be higher than <start-address>.

The final record in a paper tape file must be an end-of-file record. After you have written the last memory area to tape, you must write an end-of-file record with the End-of-File command.

Examples of the Write Command: To write the contents of memory locations 200H through 3AFH to paper tape:

```
.W200,3AF<CR>
```

To write the contents of memory locations 450H through 54FH and locations 1000H through 1FFFH to paper tape as a single file:

```
.W450,54F<CR>

.W1000,1FFF<CR>
```

E - End-of-File Command

Every paper tape file must have an end-of-file record as the last record. If the end-of-file record is missing, the reader will read off the end of the tape.

You can specify an entry point address in the end-of-file record written with the End-of-File command. The entry point address is the address of the first instruction in the program to be executed. When this address is specified in the end-of-file record, the address is loaded into the Intellec program counter when the tape is read with a Read command. The program can then be executed by a simple Execute (G) command (described later). If the load address field is 0, the program counter is not altered by the Read command.

The format of the End-of-File command is:

E<entry-point><CR>

where:

E is the End-of-File command code.

<entry-point> specifies the entry point address of the program in the file to which the end-of-file record is being added. <entry-point> must be specified in hexadecimal. Of course an <entry-point> value does not make any sense unless the file contains executable code. A zero should be specified if an entry point address is not wanted.

Examples of the End-of-File Command:

To punch an end-of-file record in a tape that has just been written and specify an entry point address to be used when the tape is read with a Read command:

```
.E0<CR>
```

To punch an end-of-file record in a tape that has just been written and specify an entry point address to be used when the tape is read with a Read command:

```
.E1000<CR>
```

N - Null Command

The Null command punches a 60 null character leader or trailer. The null character is a 00H. You should punch a leader before writing data to a tape and after the end-of-file record. It makes the tape easier to load and saves the data on the tape from the usual damage that tape ends incur through normal handling.

The format of the Null command is:

N<CR>

where:

N is the Null command code.

Example of the Null Command: To punch a leader or trailer in a paper tape:

```
.N<CR>
```

Execute Command

The Execute command lets you transfer control of the Intellec system from the Monitor to your own program. You can specify the starting address and one or two breakpoints with the command. The starting address is optional. If it isn't specified in the command, the address in the program counter will be used. In the following conditions, you can be sure that the desired address is in the program counter:

- You interrupted your program with the interrupt 0 switch on the front of the Intellec and you have done nothing to destroy the program's registers.
- You loaded the program from paper tape and the end-of-file record contained the entry point address. This entry point address is loaded into the program counter by the Monitor.
- You modified the program counter to your entry point address with the Register command.
- Your program returned control to the Monitor because a breakpoint was encountered.

A breakpoint is the address of an instruction within your program that, if fetched, results in the return of control to the Monitor. You may want to do this so you can check the contents of certain registers or data fields at this point in your program. When a breakpoint is reached, the instruction at the address is not executed before control is returned to the Monitor. The instruction at the breakpoint is executed when you return control to your program with the Execute command.

There are two important points you must know when using breakpoints to debug and test your program:

- In saving the CPU status for your program, the Monitor uses the top 12 bytes of your program stack. This pushes the status of your registers and program counter into the stack. You should be aware of this when examining the stack. When control is returned to your program, your registers are restored and the stack pointer is reset as if the breakpoint had never occurred.
- The Intellec interrupt system is enabled when the Monitor is entered. The Monitor cannot determine the state of the interrupt system just prior to exit from your program. It is assumed that the interrupt system was enabled and so interrupts are enabled when control is returned to your program. It is your responsibility to either enable or disable the interrupt system.

G - Execute Command

The Execute command gives control of the Intellec to the program at the address specified or implied in the command. It optionally sets one or two breakpoints in the program to which control is passed.

If breakpoints are specified, the Execute command functions differently than most of the Monitor commands. Before the carriage return is entered, the Monitor prompts for breakpoints if a comma is entered after typing G. This use of the command is described after the syntax and parameter descriptions. The format of the Program Execute command is:

G[<start-address>][,<breakpoint1>][,<breakpoint2>]<CR>

where:

G is the Execute command code.

<start-address> specifies the address to be placed in the program counter. Control of the Intellec is passed to this address. The address must be specified in hexadecimal.

<breakpoint1> and <breakpoint2> specify points in the program where control will be passed back to the Monitor. The breakpoints are entered in hexadecimal.

When breakpoints are specified, command entry is in the following sequence:

1. Enter the command code and, optionally, the start-address followed by a comma:

   ```
   G1FA,
   ```

2. The Monitor displays a dash:

   ```
   G1FA,—
   ```

3. Enter the first breakpoint address:

   ```
   G1FA,—22C
   ```

4. Enter another comma if a second breakpoint is to be specified or a carriage return if only one breakpoint is wanted. If a second breakpoint is entered, follow it with a carriage return:

   ```
   G1FA,—22C,—24E<CR>
   ```

   or

   ```
   G1FA,—22C<CR>
   ```

If the command contains a syntax error, no breakpoints are set. The command must be re-entered and the breakpoints again specified.

When either of the breakpoints are reached and control is returned to the Monitor, or when control is returned to the Monitor because of an interrupt 0, both breakpoints are eliminated. If you want them when you resume execution of your program you must specify them again.

Examples of the Execute Command: To pass control to the program address in the program counter:

```
.G<CR>
```

To pass control to the program whose entry point is 30A:

```
.G30A<CR>
```

To pass control to the program whose entry point is 30A and to set a breakpoint at address 400 within that program:

```
.G30A,—400<CR>
```

To pass control to the program whose entry point is 30A and to set two breakpoints, at addresses 400 and 500, within that program:

```
.G30A,—400,—500<CR>
```

Utility Command

There is one utility command that does hexadecimal addition and subtraction for you. Many of your early errors in programming will come from mistakes in performing hexadecimal arithmetic.

H - Hexadecimal Command

The Hexadecimal command adds and subtracts two hexadecimal numbers for you. The numbers can be up to and including four hexadecimal digits. The format of the Hexadecimal command is:

H<number1>,<number2><CR>

where:

H is the Hexadecimal command code.

<number1> specifies the first number to be added. This is number is used as minuend for the subtraction. The number must be entered in hexadecimal.

<number2> specifies the second number to be added. This number is used as the subtrahend for the subtraction.

The command displays two four-digit values. The first is the addition of the two numbers and second is the subraction. Negative numbers must be entered in their twos complement form.

If more than four digits are entered, the command uses the rightmost four digits. The leading digits are lost.

Example of the Hexadecimal Command: To add and subtract E49 (minuend) and 111 (subtrahend):

```
.HE49,111<CR>
```

CHAPTER 7
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.

Interrupts 0, 1, and 2 are reserved for internal use and cannot 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 be 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
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
```

places the current value of the Interrupt Mask Register into the A-register.

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 the new interrupt.

When the new interrupt can be serviced all interrupts from the Multibus 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:

```
<remove interrupt signal from external device>
 DI                           ;DISABLE INTERRUPTS
 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 1.

```
                ASEG                ;VECTOR GOES AT ABSOLUTE
                                    ;LOCATION
                ORG       8         ;RST ADDRESS FOR INTERRUPT 1
                JMP       INT1
                CSEG                ;PUT CODE IN RELOCATABLE CODE
                                    :SEGMENT
INT1:
                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
```

APPENDIX A
HEXADECIMAL PAPER TAPE FORMAT

Object code is stored on paper tape in an ASCII representation of the program in memory. The code is blocked into records, each of which contains the record type, length, type, memory load address, and checksum in addition to the data. Figure A-1 shows the frames of a tape record.

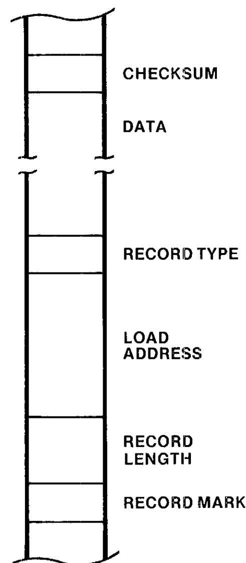


Figure A-1. Paper Tape Record Format

The Record Mark is a colon (3AH) and is used to signal the start of a record.

The Record Length is the count of the data bytes in the record. A record length of zero indicates end-of-file.

The Load Address specifies the address at which the first data byte will be loaded. The successive data bytes will be stored in successive memory locations.

The Record Type specifies the type of this record. All data records are type 0. End-of-file records can be type 0 or 1.

The Data consists of two frames per memory word. The data is represented by hexadecimal values 00H through FFH.

The Checksum is the negative of the sum of all 8-bit bytes in the record, beginning with the Record Length and ending with the last Data byte, evaluated modulo 256. The sum of all bytes in the record (including the checksum) should be zero.

APPENDIX B
HEXADECIMAL-DECIMAL CONVERSION

The following table is for hexadecimal to decimal and decimal to hexadecimal conversion. To find the decimal equivalent of a hexadecimal number, locate the hexadecimal number in the correct position and note the decimal equivalent. Add the decimal numbers.

To find the hexadecimal equivalent of a decimal number, locate the next lower decimal number in the table and note the hexadecimal number and its position. Subtract the decimal number from the table from the starting number. Find the difference in the table. Continue this process until there is no difference.

BYTE				BYTE			
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,548	E	224	E	14
F	61,440	F	3,840	F	240	F	15

APPENDIX C
ERROR MESSAGES

This appendix lists the error messages issued by the various ISIS-II commands.

The numbered messages are listed in the first section. The unnumbered messages that are issued by specific commands are listed in subsequent sections.

Numbered ISIS-II Error Messages

By convention, error numbers 1-99 inclusive are reserved for errors that originate in or are detected by the resident routines of ISIS-II; error numbers 100-199 inclusive are reserved for user programs; and numbers 200-255 inclusive are used for errors that may be encountered by nonresident system routines. In the following list an asterisk precedes errors that are always fatal. The other errors are generally nonfatal unless they are issued by the CONSOL system call. See Tables C-1 and C-2.

0 No error detected.
* 1 Limit of 19 buffers exceeded.
2 AFTN does not specify an open file.
3 Attempt to open more than six files simultaneously.
4 Illegal filename specification.
5 Illegal or unrecognized device specification in pathname.
6 Attempt to write to a file open for input.
* 7 Operation aborted; insufficient disk space.
8 Attempt to read from a file open for output.
9 No more room in disk directory.
10 Pathnames do not specify the same disk.
11 Cannot rename file; name already in use.
12 Attempt to open a file already open.
13 No such file.
14 Attempt to open for writing or to delete or rename a write-protected file.
*15 Attempt to load into ISIS-II area or buffer area.
16 Illegal format record.
17 Attempt to rename/delete a non-disk file.
*18 Unrecognized system call.
19 Attempt to seek on a non-disk file.
20 Attempt to seek backward past beginning of file.
21 Attempt to rescan a non-lined file.
22 Illegal ACCESS parameter to OPEN or access mode impossible for file specified.
23 No filename specified for a disk file.
*24 Disk error (see below).
25 Incorrect specification of echo file to OPEN.
26 Incorrect second parameter in ATTRIB system call.
27 Incorrect MODE parameter in SEEK system call.
28 Null file extension.
*29 End of file on console input.
*30 Drive not ready.
31 Attempted seek on write-only (output) file.
32 Can't delete an open file.
*33 Illegal system call parameter.
34 Bad third parameter in LOAD system call.
35 Attempt to extend a file opened for input by seeking past end-of-file.

201 Unrecognized switch.
202 Unrecognized delimiter character.
203 Invalid command syntax.
204 Premature end-of-file.
206 Illegal disk label.
207 No END statement found in input.
208 Checksum error.
209 Illegal records sequence in object module file.
210 Insufficient memory to complete job.
211 Object module record too long.
212 Bad object module record type.
213 Illegal fixup record specified in object module file.
214 Bad parameter in a SUBMIT file.
215 Argument too long in a SUBMIT invocation.
216 Too many parameters in a SUBMIT invocation.
217 Object module record too short.
218 Illegal object module record format.
219 Phase error in LINK.
220 No end-of-file record in object module file.
221 Segment overflow during Link operation.
222 Unrecognized record in object module file.
223 Fixup record pointer is incorrect.
224 Illegal record sequence in object module file in LINK.
225 Illegal module name specified.
226 Module name exceeds 31 characters.
227 Command syntax requires left parenthesis.
228 Command syntax requires right parenthesis.
229 Unrecognized control specified in command.
230 Duplicate symbol found.
231 File already exists.
232 Unrecognized command.
233 Command syntax requires a "TO" clause.
234 File name illegally duplicated in command.
235 File specified in command is not a library file.
236 More than 249 common segments in input files.
237 Specified common segment not found in object file.
238 Illegal stack content record in object file.
239 No module header in input object file.
240 Program exceeds 64K bytes.

When error number 24 occurs, an additional message is output to the console:

```
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 floppy 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 Protected drive.
40 Drive has indicated a Write error.
80 Drive not ready.

For hard disks, nn has the following meanings:

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.

Table C-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 C-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.

Link Error Messages

Fatal Error Messages

Fatal errors encountered by LINK cause a message to be sent to the console output device. LINK terminates processing and returns control to ISIS-II.

The errors that are caused by improper command entry are followed by a partial image of the command with a cross hatch (#) in the vicinity of the error.

INVALID SYNTAX
partial command image

There is an error in the command as entered. The error may be an unrecognized keyword, a missing comma, or a non-blank character following an ampersand.

DUPLICATE FILE NAME
partial command image

The same file name is specified both as an input and output file.

'TO' EXPECTED
partial command image

The command syntax requires a "TO" clause to specify the output file.

LEFT PARENTHESIS EXPECTED
partial command image

A PUBLICS, NAME, or PRINT keyword is not followed by a "(".

RIGHT PARENTESIS EXPECTED
partial command image

The list following a PUBLICS, NAME, or PRINT keyword is not ended with a ")".

INVALID NAME
partial command image

A module name contains an illegal character.

NAME TOO LONG
partial command image

A module name greater than 31 characters was encountered in the command.

UNRECOGNIZED CONTROL
partial command image

A character string other than NAME, MAP, or PRINT was encountered when a control keyword was expected.

filename, PREMATURE EOF
filename, CHECKSUM ERROR
filename, RECORD TOO LONG
filename, ILLEGAL RELO RECORD
filename, FIXUP BOUNDS ERROR
filename, ILLEGAL RECORD FORMAT
filename, NO EOF
filename, BAD RECORD SEQUENCE
filename, ILLEGAL STACK CONTENT RECORD
filename, NO MODULE HEADER RECORD

There is an error in the internal format of the specified file. This error may be the result of a misspelled file name. The error may have occurred when the language translator produced the module or in a previous LINK, in which case you should translate the source module again and relink. If the problem persists, it should be reported on a Software Problem Report (SPR).

filename, NOT A LIBRARY

The file specified in the message was used with a list of module names in the LINK command, therefore it was expected to be a library file. The file actually was not a library file. This could be caused by a misspelled file name.

filename, SEGMENT TOO LARGE

An output segment is greater than 64K bytes. The segment portion in the file specified in the message cannot be added to the output segment because it would then be greater than 64K.

filename, INSUFFICIENT MEMORY

LINK cannot generate the output file specified in the message because there is insufficient memory for its work space, which is needed primarily for the symbol table.

Non-Fatal Error Messages

Nonfatal errors issued by LINK are written to the map file and the current console output device (if different). LINK completes processing before returning to ISIS-II.

MORE THAN 1 MAIN MODULE, CONFLICT IN modname

LINK found more than one main module in the input list. The module named in the message is found after another main module was detected previously. All main modules are included in the output module, but the starting address of the output module is taken from the first main module in the input list.

name - MULTIPLY DEFINED, DUPLICATE IN modname

The public name specified in the message was defined in more than one module, the second definition was detected in the module specified in the message.

MODULE NOT IN LIBRARY, LOOKING FOR filename (modname)

The module in the input list was not found in the specified library.

/name/ - UNEQUAL COMMON LENGTH, CONFLICT IN modname

Two named common segments with the same name but different lengths were encountered. The module containing the conflicting definition is specified in the message.

LOCATE Error Messages

Fatal Error Messages

Fatal errors encountered by LOCATE cause a message to be sent to the console output device. LOCATE terminates processing and returns control to ISIS-II.

The errors that are caused by improper command entry are followed by a partial image of the command with a cross hatch (#) in the vicinity of the error.

INVALID SYNTAX
partial command image

There is an error in the command as entered. The error may be an unrecognized keyword, a missing comma, or a non-blank character following an ampersand.

'TO' EXPECTED
partial command image

The "TO" clause was not specified in the LOCATE command and the input file name does not contain an extension.

LEFT PARENTHESIS EXPECTED
partial command image

A COLUMNS, ORDER, START, STACKSIZE, CODE, DATA, STACK, MEMORY, /common name/, //, NAME, or PRINT keyword is not followed by a "(".

RIGHT PARENTHESIS EXPECTED
partial command image

The list following a COLUMNS, ORDER, START, STACKSIZE, CODE, DATA, STACK, MEMORY, /common name/, //, NAME, or PRINT keyword is not ended with a ")".

INVALID NAME
partial command image

A module name or /common name/ contains an illegal character.

NAME TOO LONG
partial command image

A module name or /common name/ greater than 31 characters was encountered in the command.

common name, COMMON NOT FOUND

The input module does not contain the common segment that was specified in the command.

UNRECOGNIZED CONTROL
partial command image

A character string other than NAME, MAP, PRINT, COLUMNS, SYMBOLS, LINES, PUBLICS, PURGE, ORDER, CODE, DATA, STACK, MEMORY, /, RESTART0, START, or STACKSIZE was encountered when a control keyword was expected.

filename, PREMATURE EOF
filename, CHECKSUM ERROR
filename, RECORD TOO LONG
filename, ILLEGAL RELO RECORD
filename, FIXUP BOUNDS ERROR
filename, ILLEGAL RECORD FORMAT
filename, NO EOF
filename, BAD RECORD SEQUENCE
filename, ILLEGAL STACK CONTENT RECORD
filename, NO MODULE HEADER RECORD

There is an error in the internal format of the specified file. This error may be the result of a misspelled file name. The error may have occurred when the language translator produced the module or in a previous LINK, in which case you should translate the source module again and re-LOCATE. If the problem persists, it should be reported on a Software Problem Report (SPR).

filename, PROGRAM EXCEED 64K

The output module to be placed in the output file exceeds the maximum of 64K bytes.

filename, INSUFFICIENT MEMORY

LOCATE cannot process the input file specified in the message because there is insufficient memory for its work space.

Non-fatal Error Messages

Non-fatal error messages issued by LOCATE are written to the map file and the current console output device (if different). LOCATE completes processing before returning the ISIS-II

INPAGE SEGMENT > 256 BYTES COERCED TO PAGE BOUNDARY

If this message is issued, a segment marked as inpage relocatable is greater than 256 bytes. Relocation types are described in Chapter 4.

UNSATISFIED EXTERNAL(n) external name

This message is printed when an unsatisfied external name is encountered in the input file. The number (n) is the running count of the number of unsatisfied external names encountered previously, it is also used in the following message when a reference is made to the unsatisfied external name.

REFERENCE TO UNSATISIFIED EXTERNAL(n) AT xxxxH

A reference was made to the unsatisfied external name identified by the count (n) at the address specified.

(MEMORY OVERLAP FROM xxxxH THROUGH xxxxH)

This message is issued if the same memory location is defined in more than one program segment.

Lib Error Messages

All LIB command error messages are nonfatal because LIB is an interactive program. The command (ADD, CREATE, DELETE, EXIT, or LIST) that caused the error is aborted.

The errors that are caused by improper command entry are followed by a partial image of the command with a cross hatch (#) in the vacinity of the error.

INSUFFICIENT MEMORY

There is not enough memory available for execution of the command.

INVALID MODULE NAME
partial command image

A module name in the command is invalid. It may have in illegal first character.

INVALID SYNTAX
partial command image

There is an error in the command. Check for the following:

- Misspelled keywords.
- Ampersand followed by a non-blank character.
- ADD: TO <filename> not followed by a <CR>.
- DELETE: <libname>(<modname>) not followed by a <CR>.
- DELETE: <modname> not specified.
- CREATE: <filename> not followed by <CR>.
- LIST: TO <filename> not followed by PUBLICS or <CR>.

FILE ALREADY EXISTS

The file specified in a CREATE command already exists.

filename, BAD RECORD SEQUENCE

The file specified in the command has an unexpected record sequence. It may not be terminated with an EOF record. You may have attempted to ADD a non-object or non-library file to a library.

filename, CHECKSUM ERROR

The specified file contains a record that has an invalid checksum. Go back and generate the file again.

filename, DUPLICATE SYMBOL IN INPUT

You have attempted to ADD a file that contains a PUBLIC symbol that already exists in the library.

filename, ILLEGAL RECORD FORMAT

The file specified in the command has an illegal format. The object file may contain a name that has more than 31 characters. The file may contain records in an improper order.

filename(modname): NOT FOUND

You have attempted to delete a non-existant module. You may have misspelled the name.

filename, NOT LIBRARY

The specified file is not a library.

filename, OBJECT RECORD TOO SHORT

The specified file contains a record of insufficient length.

filename, PREMATURE EOF

The EOF record occurred before the length of the file indicated it should.

LEFT PARENTHESIS EXPECTED
partial command image

There is a missing left parenthesis "(" in the command.

modname- ATTEMPT TO ADD DUPLICATE MODULE

The specified module already exists in the library.

MODULE NAME TOO LONG
partial command image

The specified module name exceeds 31 characters.

RIGHT PARENTHESIS EXPECTED
partial command image

There is a missing right parenthesis ")" in the command.

symbol- PUBLIC SYMBOL ALREADY IN LIBRARY

You attempted to add a module that contains a PUBLIC symbol that already exists in the library.

'TO' EXPECTED
partial command image

The TO file is not specified in the ADD command.

UNRECOGNIZED COMMAND

An illegal or misspelled command (i.e., not ADD, CREATE, DELETE, EXIT, or LIST) was entered.

Editor Error Messages

The Editor issues the following error messages:

"n" ILLEGAL IN THIS CONTEXT

The "n" represents an alphanumeric character that is not an Editor command. Any commands that precede the illegal character are executed. Any commands that follow the illegal character are not executed.

CANNOT FIND "text"

The "text" represents a search argument in an F or S command that was not found in the file, from the pointer location to the end. When this message is displayed, the pointer location is not changed by the F or S command. If any commands follow the F or S command in a command string, they are not executed.

ITERATION STACK FAULT

This message is issued when command string iterations are nested more than eight deep. None of the nested commands are executed. Any commands that preceded the first iteration string are executed. Any commands that follow the outer iteration string are not exectuted.

APPENDIX D
ISIS-II SAMPLE PROGRAMS—TYPE

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 :C0: device by specifying:

```
TYPE filename
```

rather then

```
COPY filename to :C0:
```

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 E ASCII CODES

Table E-1. ASCII Code List

Decimal	Octal	Hexadecimal	Character	Decimal	Octal	Hexadecimal	Character
0	000	00	NUL	64	100	40	@
1	001	01	SOH	65	101	41	A
2	002	02	STX	66	102	42	B
3	003	03	ETX	67	103	43	C
4	004	04	EOT	68	104	44	D
5	005	05	ENQ	69	105	45	E
6	006	06	ACK	70	106	46	F
7	007	07	BEL	71	107	47	G
8	010	08	BS	72	110	48	H
9	011	09	HT	73	111	49	I
10	012	0A	LF	74	112	4A	J
11	013	0B	VT	75	113	4B	K
12	014	0C	FF	76	114	4C	L
13	015	0D	CR	77	115	4D	M
14	016	0E	SO	78	116	4E	N
15	017	0F	SI	79	117	4F	O
16	020	10	DLE	80	120	50	P
17	021	11	DC1	81	121	51	Q
18	022	12	DC2	82	122	52	R
19	023	13	DC3	83	123	53	S
20	024	14	DC4	84	124	54	T
21	025	15	NAK	85	125	55	U
22	026	16	SYN	86	126	56	V
23	027	17	ETB	87	127	57	W
24	030	18	CAN	88	130	58	X
25	031	19	EM	89	131	59	Y
26	032	1A	SUB	90	132	5A	Z
27	033	1B	ESC	91	133	5B	[
28	034	1C	FS	92	134	5C	\
29	035	1D	GS	93	135	5D	]
30	036	1E	RS	94	136	5E	∧
31	037	1F	US	95	137	5F	—
32	040	20	SP	96	140	60	'
33	041	21	!	97	141	61	a
34	042	22	"	98	142	62	b
35	043	23	#	99	143	63	c
36	044	24	$	100	144	64	d
37	045	25	%	101	145	65	e
38	046	26	&	102	146	66	f
39	047	27	'	103	147	67	g
40	050	28	(	104	150	68	h
41	051	29	)	105	151	69	i
42	052	2A	*	106	152	6A	j
43	053	2B	+	107	153	6B	k
44	054	2C	,	108	154	6C	l
45	055	2D	−	109	155	6D	m
46	056	2E	.	110	156	6E	n
47	057	2F	/	111	157	6F	o
48	060	30	0	112	160	70	p
49	061	31	1	113	161	71	q
50	062	32	2	114	162	72	r
51	063	33	3	115	163	73	s
52	064	34	4	116	164	74	t
53	065	35	5	117	165	75	u
54	066	36	6	118	166	76	v
55	067	37	7	119	167	77	w
56	070	38	8	120	170	78	x
57	071	39	9	121	171	79	y
58	072	3A	:	122	172	7A	z
59	073	3B	;	123	173	7B	{
60	074	3C	<	124	174	7C	|
61	075	3D	=	125	175	7D	}
62	076	3E	>	126	176	7E	~
63	077	3F	?	127	177	7F	DEL

Table E-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

INDEX

*, 3-41
// 4-14
/common name/, 4-14
?, 3-7
:, 2-4
:BB:, 2-5
:CI:, 2-4
:CO:, 2-4
:Fn:, 2-4
:HP:, 2-4
:HR:, 2-4
:I1:, 2-5
:L1:, 2-5
:LP:, 2-4
:O1:, 2-5
:P1:, 2-5
:P2:, 2-5
:R1:, 2-5
:R2:, 2-5
:TI:, 2-4
:TO:, 2-4
:TP:, 2-4
:TR:, 2-4
:VI:, 2-4
:VO:, 2-4

A command
  Editor, 3-53
  Monitor, 6-4
A switch of FORMAT command, 3-13
aborting commands, Editor, 3-41
absolute binary format, 2-3
absolute code
  converted from binary code, 3-38
  converted from hexadecimal code, 3-39
absolute object module, 4-3
absolute information, 4-3
accepting interrupts, 7-2
accessing
  devices, 2-5
  files, 2-5
ADD command, 4-18
addressing relative, 4-4
AFTN, 5-8
aids, user, 3-1
angle brackets, 3-54
append command, 3-53
arithmatic with hex, 6-16
ASCII Codes, E-1
ASEG, 4-3
ASM80, 4-1
assembler, 1-2
assembler and relocation, 4-21
assembly language calls, 5-6
assign command, 6-4
asterisk (*), 3-41
ATTRIB command, 3-37
ATTRIB system call, 5-20
attributes, 3-6
attributes, change
  from console, 3-37
  with system call, 5-21

B command, 3-44
B switch of COPY command, 3-31
base address, 4-2
BB (byte bucket), 2-5
beginning of text command, 3-44
binary code, converting to absolute
  object, 3-38
BINOBJ command, 3-38
blocks, 3-5
BOOT switch, 2-7
breakpoint, 6-14
buffer use by SUBMIT, 3-25
byte bucket (:BB:), 2-5
byte relocation, 4-22

C command, 3-45
C switch of COPY command, 3-21
calls, system
  ATTRIB, 5-20
  CI, 5-28
  CLOSE, 5-15
  CO, 5-29
  CONSOL, 5-22
  CSTS, 5-37
  DELETE, 5-18
  ERROR, 5-24
  EXIT, 5-26
  IOCHK, 5-40
  IODEF, 3-39
  IOSET, 5-42
  LO, 5-32
  LOAD, 5-25
  MEMCHK, 5-42
  OPEN, 5-8
  PO, 5-31
  READ, 5-9
  RENAME, 5-19
  RESCAN, 5-14
  RI, 5-30
  SEEK, 5-12
  SPATH, 5-16
  UI, 5-33
  UO, 5-35
  UPPS, 5-38
  WHOCON, 5-23
  WRITE, 5-10
care of disks, 2-6
carriage return and line feed
  characters, 3-42
change console device, 5-22, 5-42
change disk file attributes, 5-20
change disk file name, 5-19
changing disk file attributes, 3-37
character command, 3-45
characteristics, recording disk, 3-5

CI (device), 2-4
CI (system call), 5-28
CLOSE, 5-15
closing files, 2-5, 5-16
CO (device), 2-4
CO (system call), 5-29
CODE, 4-13
code conversion commands
  BINOBJ, 3-38
  HEXOBJ, 3-39
  OBJHEX, 3-39
CODE segment, 4-2
COLUMNS, (number), 4-12
command
  aborting, Editor, 3-41
  categories, Monitor, 6-3
  entry, Editor, 3-41
  entry, Monitor, 6-1
  line, reading, 5-4
  parameters, Editor, 3-41
  sequence definition file, 3-24
  string iterations, 3-54
  syntax
    Editor, 3-42
    ISIS-II, 3-10
    Monitor, 6-1
commands
  ATTRIB, 3-37
  BINOBJ, 3-38
  COPY, 3-28
  DEBUG, 3-23
  DELETE, 3-34
  DIR, 3-26
  EDIT, 3-40
  Editor
    A, 3-53
    B, 3-44
    C, 3-45
    D, 3-49
    E, 3-51
    F, 3-46
    I, 3-47
    K, 3-49
    L, 3-45
    M, 3-53
    Q, 3-52
    S, 3-48
    T, 3-50
    W, 3-52
    Z, 3-44
  FIXMAP, 3-14
  FORMAT, 3-12
  HDCOPY, 3-32
  HEXOBJ, 3-39
  IDISK, 3-10
  LIB, 4-17
    ADD, 4-18
    CREATE, 4-18
    DELETE, 4-18
    EXIT, 4-19
    LIST, 4-19
  LINK, 4-7
  LOCATE, 4-10
  Monitor
    A, 6-4
    D, 6-6
    E, 6-12
    F, 6-7
    G, 6-14
    H, 6-16
    M, 6-7
    N, 6-13
    Q, 6-5
    R, 6-11
    S, 6-8
    W, 6-12
    X, 6-9, 6-10
  OBJHEX, 3-39
  RENAME, 3-36
  SUBMIT, 3-24
common segments 4-3
compiler, 1-2
composition, system, 2-1
configuration commands, Monitor, 6-3
Configurations, supported, 2-1
CONSOL, 5-22
console
  assignment, 5-22, 5-42
  changing, 6-4
  current, 3-2
  initial system, 3-2
  input, 5-28
  input status, 5-37
  output, 5-32
  resuming output, 3-2
  stopping output, 3-2
  system, 3-2
  user aids, 3-1
content, disk directory, 3-6
continuation lines, 4-8, 4-11
control characters
  control/c, 3-41
  control/e, 3-24
  control/i, 3-43, 3-47
  control/p, 3-1 ,5-4
  control/q, 3-2
  control/r, 3-1 ,5-4
  control/s, 3-2
  control/x, 3-1 ,5-4
  control/z, 3-2 ,5-4
control descriptions
  LINK, 4-7
  LOCATE, 4-11
control/c, 3-41
control/e, 3-24
control/i, 3-43, 3-47
control/p, 3-1 ,5-4
control/q, 3-2
control/r, 3-1 ,5-4
control/s, 3-2
control/x, 3-1 ,5-4
control/z, 3-2 ,5-4
conversion commands
  BINOBJ, 3-38
  HEXOBJ, 3-39
  OBJHEX, 3-39
converting
  absolute object code to
    hexadecimal, 3-39
  binary code to absolute object code, 3-38
  hexadecimal code to absolute object
    code, 3-39

COPY command. 3-28
COPY command switches, 3-30
copying a disk file, 3-28
copying a hard disk file, 3-32
COUNT, 3-19
CPU registers, 6-9
CREATE, 4-18
creation and mangement of files, 3-1
CRT terminal, 1-1
CSD, 3-24
CSEG, 4-2
CSTS, 5-37
current system console, 3-2

D command
  Editor, 3-49
  Monitor,6-6
DATA, 4-14
data segment, 4-2
data, reading from disk, 3-53
DEBUG command, 3-23
debug mode, 3-3
debug toggle, 3-3
debugging and processing errors, 3-3
decimal to hexadecimal conversion, B-1
DELETE (LIB command), 4-18
DELETE (system call), 5-18
DELETE command, 3-34
delete characters, 3-49
delete file from directory, 5-18
deleting a disk file, 3-34
descriptions of LOCATE controls
  //, 4-14
  /common name/, 4-14
  CODE, 4-14
  COLUMNS (number), 4-12
  DATA, 4-14
  LINES, 4-13
  MAP, 4-11
  MEMORY, 4-14
  NAME, 4-14
  NOPRINT, 4-12
  ORDER, 4-13
  PRINT, 4-12
  PUBLICS, 4-13
  PURGE, 4-13
  RESTART0, 4-15
  STACK, 4-14
  STACKSIZE, 4-15
  START, 4-15
  SYMBOLS, 4-12
determine console assignment 5-23, 5-40
determining memory space
  from console, 3-53
  from program, 5-42
device
  accessing, 2-5
  designations, 2-4
  management, 2-3
  name format, 2-3
  names, system, 2-4
different source languages, using, 4-4
DIR command, 3-26
directory
  content, disk, 3-6
  listing command, 3-26
  maintenance calls, 5-19
disk
  addressing, 2-5
  care, 2-6
  directory content, 3-6
  directory listing command, 3-26
  directory maintenance calls, 5-19
  drives, 1-1, 2-1
  errors, 3-3
  file attribute changing, 3-37
  file copying, 3-28
  file deletion, 3-34
  file marker, positioning, 5-12
  file renaming, 3-36
  files, 1-2
  format, 3-10, 3-12
  formatting commands, 3-10, 3-12
  hard platter, 2-2
  initialization command, 3-10
  maintenance commands, 3-10
  operating system, 1-1
  organization, 3-5
  recording characteristics, 3-5
  reading data, 3-53
display command, 6-6
double density disks, 2-1
DSEG, 4-2

E command
  Editor, 3-51
  Monitor, 6-12
easier debugging and program
  modification, 4-4
echo file, 5-8
EDIT command, 3-40
editing, 2-3
editing characters, 5-4
editing, line, 3-1
Editor
  command aborting, 3-41
  command entry, 3-41
  command parameters, 3-41
  commands
    A, 3-53
    B, 3-44
    C, 3-45
    D, 3-49
    E, 3-51
    F, 3-46
    I, 3-47
    K, 3-49
    L, 3-45
    M, 3-53
    Q, 3-52
    S, 3-48
    T, 3-50
    W, 3-52
    Z, 3-44
  error messages, C-9
  formatting characters, 3-41
  functions, 3-40
  activating, 3-42
  saving your work, 3-51

terminating a session, 3-51
using, 3-40
end of file command, 6-7
end of file control, 3-2
end of text command, 3-44
entry error, Monitor, 6-2
ERROR, 5-24
error message output, 5-24
error messages
Editor, C-9
ISIS-II, C-1
LIB, C-7
LINK, C-4
LOCATE, C-5
error processing and debugging, 3-3
ERRORS, 3-21
execute command, 6-14
executing the Editor, 3-40
execution commands, program, 3-22
execution of program under Monitor, 3-23
execution, non-interactive, 3-23
EXIT (FIXMAP), 3-20
EXIT (LIB command), 4-19
EXIT (system call), 5-26
exit command, 3-51
extension, 2-4
external references and public symbols, 4-5

F command
Editor, 3-46
Monitor, 6-7
F switch of DIR command, 3-27
F0 parameter of ATTRIB command, 3-37
F1 parameter of ATTRIB command, 3-37
faster program development, 4-3
fatal errors, 3-3
file
accessing, 2-5
attribute changing, 3-37
attributes, 3-6
blocks, 3-6
character coding, 3-5
control commands, 3-26
copying, 3-28
creation and management, 3-1
deletion, 3-34
extensions, 2-4
from memory, 5-9
input/output calls, 5-7
length, 3-6
marker, positioning, 5-12
name format, 2-3
name, change, 3-36, 5-19
names, wild card, 3-7
renaming, 3-36, 5-19
saving, 3-51
to memory, 5-9
types, 2-4
typing, 3-50
filename, 3-6
filename command, 3-22
filename format, 2-4
files, disk, 1-2
files, initializing for I/O, 5-8
fill command, 6-7
find command, 3-46
flexible disk care, 2-6
start-up, 2-8
FIXMAP, 3-14
commands, 3-15
Mark, 3-16
Free, 3-17
List, 3-18
Count, 3-19
Record, 3-19
Quit, 3-20
Exit, 3-20
Errors, 3-21
FOR parameter of DIR command,3-26
format attribute, 3-6
FORMAT command, 3-12
format of filename, 2-3
format of ISIS-II commands, 3-10
format, name, 2-4
formatting characters, Editor 3-42
formatting nonsystem disks, 3-9
formatting system disks, 3-10, 3-12
FORT80, 4-1
FREE, 3-17
front panel, 2-7
functions of the Editor, 3-40

G command, 6-14
gap, 4-9
getting started with ISIS-II, 2-1
go command, 6-14

H and L registers and memory pages, 4-21
H command, 6-16
hard disk, 2-2
hardware configuration, 1-1
HDCOPY command, 3-32
hexadecimal
arithmatic, 6-16
code, converting to absolute object
code, 3-38
command, 6-16
format, 2-3
paper tape format, A-1
to decimal conversion, B-1
HEXOBJ command, 3-39
how LOCATE locates segments, 4-15
HP, 2-4
HR, 2-4

I command, 3-47
I switch of DIR command, 3-27
I/O
configuration commands, 6-3
device management, 2-3
interface, 2-2
operations, terminate, 5-15
I0 parameter of ATTRIB command, 3-37
I1, 2-5
I1 parameter of ATTRIB command, 3-37
IDISK command, 3-10
In-Circuit Emulators, 1-1
initial system console, 3-2
initialization command, disk, 3-10
initialization of mask register ,7-1
initializing (formatting) disks, 2-2 ,3-10

initializing files for I/O, 5-8
inpage relocation, 4-22
input/output calls, 5-7
insert command, 3-46
inserting text, 3-47
Intellec MDS, 1-1
Intellec Monitor, 6-1
Intellec series II, 1-1
interations of command strings, 3-54
interfaces peripheral, 1-1
interrupt
  acceptance, 7-2
  mask register, 7-2
  processing, 7-1
  removal, 7-2
  switches, 3-3
intersegment references, 4-5
intrasegment references, 4-5
introduction, 1-1
invisible attribute, 3-6
IOCHK, 5-40
IOSET, 5-42
ISIS-II command format, 3-10
ISIS-II commands
  ATTRIB, 3-37
  BINOBJ, 3-38
  COPY, 3-28
  DEBUG, 3-23
  DELETE, 3-34
  DIR, 3-26
  EDIT, 3-40
  Editor
    A, 3-53
    B, 3-44
    C, 3-45
    D, 3-49
    E, 3-51
    F, 3-46
    I, 3-47
    K, 3-49
    L, 3-45
    M, 3-53
    Q, 3-52
    S, 3-48
    T, 3-50
    W, 3-52
    Z, 3-44
  FORMAT, 3-12
  Functions, 1-2
  HEXOBJ, 3-39
  IDISK, 3-10
  LIB, 4-17
    ADD, 4-18
    CREATE, 4-18
    DELETE, 4-18
    EXIT, 4-19
    LIST, 4-19
  LINK, 4-7
  LOCATE, 4-10
  Monitor
    A, 6-4
    D, 6-6
    E, 6-12
    F, 6-7
    G, 6-14
    H, 6-16
    M, 6-7
    N, 6-13
    Q, 6-5
    R, 6-11
    S, 6-8
    W, 6-12
    X, 6-9, 6-10
  OBJHEX, 3-39
  RENAME, 3-36
  SUBMIT, 3-24
ISIS-II resident area, 5-1
ISIS-II system calls
  ATTRIB, 5-20
  CI, 5-28
  CLOSE, 5-15
  CO, 5-29
  CONSOL, 5-22
  CSTS, 5-37
  DELETE, 5-18
  ERROR, 5-24
  EXIT, 5-26
  IOCHK, 5-40
  IOSET, 5-42
  LO, 5-32
  LOAD, 5-25
  MEMCK, 5-42
  OPEN, 5-8
  PO, 5-31
  READ, 5-9
  RENAME, 5-19
  RESCAN, 5-14
  RI, 5-30
  SEEK, 5-12
  SPATH, 5-16
  UI, 5-33
  UO, 5-35
  UPPS, 5-38
  WHOCON, 5-23
  WRITE, 5-10

K command, 3-49
kill line command, 3-49

L command, 3-45
L1, 2-5
language translations, 1-2
length, 3-6
LIB command, 4-17
LIB commands
  ADD, 4-18
  CREATE, 4-17
  DELETE, 4-18
  EXIT, 4-19
  LIST, 4-19
librarian, 4-17
libraries, use of, 4-6
library files, 4-6
line
  command, 3-45
  edit buffer, reading, 5-3
  edited input files, 5-3
  editing, 3-1
  editing characters, 3-1
  printer, 1-1

LINES, 4-13
LINK command, 4-7
  error messages, C-4
    fatal, C-4
    non-fatal, C-5
link map, 4-8
linkage and relocation, mechanics of, 4-4
linked loading and program overlays, 4-20
LIST, 4-19
LIST (FIXMAP), 3-18
listing directory, command, 3-26
LO, 5-32
LOAD, 5-25
LOCATE command, 4-10
LOCATE control descriptions
  / /, 4-14
  /common name/, 4-14
  CODE, 4-14
  COLUMNS (number), 4-12
  DATA, 4-14
  LINES, 4-13
  MAP, 4-11
  MEMORY, 4-14
  NAME, 4-14
  ORDER, 4-13
  PRINT, 4-12
  PUBLICS, 4-13
  PURGE, 4-13
  RESTART0, 4-14
  STACK, 4-14
  STACKSIZE, 4-14
  START (address), 4-14
  SYMBOLS, 4-12
LOCATE errors, C-5
  fatal, C-5
  non-fatal, C-7
LOCATE map, 4-11
LOCATE with default order, 4-15
Locating with the default, ORDER control, and specific addresses, 4-15
LP, 2-4

M command
  Editor, 3-53
  Monitor, 6-6
maintenance, directory calls, 5-18
management and creation of files, 3-1
management, I/O device, 2-3
management, module, 2-3
MAP parameter of link command, 4-8
MAP, LINK, 4-8
MAP, LOCATE, 4-11
MARK, 3-16
MARKER, 5-6
mechanics of relation and linkage, 4-4
MEMCK, 5-42
memory
  allocation, 4-1
  command, 3-53
  control commands, 6-6
  from file, 5-9
  organization, 5-1
  pages and H and L registers, 4-21
  requirements, 1-1
  segment, 4-3
  space, determing, 3-53
  to file, 5-10
  usage, 2-2
MEMORY control, 4-14
merging files, 3-28
microprocessor memory allocation, 4-1
minimum memory, 2-2
modular program development, 4-3
modular programming, 1-3
module management, 2-3
module name, 4-18
Monitor
  calls
    CI, 5-28
    CO, 5-29
    CSTS, 5-37
    IOCHK, 5-40
    IOSET, 5-42
    LO, 5-32
    MEMCHK, 5-42
    PO, 5-31
    RI, 5-30
    UI, 5-23
    UO, 5-35
    UPPS, 5-38
  command categories, 6-3
  command entry, 6-1
  commands
    A, 6-4
    D, 6-6
    E, 6-12
    F, 6-7
    G, 6-14
    H, 6-16
    M, 6-7
    N, 6-13
    Q, 6-5
    R, 6-11
    S, 6-8
    W, 6-12
    X, 6-9, 6-10
  entry errors, 6-2
  I/O configuration commands, 6-3
  I/O Interface routines, 5-27
  memory control commands, 6-5
  program execution under, 3-23
move command, 6-7
moving data, 5-9

N command, 6-13
N switch of COPY command, 3-30
name format, 2-4
NAME parameter of LINK command, 4-8
names system devices
  :BB:, 2-5
  :CI:, 2-4
  :CO:, 2-4
  :Fn:, 2-4
  :HP:, 2-4
  :HR:, 2-4
  :I1:, 2-5
  :L1:, 2-5
  :LP:, 2-4
  :O1:, 2-5
  :P1:, 2-5
  :P2:, 2-5
  :R1:, 2-5

:R2:, 2-5
:T1:, 2-4
:TO:, 2-4
:TP:, 2-4
:TR:, 2-4
:VI:, 2-4
:VO:, 2-4
nesting SUBMIT files, 3-24
non-interactive program execution, 3-24
nonfatal errors, 3-3
nonstandard devices, 2-5
nonsystem disk formatting, 3-10
null command, 6-13

O1, 2-5
object file formats, 2-3
object module, 4-4
OBJHEX command, 3-39
obtaining file information 5-16
OPEN, 5-8
operating system, 1-1
ORDER, 4-13
order of modules in an output file, 4-9
organization of disks, 3-5
organization of memory, 5-1
overlay, 4-19
overlay management, 4-19

QUIT, 3-20

P switch of COPY command, 3-30
P switch of DIR command, 3-27
P1, 2-5
P2, 2-5
page boundary, 4-24
page relocation, 4-22
paper tape I/O commands, 6-11
paper tape punch, 1-1
paper tape reader, 1-1
parameter passing to SUBMIT, 3-24
parameters, text Editor command, 3-43
passing parameters to SUBMIT, 3-24
pause switch
  COPY command, 3-30
  DELETE command, 3-34
  DIR command, 3-26
peripheral devices, 1-1
peripheral interfaces, 1-1
PL/M, 1-2
PL/M calls, 5-6
PLM80, 4-1
PO, 5-31
pointer, text, 3-43
position disk file marker, 5-12
position file marker at beginning, 5-14
power up, 2-7
PRINT, 4-12
PRINT parameter of LINK command, 4-8
printer, 1-1
processing errors and debugging, 3-3
processing interrupts, 7-1
program
  development, 4-4
  execution calls, 5-25
  execution commands, 3-23
  execution under Monitor, 3-23, 6-13
  execution, non-interactive, 3-24
  loading, 5-25
  overlays and linked loading, 4-20
  segments, 4-2
public symbols and external references, 4-5
PUBLICS, 4-13
punch, 1-1
punch output, 5-31
PURGE, 4-13

Q command
  Editor, 3-52
  Monitor, 6-5
Q switch
  of ATTRIB command, 3-37
  of COPY command, 3-30
  of DELETE command, 3-34
query command, 6-5
query switch
  of ATTRIB command, 3-37
  of COPY command, 3-30
  of DELETE command, 3-34
question mark (?), 3-7, 3-30
quit command, 3-52

R command, 6-11
R1, 2-5
R2, 2-5
READ, 5-9
read command, 6-11
reader, 1-1
reader input, 5-30
reading a command line, 5-4
reading data from disk, 3-53
reading from line edit buffer, 5-3
RECORD, 3-19
recording characteristics, disk, 3-4
references, external, 4-5
references, intersegment, 4-5
references, intrasegment, 4-5
register command, 6-9, 6-10
relative addressing, 4-4
relocatable object module, 4-2
relocation, 4-4
relocation and linkage, mechanics of, 4-4
relocation types, 4-22
relocation with the assembler, 4-22
removing interrupts, 7-2
RENAME command, 3-36
RENAME system call, 5-19
renaming a disk file, 3-36
RESCAN, 5-14
reset button, 2-7
RESTART0, 4-14
RI, 5-30
rubout, 3-1, 5-4

S command
  Editor, 3-48
  Monitor, 6-8
S switch
  of COPY command, 3-30
  of FORMAT command, 3-13
  of IDISK command, 3-11
S0 parameter of ATTRIB command, 3-37
S1 parameter of ATTRIB command, 3-37

saving your work, 3-51
sectors, 3-5
SEEK, 5-12
segments, 4-2
shared subprograms, 4-4
single density disks, 2-1
source program, 4-1
single disk drive copying, 3-28
SPATH, 5-16
STACK, 4-14
stack segment, 4-3
STACKSIZE, 4-15
standard devices
  :BB:, 2-5
  :CI:, 2-4
  :CO:, 2-4
  :Fn:, 2-4
  :HP:, 2-4
  :HR:, 2-4
  :I1:, 2-5
  :L1:, 2-5
  :LP:, 2-4
  :O1:, 2-5
  :P1:, 2-5
  :P2:, 2-5
  :R1:, 2-5
  :R2:, 2-5
  :T1:, 2-4
  :TO:, 2-4
  :TP:, 2-4
  :TR:, 2-4
  :V1:, 2-4
  :VO:, 2-4
START, 4-15
start-up procedure, 2-7
starting the Editor, 3-40
starting the system, 2-7
STKLN, 4-2
storage, 2-2
string, command iterations, 3-54
SUBMIT command, 3-24
SUBMIT files, preparing, 3-24
substitute command
  Editor, 3-48
  Monitor, 6-8
summary of system calls, 5-5
switches, interrupt, 3-2
SYMBOLS, 4-12
syntax of system calls, 5-6
system
  attribute, 3-6
  call syntax and usage, 5-6
  calls
    ATTRIB, 5-20
    CI, 5-28
    CLOSE, 5-15
    CO, 5-29
    CONSOL, 5-22
    CSTS, 5-37
    DELETE, 5-18
    ERROR, 5-24
    EXIT, 5-26
    IOCHK, 5-40
    IODEF, 5-39
    IOSET, 5-42
    LO, 5-32
    LOAD, 5-25
    MEMCHK, 5-42
    OPEN, 5-8
    PO, 5-31
    READ, 5-9
    RENAME, 5-19
    RESCAN, 5-14
    RI, 5-30
    SEEK, 5-12
    SPATH, 5-16
    UI, 5-33
    UO, 5-35
    UPPS, 5-37
    WHOCON, 5-23
    WRITE, 5-10
  calls, cautions, 5-7
  calls, summary, 5-5
  composition, 2-1
  console, 3-2
  designated device names, 2-5
  disk formatting, 3-10, 3-12
  start-up, 2-7
  operating, 1-1
  status routines, 5-37
SYSTEM.LIB, 5-6

T command, 3-50
TAB character, 3-42
temporary files, 3-5
terminate I/O operations, 5-15
terminate program, 5-26
terminating a line, 5-3
terminating a session, 3-51
text commands, 3-47
Text Editor
  command entry, 3-41
  command parameters, 3-41
  command string iteration, 3-54
  commands
    A, 3-53
    B, 3-44
    C, 3-45
    D, 3-49
    E, 3-51
    F, 3-46
    I, 3-47
    K, 3-49
    L, 3-45
    M, 3-53
    Q, 3-52
    S, 3-48
    T, 3-50
    W, 3-52
    Z, 3-44
  errors, C-9
  formatting characters, 3-42
  functions, 3-40
  activating, 3-42
  saving your work, 3-51
  terminating a session, 3-51
  using, 3-40
text pointer, 3-43
T1, 2-4
TO, 2-4

TO parameter
  of COPY command, 3-30
  of DIR command, 3-26
  of RENAME command, 3-36
top of memory, 5-42
TP, 2-4
TR, 2-4
tracks, 3-5
transferring data, 5-7
TTY, 1-1
TYPE program, D-1
type command, 3-50
types of disk files, 1-3
typing a file, 3-50

U switch of COPY command, 3-30
UI, 5-33
universal PROM programmer, 1-1
unsatisfied external reference, 4-5
unsatisfied module, 4-5
UO, 5-35
UPP
  input, 5-33
  output, 5-35
  status, 5-38
UPPS, 5-38
use of different source languages, 4-4
use of ISIS-II by other programs, 5-1
use of libraries, 4-6
user aids, 3-1
using ISIS-II commands, 3-8
using the Text Editor, 3-40
utility command, 6-16

VI, 2-4
VO, 2-4

W command,
  Editor, 3-50
  Monitor, 6-12
W0 parameter of ATTRIB command, 3-37
W1 parameter of ATTRIB command, 3-37
WHOCON, 5-23
wild card file names, 3-7, 3-30
working with program modules, 4-1
WRITE, 5-10
write command
  Editor, 3-52
  Monitor, 6-12
write-protect attribute, 3-6

X command, 6-9, 6-10

Z command, 3-44

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 CORPORATION, 3065 Bowers Avenue, Santa Clara, California 95051 (408) 987-8080

Printed in U.S.A.