CAPS-11 USER'S GUIDE

DEC-11-OTUGA-A-D

digital

CAPS-11 USER'S GUIDE

DEC-11-OTUGA-A-D

digital equipment corporation · maynard, massachusetts

First Printing, October, 1973

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this manual.

The software described in this document is furnished to the purchaser under a license for use on a single computer system and can be copied (with inclusion of DIGITAL's copyright notice) only for use in such system, except as may otherwise be provided in writing by DIGITAL.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.

The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation. All comments received will be considered when subsequent documents are prepared.

The following are trademarks of Digital Equipment Corporation:

CDP	DIGITAL	INDAC	PS/8
COMPUTER LAB	DNC	KA10	QUICKPOINT
COMSYST	EDGRIN	LAB-8	RAD-8
COMTEX	EDUSYSTEM	LAB-8/e	RSTS
DDT	FLIP CHIP	LAB-K	RSX
DEC	FOCAL	OMNIBUS	RTM
DECCOM	GLC-8	OS/8	RT-11
DECTAPE	IDAC	PDP	SABR
DIBOL	IDACS	PHA	TYPESET 8
			UNIBUS

CONTENTS

TABLES

FIGURES

PREFACE

This manual describes the PDP-11 Cassette Programming System and provides all the information necessary for normal usage. It requires no prior experience on a PDP-11 computer, but does assume some exposure to assembly language programming and computer systems in general. Upon receiving his system, the user should first read through the entire CAPS-11 manual, then reconfigure his system (if necessary) according to the instructions provided in Appendix F; lastly, he should try the demonstration program run in Appendix D.

Frequent reference is made to two supplementary handbooks which the user should also receive with his system. These are: THE PDP-11 PERIPHERALS AND INTERFACING HANDBOOK and THE PDP-11 PROCESSOR HANDBOOK. The latter handbook may be any one of several Processor Handbooks, each geared to a particular PDP-11 Processor (11/20/15/R20, 11/45, etc.); the handbook received with a CAPS-11 System depends upon the processor purchased.

If the user intends to write his own cassette handler or if he will use Monitor cassette primitives, he should familiarize himself with the TA11 CASSETTE INTERFACE SYSTEM manual (DEC-11-HTACA-A-D), the TU60 CASSETTE TAPE TRANSPORT MAINTENANCE MANUAL (DEC-00-TU60-DA), and the Cassette Standard (Appendix F of this manual).

Several different configurations are possible with the Cassette Programming System. For documentation purposes, the following configuration is assumed: PDP-11/10 processor, LA30 DECwriter, LS11 line printer.

Documentation conventions include the following:

1. Actual computer output is used in examples wherever possible. When necessary, computer printout is underlined to differentiate it from user responses.

2. To avoid confusion, a line feed is represented in the text as ↓ ; a carriage return is represented by) . Unless otherwise indicated, all commands and command strings are terminated by a carriage return.

3. Terminal, console terminal, and teleprinter are general terms used throughout the documentation to represent any one of the following: LA30 DECwriter, VT05 Display, LT33 or 35 Teletype.

4. Several characters used in system commands are produced by a combination of two keys typed at the same time. Generally, the combinations are SHIFT and some other key (such as SHIFT and N to produce the uparrow character on an LT33 or 35) or CTRL and another key (for example, CTRL and O produces a command which causes suppression of teleprinter output). These key combinations are documented as SHIFT/N, CTRL/O, etc., respectively.

5. Portions of command strings which are enclosed in square brackets are optional--the user may type them or not as he chooses without changing the intention of the command.

6. Certain keyboard variances prevail among teleprinters which may be used as the console terminal in a CAPS-11 System; these concern labeling of keyboard keys and characters output upon receipt of particular ASCII character codes. Refer to Appendix A for a list of possible differences.

CHAPTER 1

THE CAPS-11 PROGRAMMING SYSTEM

The PDP-11 Cassette Programming System (CAPS-11) is a small programming system for the PDP-11 computer designed around the use of cassettes for program and data storage. CAPS-11 provides the user with the capability of performing all file transfers, program development, loading, and storage via cassette. The system also provides minimal support for using paper tape by allowing the user who has paper tape programs to transfer these programs to cassette and vice versa.

CAPS-11 provides the user with a Keyboard Monitor, I/O facilities at the Monitor level, and a library of system programs, including a machine language assembler, an editor, and a debugging program.

Figure 1-1 CAPS-11 Programming System

1.1 SYSTEM CONFIGURATION

A CAPS-11 minimal system configuration consists of the following hardware and software components. Optional memory and peripheral devices may be added as desired.

1.1.1 Hardware Components

The PDP-11 Cassette Programming System is built around any PDP-11 processor with one (only) TA11 controller, a console terminal (LA30 DECwriter, LT33 or LT35 Teletype, or VT05 DECterminal), and 8K (or as much as 28K) of memory. A line printer (LP11 or LS11) is optional. A high-speed paper tape reader and punch are also optional and may be used by PIP.

Section 1.5 describes operational procedures for the PDP-11/10 processor, LA30 DECwriter, and LS11 line printer, as these devices are considered representative of a standard CAPS-11 System configuration.

1.1.2 Software Components

CAPS-11 software is provided on three cassettes--two OBJ Cassettes and a System Cassette. The OBJ Cassettes are used exclusively for changing and building system configurations and are explained in Appendix E. A brief description of the software package stored on the System Cassette follows. Each program is discussed in greater detail later in the manual.

1. Monitor - The Keyboard Monitor provides communication between the user and the Cassette System executive routines by accepting commands from the console terminal keyboard. The commands allow the user to run system and user programs, load and start programs using maximum memory space, and obtain directories of cassettes.

2. Symbolic Editor - The Editor allows the user to modify or create source files for use as input to the Assembler. The Editor contains powerful text manipulation commands for quick and easy editing.

3. PAL Assembler - The Assembler (Program Assembly Language) accepts source files in the PAL machine language and generates binary object modules (and/or assembly listings) as output. These object modules can then be linked, loaded and executed.

4. Linker - The Linker converts relocatable object modules produced by the Assembler into absolute load modules for program loading and execution. The Linker also produces a load map which displays the assigned absolute addresses.

5. ODT - The ODT (On-Line Debugging Technique) program aids the programmer in debugging his object program by allowing him to examine, change, and run any portion of his program on-line using simple commands typed on the console terminal.

6. RESMON - The Input/Output package (RESMON) provides routines for all input/output programming in the CAPS-11 System. User programs can communicate with RESMON (via IOT instructions which utilize RESMON) to create cassette files and perform all console terminal and line printer I/O.

7. PIP - PIP (Peripheral Interchange Program) allows the user to transfer files from one cassette to another or to the console terminal or line printer and to delete files from cassette. PIP also provides minimal support for paper tape usage by allowing programs to be transferred from cassette to the high-speed paper tape punch and from the high-speed paper tape reader to cassette.

1.2 WHAT IS A CAPS-11 CASSETTE?

A CAPS-11 cassette is a magnetic tape device much like that used in a cassette tape recorder. The tape itself and the reels it is wound on are enclosed inside a rectangular plastic case (see Figure 1-2), making handling, storage, and care of the cassette convenient for the user.

On either end of one side of the cassette are two flexible plastic tabs called write-protect tabs (see A in Figure 1-2). There is one tab for each end of the tape; since data should only be written in one direction, the user will need to be concerned with only the tab specifically marked on the cassette label. Depending upon the position of this tab the user is able to protect his tape against accidental writing and destruction of data. When the tab is pulled in toward the middle of the cassette so that the hole is uncovered, the tape is write-locked; data cannot be written on it and any attempt to do so will result in an error message. When the tab is pushed toward the outside of the cassette so that the hole is covered, the tape is write-enabled and data can be written onto it. Data can be read from the cassette with the tab in either position.

The bottom of the cassette (B in Figure 1-2) provides an opening where the magnetic tape is exposed. The cassette is locked into position on a cassette unit drive so that the tape comes in contact with the read/write head through this opening.

Both ends of the magnetic tape in a cassette consist of clear plastic leader/trailer tape; this section of the tape is not used for information storage purposes, but as a safeguard in handling and storing the cassette itself. Since magnetic tape is susceptable to dust and fingerprints, a cassette should always be rewound so that the leader/trailer tape is the only part of the tape exposed whenever the cassette is not on a drive.

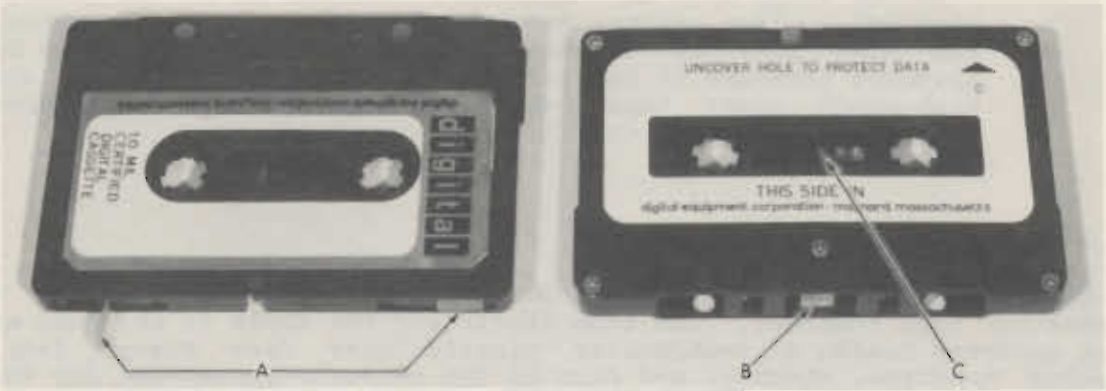


Figure 1-2 CAPS-11 Cassette

1.2.1 The Format of a Cassette

A cassette is formatted so that it consists of a sequence of one or more files. Files on cassette are sequential, and each file is preceded and followed by a file gap. (A gap in this sense is a fixed length of blank tape.) All cassettes must start with a file gap; information preceding the initial file gap is unreliable.

A file consists of a sequence of one or more data records separated from one another by a record gap. The records of any given file must follow one another in succession, as there is no provision for record linking. The first record of a file is called the header record and contains information concerning the name of the file, its type, length, and so on. (The Cassette Standard may be referenced in Appendix F.) There are approximately 600 records per cassette tape. CAPS-11 recognizes an end-of-file by the presence of either a file gap or clear leader following a data record.

Data records in the CAPS-11 System consist of 128 (decimal) cassette bytes; a byte in turn consists of eight bits each representing a binary zero or one. Characters and numbers are stored in bytes using the standard ASCII character codes (see Appendix A) and binary notation.

The number of records of information on a cassette tape may be estimated by the user. On the outside of the cassette case is a clear plastic window (C in Figure 1-2). Along the bottom of this window is a series of marks; each mark represents about 50 inches of magnetic tape. Knowing that approximately 2 records fit on an inch of tape, the user is able to make a reasonable guess as to the length of tape and number of records available for use. By simply glancing at the width of the tape reel showing in the window, the user can tell

quickly if he is very close to the end. Since he is given no advance warning of a full tape condition, the user must visually keep track of the length of tape he has available. Should the tape become full before his file transfer has completed, another cassette may be substituted and the transfer or output operation repeated, or the /O overflow option may be used to allow continuous transfer (see Section 3.2.4 in Chapter 3).

1.2.2 The Sentinel File

The last file on a cassette tape is called the sentinel file. This file consists of only a 32 (decimal) byte header record and represents the logical end-of-tape (CAPS-11 also recognizes clear trailer as logical end-of-tape). A sentinel file is identified by a null character (ASCII=000) as the first name character in the header record. A zeroed or blank cassette tape is one consisting of only the sentinel file.

1.3 THE SYSTEM CASSETTE

The software discussed in Section 1.1.2 is provided to the user on a single cassette called the System Cassette. This is the cassette on which the entire CAPS-11 System resides and which is utilized for all normal system functions. When in use, the System Cassette should always be mounted on drive 0 (the drive on the left of the TA11 controller); drive 0 serves as the default device when the user fails to specify another.

The write-protect tab on the System Cassette should usually be in the write-locked position so that data will not accidently be written on it; it is suggested that the user make several copies of this cassette as protection against loss or accidental destruction.

1.4 MOUNTING AND DISMOUNTING A CASSETTE

To mount a tape on a drive, hold the tape so that the open part of the cassette is to the left and the full reel is at the top. Set the top write-protect tab to the desired position depending upon whether data is to be written on the tape.

Open the locking bar on the cassette drive by pushing it to the right, away from the drive (see A in Figure 1-3). Next hold the tape up to the cassette drive at approximately a 45-degree angle and insert the tape into the drive by applying a leftward pressure while simultaneously pushing the cassette onto the drive sprockets. This brings the tape into position against the read/write head. When the cassette is properly mounted, the locking bar will automatically close over the cassette back edge. Figure 1-3 illustrates this procedure.

Press the rewind button on the cassette unit (see B in Figure 1-3; there is a rewind button for each drive). This causes the cassette to rewind to the beginning of its leader/trailer tape. (Pressing the rewind button a second time causes the cassette to rewind to the end

of the leader/trailer tape and to the physical end-of-tape. The cassette unit will click; this sound is almost inaudible and the user may not hear it unless he is listening carefully. Normal usage requires that the user press the rewind button only once whenever he wishes to rewind a cassette). Even though tapes which are not actively being used on a drive should already be positioned at the beginning, the user should develop the habit of automatically rewinding a cassette.

NOTE

Rewinding a cassette is particularly important since certain functions (such as space reverse file and space reverse block), initiated on a newly mounted cassette prior to the use of any other function, could cause the cassette controller to function improperly. This condition is remedied whenever the START key is depressed, or when a hardware RESET instruction is executed.

When the tape has finished winding, the cassette will stop moving. The cassette is now in place and ready for transfer operations.

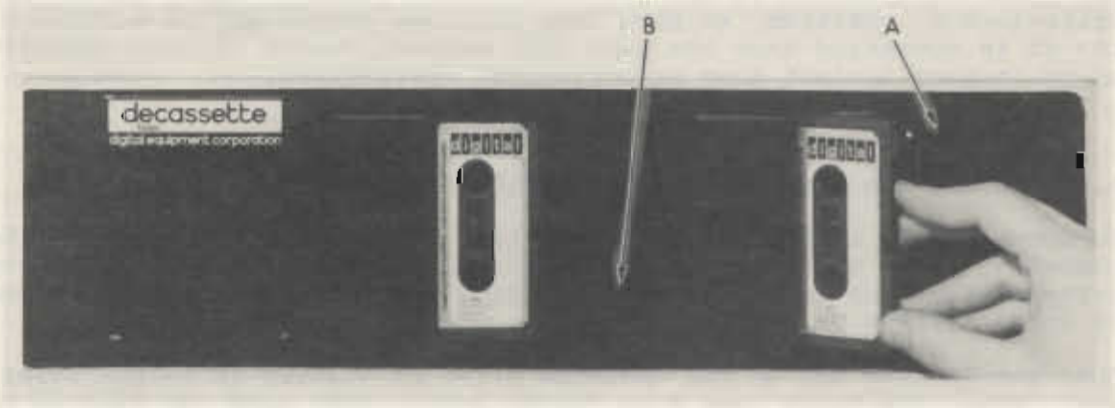


Figure 1-3 Mounting a Cassette

Before removing a cassette from a drive, the tape should always be rewound to its beginning by pressing the rewind button on the cassette unit. Rewinding a tape ensures that the clear leader/trailer tape will be the only tape exposed at the open part of the cassette. To remove a cassette from the cassette drive, open the locking bar and the cassette will pop out. When cassettes are not being actively used on a cassette drive, they can be stored in the small plastic boxes provided for this purpose by the manufacturer.

NOTE

Before using a new cassette, or prior to using a cassette that has just been shipped or accidently dropped, mount the cassette on a drive so that the Digital label faces the inside of the unit and perform a rewind operation. Remove the cassette, turn it over, and perform another rewind operation. This packs the tape neatly in the cassette and places the full tape reel at the proper tension.

1.5 CONSOLE OPERATION

The operation of the computer console and console terminal, using the PDP-11/10 processor, LA30 DECwriter, and LS11 line printer as examples, follows.

1.5.1 PDP-11/10 Programmer's Console

The PDP-11/10 console is designed to provide convenient manual control of the system. Using switches and keys located on the console, programs and information can be directly inserted into memory and modified. The PDP-11/10 console is shown in Figure 1-4, and each switch and key is explained in the paragraphs following the figure.

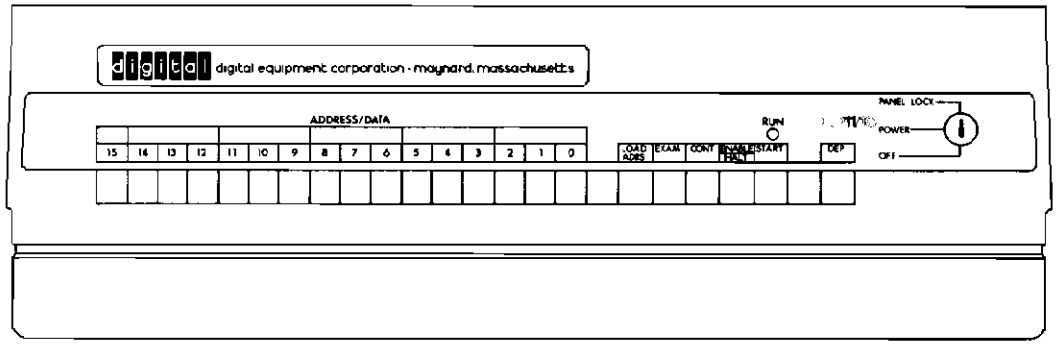


Figure 1-4 The PDP-11/10 Console

Elements of the Console

The console has the following indicators and switches:

1. A RUN lamp which, if lit, indicates that the processor is running

2. A 16-bit Address and Data Register display

3. A 16-bit Switch Register

4. The following control switches:

   a) LOAD ADRS

   b) EXAM

   c) CONT

   d) ENABLE/HALT

   e) START

   f) DEP

The programmer's console has one 16-bit register display used for displaying both addresses and data. When displaying the contents of the Address Register this display register is tied directly to the output of a 16-bit flip-flop register called the Bus Address Register and displays the address of any data examined or deposited. It may also be used to display the contents of the Data Register by displaying data in any memory location or the results of program execution.

The programmer may reference 16-bit addresses by manipulating the Switch Register. A switch in the up position is considered to have a 1 value; a switch in the down position is considered to have a 0 value. Thus, the address indicated by the switch setting can then be loaded into the Address Register or data can be loaded into any memory location by using the appropriate control switches as follows (when the system is executing a program, the LOAD ADRS, EXAM, and DEPosit functions are disabled to prevent any disruption of the running program):

Table 1-1
PDP-11/10 Control Switches

Switch	Action
LOAD ADRS	Transfer the contents of the 16-bit Switch Register into the Address Register.
EXAM	Display in the 16-bit register display the contents of the location stored in the Address Register.
DEP	Deposit the contents of the 16-bit Switch Register into the address stored in the Address Register. (This switch is actuated by raising it.)
ENABLE/HALT	Allow or prevent program execution. To allow a program to run, the switch must be in the ENABLE position (up). Placing the switch in the HALT position (down) will halt the system at the end of the current instruction.
START	Begin execution of a program (the ENABLE/HALT switch must be in the ENABLE position). When the START switch is depressed, it asserts a system initialization signal and actually starts the system when it is released. The processor will begin execution at the address which was last loaded using the LOAD ADRS switch.
CONT	Allow the computer to continue without initialization from whatever state it is in after halting.

Operating the Control Switches

After the processor has halted at the end of an instruction, it is possible to examine and update the contents of locations. To examine a specific location, set the Switch Register to correspond to the location's address, and press LOAD ADRS; this transfers the contents of the Switch Register into the Address Register. The location of the address to be examined is displayed in the 16-bit register display. The user can then depress EXAM, and the data in that location will appear in the register display.

NOTE

If the user attempts to examine data from or deposit data into a nonexistent memory location, an error will occur and the register display will reflect the contents of location 000004 (the trap location for references to nonexistent locations). To verify that this trap has occurred, deposit some number other than four in the location. If four is still indicated, either nothing is assigned to that location or whatever is assigned is not working properly.

By depressing EXAM again, the Address Register will be incremented by two to the next word address, and the contents of this next location may be examined.

The examine function operates such that if LOAD ADRS is depressed and then EXAM, the Address Register will not be incremented. However, successive use of the EXAM switch increments the Address Register by two for each depression.

If the user finds an incorrect entry in the Data Register, he can change it by setting the correct data in the Switch Register and raising the DEP switch. The Address Register will not increment when this data is deposited. Therefore, by pressing the EXAM switch the user can examine (verify) the data just deposited. Pressing EXAM a second time will increment the register to the next word address.

When performing consecutive examines or deposits as previously mentioned, the address will increment by two to successive word locations. However, when examining the general-purpose registers (R0-R7), the system will only increment by one.

To start a program after it is loaded into memory, set the starting address of the program in the Switch Register and press LOAD ADRS. Be sure that the ENABLE/HALT switch is in the ENABLE position; depress START. The program should begin executing as soon as the START switch is released.

While in the halt mode, the user may execute a single instruction by pressing CONT. When CONT is pressed, the console momentarily passes control to the processor, allowing it to execute one instruction before regaining control. Each time the CONT switch is pressed the computer will execute one instruction.

To start the program again, place the ENABLE/HALT switch in the ENABLE position and press CONT.

1.5.2 Operating the Console Terminal (LA30 DECwriter)

The LA30 DECwriter consists of a printer and keyboard, and is illustrated in Figure 1-5.

Figure 1-5 LA30 DECwriter (Serial)

The printer provides a typed copy of input and output at 30 characters per second, maximum. Keyboard functions such as TAB and RETURN and all characters, including \, [and], have a distinct key associated with them (unlike the LT33 and 35 keyboards which must use key combinations to produce these and other characters and functions). The keyboard is illustrated in Figure 1-6.

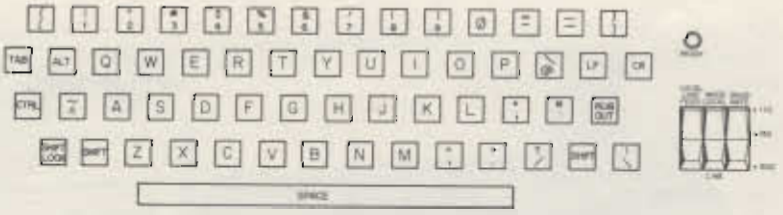


Figure 1-6 LA30 DECwriter (Serial) Keyboard

On the back of the LA30 console stand is a switch which is used to turn the terminal on and off. When the switch is raised, the READY indicator lamp on the keyboard panel lights to designate that the terminal is ready for use. The DECwriter is shut off by pushing the switch down.

Below the READY lamp is a key labeled LOCAL LINE FEED; while this key is pressed, paper is advanced from the printer. The MODE key next to it should be set to LINE for all on-line operations; the baud rate is generally fixed at 300 and the BAUD RATE key should be set to this figure. Random characters will be generated if this key is not set to match the baud rate. The remaining keys on the keyboard are used for producing typed copy and are similar to those found on a typewriter keyboard.

A parallel LA30 DECwriter varies in appearence slightly from a serial LA30 and is pictured in Figure 1-7; the user does not have to set a BAUD RATE or LINE key; all other operations are the same.

Figure 1-7 LA30 DECwriter (Parallel)

1.5.3 Operating the LS11 Line Printer

The LS11 line printer may be used to output listings at a rate of 165 characters per second with as many as 132 characters per line. The unit is very compact and can sit on a small table.

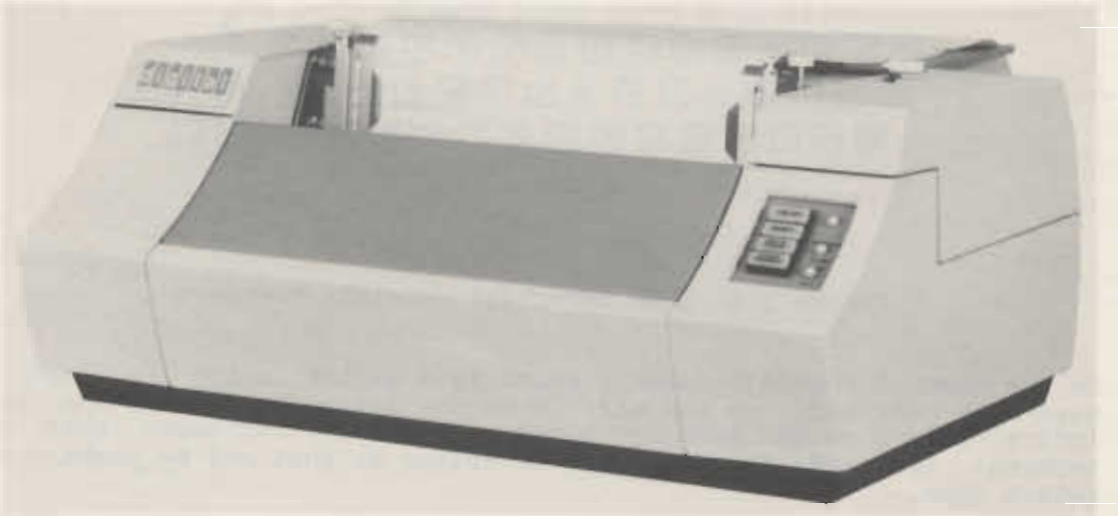


Figure 1-8 LS11 Line Printer

The operator panel is illustrated in Figure 1-9 and provides the user with the following functions:

Table 1-2
LS11 Operator Panel Functions

Key	Action
ON/OFF	Pushing the key once turns the printer on and lights the switch; pushing the key a second time shuts the printer off.
SELECT	Pushing the SELECT key enables the printer for use.
TOP OF FORM	Pushing this key causes the paper to advance vertically allowing manual form control.
FORMS OVERRIDE	Pushing this key allows the user to complete the form being printed if the paper needs to be replenished (i.e., it overrides a paper-out condition).
SINGLE LINE ADVANCE	Pushing this key allows the user to vertically advance the paper by one line.
Indicator	**Meaning**
HARDWARE ALARM	Lights to indicate a hardware error.
PAPER OUT	Lights to indicate an out-of-paper or paper-handling malfunction.

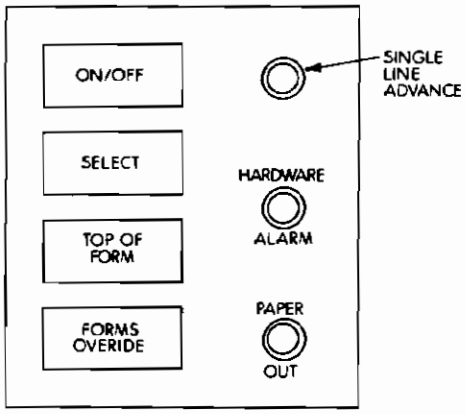


Figure 1-9 LS11 Operator Panel

CHAPTER 2

PROGRAMMING THE PDP-11

The PDP-11 processor is a 16-bit, general-purpose, parallel-logic computer using two's complement arithmetic. Programmers can directly address 32,768 16-bit words, or 65,536 8-bit bytes. All communication between system components is done on a single high-speed bus called the UNIBUS. Standard features of the system include eight general-purpose registers which can be used as accumulators, index registers, or address pointers; and a multi-level automatic priority interrupt system. A simplified block diagram of the PDP-11 System is presented in Figure 2-1.

This chapter gives the PDP-11 programmer an overview of system architecture, points out unique hardware features, and presents programming concepts basic to its use. Reference should also be made to the appropriate PDP-11 PROCESSOR HANDBOOK and the PDP-11 PERIPHERALS AND INTERFACING HANDBOOK.

2.1 GENERAL SYSTEM STRUCTURE

The architecture of a PDP-11 system and the design of its central processor provide:

Single and double operand addressing

Full word and byte addressing

Simplified list and stack processing through auto-address stepping (autoincrementing and autodecrementing)

Eight programmable general-purpose registers

Data manipulation directly within external device registers

Addressing of device registers using normal memory reference instructions

Asynchronous operation of memory, processor and I/O devices

A hardware interrupt priority structure (multi-line, multi-level) for peripheral devices

Automatic interrupt identification without device polling

Cycle stealing direct memory access for high-speed data transfer devices

Direct addressing of 32K words (64K bytes), including the 4K external page

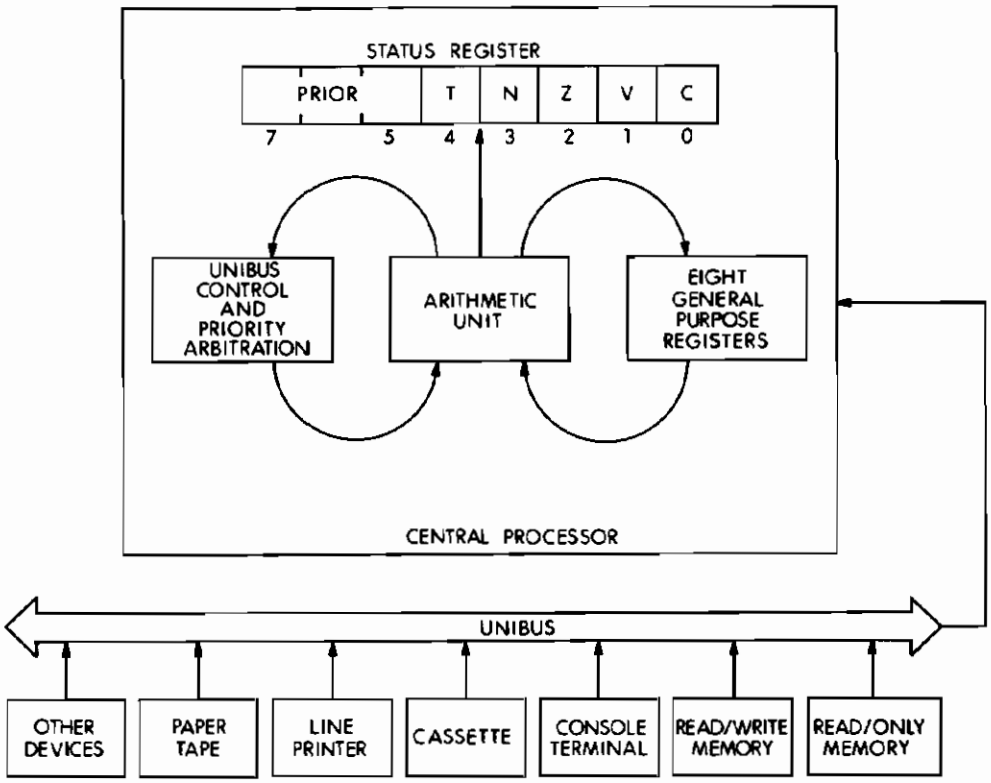


Figure 2-1 System Diagram

Two design features of the central processor serve to increase system throughput:

1. The eight programmable general-purpose registers within the central processor can be used to store data and intermediate results during the execution of a sequence of instructions. Register-to-register addressing provides reduced execution time for most instructions.

2. The ability to code two addresses within a single instruction allows operations on data within memory. This eliminates the need to load processor registers prior to data operations, and greatly reduces fetch and store operations.

2.1.1 Status Register Format

The Central Processor Status Register (PS) contains information on the current priority of the processor, the result of previous operations, and an indicator for detecting the execution of an instruction to be trapped during debugging. The priority of the central processor can be set under program control to any one of eight levels. This information is held in bits 5, 6, and 7 of the PS. Four bits are assigned to monitor the results of a previous instruction. These bits are set as follows:

Bit	Set
Z	-- if the result was zero
N	-- if the result was negative
C	-- if the operation resulted in a carry from the most significant bit
V	-- if the operation resulted in an arithmetic overflow

The T bit is used in program debugging and can be set or cleared under program control. If this bit is set when an instruction is fetched from memory, a processor trap will occur at the completion of the instruction's execution.

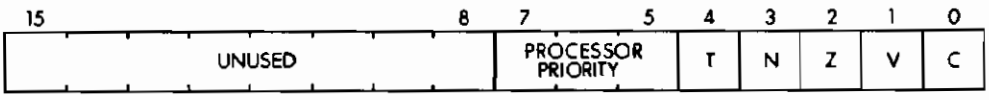


Figure 2-2 Processor Status Register

2.1.2 UNIBUS

The UNIBUS is a key component of the PDP-11's unique architecture. The central processor, memory, and all peripheral devices share the same bus. This means that device registers can be addressed as memory, and data transfers from input to output devices can by-pass the processor. No special input/output instructions exist; all PDP-11 instructions are available for I/O operations.

2.1.3 Device Interrupts

Interrupt request lines provide for device interrupts at processor priority levels 4 through 7. Attachments of a device to a specific line determines the device's hardware priority. Since multiple

devices can be attached to a specific line, the priority for each is determined by position; devices closer to the central processor have higher priority.

Peripheral device interrupts are linked to specific memory locations called "interrupt vectors" in such a way that device polling is eliminated. When an interrupt occurs, the interrupt vector supplies a new Processor Status word (i.e., new contents for the Processor Status register) and a new value for the Program Counter. The new PC value causes execution to start at the proper handler at the priority level indicated by the priority bits of the new Status Register.

2.1.4 Instruction Set

The instruction set (explained fully in the PDP-11 PROCESSOR HANDBOOK and summarized in Appendix B of this manual) provides operations that act upon 8-bit bytes and 16-bit words. Coupled with varying address modes (Relative, Index, Immediate, Register, Autoincrement, or Autodecrement, each of which can be deferred) more than 400 unique instructions are available. Instruction length is variable (from one to three 16-bit words) depending upon the addressing mode(s) used.

2.1.5 Addressing

Every byte has its own unique address. It is the instruction which determines whether 8-bit bytes or 16-bit words are being referenced. Words are addressed by their low-order (even-numbered) byte. Although byte addressing can be to odd or even numbered addresses, referencing words at odd numbered addresses is illegal. Bits are numbered from 0 at the lowest-order bit (2(0)) , to 15 (for a word) or 7 (for a byte) at the highest-order bit (2(15) or 2(7)).

Most data in programs is structured in some way, often by means of tables consisting of the data itself or of addresses which point to the data. The PDP-11 handles common data structures with operand addressing modes specifically designed for each kind of access. In addition, addressing for unstructured data permits direct random access to all of memory. The actual formats of the modes are described in Chapter 5, concerning the Assembler.

Registers

Addressing in the PDP-11 is done through the general registers. These registers can be specified by preceding a number in the range 0 to 7 by a % sign. However, it is common practice to assign register identities to symbols; often R0=%0, R1=%1, etc. (see Chapter 5, Section 5.4.4). Throughout this manual, reference to R0, R1,...R7, as well as to SP and PC, assumes such prior direct assignment. All eight general registers are accessible to the programmer, but two of these have additional specialized functions: R6 is the processor Stack Pointer (SP), and R7 is the Program Counter (PC). Both are discussed in more detail later in this chapter.

To make use of a register as an accumulator, index register, or sequential address pointer, data needs to be transferable to and from the register. This is accomplished using Register Mode, which specifies that the instruction is to operate on the contents of the indicated register itself. For example:

```
CLR R3      ;CLEAR REGISTER 3 OF ITS CONTENTS
```

Address Pointers

The instruction can be made to interpret the register contents as the address of the data to be operated on by specifying that Register Mode be deferred. For example, if register 3 contains 1000, either instruction:

```
CLR @R3

CLR (R3)
```

will clear the address 1000. Moreover, if it is desired to perform the instruction successively upon data at sequential addresses (i.e., in a table), Autoincrement Mode can be selected. This will automatically increment the contents of the register after its use as a pointer to the next sequential byte or word address. Note that Autoincrement Mode (as well as Autodecrement Mode) is automatically deferred one level to cause the register contents to function as a pointer.

When it is specified that Autoincrement Mode be deferred, it is deferred two levels so that the instruction interprets the autoincremented sequential locations as a table of addresses rather than as a table of data, as in nondeferred Autoincrement Mode. The instruction then operates upon the data at the addresses specified by the table entries.

Each execution of each of the following ADD instructions increments the value of the register contents by two to the next word address (always an even number).

```
ACCUM: ADD (R0)+,(R1)+    ;IF R0 INITIALLY CONTAINS 1000
       .                  ;AND R1 INITIALLY CONTAINS 1450,
       .                  ;THE VALUES AT LOCATIONS 1000,
       .                  ;1002, ETC., ARE ADDED TO THOSE AT
       .                  ;LOCATIONS 1450, 1452, ETC., AND
       .                  ;THE RESULT STORED AT 1450, ETC.
       JMP ACCUM

ACCUM: ADD @(R3)+,R2      ;IF R3 INITIALLY CONTAINS 1000
       .                  ;AND LOCATION 1000 CONTAINS 3420,
       .                  ;THE VALUE AT LOCATION 3420 IS
       .                  ;ADDED TO THE CONTENTS OF R2 AND
       .                  ;THE RESULT IS STORED THERE. AT
       .                  ;THE NEXT EXECUTION OF THE
       .                  ;INSTRUCTION, R3=1002.
       JMP ACCUM
```

Byte instructions such as TSTB (R2)+ (using Autoincrement Mode) increment the register contents by one.

In addition to this capability of incrementing a register's contents after their use as a pointer, an address mode complementary to this exists. Autodecrement Mode decrements the contents of the specified register before the contents are used as a pointer. This mode, too, can be deferred an additional level if the table contains addresses rather than data.

Stack Operations

Both Autoincrement and Autodecrement Modes are used in stack operations. Stacks, also called push-down or last-in-first-out lists, are important for temporarily saving values which might otherwise be altered. Their characteristic is that the most recent piece of data saved is the first to be restored. The PDP-11 processor makes use of stack structure to save and restore the state of the machine on interrupts, traps, and subroutines. To save, data is "pushed" onto a stack by autodecrementing the contents of a register (e.g., MOV R3,-(R6)); to restore, data is "popped" from a stack by autoincrementing (e.g., MOV (R6)+,R3). The register being used as the Stack Pointer always points to the top word of the stack.

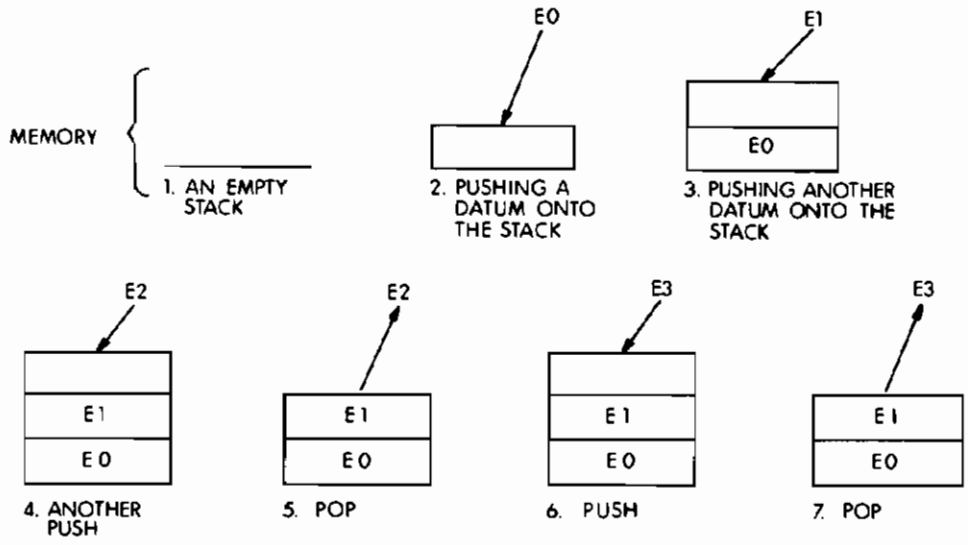
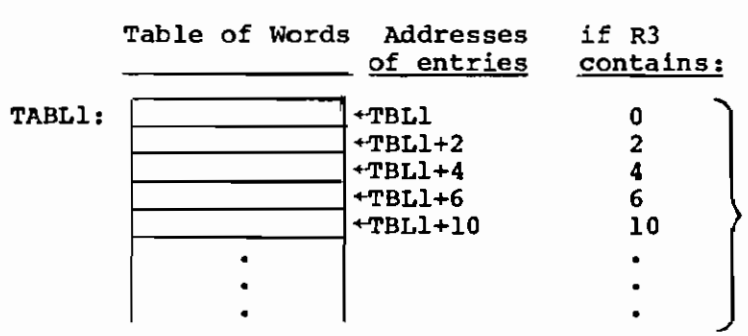


Figure 2-3 Illustration of Push and Pop Operations

Random Access of Tables

Direct access to an entry in the middle of a stack, or in any kind of table, is accomplished through Index Mode. The contents of a register are added to a base (fetched from the word or second word following the instruction) to calculate an address. With this facility a fixed-order element of several tables, or several elements of a single table, may be accessed.



When deferred Index Mode is specified (i.e., @TBL1(R3)), the calculated address contains a pointer to the data, rather than containing the data itself. Byte tables are discussed in Section 2.2.

Address Modes

Addressing modes may be summarized as follows and are discussed in detail in Chapter 5.

Table 2-1
Addressing Modes

Non-deferred Modes		
Assembler Syntax	Mode	Typical Use
Rn	Register	Accumulator
(Rn)+	Autoincrement	Sequential pointer to data in a table; popping data off a stack
-(Rn)	Autodecrement	Sequential pointer to data in a table; pushing data on a stack
A(Rn)	Index	Random access to stack or table entry
Deferred Modes		
Assembler Syntax	Mode	Typical Use
@Rn or (Rn)	Deferred Register	Pointer to an address

(Continued on next page)

Table 2-1 (Cont.)
Addressing Modes

Assembler Syntax	Mode	Typical Use
@(Rn)+	Deferred Autoincrement	Sequential pointer to addresses in a table; popping address pointers off a stack
@-(Rn)	Deferred Autodecrement	Sequential pointer to addresses in a table; pushing address pointers on a stack
@A(Rn)	Deferred Index	Random access to table of address pointers

Accessing Unstructured Data

Addressing of unstructured data becomes greatly facilitated through the use of the Program Counter (R7) as the specified register in these modes. This is particularly true of Autoincrement and Index Modes, which are mentioned below, but discussed more fully in Chapter 5.

Autoincrement Mode using R7 is the way immediate data is assembled. This mode causes the operand itself to be fetched from the word (or second word) following the instruction. It is designated by preceding a numeric or symbolic value with #, and is known as Immediate Mode. The instruction:

```
ADD #50,R3
```

causes the value 50(octal) to be added to the contents of register 3. If the # is preceded by @, the immediate data is interpreted as an absolute address; i.e., an address that remains constant no matter where in memory the assembled instruction is executed.

Index Mode using R7 is the normal way memory addresses are assembled. This is relative addressing because the number of byte locations between the Program Counter (which contains the address of the current word+2) and the data referenced (destination address minus PC) is placed in the word (or second word) following the instruction. It is this value that is indexed by R7--the Program Counter--as follows:

(Destination-PC)+PC=Destination

Relative Mode is designated by specifying a memory location either numerically or symbolically (e.g., TST 100 or TST A). If a memory address specification is preceded by @, it is in deferred Relative Mode and the contents of the location are interpreted by the instruction as a pointer to the address of the data.

2.2 INSTRUCTION CAPABILITY

The twelve ways of specifying an operand demonstrate the flexibility of the PDP-11 in accessing data according to how it is structured, and even if it is not structured. Each instruction adds to this versatility by acting on an operand in a way particularly suited to its task. For example, the task of adding, moving, or comparing implies the use of two operands in any of the twelve addressing forms; whereas the task of clearing, testing, or negating implies only one operand. Examples:

```
ADD #12,GROUP(R2)
MOV MEM1,MEM2
CMP (R4)+,VALUE
CLR R3
TST SUM
NEG @-(R5)
```

Some instructions have counterparts which operate on byte data rather than on full words. These byte instructions are easily recognized by the suffixing of the letter B to the word instruction. MOV is one such word instruction; e.g., MOVB #12,GROUP(R2) would move an 8-bit value of 12(octal) to the 8-bit byte at the address specified. One implication of byte instructions is that when using Autoincrement or Autodecrement Mode, a table of bytes is being scanned. The Autoincrement or Autodecrement therefore goes by one in byte instructions, rather than by two. However, because of their specialized processor functions, R6 and R7 in these modes always increment or decrement by two.

Forms other than single or double operand instructions include operate instructions such as HALT and RESET which take no operands, branch instructions which transfer program control under specified conditions (see Chapter 5), subroutine calls and returns, and trap instructions (see Appendix B for the complete instruction set).

2.3 PROCESSOR USE OF STACKS

Because of the nature of last-in-first-out data structures, the same stack can be used to nest multiple levels of interrupts, traps, and subroutines (see Figure 2-4).

2.3.1 Subroutines

In subroutine calls (JSR Register,Destination) the contents of the specified register are saved on the stack (the processor always uses R6 as its Stack Pointer) and the value of the PC (return address following subroutine execution) becomes the new value of the register. This allows any arguments following the call to be referenced via the register. The command RTS Register causes the return from the subroutine by moving the register value into the PC. It then pops the saved register contents back into the register. (Return from a subroutine is made through the same register that was used in its call.)

2.3.2 Interrupts

When the processor acknowledges a device interrupt request, the device sends an interrupt vector address to the processor. The processor then pushes the current Status (PS) and PC onto the stack and picks up a new PS and PC (the interrupt vector) from the address specified by the device. Another acknowledged interrupt before dismissal will cause the PS and PC of the running device service routine to be pushed onto the stack and the address and status of the new service routine to be loaded into the PC and PS. A process can be resumed by popping the old PC and PS from the stack into the current PC and PS with the ReTurn from Interrupt (RTI) instruction.

2.3.3 Traps

Traps are processor generated interrupts. Error conditions, certain instructions, and the completion of an instruction fetched while the T bit was set all cause traps. As in interrupts, the current PC and Status are saved on the stack and a new PC and Status are loaded from the appropriate trap vector. The instruction RTI provides for a return from an interrupt or trap by popping the top two words of the stack back into the PC and PS.

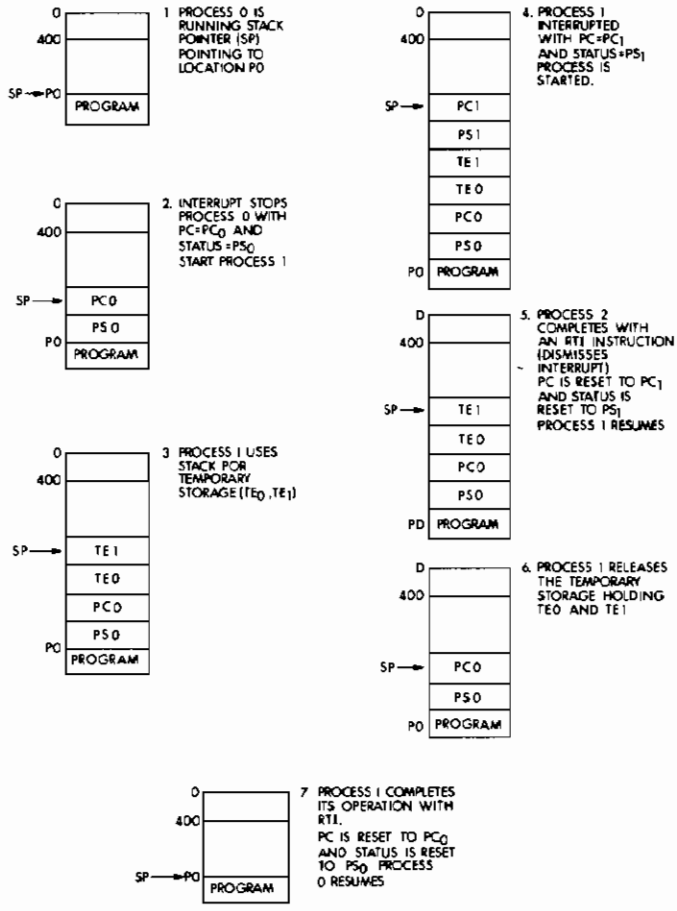


Figure 2-4 Nested Device Servicing

CHAPTER 3

USING THE CAPS-11 MONITOR

The Cassette Programming System is stored on a single cassette, called the System Cassette, which contains all the programs necessary for loading the Keyboard Monitor into memory and creating and running system and user programs. (The System Cassette supplied to the user is configured for an 8K system. Refer to Appendix E for instructions concerning building a System Cassette for any size configuration.) The directory of the System Cassette is as follows:

```
CTLOAD SYS
CAPS11 S8K
PIP    SRU
EDIT   SLG
LINK   SRU
ODT    SLG
PAL    SRU
DEMO   PAL
```

The Monitor is loaded into memory from the System Cassette by means of a short bootstrap program. Once in memory, the Monitor accepts commands from the console terminal keyboard which allow the user to run system and user programs, and create, assemble, load, execute, and debug programs, utilizing cassettes for all data storage.

3.1 LOADING INSTRUCTIONS

The first operation in using the CAPS-11 System involves loading the Monitor into memory from the System Cassette. The loading process may be accomplished by following steps 1 through 4 below:

1. Ensure that the computer and console terminal are on-line.

2. Place the System Cassette (write-locked to protect data) onto cassette drive 0 (facing the computer, drive 0 is to the left of the cassette unit).

3. Press and raise the HALT key (leaving it in the ENABLE position).

4. Load and start the system bootstrap loader (called CBOOT). This can be done in one of two ways:

   a) If the system has a hardware bootstrap, set 173300 in the Switch Register, press LOAD ADRS and START (part b may be ignored).

   b) If no hardware bootstrap is available, CBOOT must be manually loaded and started by the user. Two versions of CBOOT are provided. The standard version is the version used in the hardware bootstrap and consists of the 28 words listed in Table 3-1. A complete listing and more information concerning CBOOT is provided in Appendix E.

      A shorter (20 word) version called QCBOOT may optionally be loaded by the user. This version does not provide some of the error checking and handling which the longer CBOOT does, but allows a faster means of manually booting the system. A complete listing of QCBOOT is also provided in Appendix E; the binary instructions are listed in the following table:

Table 3-1
CBOOT (QCBOOT) Instructions

Location	CBOOT Contents	QCBOOT Contents
001000	012700	012700
001002	177500	177500
001004	005010	005010
001006	010701	010701
001010	062701	062701
001012	000052	000034
001014	012702	112102
001016	000375	112110
001020	112103	032710
001022	112110	100240
001024	100413	001775
001026	130310	100001
001030	001776	005007
001032	105202	005202
001034	100772	100770
001036	116012	116012
001040	000002	000002
001042	120337	000766
001044	000000	017775
001046	001767	002415
001050	000000	
001052	000755	
001054	005710	
001056	100774	
001060	005007	
001062	017640	
001064	002415	
001066	112024	

After the bootstrap has been manually loaded (using the Switch Register, LOAD ADRS, and DEP keys), set 001000 in the switches, press LOAD ADRS and START.

At this point the RUN lamp should be lit and the System Cassette should begin to move. The bootstrap loader (CBOOT or QCBOOT) calls the first program on the System Cassette (CTLOAD.SYS) which in turn loads the Keyboard Monitor (CAPS11.SYS) into memory. If an error occurs during the loading process (an error may be caused by the cassette being improperly mounted, by a missing file on the tape, or by the occurrence of an I/O error) no error message will inform the user. Instead, the System Cassette may stop moving and the computer will halt. If this condition occurs and the reason for the halt is not immediately apparent, consult Appendix E, which provides more information concerning errors during the loading process.

Once the Monitor has been loaded, the System Cassette stops moving and a dot is typed at the left margin of the console terminal page. A Monitor identification line may also be typed; however, this line will be output only if the Monitor is being loaded for the first time, or if a previously loaded CAPS-11 system has been completely deleted from memory. The total time involved in the loading process (i.e., from the bootstrap initialization on a rewound cassette to the appearance of the dot) is approximately 30 seconds. The dot instructs the user that the Monitor is now in memory and ready to accept input commands.

3.2 SYSTEM CONVENTIONS

File naming procedures, special character commands, error formats, and other conventions which are standard for the CAPS-11 System are presented next. The user should be familiar with these conventions before using the system.

3.2.1 File Formats

The Cassette Programming System makes use of two types of file formats--ASCII and binary.

Files in ASCII format conform to the American National Standard Code for Information Interchange in which alphanumeric characters are represented by an 8-bit code. A chart containing ASCII character codes is provided in Appendix A. Files in ASCII format are generally those created using the Editor.

Files in binary format consist of 8-bit bytes representing data and PDP-11 machine language code. Binary files contain addresses and machine instructions and may be read directly into memory for immediate execution. System programs and object programs the user has created using the Assembler and Linker are in binary format.

3.2.2 Input/Output Devices

There are four categories of input/output devices in the CAPS-11 System; these are the console terminal keyboard and printer, cassette drives 0 and 1, an optional line printer, and an optional high-speed paper tape reader and punch (which may be used only by PIP as discussed in chapter 8). Each device is referenced by means of a standard permanent device name which is recognized by the CAPS-11 System when encountered in an I/O command string. These names are listed in Table 3-2:

Table 3-2
Permanent Device Names

Name	Device
CT0 (or 0)	Cassette Drive 0
CT1 (or 1)	Cassette Drive 1
PP	High-speed Paper Tape Punch
PR	High-speed Paper Tape Reader
LP	Line Printer (LP11 or LS11)
TT	Console Terminal (LT33 or LT35 Teletype, VT05 Display, or LA30 DECwriter)

3.2.3 Filenames and Extensions

System and user files are referenced symbolically by a name of as many as six alphabetic characters (A-Z) or digits (0-9), followed by a period and an optional extension of from 1 to 3 alphabetic characters or digits; the extension is generally used as an aid in remembering the format of a file. The following are examples of legal and illegal filenames:

Legal	Illegal
1TYPE.PAL	@STOW.PAL
ABCDEF.OBJ	FO RM
DATA.	PROGRAM.DAT
PRO.21R	LOAD.34AN
FILE (extension assumed)	

Although the user may call his files by any mnemonic filename and extension he chooses, in most cases, he will want to conform to the standard extensions established for CAPS-11 and listed in Table 3-3. There are two reasons why the standard extensions should be used:

a) If an extension is not specified for an input file (for example, FILE in the preceding list of legal filenames), certain system programs and Monitor commands will perform a search for the filename and an assumed default extension; the Monitor RUN command is one example of a system routine which assumes an extension if no other is indicated.

b) If an extension is not specified for an output file, some system programs will append standard extensions during the output operation; for example, the Assembler will append the extension .LST for the output listing file unless the user designates another.

Standard extensions save the user time in typing the command line and provide consistency in filenaming procedures; the following table lists the default extensions; greater detail is presented in the individual chapters.

Table 3-3
CAPS-11 Default Extensions

Extension	Meaning
.LDA	Linker binary output load module
.LST	Assembler listing output file
.MAP	Linker load map output
.OBJ	Relocatable binary object module (Assembler output, Linker input)
.PAL	Assembler source file (Editor input and output, Assembler input)
	NOTE The next three extensions are default extensions for the Monitor RUN command. See Section 3.3.1 for details.
.SLO	Absolute binary object file (default extension for RUN command, causing an automatic load and overlay of the Monitor as necessary up to CABLDR)
.SLG	Absolute binary object file (default extension for RUN command, causing an automatic Load and Go)
.SRU	Absolute binary object file (normal default extension for the RUN command)
.SYS	CAPS-11 system file (i.e., CTLOAD.SYS, CAPS11.SYS; the extension is reserved for these two files)

3.2.4 Entering I/O Information

As soon as the Monitor has been completely loaded into memory, it responds by printing a dot (.) at the left margin of the console terminal page indicating that it is ready to accept a command from the user. A part of the Monitor called the Keyboard Listener (KBL) is responsible for printing the dot. There are eight commands which the user may type in response to this dot: DATE, ZERO, SENTINEL, DIRECTORY, RUN, LOAD, START, and VERSION. The KBL interprets these commands and in most cases executes them; however, since the Monitor RUN command requires more information from the user, another important part of the Monitor--the Command String Interpreter--must be involved.

The Command String Interpreter (CSI) allows the user to enter command strings which provide necessary information concerning input and output files and devices, file formats to be used in I/O operations, and any other important information needed for the I/O process. The CSI prints an asterisk (*) at the left margin of the console terminal page as soon as it is ready to accept this information.

NOTE

> The user may enter his I/O command string as soon as the asterisk is printed even though program loading (as a result of using the RUN command) may be occurring at the same time. The user should be careful not to manually rewind or dismount the System Cassette while loading is continuing. After loading is complete, the System Cassette will automatically rewind.

The command string which the user enters in response to the asterisk contains all input and output specifications in the following general format:

```
*DEV:OUTPUT.EXT/OPT=DEV:INPUT.EXT/OPT
```

DEV represents one of the permanent device names listed in Table 3-1. If a cassette is the device, only the drive number need be entered separated from the filename by a colon. OUTPUT.EXT and INPUT.EXT represent filenames and extensions, as explained in Section 3.2.3. /OPT represents an option letter from the list described briefly in Table 3-4. Options are separated from the rest of the command line and from one another by a slash character (/) and are indicated in the command string only when the user wishes the associated action to occur. Option usage varies according to the program being used; refer to individual chapters to learn which options are used by each CAPS-11 system program.

Table 3-4
CSI Options

Option	Meaning
/A	ASCII; the file type is set to ASCII (used during a PIP file transfer).
/B:n	Bottom; links the user program with its lowest location at n (used by Linker).
/C	Continuation; indicates that the command string is to be broken into one or more lines. The /C option must be used at the end of each line that is to be continued.
/D	Delete; indicates file deletion (used by PIP).
/F	Forward; indicates that the cassette need not be rewound before searching for the file (i.e., the filename preceding the option is in a forward direction in regard to the tape's current position on the drive). The RUN command assumes this option.
/H:n	High; links the user program with its highest location at n (used by the Linker).
/O	Overflow; used after an output filename, indicating that the file preceding the option is to be created and used only for output overflow conditions. If no filename is indicated, the overflow file will be created under the same name as the most recently opened output file.
/P	Prompt; requests that the system prompt the user to change cassettes on an indicated drive before attempting to access a file. The system prints: #? where # represents the number of the appropriate drive.
/S	Several; used after a Linker input filename to indicate that this filename contains more than one input object module. (Several object modules may be combined under one filename using PIP.)
/T	Transfer address; used after a Linker input filename (object module) to indicate that the transfer address of this object module is to be used as the transfer address of the final load module.
/X	Extended; suppresses extended binary output in an assembly listing (used by PAL).
/Z	Zero; causes all output cassettes indicated in the command line to be zeroed, or completely deleted of files (used by PIP).

The general form of the command line as shown previously consisted of only one input and one output file indicated on a single line. However, from 0 to an unlimited number of filenames may be entered depending upon the system program in use, and the command string can be broken into two or more lines by using the special option character /C. A separator always divides the input specifications from the output specifications and may be any one of the following:

= equal sign

< left angle bracket

← back arrow

The user may omit indicating a permanent device name entirely in a command string if he is aware of how his command line will be interpreted by the Monitor. Consider the following command string:

```
*CT0:FIRST.PAL,LP:=CT1:TASK.1,CT1:TASK.2,CT0:TASK.3/C
,CT0:TASK.4
```

This command string contains two 'lists' of device designations--the output 'list' contains CT0 and LP; the input 'list' is made up of CT1, CT1, CT0, and CT0. Unless the user designates otherwise, the Monitor will always assume that the first device in any 'list' (input or output) is cassette drive 0; all immediately following default (unnamed) devices in this 'list' will also refer to drive 0. This continues until the user specifies a different device using a permanent device name from Table 3-2. Thereafter, all immediate default devices will reflect the most recent user-indicated device. If the first device in a 'list' is not drive 0 (i.e., the user has specified another permanent device name as in the input 'list' above), all default devices will reflect this user-indicated device until a different device is specified, and so on. Thus, the above command line could have been written:

```
*FIRST.PAL,LP:=1:TASK.1,TASK.2,0:TASK.3,TASK.4
```

The Command String Interpreter scans the user's command string and constructs a table containing all the input and output information which has been entered. Details concerning this table and more information regarding both the KBL and CSI is provided later in the chapter and in Appendix E.

3.2.5 Special Characters and Commands

The following special characters and commands can be used by the programmer to control execution and correct command lines; these commands may be used while under control of any of the system programs.

Table 3-5
Special Characters/Commands

Character/Command	Meaning
CTRL/C	Control can be returned to the Keyboard Monitor while running any of the system programs by typing a CTRL/C (produced by holding down the CTRL key and simultaneously pressing the C key). A CTRL/C causes a complete rebootstrap (if necessary) of the Keyboard Monitor by reading the appropriate files from the System Cassette on drive 0. The system prints: ↑C? which prompts the user to mount the System Cassette on drive 0 (in the event that it may not already be mounted); typing any character will continue execution of the reboot. If the Monitor is still intact in memory, no reboot is necessary and typing a CTRL/C will echo only ↑C and cause an immediate return to the KBL. When it is ready to accept input, the KBL types a dot at the left margin of the teleprinter page.
CTRL/O	Teleprinter output can be surpressed by typing a CTRL/O (produced by holding down the CTRL key and simultaneously pressing the O key). This allows execution of the program to continue but stops all console printout. Typing a second CTRL/O will resume printout again. Unless output is extremely lengthy, or unless the program is waiting for input from the user, processing of a program after an initial CTRL/O has been typed will usually be completed before the user is able to type a second CTRL/O. Printout will automatically resume when control is returned to the Keyboard Listener (indicated by a dot at the left margin). NOTE CTRL/O does not suppress line printer output, and does not prevent certain important error messages from printing on the console terminal. CTRL/O is treated somewhat differently when using the CAPS-11 Linker to produce a load map. Refer to Chapter 6 for details.

(Continued on next page)

Table 3-5 (Cont.)
Special Characters/Commands

Character/Command	Meaning
CTRL/P	A CTRL/P (produced by typing the CTRL and P keys simultaneously) during the initiating of a Monitor command echoes ↑P and causes control to return to the Keyboard Listener, indicated by a dot at the left margin. A CTRL/P typed during the initiating of a CSI command string echoes ↑P and causes a re-initialization of the Command String Interpreter, indicated by an asterisk at the left margin. During execution of a user program, a restart address may have been specified by the user within his program so that a CTRL/P will cause a restart of that program rather than of the Monitor. Refer to the CTRL/P RESTART IOT (Chapter 9, Section 9.4.1) for details.
CTRL/U	A line currently being entered (whether as part of a command or as text) may be ignored by typing a CTRL/U (produced by typing the CTRL and U keys simultaneously). A ↑U is echoed followed by a carriage return/line feed (when using the Editor, an asterisk is also printed); the user may enter a new line. (This command produces the same results as typing RUBOUTs back to the beginning of a line.)
RUBOUT	A RUBOUT (produced by pressing the RUBOUT key) causes a deletion of the most recently typed character and echoes the deleted character on the terminal. Each successive RUBOUT deletes and echoes one more character (up to the preceding carriage return/line feed, after which successive RUBOUTs will not echo nor delete any characters).

3.2.6 Error Message Format

Error messages are printed whenever the Keyboard Monitor is used incorrectly, or when an I/O error occurs while using Monitor commands and system programs, or upon occurrence of a hardware error. The appropriate message is printed on the console terminal at the time the error occurs; the message is preceeded by either a question mark or a percent sign to indicate one of the following:

%	Fatal error; execution of the command cannot be continued further and control returns to the KBL. A dot is printed at the left margin of the teleprinter page when the Monitor is ready to accept another command.
?	Non-fatal error; if possible, execution of the command will continue after the error message is printed on the console terminal; if further execution is not possible, control will return to the CSI and the user may enter another command string.

A list of Monitor error messages is provided in Section 3.7.

3.3 KEYBOARD MONITOR COMMANDS

There are eight Keyboard Monitor commands which may be typed in response to the dot printed by the Keyboard Listener; they are entered when the RETURN key is pressed. Any error made while utilizing these commands will result in a message informing the user. After occurrence of an error, control returns to the KBL and the command must be retyped. Monitor commands generally require only a single command line which specifies the device, filename(s), and switch(es) in the following format:

```
.COMMAND/SW DEV:FILENA.EXT
```

COMMAND represents one of the eight Monitor commands. SW represents a switch--an alphabetic character separated from the command and from another switch character by a slash (/); switches are similar to the CSI options discussed in Section 3.2.4, but perform different functions and are valid only when used with Monitor commands; switches are discussed individually in sections concerning the commands with which they are used. The device (DEV), if specified, will always be a cassette, so the user may enter only the drive number rather than the entire permanent device name if he wishes. With the exception of the ZERO command, drive 0 is always assumed, so the user may omit the device specification entirely if CT0 is the device. FILENA.EXT represents the file being accessed; the filename must be separated from the drive number (if indicated) by a colon.

Throughout this section, optional entries in the command line are enclosed in square brackets.

3.3.1 RUN Command

The RUN command is of the form:

```
.R[UN] [[CT]#:]FILENA[.EXT]
```

The RUN command instructs the Monitor to load and execute the file specified in the command line; this file must be in absolute binary format. If the user omits the extension (as is generally the case

when calling system programs), the Monitor will search the indicated cassette for the file as the user has designated it in the command line. However, it assumes that it will find the filename followed by one of three extensions: .SLO, .SLG, or .SRU; the first file found which has the indicated filename and one of these extensions will be accessed.

The extensions used by the RUN command are interpreted as follows:

.SLO — The file is an absolute binary object file and will be loaded into memory overlaying as necessary all parts of the Monitor as far as CABLDR (see Section 3.5); after the file is loaded, it is automatically started. RUNning a file with this extension is identical to a LOAD/O of the file. Presently, no system programs use the .SLO extension; however, it is available for future system expansion and for general use.

.SLG — The file is an absolute binary object file and will be loaded into memory to the bottom location of CLODll (see Section 3.5). Execution is automatic. System programs which use this extension are EDIT.SLG and ODT.SLG. Using this extension is the same as using the /G switch with the LOAD command.

.SRU — The file is an absolute binary object file and will be loaded into memory and automatically started. This is the normal default extension for the RUN command and is used by the following system programs: PIP.SRU, PAL.SRU, LINK.SRU. Using this extension is similar to using the LOAD command except that execution is automatic and more I/O information must be provided by the user, thus involving the CSI.

For example, assume the directory of cassette drive 1 is as follows:

```
TABLE  1
FORM   SRU
FIELD  PAL
FORM   SLG
```

If the user types:

```
.R 1:FORM
```

The cassette on drive 1 will be searched for the first file consisting of the name FORM and one of the three extensions; in this case the first file meeting these requirements is FORM.SRU. This file is loaded into memory and executed. After the file is loaded, the cassette is automatically rewound; thus, if the user wishes to access the file FORM.SLG, he must either delete the file FORM.SRU from the cassette (see Chapter 8), or specify the entire filename in the command line as follows:

```
.R 1:FORM.SLG
```

If a user program with no specified transfer address is loaded via RUN, the fatal error message:

```
%NO START ADDR
```

will be printed. If the file indicated in the command line is not present on the cassette, the fatal error message:

```
%FILE NOT FND
```

will be printed.

3.3.2 LOAD Command

The LOAD command is used to load an absolute binary file into memory and takes the following form:

```
.L[OAD][/SW] [[CT]#:]FILENA.EXT
```

/SW represents either a /O or /G switch. If neither switch is indicated in the command line, the command allows loading only to the bottom location of the KBL without error (see Figure 3-1 in Section 3.4). At the completion of the load, the KBL prints a dot to indicate that it is still intact and ready to receive another Monitor command (typically either START or another LOAD).

LOAD used with a /G switch directs a program load to the bottom location of CLOD11, and then initiates a 'GO' (START) at the specified transfer address. If this is absent, the fatal error message:

```
%NO START ADDR
*C?
```

will be printed. Since the KBL and CSI are 'marked' (or assumed) as being overwritten when the /G switch is used, the Monitor must be rebooted from the System Cassette on drive 0.

LOAD used with a /O switch allows a program to be loaded even if its size requires overwriting the entire Monitor. Such a program must handle its own I/O and other functions since no part of the Monitor may be available to do this. This type of program is started at its transfer address; if none has been indicated, CABLDR will halt and expect user console action (information concerning a CABLDR halt is provided in Appendix E).

Section 3.5 provides greater detail concerning the loading process when RUN or any form of the LOAD command is used.

3.3.3 START Command

The START command is of the form:

```
.ST[ART] [nnnnnn]
```

and is used to start a program which has been loaded into memory using the LOAD command without a switch. nnnnnn is an optional absolute starting address for the program, and if indicated, will cause program control to be transferred to this address. If not indicated, the last specified transfer address of the program(s) loaded will be used. If no transfer address exists, an error message is printed and control returns to the KBL.

For example, the program LDT.SLG on cassette drive 1 is loaded and started at location 1000 as follows:

```
.LOAD 1:LDT.SLG

.ST 1000
```

3.3.4 DATE Command

The DATE command is of the form:

```
.DA[TE] dd-mmm-yy
```

where dd, mmm, and yy represent the current day, month, and year as entered by the user. One- or two-digit numbers in the range 1-31 are entered in the day portion; the first 3 characters of the month are entered in the month portion of the command; digits in the range 0-99 are entered in the year portion. The Keyboard Monitor checks for the entry of a number which is outside the ranges allowed and for characters which are not the first three characters of one of the twelve months; if any error is found, a message is printed and a blank date is produced (i.e., the location in which the date is stored is padded with nulls and dashes are printed during directory listings).

The current date as entered by the user will appear in directory listings (see Section 3.3.5), in Linker load maps, and in PAL assembly listings, and the date of creation of all new files will also be included. If the date command is not used, directory listings will contain only filenames, extensions, and previous creation dates.

When the user enters a date, it is stored in a part of memory that is not likely to be overwritten (and therefore destroyed) by the user or by the CAPS-11 System. The user should update his system from day to day to prevent wrong dates from being assigned to files. Very infrequently (if ever) that part of memory holding the date may be overwritten in such a way as to cause random characters to be printed in place of the date. The user need only type in the current date using the DATE command to correct this condition.

3.3.5 DIRECTORY Command

The DIRectory command is of the form:

```
.DI[R][/F] [[CT]#:]
```

and causes a directory listing of the cassette on the indicated drive to be output on the console terminal. For example:

```
.DIR CT1:

 03-APR-73

FAD     SLG  03-APR-73
BA      1    27-MAR-73
```

The /F switch is optional; if used, it causes a "fast" listing to be produced by omitting current and creation dates and listing only filenames and extensions. For example:

```
.DIR/F 1:

ABC     SLG
*EMPTY
PRO     LDA
```

If a file has been deleted from a cassette using PIP (see Chapter 8) its filename and extension will be replaced by the header *EMPTY in the directory listing. To delete *EMPTY files from cassettes, the user must first transfer all needed files to another cassette (using PIP) and then zero the first cassette, or use the SENTINEL command, explained in Section 3.3.7.

If no sentinel file is present on the cassette, the error message:

```
%NO SENTINEL FILE
```

will be printed following the directory listing. (This condition occurs when an open file on cassette has not been properly closed.) The user should write a sentinel file on the cassette using the SENTINEL command. While files may be read from a cassette which contains no sentinel file, they may not be written.

3.3.6 ZERO Command

The ZERO command is of the form:

```
.Z[ERO] [CT]#:
```

and causes the indicated cassette to be zeroed, or completely deleted of files; the sentinel file is written at the beginning of the cassette so that the entire tape is available for use. A cassette number must always be indicated as the ZERO command does not assume drive 0.

All new cassettes should be zeroed before they are first used. This ensures that a sentinel file is present at the beginning of the tape.

3.3.7 SENTINEL Command

The SENTINEL command allows the user to 'zero' part of a cassette by deleting all files following a given filename. The form of the command is:

```
.SE[NTINEL][[CT]#:]FILENA.EXT
```

This command causes the sentinel file to be written immediately following FILENA.EXT, thereby effectively 'zeroing' the remainder of the cassette. For example, assume the directory of the cassette on unit drive 1 is:

```
SIZE   LST
*EMPTY
BLANK  SLG
FORTY  DAT
```

and the user types:

```
.SE 1:SIZE.LST
```

The directory of the cassette will now read:

```
SIZE   LST
```

Cassette drive 0 is assumed if no drive number is indicated in the command line.

3.3.8 VERSION Command

The VERSION command is used to find out the version number of the Monitor currently in use. Typing:

```
.V[ERSION]
```

instructs the Monitor to respond with the Monitor identification, version number, and current date. For example:

```
.V

CAPS-11 V01-02
 27-AUG-73
```

Version 01-02 is currently in use. As new versions of the Monitor are released, this number will be updated accordingly. Any communications with Digital Equipment Corporation concerning the CAPS-11 System should indicate the version number of the Monitor currently in use.

3.4 KEYBOARD MONITOR SECTIONS

That part of the CAPS-11 System termed the Keyboard Monitor (and stored on the System Cassette as CAPS11.SYS) is actually composed of several subprograms (such as CSI and KBL) which are responsible for various stages of system and user interaction. As already mentioned, the first step in using the CAPS-11 System is to bring these subprograms into memory and begin their execution. The user begins the loading process when he starts the bootstrap loader (CBOOT). When the Monitor has been completely loaded and is ready for use, it resides in memory as shown in Figure 3-1:

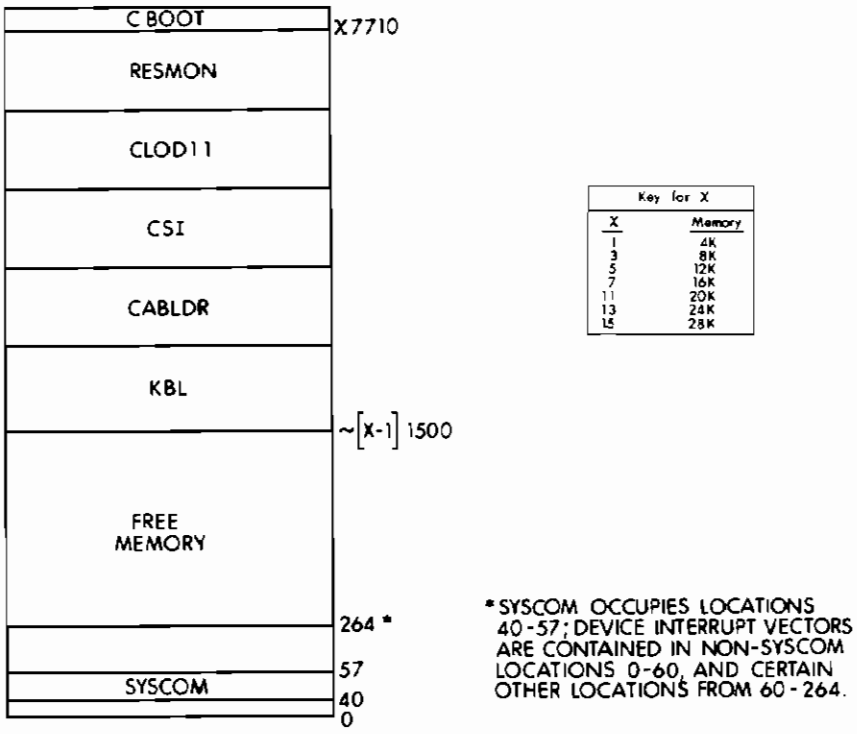


Figure 3-1 CAPS-11 Memory Map

Each of the Monitor subsections will be discussed briefly. A detailed description of the Monitor loading process, information concerning loader formats, and Switch Register settings for use with Monitor loads and error halts may be referenced in Appendix E.

3.4.1 Cassette Bootstrap (CBOOT)

The Cassette Bootstrap is used to load and start any program which is in 'CBOOT Loader Format' (such as CTLOAD.SYS). CBOOT has already been mentioned in Section 3.1 as being instrumental in loading the CAPS-11 Monitor into memory. A complete listing of CBOOT and most information concerning its use in the CAPS-11 System is provided in Appendix E.

3.4.2 Resident Monitor (RESMON)

Input and output operations are handled by RESMON, which contains routines for all file-structured cassette I/O, and all teleprinter, keyboard, and line printer I/O (with the exception of CABLDR which contains the I/O routines necessary for performing the LOAD/O command, as described in Section 3.4.5) Usually RESMON is never overwritten but is always available in memory for access by the user (again, an exception occurs when processing the LOAD/O command). Chapter 9 provides specific information concerning the way RESMON works and methods by which the programmer can utilize RESMON in his own programs.

RESMON also contains the System Communication Area (SYSCOM), which provides to the user and to various system programs information concerning available memory and locations of important Monitor routines (see Section 3.4.7).

3.4.3 Cassette Loader for CAPS-11 (CLOD11)

CLOD11 is used in the execution of the RUN, LOAD, and LOAD/G Monitor commands by directing the loading of programs when these comands are issued. In the case of the RUN command, the user may simultaneously interact with the CSI while program loading is occurring (i.e., he may enter his I/O command string even though the program load is in progress). CLOD11 performs error checking and reports certain types of errors to the user; these are listed in Table 3-8.

3.4.4 Command String Interpreter (CSI)

The Command String Interpreter (CSI) is used by all system programs (with the exception of the Editor and ODT) and may be used by any user program which is loaded and started via the Monitor RUN command. When the user runs a program, the CSI responds by printing an asterisk (*) at the left margin of the console terminal page; the user responds by entering all device and file I/O information needed by the program. The CSI then constructs a table which contains the information entered by the user. This table is described in more detail in Section 3.5.

3.4.5 Cassette Absolute Loader (CABLDR)

CABLDR is used to load programs written in 'Absolute Binary Format' which is the format of all system programs and all Linker output (see Chapter 6). CABLDR performs error checking during program loads and halts upon any error indication, at which time the user may set the Switch Register to direct further action. Refer to Appendix E for detailed information concerning user interaction with CABLDR.

3.4.6 Keyboard Listener (KBL)

The Keyboard Listener is that part of the Monitor responsible for printing the dot at the left margin of the teleprinter page, indicating to the user that he may enter any one of the eight Monitor commands discussed in Section 3.3. The KBL is also responsible for positioning the cassette tape for proper loading during a RUN, LOAD, or LOAD/G command; it then passes control on to CLOD11, which handles the actual loading during processing of these commands.

3.4.7 System Communication (SYSCOM)

The System Communications Area (SYSCOM) resides in absolute locations 40 through 57 and is loaded into memory (as part of the RESMON source code) as shown previously in Figure 3-1. This area provides a means of communication between the Monitor and other programs not linked with it, such as system and user programs.

The following information is classed into two sections--that which is of general interest to the user, and that which is used by CAPS-11 system programs and which may be helpful to user programs requiring

non-standard services. The user should refer to Section 3.5 in conjunction with this information.

SYSCOM--General Information

During normal system use, the absolute locations listed in Table 3-6 are accessed and manipulated by the CAPS-11 System as noted:

Table 3-6
General Locations

Location	Function
HIFREE	Absolute location 42--this word contains the address of the highest location available to the user for program loading and storage which precedes the 'expected' portion of the Monitor still residing in memory. For example, after a LOAD/G command, the user can 'expect' that all of RESMON will remain intact, and thus HIFREE will contain an address equal to the start of RESMON minus two (bytes). After a RUN command, HIFREE will usually contain the starting location of the CSI table minus two. After a LOAD/O command, HIFREE will contain the address immediately preceding the beginning of relocated CABLDR; the user can plan to use all locations through the location contained in HIFREE and still preserve CABLDR.
DATPTR	Absolute location 54--this word contains the address in RESMON of the current date (as input by the user via the Monitor DATE command). The six bytes starting at this location contain, in order: two ASCII bytes containing the day two ASCII bytes containing the month number two ASCII bytes containing the year
LPSIZE	Absolute location 40--this byte contains a number which is one greater than the total number of character columns existing on the user's line printer (i.e., 133 (or 205 octal) for the standard system; 81 (or 121 octal) for a non-standard line printer).
HLTERR	Absolute location 41--this byte is examined by the cassette interrupt handler upon every occurrence of a controller error. If this byte has been set to non-zero by the user (never by the system), the interrupt routine will halt whenever an error is detected so that the user may examine the cassette status register. Pressing the CONTinue key on the processor console will cause the software to continue. This byte is provided primarily as a hardware debugging aid.

SYSCOM--Special Information

The following SYSCOM locations exist primarily for use by CAPS-11 system programs and should be accessed by the user with caution. These locations should not be modified except as indicated.

Table 3-7
Special Locations

Location	Function
KBLRES	Absolute location 52--this byte is a flag indicating the state of the non-resident portion of the Monitor. It is initially set to -1 when the system is bootstrapped to indicate that the entire Monitor is resident; it is cleared when a LOAD/G, LOAD/O, or RUN command sets HIFREE above portions of the Monitor in order to allow maximum loading and storage space. KBLRES is interogated by the CTRL/C and fatal error routines to determine whether a complete reboot is necessary or whether the Monitor need only be restarted (if KBLRES=0, the Monitor may not be entirely resident). Certain system programs (EDIT, LINK, ODT, PIP) do not require the extra memory space made available when the Monitor is overwritten. Thus, even though KBLRES is cleared when these programs are loaded, they do not actually use any of the memory space provided between the beginning of the Monitor and the beginning of RESMON. In order to prevent a CTRL/C or fatal error from causing a complete reboot of the Monitor, these programs each reset KBLRES to -1. User programs may also set KBLRES to -1; the user program should be linked with the program KBLRES.OBJ which is supplied on one of the OBJ Cassettes; this process is described in Section 3.5. Since a START command always clears KBLRES after a load is complete, KBLRES must be set to -1 at run-time rather than at load-time (using the instruction MOVB #-1,@#52) in order to ensure that the system will assume the Monitor is intact.
CSIADR	Absolute location 46--this word contains the starting address of the CSI. It is used by certain system programs to call the CSI, enabling entry of another command string after action on a previous string has been completed. Note that the CSI is reusable only if it has not been over-written.

(Continued on next page)

Table 3-7 (Cont.)
Special Locations

Location	Function
KBLADR	Absolute location 50--this word contains the starting address of the KBL which is also the lowest address in the Monitor. System software restarts the Monitor at this address whenever a CTRL/C or fatal error condition occurs providing the Monitor is resident (i.e., if KBLRES is non-zero). Note that the user may compare the address in KBLADR against his use of memory to determine whether his program must set KBLRES in order to allow a quick restart of the Monitor.
ZERCORE	Absolute location 53--this byte (which is initialized to -1) is cleared by the Assembler (PAL) to indicate that memory should be cleared before the final section of PAL is loaded. This is necessary since the portion of PAL containing the symbol table must be loaded into zeroed memory. This byte may be cleared by any user program which requires use of the CSI. Such use is not recommended without a careful reading of the CLOD11 source listing (available from the Software Distribution Center).
CSITBST	Absolute location 44--this word holds the starting address of the CSI table as it resides in memory. It is used by system programs which make use of the CSI, and may be utilized by any user programs which use the CSI.
FILWRD	Absolute location 56--this word contains information which is used by RESMON and ODT to handle differences in console terminals. Some terminals (such as a serial LA30 and a VT05) require that a certain character (e.g., carriage return or line feed which both take longer than most characters to print on the terminal) be followed by a number of 'pad' or 'fill' characters. The low-order byte of FILWRD (absolute location 56) contains the character which must be filled (or 0 if none must be filled); the high-order byte (absolute location 57) contains the number of 'fill' characters required. RESMON and ODT will type this number of nulls (ASCII 000) after the character specified by byte 56.

3.5 USER PROGRAM LOADING PROCESS

The CAPS-11 Monitor attempts to provide the user at all times with maximum loading space and maximum storage space for system and user programs. It does this by allowing unneeded parts of the Monitor to

be overwritten and by moving necessary sections to higher positions in memory. The SYSCOM parameter HIFREE is used at various times during the loading process to indicate the highest location into which user (or system) programs may be loaded and the highest free location available for use which still preserves RESMON (and possibly the CSI table). Since the goal of the system is to maximize such areas, HIFREE will usually be set at points above Monitor components which are not needed for loading or I/O, even though a user program may not actually overwrite the Monitor. When HIFREE is to be set to locations above any Monitor location, the SYSCOM flag KBLRES is cleared to indicate that the Monitor may not be intact. This has the result of causing a physical reboot (requiring approximately 30 seconds) upon occurrence of any fatal error or CTRL/C command, instead of a simple restart of the KBL. In order to avoid a possible physical reboot of the system in cases such as this, the user may link his program with the object module KBLRES.OBJ on the Build Cassette. This program is merely:

```
.ASECT
.=52            ;LOC. OF KBLRES IN SYSCOM
.BYTE-1         ;MARK KBL AS RESIDENT
.END
```

KBLRES.OBJ should be the first program in the Linker input string. Loading of this code will reset the SYSCOM KBLRES flag which is cleared before loading.

The user who wishes to load and execute a program has four methods available to him (reference should be made to Figure 3-1 while reading the following):

1. Assuming the program has an extension of .SRU or any user-assigned extension other than .SLG or .SLO, the RUN command may be used to automatically load and start the program. RUN allows use of the CSI, and RESMON is available to handle all I/O within the user program. The loading procedure is as follows:

   The cassette is first properly positioned for the load; when this is done, the KBL and CABLDR are no longer needed and may be overwritten. The CSI builds a table which contains all the I/O information which the user has entered. 300 bytes are initially reserved for the table, and once it is built and its actual size is determined, it is moved to occupy memory just below CLOD11. Thus, program loading may use all memory space to the bottom location of the CSI table.

   When loading is complete, CLOD11 is no longer needed. In order to maximize free memory space, the CSI table is standardly moved up over CLOD11 so as to be positioned immediately under RESMON. This destroys CLOD11 and makes it necessary to reboot the system upon occurrence of a CTRL/C or fatal error. To avoid this action (in cases where the user program does not need space above the start of the Monitor, i.e., above KBL), the user may link KBLRES.OBJ with his program as described previously; the second movement of the CSI table will thus be prevented.

The 300 bytes originally reserved for the CSI table is a parameter which may be changed by the user during reassembly of the CAPS-11 Monitor.

2. The second choice available to the user for loading and executing a program is to use the LOAD (and START) command. The LOAD command allows a program to be loaded only to the bottom location of the KBL, preserving the entire Monitor for future use.

3. LOAD/G (and RUN used with the .SLG extension) may be chosen to load and start a program as follows:

   Before the file is loaded, the cassette is physically positioned before the data of the file. KBL and CABLDR may then be overwritten since CLOD11 now directs the program load. Program loading may occur to the bottom location of CLOD11. After the load, CLOD11 is no longer needed, so the user has the entire memory below RESMON available for storage space. RESMON is preserved to handle I/O within the user's program. However, the rest of the Monitor is not preserved and no future Monitor commands are possible.

4. The LOAD/O command (and RUN used with a .SLO extension) allows a program to be loaded providing maximum load and storage space. The cassette is positioned for the data, and CABLDR is moved into highest memory with CBOOT, where it directs program loading. Loading may occur to the bottom location of CABLDR in its new position, and after loading, the entire memory is available for storage. Since no part of the Monitor is preserved, the user program must handle its own I/O, and no further Monitor commands or functions are available for use.

3.6 NOTES ON DEVICE HANDLERS

The line printer prints characters as they appear in the buffer. Tabs are output as spaces to the next tab stop (stops occur every 8 character positions). Carriage returns are ignored since a form feed or line feed is assumed to follow causing the carriage to advance to the beginning of the next line. If more than 132 characters in a single line are output, the line printer handler issues a carriage return/line feed after the 132nd character and continues output on the next line. (See Appendix E for instructions regarding changing the length of the LPT line from 132 to 80 columns.)

If the console terminal is an LT33 Teletype containing reader and punch units, these may be used as input/output devices in conjunction with the Teletype keyboard. To punch a tape, simply place the punch unit to ON; to read a tape, place the reader unit to START. Characters will be printed on the Teletype keyboard as they are read or punched.

The high-speed reader and punch may be used by PIP. Refer to Chapter 8 for details.

3.7 KEYBOARD MONITOR ERROR MESSAGES

Table 3-8 lists all error messages output by the system and lists the source of each error. These messages are preceeded by one of two symbols:

?	Non-fatal error; execution continues if possible; otherwise control returns to the CSI after the message is printed.
%	Fatal error; control returns to the KBL (if the Monitor is entirely resident, the user will see the dot printed after the message; if the Monitor is not resident, the system will type ↑C? on the line following the message; the user should ensure that the System Cassette is mounted on drive 0, and then type any character on the keyboard to initiate a reboot).

Some messages may have numeric arguments which follow the message itself; these usually indicate either the drive number or the program counter. Note that messages which have RESMON as their source are those which the user may see during operation of his program.

Note also that CSI error messages ending with a colon (:) are followed by a line containing all command string characters entered until detection of the character in error (which is indicated by a ?).

Table 3-8
Keyboard Monitor Error Messages

Message	Arg	Meaning	Source
IOT	PC	Illegal IOT; user specified an illegal device or data mode, or an illegal RESMON IOT code.	RESMON
NO FILE OPEN	drive #	READ or WRITE with no SEEK or ENTER	RESMON
OFFLINE	drive #	Cassette not mounted; if non-fatal, execution is automatically resumed when the cassette is mounted (if the user improperly mounts the cassette, a fatal error will probably occur)	RESMON
TIMING	drive #	System software did not service an initiated request fast enough	RESMON

(Continued on next page)

Table 3-8 (Cont.)
Keyboard Monitor Error Messages

Message	Agr	Meaning	Source
TRAP	PC	Stack overflow, reference to non-existent memory, illegal or reserved instruction, attempt to reference a word on a byte boundary; the SP at the time of the trap is stored in location 44	RESMON
WRT LOCK	drive #	Cassette write-locked; if non-fatal, execution is automaticaly resumed when the cassette is write-enabled	RESMON
FILE NOT FND		Specified file not found	KBL
ILL CMD		Illegal command	KBL
NO SENTINEL FILE		No sentinel file is present on the tape; this message may occur during use of the DIRECTORY command at that point during the directory listing where the sentinel file is missing	KBL
SYNTAX ERROR		Arguments following a command are illegal	KBL
BAD TAPE		Hardware checksum error (note that this error may also be caused by READ operations initiated on a cassette which is positioned after the sentinel file)	KBL, CLOD11
NO START ADDR		Loaded program had no transfer address	KBL, CLOD11
PROG TOO BIG		Program too big for the memory limits defined by the type of load used	CLOD11
SFTWR CHKSM ERR		Software checksum error (message followed by number of errors)	CLOD11
TRUNCATED FILE		File ends before transfer address load block is found	CLOD11

(Continued on next page)

Table 3-8 (Cont.)
Keyboard Monitor Error Messages

Message	Arg	Meaning	Source
CSI TABLE OVERFLOW		Command string too big for the table	CSI
ILLEGAL CHAR:	(C.S. line)	Illegal character in command string	CSI
ILLEGAL DEVICE:	(C.S. line)	illegal device specification	CSI
ILLEGAL SYNTAX:	(C.S. line)	Illegal syntax in command string	CSI

CHAPTER 4

EDITING THE SOURCE PROGRAM

The Text Editor (EDIT) is used to create and modify ASCII source files. Controlled by user commands from the keyboard, EDIT reads ASCII files from cassette, makes specified changes, and writes ASCII files back to cassette or lists them on the line printer or console terminal.

The Editor considers a file to be divided into logical units called pages. A page of text is generally 50-60 lines long (delimited by form feed characters) and corresponds approximately to a physical page of a program listing. The Editor reads text from the input file into two internal buffers; from these buffers text is then called, a page at a time, into the Text Buffer where the page becomes available for editing. Editing commands can then be used to:

Locate text to be changed

Execute and verify changes

Output a page of text to the output file

List an edited page on the line printer or console terminal

4.1 CALLING AND USING THE EDITOR

The Editor is called from the System Cassette by typing:

```
.R EDIT
```

in response to the dot printed by the Keyboard Listener. When the Editor is in memory and ready to accept I/O specifications, an asterisk (*) is printed at the left margin of the console terminal page.

4.1.1 Editor Options

None of the options previously listed in Chapter 3 are used by the Editor. An automatic overflow feature is provided, however; if the Editor discovers an end-of-tape condition, it prompts the user to mount a new cassette and output is continued on this cassette under the same output filename originally specified by the user (see Section 4.4.6).

4.1.2 Input and Output Specifications (Edit Read and Edit Write)

The Edit Read command opens a file for input. The form of the command is:

```
*ER#:FILENA.EXT
```

where # represents the unit drive number and FILNAM.EXT the file to be opened. If no drive number is specified, the System Cassette--drive 0--is assumed; if no extension is indicated, .PAL is assumed. Any file currently open for input is closed. Edit Read inputs enough text to fill its two internal input buffers; text is not read into the Text Buffer however, and the contents of the other user buffers are not affected.

For example:

```
*ER1:SAMP$$        Open for input the file SAMP.PAL
                   on cassette drive #1 ($ represents
                   typing the ALTMODE key)
```

The Edit Write command sets up a new file for output (however, no text is output to cassette and the contents of the user buffers are not affected). Any current output files are closed and a new output file with the specified name is opened on the indicated cassette drive. The form of the command is:

```
*EW#:FILENA.EXT
```

NOTE

A cassette which is currently open for an output operation cannot be simultaneously opened for an input operation. If this is attempted, the error message:

```
?*I/O CHAN CONFLICT*?
```

is printed. However, a cassette which is currently open for input can be opened for output; the cassette is repositioned to write the output file; no further input from that cassette is then possible until the output file is closed.

If a file with the same name already exists on the cassette indicated in the Edit Write command, the old file will be destroyed when the user executes an EXit or End File command.

The user may create a new file by first opening an output file (via the EW command) and then creating the text using the Insert command (see Section 4.4.9); the new text will be stored on the drive under the filename indicated in the EW command. Since a new file is being created, no input file need be open to perform this operation.

Examples of use of these commands are:

***ER1:TEST.LS$$**	Open the file TEST.LS on cassette drive 1 for input.
***EWFILE1.DAT$$**	Open the file FILE1.DAT on drive 0 (the System Cassette) for output.
***EW1:OUT.TXT$$** ***I...text...**	Open the file OUT.TXT on drive 1 for output. There is no input file; a new file will be created using the Insert command.

4.1.3 Restarting the Editor

The Editor may be restarted at any time (while it is in memory) by typing CTRL/P. This echoes as P on the console terminal followed by a carriage return/line feed. The Command String Interpreter prints an asterisk at the left margin indicating that it is ready to accept another Editor command. All open files are closed and all buffers are cleared.

4.2 MODES OF OPERATION

The Editor operates in one of two different modes: Command Mode or Text Mode. In Command Mode all input typed on the keyboard is interpreted as commands instructing the Editor to perform some operation. In Text Mode all typed input is interpreted as text to replace, be inserted into, or be appended to the contents of the Text Buffer.

Immediately after being loaded into memory and started, the Editor is in Command Mode. The special character (*) is printed at the left margin of the console terminal page indicating that the Editor is waiting for the user to type a command. All commands are terminated by pressing the ALTMODE key twice in succession. Execution of commands proceeds from left to right. Should an error be encountered during execution of a command string, the Editor will print an error message followed by an (*) at the beginning of a new line indicating that it is still in Command Mode and awaiting a legal command. The command in error (and any succeeding commands) are not executed and must be corrected and retyped.

Text mode is entered whenever the user types a command which must be followed by a text string. These commands insert, replace, exchange, or otherwise manipulate text; after such a command has been typed, all succeeding characters are considered part of the text string until an ALTMODE is typed. The ALTMODE terminates the text string and causes the Editor to reenter Command Mode, at which point all characters are considered commands again.

4.3 SPECIAL KEY COMMANDS

Special EDIT key commands are listed in Table 4-1. (Control commands are typed by holding down the CTRL key while typing the appropriate character.)

Table 4-1
EDIT Key Commands

Command	Meaning
ALTMODE	Echoes as a $ character. A single ALTMODE terminates a text string. A double ALTMODE executes the command string. For example: ***GMOV A,BL$-1D$$**
CTRL/C	Echoes at the terminal as C. Typing this command terminates execution of EDIT commands and initiates a return to the KBL. Any open files are first closed, and the contents of the Text Buffer are lost. (see Chapter 3, Section 3.2.5).

(Continued on next page)

Table 4-1 (Cont.)
EDIT Key Commands

Command	Meaning
CTRL/O	Echoes as ↑O. This command inhibits printing on the console terminal until completion of the current command string. Typing a second CTRL/O will resume output (see Chapter 3, Section 3.2.5).
CTRL/P	Echoes as ↑P and restarts the Editor (see Section 4.1.3)
CTRL/U	Echoes as ↑U. This command deletes all the characters on the current input line (see Chapter 3; Section 3.2.5).
CTRL/X	Echoes as ↑X and causes the Editor to ignore the entire command string currently being entered. The Editor prints a carriage return/line feed and an asterisk to indicate that the user may enter another command.
RUBOUT	The RUBOUT key is used to delete a character from the current line and may be used in both Command and Text Modes; it echoes a backslash followed by the character deleted. Each succeeding RUBOUT typed by the user deletes and echoes another character. An enclosing backslash is printed when a key other than RUBOUT is typed. This erasure is done right to left up to the last CR/LF. Note that RUBOUT used under control of the Editor echoes deleted characters somewhat differently than when using other system programs.
TAB	Spaces to the next tab stop. Tab stops are positioned every 8 spaces on the terminal; typing the TAB key causes the carriage to advance to the next tab position.

4.4 COMMAND STRUCTURE

Editor commands can be categorized as belonging to one of five groups: a) those commands which allow text to be input from cassette and output to either cassette, line printer, or the console terminal; b) those commands which allow the character location pointer to be moved; c) those commands which perform searches in the text for specific

characters or strings of characters; d) those commands which cause the text to be modified either by insertion of new text, or deletion or relocation of existing text; and e) a special classification of commands called utility commands.

The general format for the EDIT command string is:

```
nCtext$
or
nC$
```

where n represents one of the legal arguments listed in Table 4-2, C is a one or two letter command, and text is a string of successive ASCII characters. As a rule, commands are separated from one another by a single ALTMODE; however, if the command requires no text, the separating ALTMODE is not necessary. Commands are terminated by a single ALTMODE; typing a second ALTMODE begins execution.

4.4.1 Arguments

An argument is positioned before a command letter and is used either to specify the particular portion of text to be affected by the command or to indicate the number of times the command should be performed. With some commands this specification is implicit and no arguments are needed; other Editor commands require an argument. Table 4-2 lists the formats of arguments which are used by commands of this last type.

Table 4-2
Command Arguments

Format	Meaning
n	n stands for any integer in the range -16383 to +16383 and may, except where noted, be preceded by a + or -. If no sign precedes n, it is assumed to be a positive number. Whenever an argument is acceptable in a command, its absence implies an argument of 1 (or -1 if only the - is present).
0	0 refers to the beginning of the current line.
/	/ refers to the end of text in the current Text Buffer.
=	= is used with the J, D and C commands only and represents -n, where n is equal to the length of the last text argument used.

The roles of all arguments are explained more specifically in following sections.

4.4.2 Command Strings

All EDIT command strings are terminated by two successive ALTMODE characters. Spaces, carriage returns and line feeds within a command string are ignored (they are not ignored if they appear within a text string). Commands used to insert text can contain text strings that are several lines long, in which case each individual line is terminated with a carriage return/line feed (CR/LF) and the entire command is terminated with a double ALTMODE.

Several commands can he strung together and executed in sequence. For example:

```
*BGMOV PC,RØS-2CR1$5KGCLR ØR2$$
```

NOTE

If a command currently being entered by the user is within 10 characters of exceeding the space available in the Command Buffer, the message:

```
* CB ALMOST FULL *
```

is printed (the Command Buffer holds the command string until it is executed; see Section 4.4.10). If the command can be completed within 10 characters, the user may finish entering the command; otherwise he should type the ALTMODE key twice to execute that portion of the command line already completed. The message is printed each time a character is entered in one of the last 10 spaces.

If the user attempts to enter more than 10 characters the message:

```
?CB FULL?
```

is printed and all commands typed within the last 10 characters are ignored. The user again has 10 characters of available space in which to correct the condition.

Execution of a command string begins when the double ALTMODE is typed and proceeds from left to right.

4.4.3 The Current Location Pointer

Most EDIT commands function with respect to a movable reference pointer which is normally located between the most recent character operated upon and the next character in the buffer. At any given time during the editing procedure, the pointer can be thought of as

representing the current position of the Editor in the text. Most commands use this pointer as an implied argument; commands are available for moving the pointer anywhere in the text, thereby redefining the current location and allowing greater facility in the use of other commands.

4.4.4 Character and Line Oriented Command Properties

When using character oriented commands, a numeric argument specifies the number of characters that are involved in the operation. Positive arguments represent the number of characters in a forward direction (in relation to the pointer), negative arguments the number of characters in a backward direction. Carriage return and line feed characters are treated as any other character. For example, assume the pointer is positioned as indicated in the following text; each line of text is terminated by a carriage return/line feed, indicated here by ↵↓:

```
MOV #VECT,R2↵↓
CLR @R2↵↓
```
Pointer is here

The EDIT command -2J causes the Editor to move the pointer backwards by 2 characters.

```
MOV #VECT,R2↵↓
CLR @R2↵↓
```
Pointer is now here

The command 10J advances the pointer forward by 10 characters and places it between the carriage return and line feed characters at the end of the second line.

```
MOV #VECT,R2↵↓
CLR @R2↵↓
```
Pointer is now here

Finally, to place the pointer after the "C" in the first line, a -14J command is used.

```
MOV #VECT,R2↵↓
CLR @R2↵↓
```
Pointer is here

The J (Jump) command is explained in detail in Section 4.4.7.

When using line oriented commands, the numeric argument represents the number of lines involved in the operation. The Editor recognizes a line as a unit when it detects a CR/LF combination in the text. When the user types a carriage return, the Editor automatically inserts a line feed. Positive arguments represent the number of lines forward (in relation to the pointer); this is accomplished by counting CR/LF combinations beginning at the pointer. Hence, if the pointer is at the beginning of a line, a line oriented command argument of +1 represents the entire line between the current pointer and the terminating line feed. If the current pointer is in the middle of the line, an argument of +1 represents only that portion of the line between the pointer and the terminating CR/LF. For example, assume a buffer of:

```
MOV PC,R1)↓                          Pointer is here
ADD #DRIV-.,R1)↓
MOV #VECT,R2)↓
CLR @R2)↓
```

The command to advance the pointer one line (1A) causes the following change:

```
MOV PC,R1)↓
ADD #DRIV-.,R1)↓                     Pointer is now here
MOV #VECT,R2)↓
CLR @R2)↓
```

The command 2A moves the pointer over 2 CR/LF combinations:

```
MOV PC,R1)↓
ADD #DRIV-.,R1)↓
MOV #VECT,R2)↓
CLR @R2)↓                            Pointer is now here
```

Negative arguments represent the number of lines backward in relation to the pointer. Consequently, if the pointer is at the beginning of the line, a line argument of -1 means "the previous line" (moving backward past the first CR/LF and up to but not including the second CR/LF); if the pointer is in the middle of a line, an argument of -1 means the preceding 1 1/2 lines. For example, given the text:

```
MOV PC,R1)↓
ADD #DRIV-.,R1)↓
MOV #VECT,R2)↓                       Pointer is here
CLR @R2)↓
```

A command of -1A moves the pointer back 1 1/2 lines.

```
MOV PC,R1)↓
ADD #DRIV-.,R1)↓                     Pointer is here
MOV #VECT,R2)↓
CLR @R2)↓
```

Now a command of -1A backs the pointer by only 1 line.

4.4.5 Repetitive Execution

Portions of a command string may be executed more than once by enclosing the desired portion in angle brackets (<>) and preceding the left angle bracket with the number of iterations desired. The structure is:

```
C1$C2$n<C3$C4$>C5$$
```

where C1,C2...C5 represent commands and n represents an iteration argument. Commands C1 and C2 are each executed once, then commands C3 and C4 are executed n times. Finally, command C5 is executed once and the command line is finished. The iteration argument (n) must be a positive number (1 to 16384); if not specified, it is assumed to be 1. If the number is negative or too large, an error message is printed. Iteration brackets may be nested up to 20 levels. Command lines are

checked to make certain the brackets are correctly used and match. For example, the following bracket structure is legal:

```
<<><<<><>>>>
```

while these structures are considered illegal and will cause an error message:

```
><><
```

```
<<<>>
```

As an example, assume the user wishes to input a file called SAMP (stored on cassette drive 1) and change the first four occurrences of the instruction MOV #200,R0 on each of the first five pages to MOV #244,R4. He enters the following command line (commands used in this example are explained in detail later in the chapter):

```
*ER1:SAMP$5<R4<BGMOV #200,R0$=J$3<G0$=C4$>>>$$
                              C
                    B
          A
```

The command line contains 3 'sets' of iteration loops (A, B, C) and is executed as follows:

Execution initially proceeds from left to right; the file SAMP on drive 1 is opened for input and the first page is read into memory. The pointer is moved to the beginning of the buffer and a search is initiated for the character string MOV #200,R0. When the string is found, the pointer is positioned at the end of the string, but the =J command moves the pointer back so that it is positioned immediately preceding the string. At this point, execution has passed through each of the first two 'sets' of iteration loops (A, B) once. The innermost loop (C) is next executed three times, changing the 0's to 4's. Control now moves back to pick up the second iteration of loop B and again moves from left to right. When loop C has executed three times control again moves back to loop B. When loop B has executed a total of 4 times, control moves back to the second iteration of loop A, and so forth until all iterations have been satisfied.

4.4.6 Input and Output Commands

Input commands are used to read text into the Text Buffer where it then becomes available for editing or listing. Output commands cause text to be listed on the console terminal or line printer, or written out to cassette. Some commands are specifically designed for either input or output functions, while a few commands serve both purposes.

If an output cassette becomes full during any output operations, the Editor will prompt the user to mount another cassette by printing:

```
#?
```

where # represents one of the drive numbers. After the user has mounted the new cassette the output operation continues. The files may later be combined under one filename using PIP (see Chapter 8).

READ

The Read command (R) causes a page of text to be read from the input file (previously specified in an ER command) and appended to the current contents, if any, of the Text Buffer. The form of the command is:

R

No arguments are used with the R command and the pointer is not moved. Text is input until one of the following conditions occurs:

1. A form feed character (signifying the end of the page) is encountered. At this point, the form feed will be the last character in the buffer; or

2. The Text Buffer is within 500 characters of being full. (When this condition occurs, Read inputs up to the next carriage return/line feed combination, then returns to command mode. An asterisk is printed as though the Read were complete, but text will not have been fully input); or

3. An end-of-file condition is detected (the *EOF* message is printed when all text in the file has been read into memory and no more input is available).

The maximum number of characters which can be brought into memory using a Read command is approximately 5,000 for an 8K system. Each additional 4K of memory allows another 5,000 characters to be input. An error message is printed if the Read exceeds the memory space available, or if no input is available.

WRITE

The Write command (W) moves lines of text from the Text Buffer to the output file (as specified in the EW command). The formats are:

nW Write all characters beginning at the pointer and ending at the nth CR/LF to the output file.

-nW Write all characters beginning on the -nth line and terminating at the pointer to the output file.

0W Write the text from the beginning of the current line to the pointer.

/W Write the text from the pointer to the end of the buffer.

The pointer is not moved and the contents of the buffer are not affected. If the buffer is empty when the Write is executed, no characters are output.

Examples:

*5W$$	Write the next 5 lines of text starting at the pointer to the current output file.
*-2W$$	Write the previous 2 lines of text, ending at the pointer to the current output file.

NEXT

The Next command acts as both an input and output command since it performs both functions. First it writes the current Text Buffer to the output file, then clears the buffer, and finally reads in the next page of the input file. The Next command can be repeated n times by indicating an argument before the command. The command format is:

nN

Next accepts only positive arguments and leaves the pointer at the beginning of the buffer upon completion of the operation. If fewer than n pages are available in the input file, all available pages are read in, output to the output file, and deleted; the pointer is left positioned at the beginning of an empty buffer, and an error message is printed. (N is equivalent to typing the command combination n<B/W/DR> and provides a means of spacing forward, in page increments, through the input file.)

Example:

*2N$$	Write the contents of the current Text Buffer to the output file, clear the buffer and read and write the next page of text; clear the buffer and then read another page.

LIST

The List command prints the specified number of lines on the console terminal. The format of the command is:

- nL Print all characters beginning at the pointer and ending with the nth CR/LF.
- -nL Print all characters beginning with the first character on the -nth line and terminating at the pointer.
- 0L Print from the beginning of the current line up to the pointer.
- /L Print from the pointer to the end of the buffer.

The pointer is not moved after the command is executed.

Examples:

*-2L$$	Print all characters starting at the second preceding line and ending at the pointer.
*4L$$	Print all characters beginning at the pointer and terminating at the the 4th CR/LF.

Assuming the pointer location is as follows:

```
MOVB 5(R1),@R2)↓
ADD R1,(R2)+)↓        Pointer is here
```

The command:

```
*-1L$$
```

Prints the preceding 1 1/2 lines:

```
MOVB 5(R1),@R2)↓
ADD
```

VERIFY

The Verify command prints the current text line (the line containing the pointer) on the terminal. The position of the pointer within the line has no effect and the pointer does not move. The command format is:

```
V
```

No arguments are used. (V is equivalent to typing 0LL.)

Example:

*V$$ ADD R1,(R2)+	The command causes the current line of text to be printed.

END FILE

The End File command closes the current output file. This command does no input/output operations and does not move the pointer; the buffer contents are not affected. The output file is closed, containing only that text previously output. The form of the command is:

```
EF
```

No arguments are used in the EF command.

EXIT

The EXit command is used to terminate editing, copy the remainder of the input file to the output file, and return control to the Keyboard Listener (the Monitor should be entirely resident in memory so that a reboot is unnecessary). The EXit command performs consecutive Next commands until the end of the input file is reached, then closes both the input and output files. The command format is:

EX

No arguments are used in the EX command.

NOTE

Either an EF or EX command is necessary to make an output file permanent. If a ↑C is typed prior to executing an EF, the current output file will not be saved.

An example of the contrasting uses of the EF and EX commands might be the following: assume an input file called SAMPLE (on cassette drive 0) contains several pages of text. The user wishes to make the first and second pages of the file separate files called SAM1 and SAM2 respectively; the remaining pages of text will then make up the file SAMPLE. This can be done using the following commands:

```
*ER0:SAMPLE$$
*EW1:SAM1$$
*NEF$$
*EW1:SAM2$$
*NEF$$
*EW1:SAMPLE$EX$$
```

The user might note that the EF commands are actually not necessary in this example, since the EW command closes a currently open output file before opening another.

4.4.7 Pointer Relocation Commands

Pointer relocation commands allow the current location pointer to be moved within the Text Buffer. Several commands are available for this purpose.

BEGINNING

The Beginning command moves the current location pointer to the beginning of the Text Buffer. The command format is:

B

There are no arguments. For example, assume the buffer contains:

```
MOVB    5(R1),@R2)↓
ADD     R1,(R2)+)↓
CLR     @R2)↓          Pointer is here
MOVB    6(R1),@R2)↓
```

The B command:

```
*B$$
```

will move the pointer to the beginning of the Text Buffer:

```
MOVB    5(R1),@R2)↓    Pointer is now here
ADD     R1,(R2)+)↓
CLR     @R2)↓
MOVB    6(R1),@R2)↓
```

JUMP

The Jump command moves the pointer over the specified number of characters in the Text Buffer. The form of the command is:

(+ or -)nJ — Move the pointer (backward or forward) n characters.

0J — Move the pointer to the beginning of the current line (equivalent to 0A).

/J — Move the pointer to the end of the Text Buffer (equivalent to /A).

=J — Move the pointer backward n characters, where n equals the length of the last text argument used.

Negative arguments move the pointer toward the beginning of the buffer, positive arguments toward the end. Jump treats CR, LF and form feed characters as any other character, counting one buffer position for each.

Examples:

*5J$$	Move the pointer forward 5 characters
*-4J$$	Move the pointer back 4 characters
*B$GABC$=J$$	Move the pointer so that it immediately precedes the first occurrence of 'ABC' in the buffer.

ADVANCE

The Advance command is similar to the Jump command except that it moves the pointer a specified number of lines (rather than single characters) and leaves the pointer positioned at the beginning of the line. The form of the command is:

nA Advance the pointer forward n lines and position it at the beginning of the n+1 line.

-nA Move the pointer backward past n CR/LFs and position it at the beginning of the -nth line.

0A Advance the pointer to the beginning of the current line (equivalent to 0J).

/A Advance the pointer to the end of the Text Buffer (equivalent to /J).

For example, assume the buffer contains:

```
CLR @R2),↓          Pointer is here
```

The command:

```
*@A$$
```

Moves the pointer as follows:

```
CLR @R2),↓          Pointer is now here
```

4.4.8 Search Commands

Search commands are used to locate specific characters or strings of characters within the Text Buffer.

GET

The Get command is of the form:

```
nGtext$$
```

and searches the current Text Buffer starting at the pointer for the nth occurrence of the text string. If the search is successful, the pointer is left immediately following the nth occurrence of the text string. If the search fails, an error message is printed and the pointer is left at the end of the Text Buffer.

The argument must be positive and is assumed to be 1 if not otherwise specified. The text string may be any length and immediately follows the G command. The search is made on the portion of the text between the pointer and the end of the buffer.

Example:

Assuming the buffer contains:

```
MOV     PC,R1
ADD     #DRIV-.,R1
MOV     #VECT,R2
CLR     @R2          Pointer is here
MOVB    5(R1),@R2
ADD     R1,(R2)+
CLR     @R2
MOVB    6(R1),@R2
```

The command:

```
*BGADD$$
```

positions the pointer as follows:

```
ADD     #DRIV-.,R1   Pointer is here
```

The command:

```
*3G@R2$$
```

positions the pointer:

```
ADD     R1,(R2)+
CLR     @R2          Pointer is here
```

After search commands, the pointer is left immediately following the text object. Using a search command in connection with =J will place the pointer before the text object, as follows:

```
*GOBJ1$=J$$
```

The pointer will now be placed immediately before 'OBJ1':

```
INC OBJ1             Pointer is here
```

FIND

The form of the Find command is:

```
nFtext$$
```

Starting at the pointer, this command searches through the entire text file for the nth occurrence of the character string specified in the command. It combines the Get and Next commands such that if the search is not successful in the current buffer, the contents of the buffer are output to cassette, the buffer contents are then deleted, a new page is read in, and the search is continued. This will proceed until either the search string is found or until the complete source text has been searched. If the search is successful, the pointer is left immediately following the nth occurrence of the text string. If the search fails (i.e., the end-of-file is detected for the input file and the nth occurrence of the text string has not been found), an

error message is printed and the pointer is left at the beginning of an empty Text Buffer. (By deliberately specifying a non-existent search string, the user can close out his file; that is, he can copy all remaining text from the input cassette to the output cassette.)

The argument must be positive and is assumed to be 1 if not otherwise specified.

Example:

***2FMOVB 6(R1),@R2$$**	Search the entire input file for the second occurrence of the text string MOVB 6(R1),@R2. Each unsuccessfully searched buffer is written to the output file.

POSITION

The Position command searches the input file for the nth occurrence of the text string. If the text string is not found, the buffer is cleared and a new page is read from the input file. The format of the command is:

nPtext$

The argument must be positive, and is assumed to be 1 if not otherwise specified. When a P command is executed, the current contents of the buffer are searched from the location of the pointer to the end of the buffer. If the search is unsuccessful, the buffer is cleared and a new page of text is read and the cycle is continued. (The difference between the Find and Position commands is that Find writes the contents of the searched buffer to the output file while Position deletes the contents of the buffer after it is searched.)

If the search is successful, the pointer is positioned after the nth occurrence of the text. If it is not, the pointer is left at the end of an empty buffer.

Example:

***PADD R1,(R2)+$$**	Search the entire input file for the string ADD R1,(R2)+, deleting unsuccessfully searched buffers.

4.4.9 Text Modification Commands

The following commands are used to insert, relocate, and delete text in the Text Buffer.

INSERT

The Insert command causes the Editor to enter Text Mode and allows text to be inserted immediately following the pointer. Text is inserted until an ALTMODE is typed and the pointer is positioned after the last character of the insert. The command format is:

Itext$

No arguments are used in the Insert command and the text string is limited only by the size of the Text Buffer and the space available. All characters except ALTMODE are legal in the text string.

EDIT automatically protects against overflowing the Text Buffer during an Insert. If the I command is the first command in a repetitive command line, EDIT ensures that there will be enough space for the Insert to be executed at least once. If repetition of the command exceeds the available memory, an error message is printed.

Example:

```
*IMOV #BUFF,R2
MOV #LINE,R1
MOVB -1(R2),R0
$$
*
```

Insert the specified text at the current location of the pointer and leave the pointer positioned at the beginning of the line following R0.

DELETE

The Delete command removes a specified number of characters from the Text Buffer. Characters are deleted starting at the pointer; upon completion of the command, the pointer is positioned at the first character following the deleted text. The form of the command is:

(+ or -)nD — Delete n characters (forward or backward from the pointer).

0D — Delete from beginning of current line to pointer (equivalent to 0K).

/D — Delete from pointer to end of Text Buffer (equivalent to /K).

=D — Delete -n characters, where n equals the length of the last text argument used.

Examples:

```
*-2D$$
```

Delete the two characters immediately preceding the pointer.

```
*B$FMOV R1$=D$
```

Delete the text string 'MOV R1.' (=D used in conjunction with a search command will delete the indicated text string)

Assuming a buffer of:

```
ADD    R1,(R2)+
CLR    @R2
```

Pointer is here

The command:

```
±ØD$$
```

leaves the buffer with:

```
ADD     R1,(R2)+
@R2                    Pointer is here
```

KILL

The Kill command removes n lines from the Text Buffer. Lines are deleted starting at the current location pointer; upon completion of the command, the pointer is positioned at the beginning of the line following the deleted text. The command format is:

nK Delete lines beginning at the pointer and ending at the nth CR/LF.

-nK Delete lines beginning with the first character in the -nth line and ending at the pointer.

0K Delete from the beginning of the current line to the pointer (equivalent to 0D).

/K Delete from the pointer to the end of the Text Buffer (equivalent to /D).

Example:

```
±2K$$
```

Delete lines starting at the current location pointer and ending at the 2nd CR/LF.

Assuming a buffer of:

```
ADD     R1,(R2)+
CLR     @R2            Pointer is here
MOVB    6(R1),@R2
```

The command:

```
±/K$$
```

Alters the contents of the buffer to:

```
ADD     R1,(R2)+
CLR                    Pointer is here
```

CHANGE

The CHANGE command replaces n characters, starting at the pointer, with the indicated text string and leaves the pointer positioned immediately following the changed text. The format of the command is:

(+ or -)nCtext$ Replace n characters (forward or backward from the pointer) with the specified text.

0Ctext$ Replace all characters from the beginning of the line up to the pointer with the specified text. (Equivalent to 0X)

/Ctext$ Replace all characters from the pointer to the end of the buffer with the specified text. (Equivalent to /X)

=Ctext$ Replace -n characters with the indicated text string, where n represents the length of the last text argument used.

The size of the text is limited only by the size of the Text Buffer and the space available. All characters are legal except ALTMODE which terminates the text string.

If the C command is enclosed within angle brackets so that it will be executed more than once, and if there is enough space available so that the command can be entered, it will be executed at least once (provided it is first in the command string). If repetition of the command exceeds the available memory, an error message is printed.

Example:

```
*5C#VECT$$
```

Replace the 5 characters to the right of the pointer with #VECT.

=C can be used in conjunction with a search command to replace a specific text string as follows:

```
*GFIFTY:$=CFIVE:$
```

Find the occurrence of the text string FIFTY and replace it with the text string FIVE.

Assuming a buffer of:

```
CLR     @R2
MOV     5(R1),@R2
```

Pointer is here

The command:

```
*0CADDB$$
```

Leaves the buffer with:

```
CLR     @R2
ADDB    5(R1),@R2
```

Pointer is here

Typing nCTEXT$ is equivalent to typing -nDITEXT$.

EXCHANGE

The Exchange command replaces n lines, beginning at the pointer, with the indicated text string and leaves the pointer positioned after the changed text.

The form of the command is:

nXtext$	Replace all characters beginning at the pointer and ending at the nth CR/LF with the indicated text.
-nXtext$	Replace all characters beginning with the first character on the -nth line and ending at the pointer with the indicated text.
0Xtext$	Replace the current line from the beginning to the pointer with the specified text. (Equivalent to 0C)
/Xtext$	Replace the lines from the pointer to the end of the buffer with the specifed text. (Equivalent to /C)

All characters are legal in the text string except ALTMODE which terminates the text.

For example, assuming a buffer of:

```
ADD @R1,(R2)+
CLR @R2                 Pointer is here
```

The command:

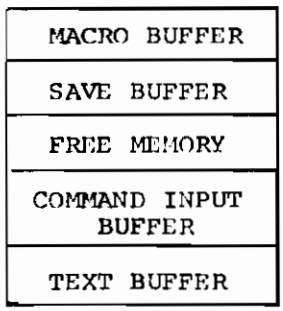
```
*XR1,(R3)+
```

replaces the text to the right of the pointer (on the current line) with the indicated text.

If the X command is enclosed within angle brackets so that it will be executed more than once, and if there is enough memory space available so that the X command can be entered, it will be executed at least once (provided it is first in the command string). If repetition of the command exceeds available memory, an error message is printed.

4.4.10 Utility Commands

The memory area used by the Editor is divided into logical buffers as follows:



The Text Buffer contains the current page of text being edited and the Command Input Buffer holds the command currently being typed at the terminal. Both of these buffers have been previously mentioned.

The Save Buffer contains text stored with the Save (S) command and the Macro Buffer contains the command string macro entered with the Macro (M) command (each are explained next). The Macro and Save Buffers are not allocated space until an M or S command is executed. Once an M or S command is executed, a 0M or 0U (Unsave) command must be executed to return that space to the free area.

The buffers expand and contract to accomodate the text being entered.

SAVE

The Save command copies a specified number of lines starting at the pointer into the Save Buffer. The form of the command is:

nS

The argument (n) must be positive. The pointer position does not change and the contents of the Text Buffer are not altered. Each time a Save is executed, the previous contents of the Save Buffer, if any, are destroyed. If the Save command causes an overflow of the Save Buffer, an error message is printed.

Example:

Assuming the Text Buffer contains the following assembly language subroutine:

```
;SUBROUTINE MSGTYP
;WHEN CALLED, EXPECTS R0 TO POINT TO AN
;ASCII MESSAGE THAT ENDS IN A ZERO BYTE
;TYPES THAT MESSAGE ON THE USER TERMINAL

        .ASECT
MSGTYP: TSTB (%0)               ;DONE?
        BEQ MDONE               ;YES-RETURN
MLOOP:  TSTB @#177564           ;NO-IS TERMINAL READY?
        BPL MLOOP               ;NO-WAIT
        MOVB (%0)+,@#177566     ;YES-PRINT CHARACTER
        BR MSGTYP               ;LOOP
MDONE:  RTS PC                  ;RETURN
```

The command:

```
*B13S$$
```

stores the entire subroutine in the Save Buffer; it may then be inserted in a program whenever needed using the U command.

UNSAVE

The Unsave command inserts the entire contents of the Save Buffer into the Text Buffer at the pointer location and leaves the pointer positioned following the inserted text.

The form of the command is:

U	Insert the contents of the Save Buffer into the Text Buffer.
0U	Clear the Save Buffer and reclaim the area for text.

Zero is the only legal argument to the U command.

The contents of the Save Buffer are not destroyed by the U command (only by the 0U command) and may be Unsaved as many times as desired.

If the Unsave command causes an overflow of the Text Buffer, an error message is displayed.

MACRO

The Macro command inserts a command string into EDIT's Macro Buffer, and is of the form:

M/command string/	Store the command string in the Macro Buffer
0M or M//	Clear the Macro Buffer and reclaim the area for text

/ represents a delimiter character. The delimiter is always the first character following the M command and may be any character which does not appear within the Macro command string itself.

Starting with the character following the delimiter, EDIT places the Macro command string characters into its internal Macro Buffer until the delimiter is encountered again. A double ALTMODE then returns EDIT to Command Mode. The Macro command does not execute the Macro string; it merely stores the command string so that it can be executed later by the Execute Macro (EM) command. Macro does not affect the contents of the Text or Save Buffers.

All characters except the delimiter are legal Macro command string characters, including single ALTMODE's to terminate text commands. All commands except the M and EM Commands are legal in a Macro command string.

In addition to the 0M command, typing the M command immediately followed by two identical characters (assumed to be delimiters) and two ALTMODE characters also clears the Macro Buffer.

Examples:

*M//$$ or *0M$$	Clear the Macro Buffer
*M/BGR0$-C1$/$$	Store a Macro to change R0 to R1

EXECUTE MACRO

The Execute Macro command executes the command string specified in the last Macro command and is of the form:

nEM

The Macro is executed n times and returns control to the next command in the original command string.

The argument must be positive.

Examples:

*B1000EM$$?*SRCH FAIL IN MACRO*?	Execute the Macro stored in the previous example. An error message is returned when the end of buffer is reached. (This Macro effectively changed all occurrences of R0 in the Text Buffer to R1.)
*IMOV PC,R1$2EMICLR @R2$$	In a new program, insert MOV PC,R1, then execute the command string in the Macro Buffer twice before inserting CLR @R2.

4.5 ERROR MESSAGES

The Editor prints an error message whenever one of the error conditions in Table 4-3 occurs. Prior to executing any commands, the Editor first scans the entire command string for syntax errors (format errors such as illegal arguments, illegal combinations of commands, etc.). If an error of this type is found, an error message is printed in the following format:

?ERROR MSG?

and no commands are executed; the user must retype the command.

If the command string contains no syntax errors, execution is started; however, errors during execution are also possible (buffer overflow, I/O errors, etc). If an error is found at during execution, a message of the form:

```
?*ERROR MSG*?
```

is printed. In this case, all commands preceding the one in error will have been executed; the command in error and those following will not be executed. Most errors will generally be of the syntax type and can be corrected before execution.

When an error occurs during execution of a Macro, the message format is:

```
?*message IN MACRO*?
*
```

Table 4-3
EDIT Error Messages

Message	Explanation
?"<>"ERR?	Too deep nesting or illegal use of brackets, or unmatched brackets.
* CB ALMOST FULL *	The command currently being entered by the user is within 10 characters of exceeding the space available in the Command Buffer (see Section 4.4.2).
?CB FULL?	Command exceeded the space allowed for a command string in the Command Buffer.
?*EOF*?	Attempted a Read or Next command and no data was available.
?*FILE NOT FOUND*?	Attempted to open a nonexisting file for editing.
?*HDW ERR*?	A hardware error occurred during I/O.
?ILL ARG?	The argument specified was illegal for the command used. A negative argument was specified where only a positive argument was allowed, or an argument exceeded the range + or -16384.
?ILL CMD?	EDIT does not recognize the command specified.

(Continued on next page)

Table 4-3 (Cont.)
EDIT Error Messages

Message	Explanation
?ILL MAC?	Delimiters were improperly used, or an attempt was made to enter an M command during execution of a Macro, or an attempt was made to execute an EM command while an EM was in progress.
?*ILL NAME*?	The filename or device specified in an EW or ER command is illegal.
?*I/O CHAN CONFLICT*?	An attempt was made to open an input file on a cassette already open for output, or vice versa.
?*NO FILE*?	Attempted to Read or Write when no I/O file was open.
?*NO ROOM*?	Attempted to Insert, Save, Unsave, Read, Next, Change or Exchange when there was not enough room in the appropriate buffer.
?*SRCH FAIL*?	The text string specified in a Get, Find or Position command was not found in the available data.
?*TAPE FULL*?	Available space for an output file is full (i.e., there is no room for any part of the output file).

4.6 EXAMPLE USING THE EDITOR

The following example illustrates the use of the Editor to change a program which is stored on cassette drive 0. Sections of the printout are coded by letter and corresponding explanations follow the example.

```
.R EDIT

*ER0:TEST1.PAL$$
*EW1:TEST2.PAL$$
*R$$
*/L$$
;TEST PROGRAM
        PC=%7
        .GLOBL MSGTYP
START:  MOV #1000,%6    ;INITIALIZE STACK
        MOV #MSG,%0     ;POINT R0 TO MESSAGE
        JSR PC,MSGTYP   ;PRINT IT
        HLT             ;STOP
MSG:    .ASCII /IT WORKS/
        .BYTE 15
        .BYTE 12
        .BYTE 0
*B1J5D$$
*GPROGRAM$V$$
;PROGRAM
*I TO TEST SUBROUTINE MSGTYP. TYPES
;"THE TEST PROGRAM WORKS"
;ON THE COMSO\OSM\NSOLE TERMINAL
$$
*F.ASCII /$8CTHE TEST PROGRAM WORKS$$
*P.BYTE↑U
*F.BYTE 0$V$$
        .BYTE 0
*I
        .END
$B/L$$
;PROGRAM TO TEST SUBROUTINE MSGTYP. TYPES
;"THE TEST PROGRAM WORKS"
;ON THE CONSOLE TERMINAL

        PC=%7
        .GLOBL MSGTYP
START:  MOV #1000,%6    ;INITIALIZE STACK
        MOV #MSG,%0     ;POINT R0 TO MESSAGE
        JSR PC,MSGTYP   ;PRINT IT
        HLT             ;STOP
MSG:    .ASCII /THE TEST PROGRAM WORKS/
        .BYTE 15
        .BYTE 12
        .BYTE 0
        .END

*BGHLT$=CHALT$V$$
        HALT            ;STOP
*EX$$

.
```

A The EDIT program is called and prints an *. The input file is TEST1.PAL on drive 0, and the output file is TEST2.PAL on drive 1; the first page of input is read.

B The buffer contents are listed.

C Be sure the pointer is at the beginning of the buffer. Advance the pointer 1 character (past the ;) and delete "TEST ".

D Position the pointer after PROGRAM and verify the line; the pointer is not moved.

E Text is inserted. RUBOUT is used to correct a typing error.

F Search for .ASCII / and change "IT WORKS" to "THE TEST PROGRAM WORKS".

G CTRL/U is typed to cancel the P command. the F command is then used to search for .BYTE 0 and verify the location of the pointer with V command.

H Insert text. The pointer is returned to the beginning of the buffer and the entire contents of the buffer are listed.

I The user notices that HALT is spelled incorrectly, makes the change and verifies it.

J The input and output files are closed after copying the current Text Buffer as well as the rest of the input file into the output file. EDIT returns control to the Monitor.

CHAPTER 5

ASSEMBLING THE SOURCE PROGRAM

The CAPS-11 Assembler is a two pass assembler (with an optional third pass) which allows the user to create a binary object file from a source program. In the first two passes, the source program (which is generated on-line using the Editor) is translated into an object module which may contain both absolute and relocatable code. Separately assembled object modules may reference one another using special symbols called global symbols. Object modules are then processed by the Linker, producing a load module which may be loaded into memory and executed (the linking process is explained in Chapter 6). During the second (or the optional third) pass, the Assembler produces a complete octal/symbolic listing of the assembled program. The listing is especially useful for documentation and debugging purposes.

This chapter not only explains how to write PAL assembly language programs, but also how to assemble the source programs into object modules. In explaining how to write source programs it is necessary, especially at the beginning of the chapter, to make frequent forward references. The user should first read through the entire chapter to get a "feel" for the language, and then reread the chapter, this time referring to appropriate sections as indicated in order to gain a thorough understanding of the language and assembling procedures.

It is assumed that the user is familiar with the PDP-11 PROCESSOR HANDBOOK and the PDP-11 PERIPHERALS AND INTERFACING HANDBOOK, with emphasis on those sections which deal with the PDP-11 instruction repertoire, formats, and timings; a thorough knowledge of these is vital to efficient assembly language programming.

5.1 CALLING AND USING THE ASSEMBLER

The Assembler is called from the System Cassette by typing:

```
.R PAL
```

in response to the dot printed by the Keyboard Listener. The Command String Interpreter responds by printing an asterisk (*) at the left margin indicating that it is ready to accept input/output

specifications. The user may enter his command line followed by a carriage return even though the remainder of PAL is simultaneously being loaded into memory.

5.1.1 Assembler Options

The options listed in Table 5-1 are valid for use with the Assembler and are indicated by the user in the I/O specification line.

Table 5-1
PAL Options

Option	Meaning
/C	This option allows an I/O specification line to be broken into several segments. The option character is followed immediately by a carriage return and the command string is continued on the next line; this next line must begin with a comma.
/F	This option is valid only after an input filename and specifies that the Assembler should not perform a REWIND operation but should continue searching the cassette in a forward direction for this file. The /F feature saves the user time when he wishes to input several files from one cassette and these files appear on the cassette in the same order as they are to be assembled. The /F option prevents the Assembler from performing a REWIND before accessing each file.
/O	This option is valid only after an output filename and indicates that the file (immediately preceding the option) is to be created and used only if a previously opened output file has been written to the end of the cassette and more output remains. All output files should later be combined under one name using PIP (see Chapter 8).
/P	This option is used whenever a file referenced in the I/O specification line exists on a cassette which is not currently mounted on a drive. Before attempting to search for the file, the Assembler instructs the user to mount the proper cassette on the drive by printing #? where # represents the drive number. After the user has switched cassettes on the drive, he may continue execution by typing any character on the keyboard.

(Continued on next page)

Table 5-1 (Cont.)
PAL Options

Option	Meaning
/X	This option is valid only after an output filename and causes extended binary output (i.e., those locations and binary contents beyond the first binary word per source statement) to be suppressed from the listing.

5.1.2 Input and Output Specifications

Input and output specifications are typed by the user in response to the asterisk printed by PAL. The format of the command string is:

```
*DEV:FILE.BIN/OPT,DEV:FILE.LST/OPT=DEV:INPUT.1/C
,DEV:INPUT.2/OPT,...DEV:INPUT.n/OPT
```

DEV represents the device, FILE.BIN represents the binary output file and FILE.LST represents the listing output file. Null output of either the binary or listing file is represented by a single comma in the command line. For example:

```
*1:FILE.BIN,=INPUT.PAL
```

causes only the binary file to be produced. Any number of input files (INPUT.1...INPUT.n) is permitted. OPT represents any one (or more) of the options listed in Table 5-1.

If both the binary and listing output files are to be sent to cassette, the Assembler will require three passes since it cannot output these two files simultaneously. Otherwise only two passes are required.

Under an 8K system, control returns to the Monitor following the assembly process; under systems greater than 8K, control returns to the CSI, indicated by an asterisk, and the user can enter another command line.

5.1.3 Restarting the Assembler

The Assembler may be restarted at any time (while it is in memory) by typing CTRL/P. This echoes as ↑P on the console terminal and is followed by a carriage return/line feed. Note that this restarts the Assembler but does not always allow the user to input a new command string. In 8K systems, the CSI has been overlaid by the Assembler and cannot be accessed; therefore, typing CTRL/P will restart the assembly already in progress. In larger systems, the CSI is not destroyed and typing '↑P' while PAL is running will allow the user to enter a new command string.

5.2 CHARACTER SET

The following ASCII characters are used in writing a PAL source program (see Appendix A):

1. The letters A through Z. (Both upper and lower case letters are acceptable, although lower case letters will be converted to upper case letters upon input.)

2. The numbers 0 through 9.

3. The following separating or terminating symbols:

   : = % # @ ( ) , ; " ' + - & !

   carriage return tab space line feed form feed

4. The characters . and $ are valid but are generally reserved for use by system software.

5.3 STATEMENTS

A PAL source program is composed of a sequence of statements, each on a single line terminated by a carriage return/line feed (CR/LF) or carriage return/form feed combination.

NOTE

Since the carriage return is a required statement terminator, the Assembler inserts a carriage return before any line feed or form feed not immediately preceeded by one. If the CAPS-11 Editor is used to create the source program, any carriage return typed by the user automatically generates a line feed character.

The statement itself may be composed of as many as four fields which are identified by their order of appearance and by specific terminating characters. The four fields are catagorized as:

```
Label: Operator Operand      ;Comment
```

The label and comment fields are optional. The operator and operand fields are interdependent; that is, either one may be omitted depending upon the contents of the other.

5.3.1 Labels

A label is a symbolic name created by the programmer (see Section 5.4.2) to identify the location of a statement in the program. It always occurs first in a statement and must be terminated by a colon.

It is assigned the value of the assembly location counter (see Section 5.5.4), which may be either absolute (fixed in memory independently of the position of the program) or relocatable (not fixed in memory). For example, if the current assembly location is absolute 100(octal), the statement:

```
ABCD:   MOV A,B
```

will assign the value 100 to the label ABCD; subsequent reference to ABCD will be to location 100.

In the above case if the assembly location counter were relocatable, then the final value of ABCD would be 100(octal) plus a value assigned by the Linker when it relocates the code, called the relocation factor. (The final value of ABCD would therefore not be known until link-time. This is explained in Sections 5.6 and 5.8.3 of this chapter, and in Chapter 6).

More than one label may appear within a label field in which case each label within the field will have the same value. For example, if the current location counter is 100(octal), the statement:

```
ABC:    $DD:    A7.7:   MOV A,B
```

will assign each of the three labels ABC, $DD, and A7.7 the value 100 (the characters $ and . designate that these labels are used in system software).

A label may be composed of more than six characters, but only the first six are recognized by the Assembler. An error code will be generated during assembly if two or more labels have the same first six characters.

5.3.2 Operators

An operator follows the label field in a statement and may be an instruction mnemonic or an assembler directive (the instruction set is discussed in the PDP-11 PROCESSOR HANDBOOK; Section 5.8 of this chapter provides information concerning assembler directives). When the operator is an instruction mnemonic, it specifies an action to be performed on any operand(s) which follows it. When it is an assembler directive, the operator specifies a certain function or action to be performed during the assembly process.

An operator may be preceded only by labels and may be followed by one or more operands and/or a comment. An operator is legally terminated by any of the following characters:

```
#  +  -  @  (  "  '  %  !  &  ,  ;

line feed  form feed  carriage return  space  tab
```

(The use of each of these characters will be explained later in the chapter.) For example:

```
JMP     BEGIN   ;(TAB) TERMINATES OPERATOR JMP

MOV@A,B         ;@ TERMINATES OPERATOR MOV
```

When an operator is not followed by an operand or a comment, it is terminated by a carriage return followed by either a line feed or form feed character.

5.3.3 Operands

An operand is that part of the statement which is acted upon by the operator and may be a symbol, expression, or number. Multiple operands are separated from one another by a comma. For example:

```
LABEL:  MOV R0,R1       ;THIS IS A COMMENT
```

The space between MOV and R0 terminates the operator field (MOV) and begins the operand field; the comma separates the operands R0 and R1. When the operand field is not followed by a comment, it is terminated by a carriage return followed by a line feed or form feed character. An operand is separated from a comment by a semi-colon.

5.3.4 Comments

The comment field is optional and may contain any ASCII character except null or rubout; all other characters are ignored by the Assembler when used in the comment field.

The comment field may be preceded by any or all of the other three fields, or it may be on a line by itself. It must begin with a semicolon and end with a carriage return followed by a line feed or form feed character. For example:

```
LABEL:  CLR HERE        ;THIS IS A COMMENT
```

Comments do not affect assembly processing or program execution, but are useful in program listings for later analysis, checkout or documentation purposes.

5.3.5 Format Control

The format of an assembly listing is controlled by the space and tab characters. These characters have no effect on the assembly process of the source program unless they are embedded within a symbol, number, or ASCII text, or unless they are used as the operator field terminator. They are generally used in the source program to provide a neat readable listing. For example, a statement can be written:

```
LABEL:MOV(SP)+,TAG;POP VALUE OFF STACK
```

This statement is correct and will assemble properly. However, using the format control characters it can also be written:

```
LABEL:  MOV (SP)+,TAG   ;POP VALUE OFF STACK
```

which is much easier to read.

Page size is controlled by the form feed character (CTRL/L). A page of n lines is created by inserting a form feed after the nth line. If no form feed is present, the Assembler automatically terminates a page after 56 lines of text.

5.4 SYMBOLS

A symbol is a string of alphanumeric characters and may be any length. However, the Assembler only recognizes the first six characters; thus symbols which contain the same first six characters are considered identical. There are two types of symbols, permanent and user-defined.

5.4.1 Permanent Symbols

The Assembler contains a table (called its permanent symbol table) which lists the symbols for all instruction mnemonics and assembler directives. The value of a permanent symbol is unique and independent of the program's position in memory. That is, its value is fixed and need not be redefined by the programmer. Appendix B provides a list of all permanent symbols in the CAPS-11 Assembler.

5.4.2 User-Defined Symbols

All symbols not already defined in the Assembler (and therefore represented in its permanent symbol table) must be defined by the programmer within the source program. These user-defined symbols are those either designated as labels or created by direct assignment (as explained in the next section). User-defined symbols are added to the permanent symbol table as they are encountered during the first pass of the assembly; they may be composed of alphanumeric characters, dollar signs, and periods only (again $'s and .'s are usually reserved for system software). Any other characters are illegal and, if used, will result in an error message. The following rules also apply to user-defined symbols:

1. The first character must not be a number.

2. Each symbol must be unique within the first six characters. A symbol may be written with more than six characters but the seventh and subsequent characters are only checked for legality and are not otherwise recognized by the Assembler.

3. Spaces and tabs must not be imbedded within a symbol.

A user-defined symbol may duplicate a permanent symbol; the value associated with it depends upon its use as follows:

1. A permanent symbol encountered in the operator field is always assigned its pre-defined value.

2. A permanent symbol encountered in the operand field is assigned its pre-defined value unless this value has been re-defined by the user; in that case, it is assigned the user-defined value.

User-defined symbols may be of two types--global or internal. Global symbols are used to provide links between object modules and are explicitly specified as global using a special assembler directive (see Section 5.8.2). A global symbol may be defined by the user (by either direct assignment or as a label), in which case it is called an entry symbol or entry point; such symbols may be referenced by other assemblies or object modules. A global symbol which is not defined in the current assembly is called an external symbol and must be defined (as an entry symbol) in another assembly.

All other user-defined symbols are termed internal; these symbols are referenced only from within the current assembly.

Under an 8K system, the Assembler provides space in its symbol table for approximately 240 user-defined symbols; a 12K system has room for approximately 880 user-defined symbols, and a 16K (or greater) system allows more than 2000 user-defined symbols.

5.4.3 Directly Assigning Values to Symbols

A direct assignment statement assigns a value to a symbol. The newly-defined symbol is then added to the Assembler's permanent symbol table; no word is reserved at the address where the definition occurs. The format of the statement is:

SYMBOL=EXPRESSION

where the expression is another symbol, numeric value, operator, or other expression as defined in Section 5.5.

The following conventions apply:

1. An equal sign (=) must separate the symbol from the expression defining the symbol.

2. A direct assignment statement may be preceded by a label and may be followed by a comment.

3. Only one symbol may be defined by any one direct assignment statement.

Examples of direct assignment statements follow:

```
A=1             ;THE SYMBOL A IS EQUATED
                ;WITH THE VALUE 1
```

```
B='A-1&MASKLOW  ;THE SYMBOL B IS EQUATED WITH THE
                ;VALUE OF THE EXPRESSION ('A-1&MASKLOW)

C:      D=3             ;THE SYMBOL D IS EQUATED WITH
E:      MOV #1,ABLE     ;THE VALUE 3. (SINCE NO WORD IS
                        ;RESERVED, LABELS C AND E ARE
                        ;BOTH EQUATED WITH THE NUMERICAL
                        ;MEMORY ADDRESS OF THE MOV COMMAND)
```

A symbol may be redefined by assigning it a new value; the new value will replace the old value in the permanent symbol table.

NOTE

If the defining expression is a global symbol, the defined symbol will not be global unless it has previously been defined as such (see Section 5.5).

Only one level of forward referencing is allowed in a direct assignment statement. That is, the following arrangement is illegal:

```
X=Y
Y=Z
Z=300
```

In a case such as this, X and Y will both be undefined throughout pass 1 of the assembly and will be listed as such at the end of that pass. Y will be defined during pass 2, but X will remain undefined throughout that pass and will generate an error message following the pass.

A symbol is relocatable or absolute depending upon the mode of the defining expression. Section 5.5.5 explains how to determine the mode of an expression.

5.4.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7. The programmer may assign symbolic names to these registers and thereafter reference them as symbols.

A register symbol is defined by means of a direct assignment statement where the defining expression contains at least one term (that is, symbol or numeric value) preceded by a % sign, or at least one term (symbol or numeric value) previously defined as a register symbol. In addition, the defining expression of a register symbol must be absolute. For example:

```
R0=%0           ;DEFINE R0 AS REGISTER 0

R3=R0+3         ;DEFINE R3 AS REGISTER 0 + 3
```

```
R4=1+%3          ;DEFINE R4 AS REGISTER 3 + 1

THERE=%2         ;DEFINE "THERE" AS REGISTER 2
```

It is important to note that all register symbols must be defined before they may be referenced. Any reference to an undefined register symbol will generally cause errors.

After a register symbol has been defined, any expression containing a % sign indicates a reference to a register; such an expression is called a register expression. Thus, the statement:

```
CLR %6
```

indicates that register 6 will be cleared, while:

```
CLR 6
```

will clear the word at memory address 6.

In certain cases a register can be referenced without the use of a register symbol or register expression. These cases are recognized through the context of the statement and are explained in Sections 5.7.13 and 5.7.14.

5.5 EXPRESSIONS

Expressions are formed by the combination of terms. Terms may be symbols, numbers, ASCII data, or the present value of the assembly location counter (as represented by the special character, period) and are joined to one another by logical or arithmetic operators. A single term may form an expression, or several terms may be combined by operators to make up the expression. (Symbols have already been explained; the remaining terms are covered in this section.)

Expressions are evaluated by the Assembler from left to right and are assigned word locations; parenthetical grouping is not allowed. The evaluation of an expression includes the evaluation of the mode of the resultant expression (i.e., absolute, relocatable, or external; see Section 5.5.5.)

In evaluating expressions, the Assembler will interpret the following illegal conditions as indicated:

1. A missing term, expression or external symbol will be interpreted as 0. For example:

   ```
   A+-100          ;OPERAND MISSING
   ```

   will be evaluated as A+0-100.

2. A missing operator will be interpreted as + . For example:

   ```
   TAG ! LA 177777  ;OPERATOR MISSING
   ```

   will be evaluated as TAG ! LA+177777; an error code will be printed.

3. The value of an external expression (one which contains a symbol not defined in the current program) will be the value of only the absolute part of the expression; e.g., EXT+A will have a value of A. (This is later modified by the Linker, after program relocation and linking is complete, to become EXT+A.)

5.5.1 Arithmetic and Logical Operators

An operator is a symbol which indicates an action (or operation) to be performed. Two arithmetic and two logical operators are used by the CAPS-11 Assembler. The arithmetic operators are:

+	indicates addition or a positive number
-	indicates subtraction or a negative number

The logical operators are:

&	indicates the logical AND operation
!	indicates the logical inclusive OR operation

The logical operators cause bit by bit comparisons (between two 16-bit words) to be performed with the following results:

AND

0	&	0	=	0
0	&	1	=	0
1	&	0	=	0
1	&	1	=	1

OR

0	!	0	=	0
0	!	1	=	1
1	!	0	=	1
1	!	1	=	1

5.5.2 Numbers

A number is any sequence of digits delimited by the termination characters discussed in Section 5.2. The Assembler accepts numbers indicated in both octal and decimal bases. Octal numbers consist of the digits 0 through 7 only; decimal numbers consist of the digits 0 through 9 followed by a decimal point. (If a number contains an 8 or 9 and is not followed by a decimal point, an error code will be printed and the number will be interpreted by the Assembler as decimal.) A number which is preceded by a minus sign is interpreted as a negative number (thus it is not necessary to express a negative number in its two's complement form); positive numbers may be preceded by a plus sign although this is not required.

If a number is too large to fit into 16 bits, the number is truncated from the left and an error code is printed in the assembly listing. Numbers and generated data are always considered as absolute quantities.

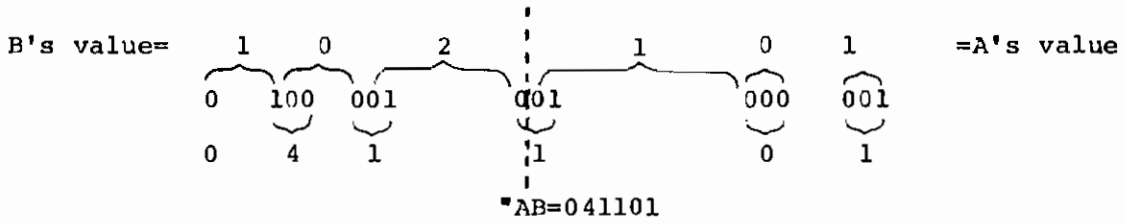
5.5.3 ASCII Conversion

When preceded by an apostrophe, any ASCII character (except null, rubout, carriage return, line feed, or form feed) is assigned its 7-bit ASCII value (see Appendix A for a chart containing ASCII codes). For example:

```
'A
```

assigns the ASCII character A the value 000101(octal).

When two ASCII characters are preceded by a quotation mark, (again the characters must not be null, rubout, carriage return, line feed, or form feed) they are both assigned their corresponding 7-bit ASCII values; each 7-bit value is stored in an 8-bit byte and the bytes are combined to form a word. For example, "AB will store the ASCII value of A in the low-order (even) byte and the value of B in the high-order (odd) byte, as follows:



ASCII text is always considered absolute by the Assembler.

5.5.4 Assembly Location Counter

As assembly proceeds, consecutive memory locations are assigned to each byte of object data generated. Thus, each word of object data is normally assigned even consecutive locations.

The special character period (.) is the symbol for the assembly location counter; when used in the operand field following an instruction, a period represents the address of the first word of the instruction.

NOTE

The assembly location counter is not the same as the Program Counter as described in Section 5.7.

For example, assume the following statement occurs at location 502:

```
A:      MOV #.,R0
```

The period refers to location 502, that is, the address of the MOV instruction. When used in the operand field following an assembler directive (see Section 5.8), the period represents the address of the current byte or word. Assume the following statement occurs at location 450:

```
.BYTE 73,.,ADR
```

In this case, the period refers to location 451.

The Assembler clears the location counter at the beginning of each assembly pass. Information is then normally stored in consecutive memory locations beginning at location 0 for relocatable sections, and wherever the programmer indicates for absolute sections. The user may at any time change the location where the object data is to be stored by a direct assignment statement of the form:

```
.=EXPRESSION
```

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

In the following example the programmer uses .ASECT and .CSECT directives, which designate that code will be assigned either absolute or relocatable locations. These directives are explained in detail in Section 5.8.3.

```
.ASECT
        .=500           ;SET LOCATION COUNTER TO ABSOLUTE 500

FIRST:  MOV .+10,COUNT  ;THE LABEL FIRST HAS THE VALUE 500(8)
                        ;.+10 EQUALS 510(8), THE CONTENTS OF
                        ;LOCATION 510(8) WILL BE DEPOSITED IN
                        ;LOCATION COUNT.
        .=520           ;THE ASSEMBLY LOCATION COUNTER NOW
                        ;HAS A VALUE OF ABSOLUTE 520(8).
SECOND: MOV .,INDEX     ;THE LABEL SECOND HAS THE VALUE 520(8).
                        ;THE CONTENTS OF LOCATION 520(8), THAT
                        ;IS, THE BINARY CODE FOR THE INSTRUCTION
                        ;ITSELF, WILL BE DEPOSITED IN LOCATION
                        ;INDEX


.CSECT
        .=.+20          ;SET LOCATION COUNTER TO RELOCATABLE 20.

THIRD:  .WORD 0         ;THE LABEL THIRD HAS THE VALUE OF
                        ;RELOCATABLE 20 (DETERMINED BY THE
                        ;LINKER).
```

Storage area may be reserved by advancing the location counter. For example, if the current value of the location counter is 1000, the direct assignment statement:

```
.=.+100
```

will reserve 100(8) bytes of storage space in the program. The next instruction will be stored at 1100.

Similar to other symbols, the assembly location counter has a mode associated with it. The mode is determined by the mode of the section in which it appears (absolute or relocatable). This mode cannot be changed by using a defining expression of a different mode. However, it may be changed by changing the mode of the section using either the .ASECT or .CSECT directives as explained in Section 5.8.3. The mode cannot at any time be external.

5.5.5 Modes of Expressions

As already mentioned, expressions consist of a term or the combination of terms (terms being any symbol, number, ASCII data, or the value of the current location counter). Just as each term of the expression can be assigned a mode (absolute, relocatable, or external), the mode of the expression itself may be determined as follows:

An absolute expression is defined as:

1. An absolute term preceded optionally by a single arithmetic operator, or
2. A relocatable expression minus a relocatable term, or
3. An absolute expression followed by an operator followed by an absolute expression.

A relocatable expression is defined as:

1. A relocatable term, or
2. A relocatable expression followed by an arithmetic operator followed by an absolute expression, or
3. An absolute expression followed by a plus operator followed by a relocatable expression.

An external expression is defined as:

1. An external term, or
2. An external expression followed by an arithmetic operator followed by an absolute term, or
3. An absolute expression followed by a plus operator followed by an external expression.

In the following examples ABS represents an absolute term, REL represents a relocatable term, and EXT represents an external term. Thus, these are valid expressions:

```
EXT+ABS             ;EXTERNAL EXPRESSION

REL+REL-REL         ;RELOCATABLE EXPRESSION

ABS+REL-REL&ABS     ;ABSOLUTE EXPRESSION
```

The following are illegal expressions (and cannot be handled properly by the Linker):

```
EXT+REL
REL+REL
ABS-EXT
```

5.6 RELOCATION AND LINKING

The output of the Assembler is a relocatable object module which must be processed by the Linker before it can be loaded and executed. The object module contains the assembled binary output of absolute, relocatable, and external expressions. Since absolute expressions are fixed in memory, the Linker does no manipulation. However, the values of external or relocatable expressions must be fixed (or made absolute) by the Linker before it can create the load module which will contain the binary data to actually be loaded and executed.

To enable the Linker to fix the value of an expression, the Assembler must pass certain information concerning the expression on to the Linker. For example, each relocatable section of code in the source program has been assembled sequentially with the first section beginning at location 0 (called relocatable 0); thus each relocatable expression is a relative number of locations from 0. (This value--relocatable n--and other information is passed to the Linker by means of the Global Symbol Directory and the Relocation Directory, as described in Section 5.14.) When the Linker relocates the section of code, it adds the relocatable value of the expression as provided by the Assembler to the base (or beginning location of the section after relocation) thereby producing an absolute value for the expression.

In the case of an external expression, the value of the external symbol in the expression is determined by the Linker (since the external symbol must be defined in one of the other object modules being linked) and this value is then added to the value of the external expression provided by the Assembler (see Section 5.5, #3).

All instructions that are to be modified by the Linker in this manner will be marked by a single apostrophe in the assembly listing, as illustrated in the following examples:

```
005065  CLR EXTERNAL(5)          ;VALUE OF EXTERNAL SYMBOL
000000'                          ;IS ASSUMED TO BE ZERO, WILL
                                 ;BE MODIFIED BY THE LINKER

005065  CLR EXTERNAL+6(5)        ;VALUE OF EXTERNAL SYMBOL
000006'                          ;(ASSUMED ZERO) + 6 WILL BE
                                 ;MODIFIED BY THE LINKER


005065  CLR RELOCATABLE(5)       ;ASSUME CODE IS IN AN
000040'                          ;ABSOLUTE SECTION AND
                                 ;VALUE OF RELOCATABLE SYMBOL
                                 ;IS RELOCATABLE 40
```

5.7 ADDRESSING MODES

The eight general registers may be used for storing and manipulating data. Accessing these registers is done by means of register addressing modes. In order to understand how the addressing modes operate and how they are assembled, the action of the Program Counter must be understood. The Program Counter (register 7 of the eight general registers) always contains the address of the next word to be fetched; i.e., either the address of the next instruction to be executed, or the address of the second or third word of the current instruction. The key rule is:

> Whenever the processor implicitly uses the Program Counter (PC) to fetch a word from memory, the Program Counter is automatically incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two so that it is pointing to the next word in memory.

The following conventions are used in explaining the addressing modes:

1. E represents any expression as defined in Section 5.5.

2. R represents a register expression. This is any expression containing a term preceded by a % character or a symbol previously equated to such a term, as explained in Section 5.4.4.

3. ER represents: a) a register expression as explained in 2 above, or b) an expression in the range 0 to 7 inclusive.

4. A represents a general address specification which produces a 6-bit mode address field (source or destination address) as described in the PDP-11 PROCESSOR HANDBOOK under the sections entitled Single Operand Addressing and Double Operand Addressing.

Addressing modes for general registers 0-6 will be described first and then addressing using the Program Counter (register 7). The format for the addressing specification, A, is explained in terms of E, R, and ER as defined above. Each will be illustrated with the single operand instruction CLR or double operand instruction MOV. (The user may also refer to the PDP-11 PROCESSOR HANDBOOK for information concerning addressing modes.)

5.7.1 Register Mode (Mode 0)

The register contains the operand.

Format: R

Example:

```
R0=%0           ;DEFINE R0 AS REGISTER 0
CLR R0          ;CLEAR REGISTER 0
```

5.7.2 Deferred Register Mode (Mode 1)

The register contains the address of the operand.

Format: @R or (ER)

Example:

```
        CLR @R1          ;CLEAR THE WORD AT THE
    or                   ;ADDRESS CONTAINED IN
        CLR (1)          ;REGISTER 1
```

5.7.3 Autoincrement Mode (Mode 2)

The contents of the register are incremented immediately after being used as the address of the operand.

Format: (ER)+

Examples:

```
CLR (R0)+        ;CLEAR WORDS AT ADDRESSES
CLR (R0+3)+      ;CONTAINED IN REGISTERS 0,3,
CLR (2)+         ;AND 2, AND INCREMENT THE
                 ;CONTENTS OF EACH OF THESE
                 ;REGISTERS BY TWO.
```

NOTE

Both JMP and JSR instructions using mode 2 increment the register before its use on the PDP-11/20 and 11/40 (but not on the PDP-11/05, 11/10, or 11/45).

In double operand instructions of the addressing form %R,(R)+ (or %R,-(R)) where the source and destination registers are the same, the source operand is evaluated as the autoincremented (or autodecremented) value, but the destination register, at the time it is used, still contains the originally intended effective address. In the following two examples, as executed on the PDP-11/20 and 11/40, R0 originally contains 100.

```
MOV R0,(0)+      ;THE QUANTITY 102 IS
                 ;MOVED TO LOCATION 100

MOV R0,-(0)      ;THE QUANTITY 76 IS
                 ;MOVED TO LOCATION 76
```

The PDP-11/05, 11/10, and 11/45 handle these instructions as follows:

```
MOV R0,(0)+      ;THE QUANTITY 100 IS
                 ;MOVED TO LOCATION 100

MOV R0,-(0)      ;THE QUANTITY 100 IS
                 ;MOVED TO LOCATION 76
```

The use of these forms should be avoided as they are not compatible among PDP-11 processors.

5.7.4 Deferred Autoincrement Mode (mode 3)

The register is used as a pointer to the address of the operand. The contents of the register are incremented after being used.

Format: @(ER)+

Example:

```
CLR @(3)+        ;CONTENTS OF REGISTER 3 POINT
                 ;TO ADDRESS OF WORD TO BE
                 ;CLEARED, CONTENTS OF REGISTER
                 ;3 ARE THEN INCREMENTED BY 2
```

5.7.5 Autodecrement Mode (Mode 4)

The contents of the register are decremented before being used as the address of the operand (see note in Section 5.7.3).

Format: -(ER)

Examples:

```
CLR -(R0)        ;DECREMENT CONTENTS OF REGISTERS
CLR -(R0+3)      ;0, 3 AND 2 BEFORE USING CONTENTS
CLR -(2)         ;AS ADDRESSES OF WORDS TO BE CLEARED
```

5.7.6 Deferred Autodecrement Mode (Mode 5)

The contents of the register are decremented before being used as a pointer to the address of the operand.

Format: @-(ER)

Example:

```
CLR @-(2)        ;DECREMENT CONTENTS OF REGISTER 2
                 ;BEFORE USING CONTENTS AS POINTER TO
                 ;ADDRESS OF WORD TO BE CLEARED
```

5.7.7 Index Mode (Mode 6)

The contents of the register (ER) and the value of the expression E are summed to form the address of the operand. The value of the expression E is stored as the second or third word of the instruction and is called the base. The processor uses the Program Counter to fetch the base from memory; the PC is then incremented by two and points to the next word.

Format: E(ER)

Examples:

```
CLR X+2(R1)     ;EFFECTIVE ADDRESS IS X+2 PLUS
                ;THE CONTENTS OF REGISTER 1

CLR -2(3)       ;EFFECTIVE ADDRESS IS -2 PLUS
                ;THE CONTENTS OF REGISTER 3
```

5.7.8 Deferred Index Mode (Mode 7)

The value produced when the expression and the contents of the register are added is a pointer to the address of the operand.

Format: @E(ER)

Example:

```
CLR @14(4)      ;IF REGISTER 4 CONTAINS 100, AND
                ;LOCATION 114 CONTAINS 2000, LOC.
                ;2000 IS CLEARED
```

ADDRESSING USING REGISTER 7 (PC)

Although Register 7 serves as the Program Counter, it may also be used as a general purpose register. The PC responds to all the standard PDP-11 addressing modes; however four of these modes are especially useful when writing Position Independent Code (explained in Section 5.9); these are summarized below.

5.7.9 Immediate Mode (mode 2)

Immediate mode allows the operand itself to be stored as the second or third word of the instruction. It is assembled as an autoincrement of register 7.

Format: #E

Examples:

```
MOV #100,R3     ;MOVE AN OCTAL 100 TO REGISTER 3
```

```
MOV #X,R0       ;MOVE THE VALUE OF SYMBOL X TO
                ;REGISTER 0
```

An explanation of this mode follows. Using the first example above, the statement MOV #100,R3 assembles as two words; these are:

```
012703
000100
```

Just before this instruction is fetched and executed, the PC points to the first word of the instruction (012703). The processor fetches this word and increments the PC by two. Since the source operand mode is 27 (autoincrement the PC), the PC is used as a pointer to fetch the operand (the second word of the MOV instruction, 000100). The PC is then incremented by two to point to the next instruction.

5.7.10 Absolute Mode (Mode 3)

In absolute (or deferred immediate) mode, the expression specifies an absolute address; the second word of the instruction contains the address of the operand. Absolute mode is assembled as a deferred autoincrement of register 7.

Format: @#E

Examples:

```
MOV @#240,R3    ;MOVE CONTENTS OF LOCATION
                ;240 TO REGISTER 3

CLR @#X         ;CLEAR THE CONTENTS OF THE
                ;LOCATION WHOSE ADDRESS IS X
```

5.7.11 Relative Mode (Mode 6)

Relative mode is assembled as index mode using register 7 and is the normal mode for memory references.

Format: E

Examples:

```
CLR 100         ;CLEAR LOCATION 100

MOV X,Y         ;MOVE CONTENTS OF LOCATION
                ;X TO LOCATION Y
```

The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand. Rather, it is the number which, when added to the PC, becomes the address of the operand. Thus, the base is this address - PC, and is called an offset. The operation is explained as follows:

If the statement MOV 100,R3 is assembled at absolute location 20, then the assembled code is:

```
Location 20        016703
Location 22        000054
```

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67 (indexed by the PC). To pick up the base, the processor fetches the word pointed to by the PC (location 22); the PC is then incremented by two and points to location 24. To calculate the address of the source operand, the base is added to the updated PC. Thus, base+PC=54+24=100, the operand address.

Since the Assembler considers the assembly location counter (.) as the address of the first word of the instruction, an equivalent index mode statement would be:

```
MOV 100-.-4(PC),R3
```

This mode is called relative because the the operand address is calculated relative to the current PC. The base is the distance or offset (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly anywhere in memory.

5.7.12 Deferred Relative Mode (Mode 7)

Deferred relative mode is indicated when the expression is preceded by @; the expression's value is the pointer to the address of the operand.

Format: @E

Examples:

```
CLR @A1          ;ADD SECOND WORD OF INSTRUCTION
                 ;TO THE PC TO OBTAIN A POINTER TO
                 ;THE ADDRESS OF THE OPERAND,
                 ;CLEAR OPERAND

MOV @X,R0        ;MOVE THE CONTENTS OF THE
                 ;LOCATION WHOSE ADDRESS IS IN X
                 ;INTO REGISTER 0
```

5.7.13 Table of Mode Forms and Codes

Table 5-2 summarizes the addressing modes. Each instruction assembles as at least one word. Operands of the first six forms listed below do not increase the length of an instruction. Each operand in one of the other modes, however, increases the instruction length by one word (n represents the register).

Table 5-2
Mode Forms and Codes

Form	Mode	Meaning
(Instruction length is not increased)		
R	0n	Register
@R or (ER)	1n	Register deferred
(ER)+	2n	Autoincrement
@(ER)+	3n	Autoincrement deferred
-(ER)	4n	Autodecrement
@-(ER)	5n	Autodecrement deferred
(Instruction length increased by one word)		
E(ER)	6n	Index
@E(ER)	7n	Index deferred
#E	27	Immediate
@#E	37	Absolute memory reference
E	67	Relative
@E	77	Relative deferred reference

NOTE

An alternate form for @R is (ER). However, the form @(ER) is equivalent to @0(ER).

The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the instruction CLR @#100 will clear absolute location 100 even if the instruction is moved from the point at which it was assembled.

The Assembler is not particular about left and right and dangling + and - signs in address fields. The following are some examples of incorrect user syntax that will assemble as shown without any error indication, (X and Y are 16-bit address offsets):

Form	Assembles As:
(R2)X	X(R2)
X-(R2)	X(R2) or X-0(R2)
X(R2)+	X(R2)
+(R2)	(R2)+
(R2)-	-(R2)
@(R2)X	@X(R2)
X(R2)+Y	X+Y(R2)

5.7.14 Instruction Forms

Instruction mnemonics are detailed in the PDP-11 PROCESSOR HANDBOOK and summarized in Appendix B. This section defines the number and nature of the operand fields for these instructions. In the table that follows, let R, E, and ER represent expressions as defined in Sections 5.5 and 5.7; let OPR represent the operator; let A be a 6-bit address specification in one of these forms:

E	@E	-(ER)	@-(ER)
R	@R or (R)	E(ER)	@E(ER)
(ER)+	@(ER)+	#E	@#E

Table 5-3
Instruction Operand Fields

Instruction	Form	Example
Double Operand	OPR A,A	MOV (R6)+,@Y
Single Operand	OPR A	CLR -(R2)
Operate	OPR	HALT
Branch	OPR E	BR X+2
	where -128<(E-.-2)/2<=127	BLO .-4
Subroutine Call	JSR ER,A	JSR PC,SUBR
Subroutine Return	RTS ER	RTS PC
EMT/TRAP	OPR or OPR E	EMT
	where 0<=E<=377(octal)	EMT 31

Branch instructions are one word instructions. The high byte contains the op code and the low byte contains an 8-bit signed offset (7 bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. The sign of the offset is extended through bits 8-15.
2. The result is multiplied by 2; this creates a word offset rather than a byte offset.
3. The result is added to the PC to form the final branch address.

The Assembler performs the reverse operation to form the byte offset from the specified address. When the offset is added to the PC, the PC is pointing to the word following the branch instruction, hence the factor -2 in the calculation.

Byte offset = (E-PC)/2 truncated to eight bits.

Since PC=.+2, the

Byte offset = (E-.-2)/2 truncated to eight bits.

NOTE

It is illegal to branch to a location specified as an external symbol, or to a relocatable symbol when within an absolute section, or to an absolute symbol when within a relocatable section.

The EMT and TRAP instructions do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big (>377(octal)) it is truncated to eight bits and an error code is printed.

The programmer should not try to micro-program the condition code operators (see Appendix B) as the CAPS-11 Assembler does not support this capability. Thus:

```
CLC!CLV
```

results in an error message and the statement is assembled as CLC.

However, expressions allow logical operators and the use of instruction mnemonics. Thus, the following words are correctly written:

```
.WORD CLC!      ;OPERAND OF .WORD DIRECTIVE
                (see Section 5.8.7)
+CLC!CLV        ;OPERAND OF DEFAULT .WORD
!CLC!CLV        ;OPERAND OF DEFAULT .WORD
```

5.8 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops) direct the assembly process and may generate data. Directives may be preceded by a label and may be followed by a comment. The assembler directive occupies the operator field and only one directive may be placed in any one statement. A directive and its operand should be separated by a space or other legal terminator. Operands which are used with directives vary and are discussed individually.

5.8.1 .TITLE

The .TITLE directive is used to name the object module. The name is assigned by the first symbol following the directive. If there is no .TITLE statement the default name assigned is ".MAIN.". Thus:

```
        .TITLE
FILE1:  MOV #NAME,R0
```

assigns the name FILE1 to the current object module.

5.8.2 .GLOBL

The .GLOBL directive is used to declare a symbol as being global. A global symbol is generally referenced by more than one object module. It may be an entry symbol, in which case it is defined in the current program, or it may be an external symbol, in which case it is defined in another program which will be linked with the current program by the Linker. The form of the .GLOBL directive is:

```
.GLOBL  NAMA,NAMB,...,NAMN
```

where symbols NAMA,NAMB,...NAMN are all defined as global symbols.

NOTE

A symbol cannot be declared global by defining it as a global expression in a direct assignment statement.

If an illegal character is detected in the operand field of a .GLOBL statement an error message is not generated but the Assembler may ignore the remainder of the statement. Thus:

```
.GLOBL A,B,@C,D
```

assembles without error as:

```
.GLOBL A,B
```

5.8.3 Program Section Directives (.ASECT and .CSECT)

The relocatable Assembler provides two directives enabling the programmer to specify that parts of his program be assembled in absolute sections and other parts in relocatable sections. The scope of each directive extends until a directive to the contrary is given. The Assembler initially starts in a relocatable section; to enter an absolute section, the .ASECT directive is indicated. Thus, if the first statement of a program is:

```
A:      .ASECT
```

the label "A" would be a relocatable symbol which is assigned the value of relocatable zero. The Linker will later calculate the absolute value of A by adding the value of the base of the relocatable section. For example:

```
        .ASECT          ;ASSEMBLER IN ABSOLUTE SECTION
        .=1000          ;PC=1000 ABSOLUTE
A:      CLR X           ;A=1000 ABSOLUTE
        .CSECT          ;ASSEMBLE IN RELOCATABLE SECTION
X:      JMP A           ;X=0 RELOCATABLE
        .END
```

The programmer may alternate between relocatable and absolute sections as follows:

```
.CSECT
.WORD 0,1,2      ;ASSEMBLED AT RELOCATABLE 0, 2, AND 4
.ASECT
.WORD 0,1,2,     ;ASSEMBLED AT ABSOLUTE 0, 2, AND 4
.CSECT
.WORD 0          ;ASSEMBLED AT RELOCATABLE 6
.END
```

If a label is defined twice, first in an absolute section and then in a relocatable section, the symbol will be relocatable but its value will be as defined in the absolute section.

Chapter 6 provides details concerning how the Linker handles absolute and relocatable program sections at link-time.

5.8.4 .EOT

NOTE

> The CAPS-11 Assembler provides the .EOT directive for the user who may wish to write a program for execution under another system allowing the use of paper tape. For that reason, it is described here, although the average CAPS-11 user will have no need to reference it and the CAPS-11 Assembler will ignore it. The following discussion of the .EOT directive details its use as it pertains to the Papertape Software System.

The .EOT directive indicates the physical End Of Tape though not the logical end of the program. If the .EOT is followed by a single line feed or form feed, the Assembler will still read to the end of the tape, but will not process anything past the .EOT directive. If .EOT is followed by at least two line feeds or form feeds, the Assembler will stop before the end of the tape. Either case is proper; even though it may appear as though the Assembler has read too far, it actually has not.

If .EOT is embedded in a tape, and more information to be assembled follows it, .EOT must be immediately followed by at least two line feeds or form feeds. Otherwise, the first line following the .EOT will be lost.

Any operands following a .EOT directive will be ignored. The .EOT directive allows several physically separate tapes to be assembled as one program. The last tape should be terminated by a .END directive (see Section 5.8.6) but may be terminated with .EOT.

5.8.5 .EVEN

The .EVEN directive ensures that the assembly location counter is even by adding one if it is odd. Any operands following a .EVEN directive will be ignored.

5.8.6 .END

The .END directive indicates the logical and physical end of the source program. The .END directive may be followed by only one operand--an expression indicating the program's transfer address. At load time, the load module will be loaded and program execution will begin at the transfer address indicated by the .END directive. If the address is not specified and a RUN or LOAD/G command is used, a fatal error message will be printed; if a LOAD/O command is used, CABLDR will halt and expect user console action (see Appendix E); a LOAD command in conjuction with the START command allows the user to indicate an optional starting address for the program.

If there is no .END directive in the user's program, the Assembler will issue the message:

```
?NO END STMT
```

at the end of the last input file and will continue as if there had been an .END statement there.

5.8.7 .WORD

The .WORD assembler directive may be followed by a space and one or more operands separated by commas and instructs the Assembler to store each operand in successive words of the object program. the operands may be any legally formed expression. For example:

```
.=1420
SAL=0
.WORD 177535,.+4,SAL     ;STORED IN WORDS 1420, 1422
                         ;AND 1424 WILL BE 177535,
                         ;1426 AND 0
```

Values exceeding 16 bits will be truncated from the left to word length.

A .WORD directive followed by one or more void operands separated by commas will store zeros for these operands. For example:

```
.=1430           ;ZERO, FIVE, AND ZERO ARE STORED
.WORD ,5,        ;IN WORDS 1430, 1432, AND 1434
```

If a statement contains no operator, this field will be interpreted as a .WORD directive providing the operand field contains one or more expressions. The first term of the first expression in the operand field must not be an instruction mnemonic or assembler directive unless preceded by a + or -. or one of the logical operators, ! or &. For example:

```
.=440                    ;THE OP-CODE FOR MOV (010000)
LABEL:  +MOV,LABEL       ;IS STORED IN LOCATION 440.
                         ;440 IS STORED IN LOCATION 442.
```

Note that the default .WORD directive will occur whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered in the operator or operand field which is not recognized as an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR:

```
MOR A,B
```

This will result in two errors caused by: a) an expression operator missing between MOR and A, and b) MOR being undefined. Two words will be generated; one for MOR A and one for B.

5.8.8 .BYTE

The .BYTE directive may be followed by a space and one or more operands separated by commas and instructs the Assembler to store each operand in a byte of the object program. If multiple operands are specified, they are stored in successive bytes. The operands may be any legally formed expression with a result of 8 bits or less. For example:

```
SAM=5              ;STORED IN LOCATION 410 WILL BE
.=410              ;060 (THE OCTAL EQUIVALENT OF 48).
.BYTE 48.,SAM      ;IN 411 WILL BE 005
```

Since the expression is evaluated as a word expression, if it is found to have a result of more than 8 bits, it will be truncated to its low-order 8 bits and an error will be flagged. If an operand after the .BYTE directive is left void, it will be interpreted as zero. For example:

```
.=420              ;ZERO WILL BE STORED IN
.BYTE , ,          ;BYTES 420, 421 AND 422.
```

If the expression is relocatable, a warning will be printed.

5.8.9 .ASCII

The .ASCII directive translates strings of ASCII characters (with the exception of null, rubout, carriage return, line feed, and form feed) into their 7-bit ASCII codes. The text to be translated must be enclosed by a delimiting character which may be any printing ASCII character except colon and equal sign and those characters used in the text string itself. The 7-bit ASCII code generated for each character will be stored in successive bytes of the object program. For example:

```
.=500              ;THE ASCII CODE FORY Y WILL BE
.ASCII /YES/       ;STORED IN 500, THE CODE FOR E
                   ;IN 501, THE CODE FOR S IN 502.
```

```
.ASCII /5+3/2/  ;THE DELIMITING CHARACTER OCCURS
                ;BETWEEN THE OPERANDS. THUS THE ASCII
                ;CODES FOR 5, +, AND 3 ARE STORED
                ;IN BYTES 503, 504, AND 505. 2/ IS
                ;NOT ASSEMBLED.
```

The .ASCII directive may be terminated by any legal terminator (as listed in Section 5.2 #3) except = and :. Note that if the text delimiter is also a legal terminator, it may serve a double function, terminating the directive and delimiting the text. For example:

```
.ASCII /ABCD/   ;THE SPACE IS REQUIRED
                ;BECAUSE / IS NOT A LEGAL
                ;TERMINATOR.

.ASCII+ABCD+    ;NO SPACE IS REQUIRED
                ;SINCE + IS A TERMINATOR.
```

5.8.10 .RAD50

PDP-11 system programs often handle symbols in a specially coded form called "RADIX 50" (sometimes also referred to as "MOD40"). This form allows 3 characters to be packed into 16 bits; therefore, any symbol can be held in two words. The form of the directive is:

```
.RAD50 /CCC/
```

The single operand is entered in the form /CCC/ where the delimiter (in this case, slash) can be any printable character except = and : and those characters used in the operand. Characters which may be converted are A through Z, 0 through 9, dollar ($), dot (.) and space ( ). If there are fewer than 3 characters they are left-justified and trailing spaces are assumed. Any characters following the enclosing delimiter are ignored and no error results. For example:

```
.RAD50 /ABC/    ;PACK ABC INTO ONE WORD
.RAD50 /AB/     ;PACK AB (SPACE) INTO ONE WORD
.RAD50 //       ;PACK 3 SPACES INTO ONE WORD
```

The packing algorithm is as follows:

1. Each character is first translated into its RADIX 50 equivalent as follows:

Character	RADIX 50 Equivalent (octal)
(SPACE)	0
A-Z	1-32
$	33
.	34
0-9	36-47

Note that another character can be defined for code 35.

2. The RADIX 50 equivalents for the three characters (C1, C2, C3) are then combined as follows:

```
RESULT=((C1*50)+C2)*50+C3
```

and the result is stored in the word. For example, the RADIX 50 value of ABC is 3223.

5.8.11 .LIMIT

The .LIMIT directive generates two words into which the Linker puts the low and high addresses of the relocated code. The low address (stored in the first word) is the address of the first byte of relocated code; the high address is the address of the first free byte following the relocated code. These addresses will always be even since all relocatable sections are loaded at even addresses; if a relocatable section consists of an odd number of bytes the Linker adds one to the size to make it even.

5.8.12 Listing Control Directives (.LIST and .NLIST)

The .LIST and .NLIST directives allow the user to choose which sections of his program will appear in the assembly listing. The .NLIST directive suppresses the assembly listing and the .LIST directive reinitiates it. Thus if the user is developing a program and has a large section of code which does not change from one edit to the next, he can insert a .NLIST directive at the beginning of that code and a .LIST at the end. That code will not appear in the assembly listing.

If the .NLIST directive is in control when the symbol table is ready to be output, the .NLIST directive will be terminated so that the symbol table can be listed.

5.8.13 Conditional Assembly Directives

Conditional assembly directives provide the programmer with the capability of conditionally including or not including portions of his source code in the assembly process. In the explanation which follows, E denotes an expression. The conditional directives are:

Directive	Expression	Result
.IFZ	E	Assemble if E=0
.IFNZ	E	Assemble if E≠0
.IFL	E	Assemble if E<0
.IFLE	E	Assemble if E<=0
.IFG	E	Assemble if E>0
.IFGE	E	Assemble if E=>0

If the condition is met, all statements following the conditional directive are assembled until a special delimiting directive, .ENDC, is encountered. If the condition of the directive is not met, these

statements are ignored. When the .ENDC directive is detected, assembly continues as usual.

Two more conditional directives are used; these take the following form:

```
.IFDF   S(1) [!,&] S(2) [!,&],...[!,&]S(N)
.IFNDF  S(1) [!,&] S(2) [!,&],...[!,&]S(N)
```

where S(1) though S(N) represent symbols, ! represents the logical OR operation, and & represents the logical AND operation. .IFDF and .IFNDF mean "if defined" and "if undefined" respectively. The scan is from left to right, no parentheses permitted. Nesting is permitted up to a depth of 127(decimal). For example:

`.IFDF S!T&U`	Assemble the following code (until detection of .ENDC) if either S or T is defined and U is defined
`.IFNDF T&U!S`	Assemble the following code (until detection of .ENDC) if both T and U are undefined or if S is undefined

General remarks concerning conditional directives include the following:

1. A null expression or an expression in error use the default value 0 for purposes of the conditional test.

2. An error in syntax, e.g., a terminator other than ; ! & or CR results in the undefined situation for .IFDF and .IFNDF, as does a null symbol or symbol in error.

3. All conditionals must end with the .ENDC directive. Anything in the operand field of .ENDC is ignored.

4. Labels are permitted in statements containing conditional directives; however, since the scan is purely from left to right, in the following example:

```
        .IFZ 1
        CLR X
A:      .ENDC
```

   the label A will be ignored (as the Assembler ignores all code between the conditional directive and the .ENDC directive), while in this example:

```
A:      .IFZ 1
        CLR X
        .ENDC
```

   A is entered in the symbol table.

5. If an .END directive is encountered while inside a satisfied conditional, an error will be flagged; however, the .END directive will still be processed normally.

6. If more than one .ENDC directive is encountered (per conditional directive), errors are flagged on those in excess.

5.9 WRITING POSITION INDEPENDENT CODE (PIC)

When a standard program is available for use by other programs, it is often beneficial to be able to load and execute the program in different areas of memory. There are several ways to do this:

1. Reassemble the program at the desired location.

2. Use a relocating loader which accepts specially coded binary object modules from the Assembler.

3. Have the program relocate itself after it is loaded.

4. Write code which is position independent.

On small machines, reassembly is often performed; however, the CAPS-11 System has a relocating loader (Linker; see Chapter 6), and this is preferable. Since it generally is not economical to have a program relocate itself (as hundreds or thousands of addresses may need adjustment), writing position independent code is another method of producing a relocatable program.

PIC is achieved on the PDP-11 by correct usage of those addressing modes which form an effective memory address relative to the Program Counter (PC). Thus, if an instruction and its object(s) are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. PIC usually references locations relative to the current location, although absolute references may be made as long as the locations referenced remain stationary while the PIC is relocated. For example, references to interrupt and trap vectors are absolute, as are references to device registers in the "external page" (28K to 32K), and direct references to the general registers.

5.9.1 Position Independent Modes

There are three position independent modes, or forms of instructions:

1. Branches--the conditional branches, as well as the unconditional branch, BR, are position independent since the branch address is computed as an offset to the PC (see Section 5.7.11).

2. Relative Memory References--any relative memory reference of the form:

```
CLR X
MOV X,Y
JMP X
```

is position independent because the Assembler assembles the reference as an offset indexed by the PC. The offset is the difference between the referenced location and the PC. For example, assume the instruction CLR 200 is at address 100:

```
100  005067              ;FIRST WORD OF CLR 200
102  000074              ;OFFSET = 200-104
```

The offset is added to the updated PC. (The updated PC has been incremented by two and contains 104, i.e., the address of the word following the offset).

Although the form CLR X is position independent, the form CLR @X is not. Consider the following:

```
S:      CLR @X    ;CLEAR LOCATION A
          .
          .
X:      .WORD A   ;POINTER TO A
          .
          .
A:      .WORD 0
```

The contents of location X are used as the address of the operand in the location labeled A. Thus, if all of the code is relocated, the contents of location X must be altered to reflect the new address of A. However, if A was the name associated with some fixed location (e.g., a trap vector or a device register), then statements S and X would be relocated and A would remain fixed. The following code is position independent:

```
        A=36      ;ADDRESS OF SECOND WORD
                  ;OF TRAP VECTOR
S:      CLR @X    ;CLEAR LOCATION A
          .
          .
X:      .WORD A   ;POINTER TO A
```

3. Immediate Operands--The Assembler addressing form #E specifies immediate data, that is, the operand is part of the instruction (see Section 5.7.9). Thus, immediate data is position independent and is moved with the instruction. Immediate data is fetched using the PC in the autoincrement mode.

   As with direct memory references, the addressing form @#E is not position independent since the final effective address is absolute and points to a fixed location not relative to the PC.

5.9.2 Absolute Modes

Any time a memory location or register is used as a pointer to data, the reference is absolute. If the referenced data is fixed in memory

independent of the position of the PIC (e.g., trap-interrupt vectors or device registers), absolute modes must be used. (When the programmer is not writing position independent code, references to fixed locations may be performed using either the absolute or relative form.) If the data referenced is relative to the PIC, absolute modes must not be used unless the pointers involved are modified. The absolute modes are:

@E	Location E is a pointer
@#E	The immediate word is a pointer
(R)	The register is a pointer
(R)+ and -(R)	The register is a pointer
@(R)+ and @-(R)	The register points to a pointer
E(R) R=6 or 7	The base, E, modified by (R) is the address of the operand
@E(R)	The base, modified by (R), is a pointer

The non-deferred index modes and stack operations require a little clarification. As described in Sections 5.7.11 and 5.9.1, the form E(7) is the normal mode to reference memory and is a relative mode. Index mode, using a stack pointer (SP or other register) is also a relative mode and may be used conveniently in PIC. Basically, the stack pointer points to a dynamic storage area and index mode is used to access data relative to the pointer. The stack pointer may be initially set up by a position independent program as shown in Section 5.9.4. Once the pointer is set up, all data on the stack is referenced relative to the pointer. It should also be noted that since the form 0(SP) is considered a relative mode, so is its equivalent @SP. In addition, the forms (SP)+ and -(SP) are required for stack pops and pushes.

5.9.3 Writing Automatic PIC

Automatic PIC is code which requires no alteration of addresses or pointers. Thus, memory references are limited to relative modes unless the location referenced is fixed (trap and interrupt vectors, etc.). In addition to requirements already mentioned, the following must be observed:

1. Start the program with .=0 to allow easy relocation using CABLDR (see Appendix E).

2. All location setting statements must be of the form .=.+X or .= function of tags within the PIC; for example, .=A+10 where A is a local label.

3. There must not be any absolute location setting statements. This means that a block of PIC cannot set up trap and/or interrupt vectors at load time with statements such as:

```
.=34
.WORD TRAPH,340   ;TRAP VECTOR
```

CABLDR, when it is relocating PIC, relocates all data by the load bias (see Appendix E). Thus, the data for this vector would be relocated to some other location. Vectors may be set at execution time (as discussed next).

5.9.4 Writing Non-Automatic PIC

Often it is not possible or economical to write totally automated PIC; in these cases, some relocation may be easily performed at execution time. Some of the methods of solution are presented below. Basically, the methods operate by examining the PC to determine where the PIC is actually located; a relocation factor can then be easily computed. In all examples, it is assumed that the code is assembled at zero and has been loaded somewhere else by CABLDR.

Setting up the Stack Pointer - Often the first task of a program is to set the stack pointer (SP). This may be done as follows:

```
        .=0             ;BEG IS THE FIRST INSTRUCTION
                        ;OF THE PROGRAM.
BEG:    MOV PC,SP       ;SP=ADDRESS BEG+2
        TST -(SP)       ;DECREMENT SP BY 2.
                        ;A PUSH ONTO THE STACK WILL STORE
                        ;THE DATA AT BEG-2.
```

Setting up a Trap or Interrupt Vector - Assume the first word of the vector will point to location INT which is in PIC.

```
X:      MOV PC,R0           ;R0=ADDRESS X+2
        ADD #INT-X-2,R0     ;ADD OFFSET
        MOV R0,@#VECT       ;MOVE POINTER TO VECTOR
```

The offset INT-X-2 is equivalent to INT-(X+2); X+2 is the value of the PC moved by statement X. If PC(0) is the PC that was assumed for the program when loaded at 0 and if PC(n) is the current real PC, then the calculation is:

$$INT-PC(0)+PC(n)=INT+(PC(n)-PC(0))$$

Thus, the relocation factor, PC(n)-PC(0), is added to the assembled value of INT to produce the relocated value of INT.

Relocating Pointers - If pointers must be used, they may be relocated as shown above. For example, assume a list of data is to be accessed with the instruction:

```
ADD (R0)+,R1
```

The pointer to the list, list L, may be calculated at execution time as follows:

```
M:      MOV PC,R0       ;GET CURRENT PC
        ADD #L-M-2,R0   ;ADD OFFSET
```

Another variation is to gather all pointers into a table. The relocation factor may be calculated once and then applied to all pointers in the table using a loop:

```
X:      MOV PC,R0       ;RELOCATE ALL ENTRIES IN PTRTBL
        SUB #X+2,R0     ;CALCULATE RELOCATION FACTOR
        MOV #PTRTBL,R1  ;GET AND RELOCATE A POINTER
        ADD R0,R1       ;TO PTRTBL
        MOV #TBLLEN,R2  ;GET LENGTH OF TABLE
LOOP:   ADD R0,(R1)+    ;RELOCATE AN ENTRY
        DEC R2          ;COUNT
        BGE LOOP        ;BRANCH IF NOT DONE
```

Care must be exercised when restarting a program which relocates a table of pointers. The restart procedure must not include the relocation, i.e., the table must be relocated exactly once after each load.

5.10 LOADING UNUSED TRAP VECTORS

One of the features of the PDP-11 is the ability to trap on various conditions such as illegal instructions, reserved instructions, power failure, etc. However, if the trap vectors are not loaded with meaningful information, the occurrence of any of these traps will cause unpredictable results. By loading the vectors as indicated below it is possible to avoid these problems as well as gain meaningful information about any unexpected traps that occur. This technique, which makes it easy to identify the source of a trap, is to load each unused trap vector with:

```
.=trap address
.WORD  .+2,HALT
```

This will load the first word of the vector with the address of the second word of the vector (which contains a HALT). Thus, for example, a halt at location 6 means that a trap through the vector at location 4 has occurred. The old PC and status may be examined by looking at the stack pointed to by register 6.

Trap vectors of interest are listed in Table 5-4.

Table 5-4
Trap Vectors

Vector Location	Halt At Location	Meaning
4	6	Bus Error; illegal instruction; Stack Overflow; Nonexistent Memory; Nonexistent Device; Word Referenced at Odd Address (Loaded by RESMON--causes Monitor TRAP error message)
10	12	Reserved Instruction (Loaded by RESMON--causes Monitor TRAP error message)
14	16	Trace Trap Instruction (000003) or T-bit Set in Status Word (used by ODT; loaded with a HALT by RESMON)
20	22	IOT Executed (used by RESMON)
24	26	Power Failure or Restoration (loaded with a HALT by RESMON)
30	32	EMT Executed (loaded with a HALT by RESMON)
34	36	Trap Executed (Loaded with a HALT by RESMON)

5.11 CODING TECHNIQUES

Because of the great flexibility in PDP-11 coding, time-saving and space-saving ways of performing operations may not be immediately apparent. Some special coding techniques are presented in this section.

5.11.1 Altering Register Contents

The techniques described in this section take advantage of the automatic stepping feature of autoincrement and autodecrement modes when used especially in TST and CMP instructions. These instructions do not alter operands.

NOTE

These alternative ways of altering register contents affect the condition codes differently. Register contents must be even when stepping by 2.

1. Adding 2 to a register might be accomplished by ADD #2,R0. However, this uses two words, whereas CMPB (R0)+,(R0)+ (which also adds 2 to a register), uses only one word.

2. Similarly, subtracting 2 from a register can be done by the complementary instructions SUB #2,R0 or CMPB -(R0),-(R0).

3. Two may be added or subtracted from two different registers, or 4 from the same register, in one single-word instruction as follows:

```
CMP (RØ)+,(RØ)+      ;ADD 4 TO RØ
CMP -(R1),-(R1)      ;SUBTRACT 4 FROM R1
CMP (RØ)+,-(R1)      ;ADD 2 TO RØ, SUBTRACT 2 FROM R1
CMP -(R3),-(R1)      ;SUBTRACT 2 FROM BOTH R3 AND R1
CMP (R3)+,(RØ)+      ;ADD 2 TO BOTH R3 AND RØ
```

4. Variations of the examples above can be employed if the instructions operate on bytes and one of the registers is the Stack Pointer. These examples depend on the fact that the Stack Pointer (as well as the PC) is always autoincremented or autodecremented by 2, whereas registers R0-R5 step by 1 in byte instructions.

```
CMPB (SP)+,(R3)+     ;ADD 2 TO SP AND 1 TO R3
CMPB -(R3),-(SP)     ;SUBTRACT 1 FROM R3 & 2 FROM SP
CMPB (R3)+,-(SP)     ;ADD 1 TO R3, SUBTRACT 2 FROM SP
```

5. Popping an unwanted word off the processor stack (adding 2 to register 6) and testing another value can be two separate instructions or one combined instruction:

```
TST (SP)+            ;POP WORD
TST COUNT            ;SET CONDITION CODES FOR COUNT
```

or

```
MOV COUNT,(SP)+      ;POP WORD & SET CODES FOR COUNT
```

The differences are that TST instructions use three words and clear the Carry bit, while the MOV instruction uses two words and does not affect the Carry bit.

5.11.2 Subroutines

Condition codes set within a subroutine can be used to conditionally branch upon return to the calling program, since the RTS instruction does not affect condition codes.

```
        JSR PC,X        ;CALL SUBROUTINE X
        BNE ABC         ;BRANCH ON CONDITION SET
          .             ;IN SUBROUTINE X
          .
          .
X:                      ;SUBROUTINE ENTRY
```

```
          .
          .
          CMP R2,DEF        ;TEST CONDITION
          RTS PC            ;RETURN TO CALLING PROGRAM
```

When the register in the first operand of a JSR instruction is not the PC, data stored following a subroutine call can be accessed within the subroutine by referencing the register. (The register contains the return address.) For example:

```
          JSR R5,Y
          .WORD HIGH
          .WORD LOW
            .               ;LATEST R5 VALUE WILL POINT HERE
            .
            .
Y:        MOV (R5)+,R2      ;VALUE OF HIGH ACCESSED
          MOV (R5)+,R4      ;VALUE OF LOW ACCESSED
            .
            .
          RTS R5            ;RETURN TO LOCATION
                            ;CONTAINED IN R5
```

Another possibility is:

```
          JSR R5,SUB
          BR PSTARG         ;LOW-ORDER BYTE IS OFFSET TO
                            ;RETURN ADDRESS, WHICH EQUALS NO.
                            ;OF ARGS.
          .WORD A           ;ADDRESS OF ARG A
          .WORD B           ;ADDRESS OF ARG B
          .WORD C           ;ADDRESS OF ARG C
            .
            .
PSTARG:                     ;RETURN ADDRESS
            .
            .
SUB:      MOVB @R5,COUNT    ;GET NO. OF ARGS FROM LOW BYTE
                            ;OF BR (IF DESIRED).
          MOV @14(R5),R2    ;E.G., GET 6TH ARGUMENT
          MOV @6(R5),R1     ;GET THIRD ARGUMENT
            .
            .
          RTS R5            ;RETURNS TO BRANCH WHICH JUMPS PAST
                            ;ARG LIST TO REAL RETURN ADDRESS.
```

In the example above, the branch instruction contributes two main advantages:

1. If R5 is unaltered when the RTS is executed, return will always be to the branch instruction. This ensures a return to the proper location even if the length of the argument list is shorter or longer than expected.

2. The operand of the branch, being an offset past the argument list, provides the number of arguments in the list.

Arguments can be made sharable by separating the data from the main code. This is easily accomplished by treating the JSR and its return as a subroutine itself:

```
CALL:
          .
          .
        JSR PC,ARGLST
          .
          .
ARGLST: JSR R5,SUB
        BR PSTARG
        .WORD A
          .
          .
```

The examples above all demonstrate the calling of subroutines from a non-reentrant program. The called subroutine can be either reentrant or non-reentrant in each case. The following example illustrates a method of allowing calling programs to be reentrant. The arguments and linkage are first placed on the stack, simulating a JSR R5,SUB, so that arguments are accessed from the subroutine via X(R5). Return to the calling program is executed from the stack.

```
CALL:
          .
          .
        MOV R5,-(SP)     ;SAVE R5 ON STACK
        MOV JSBR,-(SP)   ;PUSH INSTRUCTION JSR R6,@R5 ON
          .              ;STACK. PUSH ADDRESSES OF
          .              ;ARGUMENTS ON STACK IN REVERSE
          .              ;ORDER (SEE BELOW)
          .
        MOV BRN,-(SP)    ;PUSH BRANCH INSTRUCTION ON STACK
X:      MOV SP,R5        ;MOVE ADDRESS OF BRANCH TO R5
        JSR PC,SUB       ;CALL SUB AND SAVE RETURN ON STACK
RET:    MOV (SP)+,R5     ;RESTORE OLD R5 UPON RETURN.
          .
          .              ;DATA AREA OF PROGRAM
          .
JSBR:   JSR R6,@R5
BRN:    BR .+N+N+2       ;BRANCH PAST N WORD ARGUMENTS
```

The address of an argument can be pushed on the stack in several ways. Three are shown below.

1. The arguments A, B, and C are read-only constants which are in memory (not on the stack):

```
MOV #C,-(SP)     ;PUSH ADDRESS OF C
MOV #B,-(SP)     ;PUSH ADDRESS OF B
MOV #A,-(SP)     ;PUSH ADDRESS OF A
```

2. Arguments A, B, and C have their addresses on the stack at the Lth, Mth, and Nth bytes from the top of the stack.

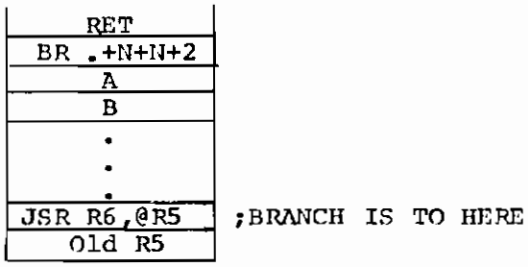
```
MOV N(SP),-(SP)     ;PUSH ADDRESS OF C
MOV M+2(SP),-(SP)   ;PUSH ADDRESS OF B
MOV L+4(SP),-(SP)   ;PUSH ADDRESS OF A
```

Note that the displacements from the top of the stack are adjusted by two for each previous push because the top of the stack is being moved on each push.

3. Arguments A, B, and C are on the stack at the Lth, Mth, and Nth bytes from the top but their addresses are not.

```
MOV #N+2,-(SP)      ;PUSH DISPLACEMENT TO ARGUMENT
ADD SP,@SP          ;CALCULATE ACTUAL ADDRESS OF C
MOV #M+4,-(SP)
ADD SP,@SP          ;ADDRESS OF B
MOV #L+6,-(SP)
ADD SP,@SP          ;ADDRESS OF A
```

When subroutine SUB is entered, the stack appears as follows:



Subroutine SUB returns by means of an RTS R5, which places R5 into the PC and pops the return address from the stack into R5. This causes the execution of the branch since R5 has been loaded at location X with the address of the branch. The JSR branched to then returns control to the calling program, and in so doing, moves the current PC value into the SP, thereby removing everything above the old R5 from the stack. Upon return at RET this too is popped, restoring the original R5 and SP values.

The next example involves a recursive subroutine (one that calls itself). Its function is to look for a matching right parenthesis for every left parenthesis encountered. The subroutine is called by JSR whenever a left parenthesis is encountered (R2 points to the character following it). When a right parenthesis is found, an RTS PC is executed and if the right parenthesis is not the last legal one, another is searched for. When the final matching parenthesis is found, the RTS returns control to the main program.

```
A:      MOVB (R2)+,R0   ;GET SUCCESSIVE CHARACTERS
        CMP #'(,R0      ;LOOK FOR LEFT PARENTHESIS
        BNE B           ;FOUND?
        JSR PC,A        ;LEFT PAREN FOUND, CALL SELF
        BR A            ;GO LOOK AT NEXT CHARACTER
```

```
B:      CMPB #'),R0     ;LEFT PAREN NOT FOUND, LOOK FOR
                        ;RIGHT PAREN
        BNE A           ;FOUND? IF NOT, GO TO A
        RTS PC          ;RETURN PAREN FOUND, IF NOT LAST,
                        ;GO TO B. IF LAST, GO TO MAIN
                        ;PROGRAM
```

The example below illustrates the use of co-routines, called by JSR PC,@(SP)+. The program uses double buffering on both input and output, performing as follows:

Write	O1	concurrently	Write	O2	concurrently
Read	I1		Read	I2	
Process	I1		Process	I1	

JSR PC,@(SP)+ always performs a jump to the address specified on top of the stack and replaces that address with the new return address. Each time the JSR at B is executed, it jumps to a different location; initially to A and thereafter to the location following the JSR executed prior to the one at B. All other JSR's jump to B+2.

```
        PC=%7
BEGIN:    .                     ;DO I/O RESETS, INITS, ETC.
          .
          .
        IOT                     ;READ INTO I1 TO START PROCESS
        .BYTE READ,INSLOT
        .WORD I1
        MOV #A,-(6)             ;INITIALIZE STACK FOR FIRST JSR
B:      JSR PC,@(6)+            ;DO I/O FOR O1 AND I1 OR O2
          .                     ;AND I2
          .                     ;PERFORM PROCESSING
          .
        BR B                    ;MORE I/O
;END OF MAIN LOOP
;I/O CO-ROUTINES
A:      IOT                     ;READ INTO I2
        .BYTE READ,INSLOT
        .WORD I2
          .
          .                     ;SET PARAMETERS TO PROCESS
          .                     ;I1, O1
        JSR PC,@(6)+            ;RETURN TO PROCESS AT B+2
        IOT                     ;WRITE FROM O1
        .BYTE WRITE,OUTSLOT
        .WORD O1
        IOT                     ;READ INTO I1
        .BYTE READ,INSLOT
        .WORD I1
          .
          .                     ;SET PARAMETERS TO PROCESS
          .                     ;I2, O2
        JSR PC,@(6)+            ;RETURN TO PROCESS B+2
        IOT                     ;WRITE FROM O2
        .BYTE WRITE,OUTSLOT
        .WORD O2
        BR A                    ;READ INTO I2
```

The trap handler below simulates a two-word JSR instruction with a one-word TRAP instruction. In this example, all TRAP instructions in the program take an operand and trap to the handler address at location 34. The table of subroutine addresses (e.g., A, B, ...) can be constructed as follows:

```
TABLE:
        CALA=.-TABLE
        .WORD A          ;CALLED BY: TRAP CALA

        CALB=.-TABLE
        .WORD B          ;CALLED BY: TRAP CALB
          .
          .
          .
```

Another way to construct the table:

```
TABLE:
        CALA=.-TABLE+TRAP
        .WORD A          ;CALLED BY: CALA
          .
          .
          .
```

The trap handler for either of the above methods follows:

```
TRAP34: MOV @SP,2(SP)          ;REPLACE STACKED PS WITH PC*
        SUB #2,@SP             ;GET POINTER TO TRAP
                               ;INSTRUCTION
        MOV @(SP)+,-(SP)       ;REPLACE ADDRESS OF TRAP WITH
                               ;TRAP INSTRUCTION ITSELF
        ADD #TABLE-TRAP,@SP    ;CALCULATE SUBROUTINE ADDR.
        MOV @(SP)+,PC          ;JUMP TO SUBROUTINE
```

*Replacing the saved PS loses the T-bit status. If a breakpoint has been set on the TRAP instruction, ODT will not gain control again to reinsert the breakpoints because the T-bit trap will not occur.

In the example above, if the third instruction had been written MOV @(SP),(SP) it would have used an extra word since @(SP) is in Index Mode and assembles as @0(SP). In the final instruction, a jump was executed by a MOV @(SP)+,PC, becuase no equivalent JMP instruction exists.

Following are some JMP and MOV equivalences (note that JMP does not affect condition codes).

```
JMP (R4)        =    MOV R4,PC
JMP @(R4)       =    MOV (R4),PC
  (2 Words)              (1 Word)
None            =    MOV @(R4),PC
JMP -(R4)       =    None
JMP @(R4)+      =    MOV (R4)+,PC
JMP @-(R4)      =    MOV -(R4),PC
None            =    MOV @(R4)+,PC
None            =    MOV @-(R4),PC
JMP X           =    MOV #X,PC
JMP @X          =    MOV X,PC
None            =    MOV @X,PC
```

The trap handler can also be useful as a patching technique. Jumping out to a patch area is often difficult because a two-word jump must be performed. However, the one-word TRAP instruction may be used to dispatch to patch areas. A sufficient number of slots for patching should first be reserved in the dispatch table of the trap handler. The jump can then be accomplished by placing the address of the patch area into the table and inserting the proper TRAP instruction where the patch is to be made.

5.12 ASSEMBLY DIALOGUE

During assembly, the Assembler will pause to print on the console terminal various messages to indicate that some response must be made by the user before the assembly process can continue. CTRL/P may be typed at any time to stop the assembly process and restart the initial dialogue, as mentioned in Section 5.1.3.

If the specified assembly listing output device is the Line Printer and it is out of paper, the Assembler prints on the terminal:

```
EOM?
```

and waits for paper to be placed in the device. Typing the RETURN key will continue assembly.

Other conditions which may cause the EOM? message for the line printer are: a) no power, b) printer drum gate open, and c) too hot.

There is no EOM if the line printer is switched off-line, although characters may be lost for this condition as well as for an EOM.

If the end-of-tape is reached during cassette output and the user has not indicated an overflow file using the /O option, the Assembler will print:

```
EOM?
RETRY?
```

The user must mount a different output cassette and then type any character on the keyboard; the Assembler will retry the same assembly using the new output cassette. Alternatively, the user can type ↑C and return to the KBL. (In systems greater than than 8K, the user may also type ↑P, which returns control to the CSI, enabling another command string to be entered.)

If a hard read error is detected on one of the input files, the Assembler will print:

```
%BAD TAPE
RETRY?
```

Typing ↑C will return control to the KBL. Typing ↑P or any other character will cause the Assembler to retry the same assembly. (In systems larger than 8K, the Assembler will return to the CSI and allow the user to input a new command.)

If the last file does not have a .END, the Assembler will print:

```
?NO END STMT
```

and will emulate a .END assembler directive. Note that when .END is emulated in this manner the error counter is incremented by one.

5.13 ASSEMBLY LISTING

The CAPS-11 Assembler produces a side-by-side assembly listing of symbolic source statements, their octal equivalents, assigned addresses, and error codes, as follows:

```
CCAAAAAA  000000'SSS.......S
          000000
          000000
```

The C's represent the error code field; error codes (listed in Table 5-4) are flagged in this field. The A's represent the 6-bit octal address, while the 0's represent the object data in octal. The S's represent the source statement, and ' represents a single apostrophe which will be printed whenever either the second, third or both words of the instruction will be modified by the Linker. The Assembler accepts on input 72(decimal) characters per line. Any additional characters on the line will be ignored and the Assembler will generate an 'L' error code.

If an instruction requires two or three words, the second and third words of the statement are listed under the command word. No addresses precede the second or third words since the address order is sequential. The second and third words can be eliminated from the assembly listing by means of the /X switch.

The object data field of a .BYTE directive is assembled as three octal digits.

The value of the defining expression in a direct assignment statement is given in the object code field although it is not actually part of the code of the object program.

The .ASECT and .CSECT directives cause the current value of the appropriate location counter (absolute or relocatable) to be printed.

Each page of the listing is headed by a PAL identification line and a page number (octal).

An example of an assembly listing is shown in Chapter 7, Section 7.6.

5.14 OBJECT MODULE OUTPUT

The output of the assembler during the binary object pass is an object module which is meaningful only to the Linker. An overview of what this object module contains and at what stage each part of it is produced follows.

The binary object module consists of three main types of data block:

1. Global Symbol Directory (GSD)
2. Text blocks (TXT)
3. Relocation Directory (RLD)

5.14.1 Global Symbol Directory

As the name suggests, the GSD contains a list of all the global symbols together with the name of the object module. Each symbol is in Radix 50 form and contains information regarding its mode and value whenever known.

The GSD is created at the start of the binary object pass.

5.14.2 Text Blocks

The text blocks consist entirely of the binary object data as shown in the listing. Operands are in the unmodified form.

5.14.3 Relocation Directory

The RLD blocks consist of directives to the Linker which may reference the text blocks preceding the RLD. These directives control the relocation and linking process.

Text and RLD blocks are constructed during the binary object pass. Outputting of each block is done whenever either the TXT or RLD buffer is full and whenever the location counter needs to be modified.

5.15 ERROR CODES

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

Table 5-5
Assembler Error Codes

Error Code	Meaning
A	Addressing error. An address within the instruction is incorrect; may also indicate a relocation error.
B	Bounding error. Instructions or word memory data are being assembled at an odd address in memory. The location counter is updated by +1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.
L	Line buffer overflow. Extra characters on a line (more than 72(decimal)) are ignored.
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.
N	Number containing 8 or 9 has decimal point missing. The number is assembled as a decimal number.
P	Phase error. A label's definition or value varies from one pass to another.
Q	Questionable syntax. There are missing arguments, the instruction scan was not completed, or a carriage return was not immediately followed by a line feed or form feed.
R	Register-type error. An invalid use of or reference to a register has been made.

(Continued on next page)

Table 5-5 (Cont.)
Assembler Error Codes

Error Code	Meaning
S	Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the symbol table, the Assembler outputs the current source line with the S error code, then returns to the Monitor (or to the CSI in systems larger than 8K).
T	Truncation error. A number generated more than 16 bits of significance, or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.

In addition to the error codes listed above, the following messages may also occur (error messages which are followed by a question mark allow the user to type a CTRL/C to return to the KBL or a CTRL/P to retry the operation):

Table 5-6
Assembler Error Messages

Message	Meaning
%BAD CMD STRING	One of the following errors has occurred in the user's command string: No output was specified; No input was specified; Input and output were specified on the same drive; Input was specified from a device other than cassette; Binary output was specified to a device other than cassette.
?BAD TAPE?	A checksum or other hard error occurred during a file lookup or enter command. Typing any character will cause the Assembler to retry the operation.

(Continued on next page)

Table 5-6
Assembler Error Messages

Message	Meaning
%BAD TAPE RETRY?	A hard read error was detected on one of the input files; typing any character (except CTRL/C) will cause the Assembler to retry the same assembly (in systems larger than 8K, the Assembler will return to the CSI and allow the user to input a new command).
EOM?	The line printer is out of paper or is not powered up; the drum gate is open; or the printer is too hot.
EOM? RETRY?	The end of the tape was reached during cassette output and no overflow file was specified. The user may mount another cassette and then type any keyboard character to instruct the Assembler to retry the assembly using the new output cassette.
?FILE NOT FND?	The Assembler could not find one of the input files. The user may mount another cassette and type any character on the keyboard to instruct the Assembler to retry the lookup on the same drive. Typing a CTRL/P will restart the Assembler (if the system is 8K the same assembly will be restarted, otherwise control will return to the CSI.)
?NO END STMT	The file does not contain an .END directive; the Assembler assumes an .END statement.
?SWITCH ERROR:'x'?	An undefined option character (x) was found in the command string. Typing any character on the keyboard will cause the Assembler to ignore the option and continue.
?TAPE FULL?	The specified output cassette is completely full. Mounting a different cassette on the same unit and typing any character instructs the Assembler to attempt to open the file on a new cassette.

CHAPTER 6

LINKING ASSEMBLED PROGRAMS

The CAPS-11 Linker converts object modules produced by the Assembler into a format suitable for loading and execution. This allows the user to assemble a large program in several small subprograms or to separately assemble a main program and each of its subroutines without assigning an absolute load address at assembly time. Object modules are processed by the Linker to:

Relocate each object module and assign absolute addresses,

Link the modules by correlating global symbols defined in one module and referenced in another module,

Print a load map which displays the assigned absolute addresses,

Output a load module which can subsequently be loaded and executed.

Advantages of using the CAPS-11 Assembler and the Linker include the following:

1. A program is divided into segments (usually subroutines) which are assembled separately. If an error is discovered in one segment, only that segment need be edited and reassembled. The new object module is then linked with the other object modules.

2. Absolute addresses need not be assigned at assembly time as the Linker automatically assigns absolute addresses. This keeps programs from overlapping each other and also allows subroutines to change size without influencing the contents of other routines.

3. Separate assemblies allow the total number of symbols to exceed the number allowed in a single assembly.

4. Since global symbols are usually referenced from more than one object module, the programmer must be sure that his names for such symbols are unique between object modules. However, this does not apply when the symbol is internal; since an internal symbol is referenced only from within the current assembly, the same symbol names may be used in several different modules.

5. Subroutines may be provided for general use in object module form to be linked into the user's program.

6.1 CALLING AND USING THE LINKER

The Linker is called from the System Cassette by typing:

```
.R LINK
```

In response to the dot printed by the Keyboard Listener. The Command String Interpreter responds by printing an asterisk (*) at the left margin of the teleprinter paper. The user may respond with his I/O specifications as soon as the asterisk appears even though the remainder of the Linker is being loaded into memory simultaneously.

The Linker requires two passes over the input object modules. During the first pass any undefined globals are listed on the console terminal, and a global symbol table is constructed which includes all the control section names and global symbols in the input modules. On the second pass, the Linker reads the object modules, performs most of the functions listed in the introductory description and produces a load module which can be loaded (using the Monitor LOAD command) and executed. The load module is output in binary image format.

After execution, control returns to the CSI, indicated by an asterisk at the left margin; the user may enter another command string.

6.1.1 Linker Options

The options listed in Table 6-1 are available for use by the Linker and are designated by the user in the I/O specification line:

Table 6-1
Linker Options

Option	Meaning
/C	This option allows the I/O specification line to be broken into several segments. The option character is followed immediately by a carriage return and the I/O specification is continued on the next line; this line must begin with a comma.
/F	This option is valid only after an input filename and indicates that the Linker should not perform a REWIND operation but should continue searching the cassette in a forward direction for this file. This feature saves the user time when he wishes to input several files from one cassette and these files appear on that cassette in the same order as they are to be linked. The /F option prevents the Linker from performing a REWIND before accessing each file.
/O	This option is valid only after an output filename and indicates that the file (immediately preceding the option) is to be created and used only if a previously opened output file has been written to the end of a cassette and more output remains. All output files can later be combined under one name using PIP (see Chapter 8).
/P	This option is used whenever a file referenced in an I/O specification line is on a cassette which is not currently mounted on the unit drive. Before attempting to search for the file, the Linker instructs the user to mount the proper cassette on the drive by printing #? where # represents the drive number. After the user has switched cassettes on the drive, he may continue execution by typing any character on the keyboard.
/S	This option is valid only after an input filename and indicates that two or more object modules have been combined (using PIP) under the single filename. The option instructs the Linker not to skip to the next input filename until it has obtained all necessary information for the files included in the first.

(Continued on next page)

Table 6-1 (Cont.)
Linker Options

Option	Meaning
/T	The /T option is valid only after an input filename and indicates that the transfer address of this particular object module is to be used as the transfer address of the final load module. If more than one /T option is indicated in the I/O specification line, only the last one is significant.
/B:n	The program is to be linked with its lowest location at n. If n is not specified, the Linker assumes location 600. (The Monitor uses locations 400-600 for stack space while loading is in progress, so the user should not attempt to link any data for loading into that area.)
/H:n	The program is to be linked with its highest location at n. If n is not specified, the Linker assumes that the last location of the user program will go just under CLOD11 (see Chapter 3, Figure 3-1). The user can then use the LOAD/G command to run his file.

NOTE

If neither the /B or /H options are indicated (or if both are indicated), the Linker will load the program with its highest location just below the KBL, so that the entire CAPS-11 Monitor will be preserved.

The Linker does not give a warning if a program is linked in memory in such a way that its lowest address falls below address 0. However, this condition can be easily recognized by examining the low and high limits which are always printed in the load map.

If the user wishes to link his program for an overlay load (via the LOAD/O command), he can link it using the /B switch with no value. The lower limit is set to 600 and the Linker will set the high limit to allow just enough memory for CABLDR and CBOOT (which the user needs to load his program and to re-boot the CAPS-11 System). The .LIMIT assembler directive (see Chapter 5) can be used to instruct the Linker to load the value of the high limit into the user-program. If the user wants to link his program at the top of memory, he should use the

/H switch designating a value which is 1214(octal) bytes less than the number of bytes in his machine. For an 8K system this would be 40000-1214 or 36564(octal); the program would then be linked using /H:36564.

6.1.2 Input and Output Specifications

The Linker allows two output specifications, one for binary output and one for the load map output. Inserting only a comma for either output specification instructs the Linker that no output of this type is to be produced. Any number of input files are acceptable. The format of the I/O specification line is:

```
*DEV:FILENA.LDA/OPT,DEV:FILENA.MAP/OPT=DEV:INPUT1.OBJ/C
,DEV:INPUT2.OBJ/OPT,...DEV:INPUTn.OBJ/OPT
```

DEV represents one of the CAPS-11 I/O devices; OPT represents any of the options listed in Table 6-1. Unless otherwise indicated, the Linker assumes the extension .LDA for the binary output file, .MAP for the load map output, and .OBJ for the relocatable binary object module input files.

For example, consider the following I/O specification lines:

```
.R LINK
*CT1:PROG,LP:=0:RES,TYPE.3RN/F,0:SIGN/P/C
,TABLE.DAT/F/H:2000
```

This command line causes the Linker to output the load module (PROG.LDA) on cassette drive 1, and the load map on the line printer; the input files are RES.OBJ and TYPE.3RN, both on cassette drive 0 (the /F option indicates that TYPE.3RN follows RES.OBJ on the cassette and that no rewind is necessary); the next input file (SIGN.OBJ) is on a cassette which is not currently mounted, so the user asks to be prompted (via the /P option) when the file is needed; the command string is continued on the next line, and the final input file is TABLE.DAT which is in a forward direction (in relation to SIGN.OBJ) on the cassette now mounted on drive 0. The program (PROG.LDA) is linked so that its highest address is at location 2000.

```
.R LINK
*,TT:=ACE.1,BAK.OBJ
```

In this example, no output load module is created; the load map is output to the console terminal; the input files are ACE.1 and BAK.OBJ, both on cassette drive 0. The cassette is rewound before BAK.OBJ is accessed. Since no linking address is specified in the command line, the program is linked so that its highest location will load just below the KBL, preserving the entire CAPS-11 Monitor.

6.1.3 Restarting the Linker

The Linker may be restarted at any time (while it is memory) by typing CTRL/P. This echoes as ↑P followed by a carriage return/line feed. Control is passed back to the Command String Interpreter and the user

may input a new command string. (An exception occurs when typing ↑P while the load map is being output--this causes the Linker to terminate the map immediately and start Pass 2.)

6.2 ABSOLUTE AND RELOCATABLE PROGRAM SECTIONS

As explained in Chapter 5, the programmer may designate sections of his program as absolute or relocatable by means of the .ASECT and .CSECT assembler directives. (The Linker assumes .CSECT if neither directive is indicated.) In an absolute section, a direct assignment statement of the form .=EXPRESSION initially assigns an absolute address to an instruction; succeeding instructions and data in the absolute section are then assigned absolute addresses in accordance with the assembly location counter.

Instructions and data encountered in relocatable sections are assigned absolute addresses by the Linker. These addresses are normally assigned such that the relocatable sections are loaded just below the lowest location of the KBL (although the user can control this with the Linker /B and /H options). All instructions and data which the programmer has designated in a relocatable section (called a control section and indicated by a .CSECT directive) are modified appropriately and as necessary by the Linker to account for their relocation.

6.2.1 Named and Unnamed Control Sections

The Linker has the capability of handling named and unnamed control sections. (Assigning names to control sections is a feature not supported by the CAPS-11 Assembler, although the programmer may have occasion to use other assemblers which do allow this feature.) An unnamed control section (which is actually assigned a special default name of 6 blanks, i.e., .CSECT) is internal to each object module and is treated independently from any other unnamed control section. The Linker assigns each unnamed section an absolute address such that it occupies an exclusive area of memory. Named control sections, on the other hand, are treated globally. That is, if different object modules have control sections with the same name, (for example, .CSECT DATA), they are all assigned the same absolute load address and the size of the area reserved for loading of the section is that of the largest. Thus, named control sections allow for the sharing of data and/or instructions among object modules. A restriction is that the name of a control section must not be the same as the name of a global entry symbol as this will result in multiple definition errors.

The absolute section is always assigned the special name .ABS (i.e., .ASECT.ABS) by the Linker.

6.3 GLOBAL SYMBOLS

Global symbols provide the links, or communication, between object modules. Global symbols are created with the .GLOBL assembler directive (as described in Chapter 5). Symbols which are not global are called internal symbols. If the global symbol is defined in an object module (as a label or by direct assignment) it is called an entry symbol and other object modules can reference it. If the global symbol is not defined in the object module, it is an external symbol and is assumed to be defined (as an entry symbol) in some other object module.

As the Linker reads the object modules it keeps track of all global symbol definitions and references. It then modifies the instructions and/or data which reference the global symbols. Undefined globals are printed on the console terminal during pass 1.

6.4 INPUT AND OUTPUT

Linker input and output is in the form of modules; one or more input modules (object modules produced by the Assembler) are used to produce a single output (load) module.

6.4.1 Object Modules

Object files, consisting of one or more object modules, are used as input to the Linker; these object modules have been previously created by the Assembler, and more than one object module may have been combined using PIP to form a single object file. The Linker reads each object module twice; that is, it is a two-pass processor. During the first pass, each object module is read so that absolute addresses can be assigned to all relocatable sections and all globals can be assigned absolute values. The information the Linker needs for this process is contained in the global symbol directory (GSD), located at the beginning of each object module. Unless the /S switch has been indicated in the command line, during the first pass the Linker reads only the GSD at the beginning of the object file.

On the second pass, the Linker reads the object modules, links and relocates the modules, and outputs the load module. During this pass it uses a block of information output by the Assembler in the object file which is called the Relocation Directory (RLD).

6.4.2 Load Module

The primary output of the Linker is the load module which may be loaded and run under the CAPS-11 Keyboard Monitor. The load module consists of formatted binary blocks holding absolute load addresses and object data. The first few words of data will be the communications directory (COMD) which will have an absolute load address equal to the lowest relocated address of the program. CABLDR or CLOD11 will load the COMD at the specified address but the COMD will then be overlayed by the program. The end of the module will be

indicated by a TRA block; that is, a block containing only a load (or transfer) address. The byte count in the formatted binary block will be 6 on this block; on all other blocks the byte count will be larger than 6. The TRA is normally selected by the Linker to be the first even transfer address seen. Thus, if four object modules are linked together and if the first and second had an .END statement, the third had a .END A and the fourth had a .END B, the transfer address would be A of module three. However, the user can specify directly which transfer address is to be used by the Linker via the /T option as described in Table 6-1.

NOTE

The overlaying of the COMD by the relocated program is a method which allows CABLDR to handle load modules with a COMD. However, a problem arises if a load module is to be loaded by CABLDR and either of the following conditions is true:

1. The object modules used to construct the load module contained no relocatable code; or

2. The total size of the relocatable code is less than 32(10) bytes (the size of the COMD).

In either case, there is not enough relocatable code to overlay the COMD which means the COMD will load into parts of memory not intended to be altered by the user. The COMD's load address, selected by the Linker in the above cases, is 400(octal). This area is reserved for the Monitor stack while loading is in progress, so no user data should be destroyed when the COMD is loaded there.

6.4.3 Load Map

The load map provides several types of information concerning the load module's make-up. The map begins with an indication of the low and high limits of the relocatable code and the transfer address. Then a section of the map is allocated for each object module included in the linking process. Each of these sections begins with the module's name followed by a list of the control sections and the entry points for each control section. The base of each control section (its low address) and its size (in bytes) is printed to the right of the section name (enclosed in angle brackets). Following each section name printout is a list of entry points and their addresses. After all information has been printed for each object module, any undefined

symbols are listed. Note that modules are loaded such that if modules A, B and C are linked together, A is lowest and C is highest in memory.

The format is self-explanatory and is illustrated in Section 6.6.

NOTE

A CTRL/O typed during output of the Linker load map is treated somewhat differently than during normal CTRL/O usage. If the user does not wish to list all entry points when the map is being output on the console terminal, typing ↑O will suppress output of the load map until the beginning of the section for the next module; the Linker will then automatically restart the load map output for this module.

6.5 ERROR MESSAGES

The following messages are printed by the Linker whenever it detects an error during the linking process. Two types of errors may occur--fatal and non-fatal.

6.5.1 Non-Fatal Errors

Table 6-2 lists errors which can occur without causing an interruption in the linking process.

Table 6-2
Linker Non-Fatal Error Messages

Message	Meaning
?BAD TAPE?	A checksum or other hard error occurred during a file LOOKUP or ENTER command. Typing any character will cause the Linker to retry the operation.
?BYTE RELOC ERROR AT ABS ADDRESS xxxxxx	This message designates a byte relocation error. The Linker will try to relocate and link byte quantities; however, relocation will usually fail and linking may fail. (Failure is defined as the

(Continued on next page)

Table 6-2 (Cont.)
Linker Non-Fatal Error Messages

Message	Meaning
	high byte of the relocated value or the linked value not being zero.) In such a case, the value is truncated to 8 bits and the error message is printed to inform the user. The Linker then automatically continues.
?FILE NOT FND?	The Linker could not find one of the input files. This is generally caused when the wrong cassette is mounted on a drive. Upon occurrence of the message, the user may mount the correct cassette; typing any character on the keyboard will cause the Linker to retry the LOOKUP on the same drive. Typing a CTRL/P will restart the Linker; typing a CTRL/C will cause a return to the Monitor.
?MAP DEVICE EOM?	The Load Map device EOM error allows the user an option to fix the device and continue or abort the map listing. Typing a carriage return (or any other character) causes the Linker to continue (if the map device was cassette, the map listing is continued on the console terminal); a ↑P will cause the map to be aborted.
?MODULE NAME xxxxxx NOT UNIQUE	This error is detected during pass 1 and results from a non-unique object module name. The module is rejected and the Linker will then ask for more input.
?SWITCH ERROR: 'x'?	An undefined option character (x) was found in the command string. Typing any character on the keyboard will cause the Linker to ignore the option and continue.
?TAPE FULL?	The specified output cassette is full. Mounting a different cassette on the same unit and typing any character instructs the Linker to attempt to open the file on the new cassette.

(Continued on next page)

Table 6-2 (Cont.)
Linker Non-Fatal Error Messages

Message	Meaning
?xxxxxx MULTIPLY DEFINED BY MODULE xxxxxx	This message results during pass 1 if globals have been defined more than once. The second definition is ignored and the Linker continues.

6.5.2 Fatal Errors

The following errors are fatal and cause control to return to the Monitor.

Table 6-3
Linker Fatal Error Messages

Message	Meaning
%BAD CMD STRING	One of the following errors has occurred in the user's command string: No output was specified; No input was specified; Input and output were specified on the same drive; Input was specified from a device other than cassette; Binary output was specified to a device other than cassette.
%CAS. CHECKSUM	A checksum error was detected while reading a cassette block.
%ODD ADDRESS	An odd address was specified using the /B or /H options in the command string.
%SYMBOL TABLE OVERFLOW-MODULE xxxxxx, SYMBOL xxxxxx	A symbol table overflow has occurred (the 8K Linker has room for approximately 225 (decimal) symbols).

(Continued on next page)

Table 6-3 (Cont.)
Linker Fatal Error Messages

Message	Meaning
%SYSTEM ERROR xx	A system error has occurred; xx represents an identifying number from the following list: 01 Unrecognized symbol table entry found. 02 A relocation directory references a global name which cannot be found in the symbol table. 03 A relocation directory contains a location counter modification command which is not last. 04 Object module does not start with a GSD. 05 The first entry in the GSD is not the module name. 06 A Relocation Directory (RLD) references a section name which cannot be found. 07 The transfer address (TRA) specification references a non-existent module name. 08 The transfer address (TRA) specification references a non-existent section name. 09 An internal jump table index is out of range. 10 A checksum error occurred on the object module. 11 An object module binary block is too big (more than 64(decimal) words of data). 12 A device error occurred on the load module output device.

All system errors except number 12 indicate a program failure either in the Linker or the program which generated the object module. Error 05 can occur if a file is read which is not an object module.

6.6 EXAMPLE USING THE LINKER

The following example demonstrates how the user can link the PAL Assembler for an 8K system. A load map is produced on the console terminal, and the load module is output to cassette drive 1 as PAL8.SRU. (Refer to Appendix E for complete instructions on how to build systems for any configuration.)

```
.R LINK

*1:PAL8.SRU,TT:=PAL11S,PAL8KS/F,CSITAC/F/B:400

CAPS-11 LINK V01
LOAD MAP

TRANSFER ADDRESS: 002754
LOW LIMIT: 000400
HIGH LIMIT: 030544
********
PAL
SECTION          ADDRESS SIZE

<. ABS.>         000000  000000
COLLID  000150   RESREG  104124  SAVERE  104122  STSIZE  001312

<      >         000400  020222
ASCII   007064   ASECT   006232  BYTE    007202  CSECT   006146
END     010074   ENDC    006076  EOT     006566  EVEN    006676
GLOBL   006324   IFDF    005712  IFG     005616  IFGE    005620
IFL     005622   IFLE    005624  IFNDF   005716  IFNZ    005614
IFZ     005632   LIMIT   007002  LIST    006130  NLIST   006136
RAD50   007042   TITLE   006520  WORD    007172
********
PALSYM
SECTION          ADDRESS SIZE

<      >         020622  004703
CHAR13  020622   CHAR46  022134  DOTFLG  025323  FLAGS   024760
IDOT    024354   SVALUE  023446
********
CSITAC
SECTION          ADDRESS SIZE

<      >         025526  003016
BADEXI  026460   CSITAC  025530  ERRCOM  026350  HDRBUF  027504
IGNRFU  025526   INOUT   025554  MESOUT  027214  MESRES  027210
NMBUF0  026662   SWMSFL  026064  TYPE00  026673  TYPE01  026733

PASS 2

*
```

CHAPTER 7

DEBUGGING THE OBJECT PROGRAM

ODT (On-line Debugging Technique) aids the user in debugging assembled and linked object programs. Using the console terminal keyboard, the user interacts with ODT and his object program to:

Print the contents of any location for examination or alteration

Run all or any portion of the object program using the breakpoint feature

Search the object program for specific bit patterns

Search the object program for words which reference a specific word

Calculate offsets for relative addresses

Fill a block of words or bytes with a designated value

During a debugging session, the user should have at the console terminal an assembly listing of the program to be debugged. Minor corrections to this program can be made on-line and the program may then be run under control of ODT to verify any change made. However, major corrections such as a missing subroutine should be noted on the assembly listing and incorporated in a subsequent updated program assembly.

7.1 CALLING AND USING ODT

ODT is supplied as a relocatable object module and is also stored on the System Cassette. It is linked so as to be loaded just under the KBL (refer to Appendix E); the procedure for loading ODT and the user program is:

```
.LOAD FILENA.EXT

.R ODT
```

These commands load the user-program to be debugged into memory and call and start the debugger. This is the most common form of ODT use, as it is expected that user programs will start in low memory and that the standard location of ODT will suffice. However, the user may alternatively relink ODT using the CAPS-11 Linker, or link ODT along with his program.

7.1.1 ODT Options

The only options allowed are those used by the LOAD command when the user program is loaded into memory. ODT itself does not utilize the CSI.

7.1.2 Input/Output Specifications

The input file is indicated in the LOAD command. No output file specifications are allowed. ODT is an on-line utility program which aids the user in determining corrections and modifications to his program; these corrections may then be implemented using the Editor and Assembler.

7.1.3 Restarting ODT

If ODT is in control, typing CTRL/P will restart ODT (indicated by an asterisk at the left margin), removing all breakpoints and clearing all relocation registers. If the user program is in control and no breakpoint or HALT instruction is encountered to stop program execution, then ODT may be restarted by following one of the techniques described in Section 7.4.2.

7.2 RELOCATION

The Assembler produces a binary relocatable object module; the base address of this module is assumed to be location 000000 and the addresses of all program locations as shown in the assembly listing are indicated relative to this base address. After the module is linked by the Linker, many values within the program and all the addresses of locations will be incremented by a constant whose value is the actual absolute base address of the module after it has been relocated. This constant is called the relocation bias for the module.

A linked program may contain several relocated modules each with its own relocation bias; since, in the process of debugging, these biases will have to be subtracted from absolute addresses continually in order to relate relocated code to assembly listings, ODT provides an automatic relocation facility.

The basis of the relocation facility lies in eight relocation registers numbered 0 through 7 (these should not be confused with general registers 0-7) which may be set to the values of the relocation biases at any given time during debugging (this procedure is explained in Section 7.3.13). Relocation biases are obtained by consulting the memory map produced by the Linker. Once set, a relocation register is used by ODT to relate relocatable code to relocated code. The relocation registers are initialized by ODT to -1. (For more information on the exact nature of the relocation process, consult Chapter 6.)

7.2.1 Relocatable Expressions

A relocatable expression is evaluated by ODT as a 16-bit (6 octal digit) number and may be typed in any one of the three forms presented in Table 7-1. In this table, n represents an integer in the range 0 to 7 inclusive and k stands for an octal number of up to six digits in length with a maximum value of 177777. If more than six digits are typed, ODT uses the last six digits truncated to the low-order 16 bits. k may be preceded by a minus sign, in which case its value is the two's complement of the number typed. For example:

k (number typed)	Value
1	000001
-1	177777
400	000400
-177730	000050
1234567	034567

Table 7-1
Forms of Relocatable Expressions (r)

	r	Value of r
a)	k	The value of r is simply the value of k.
b)	n,k	The value of r is the value of k plus the contents of relocation register n. If the n part of this expression is greater than 7, ODT uses only the last octal digit of n.
c)	C or C,k or n,C or C,C	Whenever the letter C is typed, ODT replaces C with the contents of a special register called the Constant Register. This value has the same role as the k or n that it replaces (i.e., when used in place of n it designates a relocation register). The Constant Register is accessed by typing the symbol $C and may be set to any value. (See Section 7.3.10.)

In the following examples, assume that relocation register 3 contains 003400 and that the Constant Register contains 000003:

r	Value of r
5	000005
-17	177761
3,0	003400
3,150	003550
3,-1	003377
C	000003
3,C	003403
C,0	003400
C,10	003410
C,C	003403

NOTE

For simplicity's sake, most examples in this section use form a; all three forms of r are equally acceptable, however.

7.3 COMMANDS AND FUNCTIONS

After ODT is loaded and started it indicates its readiness to accept commands by printing an asterisk (*) at the left margin of the console terminal paper. Most ODT commands are issued in response to the asterisk and are composed of the characters and symbols shown in this section. By using ODT a word can be examined and changed, the object program can be run in its entirety or in segments, and memory can be searched for certain words or references to these words. Each command is explained in detail here; a command summary is provided in Appendix C.

7.3.1 Printout Formats

Normally, when ODT prints addresses it attempts to print them in relative form (form b in Table 7-1). ODT assumes the user has set the relocation registers with the relocation biases and checks for the register whose value is closest but less than or equal to the address to be printed. It then represents this address relative to the contents of the relocation register. However, if no relocation register fits the requirement (that is, the user has not entered the relocation biases for his object modules), the address is printed in absolute form (form a in Table 7-1). Since the relocation registers are initialized to -1 (the highest number) the addresses are initially printed in absolute form. If any relocation register subsequently has its contents changed, it may then, depending on the command, qualify for relative form.

For example, suppose relocation registers 1 and 2 contain 1000 and 1004 respectively, and all other relocation registers contain numbers much higher. Then the following sequence might occur (the slash command causes the contents of the location to be printed; the line feed command (↓) accesses the next sequential location):

```
*774/000000↓
000776 /000000↓
1,000000 /000000↓    (absolute location 1000)
1,000002 /000000↓    (absolute location 1002)
2,000000 /000000     (absolute location 1004)
```

The printout format is controlled by a special register called the Format Register. Initially, this register is set to 0 which instructs ODT to print addresses relatively whenever possible. However, the user may access the Format Register by typing $F, thus allowing the register to be modified. By changing the contents to any non-zero value, the user instructs ODT to print all addresses in absolute form.

7.3.2 Opening, Changing, and Closing Locations

An open location is one whose contents ODT has printed for examination, making those contents available for change; a closed location is one whose contents are no longer available for change. Several commands are used for opening and closing locations.

Any command used to open a location when another location is already open first causes the currently open location to be closed. The contents of an open location may be changed by typing the new contents followed by a single character command which requires no argument (i.e., ↓, ↑, RETURN, ←, @, >, <).

The Slash, /

A location is opened by typing its address followed by a slash. ODT responds by printing the contents of the location; for example:

```
*1000/012746
```

Location 1000 is open for examination and is available for change.

If the contents of an open location are not to be changed, typing the RETURN key causes the location to be closed; ODT prints an asterisk and waits for another command.

To change the contents of a location, the location must first be opened, the new contents are then entered, and finally a command is given to close the location.

```
*1000/012746 012345)
*
```

In the example above, location 1000 now contains 012345. The location is closed since the RETURN key was typed after entering the new contents.

Used alone, the slash reopens the last location opened. For example:

```
*1000/012345 2340 )
*/002340
```

ODT changed the contents of location 1000 to 002340; the RETURN key instructed ODT to close the location before printing the *. The single slash command reopened the last location opened, allowing the user to verify that the word 002340 was correctly stored in location 1000.

Note that if an odd numbered address is specified using a slash, ODT opens the location as a byte, and subsequently behaves as though a backslash had been typed, as explained next.

The Backslash, \

In addition to operating on words, ODT may operate on bytes. One way to open a byte is to type the address of the byte followed by a backslash. (\ is printed by typing a SHIFT/L if using an LT33 or 35). ODT not only causes the byte value at the specified address to be printed, but also interprets the value as ASCII code and prints the corresponding character (if possible) on the terminal. For example:

```
*1001\101 =A
```

A backslash typed alone reopens the last byte opened. If a word was previously open, the backslash reopens its even byte.

```
*1002/000004 \004 =
```

The LINE FEED Key

If the LINE FEED key is typed when a location is open, ODT closes the open location and opens the next sequential location:

```
*1000/002340↓          ( ↓ denotes typing the LINE
001002 /012740          FEED key)
```

In this example, the LINE FEED key caused ODT to print the address of the next location along with its contents, and to wait for further instructions. Location 1000 is automaticaly closed by ODT and 1002 is opened. The open location may be modified by typing new contents.

If a byte location is open, typing the LINE FEED key opens the next byte location.

The Up-Arrow, ↑ or ^

If an up-arrow (or circumflex) symbol is typed when a location is open (an up-arrow is produced by typing a SHIFT/N on an LT33 or 35), ODT closes the open location and opens the previous location. To continue from the example above:

```
*001002/012740 ↑
001000 /002340
```

Now location 1002 is closed and 1000 is open. The open location may be modified by entering new contents.

If a byte location is open, then up-arrow opens the previous byte.

The Back-Arrow, ← or _

If the back-arrow (or underline) symbol (produced by typing SHIFT/O on a LT33 or 35) is typed when a location is open, ODT interprets the contents of the currently open word as an address indexed by the Program Counter (PC) and opens the location so addressed:

```
*1006/000006 ←
001016 /100405
```

Notice in this example that the open location, 1006, was indexed by the PC as if it were the operand of an instruction with address mode 67 as explained in Chapter 5.

Modification to the opened location may be made before either a line feed, up-arrow, or back-arrow is typed. Also, the new contents of the location will be used for address calculations when using the back-arrow command. For example:

```
*100/000222 4↓          (modify to 4 and open next location)
000102 /000111 6↑       (modify to 6 and open previous location)
000100 /000004 100←     (change to 100 and open the location
000202 /(contents)       indexed by PC)
```

Open the Addressed Location, @

The symbol @ (SHIFT/P on an LT33 or 35) may be used to optionally modify a location, close it, and then use its contents as the address of the location to open next.

```
*1006/001024 @          (open location 1024 next)
001024 /000500

*1006/001024 2100@      (modify to 2100 and open
002100 /177774           location 2100)
```

Relative Branch Offset, >

The right angle bracket (>) allows the user to optionally modify a location, close it, and then use its low-order byte as a relative branch offset to the next word to be opened. For example:

```
*1032/000407 301>     (modify to 301 and interpret as a
000636 /000010         relative branch)
```

Note that 301 is a negative offset (-77). The offset is doubled before it is added to the PC; therefore, 1034+(-176)=636.

Return to Previous Sequence, <

The left angle bracket (<) allows the user to optionally modify a location, close it, and then open the next location of a previous sequence which was interrupted when either a back-arrow, @ sign, or right angle bracket command was used. (As already mentioned, ↑, @, and > each cause a sequence change determined by the contents of the open location. If a sequence change has not occurred, the left angle bracket simply opens the next location as though using a line feed). This command operates on both words and bytes. For example:

```
*1032/000407 301>          (> causes a sequence change)
000636 /000010 <           (return to original sequence)
001034 /001040 @           (@ causes a sequence change)
001040 /000405 \005 = <    (< now operates on byte)
001035 \002 = <            (< acts like ↓ )
001036 \004 =
```

7.3.3 Accessing General Registers 0-7

The program's general registers 0-7 are opened using the following command format:

```
*$n/
```

where n is an integer in the range 0 through 7 and represents the desired register. When opened, these registers can be examined or changed in the same manner as any addressable location. For example:

```
*$0/0000033 )          (R0 was examined and closed)
*

*$4/000474 464)        (R4 was opened, changed, closed,)
*/000464               (and verified)
```

The ↓, ↑, ↑, or @ commands may be used whenever a register is open.

7.3.4 Accessing Internal Registers

The program's Status Register contains the condition codes of the most recent operational results and the interrupt priority level of the object program. The address of this register is accessed by typing $S. For example:

```
*$S/000311
```

In response to $S/ in the example above, ODT printed the 16-bit word of which only the low-order 8 bits are meaningful: Bits 0-3 indicate whether a carry, overflow, zero, or negative (in that order) value has resulted, and bits 5-7 indicate the interrupt priority level (in the range 0-7) of the object program. (Refer to the PDP-11 PROCESSOR HANDBOOK for the Status Register format.)

Table 7-2 lists internal registers which may be opened using the $ format.

Table 7-2
Internal Registers

Register	Function
$B	Location of the first word of the breakpoint table (see Section 7.3.6).
$M	Mask location for specifying which bits are to be examined during a bit pattern search (see Section 7.3.9).
$P	Location defining the operating priority of ODT (see Section 7.3.15).
$S	Location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7); (explained above).
$C	Location of the Constant Register (see Section 7.3.10).
$R	Location of Relocation Register 0, the base of the Relocation Register Table (see Section 7.3.13).
$F	Location of Format Register (explained in Section 7.3.1).

7.3.5 Radix 50 Mode, X

The Radix 50 mode of packing certain ASCII characters three to a word is employed by many DEC-supplied PDP-11 system programs and may be employed by any programmer using the CAPS-11 Assembler's .RAD50 directive. ODT allows a method for examining and changing memory words packed in this way by providing the X command. If the X command is typed when a location is open, ODT converts the contents of the opened word to its 3-character Radix 50 equivalent and prints these characters on the terminal. One of the responses in Table 7-3 may then be typed:

Table 7-3
Radix 50 Terminators

Response	Effect
RETURN key	Close the currently open location
LINE FEED key	Close the currently open location and open the next one in sequence
Up-Arrow key	Close the currently open location and open the previous one in sequence
Any three characters whose octal code is 040 (space) or greater	Convert the three specified characters into packed Radix 50 format

Legal Radix 50 characters for this last response are:

. $ Space 0-9 A-Z

If any other character is typed, the resulting binary number is unspecified (that is, no error message is printed and the result is unpredictable). Exactly three characters must be typed before ODT resumes its normal mode of operation. After the third character is typed, the resulting binary number may be stored in the opened location by closing the location in any one of the ways listed in Table 7-3. For example:

```
*1000/042431 X=KBI CBA)
*1000/011421 X=CBA
```

WARNING

After ODT has converted the three characters to binary, the binary number can be interpreted in one of many different ways depending on the command which follows. For example:

```
*1234/063337 X=PRO XIT/004004
```

Since the Radix 50 equivalent of XIT is 113574, the final slash in the example causes ODT to open location 113574 and type out its contents if it is a legal address. (Refer to Sections 7.4 and 7.5 for a discussion of command legality and detection of errors.)

7.3.6 Breakpoints

The breakpoint feature allows the user to monitor the progress of program execution. A breakpoint may be set at any instruction which is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a trap instruction so that program execution is suspended when the breakpoint is encountered. The original contents of the breakpoint location are then restored and ODT regains control.

As many as eight breakpoints numbered 0 through 7 can be set at any one time. A breakpoint is set by typing the address of the desired location of the breakpoint followed by ;B. Thus n;B will set the next available breakpoint (from 0-7) at address n. Specific breakpoints may be set or changed by the n;mB command where m is the number of the breakpoint. For example:

```
*1020;B          (set breakpoint 0 at address 1020)
*1030;B          (set breakpoint 1 at address 1030)
*1040;B          (set breakpoint 2 at address 1040)
*1032;1B         (reset breakpoint 1 at address 1032)
*
```

The ;B command without an argument removes all breakpoints. The ;mB command is used to remove only one of the breakpoints, where m is the number of the breakpoint. For example:

```
*;2B        (remove breakpoint 2)
*
```

A table of breakpoints is kept by ODT and may be accessed by the user. The $B/ command opens the location containing the address of breakpoint 0. The next seven locations (represented as nnnnnn) contain the addresses of the other breakpoints in order, and can be sequentially opened by using the LINE FEED key. For example:

```
*$B/001020↓
 nnnnnn/001032↓
 nnnnnn/(address internal to ODT)
```

In this example breakpoint 2 is not set. The contents printed by ODT represents an internal address and can be determined by checking the Linker Load Map (see Chapter 6).

7.3.7 Running the Program

Program execution is under control of ODT. There are two commands for running the program: n;G and n;P. The n;G command is used to start execution (Go) and n;P to continue execution (Proceed) after halting at a breakpoint. For example:

```
*1000;G
```

This causes execution to start at location 1000. The program will run until a breakpoint is encountered or until program completion. If the program enters an infinite loop, it must be either restarted or reentered as explained in Section 7.4.2.

Upon execution of either the n;G or n;P command, the general registers 0-6 are set to the values in the locations specified as $0-$6 and the processor Status Register is set to the value in the location specified as $S.

When a breakpoint is encountered, execution stops and ODT prints Bn; (where n represents the breakpoint number) followed by the address of the breakpoint. Locations can then be examined for expected data. For example:

```
*1010;3B        (breakpoint 3 is set at location 1010)
*1000;G         (execution is started at location 1000)
B3;001010       (execution is stopped at location 1010)
*
```

To continue program execution from the breakpoint, type ;P in response to ODT's last (*).

When a breakpoint is set in a loop, it may be desirable to allow the program to execute a certain number of times through the loop before recognizing the breakpoint. This is done by setting a proceed count using the n;P command. This command allows the user to specify the number of times the breakpoint is to be encountered before program execution is suspended (execution will be suspended on the nth encounter). The count, n, refers only to the numbered breakpoint which most recently occurred. A different proceed count may be specified for the breakpoint when it is encountered. For example:

```
B3;001010       (execution halted at breakpoint 3)
*1250;B         (reset breakpoint 3 at location 1250)
*4;P            (set proceed count to 4 and continue
B3;001250        execution; loop through breakpoint
*                three times and halt on fourth
                 occurrence of the breakpoint)
```

Proceed counts for other breakpoints may be reset by accessing the table of proceed counts, explained next.

Following the table of breakpoints (as explained in Section 7.3.6) is a table of the proceed command repeat counts for each breakpoint. These repeat counts can be inspected by typing $B/ followed by typing nine LINE FEED's. The repeat count for breakpoint 0 is printed (the first seven line feeds cause the table of breakpoints to be printed; the eighth types the Single-instruction mode, explained in the next section, and the ninth line feed begins the table of proceed command repeat counts). The repeat counts for breakpoints 1 through 7 and the repeat count for the single instruction trap follow in sequence (see Section 7.3.8). Before a proceed count is assigned a value by the user, it is set to 0; after the count has been executed, it is set to -1. Opening any one of these locations provides an alternate way of changing the count, as the location, once open, can have its contents modified in the usual manner (by typing the new contents and then the RETURN key). For example:

```
.
.
.
nnnnnn/001036 ↓      (address of breakpoint 7)
nnnnnn/nnnnnn ↓      (single instruction address)
nnnnnn/000000 15 ↓   (count for breakpoint 0 is
                      changed to 15)
nnnnnn/000000 ↓      (count for breakpoint 1)
.
.
nnnnnn/000000 ↓      (count for breakpoint 7)
nnnnnn/nnnnnn        (repeat count for single
                      instruction mode. The
                      single instruction address
                      will be an address internal
                      to the user program if
                      single instruction mode
                      is used.)
```

The address indicated as the single-instruction address and the repeat count for single instruction mode are explained next.

7.3.8 Single-Instruction Mode

Using this mode the programmer can specify the number of instructions to be executed before suspension of the program run. The Proceed command, instead of specifying a repeat count for a breakpoint encounter, specifies the number of succeeding instructions to be executed. Breakpoints are disabled when single-instruction mode is operative.

Commands for single-instruction mode are:

;nS	Enable single-instruction mode (n can have any value and serves only to distinguish this form from the form ;S). Breakpoints are disabled.
n;P	Proceed with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1. Trap instructions and associated handlers can affect the Proceed repeat count. See section 7.4.2).
;S	Disable single-instruction mode.

When the repeat count for single-instruction mode is exhausted and the program suspends execution, ODT prints:

```
B8;n
*
```

where n is the address of the next instruction to be executed. The $B breakpoint table contains this address following that of breakpoint 7. However, unlike the table entries for breakpoints 0-7, direct modification has no effect.

Similarly, following the repeat (proceed) count for breakpoint 7 is the repeat count for single-instruction mode. This table entry may be directly modified, and thus is an alternative way of setting the single-instruction mode repeat count. In such a case, ;P implies the argument set in the $B repeat count table rather than assuming 1.

7.3.9 Searches

With ODT all or any specified portion of memory can be searched for a specific bit pattern or for references to a specific location.

Word Search, n;W

Before initiating a word search, the mask and search limits must be specified. The location represented by $M is used to specify the mask of the search. $M/ opens the mask register. The next two sequential locations (opened by line feeds) contain the lower and upper limits of the search. Bits set to 1 in the mask are examined during the search; other bits are ignored. Then the search object and the initiating command are given using the n;W command where n is the search object. When a match is found (i.e., each bit set to 1 in the search object is set to 1 in the word being searched over the mask range), the matching word is printed. For example:

```
*$M/000000 177400↓        (test high-order eight bits)
nnnnnn /000000 1000↓      (set low address limit)
nnnnnn /000000 1040↓      (set high address limit)
*400;W                    (initiate word search)
001010 /000770
001034 /000404
*
```

In the above example, nnnnnn is an address internal to ODT; this location varies and is meaningful only for reference purposes. In the first line above, the slash was used to open $M which now contains 177400; the line feeds opened the next two sequential locations which now contain the upper and lower limits of the search.

In the search process an exclusive OR (XOR) is performed with the word currently being examined and the search object, and the result is ANDed to the mask. If this result is zero, a match has been found and is reported on the terminal. Note that if the mask is zero, all locations within the limits are printed.

Typing CTRL/U during a search printout terminates the search.

Effective Address Search, r;E

ODT provides a command to search for words which address a specified location. The mask register is opened only to gain access to the low and high limit registers. After specifying the search limits (as explained previously), the command n;E is typed (where n is the effective address) and the search is initiated.

Words which are either an absolute address (argument n itself), a relative address offset, or a relative branch to the effective address, are printed after their addresses. For example:

```
*$M/177400↓                    (open mask register only to gain
nnnnnn//001000 1010↓            access to search limits)
nnnnnn//001040 1060)
*1034;E                        (initiating search)
001016 /001006                 (relative branch)
001054 /002767                 (relative branch)
*1020;E                        (initiating a new search)
001022 /177774                 (relative address offset)
001030 /001020                 (absolute address)
*
```

Particular attention should be given to the reported references to the effective address, since a word may have the specified bit pattern of an effective address without actually being so used. ODT reports all possible references whether they are actually used as such or not.

Typing CTRL/U during a search printout terminates the search.

7.3.10 The Constant Register

It is often desirable to convert a relocatable address into its value after relocation or to convert a number into its two's complement, and then to store the converted value in one or more places in a program. The Constant Register provides a means of accomplishing this and other useful functions.

When n;C is typed, the relocatable expression n is evaluated to its six-digit octal value and is both printed on the terminal and stored in the Constant Register. The contents of the Constant Register may be called in subsequent relocatable expressions by typing the letter C. Examples are:

```
*-4432;C=173346                (The two's complement of 4432 is
*                               placed in the Constant Register)

*1000/001000 C                 (The contents of the Constant
*                               Register are stored in location 1000)

*1000;1R                       (Relocation register 1 is set to
*                               1000)

*1,4272;C=005272               (Relative location 4272 is reprinted
*                               as an absolute location and stored
                                in the Constant Register)
```

7.3.11 Memory Block Initialization

The Constant Register can be used in conjunction with the commands ;F and ;I to set a block of memory to a given value. While the most common value required is zero, other possibilities are plus one, minus one, ASCII space, etc.

When the command ;F is typed, ODT stores the contents of the Constant Register in successive memory words starting at the memory word address specified in the lower search limit and ending with the address specified in the upper search limit.

When the command ;I is typed, the low-order 8 bits in the Constant Register are stored in successive bytes of memory starting at the byte address specified in the lower search limit and ending with the byte address specified in the upper search limit.

For example, assume relocation register 1 contains 1000, 2 contains 2000, and 3 contains 3000. The following sequence sets word locations 1000-1776 to zero, and byte locations 2000-2777 to ASCII spaces.

```
*$M/000000↓                    (Open mask register to gain
                                access to search limits)
 nnnnnn/000000 1,0↓            (Set lower limit to 1000)
 nnnnnn/000000 2,-2)           (Set upper limit to 1776)
*0;C=000000                    (Constant Register set to zero)
*;F                            (Locations 1000-1776 set to zero)
*
*$M/000000↓
 nnnnnn/001000 2,0↓            (Set lower limit to 2000)
 nnnnnn/001776 3,-1)           (Set upper limit to 2777)
*40;C=000040                   (Constant Register set to 40--
*;I                             ASCII space)
*                              (Byte locations 2000-2777 are set
                                to value in low-order 8 bits of
                                Constant Register)
```

7.3.12 Calculating Offsets

Relative addressing and branching involve the use of an offset--the number of words or bytes forward or backward from the current location to the effective address. During the debugging session it may be necessary to change a relative address or branch reference by replacing one instruction offset with another. ODT calculates the offsets in response to the n;O command.

The command n;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address n. For example:

```
*346/000034 414;O 000044 022 22)
*/000022
```

In the example, location 346 is opened and the offsets from that location to location 414 are calculated and printed. The contents of location 346 are then changed to 22 (the 8-bit offset) and verified on the next line.

The 8-bit offset is printed only if it is in the range 128 (decimal) to 127 (decimal) and the 16-bit offset is even, as was the case above. For example, the offset of a relative branch is calculated and modified as follows:

```
*1034/103421 1034;O 177776 377 \021 = 377)
*/103777
```

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.

7.3.13 Relocation Register Commands

The use of the relocation registers has been defined in Section 7.2. At the beginning of a debugging session it is desirable to preset the registers to the relocation biases of those relocatable modules which will be receiving the most attention.

This can be done by typing the relocation bias followed by a semicolon and the specification of relocation registers, as follows:

```
r;nR
```

r may be any relocatable expression and n is an integer from 0 to 7. If n is omitted it is assumed to be 0. As an example:

```
*1000;5R        (Set relocation register 5 to 1000)
*5,100;5R       (Add 100 to the contents of
*                relocation register 5)
```

In certain uses programs may be relocated to an address below that at which they were assembled. This could occur with PIC coding which is moved without the use of the Linker. In this case the appropriate relocation bias would be the 2's complement of the actual downward displacement. One method for easily evaluating the bias and entering it in the relocation register is illustrated in the following example.

Assume the program was assembled at location 5000 and was moved to location 1000. Then the sequence:

```
*1000;1R
*1,-5000;1R
*
```

enters the 2's complement of 4000 in relocation register 1, as desired.

Relocation registers are initialized to -1, so that unwanted relocation registers never enter into the selection process when ODT searches for the most appropriate register.

To set a relocation register to -1, type ;nR. To set all relocation registers to -1, type ;R.

ODT maintains a table of relocation registers, beginning at the address specified by $R. Opening $R ($R/) opens relocation register 0. Successively typing the LINE FEED key opens the other relocation registers in sequence. When a relocation register is opened in this way, it may be modified just as any other memory location.

7.3.14 The Relocation Calculators

When a location has been opened, it is often desirable to relate the relocated address and the contents of the location back to their relocatable values. To calculate the relocatable address of the opened location relative to a particular relocation bias, type n!, where n specifies the relocation register. This calculator works with both opened bytes and words. If n is omitted, the relocation register whose contents are closest but less than or equal to the opened location is selected automatically by ODT. In the following example, assume that these conditions are fulfilled by relocation register 2, which contains 2000. To find the most likely module that a given opened byte is in, the user types:

```
*2500\011 = !=2,000500
```

Typing nR after opening a word causes ODT to print the octal number which equals the value of the contents of the opened location minus the contents of relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest but less than or equal to the contents of the opened location. For example, assume the relocation bias stored in relocation register 1 is 001234; then:

```
*1,500/024550 1R=1,023314
```

The value 23314 is the contents of 1,500, relative to the base 1234. An example of the use of both commands follows.

Assuming relocation register 1 contains 1000 and relocation register 2 contains 2000, then to calculate the relocatable address of location 3000 and its contents relative to 1000 and 2000, the following can be performed:

```
*3000/005670 1!=1,002000 2!=2,001000 1R=1,004670 2R=2,003670
```

7.3.15 ODT's Priority Level

$P represents a location in ODT which contains the priority level at which ODT operates. If $P contains the value 377, ODT operates at the priority level of the processor at the time ODT is entered. Otherwise $P may contain a value between 0 and 7 corresponding to the fixed priority at which ODT will operate.

To set ODT to the desired priority level, open $P. ODT prints the present contents, which may then be changed:

```
*$P/000006 377)
*
```

If $P is not specified, its value will be seven.

Breakpoints may be set in routines at different priority levels. For example, a program running at a low priority level may use a device service routine which operates at a higher priority level. If a breakpoint occurs from a low priority routine, if ODT operates at a low priority, and if an interrupt occurs from a high priority routine, then the breakpoints in the high priority routine will not be executed

since they have been removed when the low priority breakpoint occurred. That is, interrupts set at a priority higher than the one in which ODT is running will occur and any breakpoints will not be recognized.

For example:

```
*1000;B
*2000;B
*500;G
B0; 1000
*
```

If a higher level interrupt occurs while ODT is waiting for input, the interrupt will be serviced and no breakpoints will be recognized.

NOTE

If the user is debugging a program which utilizes double-buffered cassette I/O (especially in formatted modes), he may find it useful to set ODT's priority to 5. This will allow cassette flags to interrupt ODT but will lock out terminal printer, keyboard, and line printer interrupts. If this is not done and a breakpoint is encountered while cassette I/O is occurring, timing errors will occur.

7.3.16 ASCII Input and Output

ASCII text may be inspected and changed using the command:

```
r;nA
```

where r is a relocatable expression and n is a character count. If n is omitted it is assumed to be 1. ODT prints n characters starting at location r, followed by a carriage return/line feed. One of the following may then be typed:

RETURN	ODT outputs a carriage return/line feed and an asterisk and waits for another command.
LINE FEED	ODT opens the byte following the last byte output.
up to n characters of text	ODT inserts the text into memory starting at location r. If less than n characters are typed, terminate the command by typing CTRL/U, causing a carriage return/line feed and an asterisk to be output as for RETURN. However, if exactly n characters are typed, ODT responds with a carriage

return/line feed, the address of the next available byte and a carriage return/line feed/asterisk.

Note that n may actually be expressed as a relocatable expression and could accidently be quite large. There is no safeguard against this in ODT.

7.4 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, it does provide a better understanding of how ODT performs some of its functions; in certain difficult debugging situations, this understanding is necessary.

7.4.1 Functional Organization

The internal organization of ODT is almost totally modularized into independent subroutines. The internal structure consists of three major functions: command decoding, command execution, and various utility routines.

The command decoder interprets the individual commands, checks for command errors, saves input parameters for use in command execution, and sends control to the appropriate command execution routine.

The command execution routines take parameters saved by the command decoder and uses the utility routines to execute the specified command. Command execution routines exit either to the object program or back to the command decoder.

The utility routines are common routines such as SAVE-RESTORE and I/O. They are used by both the command decoder and the command executers.

7.4.2 Breakpoints

The function of a breakpoint is to give control to ODT whenever the user program tries to execute the instruction at the selected address. Upon encountering a breakpoint, all of the ODT commands can be used to examine and modify the program.

When a breakpoint is executed, ODT removes all breakpoint instructions from the user's code so that the locations may be examined and/or altered. ODT then types a message on the console terminal in the form Bm;n where n is the breakpoint address (and m is the breakpoint number). The breakpoints are automatically restored when execution is resumed.

One restriction in the use of breakpoints follows: the word where a breakpoint has been set must not be referenced by the program in any way since ODT has altered the word. Also, no breakpoint should be set at the location of any instruction that clears the T-bit. For example:

```
MOV #240,177776          ;SET PRIORITY TO LEVEL 5
```

NOTE

Instructions that cause or return from traps (e.g., EMT, RTI) are likely to clear the T-bit, since a new word from the trap vector or the stack will be loaded into the Status Register.

A breakpoint occurs when a trace trap instruction (placed in the user program by ODT) is executed. When a breakpoint occurs, the following steps are taken:

1. Set processor priority to seven (automatically set by trap instruction).
2. Save registers and set up stack.
3. If internal T-bit trap flag is set, go to step 13.
4. Remove breakpoints.
5. Reset processor priority to ODT's priority or user's priority.
6. Make sure a breakpoint or single-instruction mode caused the interrupt.
7. If the breakpoint did not cause the interrupt, go to step 15.
8. Decrement repeat count.
9. Go to step 18 if non-zero; otherwise reset count to one.
10. Save console terminal status (refer to the section entitled 'Procedure for Saving and Restoring Console Terminal Status' below).
11. Type message about the breakpoint or single-instruction mode interrupt.
12. Go to command decoder.
13. Clear T-bit in stack and internal T-bit flag.
14. Jump to the Go processor.
15. Save console terminal status.
16. Type BE (Bad Entry) followed by the address.
17. Clear the T-bit, if set, in the user status and proceed to the command decoder.
18. Go to the Proceed processor, bypassing the console terminal restore routine.

Note that steps 1-5 inclusive take approximately 100 microseconds during which time interrupts are not permitted to occur (ODT is running at level 7).

When a proceed (;P) command is given, the following occurs:

1. The proceed is checked for legality.
2. The processor priority is set to seven.
3. The T-bit flags (internal and user status) are set.
4. The user registers, status, and Program Counter are restored.
5. Control is returned to the user.
6. When the T-bit trap occurs, steps 1, 2, 3, 13, and 14 of the breakpoint sequence are executed, breakpoints are restored, and program execution resumes normally.

When a breakpoint is placed on an IOT, EMT, TRAP, or any instruction causing a trap, the following occurs:

1. When the breakpoint occurs as described above, ODT is entered.
2. When ;P is typed, the T-bit is set and an IOT, EMT, TRAP, or other trapping instruction is executed.
3. This causes the current PC and status (with the T-bit included) to be pushed on the stack.
4. The new PC and status (no T-bit set) are obtained from the respective trap vector.
5. The whole trap service routine is executed without any breakpoints.
6. When an RTI is executed, the saved PC and PS (including the T-bit) are restored. The instruction following the trap-causing instruction is executed. If this instruction is not another trap-causing instruction, the T-bit trap occurs, causing the breakpoints to be reinserted in the user program, or the single-instruction mode repeat count to be decremented. If the following instruction is a trap-causing instruction, this sequence is repeated starting at step 3.

NOTE

Exit from the trap handler must be via the RTI instruction, otherwise the T-bit is lost. ODT cannot gain control again since the breakpoints have not yet been reinserted.

Note that the ;P command is illegal if a breakpoint has not occurred (ODT responds with ?); ;P is legal, however, after any trace trap entry.

The internal breakpoint status words have the following format:

1. The first eight words contain the breakpoint addresses for breakpoints 0-7. (The ninth word contains the address of the next instruction to be executed in single-instruction mode.)

2. The next eight words contain the respective repeat counts. The following word contains the repeat count for single-instruction mode.)

These words may be changed at will, either by using the breakpoint commands or by direct manipulation with $B.

When program runaway occurs (that is, when the program is no longer under ODT control, perhaps executing an unexpected part of the program where a breakpoint has not been placed) ODT may be given control as follows:

1. Press the HALT key to stop the computer.

2. If ODT was linked with the user's program, start ODT at any one of these addresses:

   a) Its entry address (contents of locations where breakpoints were set are not restored to their original contents).

   b) Its entry address + 2 (contents of locations where breakpoints were set are restored; all breakpoints are removed and all relocation registers are cleared).

   c) Its entry address + 4 (simulates a breakpoint).

3. If ODT was not linked with the user's program, but the user executed a LOAD/G ODT or a .R ODT, the entry address of ODT in an 8K system is 14000. One of the restart addresses in 2 above may then be used.

ODT prints an (*) indicating that it is ready to accept a command.

If the program being debugged uses the teleprinter for input or output, the program may interact with ODT to cause an error since ODT uses the teleprinter as well. This interactive error will not occur when the program being debugged is run without ODT.

1. If the teleprinter interrupt is enabled upon entry to the ODT break routine and no output interrupt is pending when ODT is entered, ODT generates an unexpected interrupt when returning control to the program.

2. If the interrupt of the teleprinter reader (the keyboard) is enabled upon entry to the ODT break routine and the program is expecting to receive an interrupt to input a character, both the expected interrupt and the character are lost.

3. If the teleprinter reader (keyboard) has just read a character into the reader data buffer when the ODT break routine is entered, the expected character in the reader data buffer is lost.

Procedure for Saving and Restoring Console Terminal Status

Upon entering the console terminal SAVE routine, the following occurs:

1. Save the console terminal keyboard status register (TKS).

2. Clear interrupt enable and maintenance bits in the TKS.

3. Save the console terminal printer status register (TPS).

4. Clear interrupt enable and maintenance bits in the TPS.

To restore the console terminal status:

1. Wait for completion of any I/O from ODT.

2. Restore the TKS.

3. Restore the TPS.

WARNINGS

If the console terminal printer interrupt is enabled upon entry to the ODT break routine, the following may occur:

1. If no output interrupt is pending when ODT is entered, an additional interrupt always occurs when ODT returns control to the user.

2. If an output interrupt is pending upon entry, the expected interrupt occurs when the user regains control.

If the teleprinter keyboard is busy or done, the expected character in the reader data buffer is lost.

If the teleprinter keyboard interrupt is enabled upon entry to the ODT break routine, and a character is pending, the interrupt (as well as the character) is lost.

7.4.3 Searches

The word search allows the user to search for bit patterns in specified sections of memory. Using the $M/ command, the user specifies a mask, a lower search limit ($M+2), and an upper search limit ($M+4). The search object is specified in the search command itself.

The word search compares selected bits (where ones appear in the mask) in the word and search object. If all of the selected bits are equal, the unmasked word is printed.

The search algorithm is:

1. Fetch a word at the current address.
2. XOR (exclusive OR) the word and search object.
3. AND the result of step 2 with the mask.
4. If the result of step 3 is zero, type the address of the unmasked word and its contents. Otherwise, proceed to step 5.
5. Add two to the current address. If the current address is greater than the upper limit, type * and return to the command decoder, otherwise go to step 1.

Note that if the mask is zero, ODT prints every word between the limits, since a match occurs every time (i.e., the result of step 3 is always zero).

In the effective address search, ODT interprets every word in the search range as an instruction which is interrogated for a possible direct relationship to the search object. The mask register is opened only to gain access to the search limit registers.

The algorithm for the effective address search (where X denotes the contents of X, and K denotes the search object) is:

1. Fetch a word at the current address X.
2. If (X)=K [direct reference], print contents and go to step 5.
3. If (X)+X+2=K [indexed by PC], print contents and go to step 5.
4. If (X) is a relative branch to K, print contents.
5. Add two to the current address. If the current address is greater than the upper limit, perform a carriage return/line feed and return to the command decoder; otherwise, go to step 1.

7.5 ERROR DETECTION

ODT detects two types of error: illegal or unrecognizable command and bad breakpoint entry. ODT does not check for the legality of an address when commanded to open a location for examination or modification. Thus the command:

```
177774/
```

references nonexistent memory, thereby causing a trap through the vector at location 4. RESMON sets location 4 to produce the message:

```
%TRAP nnnnnn
```

However, if the user program modifies location 4 or 6, the results of such a trap are unpredictable.

Similarly, a command such as:

```
$20/
```

which references an address eight times the value represented by $2, may cause an illegal (nonexistent) memory reference.

Typing something other than a legal command causes ODT to ignore the command, print:

```
(echoes illegal command)?
*
```

and wait for another command. Therefore, to cause ODT to ignore a command just typed, type any illegal character (such as 9 or RUBOUT) and the command will be treated as an error and ignored.

ODT suspends program execution whenever it encounters a breakpoint by trapping to its breakpoint routine. If the breakpoint routine is entered and no known breakpoint caused the entry, ODT prints:

```
BE001542
*
```

and waits for another command. In the example above, BE001542 denotes Bad Entry from location 001542. A bad entry may be caused by an illegal trace trap instruction, setting the T-bit in the status register, or by a jump to the middle of ODT.

7.6 EXAMPLE USING ODT

The user has a program which he has assembled with PAL to produce a listing. He wishes to run the program under ODT to demonstrate the use of breakpoints:

```
CAPS11 PAL V01 05/25/73 PAGE    001


                        ;
                        ;PROGRAM TO DEMONSTRATE ODT
                        ;
                000001  R1=%1
                000002  R2=%2
                000000  R0=%0
                000003  R3=%3
                000006  SP=%6
                000000  .ASECT
                001000  .=1000
  001000 012706 000600  START:  MOV     #600,SP ;SET STACK PTR
  001004 012703 000002          MOV     #2,R3   ;SET LARGE LOOP COUNTER
  001010 012700 000200  LOOP2:  MOV     #200,R0
  001014 005001                 CLR     R1
  001016 005201         LOOP1:  INC     R1      ;INCREMENT R1 FROM 0 TO 200
  001020 005300                 DEC     R0      ;DECREMENT R0 FROM 200 TO 0
  001022 001375                 BNE     LOOP1   ;NOT DONE SMALL LOOP YET
  001024 005302                 DEC     R2      ;RESTART SMALL LOOP
  001026 001370                 BNE     LOOP2   ;IF R3 NOT ZERO YET
  001030 000000                 HALT
         001000                 .END    START

CAPS11 PAL V01 05/25/73 PAGE    002

 LOOP1     001016      LOOP2     001010      R0      =%000000
 R1      =%000001      R2      =%000002      R3      =%000003
 SP      =%000006      START     001000      .       = 001032



000000 ERRORS
```

The program is stored on cassette drive 1 as TESODT.LDA, and is loaded into memory using the LOAD command:

```
.L 1:TESODT.LDA
```

ODT is then called. The debugging process follows (NNNNNN represents an address internal to ODT).

```
   ODT V00
A  *1022;0B
   *1024;B
   *$B/001022
B  NNNNNN /001024
   NNNNNN /117776
   NNNNNN /117776
   *1000;G
   B0;001022
   *$6/000600
   *$3/000002
   *#?
C  *$0/000177
   *$1/000001
   *$B/001022
```

```
  NNNNNN /001024
D *100;P
  B0;001022
  *$3/000002
  *$0/000077
  *$1/000101
  *$B/001022
  NNNNNN /001024
E *76;P
  B0;001022
  *$3/000002
  *$1/000177
  *$0/000001
F *;P
  B0;001022
  *$0/000000
  *$1/000200
  *$3/000002
  *1024/005302 5303
G *;P
  B1;001024
  *$3/000002
  *$B/001022
  NNNNNN /001024
H *1026;B
  *$B/001022
  NNNNNN /001024
  NNNNNN /001026
  NNNNNN /117776
  *$S/000004
I *;P
  B2;001026
  *$3/000001
  *$S/000000
  *$0/000000
  *$1/000200
J *;P
  B0;001022
  *$0/000177
  *$1/000001
  *$3/000001
K *200;P
  B1;001024
  *$0/000000
  *$1/000200
  *$3/000001
  *$S/000004
L *;P
  B2;001026
  *$S/000004
  *$3/000000
  *1030;B
M *;P
  B3;001030
  *↑C
  .
```

A Set breakpoint 0 within the small loop; set the next available breakpoint (1) within the large loop.

B Examine ODT's breakpoint table - 0 and 1 are properly assigned; start the program.

C Breakpoint 0 is encountered; registers 3 and 6 are examined. An illegal command (#) is typed, which ODT answers with a ?. Registers 0 and 1 and the ODT breakpoint table are examined.

D Proceed through 100(octal) occurrences of breakpoint 0; examine the registers--3 is unchanged, 0 and 1 are decrementing and incrementing properly.

E Proceed through 76 more occurrences of breakpoint 0; the registers are examined and seem correct.

F Proceed from breakpoint 0. The small loop has finished, but the instruction at location 1024 is incorrect (it should be DEC R3); it is corrected.

G Execution proceeds; breakpoint 1 is encountered

H The next available breakpoint (which is breakpoint 2) is set as location 1026, the user status Z bit has been set.

I Breakpoint 2 is encountered; register 3 has been decremented; the Z bit is clear so the branch to loop 2 will be taken.

J Breakpoint 0 is encountered; registers 0 and 1 have been reset.

K Continue through all iterations of the small loop. Breakpoint 1 is encountered. Register 3 contains 1; the user status Z bit is set.

L Breakpoint 2 is encountered; the Z bit is still set and register 3 contains 0; program execution will fall through the branch. A breakpoint is set at the HALT instruction.

M The breakpoint at the HALT is encountered. A ↑C is typed to return to the Keyboard Monitor.

CHAPTER 8

PERIPHERAL INTERCHANGE PROGRAM

The Peripheral Interchange Program (PIP) provides the user with a means of transfering files between any of the permanent devices which are available on his system (as listed in Table 3-2) including the high-speed reader and punch. In addition, PIP provides the capibility for deleting files from a cassette, zeroing a cassette, and making multiple copies of a cassette.

8.1 CALLING AND USING PIP

PIP is called from the System Cassette by typing:

```
.R PIP
```

in response to the dot printed by the Keyboard Listener. The Command String Interpreter responds by printing an asterisk (*) when it is ready to accept input/output specifications. The user may enter his command string even though the remainder of PIP is being loaded into memory simultaneously.

Control is returned to PIP after each execution of an I/O command string.

8.1.1 PIP Options

The options listed in Table 8-1 may be used by PIP with the following results:

Table 8-1
PIP Options

Option	Meaning
/A	Used with an output filename to designate that the header bit be set to ASCII (the file type is otherwise assumed to be binary). If a file is transfered from the paper tape reader to cassette using the /A option, a ↑Z character (designating end-of-file) is automatically appended to the end of the file.
/C	Allows the command string to be broken into one or more lines.
/D	Causes the filename(s) indicated in the command line to be deleted from the specified cassette.
/P	Requests that the system prompt the user to change cassettes on the indicated drive before an attempt is made to access the file. The system prints: #? where # represents the number of the appropriate drive. When the user has mounted the proper cassette, he may type any character on the keyboard to continue execution.
/Z	Indicates that all cassettes on the unit drives specified in the command line are to be zeroed.

PIP does not support the /O overflow option. File transfers must not exceed a single cassette.

8.1.2 Input and Output Specifications

PIP allows four basic operations: cassette zero, file deletion, cassette copy, and file transfer. No default extensions are assumed by PIP, so the user must be sure to always indicate extensions in his command line.

CASSETTE ZERO

The cassette zero function is provided in PIP to allow a user who is performing a series of PIP commands the option of zeroing a cassette without returning to the Keyboard Monitor (to use the ZERO command).

The form of the command is:

```
*[CT]#:/Z/OPT,...[CT]#:/OPT[=]
```

The device, if specified, must be cassette, so only the drive number need be entered; unit 0 is assumed if no number is indicated. Any number of cassettes may be indicated in the command line; the /Z option is necessary only once after the first cassette specification. The /C and /P options are optional, as is the I/O separator (=, <, or ←). The input field must be empty if a separator is used.

An example of use of the PIP zero function might be the following case in which the user wishes to zero several cassettes:

```
*0:/Z,1:,0:/P,1:,0:/P,1:=
```

/Z indicates that the PIP zero function is requested; the cassettes on units 0 and 1 are zeroed; the user is then prompted (via /P) to change cassettes; he mounts different cassettes on drives 0 and 1 and then types any character on the console terminal keyboard to continue execution. The newly mounted cassettes are also zeroed; again he is prompted to change cassettes, and so on.

FILE DELETION

File deletion is performed using a command line in the following form:

```
*[CT]#:FILE1.EXT/OPT,[CT]#:FILE2.EXT/OPT,...[=]
```

Cassette drives 0 and 1 are the only legal devices and drive 0 is the default device. Filenames are indicated only on the output (left) portion of the command line; the input portion of the command line must remain empty. Options allowed are /D, /C and /P; the /D option is necessary only once after the first file specification.

Any number of files may be indicated in the command string. Those files specified are then deleted from the cassette directory and are replaced by an *EMPTY header in the directory listing. If PIP detects that the sentinel file immediately follows an *EMPTY file, it will also delete that *EMPTY file from the directory. For example, assume the directory of cassette drive 0 is:

```
 21-MAR-73

COPSO  LDA 01-DEC-72
BLANKS DAT 21-MAR-73
SORT   LST 21-MAR-73
TORN   ASC 19-MAR-73
```

and the user types:

```
*BLANKS.DAT/D,TORN.ASC=
```

These two files will be deleted leaving the directory as follows:

```
21-MAR-73

COPSO  LDA 01-DEC-72
*EMPTY     --
SORT   LST 21-MAR-73
```

If more than one file exists on a cassette under the same filename, all files under that name will be deleted.

CASSETTE COPY

The PIP copy function is used to 'clean up' cassettes containing *EMPTY headers and to make multiple copies of a cassette. The form of the command string is:

```
*[CT]#:=[CT]#:/OPT
```

Since cassettes are the only legal devices, only the cassette number need be specified; cassette drive 0 is the default device. The only option allowed in the copy function is /C and only one input and one output device specification may be indicated. For example:

```
*CT1:=0:
```

The cassette on drive 1 is first zeroed, and the entire contents of cassette drive 0 are then copied to the cassette on drive 1, producing an exact copy of cassette 0. Dates are copied as they appear on the original cassette. This copy function of PIP is particularly useful in making multiple copies of the System Cassette.

FILE TRANSFER

A file transfer using PIP is initiated by a command in one of the following formats:

```
*DEV:FILENA.EXT/OPT=DEV:FILE1.EXT/OPT,...DEV:FILEn.EXT/OPT
```

or

```
*DEV:OUT1.EXT/OPT,...DEV:OUTn.EXT/OPT=DEV:IN1.EXT/OPT,.../C
,DEV:INn.EXT/OPT
```

DEV represents any of the legal permanent devices (listed in Table 3-2). Any number of input specifications are allowed. If only one output specification is indicated, all input files will be combined under the filename and/or device designated in the output field; the input files will be combined in the order in which they are listed in the command string. Otherwise, each input file must have a corresponding output filename and/or device, and transfers will be performed on a one-for-one basis. Options allowed in the output portion of the command line are /P, /A, and /C. Options allowed in the input portion are /P, /F and /C.

For example:

```
*LP:=1:ABC.DAT,0:FIRST.ASC,1:FINT.DAT/P
```

A listing is to be output on the line printer. First the file ABC.DAT on cassette drive 1 is output, then without interruption FIRST.ASC on drive 0, and finally FINT.DAT. Before FINT.DAT is output, the system pauses and prints:

```
1?
```

The user should make sure the correct cassette is mounted on drive 1 and then type any character on the keyboard. The listing will continue.

After each execution of a PIP command string, control returns to PIP; the Command String Interpreter prints an asterisk to indicate that it is ready to accept another PIP command string. The user might next enter a command line such as the following:

```
*LP:,1:AFT.DAT,SIGNA.PAL/A=0:AFT.DAT,AFT.DAT,SIGNA.PAL
```

This command transfers the file AFT.DAT to both the line printer and cassette drive 1, and then transfers SIGNA.PAL in ASCII mode to cassette drive 1. If the number of input files is not equal to the number of output specifications (providing there is more than one output specification), an error message is printed.

To return to the Monitor, type ↑C.

8.1.3 Restarting PIP

PIP is automatically restarted after each execution of a command line; the CSI prints an asterisk indicatng that the user can enter a new command. A CTRL/P typed during execution of a command will cause the current output file to be closed and control will be returned to the CSI.

8.2 ERROR MESSAGES

The following error messages can occur during incorrect usage of PIP:

Table 8-2
PIP Error Messages

Message	Meaning
?BAD TAPE ?BAD TAPE?	Hardware checksum error (may also be caused by READ operations initiated on a cassette which is positioned after the sentinel file); a question mark following the message indicates that the error is not fatal; the user may mount another cassette and type any character on the keyboard to continue execution.

(Continued on next page)

Table 8-2 (Cont.)
PIP Error Messages

Message	Meaning
?EOM	Indicates an out-of-paper condition for the line printer, console terminal, or paper tape punch.
?EXCESS INPUT FILES	The number of input files exceeds the number of output files (providing the number of output files is greater than one); this error occurs during use of the file transfer function.
?EXCESS OUTPUT FILES	The number of output files exceeds the number of input files; this error occurs during use of the file transfer function.
?FILE NOT FND?	The specified file was not found on the cassette indicated; the user may mount another cassette and type any character on the keyboard to continue the search.
?ILLEGAL DEVICE	An illegal device was indicated for the PIP function used.
?ILLEGAL INPUT LIST	An input list was indicated where not allowed (as when using the zero, delete, and copy functions), or an illegal command was entered.
?ILLEGAL OUTPUT LIST	An output list was indicated where not allowed (as when using the copy function).
?I/O CHAN CONFLICT	An attempt was made to open an input file on a cassette already open for output, or vice versa.
?NO FILE NAME	A filename was not indicated in a command line which required one.
?OFFLINE x	The cassette is not properly mounted on drive x. The user should correctly mount the cassette so that execution can continue.
?SWITCH ERROR 'x'?	An illegal switch was indicated in the command line, where 'x' represents the switch in error. The check is made for as many as 10 illegal switches in any one command line. Typing any character on the keyboard will cause PIP to ignore the switch and continue execution.

(Continued on next page)

Table 8-2 (Cont.)
PIP Error Messages

Message	Meaning
?TAPE FULL ?TAPE FULL?	Available space for an output file is full. A question mark following the message indicates that the error is not fatal; the user may mount another cassette and type any character on the keyboard to continue execution.
?WRT LOCK x	The cassette is write-locked; x represents the drive number. The user should dismount the cassette (the OFFLINE error message will then be printed), write-enable the cassette, and remount it. Execution will continue.

CHAPTER 9

INPUT/OUTPUT PROGRAMMING

The majority of I/O in the CAPS-11 System is done using RESMON, the part of the Monitor which contains routines to handle all file structured cassette I/O and all teleprinter, keyboard and line printer input and output.

RESMON is brought into memory by bootstrapping the system or by typing a CTRL/C (↑C) while running another system program. RESMON loads the following interrupt and trap vectors: console terminal keyboard and printer, line printer, cassette, timeout, breakpoint, illegal memory reference, stack overflow, power fail, EMT, TRAP and IOT. The RESMON I/O handlers remain in memory unless the user does an overlay load (using the Monitor LOAD command; see Chapter 3).

Simple I/O requests can be made by specifying devices and data forms for interrupt-controlled data transfers. These requests can be occurring concurrently with the execution of a running user program; multiple I/O devices may be running single or double buffered I/O processing simultaneously.

9.1 COMMUNICATING WITH RESMON

RESMON commands can be divided into two categories:

1. Those concerned with establishing necessary conditions for performing input and output, and

2. Those concerned directly with the transfer of data.

When transfer of data is occurring, RESMON is operating at the priority level of the device. The calling program runs at its own priority level, either concurrently with the data transfer, or sequentially. Before using data transfer commands, note the following:

1. Device specifications are made by referencing device numbers. Devices and their corresponding numbers are listed in Table 9-1.

2. The buffer, whose address is specified in the code, in most cases must be set up with information about the data.

In non-data transfer commands where an address or device number does not apply, the device number should be set to zero; the address is ignored by RESMON and may be any number. Addresses or codes may be specified symbolically.

Communication with RESMON is accomplished by IOT (Input/Output Trap) instructions in the user's program. Each IOT is followed by two words consisting of one of the RESMON commands and its operands in the following format:

```
IOT
.BYTE (command code),(device #)
.WORD (address)
```

As an example, the following program segment illustrates a simple input-process-output sequence. It includes the setting up of a single buffer, a formatted ASCII READ into the buffer, a wait for completion of the READ, processing of data just read, and a WRITE command from the buffer. (RESMON commands used in this example are explained in detail later in the chapter.)

```
            000001          RESET=1          ;ASSIGN RESMON COMMAND
            000005          READ=5           ;CODES
            000003          WAITR=3
            000004          WRITE=4

000000 000004 START:    IOT              ;ISSUE RESET IOT
000002     001          .BYTE RESET,0
000003     000
000004 000000           .WORD 0

000006 000004 KREAD:    IOT              ;TRAP TO RESMON
000010     005          .BYTE READ,3     ;SPECIFY BUFFER AND READ
000011     003
000012 000130'          .WORD BUFFER     ;FROM KBD (DEVICE 3) UNTIL
                                         ;LINE FEED OR FORM FEED

000014 000004 WAIT:     IOT              ;TRAP TO RESMON
000016     003          .BYTE WAITR,3    ;WAIT FOR KBD (DEVICE 3)
000017     003
                                         ;TO FINISH
000020 000014'          .WORD WAIT       ;BUSY RETURN ADDRESS WHILE
                  .                      ;WAITING FOR KBD TO FINISH
              (process buffer)
                  .
                  .
000122 000004           IOT              ;TRAP TO RESMON
000124     004          .BYTE WRITE,2    ;WRITE TO TELEPRINTER
000125     002
000126 000130'          .WORD BUFFER     ;(DEVICE 2), SPECIFY BUFFER

000130 000100 BUFFER:   100              ;BUFFER SIZE IN BYTES
000132 000000           0                ;CODE FOR FORMATTED ASCII
000134 000000           0                ;MODE, RESMON WILL SET HERE
                                         ;THE NUMBER OF BYTES READ
            000236      .=.+100          ;STORAGE RESERVED FOR 100
                                         ;BYTES
            000000      .END START
```

In more complex programming it is likely that more than one buffer will be set up for the transfer of data, so that data processing can occur concurrently rather than sequentially, as here.

9.2 DEVICE ASSIGNMENTS

I/O devices in the CAPS-11 System are fixed. The programmer references them by using RESMON and specifying a device number from Table 9-1. The device assignment numbers are:

Table 9-1
Device Assignments

Device	Number
Cassette Drive 0	0
Cassette Drive 1	1
Console Terminal Printer	2
Console Terminal Keyboard	3
Line Printer	4

Thus, in the following example:

```
IOT
.BYTE READ,1
.WORD STORE
```

data is read from device 1, which is cassette drive 1.

9.3 BUFFER ARRANGEMENT IN DATA TRANSFER COMMANDS

Use of the data transfer commands (READ and WRITE) requires the setting up of at least one buffer. This buffer is used not only to store data for processing, but to hold information regarding the quantity, form, and status of the data. All formatted I/O and all unformatted I/O (excluding unformatted cassette I/O) use one type of buffer; unformatted cassette I/O requires a special buffer.

9.3.1 Buffer Arrangement for Formatted I/O and Unformatted I/O (Excluding Cassette)

The buffer area for all I/O except unformatted cassette consists of two sections: the buffer header and the buffer itself. The non-data portion of the buffer is called the buffer header and precedes the data portion. In data transfer commands, the address of the first word of the buffer header is specified in the second word after the IOT command.

NOTE

RESMON uses the buffer header while transferring data. The user's program must not change or reference it (other than to check status bits).

The arrangement of the buffer is as follows:

BUFFER SIZE (in Bytes)	
STATUS	MODE
BYTE COUNT	
DATA • •	

Buffer Size

The first word of the buffer contains the maximum size (in bytes) of the data portion of the buffer and is specified by the user as an unsigned integer. RESMON will not store more than this many data bytes on input. Buffer size has no meaning on output.

Mode Byte

The low-order byte of the second word holds information concerning the mode of transfer. A choice of four modes exists:

Mode	Coded as:	
Formatted ASCII	000	(or 200 to suppress echo)
Formatted Binary	001	
Unformatted ASCII	002	(or 202 to suppress echo)
Unformatted Binary	003	

The term echo applies only to the console terminal keyboard. Data transfers from other devices never involve an echo. A diagram illustrates the format of the Mode Byte:

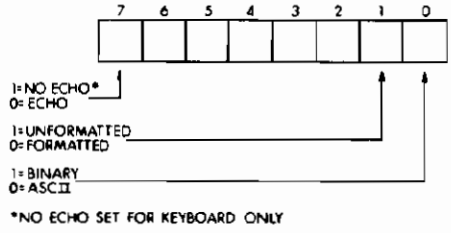


Figure 9-1 Mode Byte

Modes are further discussed in Section 9.4.

Status Byte

The high-order byte of the second word of the buffer header contains information set by RESMON on the status of the data transfer as follows:

Bits 0-4	Contain the non-fatal error codes (coded octally; see Table 9-2)
Bit 5	1 = End-Of-File has occurred (attempt at reading data after an End-Of-Medium)
Bit 6	1 = End-of-Medium has occurred
Bit 7	1 = Done (Data Transfer complete)

Thus, this byte is set up as follows:

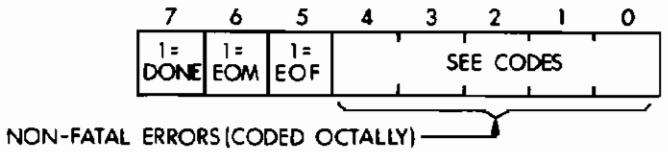


Figure 9-2 Status Byte

Non-fatal error codes for the Status Byte are described in Table 9-2.

Table 9-2
RESMON Non-Fatal Error Codes

Error Code	Meaning
1 = block check error	A block check error can occur on any cassette read (hard error; RESMON cannot read the block).
2 = checksum error	A checksum error can occur only on a formatted binary READ; (see Section 9.4.3).

(Continued on next page)

Table 9-2 (Cont.)
RESMON Non-Fatal Error Codes

Error Code	Meaning
3 = truncation of a long line	Truncation of a long line can occur on either a formatted binary or formatted ASCII READ (see Section 9.4.1). This error occurs when the binary block or ASCII line is bigger than the buffer size specified in the buffer header. In both cases, RESMON continues reading characters into the last byte in the buffer until the end of the binary block or ASCII line is encountered.
4 = improper mode	An improper mode can occur only on a formatted binary READ. Such occurrence means that the first non-null character encountered was not the proper starting character for a formatted binary block (see Section 9.4.3).

When the data transfer to or from the buffer is complete, the Done Bit (bit 7) is set by RESMON.

The following conditions cause the EOM Bit (bit 6) to be set in the Buffer Status byte. (An EOM occurrence also sets the Done Bit.)

Line Printer	Cassette
No paper	↑Z detected during
No power	formatted ASCII input
Printer drum gate open	
Overtemperature condition	

An End-Of-Medium condition occurring during use of the line printer is cleared by a manual operation such as putting paper in the line printer. RESMON does not retain any record of an EOM.

When an End-Of-Medium has occurred during a READ from cassette, there may be data in the buffer. If an EOM has occurred during a WRITE to the printer, there is no way of knowing how much of the buffer was written.

The following conditions cause the EOF Bit (bit 5) and the Done Bit to be set in the Buffer Status byte:

1. File gap or clear trailer encountered during a READ from cassette.

2. Clear trailer encountered during a WRITE to cassette.

When an End-Of-File occurs during a READ, the byte count is set to reflect the amount of data actually read. When an EOF occurs during a WRITE, there is no way of determining how much of the buffer was actually written.

Byte Count

The third word of the buffer header contains the Byte Count determined as follows:

Type of Transfer	Action
Input:	During unformatted transfers from the keyboard, RESMON reads as many data bytes as the user has specified. During formatted transfers from cassette or keyboard, RESMON inserts in this location the number of data bytes available in the buffer. During formatted ASCII mode from cassette, if an EOM or EOF occurs, RESMON will set the Byte Count equal to the number of bytes actually read. See Section 9.3.2 for information concerning unformatted cassette input.
Output:	The Byte Count determines the number of bytes output for all modes. A line printer out-of-paper condition will also terminate output and EOM will be set in the Status Byte. RESMON does not modify the Byte Count on output.

9.3.2 Buffer Arrangement for Unformatted Cassette

The distinction between formatted and unformatted cassette I/O is made at the time a cassette file is opened for input or output (at SEEK or ENTER time--see Sections 9.6.1 and 9.6.3). The mode specified at that time governs the way subsequent READs or WRITEs are interpreted for the opened file. In the special case of unformatted I/O to or from cassette, the buffer pointer in the READ or WRITE IOT command is assumed to point to a 128 byte buffer without a buffer header, and not to a buffer as previously described. The buffer specifications for unformatted cassette I/O made at SEEK or ENTER time are ignored. During an unformatted READ from cassette, a 128 byte data block is read directly into the buffer indicated by the second word of the parameter block. (See Sections 9.7.1 and 9.7.2 for a description of the parameter block). During an unformatted WRITE to cassette, 128 bytes of data are taken directly from the buffer indicated and transferred to cassette.

9.4 MODES

Modes have already been mentioned in Section 9.3; following is a detailed description of each type.

9.4.1 Formatted ASCII

A formatted ASCII READ transfers 7-bit characters (bit 8 is zero) until a line feed or form feed is encountered. RESMON sets the Byte Count word in the buffer header to indicate the number of characters in the buffer. If the line is too long, characters are read and overlaid into the last byte of the buffer until an EOM or an end-of-line (indicated by a line feed or form feed) is detected. Thus, if there is no error, the buffer will always contain a line feed or form feed.

A formatted ASCII WRITE transfers the number of 7-bit characters specified by the buffer Byte Count. Bit 8 will always be output as zero.

Device-dependent functions for the console terminal keyboard and printer, line printer, and cassette follow.

Console Terminal Keyboard

Seven-bit characters read from the keyboard are entered in the buffer and are echoed on the console terminal with the following exceptions:

Null	- Ignored. This character is not echoed or transferred to the buffer.
Tab (CTRL/TAB keys)	- Echoes as spaces up to the next tab stop. "Stops" are located at every 8th carriage position.
RUBOUT	- Deletes the previous character on the current line and echoes the character deleted. If there are no characters to delete, RUBOUT is ignored.
CTRL/U	- Deletes the current line and echoes as ↑U followed by a carriage return/line feed.
Carriage Return (RETURN key)	- Echoes as a carriage return followed by a line feed. Both characters enter the buffer.
CTRL/C	- Echoes as ↑C followed by a carriage return/line feed and a "?". The user should make sure that the System Cassette is mounted on drive 0; typing any character in response to the "?" will reboot the system. (If the system is intact in memory, no "?" is printed since no reboot is necessary--the KBL is merely restarted.)
CTRL/O	- Echoes as ↑O followed by a CR/LF. Console terminal output is supressed until either:

1. ↑O is typed again, which causes teleprinter output to be resumed;

2. The program which is executing requests keyboard input;

3. The program executes the CTRL/O RESET IOT (IOT #0);

4. ↑C is typed.

5. The program executes the RESET IOT (IOT #1).

6. The program executes a prompted SEEK, SEEKF, or ENTER command (Section 9.6.3). RESMON enables teleprinter output so the prompt message will be seen.

If ↑O is typed during a keyboard input command, it will be echoed but will not be passed to the buffer; keyboard input will continue to be echoed as usual.

CTRL/P - Echoes as ↑P and causes a jump to the restart address, if a non-zero restart address was specified via the RESTART IOT (IOT #2; see Section 9.5.2).

Lower Case ASCII - The ASCII codes 141-172 (lower case a-z) are converted to the codes 101-132 (upper case A-Z) on input and are echoed and stored in the buffer as such.

The echo may be suppressed by setting bit 7 of the buffer header Mode Byte.

If the buffer overflows, only the characters which fit into the buffer are echoed. Characters which are deleted by RUBOUT or CTRL/U do not read into the buffer even though they are echoed. If a carriage return causes an overflow, or if a carriage return is typed after an overflow has occurred, a carriage return/line feed will be echoed but only the line feed will enter the buffer.

In the following formatted ASCII examples, assume there is room for five characters in the buffer.) indicates typing a carriage return, ↓ represents typing a line feed, RUBOUT represents typing the RUBOUT key, and CTRL/U indicates that the CTRL/U combination has been typed.

User Typed:	Echoed on Console Terminal:	ASCII Code Entered into Buffer:
ABC ↵	ABC ↵↓	ABC ↵↓
ABCD ↵	ABCD ↵↓	ABCD↓
ABCDEF ↵	ABCD ↵↓	ABCD↓
ABCDEF RUB OUT ↵	ABCDD↵↓	ABC ↵↓
CTRL/U RUB OUT ↵	↑U ↵↓	↵↓
ABCDEF RUB RUB OUT OUT ↵	ABCDDC↵↓	AB ↵↓
ABCDEF RUB RUB RUB X OUT OUT OUT ↵	ABCDDCBX↵↓	AX↵↓

Console Terminal Printer

Characters are printed from the buffer as they appear except that nulls are ignored and tabs are output as spaces up to the next tab stop.

Line Printer

Characters are printed from the buffer as they appear except as follows:

Nulls	- Ignored.
Tab	- Output as spaces up to the next tab stop.
Carriage Return	- Ignored. It is assumed that a line feed or form feed follows. These characters cause the line printer "carriage" to advance.

All characters beyond the 132nd (or 80th if the optional line printer is used) are printed on the next line; RESMON issues a CR/LF and continues output.

Cassette Input

Nulls	- Ignored.
Rubout	- Ignored.
CTRL/Z	- Sets Done Bit and EOM Bit (bit 6) in Buffer Status byte.

Cassette Output

Characters are transferred from the buffer as they appear. When a formatted ASCII cassette file is closed, the Monitor writes a ↑Z into the output block and pads the unused portion of the block with nulls.

9.4.2 Unformatted ASCII

Unformatted ASCII READs and WRITEs transfer the number of 7-bit characters specified by the header Byte Count. (See Section 9.3.2 for information on unformatted transfers using cassette.)

Device-dependent functions include only the keyboard. Characters are read and echoed except as follows:

Tab	- Echoes as spaces up to the next tab stop.
CTRL/P	- Same as formatted ASCII.
CTRL/C	- Same as formatted ASCII.
CTRL/O	- Same as formatted ASCII.
Lower Case ASCII	- Same as formatted ASCII.

9.4.3 Formatted Binary

Formatted binary is used to transfer checksummed binary data (8-bit characters) in blocks. A formatted binary block appears as follows:

Byte (octal)	Meaning
001	- Start of block (output automatically by RESMON).
000	- Always null (output automatically by RESMON).
XXX XXX	- Block Byte Count (low-order followed by high-order). Count includes data and preceding four bytes (output automatically by RESMON).

```
DDD
DDD
 .
 .
 .          - Data bytes (from user's buffer).
 .
 .
DDD
DDD

CCC         - Checksum. Negation of the sum of all
              preceding bytes in the block (output
              automatically by RESMON).
```

RESMON creates the block during output from the buffer and buffer header. The Byte Count word in the buffer header specifies the number of data bytes which are to be output. Note that the number of bytes output is four larger than the header Byte Count. As the block is output, RESMON calculates the checksum which is output following the last data byte.

On formatted binary READs, RESMON ignores null characters until the first non-null character is read. If this character is a 001, a formatted binary block is assumed to follow and is read from the device under control of the Byte Count value. If the first non-null character is not 001, the READ is immediately terminated and error code 4 (see Table 9-2) is set in the Status Byte. As the block is read a checksum is calculated and compared to the checksum following the block. If the checksum is incorrect, error code 2 is set in the Status Byte of the buffer header. If the binary block is too large (i.e., [Byte Count-4] larger than the buffer size specified in the header), the last byte of the buffer is overlaid until the last data byte has been read; error code 3 is set in the Status Byte.

Device dependent functions do not apply to formatted binary READs and WRITEs. Eight-bit data characters are transferred to and from the device and buffer exactly as they appear.

9.4.4 Unformatted Binary

This mode transfers 8-bit characters with no formatting or character conversions of any kind. For both input and output, the buffer header Byte Count determines the number of characters transferred. (See Section 9.3.2 for information on unformatted transfers using cassette.)

Device dependent functions do not apply.

9.5 NON-DATA TRANSFER COMMANDS

The following commands are needed for initialization before any I/O transfers can take place.

9.5.1 RESET

The RESET command must be the first RESMON command issued by a user program and takes the form:

```
IOT
.BYTE 1,0
.WORD 0
```

It initializes many of RESMON's internal flags, resets all devices to their state at power-up (a hardware RESET instruction is issued) enables keyboard interrupts, clears the ↑O flag, and clears the ↑P RESTART address (set by the RESTART IOT). This IOT is normally issued only at the start of a user's program. It takes a significant amount of time to complete since RESMON goes into a timing loop and then issues a hardware RESET instruction. If this were not done, the last characters printed on the console terminal could be garbled.

9.5.2 RESTART

The RESTART command designates an address at which to restart a program. The format of the command is:

```
IOT
.BYTE 2,0
.WORD (address to restart)
```

After this command has been issued, typing CTRL/P on the keyboard will transfer program control to the restart address. If the restart address is designated as 0, the CTRL/P restart capability is disabled.

The RESTART command cancels keyboard interrupts. It is the program's responsibility to clean up any I/O in progress and to ensure that the Stack Pointer is reset.

It is a good programming practice for the code at the restart address to check if any cassette output files were open when ↑P was typed and to close them before actually restarting normal program execution. It is also advisable to issue a RESET IOT after a ↑P restart and before any RESMON data transfer commands are issued.

9.5.3 CNTRLO

The CNTRLO command resets the RESMON ↑O flag, thus enabling future console terminal output. The format is:

```
IOT
.BYTE 0,0
.WORD 0
```

The ↑O flag (which suppresses console terminal output) is set by the user typing ↑O on the keyboard. The flag is cleared (thus enabling teleprinter output to continue) when one of the following occurs:

1. ↑O is typed again.

2. The program running in memory requests keyboard input.

3. ↑C is typed.

4. The program running issues CNTRLO IOT.

5. The program running issues RESET IOT.

6. The program running issues a prompted SEEK, SEEKF, or ENTER IOT (see Section 9.6.3, User Prompting).

9.6 CASSETTE FILE I/O COMMANDS

The following RESMON commands are used for setting up I/O transfers.

9.6.1 SEEK

The SEEK command is for cassette only and is used to open a cassette file for input. SEEK sets up information which RESMON uses in subsequent READ's from the specified unit. The format of the SEEK command is:

```
IOT
.BYTE 10,(device #--device 0 or 1 only)
.WORD (pointer to list of arguments for SEEK)
```

The list of arguments for the SEEK command appears as:

```
.BYTE Status/Error, (Mode)

.WORD (address of 128 byte buffer for use when reading
       cassette blocks if formatted mode is specified;
       otherwise 0)

.WORD (address of a second 128 byte buffer if double
       buffered input is desired; otherwise 0)

.WORD (address of 32 byte buffer for storage of
       file headers while SEEKing)

.WORD (address of filename to SEEK)

.WORD (address to return to if error detected)
```

The 32 byte buffer for file headers is the area into which RESMON will read file headers as it is looking for the specified file. This buffer is a scratch area and will generally be the same for every SEEK command the user has in his program. The address of the filename to SEEK is a pointer to an area containing the filename and extension properly padded to nine bytes (if necessary), which is to be looked for on the specified cassette unit. For formatted I/O, the address of the 128 byte buffer tells RESMON where to read cassette data blocks once the specified file has been found. RESMON reads blocks into this

buffer from cassette and then takes data from this buffer and moves it to the user's line buffer to fulfill a READ IOT. If the user specifies a second 128 byte buffer, RESMON will use it to implement a double buffered input scheme for subsequent READ's on that device. For unformatted I/O, the buffer specifications are ignored.

RESMON sets the Status/Error Byte in the list of SEEK arguments to reflect errors, as follows:

Bit set	Error
7	Error detected
6	File not found
5	Hard error
4	Conflict (e.g. output file was open)
3	No I/O buffer specified for formatted I/O

On detection of an error, RESMON sets bit 7 and one other bit in the error byte and transfers control to the error address specified in the list of arguments.

If no error was detected, SEEK returns with the header of the desired file in the user-specified scratch area and with the cassette positioned to READ the first data block of the file. No data blocks are read as a result of a SEEK. The SEEK command always rewinds the specified cassette before doing a SEEK (sequential search).

NOTE

If the first byte of an extension specified by the user in a SEEK or SEEKF command is 000, RESMON will not attempt to match the extension, but rather will look for the first file which has the same filename.

If the first byte of a filename specified by the user in a SEEK or SEEKF command is 000, RESMON will not compare filenames at all, but rather will position the cassette so as to read the first file encountered. For SEEK, this is always the first file on the cassette, since the tape is always rewound first. For SEEKF, this is the first file encountered spacing forward from the current position.

9.6.2 SEEKF

The SEEKF (SEEK Forward) command is identical in format and operation to the SEEK command, except SEEKF does not perform a rewind before searching for the specified file. The format of the command is:

```
IOT
.BYTE 11, (device #)
.WORD (pointer to list of arguments for SEEKF)
```

The list of arguments is the same as for SEEK and can be found in Section 9.6.1.

9.6.3 ENTER

The ENTER command is for cassette only, and is used to create a new file on cassette (at the logical end of cassette). The format of the command is:

```
IOT
.BYTE 7, (device #)
.WORD (pointer to list of ENTER arguments)
```

The list of arguments for the ENTER command is similar to the list of arguments for the SEEK commands:

```
.BYTE Status/Error, (Mode)

.WORD (address of 128 byte buffer for use in writing
       cassette blocks in formatted mode; otherwise 0)

.WORD (address of second 128 byte buffer if double
       buffered output is desired; otherwise 0)

.WORD (address of 32 byte buffer for storage of
       file headers)

.WORD (address of filename to be ENTERed)

.WORD (address to return to if error detected)

.WORD (address of overflow subroutine to be called if a
       formatted file hits end-of-tape before it is
       closed; otherwise 0)
```

The ENTER command rewinds the specified cassette unit and does a SEEK for the filename supplied by the user. (ENTER assumes that the filename address supplied by the user is the beginning of a 32 byte header to be written out as the header block of the file being ENTERed. See Appendix F for a complete description of the cassette file header.) If the file is found, it is deleted by overwriting the existing header with an "*EMPTY" header. The ENTER command then moves down to the logical end of cassette and replaces the end of cassette marker with the header specified by the user. The cassette is left positioned to write the first data block of the new file. Before writing the new header, RESMON performs several operations: the sequence and continuation bytes of the user-specified header are set to zero; the length is set to 128 bytes per data record; if the first two bytes of the date are zero (or spaces--ASCII 240), RESMON will supply the current date (if the user specified a date with the Monitor DATE command).

For formatted I/O, the 128 byte buffer in the list of ENTER arguments is an intermediate buffer which RESMON uses in writing data blocks of the ENTERed file. The user normally issues a WRITE IOT specifying a line buffer; RESMON takes data from the line buffer and moves it to the user-specified 128 byte buffer; when this 128 byte buffer is full, it is written out to cassette. If the user supplies the address of a second 128 byte buffer, RESMON will double buffer cassette output for this file. Buffer information from the ENTER is stored by RESMON for reference during I/O to the specified unit. For unformatted I/O the buffer specifications are ignored.

The Mode Byte in the list of arguments is similar to the Mode Byte in the SEEK command--it indicates how the user intends to write the file being ENTERed and is stored by RESMON for reference during I/O. It can have only the values listed in Section 9.3.1 under "Mode Byte". The Status/Error Byte in the list of ENTER arguments is set by RESMON to reflect errors detected during the ENTER function; following is a list of the error bits:

Bit Set	Error
7	Error detected
6	Full tape (clear leader found)
5	Hard error
4	Conflict (output file was open)
3	No I/O buffer specified for formatted I/O

If an error is detected, RESMON sets bit 7 and one other bit and transfers control to the error address specified in the list of arguments.

The last item in the list of ENTER arguments is an overflow subroutine to be called in case the user ENTERs a formatted file and the WRITE processor encounters the end of cassette before the file is CLOSEd. If an overflow subroutine was specified when the file was ENTERed, the WRITE processor will call it via a JSR PC,SUBR. The user's subroutine should tell the user to mount a new cassette on the same drive that the file which overflowed was mounted on. It should then ENTER a file on that new cassette (using the same internal buffers as the original ENTER command) and then return to RESMON's WRITE processor via an RTS PC. The WRITE processor will continue writing onto the new file; the two files should then be combined with PIP before being used further. RESMON saves registers 0-5 before calling the user subroutine, so the user need not worry about destroying the contents of these registers. However, the user should be careful not to destroy the stack pointer (Register 6).

User Prompting

The commands SEEK, SEEKF, and ENTER have an additional feature which can aid the user who has his files on many different cassettes. If, on entry to these commands, the Status/Error byte in the list of arguments is equal to 377(octal) RESMON will prompt the user to mount a new cassette on the unit specified for the command. RESMON will type:

```
#?
```

where "#" is the unit number on which RESMON expects a new cassette to be mounted. RESMON then waits for the user to type any character on the keyboard. When the user has done this, RESMON assumes that the proper cassette has been mounted and initiates the command.

Chapter 3 provides more details concerning user prompting.

Non-Fatal Off-Line and Write-Lock Errors

SEEK, SEEKF, and ENTER have the ability to detect write-lock and off-line (no cassette mounted) errors and allow the user to correct them without aborting the command in progress. When one of these commands is initiated, if there is no cassette mounted on the specified unit, the message:

```
?OFFLINE n
```

will be generated. The user should mount on unit n the cassette containing the file he wishes to SEEK or the cassette on which he wishes a new file ENTERed. RESMON will automatically proceed with the specified command. No action other than mounting the cassette is necessary.

Likewise, when an ENTER command is initiated, if the cassette mounted on the specified unit is write-locked, RESMON will generate the message:

```
?WRT LOCK n
```

The user should dismount the cassette, write-enable it, and remount it. RESMON will continue with the specified ENTER command automatically.

NOTE

When the user dismounts the cassette, he will also see the "?OFFLINE n" message described above.

9.6.4 CLOSE

The CLOSE command is for cassette only and specifies that a certain file presently open for output is to be closed and not referenced further.

NOTE

CLOSE may be issued for any device, but it is ignored for the console terminal keyboard and printer, and line printer. It is also ignored if no output file is open on the specified unit.

The format of the command is:

```
IOT
.BYTE 6, (device #)
.WORD (address for transfer if error detected)
```

CLOSE frees the unit so that it may be opened again via a SEEK, SEEKF, or ENTER. If CLOSE is issued for a unit which is open for input, no error will occur but control will return immediately to the user.

In the case of unformatted ASCII and binary files, CLOSE waits until the last WRITE initiated is completed, then writes an end-of-tape marker and rewinds the cassette. If the user initiates an unformatted WRITE and then immediately does a CLOSE, the CLOSE processor has to wait until the WRITE is completed before it can start to write an end-of-tape marker. The error return is never taken for unformatted CLOSEs.

If a formatted output file is open, CLOSE must write out the last portion of RESMON's internal buffer (if there is any data in it), write an end-of-file on the cassette, and rewind the cassette. Control is returned to the user once the rewind has been initiated.

In the case of a formatted ASCII file open for output, CLOSE will supply a ↑Z (ASCII 32) as logical end-of-file, pad the rest of the last data block with nulls, write out the last data block, write the end-of-tape marker, and rewind the cassette.

In the case of formatted binary files, CLOSE writes out the last data block with any unused portion of it padded with nulls, writes the end-of-tape marker, and rewinds the cassette.

The only possible error which may occur during a formatted CLOSE is clear leader or full tape; this error is detected when RESMON writes out the last portion of the internal buffer. If this WRITE is not successful, the error return is taken. If clear leader is detected when writing the end-of-tape marker, it is ignored.

9.7 DATA TRANSFER COMMANDS

The following IOT's are used to transfer data between devices.

9.7.1 READ

The READ command causes RESMON to read from the device associated with the specified device number according to the information found in the buffer header. The format of the command is:

```
IOT
.BYTE 5, (device #)
.WORD (address of first word of the buffer header)
      or
.WORD (address of parameter block)--for unformatted
       cassette READs
```

For unformatted cassette READs, the parameter block has the following form:

```
.BYTE Status/Error,0
.WORD (address of 128 byte buffer for READs)
```

RESMON initiates the transfer of data, clears the Status Byte, and returns control to the calling program. If the device on the selected channel is busy, or if a conflicting device (see Section 9.7.3) is busy, RESMON retains control until the data transfer can be initiated.

Upon completion of the READ, the appropriate bits in the Status Byte are set by RESMON and the Byte Count word indicates the number of bytes in the data buffer.

For formatted cassette READs the flow of execution is as follows:

RESMON reads a data block into an intermediate cassette buffer (specified by the user at SEEK time). From that buffer, RESMON pulls characters one at a time and uses them to fill the buffer specified by the user in the READ command. The user buffer is filled exactly as if the characters were coming directly from the cassette and the process is governed as described in Section 9.3.1. If, at SEEK time, the user specified a second intermediate buffer, the cassette I/O is double buffered, thus minimizing the amount of time the user program must wait for physical I/O transfers. Note that the user can implement his own double buffering scheme by using unformatted cassette I/O, since in that case the location of cassette buffers is not fixed at SEEK or ENTER time, but can be varied with every READ (or WRITE) command simply by changing the buffer pointer in the command (see Section 9.3.2).

For formatted cassette READs, RESMON will set the Status Byte in the buffer header to reflect the status of the data transfer as described in section 9.3.1, Status Byte. For unformatted cassette READs the Status Byte in the parameter block is set to reflect the status of the operation as described in the section on Cassette I/O Primitives (Section 9.8). For formatted cassette READs, the Done Bit will be set when the user's buffer has been filled, even though there may be some physical I/O still in progress. With regard to formatted cassette I/O, as a result of the intermediate buffering scheme the user's buffer will always be full when he regains control following a READ command since the data is coming from a memory buffer. If the user tries to do a READ from cassette before doing a SEEK or SEEKF, the Monitor will give a "NO FILE OPEN n" message, where n is either 0 or 1 (for drive 0 or 1).

9.7.2 WRITE

RESMON writes on the device associated with the specified device number according to the information found in the buffer header. The format of the command is:

```
IOT
.BYTE 4, (device #)
.WORD (address of first word of the buffer header)
     or
.WORD (address of parameter block)--for unformatted
       cassette WRITEs
```

For unformatted cassette WRITEs, the parameter block has the form:

```
.BYTE Status/Error,0
.WORD (address of 128-byte buffer to WRITE)
```

Transfer of data occurs in the amount specified by the Byte Count (Buffer+4). RESMON returns control to the calling program as soon as the transfer has been initiated. If the selected device is busy or a

conflicting device is busy, RESMON retains control until the transfer can be initiated. Upon completion of the WRITE, RESMON will set the Status Byte to the latest conditions. If a WRITE causes an EOM condition, the user has no way of determining how much of his buffer has been written (the Byte Count remains the same).

The WRITE command behaves the same way as the READ command with regard to formatted and unformatted cassette I/O. When control is returned to the user after a formatted cassette WRITE, his line buffer is available. The status bytes for formatted and unformatted cassette WRITEs are interpreted like those for cassette READs.

If a WRITE is issued without first doing an ENTER (see Section 9.6.3) the Monitor will respond by typing a "NO FILE OPEN n" message, where n is the drive number.

9.7.3 Device Conflicts in Data Transfer Commands

Because there is a physical association between the printer and keyboard on the console terminal, certain devices cannot be in use at the same time. When a data transfer command is given, RESMON simultaneously checks for two conditions before executing the command:

1. Is the device requested already in use?
2. Is there some other device in use that would result in an operational conflict?

RESMON resolves both conflict situations by waiting until the first device is no longer busy before allowing the requested device to start functioning. (This is an automatic WAITR command; see the next section.) For example, if the console terminal is in use, and either a KBD request or a second request for the terminal itself is made, RESMON will wait until the current output operation has been completed before returning control to the calling program.

Table 9-3 lists the devices; corresponding to each device on the left is a list of devices (or the echo operation) which would conflict with it in operation.

Table 9-3
Device Conflicts

Device	All Possible Conflicting Devices or Operations
Terminal Keyboard (KBD) Printer (TTY)	Echo, KBD, TTY
Cassette (CT0 or CT1)	CT0, CT1
Line Printer (LP)	LP

9.7.4 WAITR (Wait, Return)

The WAITR command is used to test the status of the specified device. The format of the command is:

```
IOT
.BYTE 3, (device #)
.WORD (busy return address)
```

If the device (or any possible conflicting device) is not transferring data, control is returned to the instruction following the WAITR command. Otherwise, control is transferred to the busy return address.

Note that a not busy return from WAITR normally means the device is available. However, in the case of a WRITE to the console terminal or line printer, this means only that the last character has been output to the device. The device is still in the process of printing the character. Thus, care must be exercised when performing a hardware RESET or HALT after a WRITE-WAITR sequence, since these may prevent the last character from being physically output.

WAITR vs. Testing the Buffer Done Bit

WAITR tests the status not only of the device it specifies, but also of all possible conflicting devices. This means that when WAITR indicates that the device is not busy, the data transfer on the device of interest may have been completed for some time. Depending on the program and what devices are being used for a given run, the WAITR could have been waiting an additional amount of time for a conflicting device to become free (i.e., waiting for the KBD when the TTY is to be used, or waiting for CT0 when CT1 is to be used). Where this possibility exists and buffer availability is the main concern, testing the Done Bit of the Status Byte (set when the buffer transfer is complete) would be preferable to WAITR; alternately, WAITR would be preferable if device availability is the main interest.

In unformatted transfers to and from cassette, WAITR is equivalent to checking the Done Bit for the last READ or WRITE command.

In formatted transfers to and from cassette, a WAITR is equivalent to checking whether there is any physical I/O occurring on the specified unit. The user is not generally concerned with this--normally he only wants to know when his line buffer is free if he is doing formatted I/O. Note that in this case even though no physical I/O is going on when the not busy return is taken, there may still be data in the user's intermediate cassette buffer (as specified in the SEEK or ENTER command). WAITR would generally not be used when the programmer is writing/reading a cassette file in formatted mode.

9.7.5 Single Buffer Transfer on One Device

The program segment below includes a WAITR which goes to a busy return address that is its own IOT, continuously testing device 3 for availability; in this case, only a single device and a single buffer

are involved. A done condition in the buffer 1 Status Byte can be inferred from the availabilty of device 3. This knowledge ensures that all data requested for Buffer 1 is available for processing.

```
A:      IOT             ;TRAP TO RESMON
        .BYTE READ,3    ;SPECIFY BUFFER AND
        .WORD BUF1      ;READ FROM DEVICE 3
                        ;INTO BUFFER

BUSY:   IOT             ;TRAP TO RESMON
        .BYTE WAITR,3   ;WAIT FOR DEVICE 3
        .WORD BUSY      ;SPECIFY BUSY RETURN
        .               ;ADDRESS TO FINISH
        .               ;READING
        (Process Buffer 1)
        .
        .
        JMP A
```

Testing the Done Bit of Buffer 1 might have been used instead, but was not necessary with only one device operating.

9.7.6 Double Buffering

The example below illustrates a time-saving double buffer scheme whereby data is processed in Buffer 1 at the same time that new data is being read into Buffer 2; sequentially, data is processed in Buffer 2 at the same time that new data is being read into Buffer 1.

```
        IOT             ;TRAP TO RESMON
        .BYTE READ,3    ;SPECIFY BUFFER 1
        .WORD BUF1      ;READ FROM DEVICE
                        ;3 INTO BUFFER 1

A:      IOT             ;TRAP TO RESMON
        .BYTE READ,3    ;SPECIFY BUFFER 2
        .WDRD BUF2      ;READ FROM DEVICE
        .               ;3 INTO BUFFER 2
        (process BUF1 concurrently with READ into BUF2)
        .
        .
B:      IOT             ;TRAP TO RESMON
        .BYTE READ,3    ;SPECIFY BUFFER 1
        .WORD BUF1      ;READ FROM DEVICE
        .               ;3 INTO BUFFER 1
        .
        (process BUF2 concurrently with READ into BUF1)
        .
        .
        JMP A
```

Because RESMON ensures that the requested device is free before initiating the command, the subsequent return of control from the IOT at A implies that the READ prior to A is complete; that is, that Buffer 1 is available for processing. Similarly, the return of

control from the IOT at B implies that Buffer 2 is available. WAITR's are not required because RESMON has automatically ensured the device's availability before initiating each READ.

9.8 CASSETTE I/O PRIMITIVES

RESMON also allows the sophisticated user to access the basic routines necessary for doing cassette I/O. This is done by means of IOT's with the following format:

```
IOT
.BYTE function, (device #)
.WORD (pointer to argument list)
```

These IOT's can access only cassette, i.e., device numbers 0 and 1. The functions listed in Table 9-4 are valid.

Table 9-4
Cassette I/O Functions

Function #	Meaning
12	WRITE file gap
13	WRITE (see below)
14	READ (see below)
15	Space reverse file
16	Space reverse block
17	Space forward file
20	Space forward block
21	Rewind

For READ and WRITE, the list of arguments is as follows:

```
.BYTE Status/Error,0
.WORD (buffer address)
.WORD (byte count)
```

For functions other than READ and WRITE the list of arguments is only:

```
.BYTE Status/Error,0
```

Errors are reported in the Status/Error byte as follows:

Bit set	Meaning
7	Error Detected
6	Block Checksum (on READ)
5	Clear Leader
4	(not used)
3	File Gap Detected
2	(not used)
1	(not used)
0	Done

RESMON sets bit 7 and one (or more) other bits if an error is detected. Write-lock, off-line, and timing errors cause a fatal error message and return to RESMON.

RESMON stores the high order byte of the cassette status and command register in the user's status error byte when an error is detected. The user should check error bits in the following order:

1. Clear Leader
2. File Gap
3. Block Checksum

Because of the nature of the cassette hardware, more than one of these bits may be set. The above order should be used when checking the bits; only the first bit detected is significant.

Bit 0 of the Status/Error byte is set to 1 when the function is complete; control is returned to the user as soon as the function is initiated. If physical cassette I/O is in progress when one of these functions is called, RESMON will wait until the I/O is complete, initiate the desired function, and then return to the user.

9.9 ERROR MESSAGES

The following error messages are detected in RESMON (refer to Section 3.7 of Chapter 3):

Table 9-5
RESMON Error Messages

Message	Arg	Meaning
IOT	PC	An IOT was issued at the indicated location which referenced either an illegal RESMON command, illegal device, or illegal data mode.
NO FILE OPEN	drive #	User issued a cassette READ or WRITE without doing a SEEK or ENTER.
OFFLINE	drive #	User attempted to access a cassette which was not mounted; execution is automatically resumed when the cassette is mounted.
TIMING	drive #	A timing error occurred on the drive indicated (RESMON tries the operation 3 times.)

(Continued on next page)

Table 9-5 (Cont.)
RESMON Error Messages

Message	Arg	Meaning
TRAP	PC	A stack overflow, attempt to reference a word on a byte boundary, or illegal memory reference trap occurred at the location indicated. The stack pointer (R6) at time of error is saved in location 44.
WRT LOCK	drive #	User attempted to WRITE on a write-locked cassette; execution is automatically resumed when the cassette is write-enabled.

9.10 EXAMPLE OF PROGRAM USING RESMON

An example of the use of RESMON by both the CAPS-11 System and from within a user program is presented in Appendix D.

APPENDIX A

ASCII CHARACTER CODES

A.1 KEYBOARD DIFFERENCES

Certain console terminals vary concerning labeling of keyboard keys and characters output upon receipt of particular ASCII character codes. The following list should be referenced to determine possible differences:

Keys Which Perform the Same Function		Represent the ASCII Code
↑		136
←		137
RUBOUT	DELETE	177
ESCAPE	ALTMODE	176 175
SHIFT L	\	134
CTRL I	TAB	211
SHIFT K	[	133
SHIFT M	]	135

A.2 CHARACTER CODES

The following is a list of the 7-bit octal ASCII character codes. (ASCII is an abbreviation for American Standard Code for Information Interchange.)

7-Bit Octal	Character	7-Bit Octal	Character	7-Bit Octal	Character	7-Bit Octal	Character
000	NUL	040	SP	100	@	140	space
001	SOH	041	!	101	A	141	a
002	STX	042	"	102	B	142	b
003	ETX(↑C)	043	#	103	C	143	c
004	EOT	044	$	104	D	144	d
005	ENQ	045	%	105	E	145	e
006	ACK	046	&	106	F	146	f
007	BEL	047	'	107	G	147	g
010	BS	050	(	110	H	150	h
011	HT	051	)	111	I	151	i
012	LF	052	*	112	J	152	j
013	VT	053	+	113	K	153	k
014	FF	054	,	114	L	154	l
015	CR	055	-	115	M	155	m
016	SO	056	.	116	N	156	n
017	SI(↑O)	057	/	117	O	157	o
020	DLE(↑P)	060	0	120	P	160	p
021	DC1	061	1	121	Q	161	q
022	DC2	062	2	122	R	162	r
023	DC3	063	3	123	S	163	s
024	DC4	064	4	124	T	164	t
025	NAK	065	5	125	U	165	u
026	SYN	066	6	126	V	166	v
027	ETB	067	7	127	W	167	w
030	CAN	070	8	130	X	170	x
031	EM	071	9	131	Y	171	y
032	SUB(↑Z)	072	:	132	Z	172	z
033	ESC	073	;	133	[	173	{
034	FS	074	<	134	\	174	(|)
035	GS	075	=	135	]	175	(})
036	RS	076	>	136	↑(^)	176	(~)
037	US	077	?	137	←(_)	177	DEL

APPENDIX B

ASSEMBLY LANGUAGE SUMMARY

B.1 TERMINATORS

Character	Function
CTRL/FORM	Source line terminator
LINE FEED	Source line terminator
RETURN	Source line terminator
:	Label terminator
=	Direct assignment delineator
%	Register term delineator
TAB	Item terminator Field terminator
BLANK or SPACE	Item terminator Field terminator
#	Immediate expression field indicator
@	Deferred addressing indicator
(	Initial register field indicator
)	Terminal register field indicator
'	Operand field separator
;	Comments field delimiter
+	Arithmetic addition operator
-	Arithmetic subtraction operator

Character	Function
&	Logical AND operator
!	Logical OR operator
"	Double ASCII text indicator
'	Single ASCII text indicator

B.2 ADDRESS MODE SYNTAX

In the following syntax table, n represents an integer between 0 and 7; R is a register expression; E represents any expression; ER represents either a register expression or an absolute expression in the range of 0 to 7.

Address Mode Number	Address Mode Name	Symbol in Operand Field	Meaning
0n	Register	R	Register R contains the operand. R is a register expression.
1n	Deferred Register	@R or (R)	Register R contains the operand address.
2n	Autoincrement	(ER)+	The contents of the register specified by ER are incremented after being used as the address of the operand.
3n	Deferred Autoincrement	@(ER)+	ER contains a pointer to the address of the operand. ER is incremented after use.
4n	Autodecrement	-(ER)	The contents of register ER are decremented before being used as the address of the operand.
5n	Deferred Autodecrement	@-(ER)	The contents of register ER are decremented before being used as a pointer to the address of the operand.
6n	Index by the Register Specified	E(ER)	The value obtained when E is combined with the contents of the register specified (ER) is the address of the operand.

Address Mode Number	Address Mode Name	Symbol in Operand Field	Meaning
7n	Deferred index by the Register Specified	@E(ER)	E added to ER produces a pointer to the address of the operand.
27	Immediate Operand	#E	E is the operand.
37	Absolute address	@#E	E is the operand address.
67	Relative address	E	E is the address of the operand.
77	Deferred relative address	@E	E is a pointer to the address of the operand.

B.3 INSTRUCTIONS

The tables of instructions which follow are grouped according to the operands they take and according to the bit patterns of their op-codes.

In the representation of op-codes, the following symbols are used:

SS	Source operand	Specified by a 6-bit address mode
DD	Destination operand	Specified by a 6-bit address mode
XX	8-bit offset to a location	Branch instructions
R	Integer between 0 and 7	Represents a general register

Symbols used in the description of instruction operations are:

SE	Source effective address
DE	Destination effective address
( )	Contents of
→	Becomes

The condition codes in the processor status word (PS) are affected by the instructions; these condition codes are represented as follows:

N	Negative bit:	Set if the result is negative
Z	Zero bit:	Set if the result is zero
V	Overflow bit:	Set if the result had an overflow
C	Carry bit:	Set if the result had a carry

In the representation of the instruction's effect on the condition codes, the following symbols are used:

*	Conditionally set
-	Not affected
0	Cleared
1	Set

To set conditionally means to use the instruction's result to determine the state of the code.

Logical operators are represented by the following symbols:

!	Inclusive OR
ⓘ	Exclusive OR
&	AND
‾	Used over a symbol to represent the 1's complement of the symbol

B.3.1 Double Operand Instructions (OP A,A)

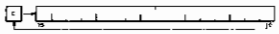
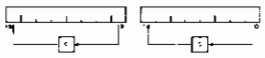
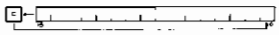
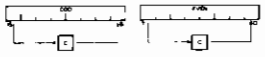
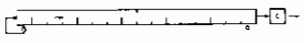
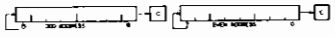
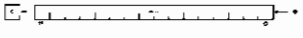
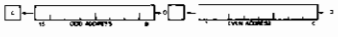
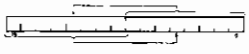
Op-code	Mnemonic	Stands for	Operation	Condition Codes N	Z	V	C
01SSDD	MOV	MOVe	(SE)→(DE)	*	*	0	-
11SSDD	MOVB	MOVe Byte					
02SSDD	CMP	CoMPare	(SE)-(DE)	*	*	*	*
12SSDD	CMPB	CoMPare Byte					
03SSDD	BIT	BIt Test	(SE) & (DE)	*	*	0	-
13SSDD	BITB	BIt Test Byte					
04SSDD	BIC	BIt Clear	$(\overline{SE})$ & (DE)→(DE)	*	*	0	-
14SSDD	BICB	BIt Clear Byte					
05SSDD	BIS	BIt Set	(SE) ! (DE)→(DE)	*	*	0	-
I5SSDD	BISB	BIt Set Byte					
06SSDD	ADD	ADD	(SE) + (DE)→(DE)	*	*	*	*
16SSDD	SUB	SUBtract	(DE) - (SE)→(DE)	*	*	*	*

B.3.2 Single Operand Instructions (OP A)

Op-code	Mnemonic	Stands for	Operation	Condition Codes N	Z	V	C
0050DD	CLR	CLeaR	0→(DE)	0	1	0	0
1050DD	CLRB	CLeaR Byte					
0051DD	COM	COMplement	$(\overline{DE})$→(DE)	*	*	0	1
1051DD	COMB	COMplement Byte					

Op-code	Mnemonic	Stands for	Operation	Condition Codes N	Z	V	C
0052DD 1052DD	INC INCB	INCrement INCrement Byte	(DE) + 1→(DE)	*	*	*	1
0053DD 1063DD	DEC DECB	DECrement DECrement Byte	(DE) - 1→(DE)	*	*	*	-
0054DD 1054DD	NEG NEGB	NEGate NEGate Byte	$(\overline{DE})$ + 1→(DE)	*	*	*	*
0055DD 1055DD	ADC ADCB	ADd Carry ADd Carry Byte	(DE) + (C)→(DE)	*	*	*	*
0056DD 1056DD	SBC SBCB	SuBtract Carry SuBtract Carry Byte	(DE) - (C)→(DE)	*	*	*	*
0057DD 1057DD	TST TSTB	TeST TeST Byte	(DE) - 0→(DE)	*	*	0	0

B.3.3 Rotate/Shift

0060DD	ROR	ROtate Right		*	*	*	*
1061DD	RORB	ROtate Right Byte		*	*	*	*
0061DD	ROL	ROtate Left		*	*	*	*
1061DD	ROLB	ROtate Left Byte		*	*	*	*
0062DD	ASR	Arithmetic Shift Right		*	*	*	*
1062DD	ASRB	Arithmetic Shift Right Byte		*	*	*	*
0063DD	ASL	Arithmetic Shift Left		*	*	*	*
1063DD	ASLB	Arithmetic Shift Left Byte		*	*	*	*
0001DD	JMP	JuMP	DE →(PC)	-	-	-	-
0003DD	SWAB	SWap Bytes		*	*	0	0

The following 3 instructions are available on the PDP-11/40, 45 only:

0065SS	MFPI	Move From Previous Instruction space	(SE)→(TEMP) (SP)-2→(SP) (TEMP)→((SP))	*	*	0	-
0066DD	MTPI	Move To Previous Instruction space	((SP))→(TEMP) (SP+2→(SP) (TEMP)→(DE)	*	*	0	-
0067DD	SXT	Sign eXTend	0 DE if N bit is clear -1 DE if N bit is set	-	*	-	-

B.3.4 Operation Instructions (OP)

Op-Code	Mnemonic	Stands for	Operation	Condition Codes N	Z	V	C
000000	HALT	HALT	The computer stops all functions.	-	-	-	-
000001	WAIT	WAIT	The computer stops and waits for an interrupt.	-	-	-	-
000002	RTI	ReTurn from Interrupt	The PC and ST are popped off the SP stack: ((SP))→(PC) (SP)+2→(SP) ((SP))→(ST) (SP)+2→(SP)	*	*	*	*
000003	000003	Breakpoint Trap	Trap to location 14. This is used to call ODT.	*	*	*	*
000004	IOT	Input/Output Trap	Trap to location 20. This is used to call RESMON.	*	*	*	*
000005	RESET	RESET	Returns all I/O device handlers to power-on state.	-	-	-	-

The following instruction is available on the PDP-11/40, 45 only:

000006	RTT	ReTurn from Interrupt	Same as RTI instruction but inhibits trace trap.	*	*	*	*

Trapping OP or OP E where 0<=E<=337(octal)

104000- 104377	EMT	EMulator Trap	Trap to location 30. This is used to call system programs.	*	*	*	*
104400- 104777	TRAP	TRAP	Trap to location 34. This is used to call any routine desired by the programmer.	*	*	*	*

CONDITION CODE OPERATES

Op-code	Mnemonic	Stands for
000241	CLC	CLear CArry Bit in PS.
000261	SEC	SEt Carry bit.
000252	CLV	CLear oVerflow bit.
000262	SEV	SEt oVerflow bit.
000244	CLZ	CLear Zero bit.
000264	SEZ	SEt Zero bit.
000250	CLN	CLear Negative bit.
000270	SEN	SEt Negative bit.
000254	CNZ	CLear Negative and Zero bits.
000257	CCC	CLear all Condition Codes.
000277	SCC	Set all Condition Codes.
000240	NOP	No-operation.

B.3.5 Branch Instructions OPR E
where -128(decimal)<(E-.-2)/2<127(decimal)

Op-Code	Mnemonic	Stands for	Condition to be met if branch is to occur
0004XX	BR	BRanch always	
0010XX	BNE	Branch if Not Equal (to zero)	Z=0
0014XX	BEQ	Branch if EQual (to zero)	Z=1

Op-Code	Mnemonic	Stands for	Condition to be met if branch is to occur
0020XX	BGE	Branch if Greater than or Equal (to zero)	N ⊕ V=0
0024XX	BLT	Branch if Less Than (zero)	N ⊕ V = 1
0030XX	BGT	Branch if Greater Than (zero)	Z ! (N ⊕ V)=0
0034XX	BLE	Branch if Less than or Equal (to zero)	Z ! (N ⊕ V)=1
1000XX	BPL	Branch if PLus	N=0
1004XX	BMI	Branch if MInus	N=1
1010XX	BHI	Branch if HIgher	C ⊕ Z=0
1014XX	BLOS	Branch if LOwer or Same	C ! Z=1
1020XX	BVC	Branch if oVerflow Clear	V=0
1024XX	BVS	Branch if oVerflow Set	V=1
1030XX	BCC (or BHIS)	Branch if Carry Clear (or Branch if HIgh or Same)	C=0
1034XX	BCS (or BLO)	Branch if Carry Set (or Branch if LOw)	C=1

B.3.6 Subroutine Call (JSR ER,A)

Op-code	Mnemonic	Stands for	Operation	
004RDD	JSR	Jump to Subroutine	Push register on the SP stack, put the PC in the register:	
			DE→(TEMP)	A temporary storage register internal to the processor
			(SP)-2→(SP)	
			(REG)→((SP))	
			(PC)+m→(REG)	M depends upon the address mode
			(TEMP)→(PC)	

B.3.7 Subroutine Return

Op-code	Mnemonic	Stands for	Operation
00020R	RTS	ReTurn from Subroutine	Put register contents in PC and pop old contents from SP stack into register.

B.4 ASSEMBLER DIRECTIVES

Mnemonic	Operand	Stands for	Operation
.EOT	none	End Of Tape	Indicates the physical end of the source input medium.
.EVEN	none	EVEN	Ensures that the assembly location counter is even by adding 1 if it is odd.
.END	E (E optional)	END	Indicates the physical and logical end of the program and optionally specifies the entry point (E).
.WORD	E, E,... E, E,...	WORD (the void operator)	Generates words of data.
.BYTE	E,E,...	BYTE	Generates bytes of data.
.ASCII	/xxx...x/	ASCII	Generates 7-bit ASCII characters for text enclosed by delimiters.
.TITLE	NAME	TITLE	Generates a name for the object module.
.ASECT	none	ASECT	Initiates the Absolute section.
.CSECT	none	CSECT	Initiates the Relocatable Control section.
.LIMIT	none	LIMIT	Generates two words containing the low and high limits of the relocatable section.
.GLOBL	NAME,NAME,...	GLOBaL	Specifies each name to be a global symbol.
.RAD50	/XXX/	RADix 50	Generates the RADIX 50 representation of the ASCII character in delimiters.
.LIST	none	LIST	Enables assembly listing (if it was off).
.NLIST	none	No LIST	Disables assembly listing (if it was on).

B.4.1 Conditional Directives

Mnemonic	Operand	Stands for	Operation
.IFZ	E	IF E=0	Assemble what follows up to the terminating .ENDC if the expression E is 0.
.IFNZ	E	IF E≠0	Assemble what follows up to the terminating .ENDC, if the expression E is not 0.
.IFL	E	IF E<0	Assemble what follows up to the terminating .ENDC, if the expression E is less than 0.
.IFLE	E	IF E<=0	Assemble what follows up to the terminating .ENDC, if the expression E is less than or equal to 0.
.IFG	E	IF E>0	Assemble what follows up to the terminating .ENDC, if the expression E is greater than 0.
.IFGE	E	IF E=>0	Assemble what follows up to the terminating .ENDC, if the expression E is greater than or equal to 0.
.IFDF	NAME	IF NAME defined	Assemble what follows up to the terminating .ENDC if the symbol NAME is defined.
.IFNDF	NAME	IF NAME undefined	Assemble what follows up to the terminating .ENDC if the symbol NAME is undefined.
.ENDC	none	End of Conditional	Terminates the range of a conditional directive.

APPENDIX C

COMMAND AND ERROR MESSAGE SUMMARIES

The following summaries are provided for the user's convenience and are grouped in chronilogical order according to the system program to which they refer. As these are only summaries, the user is referred to the appropriate chapter for details.

C.1 KEYBOARD MONITOR (Chapter 3)

Command Summary *

Command	Explanation
DATE	Allows the user to enter the day, month, and year. This date is then represented in directory listings.
DIRECTORY	Causes a directory listing of the cassette specified in the command line.
DIRECTORY/F	Causes a "fast" directory listing by omitting current and creation dates and listing only filenames and extensions.
LOAD	Instructs the Monitor to load the file specified in the command line.
LOAD/G	Instructs the Monitor to load and start the file specified in the command line.
LOAD/O	Instructs the Monitor to load the file specified in the command line, overlaying the Monitor as necessary.
RUN	Instructs the Monitor to load and execute the file specified in the command line.

*Only those characters underlined need be entered.

Command	Explanation
SENTINEL	Causes a sentinel file to be written immediately following the file specified in the command line.
START nnnnn	Instructs the Monitor to begin execution of a loaded file at the specified (nnnnn) address, or at the transfer address if nnnnn is not indicated.
VERSION	Causes the version number of the Monitor currently in use to be printed on the console terminal.
ZERO	Causes deletion of all files on the cassette; a sentinel file is written at the beginning of the cassette.

Error Message Summary

Monitor error messages are preceeded by one of two symbols indicating the type of error which occurred:

?	Non-fatal error; execution continues if possible, otherwise control returns to the CSI after the message is printed.
%	Fatal error; control returns to the KBL after the message is printed.

Message	Arg	Meaning	Source
IOT	PC	Illegal IOT	RESMON
NO FILE OPEN	drive #	READ or WRITE with no SEEK or ENTER	RESMON
OFFLINE	drive #	Cassette not mounted; if non-fatal, execution is automatically resumed when the cassette is mounted (if the user improperly mounts the cassette, a fatal error will probably occur)	RESMON
TIMING	drive #	System software did not service an initiated request fast enough	RESMON

KEYBOARD MONITOR (Cont.)

Message	Arg	Meaning	Source
TRAP	PC	Stack overflow, reference to non-existent memory, illegal or reserved instruction, attempt to reference a word on a byte boundary; the SP at the time of the trap is stored in location 44	RESMON
WRT LOCK	drive #	Cassette write-locked; if non-fatal, execution is automatically resumed when the cassette is write-enabled	RESMON
FILE NOT FND		Specified file not found	KBL
ILL CMD		Illegal command	KBL
NO SENTINEL FILE		No sentinel file is present on the tape; this message occurs during use of the DIRECTORY command at that point during the directory listing where the sentinel file is missing	KBL
SYNTAX ERROR		Arguments following a command are illegal	KBL
BAD TAPE		Hardware checksum error (note that this error may also be caused by READ operations initiated on a cassette which is positioned after the sentinel file)	KBL, CLOD11
NO START ADDR		Loaded program had no transfer address	KBL, CLOD11
PROG TOO BIG		Program too big for the memory limits defined by the type of load used	CLOD11
SFTWR CHKSM ERR		Software checksum error (message followed by number of errors)	CLOD11
TRUNCATED FILE		File ends before transfer address load block is found	CLOD11

KEYBOARD MONITOR (Cont.)

Message	Arg	Meaning	Source
CSI TABLE OVERFLOW		Command string too big for the table	CSI
ILLEGAL CHAR:	(C.S. line)	Illegal character in command string	CSI
ILLEGAL DEVICE:	(C.S. line)	Illegal device specification	CSI
ILLEGAL SYNTAX:	(C.S. line)	Illegal syntax in command string	CSI

C.2 EDITOR (Chapter 4)

Command Summary

Input and Output Commands

Command	Form	Meaning
EDIT READ	ER#filnam.ext$	Open a file for input.
EDIT WRITE	EW#filnam.ext$	Open a file for output.
READ	R	Read a page of text from the input file and append it to the contents of the buffer.
WRITE	nW -nW 0W W	Output a specified number of lines of text from the Text Buffer to the output file.
NEXT	nN	Output the contents of the Text Buffer to the output file, clear the buffer, and read in the next page of the input file.
LIST	nL -nL 0L L	Print a specified number of lines on the console terminal.
VERIFY	V	Print the current text line (the line containing the pointer) on the console terminal.

Command	Form	Meaning
END FILE	EF	Close the current output file without performing any further input/output operations.
EXIT	EX	Output the remainder of the input file to the output file and return control to the Monitor.

Pointer Relocation Commands

Command	Form	Meaning
BEGINNING	B	Move the current location pointer to the beginning of the line.
JUMP	nJ -nJ 0J J	Move the pointer over a specified number of characters in the Text Buffer.
ADVANCE	nA -nA 0A A	Move the pointer over a specified number of lines in the Text Buffer. The pointer is positioned at the beginning of the line.

Search Commands

Command	Form	Meaning
GET	nGtext$	Search the contents of the Text Buffer, beginning at the current location pointer, for the next occurrence of the text string.
FIND	nFtext$	Beginning at the current location pointer, search the entire text file for the nth occurrence of the specified character string. Pages of text are input, searched, and then output to the output file until the text string is found.

Command	Form	Meaning
POSITION	nPtext$	Search the input file for the nth occurrence of the text string; if the text string is not found, the buffer is cleared and a new page is read from the input file.

Text Modification Commands

Command	Form	Meaning
INSERT	Itext$	Insert text immediately following the current location pointer; an ALTMODE terminates the text.
DELETE	nD -nD 0D D	Remove a specified number of characters from the Text Buffer, beginning at the current location pointer.
KILL	nK -nK 0K K	Remove n lines from the Text Buffer, beginning at the current location pointer.
CHANGE	nC -nC 0C C	Replace n characters, beginning at the pointer, with the indicated text string.
EXCHANGE	nXtext$ -nXtext$ 0Xtext$ Xtext$	Replace n lines, beginning at the pointer, with the indicated text string.

Utility Commands

Command	Form	Meaning
SAVE	nS	Copy the specified number of lines, beginning at the pointer, into the Save Buffer.
UNSAVE	U	Insert the entire contents of the Save Buffer into the Text Buffer at the position of the current location pointer.

Command	Form	Meaning
	0U	Clear the Save Buffer and reclaim the area for text.
MACRO	M/command string/	Insert a command string into EDIT's Macro Buffer
	0M	Clear the Macro Buffer and reclaim the area for text.
EXECUTE MACRO	nEM	Execute the command string specified in the last macro command.

Error Message Summary

Message	Explanation
?"<>" ERR?	Too deep nesting, or illegal use of brackets, or unmatched brackets.
* CB ALMOST FULL *	The command currently being entered by the user is within 10 characters of exceeding the space available in the Command Buffer.
?CB FULL?	Command exceeds the space allowed for a command string in the Command Buffer.
?*EOF*?	Attempted a Read or Next command and no data was available.
?*FILE NOT FOUND*?	Attempted to open a nonexisting file for editing.
?*HDW ERR*?	A hardware error occurred during I/O.
?ILL ARG?	The argument specified was illegal for the command used, a negative argument was specified where only a positive argument was allowed, or an argument exceeded the range + or -16384.
?ILL CMD?	EDIT does not recognize the command specified.
?ILL MAC?	Delimiters were improperly used, or an attempt was made to enter an M command during execution of a Macro, or an attempt was made to execute an EM command while an EM was already in progress.

EDITOR (Cont.)

Message	Explanation
?*ILL NAME*?	The filename or device specified in an EW or ER command is illegal.
?*I/O CHAN CONFLICT*?	An attempt was made to open an input file on a cassette already open for output, or vice versa.
?*NO FILE*?	An attempt was made to Read or Write when no file was open.
?*NO ROOM*?	An attempt was made to execute an Insert, Save, Unsave, Read, Next, Change, or Exchange command when there was not enough room in the appropriate buffer.
?*SRCH FAIL*?	The text string specified in a Get, Find or Position command was not found in the available data.
?*TAPE FULL*?	Available space for an output file is full (i.e., there is no room for any part of the file).

C.3 ASSEMBLER (Chapter 5)

Language Summary

Reference may be made to Appendix B for the CAPS-11 PAL assembly language summary.

Option Summary

Option	Meaning
/C	This option allows an I/O specification line to be broken into several segments.
/F	This option is valid only after an input filename and specifies that the Assembler should not perform a REWIND operation but should continue searching the cassette in a forward direction for the file.

Option	Meaning
/O	This option is valid only after an output filename and indicates that the file (immediately preceding the option) is to be created and used only if a previously opened output file has been written to the end of the cassette and more output remains.
/P	This option is used whenever a file referenced in the I/O specification line exists on a cassette which is not currently mounted on a drive. Before attempting to search for the file, the Assembler instructs the user to mount the proper cassette on the drive by printing #? where # represents the drive number. After the user has switched cassettes on the drive, he may continue execution by typing any character on the keyboard.
/X	This option is valid only after an output filename and causes extended binary output (i.e., those locations and binary contents beyond the first binary word per source statement) to be suppressed from the listing.

Error Message Summary

Error Code	Explanation
A	Addressing error. An address within the instruction is incorrect; may also indicate a relocation error.
B	Bounding error. Instructions or word memory data are being assembled at an odd address in memory. The location counter is updated by +1.
D	Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
I	Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.
L	Line buffer overflow. Extra characters on a line (more then 72(decimal)) are ignored.

Error Code	Explanation
M	Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.
N	Number containing 8 or 9 has decimal point missing. The number is assembled as a decimal number.
P	Phase error. A label's definition or value varies from one pass to another.
Q	Questionable syntax. Missing arguments, the instruction scan was not completed, or a carriage return was not immediately followed by a line feed or form feed.
R	Register-type error. An invalid use of or reference to a register has been made.
S	Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the symbol table, the Assembler outputs the current source line with the S error code, then returns to the initial dialogue.
T	Truncation error. A number generated more than 16 bits of significance, or an expression generated more than 8 bits of significance during the use of the .BYTE directive.
U	Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.

In addition to the error codes listed above, the following messages may also occur (error messages which are followed by a question mark allow the user to type a CTRL/C to return to the KBL, or a CTRL/P to retry the operation):

Message	Meaning
%BAD CMD STRING	One of the following errors has occurred in the user's command string: No output was specified; No input was specified; Input and output were specified on the same drive; Input was specified from a device other than cassette; Binary output was specified to a device other than cassette.
?BAD TAPE?	A checksum or other hard error occurred during a file lookup or enter command. Typing any character will cause the Assembler to retry the operation.
%BAD TAPE RETRY?	A hard read error was detected on one of the input files; typing any character (other than ↑C) will cause the Assembler to retry the same assembly (in systems larger than 8K, the Assembler will return to the CSI and allow the user to input a new command).
EOM?	The line printer is out of paper or is not powered up; the drum gate is open; or the printer is too hot.
EOM? RETRY?	The end of the tape was reached during cassette output and no overflow file was specified. The user may mount another cassette and then type any keyboard character to instruct the Assembler to retry the assembly using the new output cassette.
?FILE NOT FND?	The Assembler could not find one of the input files. The user may mount another cassette and type any character on the keyboard to instruct the Assembler to retry the lookup on the same drive. Typing a CTRL/P will restart the Assembler (if the system is greater than 8K).

Message	Meaning
?NO END STMT	The file does not contain an .END directive; the Assembler assumes an .END statement.
?SWITCH ERROR:'x'?	An undefined option character (x) was found in the command string. Typing any character on the keyboard will cause the Assembler to ignore the option and continue.
?TAPE FULL?	The specified output cassette is completely full. Mounting a different cassette on the same unit and typing any character instructs the Assembler to attempt to open the file on a new cassette.

C.4 LINKER (Chapter 6)

Option Summary

Option	Meaning
/C	This option allows the I/O specification line to be broken into several segments.
/F	This option is valid only after an input filename and indicates that the Linker should not perform a REWIND operation but should continue searching the cassette in a forward direction for the file.
/O	This option is valid only after an output filename and indicates that the file (immediately preceding the option) is to be created and used only if a previously opened output file has been written to the end of a cassette and more output remains.
/P	This option is used whenever a file referenced in an I/O specification line is on a cassette which is not currently mounted on the unit drive. Before attempting to search for the file, the Linker instructs the user to mount the proper cassette on the drive by printing #? where # represents the drive number.

Option	Meaning
	After the user has switched cassettes on the drive, he may continue execution by typing any character on the keyboard.
/S	This option is valid only after an input filename and indicates that two or more object modules have been combined (using PIP) under the single filename. The option instructs the Linker not to skip to the next input filename until it has obtained all necessary information for the files included in the first.
/T	The /T option is valid only after an input filename and indicates that the transfer address of this particular object module is to be used as the transfer address of the final load module. If more than one /T option is indicated in the I/O specification line, only the last one is significant.
/B:n	The program is to be linked with its lowest location at n. If n is not specified, the Linker assumes location 600.
/H:n	The program is to be linked with its highest location at n. If n is not specified, the Linker assumes that the last location of the user program will go just under CLOD11; the user can then use the LOAD/G command to run his file.

Error Message Summary

Non-Fatal Errors

Message	Meaning
?BAD TAPE?	A checksum or other hard error occurred during a file LOOKUP or ENTER command. Typing any keyboard character instructs the Linker to retry the operation.

Message	Meaning
?BYTE RELOC ERROR AT ABS ADDRESS xxxxxx	This message designates a byte relocation error. The Linker has tried and failed to relocate and link byte quantities; the value is truncated to 8 bits, the message is printed, and the Linker automatically continues.
?FILE NOT FND?	The Linker could not find one of the input files. Typing any keyboard character instructs the Linker to retry the operation.
?MAP DEVICE EOM?	The Load Map device EOM error allows the user an option to fix the device and continue by typing any response terminated by a <CR> or <LF>, or to abort the map listing by typing a ↑P.
?MODULE NAME xxxxxx NOT UNIQUE	This error results from a non-unique object module name during Pass 1. The module is rejected and the Linker will then ask for more input.
?SWITCH ERROR:'x'?	An undefined option character was found in the command string. Typing any character instructs the Linker to ignore the character and continue.
?TAPE FULL?	The specified output cassette is full. A different cassette may be mounted on the same drive; typing any keyboard character then instructs the Linker to attempt to open the file on the new cassette.
?xxxxxx MULTIPLY DEFINED BY MODULE xxxxxx	This message results during Pass 1 if globals have been defined more than once. The second definition is ignored and the Linker continues.

Fatal Errors

Message	Meaning
%BAD CMD STRING	One of the following occurred in the command string: no output or no input specification; input and output were specified on the same drive; input was specified from a device other than cassette; binary output was specified to a device other than cassette.
%CAS. CHECKSUM	A checksum error occurred while reading a cassette block.
%ODD ADDRESS	An odd address was specified using the /B or /H options.
%SYMBOL TABLE OVERFLOW-MODULE xxxxxx, SYMBOL xxxxxx	A symbol table overflow has occurred.
%SYSTEM ERROR xx	A system error has occurred where xx represents an identifying number from the following list: 01 Unrecognized symbol table entry found. 02 A relocation directory references a global name which cannot be found in the symbol table. 03 A relocation directory contains a location counter modification command which is not last. 04 Object module does not start with a GSD. 05 The first entry in the GSD is not the module name. 06 A relocation directory references a section name which cannot be found. 07 The transfer address specification references a nonexistent module name.

Message	Meaning
08	The transfer address specification references a nonexistent section name.
09	An internal jump table index is out of range.
10	A checksum error occurred on the object module.
11	An object module binary block is too big (more than 64(decimal) words of data).
12	A device error occurred on the load module output device.

All system errors except number 12 indicate a program failure either in the Linker or the program which generated the object module. Error 05 can occur if a file is read which is not an object module.

C.5 ODT (Chapter 7)

Command Summary

Command	Meaning
r/	Open the word at location r.
/	Reopen the last opened location.
r\(SHIFT/L)	Open the byte at location r.
\	Reopen the last opened byte.
nR	After a word has been opened, retype the contents of the word relative to relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest but less than or equal to the contents of the opened location.
n!	After a word or byte has been opened, print the address of the opened location relative to relocation register n. If n is omitted, ODT selects the relocation register whose contents are closest, but less than or equal to the address of the opened location.

Command	Meaning
↓(LINE FEED key)	Open next sequential location.
↑or ^	Open previous location. (The circumflex, ^, appears on some keyboards and prints in place of the up-arrow.)
RETURN	Close open location and accept the next command.
←or _	Take contents of opened location, indexed by contents of PC, and open that location. (The underline, _, appears on some keyboards and prints in place of the back-arrow.
@	Take contents of opened location as absolute address and open that location.
>	Take contents of opened location n as relative branch instruction and open referenced location.
<	Return to sequence prior to last @, >, or ← command and open succeeding location.
X	Perform a Radix 50 unpack of the binary contents of the current opened word; then permit the storage of a new Radix 50 binary number in the same location.
r;0	Calculate offset from currently open location to r.
$n/	Open general register n (0-7).
$Y/	Open special register Y, where Y may be one of the following letters: S Status register (saved by ODT after a breakpoint) M Mask register B First word of the breakpoint table P Priority register

Command	Meaning
	C Constant register
	R First relocation register (register 0)
	F Format register
;F	Fill memory words with the contents of the constant register.
;I	Fill memory bytes with the contents of the low-order 8 bits of the constant register.
;	Separate commands from command arguments (used with alphabetic commands below); separate a relocation register specifier from an addend.
;B	Remove all Breakpoints.
r;B	Set Breakpoint at location r.
r;nB	Set Breakpoint n at location r.
;nB	Remove nth Breakpoint.
r;E	Search for instructions that reference effective address r.
r;W	Search for Words with bit patterns which match r.
;nS	Enable single-instruction mode (n can have any value and is not significant); disable breakpoints.
;S	Disable single-instruction mode; reenable breakpoints.
r;G	Go to location n and start program run.
;P	Proceed with program execution from breakpoint; stop when next breakpoint is encountered or at end of program. In single-instruction mode only, proceed to execute next instruction only.
k;P	Proceed with program execution from breakpoint; stop after encountering the breakpoint k times.

ODT (Cont.)

Command	Meaning
	In single-instruction mode only, proceed to execute next k instructions.
;R	Set all relocation registers to -1 (highest address value).
;nR	Set relocation register n to -1.
r;nR	Set relocation register n to the value of r. If n is omitted, it is assumed to be 0.
r;C	Print the value of r and store it in the constant register.
r;nA	Print n bytes in their ASCII format starting at location r; then allow n bytes to be typed, starting at location r.
CTRL/C	Return to Monitor and accept a command from the keyboard.

Error Message Summary

No error messages occur under ODT as illegal commands are ignored; ODT prints ? and the user may enter another command.

C.6 PIP (Chapter 8)

Option Summary

Option	Meaning
/A	Used with an output filename to designate that the header bit be set to ASCII (the file type is otherwise assumed to be binary).
/C	Allows the command string to be broken into one or more lines.
/D	Causes the filename(s) indicated in the command line to be deleted from the specified cassette.

Option	Meaning
/P	Requests that the system prompt the user to change cassettes on the indicated drive before an attempt is made to access the file. The system prints: #? where # represents the number of the appropriate drive. When the user has mounted the proper cassette, he may type any character on the keyboard to continue execution.
/Z	Indicates that all cassettes on the unit drives specified in the command line are to be zeroed.

Error Message Summary

Message	Meaning
?BAD TAPE ?BAD TAPE?	Hardware checksum error (may also be caused by READ operations initiated on a cassette which is positioned after the sentinel file); a question mark following the message indicates that the error is not fatal; the user may mount another cassette and type any character on the keyboard to continue execution.
?EOM	Indicates an out-of-paper condition for the line printer, console terminal, or paper tape punch.
?EXCESS INPUT FILES	The number of input files exceeds the number of output files (providing the number of output files is greater than one); this error occurs during use of the file transfer function.
?EXCESS OUTPUT FILES	The number of output files exceeds the number of input files; this error occurs during use of the file transfer function.
?FILE NOT FND?	The specified file was not found on the cassette indicated; the user may mount another cassette and type any character on the keyboard to continue the search.

Message	Meaning
?ILLEGAL DEVICE	An illegal device was indicated for the PIP function used.
?ILLEGAL INPUT LIST	An input list was indicated where not allowed (as when using the zero, delete, and copy functions), or an illegal command was entered.
?ILLEGAL OUTPUT LIST	An output list was indicated where not allowed (as when using the copy function).
?I/O CHAN CONFLICT	An attempt was made to open an input file on a cassette already open for output, or vice versa.
?NO FILE NAME	A filename was not indicated in a command line which required one.
?OFFLINE x	The cassette is not properly mounted on drive x. The user should correctly mount the cassette so that execution can continue.
?SWITCH ERROR 'x'?	An illegal switch was indicated in the command line, where 'x' represents the switch in error. The check is made for as many as 10 illegal switches in any one command line. Typing any character on the keyboard will cause PIP to ignore the switch and continue execution.
?TAPE FULL ?TAPE FULL?	Available space for an output file is full. A question mark following the message indicates that the error is not fatal; the user may mount another cassette and type any character on the keyboard to continue execution.
?WRT LOCK x	The cassette is write-locked; x represents the drive number. The user should dismount the cassette (the OFFLINE error message will then be printed), write-enable the cassette, and remount it. Execution will then continue.

C.7 RESMON (Chapter 9)

Error Message Summary

RESMON error messages are summarized in Section C.1 under the Keyboard Monitor error message summary.

APPENDIX D

SYSTEM DEMONSTRATION

The following is a brief demonstration of the CAPS-11 system software. Before proceeding with this demonstration, the user should read the rest of the CAPS-11 manual and become familiar with the CAPS-11 system programs and conventions. He should pay particular attention to the second half of Appendix E, which describes reconfiguring the CAPS-11 Monitor for non-standard I/O devices and different memory configurations. In particular, if the user's system contains a non-standard console terminal (either LT33 or LT35, parallel LA30 DECwriter, or VT05 display), he should use his reconfigured System Cassette for this demonstration.

Before starting, the user should have ready the proper CAPS-11 System Cassette and two scratch cassettes. The first step of the demonstration is to copy the System Cassette. The demonstration should then be continued using this newly created copy.

In general, the user should always keep at least one good copy of the System Cassette in a safe place in the event that he should accidently destroy his 'working copy'. Note that this demonstration uses the System Cassette when it is write-enabled. The purpose is to simplify the demonstration; under normal operation, the user should always use the System Cassette write-locked.

Please read through the entire system demonstration before attempting to enter any of the command lines.

D.1 SYSTEM START-UP

Write-lock the CAPS-11 System Cassette by setting the hinged red tabs so that they are pointed toward the center of the cassette, exposing the write-protect holes; mount the cassette on drive 0 (the drive to the left of the unit). Bootstrap the CAPS-11 Monitor into memory using the procedure described in Chapter 3, Section 3.1. When loaded, the Monitor will type an identification line and a dot at the left margin of the console terminal page (subsequent loads will cause only the dot to be printed):

If this does not occur, check that the terminal is turned on and that the cassette is mounted properly and retry the bootstrap procedure.

D.2 SYSTEM DEMONSTRATION

This section demonstrates briefly how to use the CAPS-11 system programs by presenting them in the context of a simple exercise. The user will copy the System Cassette, and then edit, assemble, link, load, and run a simple demonstration program. In the following discussion, computer output is underlined when necessary to differentiate it from user input; a) is used to indicate typing the RETURN key and $ indicates typing the ALT MODE key. Mistakes made while entering command strings may be corrected by typing the RUBOUT key.

Once the CAPS-11 Monitor has been bootstraped into memory and has typed a dot, enter the current date by typing a command of the form:

```
.DA 27-AUG-73 )
```

substituting the current date in place of 27-AUG-73. The Monitor indicates that it is ready to accept another command by printing a dot at the left margin of the page. When this dot appears, enter the following command:

```
.V )
```

The Version command causes the Monitor to print out the version number of the Monitor in use. The Monitor should respond by printing:

```
CAPS-11 V01-02
27-AUG-73
```

After the user has verified that the version he is using is the correct one, he should next copy the System Cassette (or his reconfigured System Cassette) using PIP. Mount a scratch cassette on unit 1, write-enabled, and type:

```
.R PIP )
```

The Command String Interpreter (CSI) will print an asterisk at the left margin of the page when it is ready to accept a command line. Type the following command to copy the System Cassette to the scratch cassette on drive 1 (the command may be entered as soon as the asterisk is printed even though program loading may be occuring simultaneously):

```
*1:=0:)
```

This command causes the output cassette (on drive 1) to be zeroed, and then copies the entire cassette from drive 0 to drive 1. When the copy function is complete, the CSI will type another asterisk. Now type CTRL/C to return to the CAPS-11 Monitor, which will print a dot:

```
*^C )
.
```

Dismount the System Cassette on unit 0 and put it away. From now on, the copy of the System Cassette just created should be used in the demonstration. Dismount this copy from unit 1 and mount it, write-enbaled, on unit 0, then type:

```
.DI )
```

The Monitor will list the directory of the copy just produced--a typical directory will appear as follows:

```
27-AUG-73

CTLOAD  SYS  08-AUG-73
CAPS11  S8K  09-AUG-73
PIP     SRU  09-AUG-73
EDIT    SLG  09-AUG-73
LINK    SRU  09-AUG-73
ODT     SLG  09-AUG-73
PAL     SRU  09-AUG-73
DEMO    PAL  09-AUG-73
```

Attach this directory listing to the System Cassette on unit 0. Now mount another scratch cassette on unit 1, write-enabled, and zero it by typing:

```
.Z 1: )
```

Again, the Monitor will print a dot when ready for the next command. Type in response to this dot:

```
.R PAL )
```

This command loads and starts the CAPS-11 Assembler. When the Assembler is found on the System Cassette, the CSI will print an asterisk at the left margin of the console terminal page. When the asterisk appears, type the following line:

```
*=DEMO )
```

The command instructs the Assembler to assemble the demonstration program stored on unit 0 (DEMO.PAL) and print any errors on the console terminal. The following error message is printed, indicating that there is an illegal character in the demonstration source file which must be corrected before the file can be assembled properly:

```
PASS 2
I 000026 000167'          JMP     *KBLADR     ;BACK TO CAPS-11 MONITOR
         000000

000001 ERRORS

.C?)

.
```

In 8K systems, the CAPS-11 Assembler overlays part of the Monitor so that the system must be re-bootstrapped after an assembly has been completed. The assembler signals the user that it is done and ready to re-bootstrap the system by typing:

```
.C?
```

The user should ensure that the System Cassette is still mounted on unit 0 and then type any character on the console keyboard--the entire CAPS Monitor will be bootstrapped into memory and will print a dot when it is ready for a command.

If the user is running with a 12K or larger CAPS-11 Monitor, the assembler will not overlay the Monitor; thus when the assembly of the demo program is complete, the assembler will transfer control to the Command String Interpreter which will print an asterisk. In this case, return to the CAPS-11 Monitor by typing:

```
*^C)
```

Again, the Monitor will print a dot when ready for a command.

To correct the error detected in the assembly just performed, the CAPS-11 Editor must be called from unit 0. Type:

```
.R EDIT )
```

This command loads and starts the CAPS-11 Editor. The Editor will print an asterisk when it is ready to accept a command. Now enter the following line ($ represents typing the ALTMODE key):

```
*ER0:DEMO$$
```

There will be a pause before the next asterisk is printed since the Editor is searching the System Cassette for the file DEMO.PAL. When the asterisk appears, type:

```
*EW1:DEMO1$$
```

This command instructs the Editor to open an output file on unit 1; when another asterisk appears, type the command:

```
*RG*$V$$
```

This will read text into the Text Buffer from the input file, search the buffer for a line containing an * and leave the pointer positioned immediately following the *. The Editor should type:

```
JMP     *KBLADR          ;BACK TO CAPS-11 MONITOR
```

This line of text contains an error which must be corrected. Type in response to the asterisk:

```
*-C@$V$$
```

This deletes the *, replaces it with @ and verifies the line. The Editor should now type:

```
JMP     @KBLADR          ;BACK TO CAPS-11 MONITOR
```

Close the output file by typing:

```
*EX$$
```

When control has returned to the CAPS-11 Monitor, a dot will be printed. Next, run the Assembler by typing:

```
.R PAL )
```

When the Assembler is found on the System Cassette, the CSI will print an asterisk. Type the following command to assemble the edited file, putting the object module on the System Cassette and printing the listing on the console terminal (the command may be entered as soon as the asterisk appears; the Assembler will be simultaneously loaded into memory):

```
*DEMO1,TT:=1:DEMO1)
```

If the user's system includes a line printer, 'LP': may be substituted for 'TT: in the above command to cause the assembly listing to be printed on the line printer rather than the console terminal.

The assembler will type 'PASS 2' and then print the listing as follows:

```
PASS 2

CAPS11 PAL V01 08/27/73 PAGE    001

                .TITLE CAPS-11 DEMO PROGRAM
                ;
                ;CAPS-11 DEMONSTRATION PROGRAM
                ;
                ;DEC-11-OTDMA-A-LA
                ;COPYRIGHT 1973 DIGITAL EQUIPMENT CORPORATION
                ;MAYNARD, MASSACHUSETTS 01754
                ;DEC ASSUMES NO RESPONSIBILITY FOR THE USE OR
                ;RELIABILITY OF ITS SOFTWARE ON EQUIPMENT WHICH IS NOT
                ;SUPPLIED BY DEC.
                ;
                ;
                        .GLOBL  START
                ;
                ;ADDRESS OF CAPS-11 KBL IS IN LOCATION 50
                ;
```

```
        000050          KBLADR=50
        000015          CR=15
        000012          LF=12
000000 012706 START:    MOV     #600,%6         ;SET STACK POINTER
        000600
000004 000004           IOT                     ;RESET CAPS-11
000006    001           .BYTE   1,0
000007    000
000010 000000           .WORD   0
000012 000004           IOT                     ;TYPE MESSAGE ON CONSOLE
000014    004           .BYTE   4,2
000015    002
000016 000032'          .WORD   MSGBUF
000020 000004 WAIT:     IOT                     ;WAIT FOR IT TO FINISH
000022    003           .BYTE   3,2
000023    002
000024 000020'          .WORD   WAIT
000026 000177'          JMP     @KBLADR         ;BACK TO CAPS-11 MONITOR
        000050

000032 000100 MSGBUF:   .WORD   100             ;MAX.SIZE OF BUFFER
000034    000           .BYTE   0,0             ;MODE IS FORMATTED ASCII
000035    000
                                                ;STATUS BYTE IS 0
000036 000054           .WORD   MSGEND-MSGBUF-5 ;BYTE COUNT
000040    015           .BYTE   CR,LF
000041    012
000042    103           .ASCII  /CAPS-11 DEMNOSTRATION /
000043    101
000044    120
000045    123
000046    055
000047    061
000050    061
000051    040
000052    104
000053    105
000054    115
000055    116
000056    117
000057    123
000060    124

CAPS11 PAL V01 08/27/72 PAGE    002

000061    122
000062    101
000063    124
000064    111
000065    117
000066    116
000067    040
000070    120           .ASCII  /PROGRAM COMPLETE./
000071    122
000072    117
000073    107
000074    122
000075    101
000076    115
```

```
000077    040
000100    103
000101    117
000102    115
000103    120
000104    114
000105    105
000106    124
000107    105
000110    056
000111    015          .BYTE   CR,LF
000112    012
000113    012 MSGEND:  .BYTE   LF
       000000          .END    START

CAPS11 PAL V01 08/27/73 PAGE    003

 CR     = 000015       KBLADR = 000050       LF     = 000012
 MSGBUF   000032R      MSGEND   000113R      START    000000RG
 WAIT     000020R      .      = 000114R


000000 ERRORS
```

↑C?

.

Again, in 8K systems, the system must be re-bootstrapped after an assembly has been completed, thus the assembler signals the user that it is done and ready to re-bootstrap the system by typing:

↑C?

Ensure that the System Cassette is still mounted on unit 0 and then type any character on the console keyboard to re-boot the system.

If the user is running with a 12K or larger CAPS-11 Monitor, the assembler will not overlay the Monitor; in this case, return to the CAPS-11 Monitor by typing:

*↑C ↵

The Monitor will print a dot when ready for a command.

The assembler's output must be linked before it can be loaded and started. Run the Linker by typing:

.R LINK ↵

When the Linker is found on the System Cassette, the Command String Interpreter will print an asterisk. Type the following command to link the assembler output file, outputting the load module (DEMO.LDA) to the scratch cassette on unit 1 and printing a load map on the console; as with the assembler, 'LP:' may be substituted for 'TT:' if the system contains a line printer:

```
*1:DEM01,TT:=DEM01/B)

CAPS-11 LINK V01  08/27/73
LOAD MAP

TRANSFER ADDRESS: 000600
LOW LIMIT: 000600
HIGH LIMIT: 000714
********
CAPS
SECTION          ADDRESS SIZE

<. ABS.>         000000  000000

<       >        000600  000114
START   000600

PASS 2
```

When the Linker has finished, it will transfer control back to the CSI, which will print an asterisk. Type CTRL/C to return control to the CAPS-11 Monitor:

```
*^C)

.
```

Now type the following command to load the demonstration program into memory:

```
.LOAD 1:DEM01.LDA )
```

Once the program has been loaded, the Monitor will print a dot. Next run the debugging program, ODT, by typing:

```
.R ODT )
```

When ODT has been loaded into memory, it will type:

```
ODT V01

*
```

From the load map printed by the Linker, notice that the lowest memory address occupied by the program is 600(octal). Therefore, set relocation register zero to 600 by typing:

```
*600;0R)
```

Next, use ODT to correct a spelling error in the output line; this correction is made only in memory and not permanently in the source file. (In the following example, data typed by ODT is underlined; note that) indicates typing a carriage return; \ is typed on an LT33 or LT35 by pressing the SHIFT and L keys simultaneously):

```
*0,55\116 =N  117)
*0,56\117 =O  116)
*
```

Start the program using the 'Go' command in ODT; the demo program will type a message and then return to the CAPS-11 Monitor, which prints a dot:

```
*600;G
CAPS-11 DEMONSTRATION PROGRAM COMPLETE.

.
```

APPENDIX E

CAPS-11 SOFTWARE SUPPORT INFORMATION

E.1 CAPS-11 KEYBOARD MONITOR LOADING PROCESS

The CAPS-11 Monitor loading process is initiated when the user loads the bootstrap loader (CBOOT) into memory either through use of the hardware bootstrap or manually via the Switch Register. CBOOT calls the first program on the System Cassette, CTLOAD.SYS, and from there, as far as the user is concerned, system loading is automatic. A detailed description of this loading process follows.

E.1.1 Cassette Bootstrap (CBOOT)

The Cassette Bootstrap is used to load and start any program which is written in "CBOOT Loader Format" and is contained entirely in a 128 (decimal) byte record; this record must be the first data record of the first file on a cassette.

"CBOOT Loader Format" programs are defined to be those of length less than or equal to 128 (decimal) bytes which are linked so as to be loaded in memory beginning at location 0. A program in this format begins execution at its first instruction, which must be NOP (=000240).

CBOOT verifies that the first byte in the program contains 240 as a method of detecting accidental attempts to boot a program in the wrong format. If this occurs, or upon occurrence of any I/O error, CBOOT halts at location CBOOT+50; at this time the user may examine either location 0 (which will contain the first byte of the program being loaded) or the cassette control and status register (TACS=777500) to determine the cause of the error. The user may restart CBOOT by pressing the CONTinue switch on the computer console.

CBOOT may be executed using an optional hardware bootstrap or it may be manually loaded by the user. Although CBOOT may be loaded anywhere in memory (with the exception of locations 0-177), it is normally loaded at location 1000, and references in this appendix will use that address.

Memory Map #1 in Figure E-3 illustrates a map of PDP-11 memory following loading of CBOOT. CBOOT is normally used to load PRELDR, which is the first record of the first file on the CAPS-11 System Cassette. (Listings of CBOOT and QCBOOT are provided in Figures E-4 and E-5.)

E.1.2 Cassette Loader (CTLOAD.SYS)

The first file on the System Cassette is CTLOAD.SYS, which consists of a data record called PRELDR followed by succeeding data records making up the program CABLDR (which ends with a copy of CBOOT), as follows:

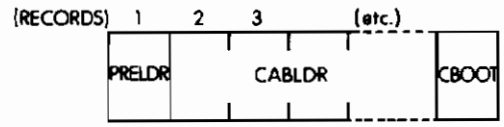


Figure E-1 CTLOAD.SYS

As seen in Figure E-1, PRELDR is the first record of the first file on the System Cassette. This cassette pre-loader is actually a small program written in "CBOOT Loader Format" which is powerful enough to determine memory size and load succeeding programs into highest memory. It is linked, loaded, and started automatically by CBOOT at location 0. A map of CAPS-11 memory now appears as shown in Memory Map #2 of Figure E-3.

The program loaded by PRELDR may be of any size, but it must exist as data in 128(decimal) byte records immediately following the record containing PRELDR, and it must be written in "PRELDR Format". PRELDR format data consists of two bytes (low-order, then high-order) containing the byte count for the rest of the data (<=77777), followed by a memory image (i.e., data only) of the program written so as to begin at its first location.

PRELDR first determines memory size, then loads this object program into highest memory (thus, the program must either be linked into highest memory or written in position independent code). If an error (generally a hardware error) occurs during loading, PRELDR halts with the contents of the cassette control and status register in Register 4. To restart the PRELDR loading process, the user should press the CONTinue key on the computer console.

The programs loaded by PRELDR are CABLDR and CBOOT, which are loaded into memory as illustrated in Memory Map #3 (Figure E-3).

NOTE

Information provided thus far assumes the user is specifically loading the CAPS-11 Keyboard Monitor from the System Cassette mounted on drive 0. In order to allow booting from cassette drive #1 or from a secondary controller, PRELDR assumes that Register 0 contains the address of the desired controller and that the appropriate drive has been selected. This should be done manually by the user before booting (it is done automatically by CBOOT during normal loading).

The main data portion of the file CTLOAD.SYS is CABLDR, the Cassette Absolute Loader. This program is used to load programs written in "Absolute Binary Format", which is the format of all system programs and all Linker output. Absolute binary format consists of a number of 'load blocks' of memory image load data with associated header information; such a load block has the following general form:

Table E-1
Absolute Binary Load Block Format

Byte #	Contents
1	001
2	000
3	Byte Count-low order
4	Byte Count-high order
5	Load Address-low order
6	Load Address-high order
.	Memory Image Data
.	.
.	.
.	.
last byte	Checksum

If the byte count of a load block is greater than 6, data is loaded into memory. If the byte count of a load block is equal to 6, the load address specified in the load block will be considered to be the desired transfer, or starting, address of the program; if this address is odd, CABLDR will halt. (It is not possible for the byte count of a load block to be less than 6.)

Immediately after being loaded into highest memory, CABLDR is started and checks the contents of the Switch Register, which must have been previously set by the user for one of the conditions listed in Table E-2:

Table E-2
CABLDR Switch Register Settings

Switch Register	Action
Bit #0=0	Normal load; use loading and starting addresses as specified in load blocks. (This is the switch setting used during the CAPS-11 system load.)
Bit #0=1 and Bit #15=1	Relocating load; CABLDR halts so that the user can set the Switch Register to the address at which the program is to be relocated (called the load bias; the program must be position independent). Bit 0 of this Switch Register setting is ignored. The user begins the load by pressing CONTinue on the computer console.
Bit #0=1 and Bit #15=0 and Bits #1-14=0	Contiguous relocating load; the program is loaded immediately following the last byte of a previously loaded program.
Bit #0=1 and Bit #15=0 and Bits #1-14=n	Non-contiguous relocating load; n-1 files are skipped and the program is positioned before the nth file; CABLDR halts for further user action.

Data will be loaded in standard cassette files with a fixed record size (128 bytes). CABLDR checks the continuation byte in the file header record (see the Cassette Standard in Appendix F), and allows for an additional header record if this byte contains 1. Record size is determined from the proper header locations.

If CABLDR halts during operation, the user may examine the contents of Register 4 to determine the reason for the halt as follows:

Table E-3
CABLDR Halts

R4 Contents	Meaning
1	File skip complete; the user should reset the Switch Register for the next desired action (see Table E-2) and then press CONTinue to load.
2	Command to relocate noted; the user should set the relocation address in the Switch Register and press CONTinue to begin the load.

(Continued on next page)

Table E-3 (Cont.)
CABLDR Halts

R4 Contents	Meaning
3	File has no fixed record length; since CABLDR cannot handle this type of file, the user should press CONTinue to cause CABLDR to skip to the next file.
4	No transfer address was found in the last load block; the user should set the address in the Switch Register and press CONTinue to go to the next file.
1xxxxx	Hardware error; The contents of the cassette and control status register are displayed in Register 4. Three basic types of hardware errors may occur:

Error	Action
Off-Line, Write-Lock	Press CONT to retry function.
Clear Leader, File Gap	File is not in legal LDA format and is ignored. Set the switch register (refer to Table E-2) to indicate which file to skip to, or insert another cassette, and press CONT.
Timing and Block Check	Retry function 3 times before halting again. Pressing CONTinue causes CABLDR to skip the current file and try the next.

Software checksum errors are noted but do not affect the loading process. At the termination of loading, the last location in CABLDR (SFTCHR) will contain the number of software checksum errors encountered.

NOTE

In order to allow booting from cassette drive 1 or from a secondary TA11 controller, CABLDR assumes (on entry) that Register 0 contains the desired controller address with the appropriate drive selected. This is ordinarily done

by CBOOT, or manually by the user before booting.

Once in memory, CABLDR may be started manually as follows:

1. Select appropriate drive in desired control and status register;

2. Deposit that controller address into R0;

3. Start CABLDR at location x6570 where x corresponds to memory size as follows:

x	Memory Size
1	4K
3	8K
5	12K
7	16K
11	20K
13	24K
15	28K

To load from drive 0 of the cassette control and status register (=777500) without setting up R0, start CABLDR at location x6572.

E.1.3 Cassette Monitor (CAPS11.SYS)

In the Monitor loading process, the file loaded by CABLDR is CAPS11.SYS, which is made up of two programs, CSYSLD.LDA and CAPS11.LDA, as follows:

Figure E-2 CAPS11.SYS

CSYLD.LDA is loaded first by CABLDR at location 1100 (refer to Memory Map #4 of Figure E.3) and is simply a special version of CABLDR modified to load a program consisting of 128 byte records; the load begins with the byte immediately following CSYSLD.LDA; this is the first byte of the second file comprising CAPS11.SYS--CAPS11.LDA. Part of this load overlays the normal CABLDR originally stored in high memory, and part is loaded into low memory, overlaying PRELDR. A map of memory now exists as shown in Memory Map #5 (Figure E-3).

The CAPS-11 System is now fully loaded into memory.

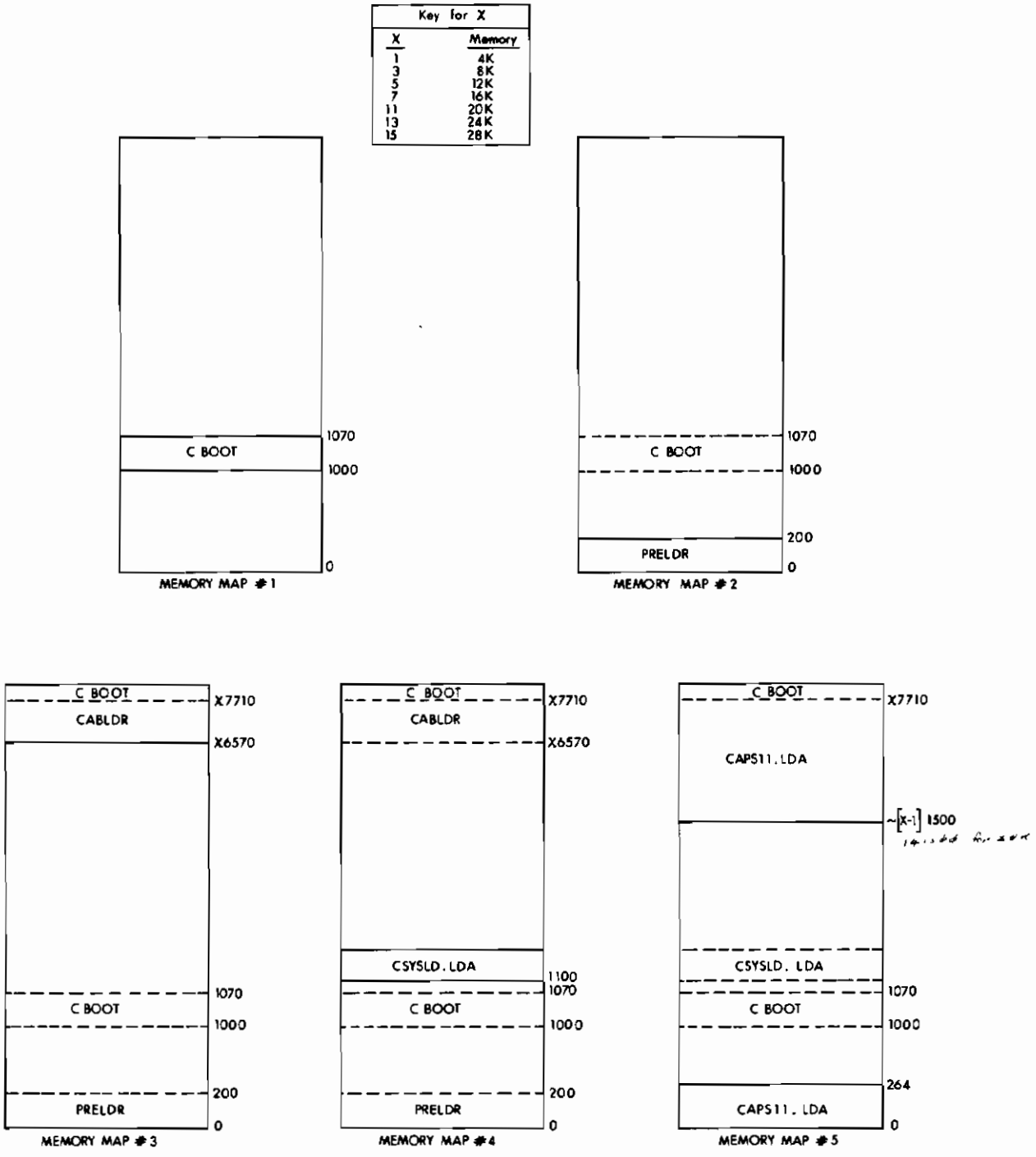


Figure E-3 CAPS-11 Loading Process

```
CAPS11 PAL V01          PAGE    001

                .TITLE  CBOOT (CAPS-11) V01-06  5/30/73
                ;
                ; PDP-11 CASSETTE BOOTSTRAP
                ;
                ; COPYRIGHT 1973 DIGITAL EQUIPMENT CORPORATION,
                ;    MAYNARD,MA.
                ;
                ; DEC-11-OTCBA-A-LA
                ;
                ; BY: P. JANSON
                ;
                ;   CBOOT WILL LOAD AND SUCCESSFULLY START
                ;   ANY PROGRAM WHICH IS WRITTEN IN 'CBOOT
                ;   LOADER FORMAT' AND IS CONTAINED ENTIRELY
                ;   IN A 128. (DECIMAL) BYTE RECORD WHICH IS THE
                ;   FIRST DATA RECORD OF THE FIRST FILE
                ;   OF A CASSETTE.
                ;
                ;   CBOOT IS POSITION INDEPENDENT.
                ;
                ;   TO BOOT FROM UNIT #1 OR FROM SOME OTHER
                ;   CONTROLLER, SET UP R0 WITH THE DESIRED
                ;   CONTROLLER ADDRESS AND SET THE UNIT SELECT
                ;   BIT AS DESIRED, THEN START THE BOOTSTRAP
                ;   AT THE THIRD INSTRUCTION.  PRELDR AND
                ;   CABLDR WILL USE R0 AS SET-UP AT BOOTSTRAP
                ;   TIME TO CONTINUE LOADING FROM THE
                ;   SELECTED UNIT.
                ;
                ;   CBOOT DOES A REWIND, SPACES FORWARD
                ;   A RECORD (TO SKIP OVER HEADER OF FIRST
                ;   FILE), AND STARTS READING THE NEXT RECORD.
                ;   A CRC CHECK IS MADE AT THE END OF THE RECORD.
                ;
                ; SIZE = 28. WORDS

CAPS11 PAL V01          PAGE    002

                ;
                        .GLOBL  CBOOT
                ;


        000000          .CSECT
                ;
        000000 R0=%0
        000001 R1=%1
        000002 R2=%2
        000003 R3=%3
        000007 PC=%7
        177500 TACS=177500      ;TA-11 CONTROL AND STATUS REG.
                ;
```

Figure E-4 CBOOT

```
000000 012700 CBOOT:  MOV     #TACS,R0
       177500
000004 005010         CLR     (R0)            ;SELECT UNIT #0
000006 010701 RESTRT: MOV     PC,R1           ;USE FOR PIC
000010 062701         ADD     #TABLE-.,R1     ;R1 HOLDS ADDR. OF
       000052                                 ;COMMAND TABLE
000014 012702         MOV     #375,R2         ;MEMORY PTR. AND
       000375                                 ;DATA FLAG
000020 112103         MOVB    (R1)+,R3        ;TEST BITS
                ;
000022 112110 LOOP1:  MOVB    (R1)+,(R0)      ;COMMAND FROM TABLE
000024 100413         BMI     DONE            ;TO TACS. WHEN COMMAND
                                              ;CODE NEG., QUIT
000026 130310 LOOP2:  BITB    R3,(R0)         ;TEST READY AND T-REQ.
                                              ;BITS IN TACS
000030 001776         BEQ     LOOP2           ;LOOP 'TILL SOMETHING
                                              ;COMES UP
000032 105202         INCB    R2              ;ADVANCE MEMORY PTR.
000034 100772         BMI     LOOP1           ;IF MINUS, TRY NEXT
                                              ;COMMAND
000036 116012         MOVB    2(R0),(R2)      ;READ DATA INTO MEMORY
       000002
000042 120337         CMPB    R3,@#0          ;FIRST BYTE READ
       000000
                                              ;SHOULD BE '240'
000046 001767         BEQ     LOOP2           ;IF O.K., GO READ
                                              ;ANOTHER BYTE
000050 000000 STOP:   HALT                    ;HALT ON ERROR
000052 000755         BR      RESTRT          ;RESTART ON CONTINUE
                ;
000054 005710 DONE:   TST     (R0)            ;CHECK FOR ERROR
000056 100774         BMI     STOP            ;HALT ON ERROR
000060 005007         CLR     PC              ;= 'JMP @#0'
                ;
000062 017640 TABLE:  .WORD   17640           ;.BYTE 240:
                                              ;READY+T-REQ.
                                              ;.BYTE  37:
                                              ;ILBS+READY+GO
000064 002415         .WORD   2415            ;.BYTE  15: SFB+GO
                                              ;.BYTE   5: READ+GO
000066 112024         .WORD   112024          ;.BYTE  24: READ+ILBS
                                              ;.BYTE 224:
                                              ;READ+ILBS+E.O.TABLE

       000001         .END
```

Figure E-4 CBOOT (Cont.)

```
CAPS11 PAL V01 06/05/73 PAGE    001

                        .TITLE  QCBOOT  V01-05  5/20/73
                    ;
                    ; PDP-11 QUICK CASSETTE BOOTSTRAP
                    ;
                    ; COPYRIGHT 1973 BY DIGITAL EQUIPMENT CORP.,
                    ;    MAYNARD, MA.
                    ; BY: ROY FOLK
                    ;
                    ;  LOADS ONE RECORD UNTIL ERROR.  LOADS AT 0.
                    ;  DOES REWIND AND SKIPS FIRST RECORD OF FIRST FILE.
                    ;  STARTS LOADED PROGRAM AT 0.
                    ;  THE CODE IS POSITION INDEPENDENT.
                    ;
                    ;  TO BOOT FROM DIFFERENT UNIT OR CONTROLLER, SET
                    ;  UP R0 AND CORRESPONDING TACS REGISTER AS DESIRED
                    ;  MANUALLY AND START AT THIRD INSTRUCTION OF BOOT.
                    ;
                    ;  SIZE = 20. WORDS
                    ;
            000000          .CSECT
                    ;
            000000  R0=     %0
            000001  R1=     %1
            000002  R2=     %2
            000007  PC=     %7
            177500  TACS=   177500  ;TA-11 CONTROL AND STATUS REGISTER
                    ;
000000  012700  QCBOOT: MOV     #TACS,R0
        177500
000004  005010         CLR     (R0)            ;SELECT UNIT #0
000006  010701         MOV     PC,R1           ;LOAD HEAD OF TABLE
000010  062701         ADD     #TABLE-.,R1     ;INTO REG.  PIC'LY
        000034
000014  112102         MOVB    (R1)+,R2        ;= 'MOV #177775,R2'
000016  112110  FUNC:  MOVB    (R1)+,(R0)      ;SELECT FUNCTION AND
000020  032710  LOOP:  BIT     #100240,(R0)    ;GO TEST ERROR, READY,
        100240
                                                ;TREQ.
000024  001775         BEQ     LOOP            ;LOOP 'TILL SOMETHING
                                                ;HAPPENS....
000026  100001         BPL     GOON            ;NO ERROR - GO ON
000030  005007         CLR     PC              ;START PROG. ON FIRST
000032  005202  GOON:  INC     R2              ;ERROR COUNTER AND
                                                ;MEMORY ADDR.
000034  100770         BMI     FUNC            ;DO REWIND, SFB, READ;
                                                ;THEN GET BYTES
000036  116012         MOVB    2(R0),(R2)      ;ACTUAL LOAD
        000002
000042  000766         BR      LOOP            ;WAIT, THEN GET MORE
                                                ;UNTIL ERROR
                    ;
000044  017775  TABLE:  .WORD   17775           ;.BYTE 375: FOR
                                                ;R2 COUNTER
                                                ;.BYTE  37:
                                                ;REWIND+ILBS+GO
000046  002415         .WORD   2415            ;.BYTE  15: SFB+GO
                                                ;.BYTE   5: READ+GO
                    ;
            000001          .END
```

Figure E-5 QCBOOT

E.2 BUILDING MEMORY CONFIGURATIONS FOR THE CAPS-11 SYSTEM

A CAPS-11 System configured for 8K is stored on the System Cassette included in the CAPS-11 software package. Upon first receiving the system, the user should read the documentation to familiarize himself with CAPS-11.

If his hardware includes additional memory or a non-standard terminal or line printer, the user will want to reconfigure his CAPS-11 system to take advantage of this hardware. The Linker stored on the 8K System Cassette is used for the reconfiguration process. In addition, the user should have ready two blank cassettes and the two OBJ Cassettes containing the following directories:

OBJ Cassette #1		OBJ Cassette #2	
CSYSLD	LDA	KBLRES	OBJ
KBLRES	OBJ	PAL	OBJ
KBL	OBJ	P12SYM	OBJ
CABLDR	OBJ	P16SYM	OBJ
CSI	OBJ	LINK	OBJ
CLOD11	OBJ	CSITAC	OBJ
RESMON	OBJ	P8SYM	OBJ
CBOOT	OBJ		
LA30P	OBJ		
VT05	OBJ		
LP80	OBJ		
ODT	OBJ		
PIP	OBJ		
CSINBF	OBJ		
EDIT	OBJ		

Ensure that the 8K System Cassette and the two OBJ Cassettes are write-locked; write-enable the two blank cassettes. Mount the 8K System Cassette on drive 0 and bootstrap the CAPS-11 System (refer to Chapter 3).

To reconfigure the Monitor files, the user must first consider which hardware options are present on his system. Standard hardware devices include serial LA30 and 132-column line printer. Non-standard devices include parallel LA30, LT33 or LT35 Teletype, VT05 and 80-column line printer. If reconfiguration is necessary because of non-standard devices, the user will find it helpful at this point to patch the Monitor so that the non-standard devices (in this case, specifically the console terminal) can be used more efficiently during the reconfiguration process itself. To make the patch, follow the procedure listed below:

1. After the 8K CAPS-11 Monitor has typed the version message on the terminal, set the ENABLE/HALT switch to HALT
2. Set the Switch Register to 000056
3. Depress LOAD ADDRESS
4. If the console terminal is a:
   a. LT33 or LT35 Teletype or parallel LA30, set the Switch Register to 000000; go to step 5
   b. VT05, set the Switch Register to 002012; go to step 5

5. Raise the DEPosit switch

6. Set the ENABLE/HALT switch to ENABLE

7. Depress the CONTinue key

NOTE

The user is advised to read through the remainder of this section before entering any of the following command lines. All command lines are terminated by a carriage return (↵).

After the patch has been made, continue with the system reconfiguration by mounting one of the blank cassettes on drive 1; enter the current date and zero the blank cassette using the commands:

```
.DA dd-mmm-yy ↵          (day-month-year)

.Z 1:↵
```

Next run the 8K System Linker and enter the command line and a carriage return as shown below; the command line may be entered as soon as the asterisk is printed even though the Linker is being simultaneously loaded into memory.

```
.R LINK↵

*1:CAPS11,TT:=KBL/P,CABLDR/F,CSI/F,CLOD11/F,RESMON/F/C ↵
```

Because of the /C option, the Linker will not initiate action until the second half of the command line is typed; see Section E.2.1, following.

NOTE

If the system includes a line printer, the output device specification LP: may be substituted for TT: in all command lines described in this section. If the user does not desire a listing of the load map, he may omit the listing output specification entirely from all command lines.

E.2.1 Reconfiguring the Monitor

To reconfigure the Monitor files for a standard system (i.e., one that includes the standard devices--serial LA30 and 132-column line printer) continue the previously entered command line by entering the following:

```
,CBOOT/F/H:xxxxxx↵
```

The response for the /H option depends upon the size of the system to be reconfigured as follows:

Table E-4
Monitor /H Option Responses

Memory Size	xxxxxx
12K	60000
16K	100000
20K	120000
24K	140000
28K (or larger)	160000

To allow reconfiguration for non-standard devices, modifications must be made to the continued portion of the command line. The user should choose the command line which corresponds to his hardware configuration from the descriptions which follow.

PARALLEL LA30 OR TELETYPE

If the console terminal is a parallel LA30 or LT33 or LT35 Teletype, the second line of the command string must be entered as follows:

```
,CBOOT/F,LA30P/F/H:xxxxxx↵
```

The response for /H is taken from Table E-4.

PARALLEL LA30 OR TELETYPE AND 80-COLUMN LINE PRINTER

If, in addition to a parallel LA30, LT33 or LT35 Teletype, the system includes an 80-column line printer, the second line of the command string becomes:

```
,CBOOT/F,LA30P/F,LP80/F/H:xxxxxx↵
```

Again, the response for /H is taken from Table E-4.

VT05

If the console terminal is a VT05, the second line of the command string is the following:

```
,CBOOT/F,VT05/F/H:xxxxxx↵
```

The response for /H is taken from Table E-4.

VT05 AND 80-COLUMN LINE PRINTER

If the system includes a VT05 and an 80-column line printer, the command line must be entered as follows:

```
,CBOOT/F,VT05/F,LP80/F/H:xxxxxx )
```

The response for /H is again chosen from Table E-4.

Thus, for example, if the system includes a VT05 and an 80 column line printer and is to be reconfigured for 16K, the entire command line would be entered as:

```
*1:CAPS11,LP:=KBL/P,CABLDR/F,CSI/F,CLOD11/F,RESMON/F/C )
,CBOOT/F,VT05/F,LP80/F/H:100000 )
```

When the entire command line has been entered followed by a carriage return, a prompt message will occur (0?); mount OBJ Cassette #1 (containing the files KBL.OBJ, CABLDR.OBJ, etc.) on unit 0 and type any character on the keyboard to continue execution. When the prompt message occurs for pass 2, again respond by typing any character on the keyboard. After the command has been executed, control returns to the Linker which prints an asterisk indicating that it is ready to receive another command. If the system includes 8K of memory and the user is reconfiguring the Monitor only to take advantage of a non-standard device, his reconfiguration is complete and he should skip to Section E.2.6 to create his new System Cassette. If the system includes more than 8K, continue the reconfiguration process as described below.

E.2.2 Reconfiguring PAL

Rewind OBJ Cassette #1 and then mount OBJ Cassette #2 (containing PAL.OBJ, etc.) on drive 0 and enter the following:

```
*1:PAL.SRU,TT:=KBLRES,PAL/F/C )
,P12SYM/F[or P16SYM/F],CSITAC/F/B:400 )
```

The file P12SYM is used for reconfiguring PAL for a 12K system; P16SYM is used for all systems which are 16K or more.

E.2.3 Reconfiguring LINK

The 8K Linker contains room for approximately 225(decimal) symbols; if the user needs more, he can next reconfigure LINK as follows:

```
*1:LINK.SRU,TT:=LINK,CSITAC/F/H:xxxxxx )
```

where xxxxxx represents one of the following:

Table E-5
Linker and ODT /H Option Responses

Memory Size	xxxxxx
12K	41500
16K	61500
20K	101500
24K	121500
28K (or larger)	141500

E.2.4 Reconfiguring ODT

Rewind OBJ Cassette #2. ODT is next reconfigured by mounting OBJ Cassette #1 on unit 0 and entering the following command line:

```
*1:ODT.SLG,TT:=KBLRES,ODT/F/H:xxxxxx )
```

The value for xxxxxx is also chosen from Table E-5.

E.2.5 Reconfiguring PIP and EDIT

There is no need to relink PIP or the Editor since these programs use the same amount of memory in any size system.

The user is now ready to create the new System Cassette.

E.2.6 Creating a New System Cassette

Return to the Monitor by typing:

```
^C )
```

Rewind OBJ Cassette #1 and mount the 8K System Cassette on unit 0; obtain a directory listing of the cassette on unit 1 (which contains all the newly reconfigured files) by typing:

```
.DI 1: )
```

When the directory has finished listing, remove the cassette from unit 1, write-protect it, and attach the directory listing to it. The second blank cassette should next be mounted on unit 1 and zeroed. Then run PIP and type the command line as shown below:

```
.Z 1:)

.R PIP )

*1:CTLOAD.SYS=CTLOAD.SYS )
```

The command line may be entered as soon as the asterisk appears. Control remains in PIP, so when this transfer is complete, mount OBJ

Cassette #1 (containing CSYSLD.LDA, etc.) on unit 0 and enter the next command:

```
*1:CAPS11.Snn=CSYSLD.LDA,CAPS11.LDA/P )
```

The values for nn are taken from the following list; this causes the file to be labeled so as to correspond with the memory size of the system:

Table E-6
System Cassette Labeling Responses

Memory Size	nn
8K	8K
12K	12
16K	16
20K	20
etc...	etc...

When the prompt message occurs, the user should rewind OBJ Cassette #1 and dismount it from unit 0; mount the cassette containing the reconfigured files (i.e., the new version of CAPS11) and type any keyboard character to continue execution.

When PIP returns an asterisk indicating that the transfer is finished, the user can copy PIP and the Editor from the 8K System Cassette using the command:

```
*1:PIP.SRU,EDIT.SLG=PIP.SRU,EDIT.SLG )
```

The remaining files comprising the CAPS-11 system should next be copied to the new System Cassette using any order desired. The recommended order of files is:

```
CTLOAD  SYS
CAPS11  S8K
PIP     SRU
EDIT    SLG
LINK    SRU
ODT     SLG
PAL     SRU
DEMO    PAL
```

If the user has reconfigured LINK, ODT, or PAL, he should copy these programs from the cassette containing the reconfigured versions. Otherwise, he should copy them from the original 8K System Cassette. The DEMO program on the 8K System Cassette should be the last program copied.

The PIP commands to perform these transfers are as follows:

Mount the cassette containing the proper version of LINK on unit 0, write-locked, and type:

```
*1:LINK.SRU=LINK.SRU )
```

Mount the cassette containing the proper version of ODT on unit 0, write-locked, and type:

```
*1:ODT.SLG=ODT.SLG)
```

Mount the cassette containing the proper version of PAL on unit 0, write-locked, and type:

```
*1:PAL.SRU=PAL.SRU)
```

Lastly, mount the 8K System Cassette on unit 0, write-locked, and type:

```
*1:DEMO.PAL=DEMO.PAL)
```

A directory listing of the new System Cassette should be obtained when all transfers are complete and compared to the directory listing above to ensure that all files are present. Several copies of this cassette should next be made (using the PIP copy function).

The user is now ready to try the demonstration program in Appendix D.

APPENDIX F

CASSETTE STANDARDS

F.1 INTRODUCTION

Following is a description of the format and labeling conventions for files and records written on Digital Equipment Corporation TU60 cassettes and specifically for those written under the CAPS-11 system. This standard must be followed when reading and writing cassettes intended for interchange between systems; it is recommended for other cassettes.

The standard describes provisions for file header records which contain information on filename, creation date, record length and data format. There is room in the standard header record for twelve bytes of additional information which can vary from system to system. There is also provision for an extra header record if twelve bytes are not sufficient for additional file information.

The subset of the standard (described in Section F.4) details the minimum requirements that any cassette system should support. This restricted standard includes header record labels, fixed-length, 128-byte records, and date. No support is required for variable-length records, multi-volume files, or expanded information in a second header record.

F.2 DEFINITIONS

A cassette consists of a sequence of one or more files, separated from each other by a single file gap. The first file on the cassette must be preceded by a file gap; the last file must be followed by a file gap and a sentinel file (refer to paragraph F.3.3), or by clear trailer.

Each file consists of a sequence of a header record plus zero or more data records separated from each other by record gaps. The first record of a file is called the file header record, or file label.

A record consists of a sequence of from one up to 2^{16-1} cassette bytes followed by a two-byte cyclic redundancy check. (This is a logical limit; there is no physical limit, except for the length of the tape.)

A cassette byte is eight bits. A bit is a binary zero (0) or one (1).

A character is a byte interpreted via the ASCII character codes. Parity is not required and CAPS-11 ignores the high-order bit of ASCII data.

F.3 THE FULL STANDARD

F.3.1 Applicability

This standard is intended to allow full utilization of the capabilities of cassettes.

F.3.2 The Header Record

THE FILE NAME

Each file must begin with a 32 (decimal) byte file header record. Figure F-1 illustrates the format of the header record. The name and the date are in seven-bit ASCII.

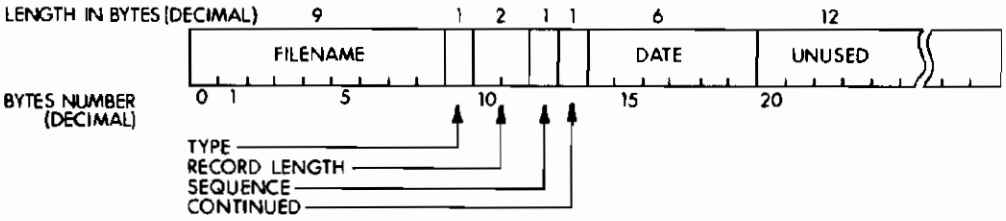


Figure F-1 File Header Record Format

The first nine bytes of a header record contain the file's name. File names are divided into a six-character "name" and a three character "extension". File names and extensions may consist of letters, numerals and blanks. The first character may not be blank; there can be no imbedded blanks within name or extension; name or extension may be padded on the right with blanks.

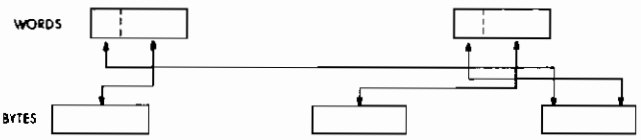
NOTE

When a file is deleted, the current systems change the name to begin with an asterisk (*), in addition to setting the type bit (described next) to 14.

THE FILE TYPE

Byte nine in the header record contains the "File Type". The File Type defines the mode in which data was recorded in that file. Table F-1 lists the file type codes and gives the meanings associated with them (CAPS-11 uses file type codes 1, 2, and 14).

Table F-1
Standard File Types

Type	Description
1	ASCII (seven bits per character--high-order bit undefined)
2	Paper Tape Image (non-ASCII): one frame per byte (operating system dependent)
3	Core Image Format #1 One 36-bit computer word in five bytes (wastes low-order four bits of the fifth byte)
4	Core Image Format #2 One 12-bit computer word in two bytes (only the low-order six bits of each byte is used)
5	Core Image Format #3 One 18-bit word in three bytes (wastes low-order six bits of the last byte)
6	Core Image Format #4 One 36-bit computer word in six bytes (only the low-order six bits of each byte is used)
7	Core Image Format #5 One 16-bit computer word in two bytes
10	Core Image Format #6 (OS/8 character packing) Three bytes for two 12-bit words, as shown below: 
11	Core Image Format #7 Two 36-bit words in nine bytes.
12	Core Image Format #8 Four 18-bit words in nine bytes.
13	Bootstrap File
14	Bad File

FILE RECORD LENGTH

Bytes 10 and 11 of the File Header Record contain the file record length (the file record length is fixed at 128 bytes per record).

NOTE

Byte 10 contains the high-order bits. Thus, record length = 256*contents of byte 10 plus contents of byte 11.

FILE SEQUENCE NUMBER

Byte 12 contains the sequence number for multi-volume files. It is normally zero, otherwise. It is used for information that is split up among files of the same name. Successive continuation files on different cassettes should be numbered 1, 2, 3, ... etc. in this field. (CAPS-11 does not support multivolume files.)

HEADER CONTINUATION BYTE

Byte 13, when non-zero, specifies the number of bytes in an auxiliary header record, which immediately follows this record. If it is zero, data begins immediately with the next record. The format of auxiliary header records is not specified at this time. (CAPS-11 does not use auxiliary header records.)

FILE CREATION DATE

The file creation date is contained in the six bytes starting at byte 14. When specified, this date shall consist of six seven-bit ASCII digits specifying the day number (01-31), the month number (01-12), and the last two digits of the year number, in the order ddmmyy. If not used, the first byte should be zero (null), or blank (ASCII=40).

UNUSED BYTES

The twelve bytes starting at byte 20 are not currently specified.

F.3.3 Logical End of Tape

Logical end of tape is signified by clear trailer or a sentinel file. The sentinel file consists of a single header record whose file name begins with a zero (null).

F.4 THE RESTRICTED STANDARD

F.4.1 Applicability

CAPS-11 supports a subset of the cassette standard described previously. Features supported and not supported are listed below.

F.4.2 Restrictions

RECORD LENGTH

Records shall be 128 bytes long.

NO CONTINUATION HEADER RECORD

The second record in a file must be a data record.

NO SUPPORT FOR MULTI-VOLUME FILES

No support for multi-volume files is required.

F.4.3 Inclusions

The restricted standard (as implemented under CAPS-11) requires support for the following items described in the full standard:

- The File Name
- Logical End of Tape
- Read the (first) header record
- File Creation Date (may be blanks)

F.5 SUPPORT FOR MULTI-VOLUME FILES

The following information should act as a guideline to users who wish to implement multi-volume cassette support in their system. The easiest way to support multi-volume files is the "fall off the tape" method. Whenever the end of a tape is reached before a file has been closed, the system should type out a message to that effect and allow the user to mount another tape, if necessary.

- On READ, the system should:

  1. Type out the message;

  2. If the user indicates that the end of file has been reached, the system should react as such;

  3. If the user indicates that end of file has not been reached, the system should allow the user to mount another tape, indicate the controller, and tell the system to continue processing;

  4. The system should verify that there is a file on the tape with the same name and the next higher volume number as the previous file;

  5. If that is the case, the system should continue processing the file.

- On WRITE, the system should:

  1. Erase any partially-written record by backspacing two and forward-spacing one, and writing an EOF to the end of the tape;*

  2. Type out the message;

  3. Allow the user to mount a new tape;

  4. The system may either assume a blank tape, or space to logical end of tape; then write a file gap followed by a header record(s) with the proper name and volume number;

  5. Continue processing.

The other method involves using the sentinel file, as outlined in the standard. The procedure is as follows:

- On READ, the system should:

  1. Examine the next header record whenever it encounters a file gap; when there is no sentinel file at end of tape, assume end of file;

  2. If the header represents a sentinel file and the sequence byte is one greater than that of the file just being read, the system should request the user to mount another tape; if not, the system should report end of file;

  3. If a sentinel file indicates more volumes exist, the system should allow the user to (1) mount another cassette, (2) indicate where it is, and (3) tell the system to continue processing;

---

*When using 128-byte records, the hardware will never mistake the inter-record gap plus the erased tape for a file gap. This is possible when using larger records. Systems using such records should consider the second method for supporting multi-volume files.

4. The system should verify that there is a file on the tape with the same name and the next higher volume number as the previous file;

5. If that is the case, the system should continue processing.

- On WRITE, when the system reaches the end of tape, it should:

1. Erase enough records to allow a file gap and a sentinel file to be written (this involves double buffering in the case of large records, and triple or even quadruple buffering, in the case of small records);

2. Write out a file gap, and request the operator to mount another cassette;

3. If the operator indicates there will be no extra cassette, the system should (1) write out a sentinel file with a null sequence byte, and (2) tell the operator he can dismount the cassette (clearly, the operator loses some data if he does not mount another cassette);

4. If the operator indicates he wishes to mount another cassette, the system should (1) write out a sentinel file with sequence byte equal to the current files sequence number plus one, and (2) tell the operator he can dismount the cassette;

5. The system should allow the operator to (1) mount another tape, (2) indicate the controller and drive number that holds the tape, and (3) tell the system to continue processing;

6. The system should space to logical end of tape, then write a file gap followed by a header record(s) with the proper name and volume number, followed by the records erased from the previous cassette; and

7. Continue processing.

When processing cassettes that may have been written on the other systems, it may be wise for systems that support the full method for multi-volume files to support the "fall off the tape" method, too.

APPENDIX G

CAPS-11 ASSEMBLY INSTRUCTIONS

G.1 GENERAL INSTRUCTIONS

Listed below are assembly instructions for the CAPS-11 Monitor and system programs. Due to symbol table size, note that some of the system components cannot be assembled under the standard 8K CAPS-11 assembler, but require at least the 12K version of PAL.

The following general instructions apply to all assemblies in this appendix:

1. Mount the System Cassette write-locked on unit 0 and bootstrap the CAPS-11 Monitor.

2. When the Monitor is loaded and responds with a dot, type:

   .R PAL

3. Mount the proper source cassette (obtained from the Software Distribution Center) on unit 1, write-locked (this will be the cassette containing the first file in the input field of the command line).

4. When the Command String Interpreter types an asterisk, enter the appropriate command string followed by a carriage return.

5. When the prompt message (0?) is typed during the second pass of the assembly, dismount and the System Cassette from unit 0 and mount an output cassette (on which the binary OBJ files will be stored) write-enabled on the unit; type any character to continue execution.

NOTE

If the user's CAPS-11 system is 12K or more, the /P (prompt option) is necessary only on the first assembly. Since the system does not need to be rebooted between assemblies, the user may mount one cassette on unit 0 and output as many OBJ files as will fit before mounting a new cassette.

6. When the assembly is complete, PAL will type the message: 000000 ERRORS (with the exception of the Editor, in which there are several line buffer overflow errors; extra characters on a line greater than 72 characters in length are ignored and are indicated on the listing by an 'L message.) In an 8K system PAL will next respond by typing ↑C?; the user should dismount the cassette on unit Ø, remount the System Cassette,and type any character on the console terminal to reboot the system. After this is done, return to step 2 above.

NOTE

If the system is 12K or larger, control will return to the CSI, which prints an asterisk. No reboot is necessary, and the user may proceed with the next assembly (step 4 above).

7. Whenever a prompt message for unit 1 occurs, mount the source cassette containing the proper file (in parentheses) on unit 1 write-locked, and type any keyboard character.

8. In all command lines, TT: may be specified in place of LP:; however, several output listings will be extremely long and the use of the console terminal as the listing output device is not recommended.

G.2 ASSEMBLY COMMAND LINES

G.2.1 Keyboard Listener (KBL)

```
*KBL/P,LP:=1:KBL.035

PASS 2

Ø?          see Step 5
```

G.2.2 CABLDR

```
*CABLDR/P,LP:=1:CABLDR.022

PASS 2

Ø?          see Step 5
```

G.2.3 Command String Interpreter (CSI)

```
*CSI/P,LP:=1:CSI.014
PASS 2
0?          see Step 5
```

G.2.4 CLOD11

```
*CLOD11/P,LP:=1:CLOD11.024
PASS 2
0?          see Step 5
```

G.2.5 RESMON*

```
*RESMON/P,LP:=1:RESMON.068
PASS 2
0?          see Step 5
```

G.2.6 CBOOT

```
*CBOOT/P,LP:=1:CBOOT.007
PASS 2
0?          see Step 5
```

G.2.7 PIP*

```
*PIP/P,LP:=1:PIP.022
PASS 2
0?          see Step 5
```

---

*Requires minimum 12K PAL assembler

G.2.8 CSINBF

```
*CSINBF/P,LP:=1:NOBUFF.PAR/P,CSITAC.032/P
1?          see Step 7 (NOBUFF.PAR)
1?          see Step 7 (CSITAC.032)
PASS 2
0?          see Step 5
1?          see Step 7 (NOBUFF.PAR)
1?          see Step 7 (CSITAC.032)
```

G.2.9 EDIT*

```
*EDIT/P,LP:=1:CAPS11.PAR/P,EDIT.023/P
1?          see Step 7 (CAPS11.PAR)
1?          see Step 7 (EDIT.023)
PASS 2
0?          see Step 5
1?          see Step 7 (CAPS11.PAR)
1?          see Step 7 (EDIT.023)
```

G.2.10 LINK*

```
*LINK/P,LP:=1:LINK.030
PASS 2
0?          see Step 5
```

C.2.11 CSITAC

```
*CSITAC/P,LP:=1:CSITAC.032
PASS 2
0?          see Step 5
```

*Requires minimum 12K PAL assembler

G.2.12 ODT

```
*ODT/P,LP:=1:ODT.015
PASS 2
0?          see Step 5
```

G.2.13 PAL*

```
*PAL/P,LP:=1:PAL1.027/P,PAL2.027/P,PAL3.027/P
1?          see Step 7 (PAL1.027)
1?          see Step 7 (PAL2.027)
1?          see Step 7 (PAL3.027)
PASS 2
0?          see Step 5
1?          see Step 7 (PAL1.027)
1?          see Step 7 (PAL2.027)
1?          see Step 7 (PAL3.027)
```

G.2.14 P8SYM (8K PAL Symbol Table)

```
*P8SYM/P,LP:=1:P8SYM
PASS 2
0?          see Step 5
```

G.2.15 P12SYM (12K PAL Symbol Table)

```
*P12SYM/P,LP:=1:P12SYM
PASS 2
0?          see Step 5
```

*Requires minimum 12K PAL assembler

G.2.16 P16SYM (16K PAL Symbol Table)

```
*P16SYM/P,LP:=1:P16SYM
PASS 2
0?          see Step 5
```

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name _____ Date _____

Organization _____

Street _____

City _____ State _____ Zip Code or Country _____

If you require a written reply, please check here. ☐

Please cut along this line.

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Digital Equipment Corporation
Software Communications
P.O. Box F
Maynard, Mass. 01754

PRINTED IN USA