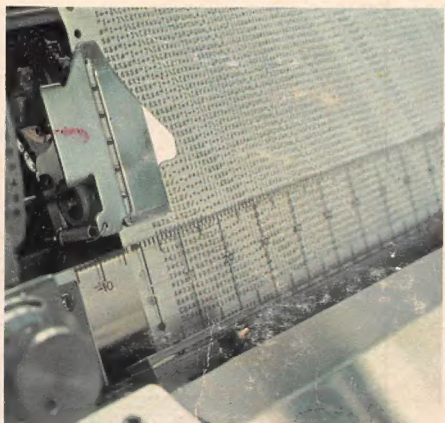
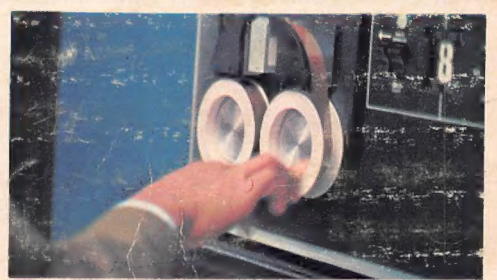


pdp10 reference handbook



Make contact with your computer facility by whatever means the facility has established (e.g., acoustic coupler, telephone, or data phone).

Turn the little plastic knob on the right-hand side of the Teletype to ON.

Type \uparrow C on the Teletype (i.e., hold down the CTRL key while striking C). This establishes communication with the Time-Sharing Monitor. The Monitor signifies its readiness to accept commands by responding with a period (.).

Type LOGIN, or LOG, followed by a carriage return. The system will respond with an informative message like the following:

```
JOB n  NAME OF SYSTEM  
#
```

JOB n is the job number the system has just assigned to you. NAME OF SYSTEM is usually the Monitor name and version number.

Type your project-programmer numbers after the number sign, followed by a carriage return.

The time-sharing system will then type
PASSWORD:

Type your secret password followed by a carriage return. The system will keep the password secret by not printing it on the paper.

If the project-programmer numbers and the password match the project-programmer numbers and password stored in the system accounting file, the system responds with the time, date, TTY number, \uparrow C, and a period.

Example:

```
1301 8-Aug-69 TTY23
```

```
 $\uparrow$  C  
.
```

Now the time-sharing system is ready to accept any commands you wish to type in. You may direct it to load and start a program from the System Library (.R prog), start a program already loaded in core (.START), or perform any of a variety of other operations. (See inside of back cover for a summary of Time-Sharing Monitor commands.)

PDP-10 REFERENCE HANDBOOK

**Prepared by
The Software Writing Group
Programming Department
Digital Equipment Corporation**

Additional copies of this handbook may be ordered from the Program Library, DEC, Maynard, Mass. 01754. Order code: AIW. \$5.00 each. Discounts available on five or more copies.

PDP-10 HANDBOOK SERIES

The material in this handbook, including but not limited to instruction times and operating speeds, is for information purposes and is subject to change without notice.

Copyright © 1969 by
Digital Equipment Corporation

PDP-10 System Reference Manual, Copyright ©, 1968, by Digital Equipment Corporation. *MACRO-10 Assembler*, Copyright ©, 1967, 1968, 1969, by Digital Equipment Corporation. *Time-Sharing Monitors*, Copyright ©, 1968, 1969, by Digital Equipment Corporation. *DDT-10*, Copyright ©, 1968, 1969, by Digital Equipment Corporation. *PIP, Peripheral Interchange Program*, Copyright ©, 1968, 1969, by Digital Equipment Corporation.

The following are trademarks of Digital Equipment Corporation, Maynard, Massachusetts:

DEC
FLIP CHIP
DIGITAL

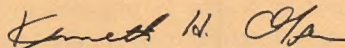
PDP
FOCAL
COMPUTER LAB

FOREWORD

One of the significant measures of the quality of a computing system like the PDP-10 is the utility and availability of systems documentation. Good manuals are the vital communications link between DEC and the people who use our systems.

This collection has been organized for the convenience of PDP-10 programmers, systems analysts, engineers and others who work at the machine language level.

I'm pleased to acknowledge here the work of the many DEC system designers, engineers, and system programmers who continue to advance the state of the time-sharing art in both hardware and software. Also, to our PDP-10 users, who during the past two years, have helped immeasurably to improve and refine the system, and to the DEC software writers and technical artists who prepared this volume, our special thanks.



President, Digital Equipment Corporation

PREFACE

This volume is a comprehensive library of information for experienced programmers, systems analysts, and engineers who are interested in writing and operating assembly-language programs in the PDP-10 time-sharing environment.

The first three chapters deal with program preparation. Chapters 4 and 5 are about loading, editing, testing, debugging, and running source language programs. Chapter 6 contains a miscellaneous collection of utility programs that have proven most useful to system designers and experienced programmers.

As we expect to improve this book in future revisions, all readers are earnestly requested to send corrections and comments to:

Manager, Software Writing Group
Programming Department
Digital Equipment Corporation
Maynard, Mass. 01754

A companion volume for beginning programmers and others who prefer to write programs in one of the popular compiler or conversational languages is scheduled for publication in 1970.

CONTENTS

Foreword	III
Preface	IV
System Overview	VI
Introduction	XI
Book 1 Programming with the PDP-10 Instruction Set	1
Description of the Central Processor Structure, General Word Format, Memory Characteristics, and Assembler Source-Programming Conventions, Followed by a Presentation of the Specific Instruction Format, Mnemonic and Octal Op Codes, Functions and Timing Formulas.	
Book 2 Assembling the Source Program	187
Reference Information for the PDP-10 Assembly System, containing Explanation of Format of Statements, Use of Pseudo-Operations and the Coding of Macro Instructions.	
Book 3 Communicating with the Monitor	283
Complete Guide to the Use of the Time-Sharing Monitors: Monitor Commands, Allocation of Facilities, Relocation and Protection of User Programs, and Description of the Reentrant Capability.	
Book 4 Editing the Source Program	491
Procedures for Creating, Modifying and Displaying Source Files Recorded in ASCII Characters.	
Book 5 Executing the Program On-Line	525
Description of Loading and Linking of Relocatable Binary Programs Generated by MACRO-10 or FORTRAN IV and Exposition of Commands and Techniques for On-Line Checkout and Testing of MACRO-10 and FORTRAN IV Programs.	
Book 6 Utility Programs	583
Transferring Data Files Between Standard Devices, Updating Files Containing Relocatable Binary Programs, Manipulating Programs within Program Files, Cross-Referencing User Defined Operators, Op Codes, Pseudo-Op Codes and Global Symbols, Comparing Source Files and Comparing Binary Files, Serving and Restoring Core Images on DECTape.	
Appendices	629
Master Index/Glossary	637

SYSTEM OVERVIEW

The PDP-10 is the successful culmination of many years of computer design research — a process which has enabled Digital Equipment Corporation to provide better computers at the lowest possible prices.

Starting with the PDP-1 in 1959, DIGITAL has pioneered the development of real-time systems for science and industry. Since then, each new system has increased in versatility, yet has consistently decreased the cost of computation. The PDP-1 was the first powerful real-time computer for under \$150,000. The PDP-8 showed that an effective computer could sell for less than \$20,000, and newer models in the PDP-8 family have lowered the cost to less than \$10,000.

In developing its time-sharing capability, DIGITAL has built a history of success very similar to the company's record in real-time applications. DIGITAL's customers have been building time-sharing systems around PDP computers since 1960. And, in 1963, DIGITAL developed its own time-sharing computer, PDP-6 — the first to be delivered with manufacturer-supplied hardware and software.

The PDP-10 reflects DIGITAL experience in both real-time and time-sharing. The system performs time-sharing and real-time operations equally well and simultaneously and provides concurrent batch processing.

In conversational time-sharing, up to 63 users at local and remote locations can simultaneously develop programs on remote consoles and receive answers to mathematical or engineering problems in seconds. PDP-10 time-sharing monitors provide instantaneous response for the users so that they can perform on-line composition, editing, and debugging of programs in FORTRAN IV, MACRO-10, COBOL, BASIC, and AID, with the use of EDITOR, TECO and DDT. The monitors can handle any mixture of these languages and programs concurrently. And most of the software is re-entrant so that multiple users can share the same compiler or utility program for increased efficiency.

For programs that don't require immediate processing, users may initiate batch processing — a task which proceeds concurrently with time-sharing and real-time

tasks. In batch processing, the PDP-10 can handle any stream of programs, such as a mixture of FORTRAN, MACRO-10 and COBOL. Normally, batch processing operates without operator attention. However, the PDP-10 allows the operator to stop or start the batch system, rearrange the queue or call for a print-out to analyze program errors. The operator can also select and assign the desired input, output, and temporary storage devices to be used in batch processing.

When real-time operations such as data acquisition and control are the primary purpose of the PDP-10, system software provides response in microseconds, processing information at speeds that meet the most demanding requirements. High priority real-time tasks are interfaced directly to the priority interrupt system and control their own input/output operations for unlimited flexibility. Less critical real-time jobs are monitor controlled with a real-time clock assuring that each task does not exceed its allotted time and destroy the response of other programs. For even greater efficiency, time not used by the real-time programs can be used for conversational time-sharing and/or batch processing.

Structure of the PDP-10

Every PDP-10 uses one of three levels of monitors to allocate resources and perform input/output functions. The single-user monitor is used for dedicated systems which operate one program at a time. The multi-programming monitor controls the execution of multiple programs in core, switching between them at microsecond speeds. The swapping monitor effectively increases the available core by swapping programs between high speed disk or drum storage and core memory. Thus more users can be served by a given amount of core. All language processors (FORTRAN, MACRO-10, COBOL, BASIC, and AID) operate identically under the multi-programming and swapping monitors.

To make efficient use of memory, language processors and important utility programs are re-entrant, that is, the pure code for each program can be shared by multiple users. Re-entrancy is possible since any program

may be separated into a pure segment that never requires modification, and a segment which contains code or data which is relevant only to a particular user. For example, a re-entrant system can service three FORTRAN users with one 8 K compiler (pure code) and three 2 K user areas, a total of 14 K, whereas a non-re-entrant system would require 30 K for the same programs. Since more users can occupy a given amount of core space, system response improves and swapping time is reduced. Dual protection and relocation registers protect the active user and allow the program segments to reside in two non-contiguous sections of core memory.

The PDP-10 has a 36-bit word length allowing it to store 25 to 30 percent more information than a 32-bit system. The system stores five 7-bit USACII characters whereas the smaller word length computer stores only four characters. It also provides more accuracy in single precision floating arithmetic than computers with 32-bit word size.

The PDP-10 has a large instruction repertoire to simplify assembly coding and reduce the size of higher level programs. Its 366 instructions divide logically into families and are easily learned. The list also includes an extensive set of floating point and byte manipulation instructions. Due to instruction set efficiency, fewer instructions are required to perform a given function. Assembly language programs are therefore shorter than with other computers and the instruction set simplifies monitor systems, language processors, and utility programs. For example, compiled programs are 30 to 50 percent shorter, require less memory, and execute faster than those of comparable computers.

Sixteen high speed integrated circuit registers also help improve program execution. Depending on program requirements, these registers can serve as accumulators, normal memory location, and/or index registers. Intermediate results of computations are stored in the registers rather than in core memory; thus, no instructions are needed to store and retrieve the data and data is available in nanoseconds. Fifteen of the registers can be used as fast memory locations so that program segments with sixteen or fewer instructions can be executed repetitively at very high rates.

The PDP-10 memory bus structure gives the central processor and high speed data channels simultaneous access to separate memory modules. Only when the processor and a data channel access the same module does the processor lose a memory cycle. Modules contain up to four ports, allowing access to a total of four processors and/or channels. Such parallel operation improves processor utilization, yielding manyfold improvements over systems which provide only a single path to memory. The bus system allows each data channel to transmit full 36-bit words at speeds of up to one million words (5 million 7-bit characters) per second.

Memory can be modularly expanded to 262,144 words of core, all of which (including the 16 accumulators and 15 index registers) can be directly addressed. Total memory capacity can be comprised of combinations of modules in 8,192-, 16,384-, 32,768-, 65,356-, and 131,072-word blocks. Memory banks are asynchronous allowing interleaving and making it possible to intermix memories of different cycle times.

To provide immediate service to real-time requests, the PDP-10 has a multi-level priority interrupt system. The system is a hardware feature, but is programmable for increased flexibility. Devices may be assigned to any level under program control and the entire interrupt system or any level may be selectively turned on or off.

PDP-10 Configurations

The modularity of PDP-10 hardware and software makes it economical to configure a wide variety of systems and easy to expand the systems in the field. (See PDP-10 hardware list in Appendix A.) An individual can buy a single user system and, at some later date, expand to a small time-sharing system. Or the small time-sharing user can expand his system to serve 63 users simultaneously. Within any basic configuration, the user has a wide choice of memory sizes and speeds, input/output equipment, and storage facilities. For example, the input/output system alone can accommodate up to 128 discrete devices and device controllers, permitting almost limitless expansion of on-line storage and other input/output equipment.

The single-user system in Figure 1 can be simple or as elaborate as the user requires. As shown, it consists of an arithmetic processor, one or more core memory modules, a DEctape control and DEctape units¹, a console teletype, and a paper tape reader and punch. By adding more core memory and data line scanner, the system can easily be converted to multi-programming.

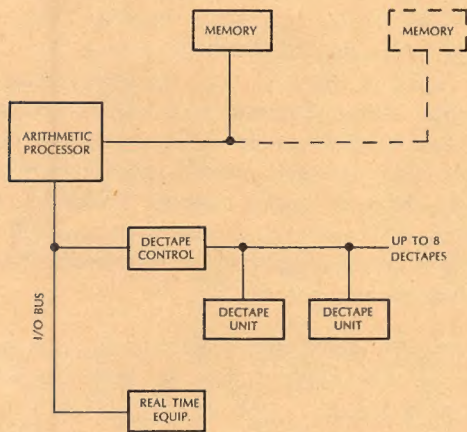


FIGURE 1. SINGLE-USER SYSTEM

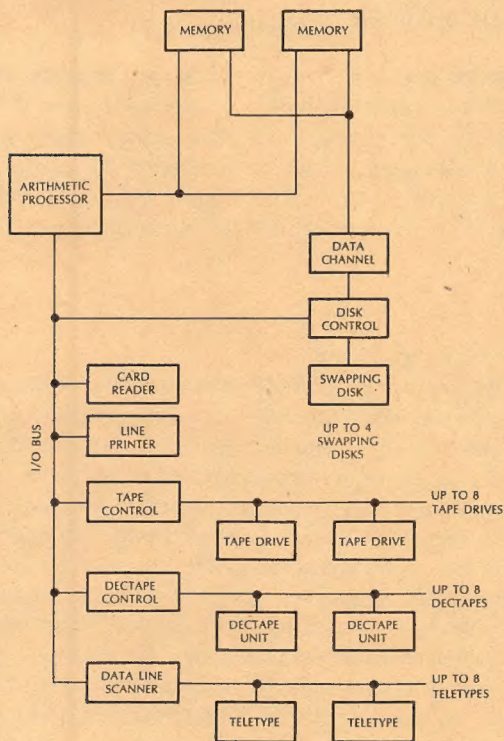


FIGURE 2. 8-USER SWAPPING SYSTEM

¹A special random access magnetic tape designed by Digital Equipment Corporation.

The small swapping system shown in Figure 2 can be expanded in eight user groups to the large time-sharing system (Figure 3) which can handle up to 63 users. The large system includes file storage and swapping storage units, additional memory, and more peripheral equipment. For very large systems, a file storage disk may replace or supplement the disk packs. A computer-based communication system may be substituted for the data line scanner, and synchronous data phone units can be used to connect the system to remote batch devices and other computers.

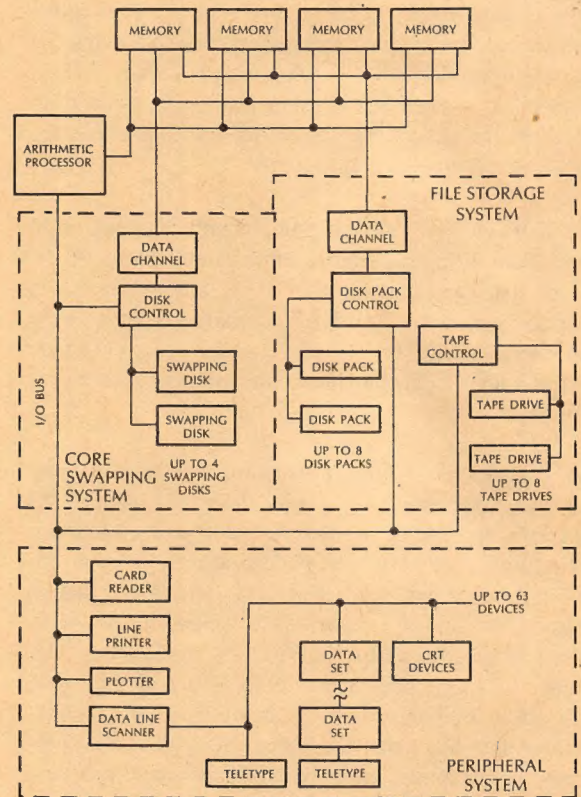


FIGURE 3. LARGE SWAPPING SYSTEM

The dual processor system in Figure 4 is one of many possible multi-processor configurations that the user can tailor his monitor to meet. Since the system shares both peripherals and core memory, both processors can access memory at the same time and can compute in parallel. Such an arrangement doubles the computing power of a single processor and more than doubles cost/effectiveness, since the cost of the additional processor is only a small fraction of overall system cost.

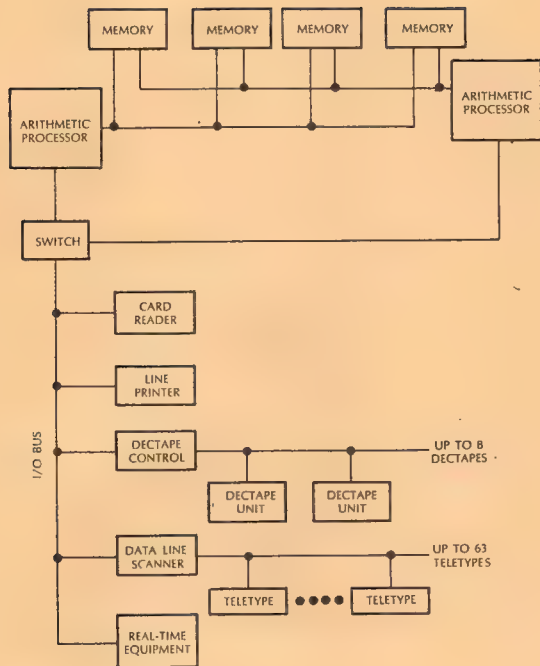


FIGURE 4. DUAL PROCESSOR SYSTEM

In other multi-processor systems, the processors may work independently or communicate through shared memory. One may serve as the input/output processor while the other performs most of the calculations. Or the processors can share all input/output and processing. Multi-processor systems can also combine a PDP-10 arithmetic processor with other DIGITAL computers.

In Summary

The PDP-10 exemplifies the versatility required for today's large computing tasks. With its 366-instruction repertoire, re-entrant software, multi-programming hardware, and flexible priority interrupt system, it provides power and excellent response for a multitude of applications. And with the system's wide range of hardware and software, the user can purchase to serve present needs, yet easily expand to meet future system requirements.

Every PDP-10 is backed by service — software support for a full range of customer assistance, service through a worldwide system of over 60 service centers, and formal training through a variety of available training courses.

At a cost of less than half that of comparable systems, the PDP-10 provides the best price/performance available today — another step toward Digital Equipment Corporation's goal of providing the most for every computing dollar.

SYSTEM DESCRIPTION

DEC PDP-10 software is divided into eight functional groupings with respect to common programming activities, as follows:

- a. Source Program Preparation
- b. Conversational Language Translators
- c. Program Loading and Library Facilities
- d. Debugging
- e. Utilities
- f. Calculators
- g. Batch Processing
- h. Monitoring

Groups a through g are programs called CUSPs (commonly used system programs) and are run under control of the Single-User, Multiprogramming non-disk, Multiprogramming disk or Swapping Monitor.

Source Program Preparation (EDITOR, LINED, TECO)

The DECTape Editor, LINED (Line Editor for Disk), and the Text Editor and Corrector (TECO) programs can be used to create (and later correct or modify) text files (e.g., Macro-10 and FORTRAN source language programs) for subsequent assembly or compilation. Editor creates and modifies files on DECTape; LINED creates and modifies files on disk; and TECO performs more complex editing functions on any standard I/O devices.

Language Translators (MACRO, F40)

The Macro-10 Assembler (MACRO) and the FORTRAN Compiler (F40) translate source programs written in the Macro-10 and FORTRAN IV languages, respectively, into binary machine language for subsequent loading and execution.

Program Loading and Library Facilities (LOADER, LIB40, JOB DAT, FUDGE2)

Loading is performed by the Linking Loader, which loads specified relocatable binary programs in core, links their references to each other, and searches the appropriate subroutine libraries (e.g., LIB40) for required subroutines. A job data area (JOB DAT) is created by the Loader for each program; this area is used to store the current status of the job during execution. Library files of binary programs can be updated by use of the File Update Generator (FUDGE2).

Debugging (DDT, CREF, GLOB)

After a program is compiled (or assembled), it can be loaded in conjunction with the Dynamic Debugging Technique (DDT) program and debugged. DDT allows the user to control program execution and to modify the program in any of several modes, including symbolic. For purposes of further program analysis (and for documentation), the user can use the Cross Reference Listing (CREF) program, which produces a cross-referenced listing of all symbols within his Macro program, and the Global Cross-Reference Listing (GLOB) program, which produces one to three listings of all global symbols encountered in one or more programs.

Utilities (PIP, CODE, SRCCOM, BINCOM)

A variety of utility programs are available for general-purpose data handling. Among these programs are: the Peripheral Interchange Program (PIP), which transfers data between any standard I/O devices; Code Translator (CODE), which performs translations between standard ASCII codes and code of other manufacturers; Source Compare (SRCCOM), which compares two versions of an ASCII file; and Binary Compare (BINCOM), which compares two versions of a binary file.

Conversational Languages (AID, BASIC)

Two problem-solving conversational languages for scientists, engineers, and students are included as part of the PDP-10 software: the Algebraic Interpretive Dialogue (AID), a program based on the RAND JOSS™ algebraic language; and Advanced BASIC®, a conversational language for scientific, business, and educational applications that includes among many other features a special set of matrix processing operations.

™ JOSS is the trademark and service mark of the RAND Corporation for its computer program and services using that program.

® Registered, Trustees of Dartmouth College.

Batch Processing (BATCH, STACK)

The Batch Processor (BATCH) monitors the sequential execution of a series of user jobs with a minimum of operator attention; operates as one of the "users" in a time-sharing environment and runs concurrently with the Batch-controlled jobs (as well as other jobs on the system); and permits constant communication by the operator. Job Stacker (STACK) prepares input stacks for BATCH and processes output stacks from BATCH.

Monitors

PDP-10 software includes two major categories of Monitors: the Single-User Monitor (10/30 configuration) and the Time-Sharing Monitors (10/40 and 10/50 configurations). The latter category includes the Multiprogramming non-disk Monitor (10/40), Multiprogramming disk Monitor (10/40) and the Swapping Monitor (10/50). The Swapping Monitor was used in the generation of all examples in this manual.

The 10/30 Monitor is a subset of the 10/40 and 10/50 Monitor. They are compatible at the source and relocatable binary levels. The 10/40 and 10/50 Monitors are compatible at the source, relocatable binary, and saved core image levels.

SYSTEM OPERATION

The following basic procedures and rules are necessary to communicate with the Monitor and load and execute DEC commonly used system programs (CUSPs), as well as user programs.

Step

Procedure

1. To establish communication with the Monitor, place the Teletype in Monitor Command Mode by typing $\uparrow C$ (i.e., hold down the CTRL key while striking C; Monitor will respond with a period (.). If you have a 10/30 or a 10/40 system, skip the LOGIN procedure in the next step.

2. Type LOGIN followed by carriage return. The system will respond with

JOB n NAME OF SYSTEM

followed by a number sign. Then type your project-programmer number, followed by a carriage return. The system will then type

PASSWORD:

Then type your secret password followed by a carriage return. The system will keep it secret by not printing it on the paper. If the project-programmer number agrees with the password, you will be logged, and any messages of the day will be typed for you.

3. Then, direct Monitor to load and start a program from the System Library (.R prog), start a program already loaded in core (.START), discontinue your job (.KJOB), or perform any of a variety of other operations. A complete list of Time-Sharing Monitor commands is given in Table 9-1.

All CUSPs (except EDITOR and LINED) supplied by DEC are device independent; therefore, the user must tell the CUSP, via a command string typein, which devices to use. Readiness to receive a command string is signalled by the CUSP via an asterisk (*) typeout after loading. For example, when the FORTRAN IV Compiler has been called and it has responded with an asterisk, the user types in a command string indicating:

- a. The device containing the source program to be compiled
- b. The device on which the binary output is to be placed and
- c. The device on which the compilation listing is to be written

*binary-output-device, listing-device ← source-device

Devices are specified by a 3-character device name (a fourth character, a digit, specifies the particular unit in the case of DECTapes, teletypes, and magnetic tapes), followed by a colon.

<u>Device</u>	<u>Device Name</u>
Card reader	CDR:
Card punch	CDP:
Line printer	LPT:
Paper tape reader	PTR:
Paper tape punch	PTP:
Teletype	TTY: or TTYn:
DECTape	DTAn:
Magnetic tape	MTAn:
Disk	DSK:

For file-oriented devices (DECTape and disk), a filename (maximum of six characters) is also required following the device name to specify either the specific file to be read or the filename to be assigned to the output. A filename can be further specialized by adding a 3-character extension name to it, preceded by a period (.). Extension names are generally used to classify a file into a particular category, and certain standard extensions are used and recognized throughout the system (e.g., .REL for relocatable binary files, .SAV for saved core image files, .MAC for Macro-10 source files, .F4 for FORTRAN source files, etc.). The following example shows a sample FORTRAN command string.

Example:

DTA1:BIN.REL,LPT: ← DTA0:SOURCE

Compile the file designated as SOURCE on DECTape 0; write the binary output on DECTape 1, designating it BIN.REL; print the listing on the line printer.

TYPOGRAPHIC CONVENTIONS IN THIS MANUAL

All computer typeouts are underscored (single line) or enclosed in brackets (multiple lines).

All operator typeins are not underscored.

SYMBOLGY USED IN CONSOLE EXAMPLES

- ↑C Hold down the CTRL key while striking C. Normally echoes as ↑C.
 ↑x Hold down the CTRL key while striking the "x" key, where "x" is any character.
 Normally echoes as ↑x.

Some special control symbols and their respective key designations for Models 33, 35, and 37 Teletypes are given below.

<u>Key Designation</u>	<u>Symbol in This Manual</u>	<u>Models 33 and 35</u>	<u>Model 37</u>
(TAPE)	↑R	Hold down CTRL key while striking R.	Same as 33/35
(-TAPE) (not-TAPE)	↑T	Hold down CTRL key while striking T.	Same as 33/35
(BELL)	↑G	Hold down CTRL key while striking G.	Same as 33/35
(TAB) (horizontal tab)	→ or ↑I	Hold down CTRL key while striking I.	Strike TAB key
(FORM)	↑L	Hold down CTRL key while striking L.	Same as 33/35
(VT) (vertical tab)	↑K	Hold down CTRL key while striking K.	Same as 33/35
(XON)	↑Q	(Initialize paper tape reader input.) Hold down CTRL key while striking Q.	Same as 33/35
(XOFF)	↑S	(Terminate paper tape reader input.) Hold down CTRL key while striking S.	Same as 33/35
←	←	Hold down the SHIFT key while striking O.	Strike ← key
RETURN	↵	Strike the RETURN key. Normally echoes back as a carriage return, line feed.	Same as 33/35
ALTMODE or PREFIX or ESC	\$	Strike the ESC key (sometimes labeled ALTMODE or PREFIX)	Same as 33/35
[[Hold down the SHIFT key while striking K.	Strike [key

<u>Key Designation</u>	<u>Symbol in This Manual</u>	<u>Models 33 and 35</u>	<u>Model 37</u>
]]]]	Hold down the SHIFT key while striking M.	Strike] key
† †	† †	When appearing alone (as in DDT), hold down the SHIFT key while striking N.	Strike † key
< <	< <	Hold down the SHIFT key while striking " , " .	Strike < key
> >	> >	Hold down the SHIFT key while striking " . " .	Strike > key
LINE FEED	↓	Strike the LINE-FEED key.	Strike LINE SPACE key
RUBOUT	RUBOUT	Strike the RUBOUT key. Normally echoes back as a backslash (\), XXX, or a repeat of the character erased.	Strike DELETE key
\ \	\ \	Hold down the SHIFT key while striking L.	Strike the \ key
SPACE BAR	␣	Strike the space bar to space to indicated position. TAB can also be used in most instances.	Same as 33/35

NOTE

Due to recent changes in ASCII, some terminals may have the keytops " ^ " (caret) and " _ " (underline); these characters have the same codes as " ↑ " and " ← ", respectively. Where possible, DEC will supply all teletypes with the arrow characters.

DEMONSTRATION PROGRAMS

The following demonstration programs illustrate the simplicity and flexibility of a PDP-10 software system:

- a. Demonstration #1 is a typical example of the procedure for creating a FORTRAN main program source file and a Macro-10 subprogram file. These two files are then translated, loaded, and executed together. A bug is encountered during execution, and the DDT (Dynamic Debugging Technique) program is used to correct the erroneous instruction. The programs are now executed successfully, their core image is saved for future execution, and the original source file is altered to reflect the correction made to the binary code.
- b. Demonstration #2 is a more complex example. The sequence of operations is similar to that of Demonstration #1 (source program file creation, translation, loading, execution, debugging, saving the core image, and altering the source code to reflect changes made to the object code). In addition, such procedures as leaving the current job, logging in and beginning a second job, and then later returning to the original job are included. Figure 1-1 contains the flow diagram for Demonstration #2.

Demonstration #1

```
↑C
.LOGIN
JOB 10 4S.51G DEC PDP-10 #40
#10,63
PASSWORD:
1641 01-JUL-69 TTY22
THE DISK WILL BE REFRESHED AT 1300 HOURS DAILY
```

```
↑C
.ASSIGN DTA
DTA2 ASSIGNED
```

```
.MAKE MAINPG
```

```
*I C FORTRAN PROGRAM FOR TYPING TTY PHYSICAL NAME
CALL GETTYN(R)
TYPE 6,R
6 FORMAT(' THE NAME OF YOUR TELETYPE IS: ',A5)
END
$$
```

```
*EX$$
```

```
[EXIT
↑C
```

```
.MAKE SUBRTE.MAC
```

```
*ITITLE GETTYN MACRO SUBROUTINE
SUBTTL SUBROUTINE TO GET TTY NAME AND CONVERT TO ASCII.
INTERNAL GETTYN
AC4=4
AC5=5
AC6=6
GETTYN: 0
CALL AC4,[SIXBIT/GETLIN/] ;GET TTY NAME.
GETBYT: ILDB AC6,NMPTR1 ;GET A SIXBIT CHARACTER.
SKIPN AC6 ;DONE?
JRST NAMDON ;YES.
ADDI AC6,40 ;NO, CONVERT CHAR TO ASCII.
IDPB AC6,NMPTR2 ;SAVE CONVERTED CHARACTER.
JRST GETBYT ;GO GET NEXT CHARACTER.
NAMDON: MOVE AC5,@(16) ;STORE NAME.
JRA 16,1(16) ;RETURN TO MAINPG.
NMPTR1: POINT 6,AC4-1,35
NMPTR2: POINT 7,AC5-1,34
END
$$
```

```
*EX$$
```

```
[EXIT
↑C
```

Log into the system by typing LOGIN, followed by the prescribed "login" information for your particular system. The monitor responds with time, date, and Teletype number.

ASSIGN DTA assigns a DECTape for storage of the completed program.

MAKE MAINPG calls in TECO (Text Editor and Corrector program) to create MAINPG, your FORTRAN IV source program file. The text of the source program is preceded by TECO insert command I. To terminate the text, type two ALTMODEs \$\$\$. TECO command EX\$\$\$ deposits the file in your disk area and returns you to the monitor.EXIT. ↑ C acknowledges the return to the monitor.

MAKE SUBRTE.MAC creates a MACRO-10 source program file SUBRTE.MAC. This subroutine with the program name of GETTYN is called from the above FORTRAN program to obtain the Teletype name in SIX-BIT code and convert it to USASCII code for outputting. The I, \$\$\$, and EX\$\$\$ commands perform the functions previously mentioned.

.EXECUTE SUBRTE/CREF,MAINPG/L

[FORTRAN: MAINPG
MACRO: GETTYN
LOADING

LOADER 5K CORE

THE NAME OF YOUR TELETYPE IS:

[EXIT
↑C

.DIRECT

[DIRECTORY 27,20 0901 29-JAN-69

003EDS.TMP 01 29-JAN-69
MAINPG 01 29-JAN-69
SUBRTE.MAC 01 29-JAN-69
003LOA.TMP 01 29-JAN-69
003MAC.TMP 01 29-JAN-69
003SVC.TMP 01 29-JAN-69
003CRE.TMP 01 29-JAN-69
MAINPG.LST 01 29-JAN-69
MAINPG.REL 01 29-JAN-69
SUBRTE.REL 01 29-JAN-69
SUBRTE.LST 03 29-JAN-69
003PIP.TMP 01 29-JAN-69

TOTAL BLOCKS 14

[EXIT
↑C

.LIST MAINPG.LST

[EXIT
↑C

.CREF

[EXIT
↑C

.DEBUG SUBRTE,MAINPG

[LOADING

LOADER 7K CORE

EXECUTE SUBRTE/CREF,MAINPG/L instructs the system to: 1. Assemble SUBRTE.MAC, generating a cross reference listing file (CREF), and compile MAINPG, creating a normal listing file (L). 2. Load the two resulting relocatable binary files together. 3. Start execution. System acknowledges each step as it is being executed and types out the total core requirement for loading.

The program has a bug. It didn't complete the message: THE NAME OF YOUR TELETYPE IS:

To find the bug, it may be helpful to list the directory of your disk area by using the command DIRECT. Besides the two text files created with TECO, there are many others. The REL files contain the relocatable binary output from the assembly and compilation. The LST files contain the listings. The TMP files are temporary command files created by the COMPIL CUSP. These include 003CRE.TMP, which contains the names of LST files to be output to the line printer when a CREF command is given.

You can now print the listing files for examination. LIST MAINPG.LST outputs the listing file produced when the FORTRAN program was compiled. CREF outputs the CREF listing file produced when the MACRO-10 subroutine was assembled. Examination shows that the MOVE instruction in the MACRO-10 subroutine should be a MOVEM.

To debug your program, load the DDT (Dynamic Debugging Technique) program by typing DEBUG SUBRTE,MAINPG. Note that assembly and compilation are not repeated since the REL files are more recent than the source files.

GETTYNS: 'NAMDON/ MOVE AC5,@0(16) MOVEM AC5,@(16)
\$G

[THE NAME OF YOUR TELETYPE IS: TTY13
EXIT
↑C

.SAVE DTA2:IMAGE
[JOB SAVED
↑C

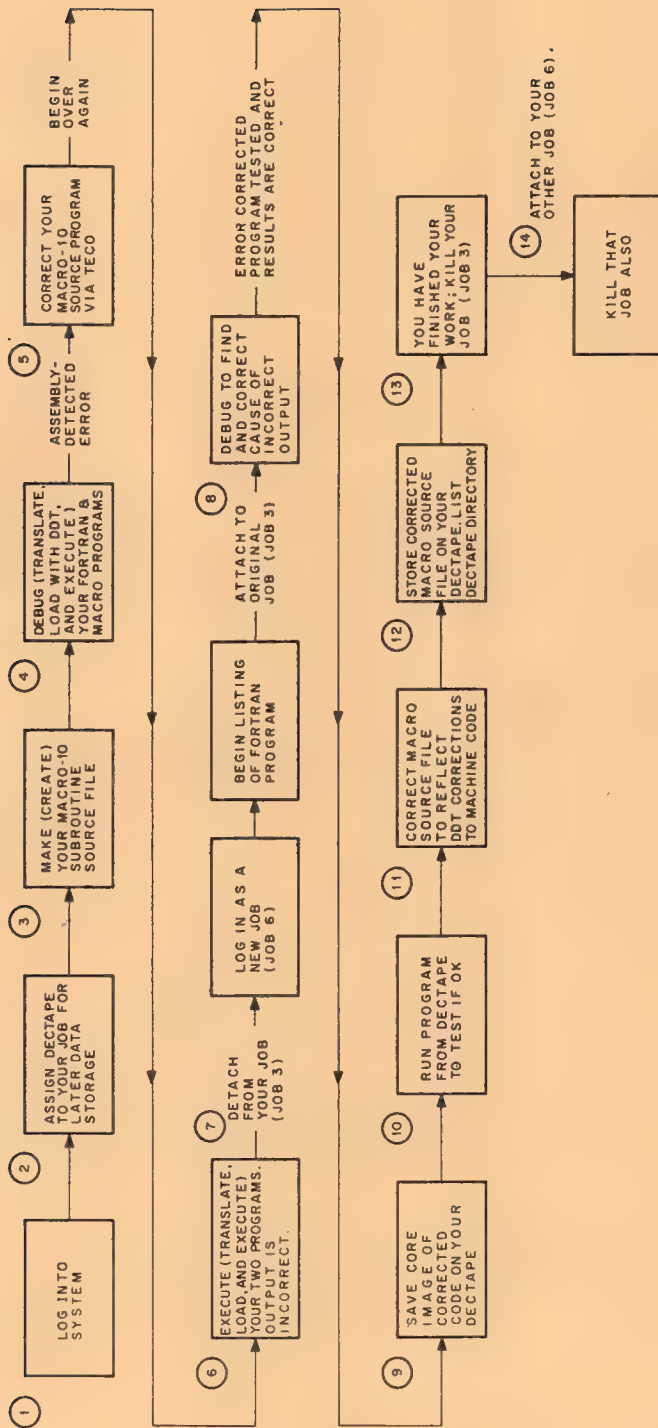
.TECO SUBRTE.MAC

* BJNMOVESIMSØLT\$\$
NAMDON: MOVEM AC5,@(16) ;STORE NAME.
*EX\$\$

[EXIT
↑C

GETTYN\$: accesses the symbol table for the MACRO subroutine, and NAMDON/ accesses the location of the erroneous instruction. Type in the correction MOVEM AC5, @ (16) and use \$G to re-execute the program so that you can check the result. The bug is out! The message is completed: THE NAME OF YOUR TELETYPE IS: TTY13.

The final steps are to store the core image of your program on DECTape and correct the source file. SAVE DTA2:IMAGE stores the program on DECTape 2, giving it the name IMAGE. TECO SUBRTE.MAC calls in TECO to correct the source file. And the TECO command string BJNMOVE\$IM\$OLT\$\$ searches for the word MOVE, inserts an M after it, and types out the corrected line. EX\$\$ deposits the corrected source program in your disk area and returns to the monitor. The user can now log off the system or start some other program.



10-0052

Figure 1-1 Demonstration Program #2 Flow Diagram

Demonstration #2

```
• LOGIN
  JOB 3      4S34
  #27,20
  [0955      27-JAN-69      TTY13
  [THE DISK WILL BE REFRESHED DAILY AT 4:45 PM UNTIL FURTHER NOTICE.

  ↑C
```

```
•ASSIGN DTA DT
  DTA2 ASSIGNED
```

•MAKE RANDOM

```
*ITITLE RANDOM NUMEEBER GENERATING SUBROUTINE
SUBTTL CHARLIE PROGRAMMER      27 JAN 1969
;RANDOM NUMBER GENERATING SUBROUTINE
$$
*I;THE FORTRAN CALLING SEQUENCE IS --
;      CALL RANDOM (ARG)
;WHERE ARG SPECIFIES THE LOCATION AT WHICH THE RESULTING
;SINGLE PRECISION FLOATING POINT RANDOM NUMBER WILL BE
;STORED.  NUMBERS PRODUCED BY THIS ROUTINE ARE PSEUDO:RANDOM
;NUMBERS BUT ARE UNIFORMLY DISTRIBUTED OVER [0,1].
$$
*INTERNAL RANDOM
      ACX=5
      ACY=6
      ACZ=ACY+1 ;ACCUMULATOR SYMBOLIC DEFINITIONS.
$$
*IRANDOM:  0      ;ENTERED BY JSA 16,RANDOM.
           CALL ACX,[SIXBIT/TIMER] ;GET TIME IN CLOCK TICKS.
           ANDI ACX,3 ;USE TIME TO SELECT 1-4 ITERATIONS.
$$
*IRLOOP:   MOVE ACY,RNUMBR ;FETCH PREVIOUS PSEUDO-RANDOM NUMBER.
           MUL ACY,MAGIC   ;MULTIPLICATIVE RANDOM NUMBER GENERATOR.
           MOVEM ACZ,RNUMBR ;SAVE NEXT PSEUDO-RANDOM NUMBER.
           SOJGE ACX,RLOOP ;ITERATE AGAIN?
$$
*I        LSH ACZ,->D8     ;CONVERT TO FLOATING POINT FORMAT.
           TLO ACZ,20000   ;IN THE RANGE [0,1].
           FADRI ACZ,0     ;NORMALIZE.
           MOVEM ACZ,@(16) ;STORE RESULT, AND
           JRA 16,1(16)    ;**RETURN**.
```

Log into the system by typing LOGIN (may be abbreviated as LOG); Monitor responds with the job number assigned to your job and the version number of the Monitor. Following the typeout of the # symbol, type in your project-programmer number. Monitor then waits for a password. Type your password (echo-typeout is suppressed). If your password matches correctly with your project-programmer number, Monitor types out the correct time, date, and the physical name of the teletype you are using. Monitor may type out some informative messages and return you to Monitor level. At this point, any Monitor command can be typed.

Assign a DECtape unit to the job for later storage of files, and assign it the logical name DT. Monitor responds that DECtape unit 2 has been assigned to the job. Mount an available reel on this unit, and place the WRITE switch in the WRITE-ENABLED position. The reel contains the FORTRAN source program for later use. From this point, refer to the DECtape unit as either DTA2: or DT:.

Now, create the source program file for the Macro-10 subroutine to be run in conjunction with the FORTRAN program. TECO (Text Editor and Corrector) can be used to create such a text file. Type MAKE RANDOM to call in the TECO program and cause TECO to open a file for creation; give it the filename RANDOM. After TECO has responded with an asterisk, type an Insert command (I) followed by the first portion of the text of your Macro-10 source program. Note that a typing mistake was made on the first line - NUME; this can be corrected by pressing the RUBOUT key to echo the previous character and then typing the correct character. If a typing error occurs several characters back, press the RUBOUT key repeatedly until you have reached the erroneous character. To avoid overflowing the input command buffer, break the text into several segments, rather than typing it as one continuous block. This is done by typing two successive ALTMODEs (an ALTMODE echoes as \$) after every six or seven lines of text to cause the contents of the input command buffer to be transferred to the TECO output buffer and the input command buffer to be cleared. Following the subsequent asterisk typeout, repeat the Insert command before continuing the text. After typing the program, request a typeout of the entire text by typing HT\$\$\$. Notice that an Insert command was not typed at the beginning of the third segment of the text. Luckily, TECO took the "I" in INTERNAL as the Insert command, but this resulted in NTERNAL. Correct this by typing BJ (set the TECO pointer at the beginning of the text), S (Search) for NTERNAL, 7R (Reverse the pointer seven characters), II (Insert an "I"), and OLT (Type out the corrected Line). Note that each command step is terminated by an ALTMODE (\$). Also, insert a space following PP in the line ;ACM.....PP83-89) and request that the corrected line be typed out. Type EX \$\$ to direct TECO to write out its output buffer onto disk, assign it the filename previously specified in the MAKE command, and close the file.

```

$$
*I;THE MULTIPLIER USED IS 5*15 (SEE COMPUTER REVIEWS, VOL 6, #3,
;REVIEW NUMBER 7725, AND THE REFERENCED PAPER IN JOURNAL OF
;ACM, JANUARY, 1965, PP83-89).
MAGIC:      5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5 ;THE MULTIPLIER.
RNUMBR:    1 ;THE NEXT RANDOM NUMBER IS ALWAYS HERE.
;THE ITERATION STARTS FROM A VALUE OF 1.
PATCH:    BLOCK 10 ;PATCHING SPACE.
          END

```

```

$$
*HT
$$

```

```

TITLE RANDOM NUMBER GENERATING SUBROUTINE
SUBTTL CHARLIE PROGRAMMER      27 JAN 1969
;RANDOM NUMBER GENERATING SUBROUTINE
;THE FORTRAN CALLING SEQUENCE IS --
;      CALL RANDOM (ARG)
;WHERE ARG SPECIFIES THE LOCATION AT WHICH THE RESULTING
;SINGLE PRECISION FLOATING POINT RANDOM NUMBER WILL BE
;STORED. NUMBERS PRODUCED BY THIS ROUTINE ARE PSEUDO:RANDOM
;NUMBERS BUT ARE UNIFORMLY DISTRIBUTED OVER [0,1].
INTERNAL RANDOM
  ACX=5
  ACY=6
  ACZ=ACY+1 ;ACCUMULATOR SYMBOLIC DEFINITIONS.
RANDOM: 0 ;ENTERED BY JSA 16,RANDOM.
        CALL ACX,[SIXBIT/TIMER] ;GET TIME IN CLOCK TICKS.
        ANDI ACX,3 ;USE TIME TO SELECT 1-4 ITERATIONS.
RLOOP:  MOVE ACY,RNUMBR ;FETCH PREVIOUS PSEUDO-RANDOM NUMBER.
        MUL ACY,MAGIC ;MULTIPLICATIVE RANDOM NUMBER GENERATOR.
        MOVEM ACZ,RNUMBR ;SAVE NEXT PSEUDO-RANDOM NUMBER.
        SOJGE ACX,RLOOP ;ITERATE AGAIN?
        LSH ACZ,-1D8 ;CONVERT TO FLOATING POINT FORMAT.
        TLO ACZ,20000 ;IN THE RANGE [0,1].
        FADRI ACZ,0 ;NORMALIZE.
        MOVEM ACZ,@(16) ;STORE RESULT, AND
        JRA 16,1(16) ;**RETURN**
;THE MULTIPLIER USED IS 5*15 (SEE COMPUTER REVIEWS, VOL 6, #3,
;REVIEW NUMBER 7725, AND THE REFERENCED PAPER IN JOURNAL OF
;ACM, JANUARY, 1965, PP83-89).
MAGIC:      5*5*5*5*5*5*5*5*5*5*5*5*5*5*5*5 ;THE MULTIPLIER.
RNUMBR:    1 ;THE NEXT RANDOM NUMBER IS ALWAYS HERE.
;THE ITERATION STARTS FROM A VALUE OF 1.
PATCH:    BLOCK 10 ;PATCHING SPACE.
          END $

```

```

*BJ$INTERNAL$7RII$0LT$$
INTERNAL RANDOM
*SPP83$-2CI $0LT$$
;ACM, JANUARY, 1965, PP 83-89).
*EXIT$$

```

```

EXIT
↑C

```


Demonstration Program #2 Continues On Next Page

.DIRECT

```
DIRECTORY 27,20 1019    27-JAN-69
003EDS.TMP      01      27-JAN-69
RANDOM           03      27-JAN-69
003PIP.TMP      01      27-JAN-69

TOTAL BLOCKS    05

EXIT
↑C
```

.RENAME RANDOM.MAC=RANDOM

```
FILES RENAMED
RANDOM
EXIT ↑C
```

.DEBUG RANDOM/CREF,DT:ARRIVE/L

FORTRAN: ARRIVE.F4

MACRO: RANDOM

A 000001 040240 000026'

T TIME IN CLOCK TICKS.

CALL ACX,[SIXBIT/TIMER] ;GE

?1 ERROR DETECTED

LOADING

?EXECUTION DELETED

LOADER 8K CORE

EXIT

↑C

.TECO RANDOM.MAC

[3 K CORE]

*BJN/TIMER\$I/\$0LT\$\$

CALL ACX,[SIXBIT/TIMER/] ;GET TIME IN CLOCK TICKS.

*EXIT\$\$

```
EXIT
↑C
```

Typing DIRECT causes the directory of the disk area to be typed out on the console. In addition to the RANDOM file, there are two other files, 003EDS.TMP and 003PIP.TMP. These files are temporary command files created by the COMPIL CUSP and contain the commands generated by MAKE and DIRECT, respectively. Note that the assigned job number forms the first three characters of the file-names.

Change the name of the text file from RANDOM to RANDOM.MAC by typing

```
RENAME RANDOM.MAC=RANDOM.
```

Standard filename extensions should be used (e.g., .MAC for Macro-10 source program files, .F4 for FORTRAN source program files, .REL for relocatable binary files, etc.).

Use the DEBUG command as follows:

- a. Assemble your Macro-10 source program, RANDOM. Its filename extension of .MAC identifies it as a Macro program. Request that a CREF (cross-reference) listing file be produced. At a later time, this file can be listed on the printer via the CREF command.
- b. Compile your FORTRAN source program, ARRIVE, which has a filename extension of .F4, identifying it as a FORTRAN program. Request a normal listing file (/L). This previously prepared program file is on the DECTape reel that was mounted on DECTape 2 (symbolic name DT:).
- c. Load the two relocatable binary files produced by the assembly and compilation processes and also load the Dynamic Debugging Technique (DDT) program to examine and debug the program coding.
- d. If no major errors were encountered during translation and loading, begin execution under control of DDT.

In this case, however, a source coding error was encountered in the Macro-10 source program (a slash was not typed following TIMER) and execution of the programs is inhibited.

Type TECO RANDOM.MAC to recall TECO and to open an already existing file, RANDOM.MAC, for editing. Type the command string shown to insert a slash after TIMER. Set the TECO pointer at the beginning (BJ) of the text, do a nonstop (N) search for /TIMER, insert a slash (I/) following it, and type the current line (OLT).

.DELETE *.REL,*.LST

```
FILES DELETED  
ARRIVE REL  
RANDOM REL  
ARRIVE LST  
RANDOM LST
```

```
EXIT  
*C
```

```
.EXECUTE RANDOM/CREF,DT:ARRIVE/L  
FORTRAN: ARRIVE.F4  
MACRO: RANDOM  
LOADING
```

```
LOADER 6K CORE
```

```
RANDOM INTERARRIVAL TIME GENERATOR FOR POISSON PROCESSES
```

```
TYPE MEAN WAITING TIME PLEASE: 100
```

```
TYPE NUMBER OF SAMPLE TIMES DESIRED:10
```

```
T = 0.77751067E+04  
T = 0.79615811E+04  
T = 0.79469139E+04  
T = 0.78677823E+04  
T = 0.78103187E+04  
T = 0.77700529E+04  
T = 0.77820501E+04  
T = 0.77725470E+04  
T = 0.79530156E+04  
T = 0.77917609E+04
```

```
TYPE MEAN WAITING TIME PLEASE: *C
```

.DETACH

```
.LOGIN  
JOB 6 4S34  
#27,20
```

```
1029 27-JAN-69 FTY13  
THE DISK WILL BE REFRESHED DAILY AT 4:45 PM UNTIL FURTHER NOTICE.
```

```
*C
```


To repeat the translation of the programs, delete those files from your disk area that were created by the DEBUG process. These files are the two relocatable binary files (these were automatically given a filename extension of .REL) and the two listing files (these were automatically given a filename extension of .LST). The form *.ext refers to all files, regardless of filename, that have the specified extension. In this example, DELETE *.REL, *.LST causes all files with an extension of .REL and all files with an extension of .LST to be deleted. To conserve disk space, note that all temporary command files generated by Monitor commands can be periodically deleted by typing DELETE *.TMP.

An attempt is made to translate, load, and execute without DDT. The EXECUTE command has the same general format as the DEBUG command. After loading, the program will automatically begin execution.

Translation was successful; the programs are loaded; and execution is begun.

However, there is an error. The output is far from random and conspicuously in the wrong range. Return to Monitor level by typing ↑C.

The following is an example of how to detach from the current job and begin a second job without affecting the status of the current job. Detach the Teletype console by typing DETACH. A new job can now be initiated. The current job (job #3) remains in its present status until you attach to it again.

Type LOGIN (or LOG) to request another job number. Job #6 is assigned. Perform the same procedures for logging in as in Step 1.

.LIST ARRIVE.LST

†C

.CCONT

2K CORE

.ATTACH 3 [27,20]

.

.DEBUG RANDOM,ARRI
LOADING

LOADER 8K CORE

SG

RANDOM INTERARRIVAL TIME GENERATOR FOR POISSON PROCESSES

TYPE MEAN WAITING TIME PLEASE: 100

TYPE NUMBER OF SAMPLE TIMES DESIRED:10.

T = 0.77751067E+04
T = 0.79615811E+04
T = 0.79469139E+04
T = 0.78677823E+04
T = 0.78103187E+04
T = 0.77700529E+04
T = 0.77725470E+04
T = 0.80251919E+04
T = 0.78686508E+04
T = 0.77917609E+04

TYPE MEAN WAITING TIME PLEASE: †C

.DDT

LIST the listing file generated by the compilation of the FORTRAN program on the line printer.

Once the listing has begun, interrupt it by typing `↑C` to return to Monitor level.

Type `CCONT` to continue the listing and maintain the console at Monitor level.

Now, detach from this job and reattach to the original job by typing `ATTACH job# [project, programmer]`. (`ATTACH` can be abbreviated to `AT`.)

Now, attached to the original job (job #3) other tasks can be performed while the listing is being completed.

The error that caused the incorrect results (a 0 was omitted in the TLO ACZ, 20000 instruction) is determined by scanning the teletype sheet. Now, `DEBUG` the programs to correct this error.

The `DEBUG` process finds that two relocatable binary files (created during the previous `EXECUTE` process) are more recent than their related source files. Therefore, no retranslation is needed, and the existing `.REL` files are immediately loaded. Execution is begun under control of `DDT`, and `DDT` awaits a command typein.

A `$G` (`ALTMODE G`) transfers control to the programs and begins execution. Again, incorrect answers result.

To return to the `DDT` program, type (`↑C`) to return to Monitor level; then, type `"DDT."`

RANDOMS: RLOOP+5/ TLO ACZ,20000 TLO ACZ,200000
=661340,,200000 +TLO ACZ,200000

MAIN.\$: 12P+10\$T/ RED:' "/RED: /
12P+10\$T/ RED: LINEFEED
12P+11/ \$) "/ '\$)/
12P+11\$T/ '\$)

\$G

RANDOM INTERARRIVAL TIME GENERATOR FOR POISSON PROCESSES

TYPE MEAN WAITING TIME PLEASE: 100

TYPE NUMBER OF SAMPLE TIMES DESIRED: 10

T = 0.13360079E+03
T = 0.59776286E+02
T = 0.31049938E+02
T = 0.43099106E+02
T = 0.12045378E+02
T = 0.20455130E+02
T = 0.21030612E+02
T = 0.39184699E+01
T = 0.39241011E+02
T = 0.45517379E+02

TYPE MEAN WAITING TIME PLEASE: +C

Open the DDT symbol table for the Macro program by typing `RANDOM$:` (the program name, taken from the `TITLE` statement of your source program, followed by an `ALTMODE` and a colon). Now, any of the symbolic tags contained in this program can be used.

Open the location containing the erroneous instruction by typing the address of the location, relative to a symbolic tag (in this case, `RLOOP+5`). DDT types out the contents of this location in symbolic form. Now, type in the correct contents, also in symbolic form.

Typing an equal (`=`) sign causes the new contents to be typed out in halfword mode. Typing a left arrow causes the contents to be typed in symbolic. The proper instruction has been entered correctly.

During execution, no spacing was performed following the second request for input. To correct this condition, open the symbol table for the FORTRAN program (FORTRAN programs are assigned the program name `MAIN`, unless otherwise titled by the programmer), and correct that portion of the literal stored in locations `12P+10$T` and `12P+11$T` by inserting a space after the colon (`DESIRED:`). Examine the two locations to ensure that the correction was made properly (a `LINEFEED` causes the next sequential location to be opened).

Type `$G` to restart execution.

The results seem to be correct. Return to Monitor level.

.SAVE DT:PROGA
JOB SAVED
↑C

.RUN DT:PROGA

RANDOM INTERARRIVAL TIME GENERATOR FOR POISSON PROCESSES

TYPE MEAN WAITING TIME PLEASE: 100

TYPE NUMBER OF SAMPLE TIMES DESIRED: 5

T = 0.19732386E+02

T = 0.68401470E+02

T = 0.16503109E+03

T = 0.36447561E+01

T = 0.97657015E+02

TYPE MEAN WAITING TIME PLEASE: 50E+1

TYPE NUMBER OF SAMPLE TIMES DESIRED: 5

T = 0.54349454E+03

T = 0.45728928E+03

T = 0.57901405E+03

T = 0.22740226E+03

T = 0.14563251E+03

TYPE MEAN WAITING TIME PLEASE: ↑C

.TECO RANDOM.MAC

*BJNSACZ,20000\$I0\$0LT\$\$

TLO ACZ,200000 ;IN THE RANGE [0,1].

*EXIT\$\$

EXIT

↑C

.R PIP

*DT:RANDOM.MAC←DSK:RANDOM.MAC

*↑C

.DIRECT DT:

SAVE the core image of the two programs and DDT on DECtape. Assign this image file the name PROGA (an extension .SAV is automatically appended by the system).

As a double check, RUN the program you just saved on DECtape. Note that R is used to call in a program from the system device (SYS:) and RUN followed by a device name is used to call in a program from other devices.

Again, the results appear to be correct.

Use TECO to correct the Macro-10 source file to reflect the DDT correction.

Run Peripheral Interchange Program (PIP) to transfer the corrected Macro-10 source file to DECtape. At the end of every console session, it is suggested that the user transfer any files he might want to reuse from the disk area to tape or other storage medium. This procedure releases the disk space for other users and ensures that a refreshing of the disk will not destroy the only copy of a file.

Obtain a DIRECTory listing of the DECtape to ensure that it contains all the files you want to preserve. (DIRECT can be abbreviated as DIR.)

524. FREE BLOCKS LEFT
PROGA .SAV 27-JAN-69
RANDOM.MAC 27-JAN-69
ARRIVE.F4 22-JAN-69

EXIT
↑C

.KJOB
JOB 3, ONE OF USER 27,20 OFF TTY13 AT 1108 ON 27-JAN-69
FILES DELETED: 0, FILES SAVED: ALL, RUNTIME 0 MIN, 20 SEC

.
AT 6 [27,20]

.K
CONFIRM: K
JOB 6, USER 27,20 OFF TTY13 AT 1118 ON 27-JAN-69
FILES DELETED: 11, FILES SAVED: 0, RUNTIME 0 MIN, 02 SEC

Kill the job. Monitor responds by printing the job number, project-programmer number; Teletype name; time, date, number of disk files deleted/saved; and the total run time.

ATTACH (or AT) to the second job (job #6), and kill it also. Following the CONFIRM: message, several options are available to determine what is to be done with the disk files. In this particular case, because all files that are to be preserved have already been stored on DECTape, type K to cause all disk files to be deleted.

Monitor prints the same information as in Step 13. You are now off the system; the core memory, disk space, and DECTape unit have been returned to the Monitor pool for others to use.

END OF DEMONSTRATION SESSION

Book 1

**Programming
with the
PDP-10
Instruction Set**

PDP-10
System Reference Manual

Changes are indicated by a triangle (Δ) in the outside margin

Contents

1	INTRODUCTION	1-1
1.1	Number System	1-4
	Floating point arithmetic	1-5
1.2	Instruction Format	1-6
	Effective address calculation	1-7
1.3	Memory	1-8
	Memory allocation	1-9
1.4	Programming Conventions	1-10
2	CENTRAL PROCESSOR	2-1
2.1	Half Word Data Transmission	2-2
2.2	Full Word Data Transmission	2-9
	Move instructions	2-10
	Pushdown list	2-12
2.3	Byte Manipulation	2-15
2.4	Logic	2-17
	Shift and rotate	2-24
2.5	Fixed Point Arithmetic	2-26
	Arithmetic shifting	2-31
2.6	Floating Point Arithmetic	2-32
	Scaling	2-33
	Operations with rounding	2-34
	Operations without rounding	2-37
2.7	Arithmetic Testing	2-41
2.8	Logical Testing and Modification	2-47
2.9	Program Control	2-54
2.10	Unimplemented Operations	2-64
2.11	Programming Examples	2-65
	Double precision floating point	2-67
2.12	Input-Output	2-68
	Readin mode	2-72
	Console data transfers	2-73

2.13	Priority Interrupt	2-73
2.14	Processor Conditions	2-78
2.15	Time Sharing	2-81
	User programming 2-82	
	Monitor programming 2-83	
2.16	Operation	2-84
	Indicators 2-84	
	Operating keys 2-87	
	Operating switches 2-89	
3	BASIC IN-OUT EQUIPMENT	
3.1	Paper Tape Reader	3-1
	Readin mode 3-4	
3.2	Paper Tape Punch	3-5
3.3	Teletype	3-7
4	HARDCOPY EQUIPMENT	4-1
4.1	Line Printer	4-1
4.2	Plotter	4-9
4.3	Card Reader	4-14
▲ 4.4	Card Punch	4-18
	APPENDICES	
A	Instruction and Device Mnemonics	A1
	Numeric listing A3	
	Alphabetic listing A6	
	Device mnemonics A10	
B	In-out Codes	B1
	Teletype code B2	
	Card codes B6	
C	Miscellany	C1
D	Algorithms	D1
	Fixed point algorithms D2	
	Floating point algorithms D7	
	INDEX	II

1

Introduction

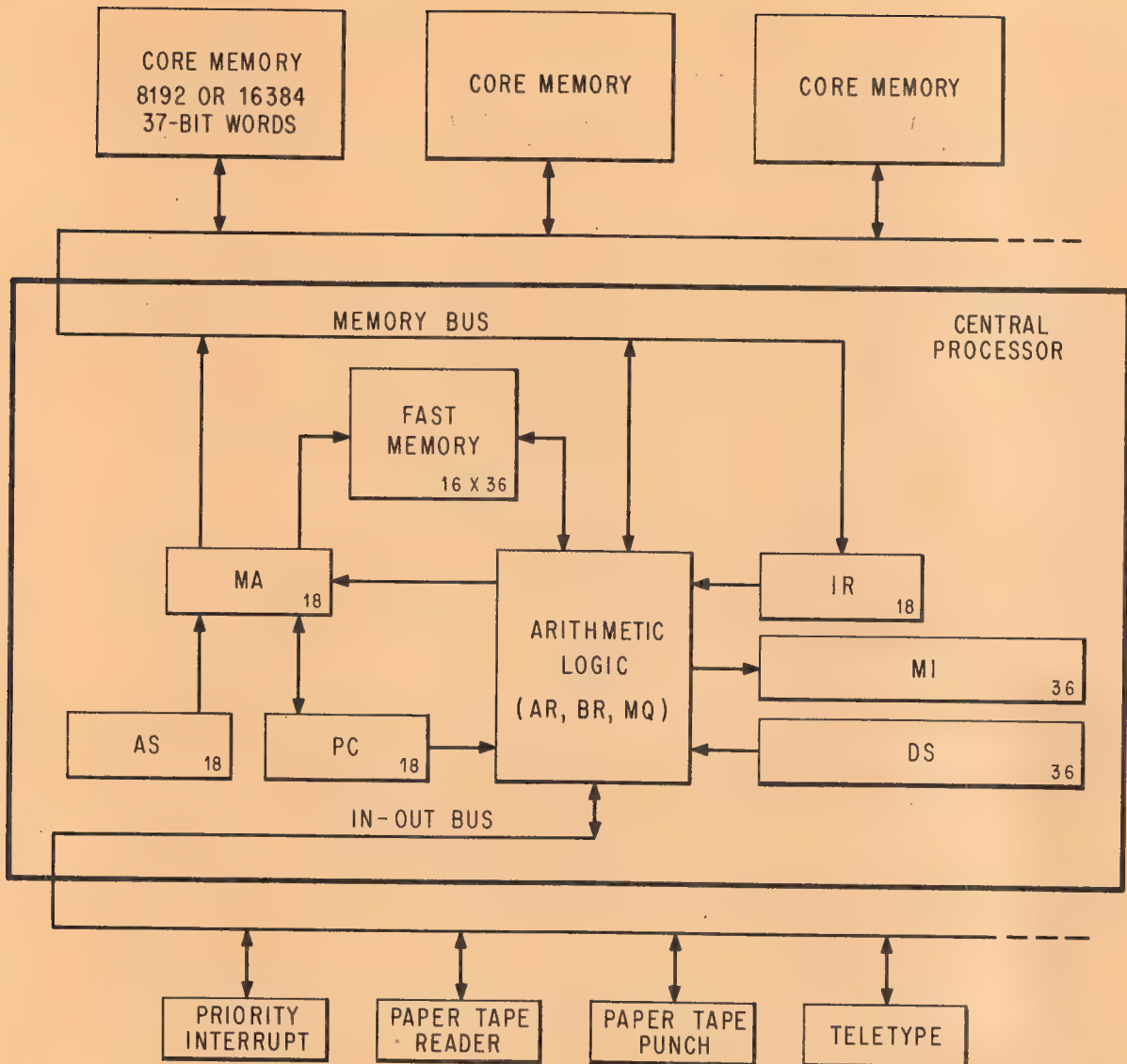
The PDP-10 is a general purpose, stored program computer that includes a central processor, a memory, and a variety of peripheral equipment such as paper tape reader and punch, teletype, card reader, line printer, DECtape, magnetic tape, disk file and display. The central processor is the control unit for the entire system: it governs all peripheral in-out equipment, sequences the program, and performs all arithmetic, logical and data handling operations. The processor is connected to one or more memory units by a memory bus and to the peripheral equipment by an in-out bus. The fastest devices, such as the disc file, although controlled by the processor over the in-out bus, have direct access to memory over a second memory bus.

The processor handles words of thirty-six bits, which are stored in a memory with a maximum capacity of 262,144 words. Storage in memory is usually in the form of 37-bit words, the extra bit producing odd parity for the word. The bits of a word are numbered 0-35, left to right, as are the bits in the registers that handle the words. The processor can also handle half words, wherein the left half comprises bits 0-17, the right half, bits 18-35. Optional hardware is available for byte manipulation — a byte is any contiguous set of bits within a word. Registers that hold addresses have eighteen bits, numbered 18-35 according to the position of the address in a word. Words are used either as computer instructions in the program, as addresses, or as operands (data for the program).

Of the internal registers shown in the illustration on the next page, only PC, the 18 bit program counter, is directly relevant to the programmer. The processor performs a program by executing instructions retrieved from the locations addressed by PC. At the beginning of each instruction PC is incremented by one so that it normally contains an address one greater than the location of the current instruction. Sequential program flow is altered by changing the contents of PC, either by incrementing it an extra time in a skip instruction or by replacing its contents with the value specified by a jump instruction. Also of importance to the programmer is the 36-bit data switch register DS on the processor console: through this register the program can read data supplied by the operator. The processor also contains flags that detect various types of errors, including several types of overflow in arithmetic and pushdown operations, and provide other information of interest to the programmer.

The processor has other registers but the programmer is not usually concerned with them except when manually stepping through a program to debug it. By means of the address switch register AS, the operator can

INTRODUCTION



PDP-10 SIMPLIFIED

examine the contents of, or deposit information into, any memory location; stop or interrupt the program whenever a particular location is referenced; and through AS the operator can supply a starting address for the program. Through the memory indicators MI the program can display data for the operator. The instruction register IR contains the left half of the current instruction word, *ie* all but the address part. The memory address register MA supplies the address for every memory access. The heart of the processor is the arithmetic logic, principally the 36-bit arithmetic register AR.

This register takes part in all arithmetic, logical and data handling operations; all data transfers to and from memory, peripheral equipment and console are made via AR. Associated with AR are an extremely fast full adder, a buffer register BR that holds a second operand in many arithmetic and logical instructions, a multiplier-quotient register MQ that serves primarily as an extension of AR for handling double length operands, and smaller registers that handle floating point exponents and control shift operations and byte manipulation.

From the point of view of the programmer however the arithmetic logic can be regarded as a black box. It performs almost all of the operations necessary for the execution of a program, but it never retains any information from one instruction to the next. Computations performed in the black box either affect control elements such as PC and the flags, or produce results that are always sent to memory and must be retrieved by the processor if they are to be used as operands in other instructions.

An instruction word has only one 18-bit address field for addressing any location throughout all of memory. But most instructions have two 4-bit fields for addressing the first sixteen memory locations. Any instruction that requires a second operand has an accumulator address field, which can address one of these sixteen locations as an accumulator; in other words as though it were a result held over in the processor from some previous instruction (the programmer usually has a choice of whether the result of the instruction will go to the location addressed as an accumulator or to that addressed by the 18-bit address field, or to both). Every instruction has a 4-bit index register address field, which can address fifteen of these locations for use as index registers in modifying the 18-bit memory address (a zero index register address specifies no indexing). Although all computations on both operands and addresses are performed in the single arithmetic register AR, the computer actually has sixteen accumulators, fifteen of which can double as index registers. The factor that determines whether one of the first sixteen locations in memory is an accumulator or an index register is not the information it contains nor how its contents are used, but rather how the location is addressed. There need be no difference physically between these locations and other memory locations, but an optional, fast flip-flop memory contained in the processor can be substituted for the bottom sixteen locations in core. This allows much quicker access to these locations whether they are addressed as accumulators, index registers or ordinary memory locations. They can even be addressed from the program counter, gaining faster execution for a short but oft-repeated subroutine.

Besides the registers that enter into the regular execution of the program and its instructions, the processor has a priority interrupt system and can contain optional equipment to facilitate time sharing. The interrupt system facilitates processor control of the peripheral equipment by means of a number of priority-ordered channels over which external signals may interrupt the normal program flow. The processor acknowledges an interrupt request by executing the instruction contained in a particular location assigned to the channel. Assignment of channels to devices is entirely under program control. One of the devices to which the program can assign a channel is the processor itself, allowing internal conditions such as overflow or a parity

error to signal the program.

The time share hardware provides memory protection and relocation. Without time sharing, all instructions and all memory are available to the program. Otherwise a number of programs share processor time, with each program relocated and restricted to a specific area in core, and certain instructions are usually illegal. An attempt by any user to execute an illegal instruction or address a memory location outside of his area results in a transfer of control back to the time-sharing monitor.

1.1 NUMBER SYSTEM

The program can interpret a data word as a 36-digit, unsigned binary number, or the left and right halves of a word can be taken as separate 18-bit numbers. The PDP-10 repertoire includes instructions that effectively add or subtract one from both halves of a word, so the right half can be used for address modification when the word is addressed as an index register, while the left half is used to keep a control count.

The standard arithmetic instructions in the PDP-10 use twos complement, fixed point conventions to do binary arithmetic. In a word used as a number, bit 0 (the leftmost bit) represents the sign, 0 for positive, 1 for negative. In a positive number the remaining 35 bits are the magnitude in ordinary binary notation. The negative of a number is obtained by taking its twos complement. If x is an n -digit binary number, its twos complement is $2^n - x$, and its ones complement is $(2^n - 1) - x$, or equivalently $(2^n - x) - 1$. Subtracting a number from $2^n - 1$ (*ie*, from all 1s) is equivalent to performing the logical complement, *ie* changing all 0s to 1s and all 1s to 0s. Therefore, to form the twos complement one takes the logical complement (usually referred to merely as the complement) of the entire word including the sign, and adds 1 to the result. In a negative number the sign bit is 1, and the remaining bits are the twos complement of the magnitude.

$$+153_{10} = +231_8 = \boxed{000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 000\ 010\ 011\ 001}$$

035

$$-153_{10} = -231_8 = \boxed{111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 111\ 101\ 100\ 111}$$

035

Zero is represented by a word containing all 0s. Complementing this number produces all 1s, and adding 1 to that produces all 0s again. Hence there is only one zero representation and its sign is positive. Since the numbers are symmetrical in magnitude about a single zero representation, all even numbers both positive and negative end in 0, all odd numbers in 1 (a number all 1s represents -1). But since there are the same number of positive and negative numbers and zero is positive, there is one more negative number than there are nonzero positive numbers. This is the most negative number and it cannot be produced by negating any positive number (its octal representa-

§1.1

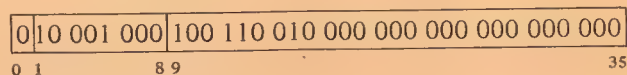
tion is $400000\ 000000_8$ and its magnitude is one greater than the largest positive number).

If ones complements were used for negatives one could read a negative number by attaching significance to the 0s instead of the 1s. In twos complement notation each negative number is one greater than the complement of the positive number of the same magnitude, so one can read a negative number by attaching significance to the rightmost 1 and attaching significance to the 0s at the left of it (the negative number of largest magnitude has a 1 in only the sign position). In a negative integer, 1s may be discarded at the left, just as leading 0s may be dropped in a positive integer. In a negative fraction, 0s may be discarded at the right. So long as only 0s are discarded, the number remains in twos complement form because it still has a 1 that possesses significance; but if a portion including the rightmost 1 is discarded, the remaining part of the fraction is now a ones complement.

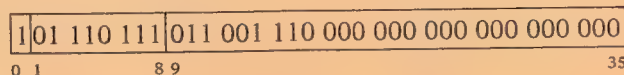
The computer does not keep track of a binary point — the programmer must adopt a point convention and shift the magnitude of the result to conform to the convention used. Two common conventions are to regard a number as an integer (binary point at the right) or as a proper fraction (binary point at the left); in these two cases the range of numbers represented by a single word is -2^{35} to $2^{35} - 1$ or -1 to $1 - 2^{-35}$. Since multiplication and division make use of double length numbers, there are special instructions for performing these operations with integral operands.

Floating Point Arithmetic. Optional PDP-10 hardware is available for processing floating point numbers. A floating point instruction interprets bit 0 of a word as the sign, but interprets the rest of the word as an 8-bit exponent and a 27-bit fraction. For a positive number the sign is 0, as before. But the contents of bits 9-35 are now interpreted only as a binary fraction, and the contents of bits 1-8 are interpreted as an integral exponent in excess 128 (200_8) code. Exponents from -128 to $+127$ are therefore represented by the binary equivalents of 0 to 255 ($0-377_8$). Floating point zero and negatives are represented in exactly the same way as in fixed point: zero by a word containing all 0s, a negative by the twos complement. A negative number has a 1 for its sign and the twos complement of the fraction, but since every fraction must ordinarily contain a 1 unless the entire number is zero (see below), it has the ones complement of the exponent code in bits 1-8. Since the exponent is in excess 128 code, an actual exponent x is represented in a positive number by $x + 128$, in a negative number by $127 - x$. The programmer, however, need not be concerned with these representations as the hardware compensates automatically. *Eg*, for

$$+153_{10} = +231_8 = +.462_8 \times 2^8 =$$



$$-153_{10} = -231_8 = -.462_8 \times 2^8 =$$

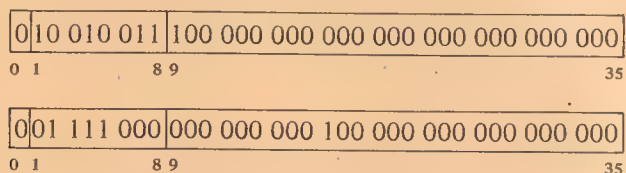


Multiplication produces a double length product, and the programmer must remember that discarding the low order part of a double length negative leaves the high order part in correct twos complement form only if the low order part is null.

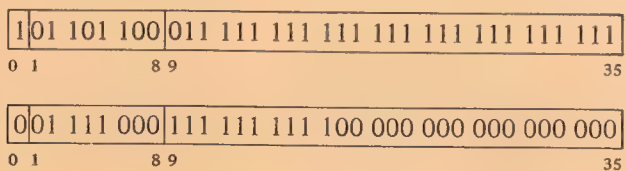
the instruction that scales the exponent, the hardware interprets the integral scale factor in standard two's complement form but produces the correct one's complement result for the exponent.

Except in special cases the floating point instructions assume that all non-zero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the magnitude of the fraction is greater than or equal to $\frac{1}{2}$ and less than 1. These numbers thus have a fractional range in magnitude of $\frac{1}{2}$ to $1 - 2^{-27}$ and an exponent range of -128 to $+127$. The hardware may not give the correct result if the program supplies an operand that is not normalized or that has a zero fraction with a nonzero exponent.

The precaution about truncation given for fixed point multiplication applies to all floating point operations as they all produce extra length results; but here the programmer may request rounding, which automatically restores the high order part to two's complement form if it is negative. In division the two words of the result are quotient and remainder, but in the other operations they form a double length number which is stored in two accumulators if the instruction is executed in "long" mode. This number contains a 54-bit fraction, half of which is in bits 9–35 of each word. The sign and exponent are in bits 0 and 1–8 respectively of the word containing the more significant half, and the standard two's complement is used to form the negative of the entire 63-bit string. In the remaining part of the less significant word, bit 0 is 0, and bits 1–8 contain a number 27 less than the exponent, but this is expressed in positive form even though bits 9–35 may be part of a negative fraction. *Eg* the number $2^{18} + 2^{-18}$ has this two-word representation:



whereas its negative is

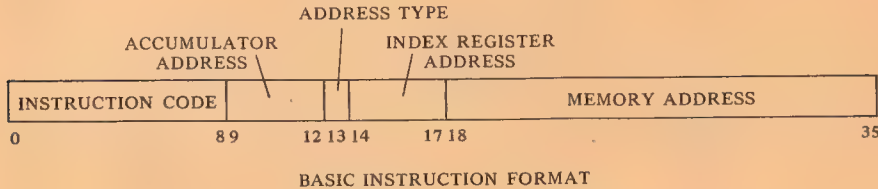


1.2 INSTRUCTION FORMAT

In all but the input-output instructions, the nine high order bits (0–8) specify the operation, and bits 9–12 usually address an accumulator but are sometimes used for special control purposes, such as addressing flags. The

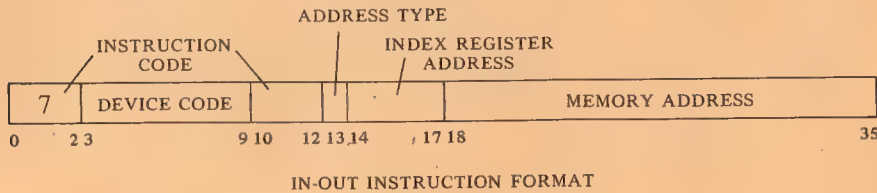
§1.2

rest of the instruction word usually supplies information for calculating the effective address, which is the actual address used to fetch the operand or alter program flow. Bit 13 specifies the type of addressing, bits 14–17 specify an index register for use in address modification, and the remaining eighteen bits (18–35) address a memory location. The instruction codes



that are not assigned as specific instructions are executed by the processor as so-called “unimplemented operations”, as are the codes for floating point and byte manipulation in any PDP-10 that does not have the optional hardware for these instructions. When the processor encounters one of these unimplemented codes in a program, it stores bits 0–12 of the instruction word and the calculated effective address in a particular memory location and then executes the instruction contained in a second location.

An input-output instruction is designated by three 1s in bits 0–2. Bits 3–9 address the in-out device to be used in executing the instruction, and bits 10–12 specify the operation. The rest of the word is the same as in other instructions.



Effective Address Calculation. Bits 13–35 have the same format in *every* instruction whether it addresses a memory location or not. Bit 13 is the



indirect bit, bits 14–17 are the index register address, and if the instruction must reference memory, bits 18–35 are the memory address Y . The effective address E of the instruction depends on the values of I , X and Y . If X is nonzero, the contents of index register X are added to Y to produce a modified address. If I is 0, addressing is direct, and the modified address is the effective address used in the execution of the instruction; if I is 1, addressing is indirect, and the processor retrieves another address word from the location specified by the modified address already determined. This new word is processed in exactly the same manner: X and Y determine the effective address if I is 0, otherwise they are used for yet another level of address

retrieval. This process continues until some referenced location is found with a 0 in bit 13; the 18-bit number calculated from the X and Y parts of this location is the effective address E .

The calculation outlined above is carried out for *every* instruction even if it need not address a memory location. If the indirect bit in the instruction word is 0 and no memory reference is necessary, then Y is not an address. It may be a mask in some kind of test instruction, conditions to be sent to an in-out device, or part of it may be the number of places to shift in a shift or rotate instruction or the scale factor in a floating scale instruction. Even when modified by an index register, bits 18–35 do not contain an address when I is 0. But when I is 1, the number determined from bits 14–35 is an indirect address no matter what type of information the instruction requires, and the word retrieved in any step of the calculation contains an indirect address so long as I remains 1. When a location is found in which I is 0, bits 18–35 (perhaps modified by an index register) contain the desired effective mask, effective conditions, effective shift number, or effective scale factor. Many of the instructions that usually reference memory for an operand even have an “immediate” mode in which the result of the effective address calculation is itself used as a half word operand instead of a word taken from the memory location it addresses.

The important thing for the programmer to remember is that the same calculation is carried out for every instruction regardless of the type of information that must be specified for its execution, or even if the result is ignored. In the discussion of any instruction, E refers to the actual quantity derived from I , X and Y and used in the execution of the instruction, be it the entire half word as in the case of an address, immediate operand, mask or conditions, or only part of it as in a shift number or scale factor.

1.3 MEMORY

All timing in the PDP-10 is asynchronous. The internal timing for each in-out device and each memory is entirely independent of the central processor. Because core memory readout is destructive, every word read must be written back in unless new information is to take its place. The basic read-write cycle time of the standard core memory is either 1.00 or 1.65 microseconds, but the processor need never wait the entire cycle time. To read, it waits only until the information is available and then continues its operations while the memory performs the write portion of the cycle; to write, it waits only until the data is accepted, and the memory then performs an entire cycle to clear and write. To save time in an instruction that fetches an operand and then writes new data into the same location, the memory executes a read-pause-write cycle in which it performs only the read part initially and then completes the cycle when the processor supplies the new data.

Access times for the accumulator-index register locations are decreased considerably by substitution of a fast memory (contained in the processor) for the first sixteen core locations. Readout is nondestructive, so the fast memory has no basic cycle: the processor reads a word directly, but to write

it must first clear the location and then load it. Access times in nanoseconds (including 20 feet of cable delay) for the three memories are as follows.

	<i>Read</i>	<i>Write</i>
MA10 or MA10A Core Memory (1.00 μ s)	580	200
MB10 Core Memory (1.65 μ s)	600 (700)*	200 (300)
KM10 Fast Memory (18-bit address)	210	210

*Numbers in parentheses are the longer times required in a multiprocessor system.

NOTE: When a fast memory location is addressed as an accumulator or index register, the access time is usually considerably shorter than that listed here.

From the simple addressing point of view, the entire memory is a set of contiguous locations whose addresses range from zero to a maximum dependent upon the capacity of the particular installation. In a system with the greatest possible capacity, the largest address is octal 777777, decimal 262,143. (Addresses are always in octal notation unless otherwise specified.) But the whole memory would usually be made up of a number of core memories each having a capacity of 8192 or 16,384 words. Hence a single 18-bit address actually selects a particular memory and a specific location within it. For an 8K memory the high order five address bits select the memory, the remaining thirteen bits address a single location in it; selecting a 16K memory takes four bits, leaving fourteen for the location. The times given above assume the addressed memory is idle when access is requested. To avoid waiting for a previously requested memory cycle to end, the program can make consecutive requests to different memories by taking instructions from one memory and data from another. The hardware also allows pairs of memories to be interleaved in such a way that consecutive addresses actually alternate between the two memories in the pair (thus increasing the probability that consecutive references are to different memories). Appropriate switch settings at the memories interchange the least significant address bits in the memory and location parts, so that in any two memories numbered n and $n + 1$ where n is even, all even addresses are locations in the first memory, all odd addresses are locations in the second. Hence memories 0 and 1 can be interleaved as can 6 and 7, but not 3 and 4 or 5 and 7.

Memory Allocation. The use of certain memory locations is defined by the hardware.

0	Holds a pointer word during a bootstrap readin
0-17	Can be addressed as accumulators
1-17	Can be addressed as index registers
40-41	Trap for unimplemented user operations (UUOs)
42-57	Priority interrupt locations
60-61	Trap for remaining unimplemented operations: these include the unassigned instruction codes that are reserved for future use, and also the byte manipulation and floating point instructions when the hardware for them is not installed
140-161	Allocated to second processor if connected (same use as 40-61 for first processor)

All information given in this manual about memory locations 40-61 applies instead to locations 140-161 for programming a second central processor connected to the same memory.

The initial control word address for the DF10 Data Channel must be less than 1000.

1.4 PROGRAMMING CONVENTIONS

The computer has five instruction classes: data transmission, logical, arithmetic, program control and in-out. The instructions in the in-out class control the peripheral equipment, and also control the priority interrupt and time sharing, control and read the processor flags, and communicate with the console. The next chapter describes all instructions mentioned above, presents a general description of input-output, and describes the effects of the in-out instructions on the processor, priority interrupt and time share hardware. Effects of in-out instructions on particular peripheral devices are discussed with the devices.

The MACRO-10 assembly program recognizes a number of mnemonics and other initial symbols that facilitate constructing complete instruction words and organizing them into a program. In particular there are mnemonics for the instruction codes (Appendix A), which are six bits in in-out instructions, otherwise nine or thirteen bits. *Eg* the mnemonic

MOVNS

assembles as 213000 000000, and

MOVNS 2570

assembles as 213000 002570. This latter word, when executed as an instruction, produces the twos complement negative of the word in memory location 2570.

NOTE

Throughout this manual all numbers representing instruction words, register contents, codes and addresses are always octal, and any numbers appearing in program examples are octal unless otherwise indicated. On the other hand, the ordinary use of numbers in the text to count steps in an operation or to specify word or byte lengths, bit positions, exponents, etc employs standard decimal notation.

The initial symbol @ preceding a memory address places a 1 in bit 13 to produce indirect addressing. The example given above uses direct addressing, but

MOVNS @2570

assembles as 213020 002570, and produces indirect addressing. Placing the number of an index register (1-17) in parentheses following the memory address causes modification of the address by the contents of the specified register. Hence

MOVNS @2570(12)

which assembles as 213032 002570, produces indexing using index register 12, and the processor then uses the modified address to continue the effective address calculation.

An accumulator address (0-17) precedes the memory address part (if any)

The assembler translates every statement into a 36-bit word, placing 0s in all bits whose values are unspecified.

§1.4

and is terminated by a comma. Thus

```
MOVNS 4,@2570(12)
```

assembles as 213232 002570, which negates the word in location *E* and stores the result in both *E* and in accumulator 4. The same procedure may be used to place 1s in bits 9–12 when these are used for something other than addressing an accumulator, but mnemonics are available for this purpose.

The device code in an in-out instruction is given in the same manner as an accumulator address (terminated by a comma and preceding the address part), but the number given must correspond to the octal digits in the word (000–774). Mnemonics are however available for all standard device codes. To control the priority interrupt system whose code is 004, one may give

```
CONO 4,1302
```

which assembles as 700600 001302, or equivalently

```
CONO PI,1302
```

The programming examples in this manual use the following addressing conventions:

◆ A colon following a symbol indicates that it is a symbolic location name.

```
A:      ADD    6,5704
```

indicates that the location that contains ADD 6,5704 may be addressed symbolically as A.

◆ The period represents the current address, *eg*

```
ADD    5,.+2
```

is equivalent to

```
A:      ADD    5,A+2
```

◆ Square brackets specify the contents of a location, leaving the address of the location implicit but unspecified. *Eg*

```
ADD    12,[7256004]
```

and

```
ADD    12,A
```

```
:
```

```
A:      7256004
```

are equivalent.

Anything written at the right of a semicolon is commentary that explains the program but is not part of it.

2

Central Processor

This chapter describes all PDP-10 instructions but does not discuss the effects of those in-out instructions that address specific peripheral devices. In the description of each instruction, the mnemonic and name are at the top, the format is in a box below them. The mnemonic assembles to the word in the box, where bits in those parts of the word represented by letters assemble as 0s. The letters indicate portions that must be added to the mnemonic to produce a complete instruction word.

For many of the non-IO instructions, a description applies not to a unique instruction with a single code in bits 0-8, but rather to an instruction set defined as a basic instruction that can be executed in a number of modes. These modes define properties subsidiary to the basic operation; *eg* in data transmission the mode specifies which of the locations addressed by the instruction is the source and which the destination of the data, in test instructions it specifies the condition that must be satisfied for a jump or skip to take place. The mnemonic given at the top is for the basic mode; mnemonics for the other forms of the instruction are produced by appending letters directly to the basic mnemonic. Following the description is a table giving the mnemonics and octal codes (bits 0-8) for the various modes.

The processor execution time for each instruction is also given at the top unless the time differs from one mode to another. The time listed is that required for direct addressing without indexing (*ie* with no effective address calculation), assuming the instruction and location *E* are both in the same 1.00 microsecond core memory, and that an accumulator is addressed only if necessary and is in fast memory. The time that can be saved (if any) by interleaving or keeping instructions and operands in different memories is indicated either with the description or with the discussion of the modes preceding a group of instructions. To determine the exact time required for an instruction under any circumstances, refer to the timing chart in Appendix C.

In a description *E* refers to the effective address, half word operand, mask, conditions, shift number or scale factor calculated from the *I*, *X* and *Y* parts of the instruction word. In an instruction that ordinarily references memory, a reference to *E* as the source of information means that the instruction retrieves the word contained in location *E*; as a destination it means the instruction stores a word in location *E*. In the immediate mode of these instructions, the effective half word operand is usually treated as a full word that contains *E* in one half and zero in the other, and is represented either as *0,E* or *E,0* depending upon whether *E* is in the right or left half.

Letters representing modes are suffixes, which produce new mnemonics that are recognized as distinct symbols by the assembler.

The times listed should be regarded as good approximations. For more exact times with the conditions given here (*ie* 1.00 microsecond core, etc) add 60 nanoseconds to the listed time, plus an additional 30 nanoseconds for each core memory access for retrieval of an operand and another 30 nanoseconds if the instruction does not write a result in core. ▲

Most of the non-IO instructions can address an accumulator, and in the box showing the format this address is represented by A ; in the description, "AC" refers to the accumulator addressed by A . "AC left" and "AC right" refer to the two halves of AC. If an instruction uses two accumulators, these have addresses A and $A+1$, where the second address is 0 if A is 17. In some cases an instruction uses an accumulator only if A is nonzero: a zero address in bits 9–12 specifies no accumulator.

It is assumed throughout that time sharing is not in effect, and the program is unrestricted. For completeness, however, the effects of restrictions on particular instructions are noted; and execution times are given both for unrestricted operation and including relocation in a user program (the latter time is given in parentheses). §2.15 lists all restrictions on user programs and explains the special effects produced by certain instructions when executed under control of the monitor while the processor is in user mode.

Some simple examples are included with the instruction descriptions, but more complex examples using a variety of instructions are given in §2.11.

2.1 HALF WORD DATA TRANSMISSION

These instructions move a half word and may modify the contents of the other half of the destination location. There are sixteen instructions determined by which half of the source word is moved to which half of the destination, and by which of four possible operations is performed on the other half of the destination. The basic mnemonics are three letters that indicate the transfer

HLL	Left half of source to left half of destination
HRL	Right half of source to left half of destination
HRR	Right half of source to right half of destination
HLR	Left half of source to right half of destination

plus a fourth, if necessary, to indicate the operation.

<i>Operation</i>	<i>Suffix</i>	<i>Effect on Other Half of Destination</i>
Do nothing		None
Zeros	Z	Places 0s in all bits of the other half
Ones	O	Places 1s in all bits of the other half
Extend	E	Places the sign (the leftmost bit) of the half word moved in all bits of the other half. This action extends a right half word number into a full word number but is valid arithmetically only for positive left half word numbers — the right extension of a number requires 0s regardless of sign (hence the Zeros operation should be used to extend a left half word number).

§2.1

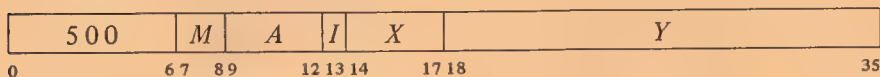
An additional letter may be appended to indicate the mode, which determines the source and destination of the half word moved.

<i>Mode</i>	<i>Suffix</i>	<i>Source</i>	<i>Destination</i>
Basic		<i>E</i>	AC
Immediate	I	The word 0, <i>E</i>	AC
Memory	M	AC	<i>E</i>
Self	S	<i>E</i>	<i>E</i> , but also AC if <i>A</i> is nonzero

Note that selecting the left half of the source in immediate mode merely clears the selected half of the destination.

Keeping instructions and operands in different memories saves .20 (.09) μ s in self mode; in memory mode the same saving results if no action is taken on the other half, otherwise .47 (.36) μ s is saved.

When *E* addresses a fast memory location, a half word transfer takes .34 μ s less in basic mode, either .46 (.35) or .54 (.43) μ s less in memory mode depending respectively on whether or not any action is taken on the other half, and .54 (.43) μ s less in self mode.

HLL Half Word Left to Left

Move the left half of the source word specified by *M* to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination are lost.

HLL	Half Left to Left	500	2.35 (2.57) μ s
HLLI	Half Left to Left Immediate	501	1.50 (1.61) μ s
HLLM	Half Left to Left Memory	502	2.90 (3.01) μ s
HLLS	Half Left to Left Self	503	2.76 (2.87) μ s

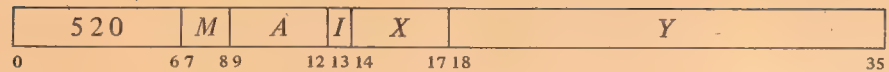
HLLI merely clears AC left. If *A* is zero, HLLS is a no-op, otherwise it is equivalent to HLL.

HLLZ Half Word Left to Left, Zeros

Move the left half of the source word specified by *M* to the left half of the specified destination, and clear the destination right half. The source is unaffected, the original contents of the destination are lost.

HLLZ	Half Left to Left, Zeros	510	2.21 (2.43) μ s
HLLZI	Half Left to Left, Zeros, Immediate	511	1.36 (1.47) μ s
HLLZM	Half Left to Left, Zeros, Memory	512	2.47 (2.58) μ s
HLLZS	Half Left to Left, Zeros, Self	513	2.76 (2.87) μ s

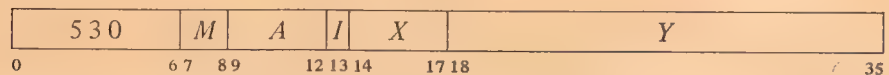
HLLZI merely clears AC. If *A* is zero, HLLZS merely clears the right half of location *E*.

HLLO Half Word Left to Left, Ones

Move the left half of the source word specified by *M* to the left half of the specified destination, and set the destination right half to all 1s. The source is unaffected, the original contents of the destination are lost.

HLLO	Half Left to Left, Ones	520	2.21 (2.43) μ s
HLLOI	Half Left to Left, Ones, Immediate	521	1.36 (1.47) μ s
HLLOM	Half Left to Left, Ones, Memory	522	2.47 (2.58) μ s
HLLOS	Half Left to Left, Ones, Self	523	2.76 (2.87) μ s

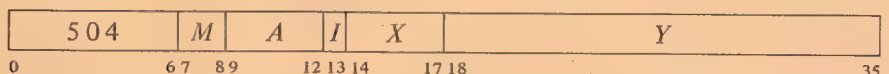
HLLOI sets AC to all 0s in the left half, all 1s in the right.

HLLE Half Word Left to Left, Extend

Move the left half of the source word specified by *M* to the left half of the specified destination, and make all bits in the destination right half equal to bit 0 of the source. The source is unaffected, the original contents of the destination are lost.

HLLEI is equivalent to HLLZI (it merely clears AC).

HLLE	Half Left to Left, Extend	530	2.21 (2.43) μ s
HLLEI	Half Left to Left, Extend, Immediate	531	1.36 (1.47) μ s
HLLEM	Half Left to Left, Extend, Memory	532	2.47 (2.58) μ s
HLLES	Half Left to Left, Extend, Self	533	2.76 (2.87) μ s

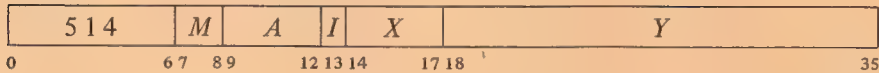
HRL Half Word Right to Left

Move the right half of the source word specified by *M* to the left half of the specified destination. The source and the destination right half are unaffected; the original contents of the destination left half are lost.

HRL	Half Right to Left	504	2.70 (2.92) μ s
HRLI	Half Right to Left Immediate	505	1.85 (1.96) μ s

§2.1

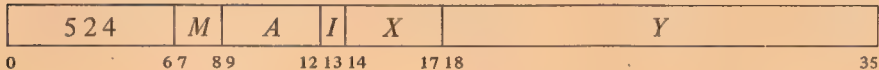
HRLM	Half Right to Left Memory	506	2.90 (3.01) μ s
HRLS	Half Right to Left Self	507	2.76 (2.87) μ s

HRLZ Half Word Right to Left, Zeros

Move the right half of the source word specified by *M* to the left half of the specified destination, and clear the destination right half. The source is unaffected, the original contents of the destination are lost.

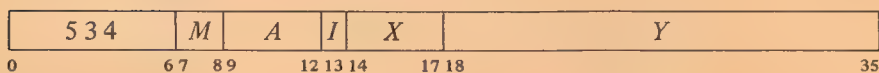
HRLZ	Half Right to Left, Zeros	514	2.21 (2.43) μ s
HRLZI	Half Right to Left, Zeros, Immediate	515	1.36 (1.47) μ s
HRLZM	Half Right to Left, Zeros, Memory	516	2.47 (2.58) μ s
HRLZS	Half Right to Left, Zeros, Self	517	2.76 (2.87) μ s

HRLZI loads the word *E,0* into AC.

HRLO Half Word Right to Left, Ones

Move the right half of the source word specified by *M* to the left half of the specified destination, and set the destination right half to all 1s. The source is unaffected, the original contents of the destination are lost.

HRLO	Half Right to Left, Ones	524	2.21 (2.43) μ s
HRLOI	Half Right to Left, Ones, Immediate	525	1.36 (1.47) μ s
HRLOM	Half Right to Left, Ones, Memory	526	2.47 (2.58) μ s
HRLOS	Half Right to Left, Ones, Self	527	2.76 (2.87) μ s

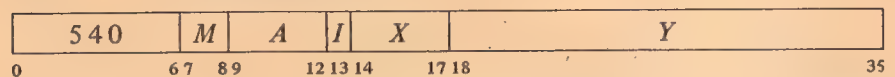
HRLE Half Word Right to Left, Extend

Move the right half of the source word specified by *M* to the left half of the

specified destination, and make all bits in the destination right half equal to bit 18 of the source. The source is unaffected, the original contents of the destination are lost.

HRLE	Half Right to Left, Extend	534	
			2.21 (2.43) μ s
HRLEI	Half Right to Left, Extend, Immediate	535	
			1.36 (1.47) μ s
HRLEM	Half Right to Left, Extend, Memory	536	
			2.47 (2.58) μ s
HRLES	Half Right to Left, Extend, Self	537	
			2.76 (2.87) μ s

HRR Half Word Right to Right

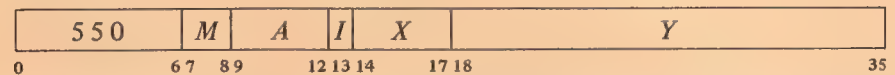


Move the right half of the source word specified by *M* to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

HRR	Half Right to Right	540	2.35 (2.57) μ s
HRRRI	Half Right to Right Immediate	541	1.50 (1.61) μ s
HRRRM	Half Right to Right Memory	542	2.90 (3.01) μ s
HRRRS	Half Right to Right Self	543	2.76 (2.87) μ s

If *A* is zero, HRRS is a no-op; otherwise it is equivalent to HRR.

HRRZ Half Word Right to Right, Zeros

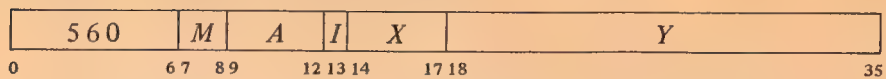


Move the right half of the source word specified by *M* to the right half of the specified destination, and clear the destination left half. The source is unaffected, the original contents of the destination are lost.

HRRZ	Half Right to Right, Zeros	550	
			2.21 (2.43) μ s
HRRZI	Half Right to Right, Zeros, Immediate	551	
			1.36 (1.47) μ s
HRRZM	Half Right to Right, Zeros, Memory	552	
			2.47 (2.58) μ s
HRRZS	Half Right to Right, Zeros, Self	553	
			2.76 (2.87) μ s

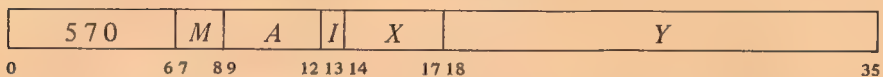
HRRZI loads the word 0, *E* into AC. If *A* is zero, HRRZS merely clears the left half of location *E*.

§2.1

HRRO Half Word Right to Right, Ones

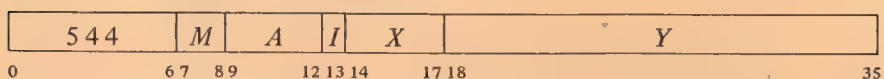
Move the right half of the source word specified by *M* to the right half of the specified destination, and set the destination left half to all 1s. The source is unaffected, the original contents of the destination are lost.

HRRO	Half Right to Right, Ones	560	2.21 (2.43) μ s
HRROI	Half Right to Right, Ones, Immediate	561	1.36 (1.47) μ s
HRROM	Half Right to Right, Ones, Memory	562	2.47 (2.58) μ s
HRROS	Half Right to Right, Ones, Self	563	2.76 (2.87) μ s

HRRE Half Word Right to Right, Extend

Move the right half of the source word specified by *M* to the right half of the specified destination, and make all bits in the destination left half equal to bit 18 of the source. The source is unaffected, the original contents of the destination are lost.

HRRE	Half Right to Right, Extend	570	2.21 (2.43) μ s
HRREI	Half Right to Right, Extend, Immediate	571	1.36 (1.47) μ s
HRREM	Half Right to Right, Extend, Memory	572	2.47 (2.58) μ s
HRRES	Half Right to Right, Extend, Self	573	2.76 (2.87) μ s

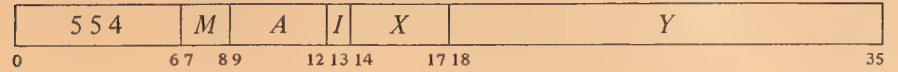
HLR Half Word Left to Right

Move the left half of the source word specified by *M* to the right half of the specified destination. The source and the destination left half are unaffected; the original contents of the destination right half are lost.

HLR	Half Left to Right	544	2.70 (2.92) μ s
HLRI	Half Left to Right Immediate	545	1.85 (1.96) μ s

HLRI merely clears AC right.

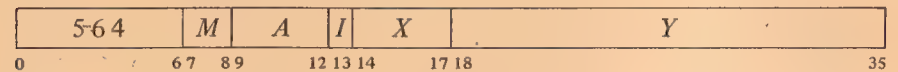
HLRM	Half Left to Right Memory	546	2.90 (3.01) μ s
HLRS	Half Left to Right Self	547	2.76 (2.87) μ s

HLRZ Half Word Left to Right, Zeros

Move the left half of the source word specified by *M* to the right half of the specified destination, and clear the destination left half. The source is unaffected, the original contents of the destination are lost.

HLRZ	Half Left to Right, Zeros	554	2.21 (2.43) μ s
HLRZI	Half Left to Right, Zeros, Immediate	555	1.36 (1.47) μ s
HLRZM	Half Left to Right, Zeros, Memory	556	2.47 (2.58) μ s
HLRZS	Half Left to Right, Zeros, Self	557	2.76 (2.87) μ s

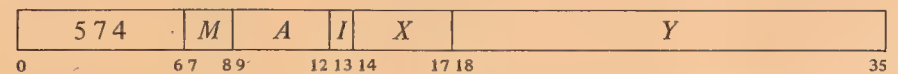
HLRZI merely clears AC and is thus equivalent to HLLZI.

HLRO Half Word Left to Right, Ones

Move the left half of the source word specified by *M* to the right half of the specified destination, and set the destination left half to all 1s. The source is unaffected, the original contents of the destination are lost.

HLRO	Half Left to Right, Ones	564	2.21 (2.43) μ s
HLROI	Half Left to Right, Ones, Immediate	565	1.36 (1.47) μ s
HLROM	Half Left to Right, Ones, Memory	566	2.47 (2.58) μ s
HLROS	Half Left to Right, Ones, Self	567	2.76 (2.87) μ s

HLROI sets AC to all 1s in the left half, all 0s in the right.

HLRE Half Word Left to Right, Extend

Move the left half of the source word specified by *M* to the right half of the specified destination, and make all bits in the destination left half equal to

bit 0 of the source. The source is unaffected, the original contents of the destination are lost.

HLRE	Half Left to Right, Extend	574 2.21 (2.43) μ s
HLREI	Half Left to Right, Extend, Immediate	575 1.36 (1.47) μ s
HLREM	Half Left to Right, Extend, Memory	576 2.47 (2.58) μ s
HLRES	Half Left to Right, Extend, Self	577 2.76 (2.87) μ s

HLREI is equivalent to HLRZI (it merely clears AC).

EXAMPLES. The half word transmission instructions are very useful for handling addresses, and they provide a convenient means of setting up an accumulator whose right half is to be used for indexing while a control count is kept in the left half. *Eg* this pair of instructions loads the 18-bit numbers M and N into the left and right halves respectively of an accumulator that is addressed symbolically as XR.

```
HRLZI XR,M
HRLRI XR,N
```

Of course the source program must somewhere define the value of the symbol XR as an octal number between 1 and 17.

Suppose that at some point we wish to use the two halves of XR independently as operands (taken as 18-bit positive numbers) for computations. We can begin by moving XR left to the right half of another accumulator AC and leaving the contents of XR right alone in XR.

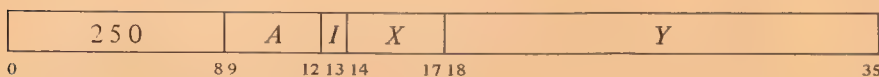
```
HLRZM XR,AC
HLLI XR, ;Clear XR left
```

It is not necessary to clear the other half of XR when loading the first half word. But any instruction that modifies the other half is faster than the corresponding instruction that does not, as the latter must fetch the destination word in order to save half of it. (The difference does not apply to self mode, for here the source and destination are the same.)

2.2 FULL WORD DATA TRANSMISSION

These are the instructions whose basic purpose is to move one or more full words of data from one place to another, usually from an accumulator to a memory location or vice versa. In a few cases instructions may perform minor arithmetic operations, such as forming the negative or the magnitude of the word being processed.

EXCH Exchange 2.90 (3.01) μ s



Keeping instructions and operands in different memories saves .20 (.09) μ s.

Move the contents of location E to AC and move AC to location E .

The time depends on the number and type of transfers. Assuming at least one word is moved a BLT takes .97 (1.08) μ s plus 2.26 (2.48) μ s per transfer from fast memory to core and 2.61 (2.83) μ s per transfer from core to fast memory or from one core location to another.

BLT Block Transfer



Beginning at the location addressed by AC left, move words to another area of memory beginning at the location addressed by AC right. Continue until a word is moved to location *E*. The total number of words in the block is thus $E - AC_R + 1$.

CAUTION

Priority interrupts are allowed during the execution of this instruction, following the processing of each word. If an interrupt occurs, the BLT stores the source and destination addresses for the next word in AC, so when the processor restarts upon the return to the interrupted program, it actually resumes at the correct point within the BLT. Therefore, unless the interrupt system is inactive, *A* and *X* must not address the same register as this would produce a different effective address calculation upon resumption should an interrupt occur; and the program must not attempt to load an accumulator addressed either by *A* or *X* unless it is the final location being loaded. Furthermore, the program cannot assume that AC is the same after the BLT as it was before.

EXAMPLES. This pair of instructions loads the accumulators from memory locations 2000–2017.

```
HRLZI 17,2000 ;Put 2000 000000 in AC 17
BLT   17,17
```

But to transfer the block in the opposite direction requires that one accumulator first be made available to the BLT:

```
MOVEM 17,2017 ;Move AC 17 to 2017 in memory
MOVEI 17,2000 ;Move the number 2000 to AC 17
BLT   17,2016
```

If at the time the accumulators were loaded the program had placed in location 2017 the control word necessary for storing them back in the same block (2000), the three instructions above could be replaced by

```
EXCH 17,2017
BLT  17,2016
```

Move Instructions

Each of these instructions moves a single word, which may be changed in the process (eg its two halves may be swapped). There are four instructions,

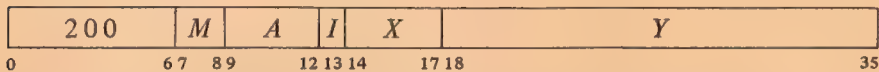
§2.2

each with four modes that determine the source and destination of the word moved.

Mode	Suffix	Source	Destination
Basic		<i>E</i>	AC
Immediate	I	The word 0, <i>E</i>	AC
Memory	M	AC	<i>E</i>
Self	S	<i>E</i>	<i>E</i> , but also AC if <i>A</i> is nonzero

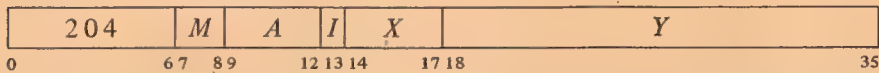
Keeping instructions and operands in different memories saves .47 (.36) μ s in memory mode, .20 (.09) μ s in self mode.

When *E* addresses a fast memory location, a move instruction takes .34 μ s less in basic mode, .46 (.35) μ s less in memory mode, .54 (.43) μ s less in self mode.

MOVE Move

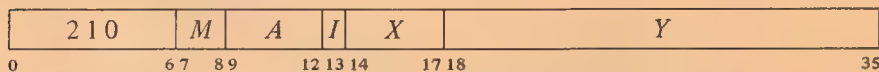
Move one word from the source to the destination specified by *M*. The source is unaffected, the original contents of the destination are lost.

MOVE	Move	200	2.21 (2.43) μ s	▲	MOVEI loads the word 0, <i>E</i> into AC and is thus equivalent to HRRZI. If <i>A</i> is zero, MOVES is a no-op; otherwise it is equivalent to MOVE.
MOVEI	Move Immediate	201	1.36 (1.47) μ s		
MOVEM	Move to Memory	202	2.47 (2.58) μ s	▲	
MOVES	Move to Self	203	2.76 (2.87) μ s		

MOVS Move Swapped

Interchange the left and right halves of the word from the source specified by *M* and move it to the specified destination. The source is unaffected, the original contents of the destination are lost.

MOVS	Move Swapped	204	2.21 (2.43) μ s	▲	Swapping halves in immediate mode loads the word <i>E</i> , 0 into AC. MOVSI is thus equivalent to HRLZI.
MOVSI	Move Swapped Immediate	205	1.36 (1.47) μ s		
MOVSM	Move Swapped to Memory	206	2.47 (2.58) μ s	▲	
MOVSS	Move Swapped to Self	207	2.76 (2.87) μ s		

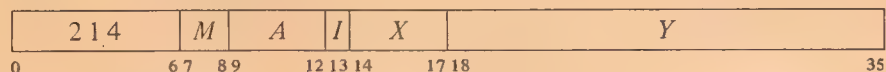
MOVN Move Negative

Negate the word from the source specified by *M* and move it to the specified destination. If the source word is fixed point -2^{35} (400000 000000) set the

Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) If the source word is zero, set Carry 0 and Carry 1. The source is unaffected, the original contents of the destination are lost.

MOVNI loads AC with the negative of the word <i>O, E</i> and can set no flags.	▲ MOVN	Move Negative	210	2.39 (2.61) μ s
	MOVNI	Move Negative Immediate	211	1.54 (1.65) μ s
	▲ MOVNM	Move Negative to Memory	212	2.65 (2.76) μ s
	MOVNS	Move Negative to Self	213	2.94 (3.05) μ s

MOVMM Move Magnitude



Take the magnitude of the word contained in the source specified by *M* and move it to the specified destination. If the source word is fixed point -2^{35} (400000 000000) set the Overflow and Carry 1 flags. (Negating the equivalent floating point -1×2^{127} sets the flags, but this is not a normalized number.) The source is unaffected, the original contents of the destination are lost.

The word <i>O, E</i> is equivalent to its magnitude, so MOVMI is equivalent to MOVEI.	▲ MOVMM	Move Magnitude	214	2.39 (2.61) μ s
	MOVMI	Move Magnitude Immediate	215	1.54 (1.65) μ s
	▲ MOVMM	Move Magnitude to Memory	216	2.65 (2.76) μ s
	MOVMS	Move Magnitude to Self	217	2.94 (3.05) μ s

An example at the end of the preceding section demonstrates the use of a pair of immediate-mode half word transfers to load an address and a control count into an accumulator. The same result can be attained by a single move instruction. This saves time but still requires two locations. *Eg* if the number 200 001400 is stored in location *M*, the instruction

MOVE AC, M

loads 200 into AC left and 1400 into AC right. If the same word, or its negative, or with its halves swapped, must be loaded on several occasions, then both time and space can be saved as each transfer requires only a single move instruction that references *M*.

Pushdown List

These two instructions insert and remove full words in a pushdown list. The address of the top item in the list is kept in the right half of a pointer in AC, and the program can keep a control count in the left half. There are also

two subroutine-calling instructions that utilize a pushdown list of jump addresses [§2.9].

PUSH Push Down 3.85 (4.07) μ s



Add $1\,000\,001_8$ to AC to increment both halves by one, then move the contents of location *E* to the location now addressed by AC right. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag. The contents of *E* are unaffected, the original contents of the location added to the list are lost.

Keeping instructions and the pushdown list in different memories saves .47 (.36) μ s.

When the word added to the list is from fast memory, PUSH takes .34 μ s less than the time given.

POP Pop Up 3.93 (4.15) μ s



Move the contents of the location addressed by AC right to location *E*, then subtract $1\,000\,001_8$ from AC to decrement both halves by one. If the subtraction causes the count in AC left to reach -1 , set the Pushdown Overflow flag. The original contents of *E* are lost.

When the word taken from the list is placed in fast memory, POP takes .46 (.35) μ s less than the time given.

Because of the order in which the operands are stored, the instruction POP AC,AC would load the contents of the location addressed by AC right into AC on top of the pushdown count, destroying it.

The incrementing and decrementing of both halves of AC simultaneously is effected by adding and subtracting $1\,000\,001_8$. Hence a count of -2 in AC left is increased to zero if $2^{18} - 1$ is incremented in AC right, and conversely, 1 in AC left is decreased to -1 if zero is decremented in AC right.

A pushdown list is simply a set of consecutive memory locations from which words are read in the order opposite that in which they are written. In more general terms, it is any list in which the only item that can be removed at any given time is the last item in the list. This is usually referred to as "first in, last out" or "last in, first out". Suppose locations *a*, *b*, *c*, ... are set aside for a pushdown list. We can deposit data in *a*, *b*, *c*, *d*, then read *d*, then write in *d* and *e*, then read *e*, *d*, *c*, etc.

Note that by using the Pushdown Overflow flag and a control count in AC left, the programmer can set a limit to the size of the list by starting the count negative, or he can prevent the program from extracting more words than there are in the list by starting the count at zero, but he cannot do both at once.

Pushdown storage is very convenient for a program that can use data stored in this manner as the pointer is initialized only once and only one accumulator is required for the most complex pushdown operations. To initialize a pointer P for a list to be kept in a block of memory beginning at $BLIST$ and to contain at most N items, the following suffices.

```
MOVSI  P,-N
HRRRI  P,BLIST-1
```

Of course the programmer must define $BLIST$ elsewhere and set aside locations $BLIST$ to $BLIST + N - 1$. Using $MACRO$ to full advantage one could instead give

```
MOVE   P,[IOWD N,BLIST]
```

where the pseudoinstruction

```
IOWD J,K
```

is replaced by a word containing $-J$ in the left half and $K - 1$ in the right. Elsewhere there would appear

```
BLIST:  BLOCK N
```

which defines $BLIST$ as the current contents of the location counter and sets aside the N locations beginning at that point.

In the PDP-10 the pushdown list is kept in a random access core memory, so the restrictions on order of entry and removal of items actually apply only to the standard addressing by the pointer in pushdown instructions — other addressing methods can reference any item at any time. The most convenient way to do this is to use the right half of the pointer as an index register. To move the last entry to accumulator AC we need simply give

```
MOVE   AC,(P)
```

Of course this does not shorten the list — the word moved remains the last item in it.

One usually regards an index register as supplying an additive factor for a basic address contained in an instruction word, but the index register can supply the basic address and the instruction the additive factor. Thus we can retrieve the next to last item by giving

```
MOVE   AC,-1(P)
```

and so forth. Similarly

```
PUSH   P,-3(P)
```

adds the third to last item to the end of the list;

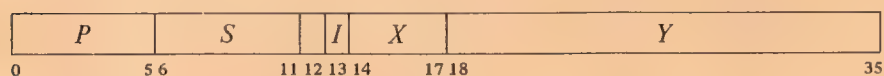
```
POP    P,-2(P)
```

removes the last item and inserts it in place of the next to last item in the shortened list.

2.3 BYTE MANIPULATION

This set of five instructions allows the programmer to pack or unpack bytes of any length anywhere within a word. Movement of a byte is always between AC and a memory location: a deposit instruction takes a byte from the right end of AC and inserts it at any desired position in the memory location; a load instruction takes a byte from any position in the memory location and places it right-justified in AC.

The byte manipulation instructions have the standard memory reference format, but the effective address E is used to retrieve a pointer, which is used in turn to locate the byte or the place that will receive it. The pointer has the format



where S is the size of the byte as a number of bits, and P is its position as the number of bits remaining at the right of the byte in the word (eg if P is 3 the rightmost bit of the byte is bit 32 of the word). The rest of the pointer is interpreted in the same way as in an instruction: I , X and Y are used to calculate the address of the location that is the source or destination of the byte. Thus the pointer aims at a word whose format is



where the shaded area is the byte.

To facilitate processing a series of bytes, several of the byte instructions increment the pointer, ie modify it so that it points to the next byte position in a set of memory locations. Bytes are processed from left to right in a word, so incrementing merely replaces the current value of P by $P - S$, unless there is insufficient space in the present location for another byte of the specified size ($P - S < 0$). In this case Y is increased by one to point to the next consecutive location, and P is set to $36 - S$ to point to the first byte at the left in the new location.

CAUTION

Do not allow Y to reach maximum value. The whole pointer is incremented, so if Y is $2^{18} - 1$ it becomes zero and X is also incremented. The address calculation for the pointer uses the original X , but if a priority interrupt should occur before the calculation is complete, the incremented X is used when the instruction is repeated.

Among these five instructions one simply increments the pointer, the others load or deposit a byte with or without incrementing. Brackets enclose the additional time required when incrementing overflows the word boundary.

2-16

Keeping the pointer in fast memory saves .34 μ s. Taking bytes from a fast memory location saves another .34 μ s.

Keeping the pointer in fast memory saves .34 μ s. Keeping instructions and the packing area in different memories saves .20 (.09) μ s. Packing bytes in fast memory saves .54 (.43) μ s.

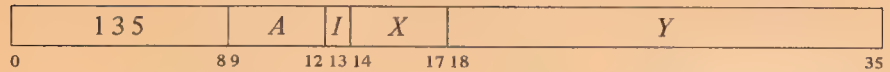
Keeping the pointer in fast memory saves .54 (.43) μ s; keeping it in a different memory from the instruction saves .20 (.09) μ s.

The *A* portion of this instruction is ignored.

Keeping the pointer in fast memory saves .34 μ s. Taking bytes from a fast memory location saves another .34 μ s.

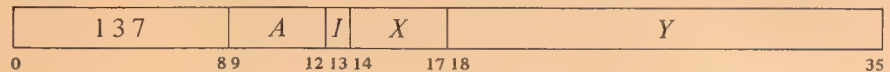
Keeping the pointer in fast memory saves .34 μ s. Keeping instructions and the packing area in different memories saves .20 (.09) μ s. Packing bytes in fast memory saves .54 (.43) μ s.

LDB **Load Byte** $4.02 (4.35) + .15(P + S) [+ .26] \mu$ s



Retrieve a byte of *S* bits from the location and position specified by the pointer contained in location *E*, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected, the original contents of AC are lost.

DPB **Deposit Byte** $4.87 (5.20) + .15(P + S) [+ .26] \mu$ s



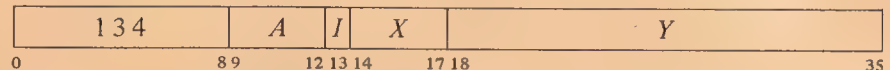
Deposit the right *S* bits of AC into the location and position specified by the pointer contained in location *E*. The original contents of the bits that receive the byte are lost, AC and the remaining bits of the deposit location are unaffected.

IBP **Increment Byte Pointer** $2.87 (2.98) [+ .26] \mu$ s



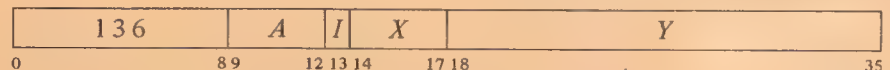
Increment the byte pointer in location *E* as explained above.

ILDB **Increment Pointer and Load Byte** $4.24 (4.57) + .15(P + S) [+ .26] \mu$ s



Increment the byte pointer in location *E* as explained above. Then retrieve a byte of *S* bits from the location and position specified by the newly incremented pointer, load it into the right end of AC, and clear the remaining AC bits. The location containing the byte is unaffected, the original contents of AC are lost.

IDPB **Increment Pointer and Deposit Byte** $5.29 (5.51) + .15(P + S) [+ .26] \mu$ s



Increment the byte pointer in location *E* as explained above. Then deposit

the right S bits of AC into the location and position specified by the newly incremented pointer. The original contents of the bits that receive the byte are lost, AC and the remaining bits of the deposit location are unaffected.

Note that in the pair of instructions that both increment the pointer and process a byte, it is the *modified* pointer that determines the byte location and position. Hence to unpack bytes from a block of memory, the program should set up the pointer to point to a byte just *before* the first desired, and then load them with a loop containing an ILDB. If the first byte is at the left end of a word, this is most easily done by initializing the pointer with a P of 36 (44_8). Incrementing then replaces the 36 with $36 - S$ to point to the first byte. At any time that the program might inspect the pointer during execution of a series of ILDBs or IDPBs, it points to the last byte processed (this may not be true when the pointer is tested from an interrupt routine [§2.13]).

Special Considerations. If S is greater than P and also greater than 36, incrementing produces a new P equal to $100 - S$ rather than $36 - S$. For $S > 36$ the byte is at most the entire word; for $P \geq 36$ no byte is processed (loading merely clears AC). If both P and S are less than 36 but $P + S > 36$, a byte of size $36 - P$ is loaded from position P , or the right $36 - P$ bits of the byte are deposited in position P .

2.4 LOGIC

For logical operations the PDP-10 has instructions for shifting and rotating as well as for performing the complete set of sixteen Boolean functions of two variables (including those in which the result depends on only one or neither variable). The Boolean functions operate bitwise on full words, so each instruction actually performs thirty-six logical operations simultaneously. Thus in the AND function of two words, each bit of the result is the AND of the corresponding bits of the operands. The table on page 2-23 lists the bit configurations that result from the various operand configurations for all instructions.

Each Boolean instruction has four modes that determine the source of the non-AC operand, if any, and the destination of the result.

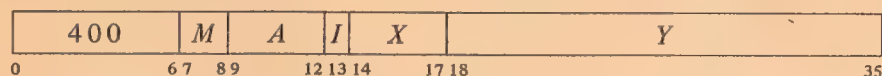
Mode	Suffix	Source of non-AC operand	Destination of result
Basic		E	AC
Immediate	I	The word 0, E	AC
Memory	M	E	E
Both	B	E	AC and E

Keeping instructions and operands in different memories saves .47 (.36) μ s in memory and both modes in the first four of these instructions (those that have no operand or only an AC operand), .20 (.09) μ s in memory and both modes in the remaining twelve (those that have a memory or immediate operand).

A Boolean instruction in which *E* addresses a fast memory location takes .46 (.35) μ s less in memory or both mode if it has no operand or only an AC operand. If it has a memory operand, it takes .34 μ s less in basic mode, .54 (.43) μ s less in memory or both mode.

For an instruction without an operand (one that merely clears a location or sets it to all 1s) the modes differ only in the destination of the result, so basic and immediate modes are equivalent. The same is true also of an instruction that uses only an AC operand. When specified by the mode, the result goes to the accumulator addressed by *A*, even when there is no AC operand.

SETZ Set to Zeros



Change the contents of the destination specified by *M* to all 0s.

SETZ and SETZI are equivalent (both merely clear AC). MACRO also recognizes CLEAR, CLEARI, CLEARM and CLEARB as equivalent to the set-to-zeros mnemonics.

SETZ	Set to Zeros	400	1.36 (1.47) μ s
SETZI	Set to Zeros Immediate	401	1.36 (1.47) μ s
SETZM	Set to Zeros Memory	402	2.33 (2.44) μ s
SETZB	Set to Zeros Both	403	2.33 (2.44) μ s

SETO Set to Ones

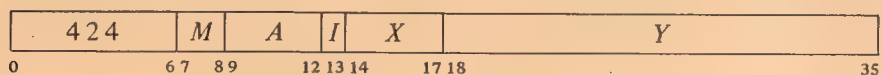


Change the contents of the destination specified by *M* to all 1s.

SETO and SETOI are equivalent.

SETO	Set to Ones	474	1.36 (1.47) μ s
SETOI	Set to Ones Immediate	475	1.36 (1.47) μ s
SETOM	Set to Ones Memory	476	2.33 (2.44) μ s
SETOB	Set to Ones Both	477	2.33 (2.44) μ s

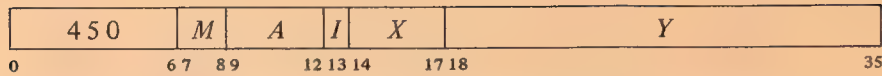
SETA Set to AC



Make the contents of the destination specified by *M* equal to AC.

SETA and SETAI are no-ops. SETAM and SETAB are both equivalent to MOVEM (all move AC to location *E*).

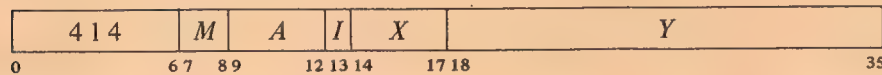
SETA	Set to AC	424	1.50 (1.61) μ s
SETAI	Set to AC Immediate	425	1.50 (1.61) μ s
SETAM	Set to AC Memory	426	2.47 (2.58) μ s
SETAB	Set to AC Both	427	2.47 (2.58) μ s

SETCA Set to Complement of AC

Change the contents of the destination specified by *M* to the complement of AC.

SETCA	Set to Complement of AC	450	
		1.50 (1.61) μ s	
SETCAI	Set to Complement of AC Immediate	451	
		1.50 (1.61) μ s	
SETCAM	Set to Complement of AC Memory	452	
		2.47 (2.58) μ s	
SETCAB	Set to Complement of AC Both	453	
		2.47 (2.58) μ s	

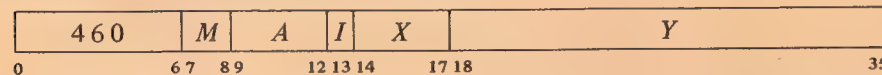
SETCA and SETCAI are equivalent (both complement AC).

SETM Set to Memory

Make the contents of the destination specified by *M* equal to the specified operand.

SETM	Set to Memory	414	2.21 (2.43) μ s
SETMI	Set to Memory Immediate	415	1.36 (1.47) μ s
SETMM	Set to Memory Memory	416	2.76 (2.87) μ s
SETMB	Set to Memory Both	417	2.76 (2.87) μ s

SETM and SETMB are equivalent to MOVE. SETMI moves the word 0, *E* to AC and is thus equivalent to MOVEI. SETMM is a no-op that references memory.

SETCM Set to Complement of Memory

Change the contents of the destination specified by *M* to the complement of the specified operand.

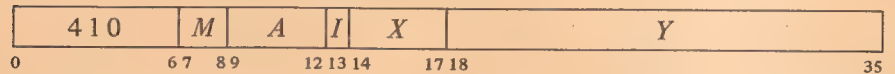
SETCM	Set to Complement of Memory	460	2.21 (2.43) μ s
SETCMI	Set to Complement of Memory Immediate	461	1.36 (1.47) μ s
SETCMM	Set to Complement of Memory Memory	462	2.76 (2.87) μ s
SETCMB	Set to Complement of Memory Both	463	2.76 (2.87) μ s

SETCMI moves the complement of the word 0, *E* to AC. SETCMM complements location *E*.

AND And with AC

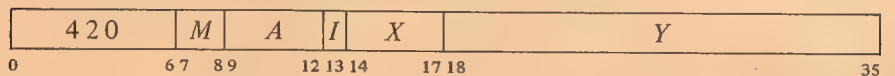
Change the contents of the destination specified by *M* to the AND function of the specified operand and AC.

AND	And	404	2.35 (2.57) μ s
ANDI	And Immediate	405	1.50 (1.61) μ s
ANDM	And to Memory	406	2.90 (3.01) μ s
ANDB	And to Both	407	2.90 (3.01) μ s

ANDCA And with Complement of AC

Change the contents of the destination specified by *M* to the AND function of the specified operand and the complement of AC.

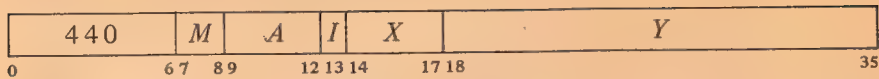
ANDCA	And with Complement of AC	410	2.70 (2.92) μ s
ANDCAI	And with Complement of AC Immediate	411	1.85 (1.96) μ s
ANDCAM	And with Complement of AC to Memory	412	3.52 (3.63) μ s
ANDCAB	And with Complement of AC to Both	413	3.52 (3.63) μ s

ANDCM And Complement of Memory with AC

Change the contents of the destination specified by *M* to the AND function of the complement of the specified operand and AC.

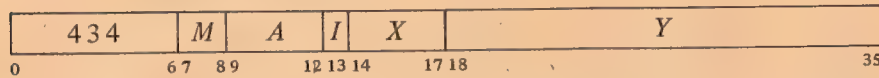
ANDCM	And Complement of Memory	420	2.35 (2.57) μ s
ANDCMI	And Complement of Memory Immediate	421	1.50 (1.61) μ s
ANDCMM	And Complement of Memory to Memory	422	2.90 (3.01) μ s
ANDCMB	And Complement of Memory to Both	423	2.90 (3.01) μ s

§2.4

ANDCB And Complements of Both

Change the contents of the destination specified by *M* to the AND function of the complements of both the specified operand and AC. The result is the NOR function of the operands.

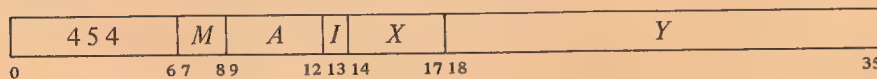
ANDCB	And Complements of Both	440	2.70 (2.92) μ s
ANDCBI	And Complements of Both Immediate	441	1.85 (1.96) μ s
ANDCBM	And Complements of Both to Memory	442	3.52 (3.63) μ s
ANDCBB	And Complements of Both to Both	443	3.52 (3.63) μ s

IOR Inclusive Or with AC

Change the contents of the destination specified by *M* to the inclusive OR function of the specified operand and AC.

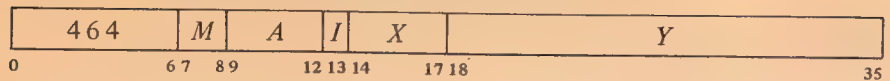
IOR	Inclusive Or	434	2.35 (2.57) μ s
IORI	Inclusive Or Immediate	435	1.50 (1.61) μ s
IORM	Inclusive Or to Memory	436	2.90 (3.01) μ s
IORB	Inclusive Or to Both	437	2.90 (3.01) μ s

MACRO also recognizes OR, ORI, ORM and ORB as equivalent to the inclusive OR mnemonics.

ORCA Inclusive Or with Complement of AC

Change the contents of the destination specified by *M* to the inclusive OR function of the specified operand and the complement of AC.

ORCA	Or with Complement of AC	454	2.70 (2.92) μ s
ORCAI	Or with Complement of AC Immediate	455	1.85 (1.96) μ s
ORCAM	Or with Complement of AC to Memory	456	3.52 (3.63) μ s
ORCAB	Or with Complement of AC to Both	457	3.52 (3.63) μ s

ORCM Inclusive Or Complement of Memory with AC

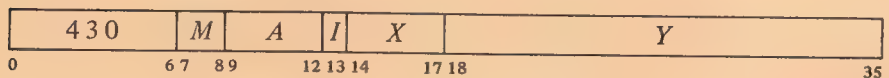
Change the contents of the destination specified by *M* to the inclusive OR function of the complement of the specified operand and AC.

ORCM	Or Complement of Memory	464
		2.35 (2.57) μ s
ORCMI	Or Complement of Memory Immediate	465
		1.50 (1.61) μ s
ORCMM	Or Complement of Memory to Memory	466
		2.90 (3.01) μ s
ORCMB	Or Complement of Memory to Both	467
		2.90 (3.01) μ s

ORCB Inclusive Or Complements of Both

Change the contents of the destination specified by *M* to the inclusive OR function of the complements of both the specified operand and AC. The result is the NAND function of the operands.

ORCB	Or Complements of Both	470
		2.70 (2.92) μ s
ORCBI	Or Complements of Both Immediate	471
		1.85 (1.96) μ s
ORCBM	Or Complements of Both to Memory	472
		3.52 (3.63) μ s
ORCBB	Or Complements of Both to Both	473
		3.52 (3.63) μ s

XOR Exclusive Or with AC

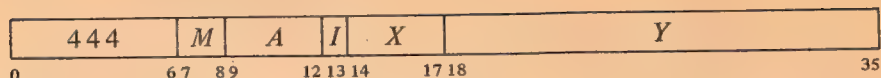
Change the contents of the destination specified by *M* to the exclusive OR function of the specified operand and AC.

XOR	Exclusive Or	430	2.35 (2.57) μ s
XORI	Exclusive Or Immediate	431	1.50 (1.61) μ s
XORM	Exclusive Or to Memory	432	2.90 (3.01) μ s
XORB	Exclusive Or to Both	433	2.90 (3.01) μ s

The original contents of the destination can be recovered except in XORB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the exclusive OR of the remaining operand and the result.

§2.4

EQV Equivalence with AC



Change the contents of the destination specified by *M* to the complement of the exclusive OR function of the specified operand and AC (the result has 1s wherever the corresponding bits of the operands are the same).

EQV	Equivalence	444	2.35 (2.57) μ s
EQVI	Equivalence Immediate	445	1.50 (1.61) μ s
EQVM	Equivalence to Memory	446	2.90 (3.01) μ s
EQVB	Equivalence to Both	447	2.90 (3.01) μ s

The original contents of the destination can be recovered except in EQVB, where both operands are replaced by the result. In the other three modes the replaced operand is restored by repeating the instruction in the same mode, *ie* by taking the equivalence function of the remaining operand and the result.

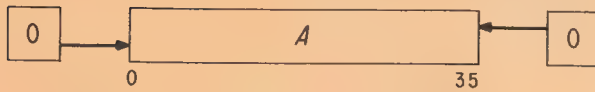
For the four possible bit configurations of the two operands, the above sixteen instructions produce the following results. In each case the result as listed is equal to bits 3-6 of the instruction word.

	<i>AC</i>	0	1	0	1
<i>Mode Specified Operand</i>					
SETZ		0	0	0	0
AND		0	0	0	1
ANDCA		0	0	1	0
SETM		0	0	1	1
ANDCM		0	1	0	0
SETA		0	1	0	1
XOR		0	1	1	0
IOR		0	1	1	1
ANDCB		1	0	0	0
EQV		1	0	0	1
SETCA		1	0	1	0
ORCA		1	0	1	1
SETCM		1	1	0	0
ORCM		1	1	0	1
ORCB		1	1	1	0
SETO		1	1	1	1

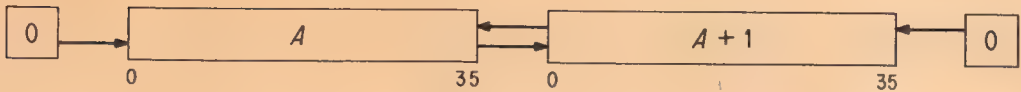
Shift and Rotate

The remaining logical instructions shift or rotate right or left the contents of AC or the contents of two accumulators, A and $A+1$ (mod 20_8), concatenated into a 72-bit register with A on the left. The illustration below shows the movement of information these instructions produce in the accu-

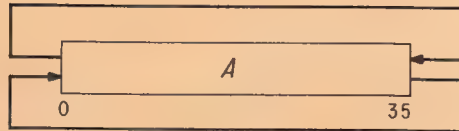
LSH



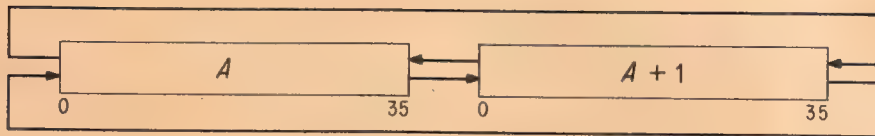
LSHC



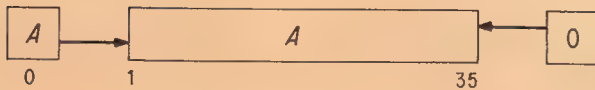
ROT



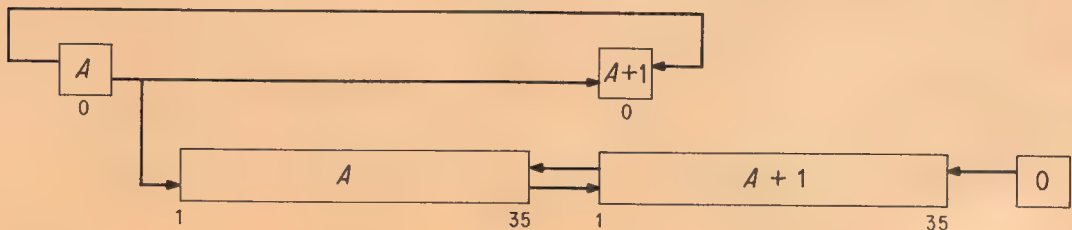
ROTC



ASH



ASHC



ACCUMULATOR BIT FLOW IN SHIFT AND ROTATE INSTRUCTIONS

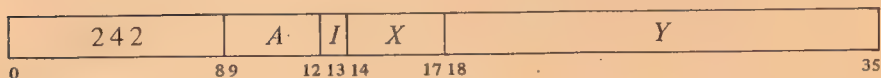
§2.4

mulators. In a (logical) shift the contents of a register are moved bit-to-bit with 0s brought in at the end being vacated; information shifted out at the other end is lost. [For a discussion of arithmetic shifting see §2.5.] In rotation the contents are moved cyclically such that information rotated out at one end is put in at the other.

The number of places moved is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the shift directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right; maximum movement is 255 places.

LSH **Logical Shift**

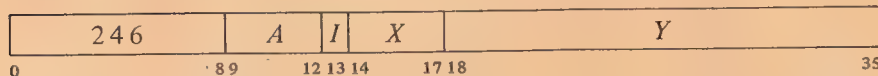
Left: $1.62 (1.73) + .15|E| \mu s$
 Right: $1.46 (1.57) + .15|E| \mu s$



Shift AC the number of places specified by E . If E is positive, shift left bringing 0s into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; data shifted out of bit 35 is lost.

LSHC **Logical Shift Combined**

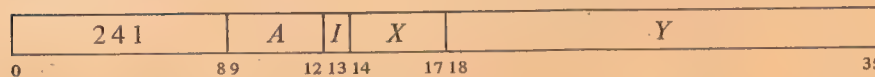
Left: $2.00 (2.11) + .15|E| \mu s$
 Right: $1.84 (1.95) + .15|E| \mu s$



Concatenate accumulators A and $A+1$ with A on the left, and shift the 72-bit combination the number of places specified by E . If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 36 is shifted into bit 35; data shifted out of bit 0 is lost. If E is negative, shift right bringing 0s into bit 0; bit 35 is shifted into bit 36; data shifted out of bit 71 is lost.

ROT **Rotate**

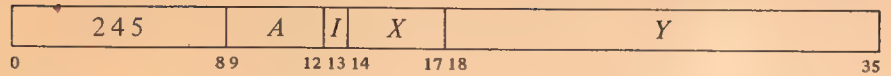
Left: $1.62 (1.73) + .15|E| \mu s$
 Right: $1.46 (1.57) + .15|E| \mu s$



Rotate AC the number of places specified by E . If E is positive, rotate left; bit 0 is rotated into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 0.

ROTC Rotate Combined

Left: 2.00 (2.11) + .15|E| μ s
 Right: 1.84 (1.95) + .15|E| μ s



Concatenate accumulators A and $A+1$ with A on the left, and rotate the 72-bit combination the number of places specified by E . If E is positive, rotate left; bit 0 is rotated into bit 71 (bit 35 of $AC A+1$) and bit 36 into bit 35. If E is negative, rotate right; bit 35 is rotated into bit 36 and bit 71 into bit 0.

2.5 FIXED POINT ARITHMETIC

For fixed point arithmetic the PDP-10 has instructions for arithmetic shifting (which is essentially multiplication by a power of 2) as well as for performing addition, subtraction, multiplication and division of numbers in fixed point format [§1.1]. In such numbers the position of the binary point is arbitrary (the programmer may adopt any point convention). The add and subtract instructions involve only single length numbers, whereas multiply supplies a double length product, and divide uses a double length dividend. The high and low order words respectively of a double length fixed point number are in accumulators A and $A+1$ (mod 20_8), where the magnitude is the 70-bit string in bits 1-35 of the two words and the signs of the two are identical. There are also integer multiply and divide instructions that involve only single length numbers and are especially suited for handling smaller integers, particularly those of eighteen bits or less such as addresses (of course they can be used for small fractions as well provided the programmer keeps track of the binary point). For convenience in the following, all operands are assumed to be integers (binary point at the right).

The processor has four flags, Overflow, Carry 0, Carry 1 and No Divide, that indicate when the magnitude of a number is or would be larger than can be accommodated. Carry 0 and Carry 1 actually detect carries out of bits 0 and 1 in certain instructions that employ fixed point arithmetic operations: the add and subtract instructions treated here, the move instructions that produce the negative or magnitude of the word moved [§2.2], and the arithmetic test instructions that increment or decrement the test word [§2.7]. In these instructions an incorrect result is indicated — and the Overflow flag set — if the carries are different, *ie* if there is a carry into the sign but not out of it, or vice versa. The Overflow flag is also set by No Divide being set, which means the processor has failed to perform a division because the magnitude of the dividend is greater than or equal to that of the divisor, or in integer divide, simply that the divisor is zero. In other overflow cases only Overflow itself is set: these include too large a product in multiplication, and loss of significant bits in left arithmetic shifting.

These flags can be read and controlled by certain program control instructions [§2.9], and Overflow is available as a processor condition (via in-out

Overflow is determined directly from the carries, not from the carry flags, as their states may reflect events in previous instructions.

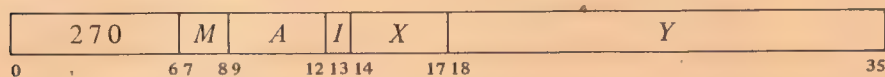
instructions [§2.14]) that can request a priority interrupt if enabled. The conditions detected can only set the flags and the hardware does not clear them, so the program must clear them before an instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected.

All but the shift instructions have four modes that determine the source of the non-AC operand and the destination of the result.

Besides indicating error types, the carry flags facilitate performing multiple precision arithmetic.

<i>Mode</i>	<i>Suffix</i>	<i>Source of non-AC operand</i>	<i>Destination of result</i>
Basic		<i>E</i>	AC
Immediate	I	The word 0, <i>E</i>	AC
Memory	M	<i>E</i>	<i>E</i>
Both	B	<i>E</i>	AC and <i>E</i>

ADD Add



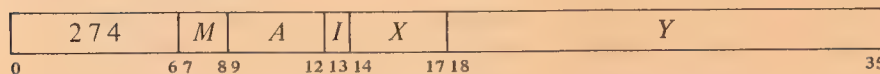
Add the operand specified by *M* to AC and place the result in the specified destination. If the sum is $\geq 2^{35}$ set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the sum less 2^{35} . If the sum is $< -2^{35}$ set Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the sum plus 2^{35} . Set both carry flags if both summands are negative, or their signs differ and their magnitudes are equal or the positive one is the greater in magnitude. ▲

ADD	Add	270	2.53 (2.75) μ s
ADDI	Add Immediate	271	1.68 (1.79) μ s
ADDM	Add to Memory	272	3.08 (3.19) μ s
ADDB	Add to Both	273	3.08 (3.19) μ s

Keeping instructions and operands in different memories saves .20 (.09) μ s in ADDM and ADDB.

When *E* addresses a fast memory location, ADD takes .34 μ s less than the time given, ADDM and ADDB take .54 (.43) μ s less.

SUB Subtract



Subtract the operand specified by *M* from AC and place the result in the specified destination. If the difference is $\geq 2^{35}$ set Overflow and Carry 1; the result stored has a minus sign but a magnitude in positive form equal to the difference less 2^{35} . If the difference is $< -2^{35}$ set Overflow and Carry 0; the result stored has a plus sign but a magnitude in negative form equal to the difference plus 2^{35} . Set both carry flags if the signs of the operands are the same and AC is the greater or the two are equal, or the signs of the operands differ and AC is negative.

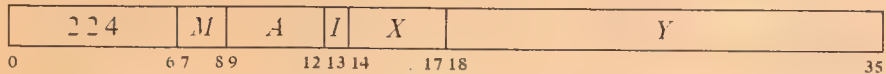
2-28

Keeping instructions and operands in different memories saves .20 (.09) μ s in SUBM and SUBB.

When E addresses a fast memory location, SUB takes .34 μ s less than the time given. SUBM and SUBB take .54 (.43) μ s less.

SUB	Subtract	274	2.53 (2.75) μ s
SUBI	Subtract Immediate	275	1.68 (1.79) μ s
SUBM	Subtract to Memory	276	3.08 (3.19) μ s
SUBB	Subtract to Both	277	3.08 (3.19) μ s

MUL Multiply



Multiply AC by the operand specified by M , and place the high order word of the double length result in the specified destination. If M specifies AC as a destination, place the low order word in accumulator $A+1$. If both operands are -2^{35} set Overflow: the double length result stored is -2^{70} .

Keeping instructions and operands in different memories saves .47 (.36) μ s in MULM. .31 (.20) μ s in MULB.

When E addresses a fast memory location, MUL takes .34 μ s less than the time given. MULM takes .80 (.69) μ s less, and MULB takes .64 (.53) μ s less.

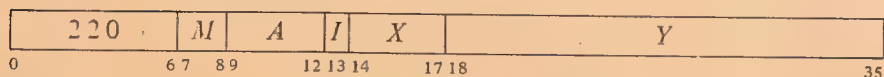
MUL	Multiply	224	10.60 (10.82) μ s
MULI	Multiply Immediate	225	8.58 (8.69) μ s
MULM	Multiply to Memory	226	11.41 (11.63) μ s
MULB	Multiply to Both	227	11.41 (11.63) μ s

Timing. The times given above are average. The algorithm modifies the running sum of partial products at each 1-0 or 0-1 transition scanning from one bit to the next in the multiplier, which is the operand specified by the mode; in other words the number of operations equals the number of pairs of adjacent bits that differ in the multiplier including the sign bit and taking the bit at the right of the LSB as 0 (an LSB of 1 is regarded as a transition). Minimum times with a zero multiplier are

MUL	8.26 (8.48) μ s
MULI	7.41 (7.52) μ s
MULM	9.07 (9.29) μ s
MULB	9.07 (9.29) μ s

These must be increased by .13 μ s for each transition. The programmer can minimize the time by using as the multiplier the operand with fewer transitions.

IMUL Integer Multiply



Multiply AC by the operand specified by M , and place the sign and the 35 low order magnitude bits of the product in the specified destination. Set Overflow if the product is $\geq 2^{35}$ or $< -2^{35}$ (ie if the high order word of the double length product is not null); the high order word is lost.

§2.5

IMUL	Integer Multiply	220	9.59 (9.81) μ s
IMULI	Integer Multiply Immediate	221	8.09 (8.20) μ s
IMULM	Integer Multiply to Memory	222	10.56 (10.78) μ s
IMULB	Integer Multiply to Both	223	10.56 (10.78) μ s

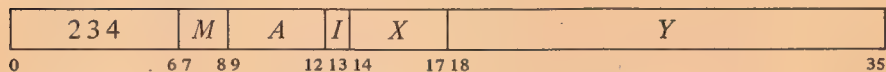
Keeping instructions and operands in different memories saves .47 (.36) μ s in IMULM and IMULB.

When E addresses a fast memory location, IMUL takes .34 μ s less than the time given, IMULM and IMULB take .80 (.69) μ s less.

Timing. The times given above are average. Refer to the description of MUL for the timing effects of the multiplication algorithm. Minimum times with a zero multiplier are

IMUL	8.42 (8.64) μ s
IMULI	7.57 (7.68) μ s
IMULM	9.39 (9.61) μ s
IMULB	9.39 (9.61) μ s

These must be increased by .13 μ s for each transition. The programmer can minimize the time by using as the multiplier the operand with fewer transitions.

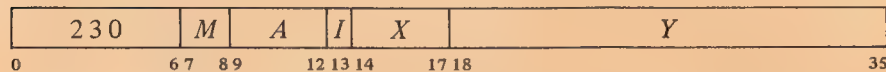
DIV Divide

If the magnitude of the number in AC is greater than or equal to that of the operand specified by M , set Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise divide the double length number contained in accumulators A and $A+1$ by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place the unrounded quotient in the specified destination. If M specifies AC as a destination, place the remainder, with the same sign as the dividend, in accumulator $A+1$.

DIV	Divide	234	16.2 (16.4) μ s
DIVI	Divide Immediate	235	15.4 (15.5) μ s
DIVM	Divide to Memory	236	17.1 (17.3) μ s
DIVB	Divide to Both	237	17.1 (17.3) μ s

Keeping instructions and operands in different memories saves .5 (.4) μ s in DIVM, .3 (.2) μ s in DIVB.

When E addresses a fast memory location, DIV takes .3 μ s less than the time given, DIVM takes .8 (.7) μ s less, and DIVB takes .6 (.5) μ s less.

IDIV Integer Divide

If the operand specified by M is zero, set Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way. Otherwise divide AC by the specified operand, calculating a quotient of 35 magnitude bits including leading zeros. Place

If the division is not performed, only 2.5–3 μ s are required.

the unrounded quotient in the specified destination. If M specifies AC as the destination, place the remainder, with the same sign as the dividend, in accumulator $A+1$.

Keeping instructions and operands in different memories saves .5 (.4) μ s in IDIVM, .3 (.2) μ s in IDIVB.

When E addresses a fast memory location, IDIV takes .3 μ s less than the time given, IDIVM takes .8 (.7) μ s less, and IDIVB takes .6 (.5) μ s less.

If the division is not performed, only 3–3.5 μ s are required.

IDIV	Integer Divide	230	16.5 (16.7) μ s
IDIVI	Integer Divide Immediate	231	15.7 (15.8) μ s
IDIVM	Integer Divide to Memory	232	17.4 (17.6) μ s
IDIVB	Integer Divide to Both	233	17.4 (17.6) μ s

EXAMPLE. The integer multiply and divide instructions are very useful for computations on addresses or character codes, or performing any integral operations in which the result is small enough to be accommodated in a single register.

As an example suppose we wish to determine the parity of the 8-bit character $abcdefgh$, where the letters represent the bits of the character. Assuming the character is right-justified in AC, we first duplicate it twice to the left producing

$abc\ def\ gha\ bcd\ efg\ hab\ cde\ fgh$

where the bits (in positions 12–35) are grouped corresponding to the octal digits in the word. Anding this with

001 001 001 001 001 001 001 001

retains only the least significant bit in each 3-bit set, so we can represent the result by

$cfadgbeh$

where each letter represents an octal digit having the same value (0 or 1) as the bit originally represented by the same letter. Multiplying this by 11111111_8 generates the following partial products:

```

          c f a d g b e h
         c f a d g b e h
        c f a d g b e h
       c f a d g b e h
      c f a d g b e h
     c f a d g b e h
    c f a d g b e h
   c f a d g b e h
  c f a d g b e h
 c f a d g b e h

```

- ▲ Since any digit is at most 1, there can be no carry out of any column with fewer than eight digits unless there is a carry into it. Hence the octal digit produced by summing the center column (the one containing all the bits of the character) is even or odd as the sum of the bits is even or odd. Thus its least significant bit (bit 14 of the low order word in the product) is the parity of the character, 0 if even, 1 if odd.

The above may seem a very complicated procedure to do something trivial, but it is effected by this quite simple sequence (with the character

§2.5

right-justified in AC):

```

IMULI  AC,200401
AND    AC,ONES
IMUL   AC,ONES

```

ONES: 11111111

where the parity is indicated by AC bit 14. Of course, following the IMUL would be a test instruction to check the value of the bit.

Arithmetic Shifting

These two instructions produce an arithmetic shift right or left of the number in AC or the double length number in accumulators A and $A+1$. Shifting is the movement of the contents of a register bit-to-bit. The operation discussed here is similar to logical shifting [see §2.4 and the illustration on page 2-24], but in an arithmetic shift only the magnitude part is shifted — the sign is unaffected. In a double length number the 70-bit string made up of the magnitude parts of the two words is shifted, but the sign of the low order word is made equal to the sign of the high order word.

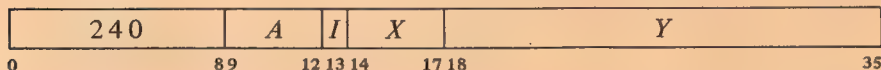
Null bits are brought in at the end being vacated: a left shift brings in 0s at the right, whereas a right shift brings in the equivalent of the sign bit at the left. In either case, information shifted out at the other end is lost. A single shift left is equivalent to multiplying the number by 2 (provided no bit of significance is shifted out); a shift right divides the number by 2.

The number of places shifted is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective shift E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the shift directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating the shift. A positive E produces motion to the left, a negative E to the right; E is thus the power of 2 by which the number is multiplied. Maximum movement is 255 places.

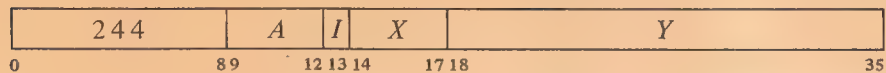
ASH **Arithmetic Shift**

Left: $1.62 (1.73) + .15|E| \mu s$

Right: $1.46 (1.57) + .15|E| \mu s$



Shift AC arithmetically the number of places specified by E . Do not shift bit 0. If E is positive, shift left bringing 0s into bit 35; data shifted out of bit 1 is lost; set Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). If E is negative, shift right bringing 0s into bit 1 if AC is positive, 1s if negative; data shifted out of bit 35 is lost.

ASHC Arithmetic Shift CombinedLeft: $2.00(2.11) + .15|E| \mu s$ Right: $1.84(1.95) + .15|E| \mu s$ 

Concatenate the magnitude portions of accumulators A and $A+1$ with A on the left, and shift the 70-bit combination in bits 1–35 and 37–71 the number of places specified by E . Do not shift AC bit 0, but make bit 0 of AC $A+1$ equal to it if at least one shift occurs (*ie* if E is nonzero). If E is positive, shift left bringing 0s into bit 71 (bit 35 of AC $A+1$); bit 37 (bit 1 of AC $A+1$) is shifted into bit 35; data shifted out of bit 1 is lost; set Overflow if any bit of significance is lost (a 1 in a positive number, a 0 in a negative one). If E is negative, shift right bringing 0s into bit 1 if AC is positive, 1s if negative; bit 35 is shifted into bit 37; data shifted out of bit 71 is lost.

2.6 FLOATING POINT ARITHMETIC

For floating point arithmetic the PDP-10 has instructions for scaling the exponent (which is multiplication of the entire number by a power of 2) and negating double length numbers as well as for performing addition, subtraction, multiplication and division of numbers in floating point format. All instructions treated here interpret all operands as floating point numbers in the format given in §1.1, and generate results in that format. The reader is strongly advised to reread §1.1 if he does not remember the format in detail.

For the four standard arithmetic operations the program can select whether or not the result shall be rounded. Rounding produces the greatest consistent precision using only single length operands. Instructions without rounding have a “long” mode, which supplies a two-word result for greater precision; the other modes save time in one-word operations where rounding is of no significance.

Actually the result is formed in a double length register in addition, subtraction and multiplication, wherein any bits of significance in the low order part supply information for normalization, and then for rounding if requested. Consider addition as an example. Before adding, the processor right shifts the fractional part of the operand with the smaller exponent until its bits correctly match the bits of the other operand in order of magnitude. Thus the smaller operand could disappear entirely, having no effect on the result (“result” shall always be taken to mean the information (one word or two) stored by the instruction, regardless of the number of significant bits it contains or even whether it is the correct answer). Long mode is likely to retain information that would otherwise be lost, but in any given mode the significance of the result depends on the relative values of the operands. Even when both operands contain twenty-seven significant bits, a long addition may store two words that together contain only one significant bit. In division the processor always calculates a one-word quotient that requires no

A subtraction involving two like-signed numbers whose exponents are equal and whose fractions differ only in the LSB gives a result containing only one bit of significance.

normalization if the original operands are normalized. An extra quotient bit is calculated for rounding when requested; long mode retains the remainder.

The processor has four flags, Overflow, Floating Overflow, Floating Underflow and No Divide, that indicate when the exponent is too large or too small to be accommodated or a division cannot be performed because of the relative values of dividend and divisor. Any of these circumstances sets Overflow and Floating Overflow. If only these two are set, the exponent of the answer is too large; if Floating Underflow is also set, the exponent is too small. No Divide being set means the processor failed to perform a division, an event that can be produced only by a zero divisor if all nonzero operands are normalized. These flags can be read and controlled by certain program control instructions [§2.9], and Overflow and Floating Overflow are available as processor conditions (via in-out instructions [§2.14]) that can request a priority interrupt if enabled. The conditions detected can only set the flags and the hardware does not clear them, so the program must clear them before a floating point instruction if they are to give meaningful information about the instruction afterward. However, the program can check the flags following a series of instructions to determine whether the entire series was free of the types of error detected.

The floating point hardware functions at its best if given operands that are either normalized or zero, and except in special situations the hardware normalizes a nonzero result. An operand with a zero fraction and a nonzero exponent can give wild answers in additive operations because of extreme loss of significance; *eg* adding $\frac{1}{2} \times 2^2$ and 0×2^{69} gives a zero result, as the first operand (having a smaller exponent) looks smaller to the processor and is shifted to oblivion. A number with a 1 in bit 0 and 0s in bits 9–35 is not simply an incorrect representation of zero, but rather an unnormalized “fraction” with value -1 . This unnormalized number can produce an incorrect answer in any operation. Use of other unnormalized operands simply causes loss of significant bits, except in division where they can prevent its execution because they can satisfy a no-divide condition that is impossible for normalized numbers.

The processor normalizes the result by shifting the fraction and adjusting the exponent to compensate for the change in value. Each shift and accompanying exponent adjustment thus multiply the number both by 2 and by $\frac{1}{2}$ simultaneously, leaving its value unchanged.

Scaling

One floating point instruction is in a category by itself: it changes the exponent of a number without changing the significance of the fraction. In other words it multiplies the number by a power of 2, and is thus analogous to arithmetic shifting of fixed point numbers except that no information is lost, although the exponent can overflow or underflow. The amount added to the exponent is specified by the result of the effective address calculation taken as a signed number (in twos complement notation) modulo 2^8 in magnitude. In other words the effective scale factor E is the number composed of bit 18 (which is the sign) and bits 28–35 of the calculation result. Hence the programmer may specify the factor directly in the instruction (perhaps indexed) or give an indirect address to be used in calculating it. A positive E increases the exponent, a negative E decreases it; E is thus the power of 2 by which the number is multiplied. The scale factor lies in the range -256 to $+255$.

2-34

N is the number of left shifts needed to normalize the result.

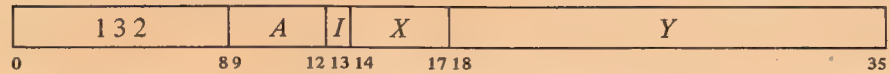
This instruction can be used to float a fixed number with 27 or fewer significant bits. To float an integer contained within AC bits 9–35,

FSC AC,233

inserts the correct exponent to move the binary point from the right end to the left of bit 9 and then normalizes ($233_8 = 155_{10} = 128 + 27$).

FSC

Floating Scale

2.75 (2.86) + .25 N μ s

If the fractional part of AC is zero, clear AC. Otherwise add the scale factor given by E to the exponent part of AC (thus multiplying AC by 2^E), normalize the resulting word bringing 0s into bit positions vacated at the right, and place the result back in AC.

NOTE

A negative E is represented in standard twos complement notation, but the hardware compensates for this when scaling the exponent.

If the exponent after normalization is > 127 , set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128 , set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

Operations with Rounding

In the hardware the rounding operation is actually somewhat more complex than stated here. If the result is negative, the hardware combines rounding with placing the high order word in twos complement form by decreasing its magnitude if the low order part is $< \frac{1}{2}$ LSB. Moreover an extra single-step re-normalization occurs if the rounded word is no longer normalized.

Keeping instructions and operands in different memories saves .47 (.36) μ s in memory and both modes.

When E addresses a fast memory location, a floating point instruction with rounding takes .34 μ s less than the time listed in basic mode, .80 (.69) μ s less in memory or both mode.

There are four instructions that use only one-word operands and store a single-length rounded result. Rounding is away from zero: if the part of the normalized answer being dropped (the low order part of the fraction) is greater than or equal in magnitude to one half the LSB of the part being retained, the magnitude of the latter part is increased by one LSB.

The rounding instructions have four modes that determine the source of the non-AC operand and the destination of the result. These modes are like those of logic and fixed point arithmetic, including an immediate mode that allows the instruction to carry an operand with it.

<i>Mode</i>	<i>Suffix</i>	<i>Source of non-AC operand</i>	<i>Destination of result</i>
Basic		E	AC
▲ Immediate	I	The word $E,0$	AC
Memory	M	E	E
Both	B	E	AC and E

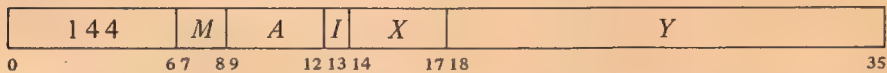
Note however that floating point immediate uses $E,0$ as an operand, not $0, E$. In other words the half word E is interpreted as a sign, an 8-bit exponent, and a 9-bit fraction.

The time required is a function of the number N of left shifts needed for normalization. Brackets enclose the additional time required when rounding actually changes the high order word.

In each of these instructions, the exponent that results from normaliza-

tion and rounding is tested for overflow or underflow. If the exponent is > 127 , set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128 , set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

FADR Floating Add and Round

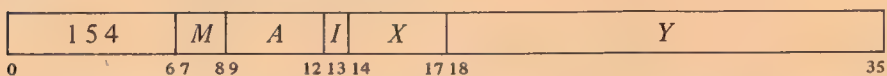


Floating add the operand specified by *M* to AC. If the double length fraction in the sum is zero, clear the specified destination. Otherwise normalize the double length sum bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

FADR	Floating Add and Round	144
	$4.46 (4.68) + .15D + .25N [+ .96]$	μs
FADRI	Floating Add and Round Immediate	145
	$3.70 (3.81) + .15D + .25N [+ .96]$	μs
FADRM	Floating Add and Round to Memory	146
	$5.43 (5.65) + .15D + .25N [+ .96]$	μs
FADRB	Floating Add and Round to Both	147
	$5.43 (5.65) + .15D + .25N [+ .96]$	μs

D is the difference between the operand exponents provided that difference is ≤ 63 . Otherwise $D = 0$.

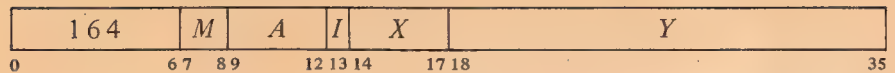
FSBR Floating Subtract and Round



Floating subtract the operand specified by *M* from AC. If the double length fraction in the difference is zero, clear the specified destination. Otherwise normalize the double length difference bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

FSBR	Floating Subtract and Round	154
	$4.64 (4.86) + .15D + .15N [+ .96]$	μs
FSBRI	Floating Subtract and Round Immediate	155
	$3.88 (3.99) + .15D + .15N [+ .96]$	μs
FSBRM	Floating Subtract and Round to Memory	156
	$5.61 (5.83) + .15D + .15N [+ .96]$	μs
FSBRB	Floating Subtract and Round to Both	157
	$5.61 (5.83) + .15D + .15N [+ .96]$	μs

D is the difference between the operand exponents provided that difference is ≤ 63 . Otherwise $D = 0$.

FMPR Floating Multiply and Round

Floating Multiply AC by the operand specified by *M*. If the double length fraction in the product is zero, clear the specified destination. Otherwise normalize the double length product bringing 0s into bit positions vacated at the right, round the high order part, test for exponent overflow or underflow as described above, and place the result in the specified destination.

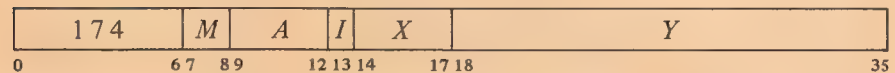
Use of normalized operands requires at most one normalization step for the result. If unnormalized operands are used, all times must be increased by .25*N*.

FMPR	Floating Multiply and Round	164 10.29 (10.51) [+96] μ s
FMPRI	Floating Multiply and Round Immediate	165 8.36 (8.47) [+96] μ s
FMPRM	Floating Multiply and Round to Memory	166 11.26 (11.48) [+96] μ s
FMPRB	Floating Multiply and Round to Both	167 11.26 (11.48) [+96] μ s

Timing. The times given above are average for normalized operands. Refer to the description of MUL [§2.5] for the timing effects of the multiplication algorithm. Minimum times with a zero multiplier are

FMPR	8.47 (8.69) [+96] μ s
FMPRI	7.71 (7.82) [+96] μ s
FMPRM	9.44 (9.66) [+96] μ s
FMPRB	9.44 (9.66) [+96] μ s

These must be increased by .13 μ s for each transition. The programmer can minimize the time by using as the multiplier the operand with fewer transitions.

FDVR Floating Divide and Round

Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in the operand specified by *M*, set Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

If the division can be performed, floating divide AC by the operand specified by *M*, calculating a quotient fraction of 28 bits (this includes an extra bit for rounding). If the fraction is zero, clear the specified destination.

- ▲ Otherwise round the fraction using the extra bit calculated. If the original operands were normalized, the single length quotient will already be normalized; if it is not, normalize it bringing 0s into bit positions vacated at the right. Test for

exponent overflow or underflow as described above, and place the result in the specified destination.

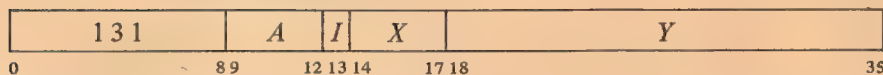
FDVR	Floating Divide and Round	174
		14.1 (14.3) μ s
FDVRI	Floating Divide and Round Immediate	175
		13.3 (13.4) μ s
FDVRM	Floating Divide and Round to Memory	176
		15.1 (15.3) μ s
FDVRB	Floating Divide and Round to Both	177
		15.1 (15.3) μ s

If unnormalized operands are used, all times must be increased by $.25N$. If the division is not performed, only 3.5–4 μ s are required.

Operations without Rounding

Instructions that do not round are faster for processing floating point numbers with fractions containing fewer than 27 significant bits. On the other hand the long mode provides double precision or allows the programmer to use his own method of rounding. Besides the four usual arithmetic operations with normalization, there are two nonnormalizing instructions that facilitate double precision arithmetic [§2.11 gives examples of double precision floating point routines]. These two instructions have no modes.

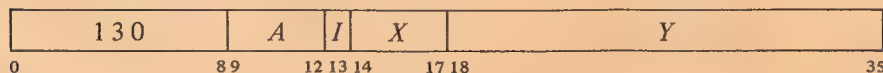
DFN **Double Floating Negate** 3.43 (3.54) μ s



Negate the double length floating point number composed of the contents of AC and location E with AC on the left. Do this by taking the two's complement of the number whose sign is AC bit 0, whose exponent is in AC bits 1–8, and whose fraction is the 54-bit string in bits 9–35 of AC and location E . Place the high order word of the result in AC; place the low order part of the fraction in bits 9–35 of location E without altering the original contents of bits 0–8 of that location.

Usually the double length number is in two adjacent accumulators, and E equals $A+1$. In this case DFN takes \blacktriangle only 2.89 (3.11) μ s.

UFA **Unnormalized Floating Add** 4.62 (4.84) + .15 D μ s



Floating add the contents of location E to AC. If the double length fraction in the sum is zero, clear accumulator $A+1$. Otherwise normalize the sum only if the magnitude of its fractional part is ≥ 1 , and place the high order part of the result in AC $A+1$. The original contents of AC and E are unaffected.

D is the difference between the operand exponents provided that difference is ≤ 63 . Otherwise $D = 0$.

When E addresses a fast memory location, UFA takes .34 μ s less than the time given.

The exponent of the sum is equal to that of the larger summand unless addition of the fractions overflows, in which case it is greater by 1. Exponent overflow can occur only in the latter case.

NOTE

The result is placed in accumulator $A+1$. This is the only arithmetic instruction that stores the result in a second accumulator, leaving the original operands intact.

If the exponent of the sum following the one-step normalization is > 127 , set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one.

The remaining floating point instructions perform the four standard arithmetic operations with normalization but without rounding. All use AC and the contents of location E as operands and have four modes.

	<i>Mode</i>	<i>Suffix</i>	<i>Effect</i>
Keeping instructions and operands in different memories saves .47 (.36) μ s in memory and both modes.	Basic		High order word of result stored in AC.
	Long	L	In addition, subtraction and multiplication, the two-word result (in the double length format described in §1.1) is stored in accumulators A and $A+1$. In division the dividend is the double length word in A and $A+1$; the single length quotient is stored in AC, the remainder in AC $A+1$.
When E addresses a fast memory location, a floating point instruction without rounding takes .34 μ s less than the time listed in basic or long mode, .80 (.69) μ s less in memory or both mode.	Memory	M	High order word of result stored in E .
	Both	B	High order word of result stored in AC and E .

In each of these instructions, the exponent that results from normalization is tested for overflow or underflow. If the exponent is > 127 , set Overflow and Floating Overflow; the result stored has an exponent 256 less than the correct one. If < -128 , set Overflow, Floating Overflow and Floating Underflow; the result stored has an exponent 256 greater than the correct one.

The time required is a function of the number N of left shifts needed for normalization.

FAD Floating Add

140	M	A	I	X	Y
0	67 89	12 13 14	17 18		35

Floating add the contents of location E to AC. If the double length fraction in the sum is zero, clear the destination specified by M , clearing both accu-

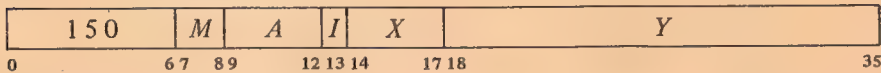
mulators in long mode. Otherwise normalize the double length sum bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the sum is $> 154 (127 + 27)$ or $< -101 (-128 + 27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the sum in bits 1-8, and the low order part of the fraction in bits 9-35.

FAD	Floating Add	140
		$4.46 (4.68) + .15D + .25N \mu s$
FADL	Floating Add Long	141
		$5.31 (5.53) + .15D + .25N \mu s$
FADM	Floating Add to Memory	142
		$5.43 (5.65) + .15D + .25N \mu s$
FADB	Floating Add to Both	143
		$5.43 (5.65) + .15D + .25N \mu s$

D is the difference between the operand exponents provided that difference is ≤ 63 . Otherwise $D = 0$.

FSB Floating Subtract

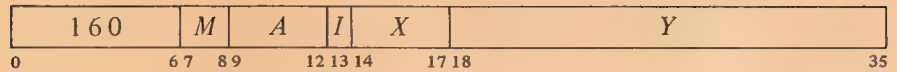


Floating subtract the contents of location E from AC. If the double length fraction in the difference is zero, clear the destination specified by M , clearing both accumulators in long mode. Otherwise normalize the double length difference bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the difference is $> 154 (127 + 27)$ or $< -101 (-128 + 27)$ or the low order half of the fraction is zero, clear AC $A+1$. Otherwise place a low order word for a double length result in $A+1$ by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the difference in bits 1-8, and the low order part of the fraction in bits 9-35.

FSB	Floating Subtract	150
		$4.64 (4.86) + .15D + .25N \mu s$
FSBL	Floating Subtract Long	151
		$5.49 (5.71) + .15D + .25N \mu s$
FSBM	Floating Subtract to Memory	152
		$5.61 (5.83) + .15D + .25N \mu s$
FSBB	Floating Subtract to Both	153
		$5.61 (5.83) + .15D + .25N \mu s$

D is the difference between the operand exponents provided that difference is ≤ 63 . Otherwise $D = 0$.

FMP Floating Multiply

Floating multiply AC by the contents of location *E*. If the double length fraction in the product is zero, clear the destination specified by *M*, clearing both accumulators in long mode. Otherwise normalize the double length product bringing 0s into bit positions vacated at the right, test for exponent overflow or underflow as described above, and place the high order word of the result in the specified destination.

In long mode if the exponent of the product is > 154 ($127 + 27$) or < -101 ($-128 + 27$) or the low order half of the fraction is zero, clear AC *A*+1. Otherwise place a low order word for a double length result in *A*+1 by putting a 0 in bit 0, an exponent in positive form 27 less than the exponent of the product in bits 1–8, and the low order part of the fraction in bits 9–35.

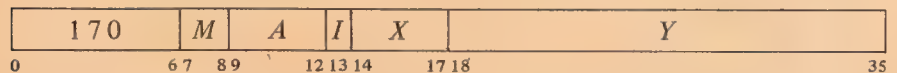
Use of normalized operands requires at most one normalization step for the result. If unnormalized operands are used, all times must be increased by .25*N*.

FMP	Floating Multiply	160	10.29 (10.51) μ s
FMPL	Floating Multiply Long	161	11.14 (11.36) μ s
FMPM	Floating Multiply to Memory	162	11.26 (11.48) μ s
FMPB	Floating Multiply to Both	163	11.26 (11.48) μ s

Timing. The times given above are average for normalized operands. Refer to the description of MUL [§2.5] for the timing effects of the multiplication algorithm. Minimum times with a zero multiplier are

FMP	8.47 (8.69) μ s
FMPL	9.32 (9.54) μ s
FMPM	9.44 (9.66) μ s
FMPB	9.44 (9.66) μ s

These must be increased by .13 μ s for each transition. The programmer can minimize the time by using as the multiplier the operand with fewer transitions.

FDV Floating Divide

Division fails if the divisor is zero, but the no-divide condition can otherwise be satisfied only if at least one operand is unnormalized.

If the magnitude of the fraction in AC is greater than or equal to twice that of the fraction in location *E*, set Overflow, Floating Overflow and No Divide, and go immediately to the next instruction without affecting the original AC or memory operand in any way.

If division can be performed, floating divide the AC operand by the contents of location *E*. In long mode the AC operand (the dividend) is the double length number in accumulators *A* and *A*+1; in other modes it is the single word in AC. Calculate a quotient fraction of 27 bits. If the fraction

§2.7

is zero, clear the destination specified by M , clearing both accumulators in long mode if the double length dividend was zero. A quotient with a non-zero fraction will already be normalized if the original operands were normalized; if it is not, normalize it bringing 0s into bit positions vacated at the right. Test for exponent overflow or underflow as described above, and place the single length quotient part of the result in the specified destination.

In long mode calculate the exponent for the fractional remainder from the division according to the relative magnitudes of the fractions in dividend and divisor: if the dividend was greater than or equal to the divisor, the exponent of the remainder is 26 less than that of the dividend, otherwise it is 27 less. If the remainder exponent is > 127 or < -128 or the fraction is zero, clear AC $A+1$. Otherwise place the floating point remainder (exponent and fraction) with the sign of the dividend in AC $A+1$.

FDV	Floating Divide	170	14.1 (14.3) μ s
FDVL	Floating Divide Long	171	15.6 (15.8) μ s
FDVM	Floating Divide to Memory	172	15.1 (15.3) μ s
FDVB	Floating Divide to Both	173	15.1 (15.3) μ s

In long mode a nonzero unnormalized dividend whose entire high order fraction is zero produces a zero quotient. In this case the second AC receives rubbish.

If unnormalized operands are used, all times must be increased by .25N. If the division is not performed, only 4-4.5 μ s are required.

2.7 ARITHMETIC TESTING

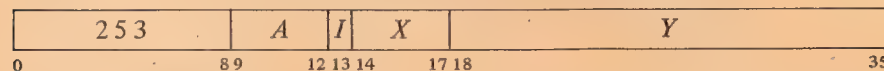
These instructions may jump or skip depending on the result of an arithmetic test and may first perform an arithmetic operation on the test word. Two of the instructions have no modes.

AOBJP **Add One to Both Halves of AC and Jump if Positive** 1.68 (1.79) μ s



Add 1000001_8 to AC and place the result back in AC. If the result is greater than or equal to zero (*ie* if bit 0 is 0, and hence a negative count in the left half has reached zero or a positive count has not yet reached 2^{17}), take the next instruction from location E and continue sequential operation from there.

AOBJN **Add One to Both Halves of AC and Jump if Negative** 1.68 (1.79) μ s



Add 1000001_8 to AC and place the result back in AC. If the result is less than zero (*ie* if bit 0 is 1, and hence a negative count in the left half has not yet reached zero or a positive count has reached 2^{17}), take the next instruction from location E and continue sequential operation from there.

The incrementing of both halves of AC simultaneously is effected by adding 1000001_8 . A count of -2 in AC left is therefore increased to zero if $2^{18} - 1$ is incremented in AC right.

These two instructions allow the program to keep a control count in the left half of an index register and require only one data transfer to initialize. Problem: Add 3 to each location in a table of N entries starting at TAB. Only four instructions are required.

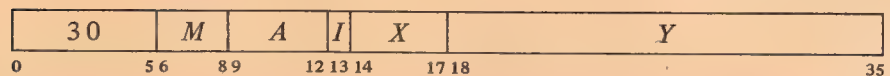
MOVSI	XR, -N	;Put -N in XR left (clear XR right)
MOVEI	AC, 3	;Put 3 in AC
ADDM	AC, TAB(XR)	;Add 3 to entry
AOBJN	XR, -1	;Update XR and go back unless all ;entries accounted for

The eight remaining instructions jump or skip if the operand or operands satisfy a test condition specified by the mode.

<i>Mode</i>	<i>Suffix</i>
Never	
Less	L
Equal	E
Less or Equal	LE
Always	A
Greater or Equal	GE
Not Equal	N
Greater	G

Instructions with one operand compare AC or the contents of location E with zero, those with two compare AC with E or the contents of location E . The processor always makes the comparison even though the result is used in only six of the modes. If the mnemonic has no suffix there is never any program control function, and the instruction may be a no-op; an A suffix produces an unconditional jump or skip — the action is always taken regardless of how the two quantities compare.

CAI **Compare AC Immediate and Skip if Condition Satisfied** 1.68 (1.79) μ s



Compare AC with E (ie with the word 0, E) and skip the next instruction in sequence if the condition specified by M is satisfied.

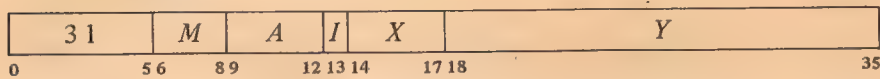
§2.7

CAI	Compare AC Immediate but Do Not Skip	300
CAIL	Compare AC Immediate and Skip if AC Less than E	301
CAIE	Compare AC Immediate and Skip if Equal	302
CAILE	Compare AC Immediate and Skip if AC Less than or Equal to E	303
CAIA	Compare AC Immediate but Always Skip	304
CAIGE	Compare AC Immediate and Skip if AC Greater than or Equal to E	305
CAIN	Compare AC Immediate and Skip if Not Equal	306
CAIG	Compare AC Immediate and Skip if AC Greater than E	307

CAI is a no-op.

CAM **Compare AC with Memory and Skip if Condition Satisfied** 2.53 (2.75) μ s

When E addresses a fast memory location, this instruction takes .34 μ s less than the time given.

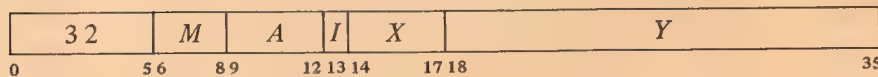


Compare AC with the contents of location E and skip the next instruction in sequence if the condition specified by M is satisfied. The pair of numbers compared may be either both fixed or both normalized floating point.

CAM	Compare AC with Memory but Do Not Skip	310
CAML	Compare AC with Memory and Skip if AC Less	311
CAME	Compare AC with Memory and Skip if Equal	312
CAMLE	Compare AC with Memory and Skip if AC Less or Equal	313
CAMA	Compare AC with Memory but Always Skip	314
CAMGE	Compare AC with Memory and Skip if AC Greater or Equal	315
CAMN	Compare AC with Memory and Skip if Not Equal	316
CAMG	Compare AC with Memory and Skip if AC Greater	317

CAM is a no-op that references memory.

JUMP **Jump if AC Condition Satisfied** 1.68 (1.79) μ s



Compare AC (fixed or floating) with zero, and if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there.

JUMP	Do Not Jump	320
JUMPL	Jump if AC Less than Zero	321
JUMPE	Jump if AC Equal to Zero	322

JUMP is a no-op (instruction code 320 has this mnemonic for symmetry).

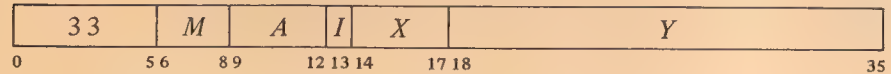
JUMPLE	Jump if AC Less than or Equal to Zero	323
JUMPA	Jump Always	324
JUMPGE	Jump if AC Greater than or Equal to Zero	325
JUMPN	Jump if AC Not Equal to Zero	326
JUMPG	Jump if AC Greater than Zero	327

When E addresses a fast memory location, this instruction takes $.34 \mu s$ less than the time given.

If A is zero, SKIP is a no-op; otherwise it is equivalent to MOVE. (Instruction code-330 has mnemonic SKIP for symmetry.)

SKIPA is a convenient way to load an accumulator and skip over an instruction upon entering a loop.

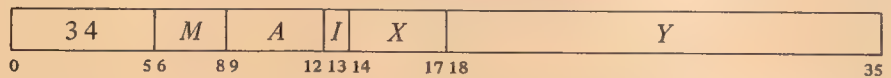
SKIP Skip if Memory Condition Satisfied 2.39 (2.61) μs



Compare the contents (fixed or floating) of location E with zero, and skip the next instruction in sequence if the condition specified by M is satisfied. If A is nonzero also place the contents of location E in AC.

SKIP	Do Not Skip	330
SKIPL	Skip if Memory Less than Zero	331
SKIPE	Skip if Memory Equal to Zero	332
SKIPL	Skip if Memory Less than or Equal to Zero	333
SKIPA	Skip Always	334
SKIPGE	Skip if Memory Greater than or Equal to Zero	335
SKIPN	Skip if Memory Not Equal to Zero	336
SKIPG	Skip if Memory Greater than Zero	337

AOJ Add One to AC and Jump if Condition Satisfied 1.68 (1.79) μs

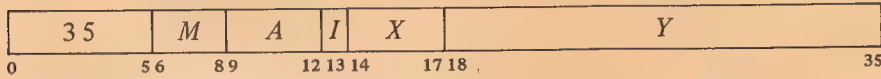


Increment AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by M is satisfied, take the next instruction from location E and continue sequential operation from there. If AC originally contained $2^{35} - 1$, set the Overflow and Carry 1 flags; if -1 , set Carry 0 and Carry 1.

AOJ	Add One to AC but Do Not Jump	340
AOJL	Add One to AC and Jump if Less than Zero	341
AOJE	Add One to AC and Jump if Equal to Zero	342
AOJLE	Add One to AC and Jump if Less than or Equal to Zero	343
AOJA	Add One to AC and Jump Always	344
AOJGE	Add One to AC and Jump if Greater than or Equal to Zero	345
AOJN	Add One to AC and Jump if Not Equal to Zero	346
AOJG	Add One to AC and Jump if Greater than Zero	347

§2.7

AOS Add One to Memory and Skip if Condition Satisfied 2.94 (3.05) μ s

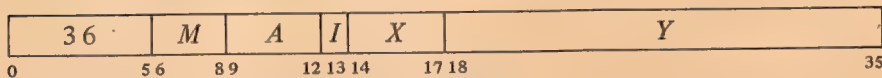


Increment the contents of location *E* by one and place the result back in *E*. Compare the result with zero, and skip the next instruction in sequence if the condition specified by *M* is satisfied. If location *E* originally contained $2^{35} - 1$, set the Overflow and Carry 1 flags; if -1 , set Carry 0 and Carry 1. If *A* is nonzero also place the result in AC.

AOS	Add One to Memory but Do Not Skip	350
AOSL	Add One to Memory and Skip if Less than Zero	351
AOSE	Add One to Memory and Skip if Equal to Zero	352
AOSLE	Add One to Memory and Skip if Less than or Equal to Zero	353
AOSA	Add One to Memory and Skip Always	354
AOSGE	Add One to Memory and Skip if Greater than or Equal to Zero	355
AOSN	Add One to Memory and Skip if Not Equal to Zero	356
AOSG	Add One to Memory and Skip if Greater than Zero	357

Keeping the count in fast memory saves .54 (.43) μ s; keeping it in a different memory from the instruction saves .20 (.09) μ s.

SOJ Subtract One from AC and Jump if Condition Satisfied 1.68 (1.79) μ s

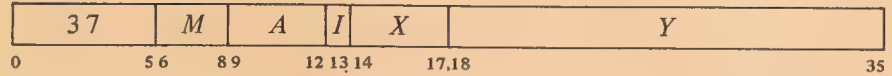


Decrement AC by one and place the result back in AC. Compare the result with zero, and if the condition specified by *M* is satisfied, take the next instruction from location *E* and continue sequential operation from there. If AC originally contained -2^{35} , set the Overflow and Carry 0 flags; if any other nonzero number, set Carry 0 and Carry 1.

SOJ	Subtract One from AC but Do Not Jump	360
SOJL	Subtract One from AC and Jump if Less than Zero	361
SOJE	Subtract One from AC and Jump if Equal to Zero	362
SOJLE	Subtract One from AC and Jump if Less than or Equal to Zero	363
SOJA	Subtract One from AC and Jump Always	364
SOJGE	Subtract One from AC and Jump if Greater than or Equal to Zero	365
SOJN	Subtract One from AC and Jump if Not Equal to Zero	366
SOJG	Subtract One from AC and Jump if Greater than Zero	367

Keeping the count in fast memory saves .54 (.43) μ s; keeping it in a different memory from the instruction saves .20 (.09) μ s.

SOS **Subtract One from Memory and Skip if Condition Satisfied** 2.94 (3.05) μ s



Decrement the contents of location *E* by one and place the result back in *E*. Compare the result with zero, and skip the next instruction in sequence if the condition specified by *M* is satisfied. If location *E* originally contained -2^{35} , set the Overflow and Carry 0 flags; if any other nonzero number, set Carry 0 and Carry 1. If *A* is nonzero also place the result in AC.

SOS	Subtract One from Memory but Do Not Skip	370
SOSL	Subtract One from Memory and Skip if Less than Zero	371
SOSE	Subtract One from Memory and Skip if Equal to Zero	372
SOSLE	Subtract One from Memory and Skip if Less than or Equal to Zero	373
SOSA	Subtract One from Memory and Skip Always	374
SOSGE	Subtract One from Memory and Skip if Greater than or Equal to Zero	375
SOSN	Subtract One from Memory and Skip if Not Equal to Zero	376
SOSG	Subtract One from Memory and Skip if Greater than Zero	377

Some of these instructions are useful for determining the relative values of fixed and floating point numbers; others are convenient for controlling iterative processes by counting. AOSE is especially useful in an interlock procedure in a multiprocessor system. Suppose memory contains a routine that must be available to two processors but cannot be used by both at once. When one processor finishes the routine it sets location LOCK to -1 . Either processor can then test the interlock and make it busy with no possibility of letting the other one in, as AOSE cannot be interrupted once it starts to modify the addressed location.

This procedure is invalid if the programmer is making use of the drum split feature (which is not used by any DEC equipment).

```

AOSE  LOCK      ;Skip to interlocked code only if
JRST  .-1      ;LOCK is zero after addition
      .         ;Interlocked code starts here
      .
      .
SETOM LOCK      ;Unlock

```

Since it takes several days to count to 2^{36} , it is alright to keep testing the lock.

2.8 LOGICAL TESTING AND MODIFICATION

These eight instructions use a mask to modify and/or test selected bits in AC. The bits are those that correspond to 1s in the mask and they are referred to as the "masked bits". The programmer chooses the mask, the way in which the masked bits are to be modified, and the condition the masked bits must satisfy to produce a skip.

The basic mnemonics are three letters beginning with T. The second letter selects the mask and the manner in which it is used.

<i>Mask</i>	<i>Letter</i>	<i>Effect</i>
Right	R	AC right is masked by <i>E</i> (AC is masked by the word 0, <i>E</i>)
Left	L	AC left is masked by <i>E</i> (AC is masked by the word <i>E</i> ,0)
Direct	D	AC is masked by the contents of location <i>E</i>
Swapped	S	AC is masked by the contents of location <i>E</i> with left and right halves interchanged

If a direct or swapped mask is taken from a fast memory location, a test instruction takes .34 μ s less than the time listed.

The third letter determines the way in which those bits selected by the mask are modified.

<i>Modification</i>	<i>Letter</i>	<i>Effect on AC</i>
No	N	None
Zeros	Z	Places 0s in all masked bit positions
Complement	C	Complements all masked bits
Ones	O	Places 1s in all masked bit positions

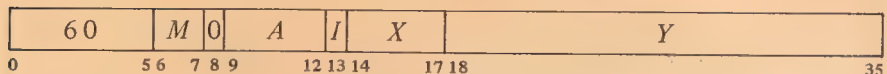
An additional letter may be appended to indicate the mode, which specifies the condition the masked bits must satisfy to produce a skip.

<i>Mode</i>	<i>Suffix</i>	<i>Effect</i>
Never		Never skip
Equal	E	Skip if all masked bits equal 0
Always	A	Always skip
Not Equal	N	Skip if not all masked bits equal 0 (at least one bit is 1)

These mode names are consistent with those for arithmetic testing and derive from the test method, which ands AC with the mask and tests whether the result is equal to zero or is not equal to zero. The programmer may find it convenient to think of the modes as Every and Not Every: every masked bit is 0 or not every masked bit is 0.

If the mnemonic has no suffix there is never any skip, and the instruction is a no-op if there is also no modification; an A suffix produces an unconditional skip — the skip always occurs regardless of the state of the masked bits. Note that the skip condition must be satisfied by the state of the masked bits *prior* to any modification called for by the instruction.

TRN **Test Right, No Modification, and Skip if Condition Satisfied** 1.85 (1.96) μ s

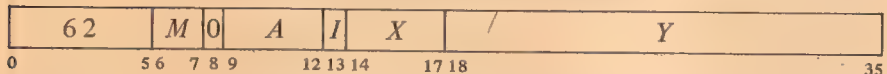


If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TRN is a no-op.

TRN	Test Right, No Modification, but Do Not Skip	600
TRNE	Test Right, No Modification, and Skip if All Masked Bits Equal 0	602
TRNA	Test Right, No Modification, but Always Skip	604
TRNN	Test Right, No Modification, and Skip if Not All Masked Bits Equal 0	606

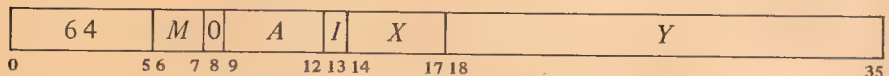
TRZ **Test Right, Zeros, and Skip if Condition Satisfied** 1.85 (1.96) μ s



If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TRZ	Test Right, Zeros, but Do Not Skip	620
TRZE	Test Right, Zeros, and Skip if All Masked Bits Equaled 0	622
TRZA	Test Right, Zeros, but Always Skip	624
TRZN	Test Right, Zeros, and Skip if Not All Masked Bits Equaled 0	626

TRC **Test Right, Complement, and Skip if Condition Satisfied** 1.85 (1.96) μ s

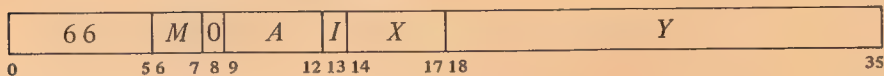


If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TRC	Test Right, Complement, but Do Not Skip	640
TRCE	Test Right, Complement, and Skip if All Masked Bits Equaled 0	642
TRCA	Test Right, Complement, but Always Skip	644
TRCN	Test Right, Complement, and Skip if Not All Masked Bits Equaled 0	646

§2.8

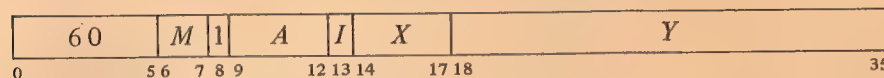
TRO Test Right, Ones, and Skip if Condition Satisfied 1.85 (1.96) μ s



If the bits in AC right corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TRO	Test Right, Ones, but Do Not Skip	660
TROE	Test Right, Ones, and Skip if All Masked Bits Equaled 0	662
TROA	Test Right, Ones, but Always Skip	664
TRON	Test Right, Ones, and Skip if Not All Masked Bits Equaled 0	666

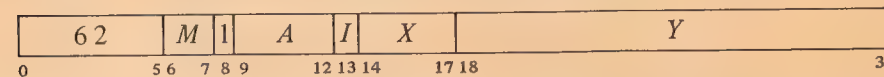
TLN Test Left, No Modification, and Skip if Condition Satisfied 1.85 (1.96) μ s



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TLN	Test Left, No Modification, but Do Not Skip	601	TLN is a no-op.
TLNE	Test Left, No Modification, and Skip if All Masked Bits Equal 0	603	
TLNA	Test Left, No Modification, but Always Skip	605	
TLNN	Test Left, No Modification, and Skip if Not All Masked Bits Equal 0	607	

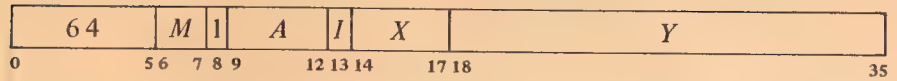
TLZ Test Left, Zeros, and Skip if Condition Satisfied 1.85 (1.96) μ s



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TLZ	Test Left, Zeros, but Do Not Skip	621
TLZE	Test Left, Zeros, and Skip if All Masked Bits Equaled 0	623
TLZA	Test Left, Zeros, but Always Skip	625
TLZN	Test Left, Zeros, and Skip if Not All Masked Bits Equaled 0	627

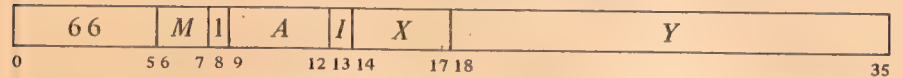
TLC Test Left, Complement, and Skip if Condition Satisfied 1.85 (1.96) μ s



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

- TLC Test Left, Complement, but Do Not Skip 641
- TLCE Test Left, Complement, and Skip if All Masked Bits Equaled 0 643
- TLCA Test Left, Complement, but Always Skip 645
- TLCN Test Left, Complement, and Skip if Not All Masked Bits Equaled 0 647

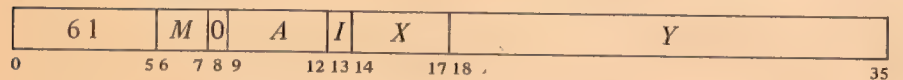
TLO Test Left, Ones, and Skip if Condition Satisfied 1.85 (1.96) μ s



If the bits in AC left corresponding to 1s in *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

- TLO Test Left, Ones, but Do Not Skip 661
- TLOE Test Left, Ones, and Skip if All Masked Bits Equaled 0 663
- TLOA Test Left, Ones, but Always Skip 665
- TLON Test Left, Ones, and Skip if Not All Masked Bits Equaled 0 667

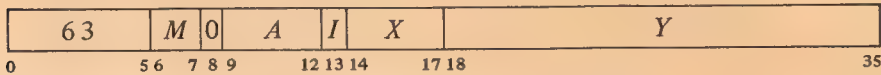
TDN Test Direct, No Modification, and Skip if Condition Satisfied 2.70 (2.92) μ s



If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

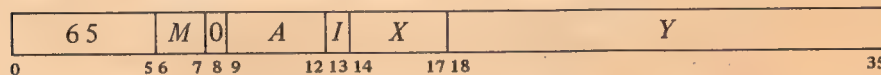
- TDN Test Direct, No Modification, but Do Not Skip 610
- TDNE Test Direct, No Modification, and Skip if All Masked Bits Equal 0 612
- TDNA Test Direct, No Modification, but Always Skip 614
- TDNN Test Direct, No Modification, and Skip if Not All Masked Bits Equal 0 616

TDN is a no-op that references memory.

TDZ Test Direct, Zeros, and Skip if Condition Satisfied 2.70 (2.92) μ s

If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TDZ	Test Direct, Zeros, but Do Not Skip	630
TDZE	Test Direct, Zeros, and Skip if All Masked Bits Equaled 0	632
TDZA	Test Direct, Zeros, but Always Skip	634
TDZN	Test Direct, Zeros, and Skip if Not All Masked Bits Equaled 0	636

TDC Test Direct, Complement, and Skip if Condition Satisfied 2.70 (2.92) μ s

If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

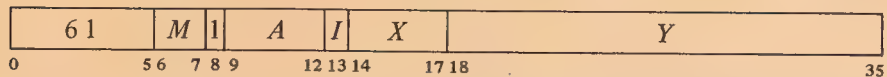
TDC	Test Direct, Complement, but Do Not Skip	650
TDCE	Test Direct, Complement, and Skip if All Masked Bits Equaled 0	652
TDCA	Test Direct, Complement, but Always Skip	654
TD CN	Test Direct, Complement, and Skip if Not All Masked Bits Equaled 0	656

TDO Test Direct, Ones, and Skip if Condition Satisfied 2.70 (2.92) μ s

If the bits in AC corresponding to 1s in the contents of location *E* satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TDO	Test Direct, Ones, but Do Not Skip	670
TDOE	Test Direct, Ones, and Skip if All Masked Bits Equaled 0	672
TDOA	Test Direct, Ones, but Always Skip	674
TDON	Test Direct, Ones, and Skip if Not All Masked Bits Equaled 0	676

TSN **Test Swapped, No Modification, and Skip if Condition Satisfied** 2.70 (2.92) μ s

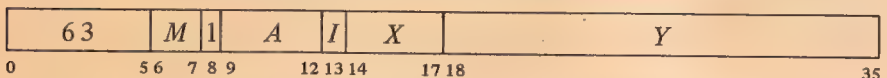


If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. AC is unaffected.

TSN is a no-op that references memory.

TSN	Test Swapped, No Modification, but Do Not Skip	611
TSNE	Test Swapped, No Modification, and Skip if All Masked Bits Equal 0	613
TSNA	Test Swapped, No Modification, but Always Skip	615
TSNN	Test Swapped, No Modification, and Skip if Not All Masked Bits Equal 0	617

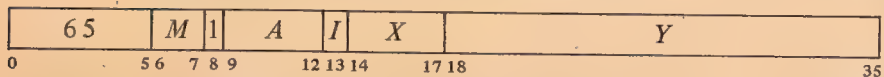
TSZ **Test Swapped, Zeros, and Skip if Condition Satisfied** 2.70 (2.92) μ s



If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 0s; the rest of AC is unaffected.

TSZ	Test Swapped, Zeros, but Do Not Skip	631
TSZE	Test Swapped, Zeros, and Skip if All Masked Bits Equaled 0	633
TSZA	Test Swapped, Zeros, but Always Skip	635
TSZN	Test Swapped, Zeros, and Skip if Not All Masked Bits Equaled 0	637

TSC **Test Swapped, Complement, and Skip if Condition Satisfied** 2.70 (2.92) μ s



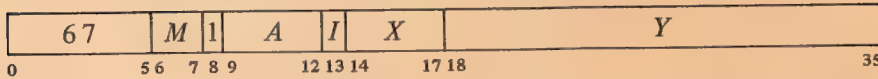
If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Complement the masked AC bits; the rest of AC is unaffected.

TSC	Test Swapped, Complement, but Do Not Skip	651
TSCE	Test Swapped, Complement, and Skip if All Masked Bits Equaled 0	653

§2.8

TSCA	Test Swapped, Complement, but Always Skip	655
TSCN	Test Swapped, Complement, and Skip if Not All Masked Bits Equaled 0	657

TSO **Test Swapped, Ones, and Skip if Condition Satisfied** 2.70 (2.92) μ s



If the bits in AC corresponding to 1s in the contents of location *E* with its left and right halves swapped satisfy the condition specified by *M*, skip the next instruction in sequence. Change the masked AC bits to 1s; the rest of AC is unaffected.

TSO	Test Swapped, Ones, but Do Not Skip	671
TSOE	Test Swapped, Ones, and Skip if All Masked Bits Equaled 0	673
TSOA	Test Swapped, Ones, but Always Skip	675
TSON	Test Swapped, Ones, and Skip if Not All Masked Bits Equaled 0	677

With these instructions any bit throughout all of memory can be used as a program flag, although an ordinary memory location containing flags must be moved to an accumulator for testing or modification. The usual procedure, since locations 1–17 are addressable as index registers, is to use AC 0 as a register of flags (often addressed symbolically as F).

Unless one frequently tests flags in both halves of F simultaneously, it is generally most convenient to select bits by 1s right in the address part of the instruction word. A given bit selected by a half word mask *M* is then set by one of these:

TRO F,*M* TLO F,*M*

and tested and cleared by one of these:

TRZE F,*M* TRZN F,*M* TLZE F,*M* TLZN F,*M*

Suppose we wish to skip if both bits 34 and 35 are 1 in location L. The following suffices.

SETCM F,L
TRNE F,3

We can refer to a flag in a given bit position within a word as flag *X*, where *X* is a binary number containing a single 1 in the same bit position as the flag. This sequence determines whether flags *X* and *Y* in the right half of accumulator F are both on:

TRC	F, X + Y	;Complement flags X and Y
TRCE	F, X + Y	;Test both and restore original states
...		;Do this if not both on
...		;Skip to here if both on

2.9 PROGRAM CONTROL

The program control class of instructions includes the unimplemented user operations [*discussed in the next section*] and the arithmetic and logical test instructions. Some instructions in this class are no-ops, as are a few of the instructions for performing logical operations. The most commonly used no-op is JFCL, which is discussed below. No-ops among the instructions previously discussed are SETA, SETAI, SETMM, CAI, CAM, JUMP, TRN, TLN, TDN, TSN. Of these, SETA, SETAI, CAI, JUMP, TRN and TLN do not use the calculated effective address to reference memory. Hence in these instructions one can store any information in bits 18–35 without fear of attempting to address a location outside a user block or in a memory that does not exist. The unassigned instruction codes 247 and 257 are used for instructions installed specially for a particular system. They execute as no-ops when run on a computer that contains no special hardware for them, but for program compatibility it is advised that they not be used regularly as no-ops.

As no-ops, code 247 takes
1.50 (1.61) μ s, 257 takes
1.36 (1.47) μ s.

The present section treats all program control instructions other than those mentioned above and in-out instructions that test input conditions [§2.12]. All but one of these are jumps, although the exception causes the processor to execute an instruction at an arbitrary location and may therefore be regarded as a jump with an immediate and automatic return. Also, all but two of the jumps are unconditional; one exception tests various flags, the other tests an accumulator.

Several of the jump instructions save the current contents of the program counter PC in the right half of an accumulator or memory location and save the states of various flags in the left half. The left bit positions that receive information are listed below; all other left bit positions are cleared. An *X* in a mnemonic indicates any letter (or none) that may appear in the given position to specify the mode, eg ADDX comprises ADD, ADDI, ADDM, ADDB.

Note that nothing is stored in bits 13–17, so when the PC word is addressed indirectly it can produce neither indexing nor further indirect addressing.

Bit

Meaning of a 1 in the Bit

0 Overflow – any of the following has occurred:

A single instruction has set one of the carry flags (bits 1 and 2) without setting the other.

An ASH or ASHC has left shifted a 1 out of bit 1 in a positive number or a 0 out in a negative number.

An MULX has multiplied -2^{35} by itself (product 2^{70}).

An IMULX has multiplied two numbers with product $\geq 2^{35}$ or $< -2^{35}$.

§2.9

Floating Overflow has been set (bit 3).

No Divide has been set (bit 12).

- 1 Carry 0 – if set without Carry 1 (bit 2) being set, causes Overflow to be set and indicates that one of the following has occurred:

An ADDX has added two negative numbers with sum $< -2^{35}$.

An SUBX has subtracted a positive number from a negative number with difference $< -2^{35}$.

An SOJX or SOSX has decremented -2^{35} .

But if set with Carry 1, indicates that one of these nonoverflow events has occurred:

In an ADDX both summands were negative, or their signs differed and their magnitudes were equal or the positive one was the greater in magnitude.

In an SUBX the signs of the operands were the same and AC was the greater or the two were equal, or the signs of the operands differed and AC was negative.

An AOJX or AOSX has incremented -1 .

An SOJX or SOSX has decremented a nonzero number other than -2^{35} .

An MOVNX has negated zero.

- 2 Carry 1 – if set without Carry 0 (bit 1) being set, causes Overflow to be set and indicates that one of the following has occurred:

An ADDX has added two positive numbers with sum $\geq 2^{35}$.

An SUBX has subtracted a negative number from a positive number with difference $\geq 2^{35}$.

An AOJX or AOSX has incremented $2^{35} - 1$.

An MOVNX or MOVMX has negated -2^{35} .

But if set with Carry 0, indicates that one of the nonoverflow events listed under Carry 0 has occurred.

- 3 Floating Overflow – any of the following has set Overflow:

In a floating point instruction other than DFN, the exponent of the result was > 127 .

Floating Underflow (bit 11) has been set.

No Divide (bit 12) has been set in an FDVX or FDVRX.

- 4 Byte Interrupt – the processor is in a priority interrupt that interrupted a byte instruction after the processing of the pointer but before the processing of the byte. Hence if an ILDB or IDPB was interrupted, the pointer now points not to the last byte, but rather to the byte that should be handled upon the return to the interrupted program [§2.13].

- 5 User – the processor is in user mode [§2.15].

Remember [§2.5], overflow is determined directly from the carries, not from the flags. The carry flags give meaningful information only if no more than one instruction that can set them occurs between clearing and reading them.

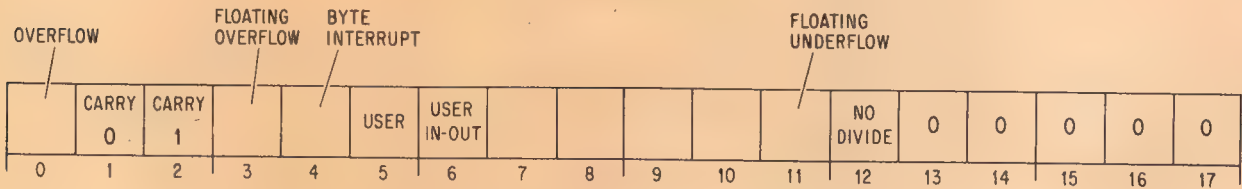
- ▲ 6 User In-out – even if the processor is in user mode, no instructions are illegal (but protection and relocation still apply) [§2.15].
- 11 Floating Underflow – in a floating point instruction other than DFN, the exponent of the result was < -128 and Overflow and Floating Overflow have been set.
- 12 No Divide – any of the following has set Overflow:

In a DIVX the dividend was greater than or equal to the divisor.

In an IDIVX the divisor was zero.

In an FDVX or FDVRX the divisor was zero, or the dividend fraction was greater than or equal to twice the divisor fraction in magnitude; in either case Floating Overflow has been set.

If normalized operands are used, only a zero divisor can cause floating division to fail.

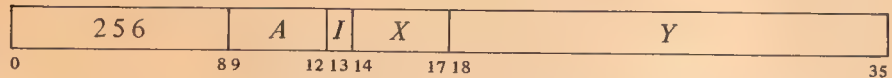


FLAG FORMAT, LEFT HALF OF PC WORD

The total time required is that listed plus the time for the instruction executed. If *E* addresses a fast memory location, the instruction executed takes .34 μ s less than the time listed for it.

The *A* portion of this instruction is ignored.

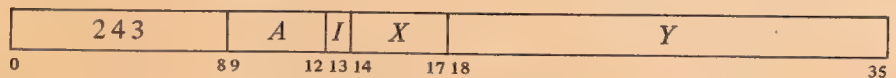
XCT **Execute** 1.36 (1.47) μ s



Execute the contents of location *E* as an instruction. Any instruction may be executed, including another XCT. If an XCT executes a skip instruction, the skip is relative to the location of the XCT (the first XCT if there are several in a chain). If an XCT executes a jump, program flow is altered as specified by the jump (no matter how many XCTs precede a jump instruction, when PC is saved it contains an address one greater than the location of the first XCT in the chain).

N is the number of leading 0s.

JFFO **Jump if Find First One** 2.19 (2.30) + .20 (*N* mod 18) μ s



If AC contains zero, clear AC *A*+1 and go on to the next instruction in sequence.

If AC is not zero, count the number of leading 0s in it (0s to the left of the leftmost 1), and place the count in AC *A*+1. Take the next instruction

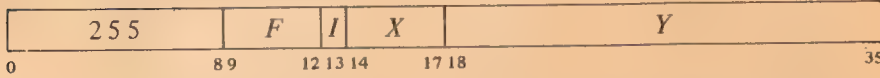
§2.9

from location *E* and continue sequential operation from there.

In either case AC is unaffected, the original contents of AC *A*+1 are lost.

Note that when AC is negative, the second accumulator is cleared, just as it would be if AC were zero.

JFCL **Jump on Flag and Clear** 1.36 (1.47) μ s



If any flag specified by *F* is set, clear it and take the next instruction from location *E*, continuing sequential operation from there. Bits 9–12 are programmed as follows.

<i>Bit</i>	<i>Flag Selected by a 1</i>
9	Overflow
10	Carry 0
11	Carry 1
12	Floating Overflow

To select one or a combination of these flags (which are among those described above) the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for some of the 13-bit instruction codes (bits 0–12).

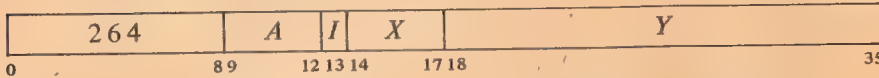
JFCL	JFCL 0,	No-op	25500
JOV	JFCL 10,	Jump on Overflow	25540
JCRY0	JFCL 4,	Jump on Carry 0	25520
JCRY1	JFCL 2,	Jump on Carry 1	25510
JCRY	JFCL 6,	Jump on Carry 0 or 1	25530
JFOV	JFCL 1,	Jump on Floating Overflow	25504

This instruction can be used simply to clear the selected flags by having the jump address point to the next consecutive location, as in

JFCL 17,+1

which clears all four flags without disrupting the normal program sequence. A JFCL that selects no flag is the fastest no-op as it neither fetches nor stores an operand, and bits 18–35 of the instruction word can be used to store information.

JSR **Jump to Subroutine** 2.68 (2.79) μ s ▲

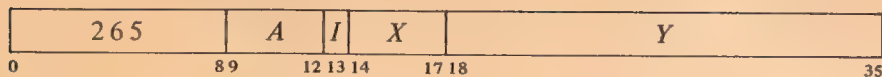


Place the current contents of the flags (as described above) in the left half of location *E* and the contents of PC in the right half (at this time PC contains an address one greater than the location of the JSR instruction). Take the next instruction from location *E*+1 and continue sequential operation from there. The flags are unaffected except Byte Interrupt, which is cleared.

While the processor is in user mode, if this instruction is executed as an interrupt instruction or in unrelocated 41 or 61, bit 5 of the PC word stored is 1 and the processor leaves user mode. ▲

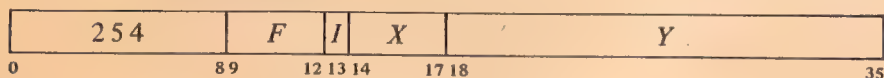
Interleaving memories saves .47 (.36) μ s.

The *A* portion of this instruction is ignored.

JSP Jump and Save PC 1.36 (1.47) μ s

Place the current contents of the flags (as described above) in AC left and the contents of PC in AC right (at this time PC contains an address one greater than the location of the JSP instruction). Take the next instruction from location *E* and continue sequential operation from there. The flags are unaffected except Byte Interrupt, which is cleared.

- ▲ While the processor is in user mode, if this instruction is executed as an interrupt instruction or in unrelocated 41 or 61, bit 5 of the PC word stored is 1 and the processor leaves user mode.

JRST Jump and Restore 1.36 (1.47) μ s

Perform the functions specified by *F*, then take the next instruction from location *E* and continue sequential operation from there. Bits 9–12 are programmed as follows.

Bit Function Produced by a 1

- 9 Restore the channel on which the highest priority interrupt is currently being held [§2.13].

Unless the User In-out flag is set, this function cannot be executed in a user program. Instead of restoring the channel, it stores its own instruction code, *F* and effective address *E* in bits 0–8, 9–12 and 18–35 respectively of unrelocated location 40 (clearing bits 13–17), and then executes the instruction contained in location 41, which is under control of the monitor [§2.15].

- 10 Halt the processor. When it stops, the MA lights on the console display an address one greater than that of the location containing the instruction that caused the halt, and PC displays the jump address (the location from which the next instruction will be taken if the operator causes the processor to resume operation without changing PC).

Unless the User In-out flag is set, this function cannot be executed in a user program. Instead of halting the processor, it stores its own instruction code, *F* and effective address *E* in Bits 0–8, 9–12 and 18–35 respectively of unrelocated location 40 (clearing bits 13–17), and then executes the instruction contained in location 41, which is under control of the monitor [§2.15].

- 11 Restore the flags listed above from the left half of the word in the last location referenced in the effective address calculation. Hence to restore flags requires that the JRST instruction use indexing or

This is identical to UUU trapping [§2.10].

MA actually displays the address of the location that would have been executed next had the JRST been replaced by a no-op. So except for a JRST in a priority interrupt, MA points to the location one beyond that containing the instruction that caused the halt. This instruction is ordinarily the JRST or perhaps an XCT, but could even be a UUU.

§2.9

indirect addressing.

Restoration of all but the user flags is directly according to the contents of the corresponding bits as given above: a flag is set by a 1 in the bit, cleared by a 0. A 1 in bit 5 sets User but a 0 has no effect, so the Monitor can restart a user program by restoring flags but the user cannot leave user mode by this method. A 0 in bit 6 clears User In-out, but a 1 sets it only if the JRST is being executed by the Monitor, *ie* if User is clear.

12 Enter user mode. The user program starts at relocated location *E*.

To produce one or a combination of these functions the programmer can specify the equivalent of an AC address that places 1s in the appropriate bits, but MACRO recognizes mnemonics for the most important 13-bit instruction codes (bits 0–12).

JRST	JRST 0,	Jump	25400
	JRST 10,	Jump and Restore Interrupt Channel	25440
HALT	JRST 4,	Halt	25420
JRSTF	JRST 2,	Jump and Restore Flags	25410
	JRST 1,	Jump to User Program	25404
JEN	JRST 12,	Jump and Enable	25450

In a JRSTF or JEN the flags are restored from bits 0–12 of the final word retrieved in the effective address calculation; hence any JRST with a 1 in bit 11 must use indirect addressing or indexing, which takes extra time. If the PC word was stored in AC (as in a JSP), a common procedure is to use AC to index a zero address (*eg*, JRSTF (AC)), so its right half becomes the effective (jump) address. If the PC word was stored in core (as in a JSR), one must address it indirectly (remember, bits 13–17 of the PC word are clear, so again its right half is the effective address). A JRSTF (AC) takes 1.64 (1.75) μ s, a JRSTF @PCWORD takes 2.34 (2.56) μ s.

CAUTION

Giving a JRSTF or JEN without indexing or indirect addressing restores the flags from the instruction code itself.

While the processor is in user mode, if this instruction is executed as an interrupt instruction or in unrelocated 41 or 61, bit 5 of the PC word stored is 1 and the processor leaves user mode. ▲

JFCL is the only jump that can test any of the flags directly. In fact it is the only basic program control instruction that can do so — several of the flags can be tested as processor conditions by in-out instructions, but these are ordinarily illegal in user programs anyway. But JFCL can test only four

By manipulating the contents of the left half word used to restore the flags, the programmer can set them up in any desired way except that a user program cannot clear User or set User In-out. Setting Byte Interrupt prevents incrementing in the next ILDB or IDPB provided there is no intervening JSR, JSP or PUSHJ.

JEN completes an interrupt by restoring the channel and restoring the flags for the interrupted program.

of the flags, and it saves no information for a subsequent return from a subroutine. Hence it serves as a branch point for entry into either one of two main paths, which may or may not have a later point in common. *Eg*, it may test the carry flags simply to take appropriate action in a double precision fixed point routine.

JSR and JSP are regularly used to call subroutines. They are unconditional, but the execution of such an instruction can be the result of a decision made by any conditional skip or jump. In the case of the flags, a basic overflow test and subroutine call can be made as follows.

The fastest skip is CAIA.

```
JOV      .+2
JRST     .+2      ;Faster than skipping
JSR      OVRFLO   ;Jump over this if Overflow clear
:
:
```

If we wish to go to the DIVERR routine when No Divide is set, we must first read the flags into a test accumulator T and then use a test instruction.

```
JSP      T, .+1   ;Store flags but continue in sequence
TLNE     T, 40    ;40 left selects bit 12
JSR      DIVERR   ;Skip this if No Divide clear
:
:
```

A subroutine called by a JSR must have its entry point reserved for the PC word. Hence it is nonreentrant: the JSR modifies memory so the subroutine cannot be shared with other programs. The JSP requires an accumulator, but it is faster and is convenient for argument passing. To return from a JSR-called subroutine one usually addresses the PC word indirectly, returning to the location following the JSR. But there are two ways to get back from a JSP. We can address the PC word indirectly with a JRST @AC (or JRSTF @AC if the flags must be restored), but we can also get it by addressing AC as an index register: JRST (AC). By using the second return method we can place *N* words of data for the subroutine immediately after the call, and return to the location following the data by giving a JRST *N*(AC).

Suppose we wish to call a print subroutine and supply the words from which the characters are to be taken. Our main program would contain the following:

```
JSP      T, PRINT ;Put PC word in accumulator T
:
:               ;Text inserted here by ASCIZ pseudo-
:               ;instruction, which automatically
:               ;places a zero (null) character at the
:               ;end
...           ;Next instruction here
```

The subroutine can use T as a byte pointer which already addresses the first word of data. For the print routine, characters are loaded into another accumulator CH.


```

      .           ;Main program
      .
      JSA      17,S1      ;Call to first subroutine (A)
      .
S1:   0           ;First subroutine starts here
      .
      JSA      17,S2      ;Call to second subroutine (B)
      .
      JRA      17,(17)    ;Return to A + 1 in main program
S2:   0           ;Second subroutine starts here
      .
      JSA      17,S3      ;Call to third subroutine (C)
      .
      JRA      17,(17)    ;Return to B + 1 in first subroutine
S3:   0           ;Third subroutine starts here
      .
      JRA      17,(17)    ;Return to C + 1 in second subroutine

```

To call the next deeper subroutine at any level, a JSA places E and PC in the left and right of AC 17, saves the previous contents of AC 17 in E (the first subroutine location), and jumps to $E + 1$. To return to the next higher level, a JRA restores the previous contents of AC 17 from the location addressed by AC 17 left (the first subroutine location) and jumps to the location addressed by AC 17 right (the location following the JSA in the higher subroutine). If N lines of data for the next subroutine follow a JSA, the return to the location following the data is made by giving a JRA 17, $N(17)$.

Keeping instructions and the pushdown list in different memories saves .47 (.36) μ s.

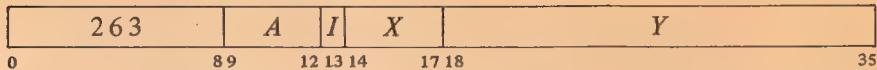
PUSHJ **Push Down and Jump** 3.00 (3.11) μ s



Add 1000001_8 to AC to increment both halves by one and place the result back in AC. If the addition causes the count in AC left to reach zero, set the Pushdown Overflow flag. Then place the current contents of the flags (as described above) in the left half of the location now addressed by AC right and the contents of PC in the right half of that location (at this time PC contains an address one greater than the location of the PUSHJ instruction). Take the next instruction from location E and continue sequential operation from there.

The flags are unaffected except Byte Interrupt, which is cleared. The original contents of the location added to the list are lost.

- ▲ While the processor is in user mode, if this instruction is executed as an interrupt instruction or in unrelocated 41 or 61, bit 5 of the PC word stored is 1 and the processor leaves user mode.

POPJ**Pop Up and Jump**2.96 (3.18) μ s

Subtract 1000001_8 from AC to decrement both halves by one and place the result back in AC. If the subtraction causes the count in AC left to reach -1 , set the Pushdown Overflow flag. Take the next instruction from the location addressed by the right half of the location that was addressed by AC right *prior* to the decrementing, and continue sequential operation from there.

The effective address E is ignored.

The address of the top item in the pushdown list is kept in the right half of the pointer in AC, and the program can keep a control count in the left half. The incrementing and decrementing of both halves of AC simultaneously is effected by adding and subtracting 1000001_8 . Hence a count of -2 in AC left is increased to zero if $2^{18} - 1$ is incremented in AC right, and conversely, 1 in AC left is decreased to -1 if zero is decremented in AC right.

Since the pushdown list is independent of the subroutine called, PUSHJ-POPJ can be used like JSA-JRA for multiple entries. Moreover, ordering by level is inherent in the structure of a pushdown list [§2.2], so paired PUSHJ-POPJ instructions are excellent for nesting subroutines: there can be any number of subroutines at any level, each with more subroutines nested within it. Recursive subroutines are also possible.

Unlike JSA-JRA, the pushdown instructions tie up an accumulator, but the usual procedure is to keep both data and jump addresses in a single list so only one AC is required for the most complex pushdown operations. The programmer must keep track of whether a given entry in the list is data or a PC word; in other words, every item inserted by a PUSH should be removed by a POP, and every PUSHJ should be matched by a POPJ. If flag restoration is desired, the returning

POPJ P,

can be replaced by

POP P,AC
JRSTF (AC)

which requires another accumulator. If the flags are not important, data may be stored in the left halves of the PC words in the stack, reducing the required pushdown depth.

By using the Pushdown Overflow flag and a control count in AC left, the programmer can set a limit to the size of the list by starting the count negative, or he can prevent the program from extracting more items than there are in the list by starting the count at zero, but he cannot do both at once. If only jump addresses are kept in the list, the first procedure limits the depth of nesting. A technique to catch extra POPJs is to put a PC word addressing an error routine at the bottom of the list.

2.10 UNIMPLEMENTED OPERATIONS

An unimplemented user operation is usually referred to as a UUO, but this mnemonic means nothing to the assembler. UUOs are also sometimes called "programmed operators".

Many of the codes not assigned as specific instructions are executed as unimplemented user operations, wherein the word given as an instruction is trapped and must be interpreted by a routine included for this purpose by the programmer. In time sharing, however, half of the codes are set aside for user communication with the Monitor and are interpreted by it. Instructions that are illegal in user mode also trap in this manner.

The total time required is that listed plus the time for the instruction in location 41. Interleaving memories 0 and 1 saves .47 (.36) μ s.

Unimplemented User Operation

2.33 (2.44) μ s

Store the instruction code, *A* and the effective address *E* in bits 0-8, 9-12 and 18-35 respectively of location 40; clear bits 13-17. Execute the instruction contained in location 41. The original contents of location 40 are lost.

▲ "Execute" here means in the sense of the instruction XCT.

All of these codes are equivalent when they occur in the Monitor or when time sharing is not in effect. But when a UUO appears in a user program, a code in the range 001-037 uses relocated locations 40 and 41 (*ie* 40 and 41 in the user's block) and is thus entirely a part of and under control of the user program. A code in the range 040-077 on the other hand uses unrelocated 40 and 41, and the instruction in the latter location is under control of the Monitor; these codes are thus specifically for user communication with the Monitor, which interprets them (refer to the Monitor manual for the meanings of the various codes). The code 000 executes in the same way as 040-077 but is not a standard communication code: it is included so that control returns to the Monitor should a user program wipe itself out.

For a second processor connected to the same memory, the UUO trap is locations 140-141 instead of 40-41.

The unimplemented operations also include the reserved (unassigned) instruction codes 100-127, which execute like the Monitor-calling UUOs but use unrelocated 60-61 (160-161 for a second processor); thus the Monitor steps in when a user gives an incorrect code. The codes 130-177, which are the floating point and byte manipulation instructions, are equivalent to the unassigned codes if unimplemented, *ie* if the optional hardware for them is not included. In this case all codes 100-177 trap to unrelocated 60-61. In general it is assumed that if software is available for floating point and byte manipulation, the Monitor is responsible for calling the appropriate routines.

2.11 PROGRAMMING EXAMPLES

Before continuing to input-output and related subjects, let us consider some simple programs that demonstrate the use of a variety of the instruction described thus far.

Suppose we wish to count the number of 1s in a word. We could of course check every bit in the word. But there is a quicker way if we remember that in any word and its two's complement the rightmost 1 is in the same position, both words are all 0s to the right of this 1, and no corresponding bits are the same to the left (the parts of both words at the left of the rightmost 1 are complements). Hence using the negative of a word as a mask for the word in a test instruction selects only the rightmost 1 for modification. The example uses three accumulators: the word being tested (which is lost) is in T, the count is kept in CNT, and the mask created in each step is stored in TEMP.

```

MOVEI  CNT,0      ;Clear CNT
MOVN   TEMP,T     ;Make mask to select rightmost 1
TDZE   T,TEMP     ;Clear rightmost 1 in T
AOJA   CNT,-2     ;Increase count and jump back
...    ;Skip to here if no 1s left in T

```

CNT is increased by one every time a 1 is deleted from T. After all 1s have been removed, the TDZE skips.

In the standard algorithm for converting a number N to its equivalent in base b , one performs the series of divisions

$$\begin{aligned}
 N/b &= q_1 + r_1/b & r_1 < b \\
 q_1/b &= q_2 + r_2/b & r_2 < b \\
 q_2/b &= q_3 + r_3/b & r_3 < b \\
 &\vdots \\
 q_{n-1}/b &= 0 + r_n/b & r_n < b
 \end{aligned}$$

The number in base b is then $r_n \dots r_3 r_2 r_1$. Eg the octal equivalent of 61 decimal is 75:

$$\begin{aligned}
 61/8 &= 7 + 5/8 \\
 7/8 &= 0 + 7/8
 \end{aligned}$$

The following decimal print routine converts a 36-bit positive integer in accumulator T to decimal and types it out. The contents of T and T+1 are destroyed. The routine is called by a PUSHJ P, DECPNT where P is the pushdown pointer.

```

DECPNT: IDIVI   T,12      ;128 = 1010
        PUSH   P,T+1     ;Save remainder
        SKIPE  T         ;All digits formed?
        PUSHJ  P,DECPNT  ;No, compute next one

```

```

DECPN1: POP      P,T          ;Yes, take out in opposite order
         ADDI    T,60        ;Convert to ASCII (60 is code for 0)
         JRST   TTYOUT      ;Type out

```

This routine repeats the division until it produces a zero quotient. Hence it suppresses leading zeros, but since it is executed at least once it outputs one "0" if the number is zero. The TTYOUT routine returns with a POPJ P, to DECPN1 until all digits are typed, then to the calling program.

Space can be saved in the pushdown stack by storing the computed digits in the left halves of the locations that contain the jump addresses. This is accomplished in the decimal print routine by making the following substitutions.

```

PUSH P,T+1 → HRLM T+1,(P)
POP P,T → HLRZ T,(P)

```

The routine can handle a 36-bit unsigned integer if the IDIVI T,12 is replaced by

MACRO interprets a number following ↑D as decimal.

```

LSHC    T,↑D35      ;Shift right 35 bits into T+1
LSH     T+1,-1      ;Vacate the T+1 sign bit
DIVI    T,12        ;Divide double length integer by 10

```

Many data processing situations involve searching for information in tables and lists of all kinds. Suppose we wish to find a particular item in a table beginning at location TAB and containing *N* items. Accumulator T contains the item. The right half of A is used to index through the table, while the left half keeps a control count to signal when a search is unsuccessful.

```

MOVSI   A,-N        ;Put -N, 0 in A
CAMN    T,TAB(A)    ;Skip if current item not the one
JRST    FOUND       ;Item found
AOBJN   A,-2        ;Try next item until left count = 0
...     ;Item not in list

```

The location of the item (if found) is indicated by the number in the right half of A (its address is that quantity plus TAB). A slightly different procedure would be

```

HRLZI   A,-N
CAME    T,TAB(A)    ;Skip if current item is the one
AOBJN   A,-1
JUMPL   A,FOUND     ;Jump if left count < 0
...     ;Item not found

```

Locations used for a list can be scattered throughout memory if data is kept in the left half of each location and the right half addresses the next location in the list. The final location is indicated by a zero right half. The following routine finds the last half word item in the list. It is entered at FIND with the first location in the list addressed by the right half of accumulator T. At the end the final item is in T right.

```

FIND:  MOVE  T,(T)      ;Move next item to T
        TRNE  T,777777 ;Skip if AC right = 0
        JRST  .-2
        HLRZS T        ;Move final item to right

```

The following counts the length of the list in accumulator CNT.

```

        MOVEI CNT,0      ;Clear CNT
        JUMPE T,OUT     ;Jump out if T contains 0
        HRRZ  T,(T)     ;Get next address
        AOJA  CNT,-2    ;Count and go back

```

Double Precision Floating Point. The following are straightforward routines for handling double precision floating point arithmetic [§2.6 describes the floating point instructions].

```

DFAD:  UFA  A+1,M+1    ;Sum of low parts to A+2
        FADL A,M       ;Sum of high parts to A, A+1
        UFA  A+1,A+2   ;Add low part of high sum to A+2
        FADL A,A+2     ;Add low sum to high sum
        POPJ P,

DFSB:  DFN  A,A+1     ;Negate double length operand
        PUSHJ P,DFAD  ;Call double floating add
        DFN  A,A+1     ;-(M - AC) = AC - M
        POPJ P,

DFMP:  MOVEM A,A+2    ;Copy high AC operand in A+2
        FMPR A+2,M+1  ;One cross product to A+2
        FMPR A+1,M    ;Other to A+1
        UFA  A+1,A+2  ;Add cross products into A+2
        FMPL A,M      ;High product to A, A+1
        UFA  A+1,A+2  ;Add low part to cross sum in A+2
        FADL A,A+2    ;Add low sum to high part of product
        POPJ P,

```

A double precision division is of the form

$$\frac{A}{B} = \frac{a + c \times 2^{-27}}{b + d \times 2^{-27}}$$

Using the relationship

$$A/b = q + r \times 2^{-27}/b$$

where q and r are the quotient and remainder produced by FDVL, the following routine computes a double length quotient by the algorithm

$$\frac{A}{B} \cong q + \frac{(r - qd) \times 2^{-27}}{b}$$

which gives a result correct to the next-to-last bit in the low order half.

DFDV:	FDVL	A,M	;Get high part of quotient
	MOVN	A+2,A	;Copy negative of quotient in A+2
	FMPR	A+2,M+1	;Multiply by low part of divisor
	UFA	A+1,A+2	;Add remainder
	FDVR	A+2,M	;Divide sum by high part of divisor
	FADL	A,A+2	;Add result to original quotient
	POPJ	P,	

2.12 INPUT-OUTPUT

The input-output instructions govern all transfers of data to and from the peripheral equipment, and also perform many operations within the processor. An instruction in the in-out class is designated by 111 in bits 0–2, *ie* its left octal digit is 7. Bits 3–9 address the device that is to respond to the instruction. The format thus allows for 128 codes, two of which, 000 and 004 respectively, address the processor and priority interrupt, and are used for the console and time share hardware as well. A chart in Appendix A lists all devices for which codes have been assigned, and gives their mnemonics and DEC option numbers.

Bits 13–35 are the same as in all other instructions: they are the *I*, *X*, and *Y* parts, which are used to calculate an effective address, set of conditions, or mask to be used in the execution of the instruction. The remaining bits, 10–12, select one of the following eight IO instructions.

NOTE

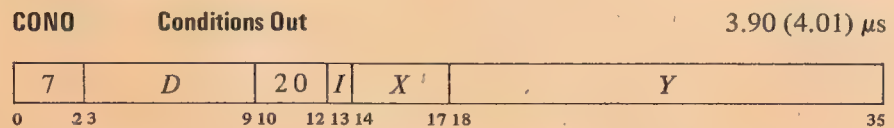
All instructions described in the remainder of this manual are in-out instructions, which cannot be executed in user programs unless the User In-out flag is set. If an in-out instruction appears in a user program while User In-out is clear, it does not perform the functions given for it in the instruction description. Instead it stores its own instruction and device codes in bits 0–12 and its effective address *E* in bits 18–35 of unrelocated location 40 (clearing bits 13–17), and then executes the instruction contained in location 41. The latter location is under control of the Monitor [§2.15].

This user restriction will not be mentioned in the instruction descriptions, as it applies to *all* instructions from this point on.

- ▲ Times are given for IO instructions when they occur alone. When two IO instructions are given consecutively, the second often takes longer (refer to the timing chart in Appendix C for details).

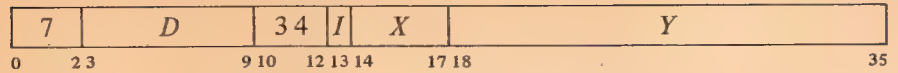
This is identical to UWO trapping [§2.10].

E will always be regarded as being bits 18–35, even though it is actually placed on both halves of the bus and many devices receive the information from the left half.



Set up device *D* with the effective initial conditions *E*. The number of condition bits in *E* that are actually used depends on the device.

CONSO **Conditions In and Skip if One** 4.11 (4.22) μ s

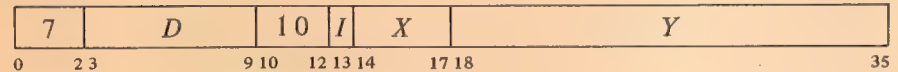


Test the input conditions from device *D* against the effective mask *E*. If any condition bit selected by a 1 in *E* is 1, skip the next instruction in sequence.

If the device supplies more than 18 condition bits, only the right 18 are tested.

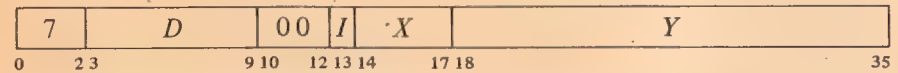
Keeping the pointer in fast memory saves .43 (.34) μ s.

▲ **BLKO** **Block Out** 6.49 (6.71) μ s



Keeping the pointer in fast memory saves .34 μ s. Keeping the instruction and the data block in different memories saves .47 (.36) μ s.

▲ **BLKI** **Block In** 6.49 (6.71) μ s



A block IO instruction is effectively a whole in-out data handling subroutine. It keeps track of the block location, transfers each data word, and determines when the block is finished.

Initially the left half of the pointer contains the negative of the number of words in the block, the right half contains an address one less than that of the first word in the block.

Add 1000001_8 to a pointer in location *E* to increment both halves by one, and place the result back in *E*. Then perform a data IO instruction in the same direction as the block IO instruction, using the right half of the incremented pointer as the effective address. If the given instruction is a BLKO, perform a DATAO; if a BLKI, perform a DATAI.

The remaining actions taken by this instruction depend on whether it is executed as a priority interrupt instruction [§2.13].

◆ *Not as an Interrupt Instruction.* If the addition has caused the count in the left half of the pointer to reach zero, execute the next instruction in sequence. Otherwise skip the next instruction.

◆ *As an Interrupt Instruction.* If the addition has caused the count in the left half of the pointer to reach zero, execute the instruction in the second interrupt location for the channel. Otherwise dismiss the interrupt and return to the interrupted program.

The above eight instructions differ from one another in their total effect, but they are not all different with respect to any given device. A BLKO acts on a device in exactly the same way as a DATAO — the two differ only in counting and other operations carried out within the processor and memory. Similarly, no device can distinguish between a BLKI and a DATAI; and a device always supplies the same input conditions during a CONI, CONSZ or CONSO whether the program tests them or simply stores them.

Hence the eight instructions may be categorized as of four types, represented by the first four instructions described above. Moreover, a complete treatment of the programming of any device can be given in terms of these four instructions, two of which are for input and two for output. The four

exhaust the types of information transfer that occur in the IO system, at least three of which are applicable to any given device. Thus all instruction descriptions in the rest of this manual will be of the CONO, CONI, DATAO and DATAI instructions combined with the various device codes. The discussion of each device will present timing information pertinent to device operation, but no instruction times will be included as they are identical to those given above.

Every device requires initial conditions; these are sent by a CONO, which can supply up to eighteen bits of control information to the device control register. The program can determine the status of the device from up to thirty-six bits of input conditions that can be read by a CONI (but only the right eighteen can be tested by a CONSZ or CONSO). Some input bits simply reflect initial conditions sent by a previous CONO; others are set up by output conditions but are subject to subsequent adjustment by the device; and still others, such as status levels from a tape transport, have no direct connection with output conditions.

Data is moved in and out in characters of various sizes or in full 36-bit words. Each transfer between memory and a device data buffer requires a single DATAI or DATAO. Every device has a CONO and CONI, but it may have only one data instruction unless it is capable of both input and output. *Eg*, the paper tape reader has only a DATAI, the tape punch has only a DATAO, but the teletype has both. (A high speed device, such as a disc file, can be connected to the DF10 Data Channel, which in turn is connected directly to memory by a separate memory bus and handles data automatically. This eliminates the need for the program to give a DATAO or DATAI for each transfer.)

A Typical IO Device. Every device has a 7-bit device selection network, a priority interrupt assignment, and at least two flags, Busy and Done, or some equivalent. The selection network decodes bits 3-9 of the instruction so that only the addressed device responds to signals sent by the processor over the in-out bus. To use the device with the priority interrupt, the program must assign a channel to it. Then whenever an appropriate event occurs in the device, it requests an interrupt on the assigned channel.

The Busy and Done flags together denote the basic state of the device. When both are clear the device is idle. To place the device in operation, a CONO or DATAO sets Busy. If the device will be used for output, the program must give a DATAO that sends the first unit of data — a word or character depending on how the device handles information. When the device has processed a unit of data, it clears Busy and sets Done to indicate that it is ready to receive new data for output, or that it has data ready for input. In the former case the program would respond with a DATAO to send more data; in the latter, with a DATAI to bring in the data that is ready. If an interrupt channel has been assigned to the device, the setting of Done signals the program by requesting an interrupt; otherwise the program must keep testing Done to determine when the device is ready.

All devices function basically as described above even though the number of initial conditions varies considerably. Besides Busy and Done flags, the tape reader and punch have a Binary flag that determines the mode of operation of the device with respect to the data it processes — alphanumeric

The word “input” used without qualification always refers to the transfer of data from the peripheral equipment into the processor; “output” refers to the transfer in the opposite direction.

A DATAI that addresses an output-only device simply clears location *E*. DATAI PI, (code 70044) produces only this effect as the priority interrupt has no data for input. On the other hand a DATAO that addresses an input-only device is a no-op.

When the device code is undefined or the addressed device is not in the system, a DATAO, CONO or CONSO is a no-op, a CONSZ is an absolute skip, a DATAI or CONI clears location *E*.

Busy and Done both set is a meaningless situation.

Occasionally a device with a second code may use a DATAI or DATAO to transmit additional control or maintenance information.

or binary. The teletype has no binary flag, but it has two Busy flags and two Done flags — one pair for input, another for output. A complicated device, such as magnetic tape, may require two device codes to handle the large number of conditions associated with it. Initial conditions for a tape system include a transport address and an actual command the tape control is to perform; input conditions include error flags and transport status levels.

Most IO devices involve motion of some sort, usually mechanical (in a display only the electron beam moves). With respect to mechanical motion there are two types of devices, those that stay in motion and those that do not. Magnetic tape is an example of the former type. Here the device executes a command (such as read, write, space forward) and the done flag indicates when the entire operation is finished. A separate data flag signals each time the device is ready for the program to give a DATAI or DATAO, but the tape keeps moving until an entire record or file has been processed.

Paper tape, on the other hand, stops after each transfer, but the program need not give a new CONO every time. The reader logic is set up so that a DATAI not only reads the data, but also clears Done and sets Busy. Hence if the instruction is given within a critical time, the tape moves continuously and only two CONOs are required for a whole series of transfers: one to start the tape, and one to stop it after the final DATAI.

Other devices operate in one or the other of these two ways but differ in various respects. The tape punch and teletype output are like the reader. Teletype input is initiated by the operator striking a key rather than by the program. The card reader reads an entire card on a single CONO, with a DATAI required for each column. The DECTape stays in motion, and the program must give a CONO to stop it or it will go all the way to the end zone.

Readin Mode

This mode of processor operation provides a means of placing information in memory without relying on a program already in memory or loading one word at a time manually. Its principal use is to read in a short loader program which is then used for loading other information. A loader program should ordinarily be used rather than readin mode, as a loader can check the validity of the information read.

Pressing the readin key on the console activates readin mode by starting the processor in a special hardware sequence that simulates a DATAI followed by a series of BLKI instructions, all of which address the device whose code is selected by the readin device switches on the small panel at the left of the paper tape reader. Various devices can be used, and for each there are special rules that must be followed. But the readin mode characteristics of any particular device are treated in the discussion of the device. Here we are concerned only with the general characteristics.

The information read is a block of data (such as a loader program) preceded by a pointer for the BLKI instructions. The left half of the pointer contains the negative of the number of words in the block, the right half contains an address one less than that of the location that is to receive the first word.

To read in, the operator must set up the device he is using, set its code into the readin device switches, and press the readin key. The processor places the device in operation, brings the first word (the pointer) into location 0, and then reads the data block, placing the words in the locations specified by the pointer. Data can be placed anywhere in memory (including fast memory) except in location 0. The operation affects none of memory except location 0 and the block area.

Upon completing the block, the processor halts only if the single instruction switch is on. Otherwise it leaves readin mode, and begins normal operation by executing the last word in the block as an instruction.

Console Data Transfers

Neither the processor nor the priority interrupt system require all four types of IO instructions, so the program can make use of their device codes for communicating with the console.

DATAI APR, Data In, Console

0	7 0 0 0 4	I	X	Y	35
	12 13 14		17 18		

MACRO also recognizes the mnemonic RSW (Read Switches) as equivalent to DATAI APR.

Read the contents of the console data switches into location *E*.

DATAO PI, Data Out, Console

0	7 0 0 5 4	I	X	Y	35
	12 13 14		17 18		

Unless the console MI program disable switch is on, display the contents of location *E* in the console memory indicators and turn on the triangular light beside the words PROGRAM DATA just above the indicators (turn off the light beside MEMORY DATA).

Once the indicators have been loaded by the program, no address condition selected from the console [§2.16] can load them until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key.

2.13 PRIORITY INTERRUPT

Most in-out devices must be serviced infrequently relative to the processor speed and only a small amount of processor time is required to service them, but they must be serviced within a short time after they request it. Failure to service within the specified time (which varies among devices) can often

result in loss of information and certainly results in operating the device below its maximum speed. The priority interrupt is designed with these considerations in mind, *ie* the use of interruptions in the current program sequence facilitates concurrent operation of the main program and a number of peripheral devices. The hardware also allows conditions internal to the processor to signal the program by requesting an interrupt.

Interrupt requests are handled through seven channels arranged in a priority chain, with assignment of devices to channels entirely at the discretion of the programmer. To assign a device to a channel, the program sends the number of the channel to the device control register as part of the conditions given by a CONO (usually bits 33–35). Channels are numbered 1–7, with 1 having the highest priority; a zero assignment disconnects the device from the interrupt channels altogether. Any number of devices can be connected to a single channel, and some can be connected to two channels (*eg* a device may signal that data is ready on one channel, that an error has occurred on another).

Interrupt Requests. When a device requires service it sends an interrupt request signal over the in-out bus to its assigned channel in the processor. If the channel is on, the processor accepts the request at the next memory access unless the processor is either starting an interrupt on any channel or holding an interrupt on the same channel. The request signal is a level, so it remains on the bus until turned off by the program (CONO, DATAO or DATAI). Thus if a request is not accepted because of the conditions given above, it will be accepted when those conditions no longer hold. A single channel will shut out all others of lower priority if every time its service routine dismisses the interrupt, a device assigned to it is already waiting with another request. The program can usually trigger a request from a device but delay its acceptance by turning on the channel later.

Starting an Interrupt. After a request is accepted the channel must wait for the interrupt to start. No interrupts can be started unless the priority interrupt system is active. Furthermore, the processor cannot start an interrupt if it is already holding an interrupt on a channel with priority higher than those on which requests have been accepted (in other words if the current program is a higher priority interrupt routine). If there is a higher priority channel waiting, the processor stops the current program to start an interrupt on the waiting channel that has highest priority. The interrupt starts following the retrieval of an instruction, following the retrieval of an address word in an effective address calculation (including the second calculation using the pointer in a byte instruction), or following a transfer in a BLT. When an interrupt starts, PC points to the interrupted instruction, so that a correct return can later be made to the interrupted program.

Two memory locations are assigned to each channel: unrelocated locations $40 + 2N$ and $41 + 2N$, where N is the channel number. Channel 1 uses locations 42 and 43, channel 2 uses 44 and 45, and so on to channel 7 which uses 56 and 57. The processor starts an interrupt on channel N by executing the instruction in location $40 + 2N$.

An instruction executed by the interrupt hardware in response to an interrupt request is referred to elsewhere in this manual as being executed “as an interrupt instruction”. Some instructions, when so executed, perform

Interrupt locations for a second processor are $140 + 2N$ and $141 + 2N$.

different functions than they do when executed in other circumstances. And the difference is not due merely to being executed in an interrupt location or in response (by the program) to an interrupt. To be an interrupt instruction, an instruction must be executed by the interrupt hardware, in location $40 + 2N$ or $41 + 2N$, because of a request on channel N . §2.12 describes the two ways a BLKO is performed. If a BLKO is contained in an interrupt routine called by a JSR, it is *not* executed “as an interrupt instruction” even if the routine is stored within the interrupt locations. There are two categories of interrupt instructions.

◆ *Non-IO Instructions.* After executing a non-IO interrupt instruction, the processor holds an interrupt on the channel and returns control to PC. Hence the instruction is usually a jump to a service routine. If the processor is in user mode and the interrupt instruction is a JSR, JSP, PUSHJ, JSA or JRST, the processor leaves user mode (the Monitor thus handles all interrupt routines [§2.15]).

If the interrupt instruction is not a jump, the processor continues the interrupted program while holding an interrupt — in other words it now treats the interrupted program as an interrupt routine. *Eg* the instruction might just move a word to a particular location. Such procedures are usually reserved for maintenance routines or very sophisticated programs.

◆ *Block or Data IO Instructions.* One or the other of two actions can result from executing one of these as an interrupt instruction.

If the instruction in $40 + 2N$ is a BLKI or BLKO and the block is not finished (*ie* the count does not cause the left half of the pointer to reach zero), the processor holds and immediately dismisses an interrupt on the channel, and returns to the interrupted program. The same action results if the instruction is a DATAI or DATAO.

If the instruction in $40 + 2N$ is a BLKI or BLKO and the count does reach zero, the processor continues to start the interrupt by executing the instruction in location $41 + 2N$. This *cannot* be an IO instruction and the actions that result from its execution as an interrupt instruction are those given above for non-IO instructions.

CAUTION

The execution, as an interrupt instruction, of a CONO, CONI, CONSO or CONSZ in location $40 + 2N$ or *any* IO instruction in location $41 + 2N$ hangs up the processor.

Dismissing an Interrupt. Automatic dismissal of an interrupt occurs only in a DATAI or DATAO, or in a BLKI or BLKO with an incomplete block. Following any non-IO interrupt instruction, the processor holds an interrupt until the program dismisses it, even if the interrupt routine is itself interrupted by a higher priority channel. Thus interrupts can be held on a number of channels simultaneously, but from the time an interrupt is started until it is dismissed, no interrupt can be started on that channel or any channel of lower priority (requests, however, can be accepted on lower priority channels).

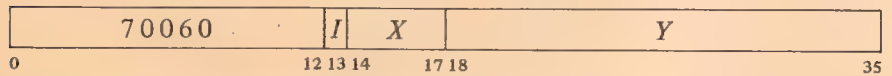
A routine dismisses the interrupt by using a JEN (JRST 12,) to return to the interrupted program (the interrupt system must be active when the JEN is given). This instruction restores the channel on which the interrupt is being held, so it can again accept requests, and interrupts can be started on it and lower priority channels. JEN also restores the flags, whose states were saved in the left half of the PC word if the routine was called by a JSR, JSP, or PUSHJ [§2.9]. If flag restoration is not desired, a JRST 10, can be used instead.

CAUTION

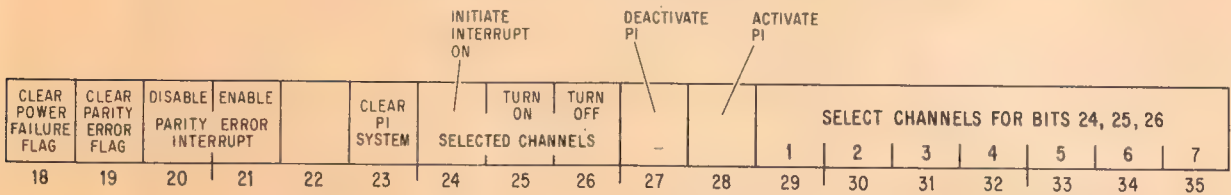
An interrupt routine must dismiss the interrupt when it returns to the interrupted program, or its channel and all channels of lower priority will be disabled, and the processor will treat the new program as a continuation of the interrupt routine.

Priority Interrupt Conditions. The program can control the priority interrupt system by means of condition IO instructions. The device code is 004, mnemonic PI.

CONO PI, Conditions Out, Priority Interrupt



Perform the functions specified by *E* as shown (a 1 in a bit produces the indicated function, a 0 has no effect).



Notes.

- 20 Prevent the setting of the Parity Error flag from requesting an interrupt on the channel assigned to the processor.
- 21 Enable the setting of the Parity Error flag to request an interrupt on the channel assigned to the processor.
- 23 Deactivate the priority interrupt system, turn off all channels, eliminate all interrupt requests that have already been accepted but are still waiting, and dismiss all interrupts that are currently being held.
- 24 Request interrupts on channels selected by 1s in bits 29–35, and force the processor to accept them even on channels that are off.

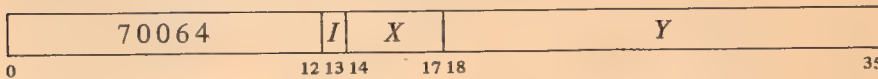
Bits 18–21 are actually for processor conditions [§2.14].

§2.13

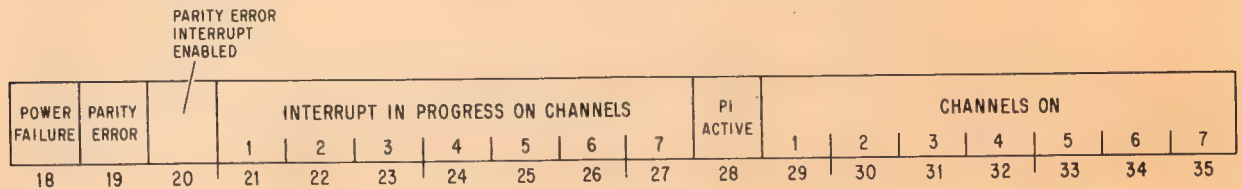
A request is lost if it is made by this means to a channel on which an interrupt is already being held.

- 25 Turn on the channels selected by 1s in bits 29–35 so interrupt requests can be accepted on them.
- 26 Turn off the channels selected by 1s in bits 29–35, so interrupt requests cannot be accepted on them unless made by a CONO PI, with a 1 in bit 24.
- 27 Deactivate the priority interrupt system. The processor can then still accept requests, but it can neither start nor dismiss an interrupt.
- 28 Activate the priority interrupt system so the processor can accept requests and can start, hold and dismiss interrupts.

CONI PI, Conditions In, Priority Interrupt



Read the status of the priority interrupt (and several bits of processor conditions) into the right half of location *E* as shown.



Notes.

- 18 Ac power has failed. The program should save PC, the flags and fast memory in core, and halt the processor.
The setting of this flag requests an interrupt on the channel assigned to the processor. If the flag remains set for 5 ms, the processor is cleared.
- 19 A word with even parity has been read from core memory. If bit 20 is set, the setting of the Parity Error flag requests an interrupt on the channel assigned to the processor.
- 28 The priority interrupt system is active.

Note that bits 18–20 actually read processor status conditions [§2.14].

Channels that are on are indicated by 1s in bits 29–35; 1s in bits 21–27 indicate channels on which interrupts are currently being held.

Timing. The time a device must wait for an interrupt to start depends on the number of channels in use, and how long the service routines are for devices on higher priority channels. If only one device is using interrupts,

it need never wait longer than the time required for the processor to finish the instruction that is being performed when the request is made. The maximum time can be considered to be about 15 μ s for FDVL, but a ridiculously long shift could take over 35 μ s.

Special Considerations. On a return to an interrupted program, the processor always starts the interrupted instruction over from the beginning. This causes special problems in a BLT and in byte manipulation.

An interrupt can start following any transfer in a BLT. When one does, the BLT puts the pointer (which has counted off the number of transfers already made) back in AC. Then when the instruction is restarted following the interrupt, it actually starts with the next transfer. This means that if interrupts are in use, the programmer cannot use the accumulator that holds the pointer as an index register in the same BLT, he cannot have the BLT load AC except by the final transfer, and he cannot expect AC to be the same after the instruction as it was before.

An interrupt can also start in the second effective address calculation in a two-part byte instruction. When this happens, Byte Interrupt is set. This flag is saved as bit 4 of a PC word, and if it is restored by the interrupt routine when the interrupt is dismissed, it prevents a restarted ILDB or IDPB from incrementing the pointer a second time. This means that the interrupt routine must check the flag before using the same pointer, as it now points to the next byte. Giving an ILDB or IDPB would skip a byte. And if the routine restores the flag, the interrupted ILDB or IDPB would process the same byte the routine did.

Programming Suggestions. The Monitor handles all interrupts for user programs. Even if the User In-out flag is set, a user program generally cannot reference the interrupt locations to set them up. Procedures for informing the Monitor of the interrupt requirements of a user program are discussed in the Monitor manual.

For those who do program priority interrupt routines, there are several rules to remember.

- ◆ No requests can be accepted, not even on higher priority channels, while a break is starting. Therefore do not use lengthy effective address calculations in interrupt instructions.
- ◆ The interrupt instruction that calls the routine must save PC if there is to be a return to the interrupted program. Generally a JSR is used as it saves both PC and the flags, and it uses no accumulator.
- ◆ The principal function of an interrupt routine is to respond to the situation that caused the interrupt. *Eg* computations that can be performed outside the routine should not be included within it.
- ◆ The routine must dismiss the interrupt (with a JEN) when returning to the interrupted program. The flags should be restored.

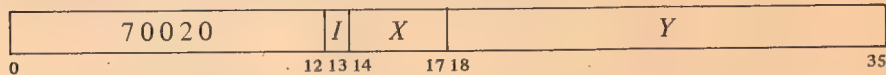
2.14 PROCESSOR CONDITIONS

There are a number of internal conditions that can signal the program by requesting an interrupt on a channel assigned to the processor. Flags for

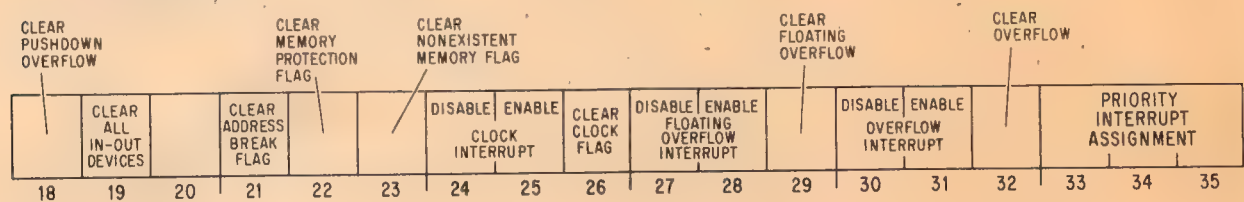
§2.14

power failure and parity error are handled by the condition IO instructions that address the priority interrupt system [§2.13]. The remaining flags are handled by condition instructions that address the processor. Its device code is 000, mnemonic APR or CPA.

CONO APR, Conditions Out, Arithmetic Processor



Perform the functions specified by *E* as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

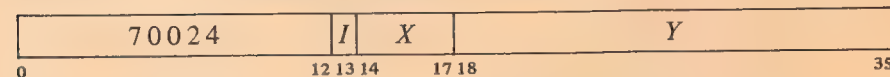


Notes:

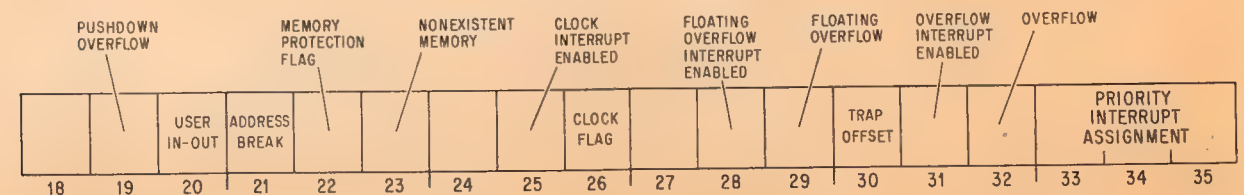
Enabling a particular flag to interrupt means that henceforth the setting of the flag will request an interrupt on the channel assigned (by bits 33-35) to the processor. Disabling prevents the flag from triggering a request.

A 1 in bit 19 produces the IO reset signal, which clears the control logic in all of the peripheral equipment (but affects neither the priority interrupt system, nor the processor flags cleared by this instruction or CONO PI,).

CONI APR, Conditions In, Arithmetic Processor



Read the status of the processor into the right half of location *E* as shown (all interrupt requests are made on the channel assigned to the processor).



Notes.

- ▲ PC bears no relation to the break if the access was requested for a console key function.
- 19 Pushdown Overflow — in a PUSH or PUSHJ the count in AC left reached zero; or in a POP or POPJ the count reached -1. The setting of this flag requests an interrupt.
- ▲ 20 User In-out — even if the processor is in user mode, no instructions are illegal (but protection and relocation still apply) [§2.15].
- 21 Address Break — while the console address break switch was on, the processor requested access to the memory location specified by the address switches and the memory reference was for the purpose selected by the address condition switches as follows:
- The instruction switch was on and access was for retrieval of an instruction (including an instruction executed by an XCT or contained in an interrupt location or a trap for an unimplemented operation) or an address word in an effective address calculation.
- The data fetch switch was on and access was for retrieval of an operand (other than in an XCT).
- The write switch was on and access was for writing a word in memory.
- The setting of this flag requests an interrupt, at which time PC points to the instruction that was being executed or to the one following it.
- ▲ This flag can also be set by an instruction executed from the console while the USER MODE light is on, in which case PC bears no relation to the violation.
- ▲ 22 Memory Protection — a user program attempted to access a memory location outside of its area or to write in a write-protected part of its area and the user instruction was terminated at that time. The setting of this flag requests an interrupt, at which time PC points either to the instruction that caused the violation or to the one following it.
- ▲ PC bears no relation to the unanswered reference if the attempted access originated from a console key function.
- 23 Nonexistent Memory — the processor attempted to access a memory that did not respond within 100 μ s. The setting of this flag requests an interrupt, at which time PC points either to the instruction containing the unanswered reference or to the one following it.
- 26 Clock — this flag is set at the ac power line frequency and can thus be used for low resolution timing (the clock has high long term accuracy). If bit 25 is set, the setting of the Clock flag requests an interrupt.
- 29 Floating Overflow — this is one of the flags saved in a PC word, and the conditions that set it are given at the beginning of §2.9. If bit 28 is set, the setting of Floating Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.
- 30 Trap Offset — the processor is using locations 140-161 for unimplemented operation traps and interrupt locations.
- 32 Overflow — this is one of the flags saved in a PC word, and the conditions that set it are given at the beginning of §2.9. If bit 31 is set, the setting of Overflow requests an interrupt, at which time PC points to the instruction following that in which the overflow occurred.

2.15 TIME SHARING

Without time sharing the system has a single user and the program has no restrictions except those inherent in the hardware: the programmer must stay within the memory capacity, observe the restrictions placed on the use of certain memory locations by the hardware [§1.3], and observe the restrictions on interrupt instructions. Optional hardware can restrict processor operation to permit time sharing by a number of programs. Each user program is run with the processor in user mode, in which the program must operate within an assigned area in core and certain operations may be illegal. A program that runs unrestricted — the Monitor — is responsible for scheduling user programs, servicing interrupts, handling input-output needs, and taking action when control is returned to it from a user program.

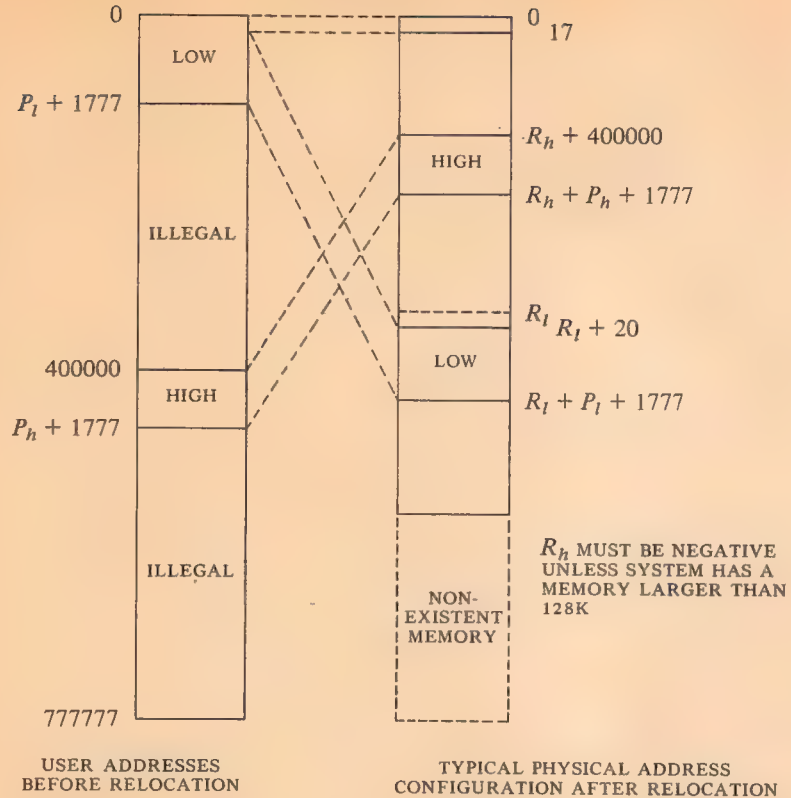
Every user is assigned a core area and the rest of core is protected from him — he cannot gain access to the protected area for either storage or retrieval of information. The assigned area is divided into two parts. The low part is unique to a given user and can be used for any purpose. The high part may be for a single user, or it may be shared by several users. The Monitor can write-protect the high part so that the user cannot alter its contents, *ie* he cannot write anything in it. The Monitor would do this when the high part is to be a pure procedure to be used reentrantly by several users. One high pure segment may be used with any number of low impure segments. The user can request that the Monitor write-protect the high part of a single program, *eg* in order to debug a reentrant program. All users write programs beginning at address 0 for the low part, and beginning usually at 400000 for the high part. The programmed addresses are retained in the object program but are relocated by the hardware to the physical area assigned to the user as each access is made while the program is running.

The size and position of the user area are defined by specifying protection and relocation addresses for the low and high blocks. The protection address determines the maximum address the user can give; any address larger than the maximum is illegal. The relocation address is the address, as seen by the Monitor and the hardware, of the first location in the block. The Monitor defines these addresses by loading four 8-bit registers, each of which corresponds to the left eight bits (18–25) of an address whose right ten bits are all 0.

To determine whether an address is legal its left eight bits are compared with the appropriate protection register, so the maximum user address consists of the register contents in its left eight bits, 1777 in its right ten bits (*ie* it is equal to the protection address plus 1777). Since the set of all addresses begins at zero, a block is always an integral multiple of 1024_{10} (2000_8) locations. Relocation is accomplished simply by adding the contents of the appropriate relocation register to the user address, so the first address in a block is a multiple of 2000. The relative user and relocated address configurations are therefore as illustrated here, where P_l , R_l , P_h and R_h are respectively the protection and relocation addresses for the low and high parts as derived from the 8-bit registers loaded by the Monitor. If the low part is larger than 128K locations, *ie* more than half the maximum memory capacity ($P_l > 400000$), the high part starts at the first location after the low

Note that the relocated low part is actually in two sections with the larger beginning at $R_l + 20$. This is because addresses 0-17 are not relocated, all users having access to the accumulators. The Monitor uses the first sixteen locations in the low user block to store the user's accumulators when his program is not running.

Some systems have only the low pair of protection and relocation registers. In this case the user program is always nonreentrant and the assigned area comprises only the low part.



part (at location $P_l + 2000$). The high part is limited to 128K. If the Monitor defines two parts but does not write-protect the high part, the user has a two-part nonreentrant program.

If the user attempts to access a location outside of his assigned area, or if the high part is write-protected and he attempts to alter its contents, the current instruction terminates immediately, the Memory Protection flag is set (status bit 22 read by CONI APR,), and an interrupt is requested on the channel assigned to the processor [§2.14].

User Programming. The user must observe the following rules when programming on a time shared basis. [Refer to the Monitor manual for further information including use of the Monitor for input-output.]

- ◆ Use addresses only within the assigned blocks for all purposes – retrieval of instructions, retrieval of addresses, storage or retrieval of operands. The low part contains locations with addresses from 0 to the maximum; the high part contains from the greater of 400000 or $P_l + 2000$ to the maximum. Either part can address the other.
- ◆ If the high part is write-protected, do not attempt to store anything in it. In particular do not execute a JSR or JSA into the high part.
- ◆ Use instruction codes 000 and 040-127 only in the manner prescribed in the Monitor manual.
- ◆ Unless User In-out is set do not give any IO instruction, HALT (JRST 4,) or JEN (JRST 12, (specifically JRST 10,)). The program can determine if User In-out is set by examining bit 6 of the PC word stored by JSR, JSP or

The user can actually write any size program: the Monitor will assign enough core for his needs. Basically the user must write a sensible program; if he uses absolute addresses scattered all over memory his program cannot be run on a time shared basis with others.

These instructions are illegal unless User In-out is set.

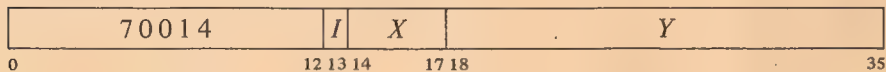
PUSHJ.

The user can give a JRSTF (JRST 2,) but a 0 in bit 5 of the PC word does not clear User (a program cannot leave user mode this way); and a 1 in bit 6 does not set User In-out, so the user cannot void any of the restrictions himself. Note that a 0 in bit 6 will clear User In-out, so a user can discard his own special privileges.

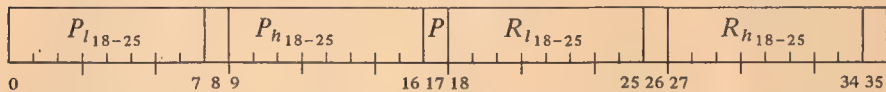
UUOs 001-037 execute normally and are relocated to addresses 40 and 41 in the low block [§2.10].

Monitor Programming. The Monitor must assign the core area for each user program, set up trap and interrupt locations, specify whether the user can give IO instructions, transfer control to the user program, and respond appropriately when an interrupt occurs or an instruction is executed in unrelocated 41 or 61.

Core assignment is made by this instruction.

DATAO APR, Data Out, Arithmetic Processor

Load the protection and relocation registers from the contents of location E as shown, where P_l , P_h , R_l and R_h are the protection and relocation



addresses defined above. If write-protect bit P (bit 17) is 1, do not allow the user to write in the high part of his area.

For a two part nonreentrant program, set $P = 0$. For a one-part nonreentrant program, make $P_h \leq P_l$. If the hardware has only one set of protection and relocation registers, the user area is defined by P_l and R_l , the rest of the word is ignored.

Giving a JRSTF with a 1 in bit 6 of the PC word allows the user to handle his own input-output. The Monitor can also transfer control to the user with this instruction by programming a 1 in bit 5 of the PC word, or it may jump to the user program with a JRST 1, which automatically sets User. The set state of this flag implements the user restrictions.

While User is set, certain instructions are not part of the user program and are therefore completely unrestricted, namely those executed in the interrupt locations (which are not relocated) and in unrelocated trap locations 41 and 61. Illegal instructions and UUO codes 000 and 040-077 are trapped in unrelocated 40; codes 100-127 are trapped in unrelocated 60. BLKI and BLKO can be used in the even interrupt locations, and if there is no overflow, the processor returns to the interrupted user program. JSR should ordinarily be used in the remaining even interrupt locations, in odd interrupt locations following block IO instructions, and in 41 and 61. The JSR clears User and should jump to the Monitor. JSP, PUSHJ, JSA and JRST are acceptable in that they clear User, but the first two require an accumulator

(all accumulators should be available to the user) and the latter two do not save the flags.

After taking appropriate action, the Monitor can return to the user program with a JRSTF or JEN that restores the flags including User and User In-out.



2.16 OPERATION

Most of the controls and indicators used for normal operation of the processor and for program debugging are located on the console operator panel shown here. The indicators are on the vertical part of the panel; in front of them are two rows of two-position keys and switches (keys are momentary contact, switches are alternate action). A key or switch is on or represents a 1 when the front part is down.

The thirty-six switches in the front row and the eighteen at the right in the back row are respectively the data and address switches through which the operator can supply words and addresses for the program and for use in conjunction with the operating keys and switches. The correspondence of switches to bit positions is indicated by the numbers at the bottom row of lights. At the left end of the back row are ten operating switches, which supply continuous control levels to the processor. At their right are ten operating keys, which initiate or terminate operations in the processor. The names of the operating keys and switches appear on the vertical part of the panel below the lights.

Also of interest to the operator is the small panel shown opposite, which is located above the main panel at the left of the tape reader. The upper section of this panel contains a total hours meter and the margin-check controls. The lower section contains the power switch, speed controls for slowing down the program, switches to select the device for reading the program, switches to select the device for reading mode (lower part in represents a 1), and four additional operating switches. The normal position for these last four is with the upper part in; in this position FM ENB (fast memory enable) is on, the others are all off.

Indicators

When any indicator is lit the associated flipflop is 1 or the associated function is true. Some indicators display useful information while the processor is running, but many change too frequently and can be discussed only in terms of the information they display when the processor is stopped. The program can stop the processor only at the completion of the HALT instruction; the operator can stop it at the end of

§2.16

every instruction or memory reference, or for maintenance purposes, after every step in any operation that uses the shift counter (shifting, multiplication, division, byte manipulation).

Of the long rows of lights at the right on the operator panel, the top row displays the contents of PC, the middle row displays the instruction being executed or just completed, and the bottom row are the memory indicators. The right half of the middle row displays MA, the left half displays IR [see page 1-2]. In an IO instruction the left three instruction lights are on, the remaining instruction lights and the left AC light are the device code, and the remaining AC lights complete the instruction code. The I, index and MA lights change with each indirect reference in an effective address calculation; at the end of an instruction I is always off.

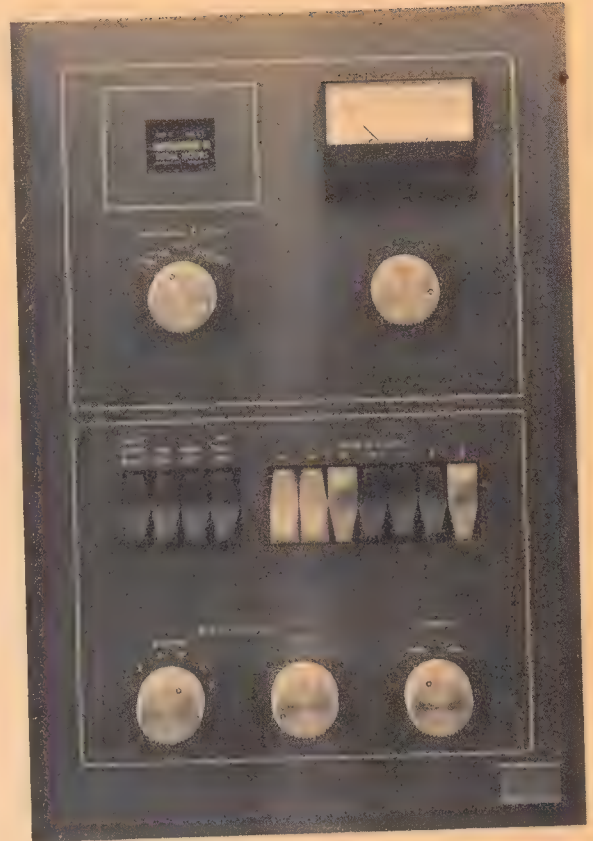
Above the memory indicators appear two pairs of words, PROGRAM DATA and MEMORY DATA. If the triangular light beside the former pair is on, the indicators display a word supplied by a DATAO PI; if any other data is displayed the light beside MEMORY DATA is on instead. While the processor is running the physical addresses used for memory reference (the relocated address whenever relocation is in effect) are compared with the contents of the address switches. Whenever the two are equal the contents of the addressed location are displayed in the memory indicators. However, once the program loads the indicators, they can be changed only by the program until the operator turns on the MI program disable switch, executes a key function that references memory, or presses the reset key (see below).

The four sets of seven lights at the left display the state of the priority interrupt channels [see pages 2-74 and 2-75]. The PI ACTIVE lights indicate which channels are on. The IOB PI REQUEST lights indicate which channels are receiving request signals over the in-out bus; the PI REQUEST lights indicate channels on which the processor has accepted requests. Except in the case of a program-initiated interrupt, a REQUEST light can go on only if the corresponding ACTIVE light is on. The PI IN PROGRESS lights indicate channels on which interrupts are currently being held; the channel that is actually being serviced is the lowest-numbered one whose light is on. When a PROGRESS light goes on, the corresponding REQUEST goes off and cannot go on again until PROGRESS goes off when the interrupt is dismissed.

At the left end of the panel are a power light and these control indicators.

RUN

The processor is in normal operation with one instruction following another. When the light goes off, the processor stops.



Above: Margin Check and Maintenance Panel
Opposite: Console Operator Panel

NOTE: If a REQUEST light stays on indefinitely with the associated PROGRESS light off and PC is static, check the PI CYC light on the indicator panel at the top of bay 2. If it is on, a faulty program has hung up the processor. Press STOP.

PI ON

The priority interrupt system is active so interrupts can be started (this corresponds to CONI PI, bit 28).

USER MODE

The processor is in user mode (this corresponds to bit 5 of a PC word).

If RUN and PROGRAM STOP are both on, the processor is probably in an indirect address loop. Press STOP.

PROGRAM STOP

IR now contains a HALT instruction. If RUN is off, MA displays an address one greater than that of the location containing the instruction that caused the halt, and PC displays the jump address (the location from which the next instruction will be taken if the operator presses the continue key).

MEMORY STOP

The processor has stopped at a memory reference. This can be due to single cycle operation, satisfaction of an address condition selected at the console, reference to a nonexistent memory location, or detection of a parity error.

The remaining processor lights are on the indicator panels at the tops of the bays [illustrated on page C8]. Bay 2 displays AR, BR and MQ, the output of the AR adder, and the parity buffer which receives every word transmitted over the memory bus. The RL and PR lights at the lower right display the relocation and protection registers for the low part of the area assigned to a user program and the left eight bits of the relocated address for that part. The remaining lights are for maintenance.

The upper four rows on the bay 1 panel include the indicators for reader, punch and teletype, which are described in Chapter 3. The bottom row displays the information on the data lines in the IO bus. The AR lights at the upper right are the flags – FXU is Floating (exponent) Underflow, DCK is No Divide (divide check). OV COND is the condition that the 0 and 1 carries are different, *ie* the condition that indicates overflow. The Byte Interrupt flag is BYF6 in the MISC lights in the top row; User In-out is IOT USER in the EX lights at the center of the panel. The CPA lights in the top row and the five lights under them at the left are the processor conditions – PDL OV is Pushdown (list) Overflow. The AS= lights in the middle row indicate when the (relocated) core memory address or the fast memory address is the same as the address switches. The remaining lights are for maintenance.

The panels on the two types of memories are shown on page C9. These are almost exclusively for maintenance, and (as with most of the lights on the processor bays) if the operator must use them he should consult the maintenance manual and the flow charts. The ACTIVE lights indicate which processor currently has access to the memory.

Operating Keys

Each key except STOP turns on one of the key indicators at the upper right on the bay 2 panel. These are for flipflops that allow the key functions to be repeated automatically and also allow certain of them to be synchronized to the processor time chain so they can be performed while the processor is running.

READ IN

Clear all IO devices and all processor flags including User; turn on the RIM light in the upper right on bay 1 and the KEY RDI light in the upper right on bay 2. Execute DATAI $D,0$ where D is the device code specified by the readin device switches on the small panel at the left of the reader. Then execute a series of BLKI $D,0$ instructions until the left half of location 0 reaches zero, at which time turn off RIM and KEY RDI. Stop only if the single instruction switch is on; otherwise turn on RUN and execute the last word read as an instruction. [*For information on the data format refer to page 2-72.*]

START

Load the contents of the address switches into PC, turn on RUN, and begin normal operation by executing the instruction at the location specified by PC.

This key function does not disturb the flags or the IO equipment; hence if USER MODE is lit a user program can be started.

CONT (Continue)

Turn on RUN (if it is off) and begin normal operation in the state indicated by the lights.

STOP

Turn off RUN so the processor stops before beginning the next instruction. Thus the processor usually stops at the end of the current instruction, which is displayed in the lights. However, if a key function that can be performed while RUN is on has been synchronized, the processor performs that function before stopping. In either case PC points to the next instruction.

If the processor does not reach the end of the instruction within 100 μ s, inhibit further effective address calculation — it is assumed the processor is caught in an indirect addressing loop. Pressing CONT when the processor is stopped in an address loop causes it to start the same instruction over.

RESET

Clear all IO devices and clear the processor including all flags. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on duplicate the action of the STOP key before clearing.

If RUN is on, pressing this key has no effect.

The rightmost device switch is for bit 9 of the instruction and thus selects the least significant octal digit (which is always 0 or 4) in the device code.

CAUTION

Do not initiate any other key function while RIM is on. If read in must be stopped (eg because of a crumpled tape), press RESET (see below).

If RUN is on, pressing this key has no effect.

If STOP will not stop the processor, pressing this key will.

Note that an instruction executed from the console can alter the processor state just like any instruction in the program: it can change PC by jumping or skipping, alter the flags, or even cause a non-existent-memory stop.

XCT

Execute the contents of the data switches as an instruction without incrementing PC. If RUN is on, insert this instruction between two instructions in the program. Inhibit priority interrupts during its execution to guarantee that it will be finished.

If USER MODE is lit all user restrictions apply to an instruction executed from the console.

NOTE

The remaining key functions all reference memory. They use an absolute address and all of memory is available to them; in other words protection and relocation are not in effect even if USER MODE is lit. However they can set such flags as Address Break and Nonexistent Memory.

EXAMINE THIS

Display the contents of the address switches in the MA lights and the contents of the location specified by the address switches in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

EXAMINE NEXT

Add 1 to the address displayed in the MA lights and display the contents of the location specified by the incremented address in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

DEPOSIT

Deposit the contents of the data switches in the location specified by the address switches. Display the address in the MA lights and the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA). If RUN is on, insert this function between two instructions in the program.

If RUN is on, pressing this key has no effect.

DEPOSIT NEXT

Add 1 to the address displayed in the MA lights and deposit the contents of the data switches in the location specified by the incremented address. Display the word deposited in the memory indicators. Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA).

CAUTION

Never press two keys simultaneously as the processor may attempt to perform both functions at once.

Operating Switches

Whenever the processor references memory at the location specified by the address switches (relocated if USER MODE is on), the contents of that location are displayed in the memory indicators (unless the light beside PROGRAM DATA is on). The group of five switches at the left of the keys allows the operator to make the processor halt or request an interrupt when reference is made to the specified location *in core memory* for a particular purpose (no action is produced by fast memory reference). The purpose is selected by the three address condition switches. INST FETCH selects the condition that access is for retrieval of an instruction (including an instruction executed by an XCT or contained in an interrupt location or a trap for an unimplemented operation) or an address word in an effective address calculation. DATA FETCH selects access for retrieval of an operand (other than in an XCT). WRITE selects access for writing in memory. Whenever reference to the specified location satisfies any selected address condition, the processor performs the action selected by the other two switches. ADR STOP halts the processor with MEMORY STOP on (PC points to the instruction that was being executed, or if the MC WR light on bay 2 is on, PC may point to the one following it); ADR BREAK turns on the CPA ADR BRK light (Address Break flag, CONI APR, bit 21) on bay 1, requesting an interrupt on the processor channel.

The description of each switch relates the action it produces while it is on.

SING INST

Whenever the processor is placed in operation, clear RUN so that it stops at the end of the first instruction. Hence the operator can step through a program one instruction at a time, by pressing START for the first one and CONT for subsequent ones. Each time the processor stops, the lights display the same information as when STOP is pressed.

CLK FLAG (Clock flag) on bay 1 is held off to prevent clock interrupts while SING INST is on. Otherwise interrupts would occur at a faster rate than the instructions.

SING INST will not stop the processor if a hangup prevents it from getting to the end of an instruction. Use STOP or RESET.

SING CYCLE

Whenever the processor is placed in operation, stop it with MEMORY STOP on at the end of the first *core memory* reference. Hence the operator can step through a program one memory reference at a time, by pressing START for the first one and CONT for subsequent ones. To determine what information is displayed in the lights, consult the flow charts.

AC and index register references can be included by turning off the FM ENB switch (see below).

To stop at AC and index register references, turn off the FM ENB switch (see below).

If IGN is on (it displays a signal from the memory), parity errors are not detected and no stop can occur.

PAR STOP

Stop with MEMORY STOP on at the end of any memory reference in which even parity is detected in a word read. A parity stop is indicated by the following: CPA PAR ERR (Parity Error flag) on bay 1 is on; and among the PAR lights in the bottom row on bay 2, IGN (ignore parity) and ODD are off, STOP is on, and BIT displays the parity bit for the word in the parity buffer at the left.

NXM STOP

Stop with MEMORY STOP on if a memory reference is attempted but the memory does not respond within 100 μ s. This type of stop is indicated by CPA NXM FLAG (Nonexistent Memory flag) on bay 1 being on.

The key function is repeated once after REPT is turned off, but this is noticeable only with very long repeat delays.

The end of a key function is equivalent to completion of all processor operations associated with the function only for read in, examine, examine next, deposit, and deposit next. In other cases the processor continues in operation. Eg the execute function is finished once the instruction to be executed is set up internally, but the processor then executes that instruction. Hence when using speed range 6, the operator must be careful not to allow the key function to restart before the processor is really finished.

REPT

If any key (except STOP) is pressed, then every time the key function is finished, wait a period of time determined by the setting of the speed control and repeat the given key function. If CONT is pressed and no switch is on that would stop the program (eg SING INST, SING CYCLE), then continue following the repeat delay whenever a HALT instruction is executed. Continue to repeat the key function until RESET is pressed or REPT is turned off.

The speed control includes a six-position switch that selects the delay range and a potentiometer for fine adjustment within the range. Delay ranges are as follows.

<i>Position</i>	<i>Range</i>
1	270 ms to 5.4 seconds
2	38 ms to 780 ms
3	3.9 ms to 78 ms
4	390 μ s to 7.8 ms
5	27 μ s to 540 μ s
6	2.2 μ s to 44 μ s

MI PROG DIS

Turn on the triangular light beside MEMORY DATA (turn off the light beside PROGRAM DATA) and inhibit the program from displaying any information in the memory indicators. The indicators will thus continually display the contents of locations selected from the console.

REPT BYP

If REPT is on, trigger the repeat delay at the *beginning* of the key function. Hence the function is repeated even if it does not run to completion.

FM ENB

This switch is left on for normal operation with a fast memory. Turning it *off* (lower part in) substitutes the first sixteen core locations for the fast memory. The switch is left off if there is no fast memory, and it can be used to allow stopping or breaking at fast memory references.

SHIFT CNTR MAINT

Stop before each step in any shift operation. Pressing CONT resumes the operation. Once a shift has been stopped, the processor will continue to stop at each step throughout the rest of the given shift operation even if the switch is turned off.

At the right end of panel 1J behind the bay doors are two toggle switches. FP TRP causes the floating point and byte manipulation instructions (codes 130-177) to trap to locations 60-61. MA TRP OFFSET moves the trap and interrupt locations to 140-161 for a second processor connected to the same memory.

Inside each memory bay are switches for selecting the memory number and interleaving memories. Also in the memory are a power switch, a restart pushbutton, and a switch for single step operation (these three are located on the indicator panel for the MB10 memory).

3

Basic In-out Equipment

The PDP-10 contains three in-out devices as standard equipment: tape reader, tape punch, and teletype. These devices are used principally for communication between computer and operator using a paper medium, tape or form paper.

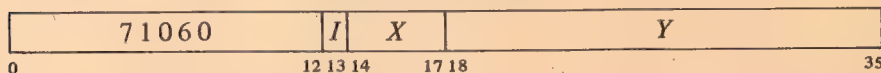
The punch supplies output in the form of 8-channel perforated paper tape in either of two modes. In alphanumeric mode, 8-bit characters are processed; in binary mode, 6-bit characters. The information punched in the tape can be brought into memory by the tape reader, which handles characters in the same two modes.

The program can type out characters on the teletype and can read characters that have been typed in at the keyboard. This device has the slowest transfer rate of any, but it provides a convenient means of man-machine interaction.

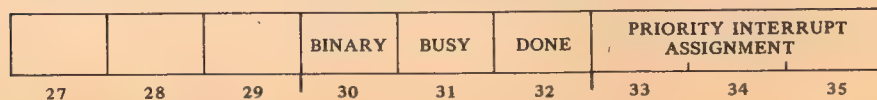
3.1 PAPER TAPE READER

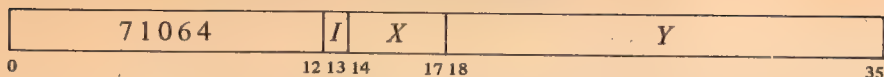
The reader processes 8-channel perforated paper tape photoelectrically at a speed of 300 lines per second. The device can operate in alphanumeric or binary mode, as specified by the 0 or 1 state respectively of the Binary flag. In alphanumeric a single tape-moving command reads all eight channels from the first line encountered. In binary the device reads six channels from the first six lines in which hole 8 is punched and assembles the information into a 36-bit word. The interface contains a 36-bit buffer from which all data is retrieved by the processor. The reader device code is 104, mnemonic PTR.

CONO PTR, Conditions Out, Paper Tape Reader

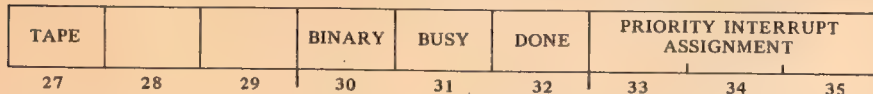


Set up the reader control register according to bits 30-35 of the effective conditions *E* as shown (a 1 in a flag bit sets the flag, a 0 clears it).

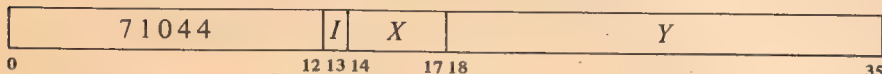


CONI PTR, Conditions In, Paper Tape Reader

Read the status of the reader into bits 27 and 30–35 of location *E* as shown.



Placing the tape in motion sets the Tape flag and it remains set as long as the tape is in the read head. A 0 in bit 27 indicates that the last time an attempt was made to read, the reader was out of tape.

DATAI PTR, Data In, Paper Tape Reader

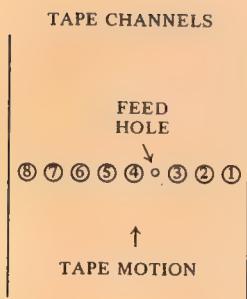
Transfer the contents of the reader buffer into location *E*. Clear Done and set Busy.

Setting Busy clears the reader buffer, sets the Tape flag (if it is not already set) and places the reader in operation. If Binary is clear, all eight channels from the first line on tape are read into bits 28–35 of the buffer with channel 1 corresponding to bit 35 (the presence of a hole produces a 1 in the buffer). If Binary is set, the device reads only channels 1–6, but it reads the first six lines encountered in which channel 8 is punched (lines without a hole in channel 8 are skipped) and assembles them into a full word in the buffer. The first line is at the left in the word and channel 1 corresponds to the rightmost bit in each 6-bit byte.

After the specified number of lines has been read, the reader clears Busy and sets Done, requesting an interrupt on the assigned channel. A DATAI brings the data into memory and also causes the reader to continue in operation. The programmer must give a CONO to clear Busy if he does not want the reader to move the tape after the final DATAI is given.

If the tape runs out or malfunctions while a read operation is in progress, the Tape flag is cleared and the reader shuts down.

Timing. At 300 lines per second the reader takes 3.33 ms per alphanumeric character, 20 ms per binary word if the binary characters are contiguous. After Done is set, the program has 1.6 ms to give a DATAI and keep the tape in continuous motion. Waiting longer causes the reader to shut down for 40 ms. Thus start-stop operation is limited to 25 lines per second.



EXAMPLES. This program reads ten binary words (60 lines) from paper tape and stores them in memory beginning at location 4000. The block pointer is kept in accumulator PNT.

```

      MOVE   PNT,[IOWD 12,4000]   ;Put pointer in PNT
      CONO   PTR,60               ;Set up reader
NEXT:  CONSO  PTR,10              ;Watch Done
      JRST   .-1
      BLKI   PTR,PNT             ;Word ready, get it
      JRST   .+2                 ;Got all data
      JRST   NEXT                ;Go back for next word
      :
      :

```

If instead of just waiting we wish to continue our program while the data is coming in, we can use the priority interrupt. The following uses channel 4 and signals the main program that the data is ready by setting bit 35 of accumulator F.

```

      MOVE   17,[BLKI PTR,[IOWD 12,4000]]
      MOVEM  17,50                ;Set up 50 and 51 for channel 4
      MOVE   17,[JSR DONE]
      MOVEM  17,51
      CONO   PTR,64               ;Set up reader on channel 4
      CONO   PI,12210            ;Clear PI, then activate it and turn on
                                   ;channel 4
      :                           ;Continue program
      :
      TRZN   F,1                 ;Check if data ready when needed
      JRST   .-1                 ;Wait if necessary
      :
      :
DONE:  0                               ;Interrupt routine, block done
      CONO   PTR,0               ;Stop tape
      TRO    F,1                 ;Set F bit 35
      JEN    @DONE               ;Dismiss and restore flags

```

Operation. Tapes must be uncoiled and opaque. The reader is located just above the console operator panel. To load it, place the fanfold tape stack vertically in the bin at the right, oriented so that the front end of the tape is nearer the read head and the feed holes are away from you. Lift the gate, take three or four folds of tape from the bin, and slip the tape into the reader from the front. Carefully line up the feed holes with the sprocket teeth to avoid damaging the tape, and close the gate. Make sure that the part of the tape in the left bin is placed to correspond to the folds, otherwise it will not stack properly. If the program requires that the Tape flag be set and it is not, briefly press the white feed button located on the face of the reader. After the program has finished reading the tape, run out the remaining trailer by pressing the feed button.

Indicators for the reader are on the panel at the top of bay 1 (the panel is

pictured in Appendix C). The paper tape reader lights in the second row from the bottom display the contents of the buffer. The PI assignment and flags are displayed in the PTR lights in the middle of the third row (EOT is the Tape flag). The remaining PTR lights are for maintenance.

Readin Mode

The only requirement (beyond those given in §2.12) for readin mode with paper tape is that the data must be in binary (hole 8 punched). To select the reader in the readin device switches, turn on the third from the left and the last on the right (104).

The program below is the RIM10B Loader, which is brought into the accumulators in readin mode, and then continues to read any number of blocks of binary data from the same tape. The tape is formatted as a series of blocks separated by a half-dozen lines of blank tape (tape with only feed holes punched). The first block is the loader in readin format. The rest of the tape contains any number of data blocks and ends with a transfer block. Each data block contains any number of words of program data, preceded by a standard IO block pointer for the data only, and followed by a checksum, which is the sum of all the data words and the pointer. It is recommended that the number of data words per block be limited to twenty for ease in repositioning the tape in case of error. The transfer block is a JRST to the starting location of the program, followed by a throw-away word to stop the reader.

This loader is written for minimum size and is quite complex. Do not approach it as a simple programming example.

```

XWD      -16,0          ;1410 words starting at location 1
ST:      CONO   PTR,60  ;Set up reader binary
ST1:     HRRI   A,RD+1  ;Put RD+1 in Y part of A
RD:      CONSO  PTR,10  ;Watch Done
          JRST  .-1
          DATAI PTR,@TBL1-RD+1(A) ;First and last words in
                                ;ADR, data in block
          XCT   TBL1-RD+1(A) ;TBL1+2 first word, +1 data,
                                ;+0 checksum
          XCT   TBL2-RD+1(A) ;TBL2+2 JRST, +1 data, +0
                                ;bad checksum
A:        SOJA   A,      ;RD+1 first word, RD data, RD-1
                                ;last word
TBL1:     CAME   CKSM,ADR ;Compare computed checksum with
                                ;one read
          ADD   CKSM,1(ADR) ;Add word read to checksum
          SKIPL CKSM,ADR ;Put first word in CKSM, skip if
                                ;pointer
TBL2:     JRST  4,ST     ;Halt if checksum bad
          AOBJN ADR,RD   ;If data done, go to A; otherwise wait
                                ;for next word
ADR:      JRST  ST1     ;Read in. executes this. First and last
                                ;word of each block also put here

CKSM=ADR+1

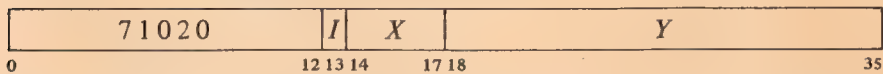
```


The processor halts if a computed checksum does not agree with the tape. To reread a block, move the tape back to the preceding blank area and press the continue key. A halt following the transfer block is not an error — many programs begin by halting.

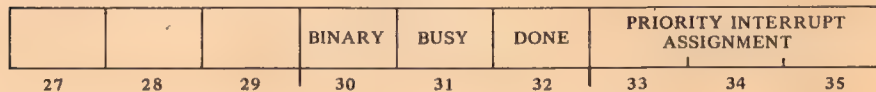
3.2 PAPER TAPE PUNCH

The punch perforates 8-channel tape at speeds up to 50 lines per second. It can operate in alphanumeric or binary mode, as specified by the 0 or 1 state respectively of the Binary flag; but in either mode a single tape-moving command punches only one line. Alphanumeric mode punches an 8-bit character supplied by the program; binary mode always punches channel 8, never punches channel 7, and punches a 6-bit character in the remaining channels. The interface contains an 8-bit buffer that receives data from the processor. The punch device code is 100, mnemonic PTP.

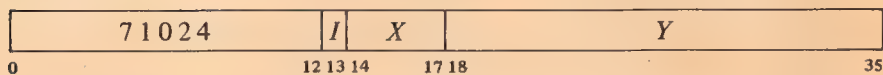
CONO PTP, Conditions Out, Paper Tape Punch



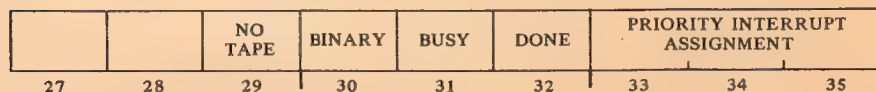
Set up the punch control register according to bits 30–35 of the effective conditions *E* as shown (a 1 in a flag bit sets the flag, a 0 clears it).



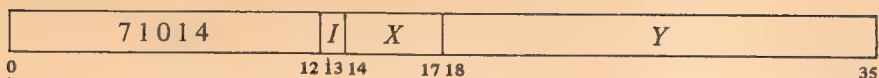
CONI PTP, Conditions In, Paper Tape Punch



Read the status of the punch into bits 29–35 of location *E* as shown.



A 1 in bit 29 indicates that the punch is out of tape.

DATAO PTP, Data Out, Paper Tape Punch

Load the contents of bits 28–35 of location *E* into the punch buffer. Clear Done and set Busy.

A CONO need be given only to change Binary or the PI assignment; DATAO sets Busy while loading the buffer. Setting Busy places the punch in operation. If Binary is clear, one line is punched in tape from bits 28–35 of the buffer with bit 35 corresponding to channel 1 (a 1 in the buffer produces a hole in the tape). If Binary is set, channel 8 is punched, channel 7 is not punched, and the remaining channels are punched from bits 30–35 of the buffer with bit 35 corresponding to channel 1. After punching is complete, the device clears Busy and sets Done, requesting an interrupt on the assigned channel.

Timing. If Busy is set when the punch motor is off, punching is automatically delayed 1 second while the motor gets up to speed. While the motor is on, punching is synchronized to a punch cycle of 20 ms. After Done sets, the program has 10 ms within which to give a new DATAO to keep punching at the maximum rate; after 10 ms punching is delayed until the next cycle. If Busy remains clear for 5 seconds the motor turns off.

EXAMPLE. Suppose we wish to punch out the same information we read from tape in the examples of the previous section. We cannot use a BLKO as an interrupt instruction unless we first spread the 6-bit characters over sixty memory locations. The example uses channel 5 and assumes that other channels are already in use.

```

MOVE    A,[JSR PUNCH]
MOVEM  A,52           ;Set up channel 5
CONO   PTP,55        ;Request interrupt for first word
CONO   PI,2004       ;Turn on channel 5
      :              ;Continue program
      :
PUNCH:  0             ;Interrupt routine
      ILDB   A,BYPPNT ;Put byte in A
      AOSL  CNT       ;Got all bytes?
      CONO  PTP,40    ;Yes, prevent interrupt after last word
      DATAO PTP,A    ;Punch byte
      JEN   @PUNCH

BYPPNT: XWD    440600,4000 ;Generate pointer here
CNT:    ↑D-60           ;Initialize count

```

Operation. The punch is located behind the reader; both are in a drawer that pulls out from the front of the console. Fanfold tape is fed from a box at the rear of the drawer. After it is punched, the tape moves into a storage

§3.3

bin from which the operator may remove it through a slot on the front. Pushing the feed button beside the slot clears the punch buffer and punches blank tape as long as it is held in. Busy being set prevents the button from clearing the buffer, so pressing it cannot interfere with program punching.

To load tape, first empty the chad box behind the punch. Then tear off the top of a box of fanfold tape (the top has a single flap; the bottom of the box has a small flap in the center as well as the flap that extends the full length of the box). Set the box in the frame at the back and thread the tape through the punch mechanism. The arrows on the tape should be underneath and should point in the direction of tape motion. If they are on top, turn the box around. If they point in the opposite direction, the box was opened at the wrong end; remove the box, seal up the bottom, open the top, and thread the tape correctly.

To facilitate loading, tear or cut the end of the tape diagonally. Thread the tape under the out-of-tape plate, open the front guide plate (over the sprocket wheel), push the tape beyond the sprocket wheel, and close the front guide plate. Press the feed button long enough to punch about a foot and a half of leader. Make sure the tape is feeding and folding properly in the storage bin. Pushing the button labeled POWER sets No Tape, pushing it again clears the flag. It can be used to hold the program at bay while a tape is being loaded.

To remove a length of perforated tape from the bin, first press the feed button long enough to provide an adequate trailer at the end of the tape (and also leader at the beginning of the next length of tape). Remove the tape from the bin and tear it off at a fold within the area in which only feed holes are punched. Make sure that the tape left in the bin is stacked to correspond to the folds; otherwise, it will not stack properly as it is being punched. After removal, turn the tape stack over so the beginning of the tape is on top, and *label it with name, date, and other appropriate information.*

Indicators for the punch are the PTP lights in the top row of the panel at the top of bay 1. The numbered lights display the last line punched.

3.3 TELETYPE

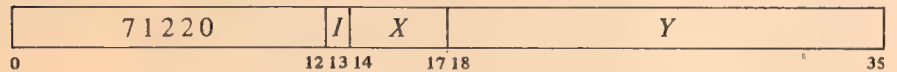
Two teletypewriter models are regularly available with the PDP-10 for use at the console: the KSR 35, which is capable of speeds up to ten characters per second, and the KSR 37, which can handle up to fifteen characters per second. The program can type out characters and can read in the characters produced when keys are struck at the keyboard.

The teletype separates its input and output functions and in effect acts like two devices with a single device code: each has its own Busy and Done flags, but the two share a common interrupt channel. Placing the code for a character in the output buffer causes the teletype to print the character or perform the designated control function. Striking a key places the code for the associated character in the input buffer where it can be retrieved by the program, but it does nothing at the teletype unless the program sends the code back as output.

Character codes received from the teletype have eight bits wherein the most significant is an even parity bit. The Model 35 ignores the parity bit in characters transmitted to it. The Model 37 ignores the parity bit in a code for a printable character, but it performs no function when it receives a control code with incorrect parity.

The Model 37 has the entire character set listed in the table in Appendix B. Lower case characters are not available on the Model 35, but transmitting a lower case code to the teletype causes it to print the corresponding upper case character. To go to the beginning of a new line the program must send both a carriage return, which moves the type box to the left margin, and a line feed, which spaces the paper. The teletype device code is 120, mnemonic TTY.

CONO TTY, Conditions Out, Teletype

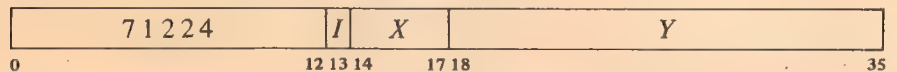


Set up the teletype control register according to bits 24–35 of the effective conditions *E* as shown (a 1 in bit 24 sets Test, a 0 clears it; all other flag functions are produced by 1s, 0s have no effect).

TEST	CLEAR INPUT BUSY	CLEAR INPUT DONE	CLEAR OUTPUT BUSY	CLEAR OUTPUT DONE	SET INPUT BUSY	SET INPUT DONE	SET OUTPUT BUSY	SET OUTPUT DONE	PRIORITY INTERRUPT ASSIGNMENT		
24	25	26	27	28	29	30	31	32	33	34	35

Setting Test connects the output buffer directly to the input buffer, allowing the program to check out the interface logic without the line and the device.

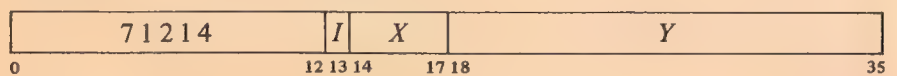
CONI TTY, Conditions In, Teletype



Read the status of the teletype into bits 24 and 29–35 of location *E* as shown.

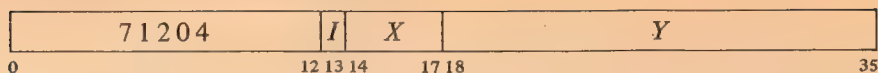
TEST					INPUT BUSY	INPUT DONE	OUTPUT BUSY	OUTPUT DONE	PRIORITY INTERRUPT ASSIGNMENT		
24	25	26	27	28	29	30	31	32	33	34	35

DATAO TTY, Data Out, Teletype



Load the contents of bits 28–35 of location *E* into the output buffer. Clear Output Done, set Output Busy, and enable the transmitter.

§3.3

DATAI TTY, Data In, Teletype

Transfer the contents of the input buffer into bits 28–35 of location *E*. Clear Input Done.

Output. A CONO need be given only to change the PI assignment; DATAO sets Output Busy and enables the transmitter while loading the buffer. Enabling the transmitter causes it to send the contents of the output buffer serially to the teletype. Completion of transmission clears Output Busy and sets Output Done, requesting an interrupt on the assigned channel.

Input. Teletype reception requires no initiating action by the program except to supply a PI assignment. Striking a key transmits the code for the character serially to the input buffer. The beginning of reception sets Input Busy; completion clears Input Busy and sets Input Done, requesting an interrupt on the assigned channel. A DATAI brings the character into memory and clears Input Done.

Timing. The Model 35 can type up to ten characters per second. After Output Done is set, the program has 9.09 ms to give a DATAO to keep typing at the maximum rate. After Input Done is set, the character is available for retrieval by a DATAI for 22.73 ms before another key strike can destroy it.

The 37 can handle fifteen characters per second, 66.7 ms per character. After Output Done is set, the program has 6.67 ms to send a new character to maintain the maximum typing rate. After Input Done is set, the character is available for at least 10 ms.

The sequence carriage return-line feed, when given in that order, allows sufficient time for the type box to get to the beginning of a new line. After tabbing, the program must wait for completion of the mechanical function by sending one or two rubouts. If the time is critical, the programmer should measure the time required for his tabs. Tabs are normally set every eight spaces (columns 9, 17, ...) and require one rubout.

Operation. The illustrations on the following two pages show the two teletype models. The teletype is actually two independent devices, keyboard and printer, which can be operated simultaneously. Power must be turned on by the operator. On the 35 the switch is beside the keyboard, and has an unmarked third position (opposite ON) which turns on power but with the machine off line so it can be used like a typewriter. A similar switch is located beneath the stand on the 37.

The keyboard resembles that of a standard typewriter. Codes for printable characters on the upper parts of the key tops on the 35 are transmitted by using the shift key; most control codes require use of the control key. Those familiar with the 35 who are using the 37 for the first time should take a close look at the keyboard. On the 37 the shift is used for real upper case characters. The control key is used for some control characters, but many



Teletype KSR 35

have separate keys. Note also that both the keyboard arrangement and the labels differ somewhat. On both, the line feed (labeled "new line" on the 37) spaces the paper vertically at six lines to the inch, and must be combined with a return to start a new line. The local advance (feed) and return keys affect the printer directly and do not transmit codes. Appendix B lists the complete teletype code, ASCII characters, key combinations, and differences between the two models.

Indicators for the teletype are the TTY lights in the second row of the



panel at the top of bay 1. The numbered lights display the last character typed in from the keyboard (bit 8 is parity). The ACT lights indicate activity in the transmitter and receiver. The remaining lights display the PI assignment and flags (the Input and Output Done flags are labeled TTI FLAG and TTO FLAG).

Teletype manuals supplied with the equipment give complete, illustrated descriptions of the procedures for loading paper, changing the ribbon, and setting horizontal and vertical tabs. The first two procedures are fairly

Teletype KSR 37

obvious: observe the paper or ribbon path and duplicate it. The other tasks are usually left for maintenance personnel. In any event, the best and easiest way to learn to do any of these things is to have someone who knows show you how.

4

Hardcopy Equipment

This chapter discusses the line printer, XY plotter, card reader, and card punch. Like the basic in-out equipment, these devices are primarily for communication between computer and operator using a paper medium: form paper, graph paper or cards.

The line printer provides text output at a relatively high rate. The program must effectively typeset each line; upon command the printer then prints the entire line. With the plotter, the program can produce ink drawings by controlling the incremental motion of pen on paper in a cartesian coordinate system. Curves and figures of any shape can be generated by proper combinations of motion in x and y .

The card equipment processes standard 12-row 80-column cards. Many programmers find cards a convenient medium for source program input and for supplying data that varies from one program run to another. Cards are convenient to prepare manually, input is much faster than paper tape, and simple changes are easy to make: individual cards can be repunched, and cards can be added or removed from the deck. The card reader cannot be used in readin mode, but a standard card-reading program in readin format can be kept on paper tape or DECTape. A possible consideration in using cards is that many installations do not include an online card punch.

These four devices are all run by the BA10 Hardcopy Control. Interface logic for a plotter can also be mounted in the TD10A DECTape Control.

4.1 LINE PRINTER

The line printer outputs hardcopy composed of lines 132 characters long at a nominal rate of 300,600 or 1000 lines per minute. The standard printer has sixty-four printing characters available to the program. The characters and codes are the figure and upper case sets, codes 040–137, in the teletype code [*Appendix B*]. When a lower case code (140–176) is given, the corresponding upper case code is loaded into the buffer. Besides accepting printing characters, the printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1–4. All other codes are ignored.

The printer has a 132-character buffer that holds the image of a single line; the program must first load the buffer up to five characters at a time, and then give a control character to print the entire line. The buffer is loaded from left to right, and only the portion filled produces a printout. Hence

for each line the program need send out characters (including spaces) only as far as the rightmost nonspace character. The characters are printed in the order that they pass the print hammers, and a given character is printed simultaneously in all positions that require it. In other words the drum has a row of 132 *Ms*, a row of *Ns*, etc; all *Ms* are printed together, all *Ns* together, and so forth. The first character printed depends only upon the position of the drum when the print command is given.

Printers having more than sixty-four characters are also available. The 96-character printer outputs 600 lines per minute and has the entire figure, upper case and lower case sets, codes 040-176. This is actually only ninety-five characters, but an option allows use of the delete code to select an extra character on the drum. A single delete code is ignored, but two consecutive 177s cause the code 177 to be loaded into the buffer. When the code for a printing character is the same as one for a nonprinting character and is loaded by giving it immediately after a delete code, the printing character is said to be "hidden" under the nonprinting one.

The 128-character printer outputs 500 lines per minute and uses the entire set of 7-bit codes for printing characters, with characters hidden under the ten control characters and also under null and delete.

Output Format. Paper motion is controlled by a format tape loop in the printer. The tape has eight columns and the loop corresponds to an integral number of pages of the fanfold form paper. With the exception of CR, every control character that prints a line from the contents of the buffer produces a different spacing by selecting a particular tape column. The paper then advances until a hole is encountered in the selected column.

The standard paper has 11-inch pages of sixty-six lines, and the standard tape for these generates the formats listed below. The fourth column gives the hole positions in terms of the numbered lines *on the tape*. The tape is usually installed at random and then positioned by pressing the top-of-form button on the printer. Then the paper is adjusted so that the desired line on the paper corresponds to line 0 on the tape. Ordinarily the paper is set with the print hammers at the fourth line, so all but one of these formats leaves a three-line margin at the top and a margin of at least three lines at the bottom of each page.

<i>Character</i>	<i>Column</i>	<i>Normal meaning</i>	<i>Hole positions</i>
FF (014)	1	Top of form	Line 0
CR (015)	<i>None</i>	No spacing (paper motion inhibited)	
LF (012)	8	Single space with automatic top of form after every 60 impressions	Every line from 0 to 59
DC1 (021)	3	Double space with automatic top of form after every 30 impressions	Every even numbered line from 0 to 58
DC2 (022)	4	Triple space with automatic top of form after every 20 impressions	Every third line from 0 to 57

Virtually any character set can be had on any printer by special order. In any event characters after the first ninety-five are always special order.

Spacing other than the standard can be produced by using a different format tape. The length of the loop should correspond to one or more pages of the printer form used, with holes punched at the lines where paper spacing is to stop.

Programmers generally treat the data for the line printer and teletype identically, using the combination CR plus LF for printing and spacing. This way a given character string can be outputted on either device. CR is used alone only when the next print command will overprint, *ie* will print another character in a column position already printed. With this technique the program can produce a character such as "≠" by overprinting a slash on an equal sign (or vice versa).

§4.1

LINE PRINTER

DC3 (023)	5	Single space	Every line
DC4 (024)	6	Space one sixth of a page	Lines 0, 10, 20, 30, 40, 50
VT (013)	7	Space one third of a page	Lines 0, 20, 40
DLE (020)	2	Space half a page	Lines 0, 30

The actual printer action of advancing the paper to the next hole in the tape produces the "normal" format only if the program consistently selects the same tape column. Always using DC1 to print produces double spaced text from line 4 to line 62 on every page. But if the last print command spaced to an odd numbered line, DC1 moves the paper only one line.

Printing Speed. The printer is available in five models with differing printing speeds.

<i>Printer</i>	<i>Nominal printing speed in lines per minute</i>	<i>Drum rotation in rpm</i>	<i>Time per revolution in ms</i>
LP10A	300	333	180
LP10B	600	750	80
LP10C	1000	1250	48
96 Character	600	750	80
128 Character	500	550	109

Printing begins as soon as a print command is given and terminates when the last required character is printed, *ie* without necessarily waiting for a complete drum revolution. Therefore print time depends on the initial drum position and the number of characters that must pass the print head before the last is printed. No time is required for spaces: the printer produces spaces in a line by not printing anything in the columns corresponding to the buffer positions that hold space characters. As a given character is printed, space codes replace the codes for the character in all buffer positions that hold it, and printing ceases when the buffer is filled with spaces.

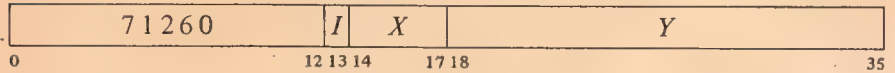
A complete print cycle consists of the print time plus the time required for advancing the paper; paper spacing begins immediately after printing terminates, and further printing is inhibited while the paper is moving. It takes about 12 ms to advance the paper one line, about 6–8 ms for each additional line. If the buffer is loaded only with spaces, the print cycle consists entirely of paper spacing.

Using an ordinary distribution of characters results in printing at or slightly above the nominal speed. Printing is faster however if paper spacing occurs while unused characters are passing the print head. *Eg* text that uses only the alphabet can be printed at the full drum rotation speed.

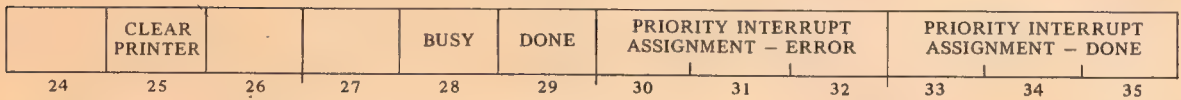
Instructions. The printer has the usual instructions for sending and reading conditions, but after initial setup it can be controlled entirely by the characters sent by a string of DATAOs. The program supplies five characters at a time to a 35-bit character buffer in the printer interface. The interface processes the characters from left to right loading valid data characters into the

line buffer, ignoring invalid characters, and sending control signals to the printer when a control character is encountered. The printer device code is 124, mnemonic LPT.

CONO LPT, Conditions Out, Line Printer



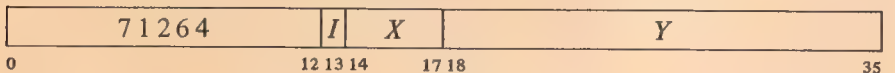
Perform the function given below if specified by a 1 in bit 25 and set up the printer control register according to bits 30–35 of the effective conditions *E* as shown (a 1 in a flag bit sets the flag, a 0 clears it).



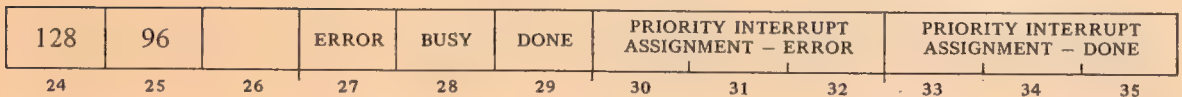
Power turnon and the IO reset signal generated by CONO APR,200000 duplicate this clear function.

If bit 25 is 1, clear Done, set Busy, clear the interface logic, and trigger a print cycle to clear the line buffer. The cycle clears the buffer by replacing the characters in it with spaces, and the time required is the same as would be required to print whatever is in it. Completion of the cycle clears Busy and sets Done, requesting an interrupt on the channel assigned by bits 33–35.

CONI LPT, Conditions In, Line Printer

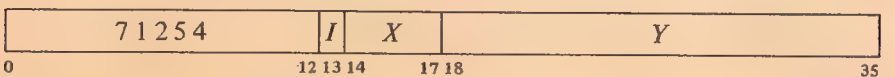


Read the status of the printer into bits 24–35 of location *E* as shown.



A 1 in bit 24 indicates that the printer has a 128-character drum; a 1 in bit 25 indicates that at least 95 characters are available to the program.

DATAO LPT, Data Out, Line Printer



Load the contents of bits 0–34 of location *E* into the character buffer, clear Done, set Busy, and trigger the interface processing cycle. The format of the

data word and the order in which the characters are processed is as shown.

FIRST	SECOND	THIRD	FOURTH	FIFTH	
0	6 7	13 14	20 21	27 28	34

Characters are assembled into words in this manner by an IDPB loop or an ASCII or ASCIIZ pseudoinstruction.

Following power turnon, the Error flag (CONI bit 27) is set if the printer cable is not connected or any other condition exists that makes the printer unavailable to the program [*these other conditions are given in the discussion of printer operation at the end of the section*]. If Error is set when a CONO gives an error PI assignment (with bits 30–32 of *E*), there is an immediate interrupt request on the error channel. Barring accident or hardware malfunction, an error interrupt is likely to occur during a printout run only when the printer is about to run out of paper or the operator stops it (in either case Error sets and the printer stops when the buffer is empty following the printing of a line).

At the beginning of a print run the program should give a CONO to clear the line buffer and assign the PI channels. After that a CONO need be given only to change the PI assignments; each DATAO starts the character-processing operations of the interface while loading the character buffer. The interface processes the characters from left to right, starting each character cycle when the line buffer is ready. Printing characters are simply sent to the buffer, with lower case codes translated to upper case for a 64-character printer. Unused codes are ignored. The interface responds as follows when a control character is encountered.

◆ A horizontal tab (HT) is simulated by sending a string of spaces to the line buffer. Tab stops are every eight columns (9, 17, . . .). The interface always sends at least one space, and then sends as many more as are necessary for the next character to be at a tab stop. Thus if a DATAO gives the sequence

A HT *B*.

These tabs are the same as the ones ordinarily used on the teletype.

where *A* is placed in column 7, *B* will go into column 9. But if *A* goes into column 8, *B* will go into column 17.

◆ Upon encountering any other printer control character, the interface signals the printer to print the contents of the line buffer, and unless the character is CR, it also selects a format tape column to space the paper as listed in the format discussion at the beginning of this section. When the buffer again becomes available, subsequent characters will be loaded starting in column 1. If printing is caused by a CR, the next line will overprint unless the paper is advanced before any nonspace characters are loaded into the buffer.

If the buffer is filled with 132 characters and the next character does not cause printing, the interface simulates a line feed to print and advance the paper, and then loads the next character at column 1 for the new line. If the program tabs to the end of a line, the interface simulates a line feed and also tabs at the beginning of the next line. In other words a printing character following the tab will be loaded at column 9 for the new line.

When the interface finishes processing the five characters supplied by a

DATAO, it clears Busy and sets Done, requesting an interrupt on the channel assigned by bits 33–35 of the conditions out.

Timing. The time from one DATAO to the next while the program is loading the buffer is simply the time required by the interface to process five characters. Loading each printing character, including each space in a horizontal tab, takes 10 μ s. Skipping an illegal character takes 8 μ s.

If the fifth character causes printing, Done is set immediately and the program can give a DATAO to send the first set of characters for the next line. However, the interface does not begin processing the new characters until the buffer becomes available after the printer finishes printing the previous line. If printing is produced by any character before the last, the print time elapses before the interface processes the next character in the current set.

The overall time required for a print run is the total printing and spacing time for all lines as given above in the discussion of the printing speed. The time required to process individual characters is a consideration in programming the DATAOs that load the buffer, but buffer loading time is not a factor in total printer operating time except when loading characters for overprinting (following a CR). This is because the buffer becomes available while the paper is moving, in plenty of time for the program to load it before the paper stops.

EXAMPLES. In the first example, which uses the line printer without the interrupt, we have simply filled in the missing part of the print subroutine given at the top of page 2-61 (it prints the characters that accompany the calling sequence given at the bottom of page 2-60).

```
PRINT:  HRLI    T,440700
        ILDB    CH,T
        JUMPE  CH,1(T)
        CONSZ  LPT,200      ;Skip when printer not busy
        JRST   .-1         ;Wait for Busy to clear
        LSH    CH,1        ;Shift character to bits 28-34
        DATAO LPT,CH      ;Send character to printer
        JRST   PRINT+1
```

The same program could be used for output on the teletype by making the substitution

CONSZ LPT,200 → CONSZ TTY,20

and deleting the LSH CH,1.

The above is perhaps an overly simple example. It assumes the line buffer is clear initially and the printer is available. Moreover the processor spends most of its time waiting. Characters are processed individually in order to detect the null, but if the processor has anything else to do, it would be much more efficient to use the interrupt and send five characters at a time.

In the following example the main program sets up each print run by giving a JSR SETUP. The number of words printed and the starting location of the block containing them are determined by the contents of PNTR1. Once a run is set up, the program can change the contents of PNTR1 for the next one.

```

§4.1
SETUP:  0
        SKIPGE PNTR
        JRST  .-1          ;Wait for current IO to finish
        MOVE  T,[JSR  ERROR]
        MOVEM T,42         ;Channel 1 for error
        MOVE  T,[JSR  DATA]
        MOVEM T,44         ;Channel 2 for data
        MOVE  T,PNTR1
        MOVEM T,PNTR      ;Set up new IO block pointer
        CONO  LPT,2012    ;Clear printer, assign channels
        CONO  PI,2340     ;Turn on PI and channels
        JRST  @SETUP

PNTR1:  0
PNTR:   0

ERROR:  0
        CONO  LPT,2       ;Drop error request by dropping error
        .      .          ;PI assignment
        .      .          ;Start typing error message
        JEN   @ERROR

DATA:   0
        CONO  LPT,12      ;Reassign error channel
        BLKO  LPT,PNTR    ;Send out word
        CONO  LPT,0       ;Turn off printer
        JEN   @DATA

```

End of clear function sets
Done, requesting a data
interrupt.

Operation. The 600-line-per-minute printer is illustrated on the following page. At the left on the front of the printer are two round indicators and two columns of square buttons and indicators, some of which are not used. The round lights indicate whether the printer has power: green light for power on, red for off.

The buttons at the top of the columns operate the printer. Pushing START places the printer on line so it can respond to the program (the button is lit while the unit is on line). Pushing STOP takes the unit off line; the operator can then use the TOP OF FORM button to position the paper (or the format tape). If the program has left anything in the buffer, it can be printed by pressing MANUAL PRINT. The maintenance button TEST can also be used while STOP is lit. START, STOP and TOP OF FORM are duplicated at the rear of the printer.

At the bottom of the columns are four alarm lights that indicate when the paper supply is low, the printer is out of paper or the paper is broken, the yoke is open, or there is a circuit malfunction (ALARM STATUS). When the operator presses STOP or there is a paper low alert, START does not go out until the buffer is empty (in other words until the printer finishes printing a line currently being loaded or printed). START goes out immediately if any other alarm condition occurs or power fails. When START is out or the cable to the interface is not connected, the Error flag is set and the printer cannot respond to the program.



Line Printer LP10B

The lights for the interface are in the top two rows on the hardcopy control indicator panel [illustrated on the opposite page]. The top row displays the contents of the character buffer; the 7-bit characters are shifted left for processing. The shift and column counters at the left end of the second row indicate the last character processed (0-4) and the last buffer position loaded. The group of lights at the right display the status condi-



Indicator Panel,
Hardcopy Control

tions. Of the group in the center, BUFF AVAIL indicates the line buffer is ready for the next character; the remaining lights are for maintenance.

To load paper, press STOP. If START does not go out, the program probably left the last line in the buffer: press MANUAL. When printing is complete the light will go out. Open the printer cover. At the front are two toggle switches: switch both of them to OPEN. The printer yoke will slide forward. Lift the guide plates over the two pairs of tractors, pull out the remaining paper, and press TOP OF FORM to line up the spacing format tape. Bring the beginning of the paper up behind the yoke, over the top and through the rollers on the back. Move the paper until line 4 of a page is lined up with the print hammers (at most installations the point at which the fold should come is marked). Make sure the tractor wheels engage the holes at the edges of the paper, close the guide plates, switch the toggles to CLOSE, close the cover, and press START.

All of the larger and faster printers are as described above! On the slowest printer the lights are at the right, the single PAPER ALARM indicates the paper is either low or broken, and there are no buttons on the back. With the cover open the yoke is controlled by two unmarked plastic switches on either side at the top. Pressing them in at the end nearer the front opens the yoke. This printer has only one pair of tractors, but it has a pair of bars below the yoke. The paper must go over the stationary bar and under the movable one.

4.2 PLOTTER

The XY10 plotter control interfaces the PDP-10 central processor to various plotters that use cartesian coordinates. The models most frequently used are manufactured by Calcomp, but others can be accommodated. The following lists the type and paper size of the most commonly supplied Calcomp models.

<i>Calcomp model</i>	<i>Type</i>	<i>Paper size in inches</i>
502, 602	Bed	31 × 34
518, 618	Bed	54 × 72
563, 663	Drum	29½ × 1440
565, 665	Drum	11 × 1440

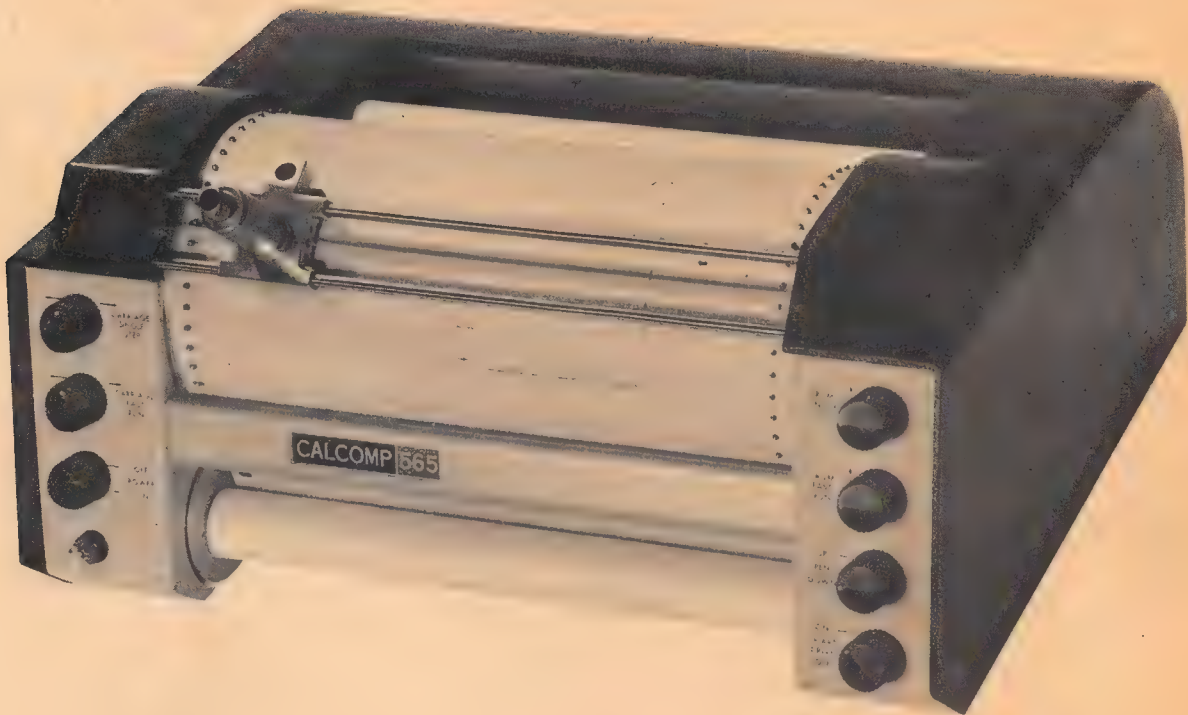
These are high accuracy, incremental digital plotters that produce fine quality ink plots of computer-generated data. Bidirectional stepping motors provide individual increments of motion in either coordinate or both at once. The program draws a continuous sequence of line segments by controlling the relative motion of pen and paper with the pen lowered, and it can raise the pen for repositioning.

Motion in y is movement of the pen carriage along a pair of rods. Motion in x is movement of the entire carriage-and-rod mechanism on a bed plotter, movement of the paper underneath the carriage on the drum type. On a bed plotter the coordinate directions are the standard ones when viewing the device from the front: positive x to the right, positive y to the back. The coordinate system on a drum is in the standard orientation when the viewer is standing at the right side, unrolling the paper from the drum with his left hand. In other words positive y is movement of the pen from right to left across the drum, positive x is drum rotation downward at the front (drawing a line toward the paper supply roll at the back).

The step sizes and plotting speeds available with the various Calcomp models are the following.

<i>Model</i>	<i>Step size</i>	<i>Plotting speed in steps per second</i>
502	<i>All sizes</i>	300
	.005 inch	200
518	.002 inch	450
	.1 mm	200
	.05 mm	400
	.010 inch	200
563	.005 inch	300
	.1 mm	300
565	<i>All sizes</i>	300
602	<i>All sizes</i>	450/900
	.005/.0025 inch	200/400
618	.002/.001 inch	450/900
	.1/.05 mm	200/400
	.05/.025 mm	450/900
	.010/.005 inch	350/700
663	.005/.0025 inch	450/900
	.0025/.00125 inch	450/900
665	<i>All sizes</i>	450/900

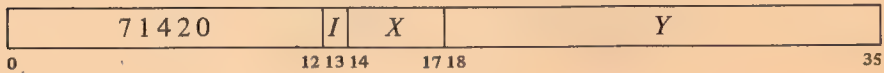
Calcomp plotters in the 600 series have two step sizes and two plotting speeds: a switch at the back selects the step size, delay settings in the plotter control determine the speed.



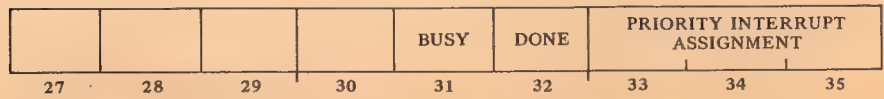
The program can draw any complete figure by giving a string of DATAOs, each of which supplies the information for one step. The plotter device code is 140, mnemonic PLT.

Calcomp Drum Plotter Model 565

CONO PLT, Conditions Out, Plotter



Set up the plotter control register according to bits 31-35 of the effective conditions *E* as shown (a 1 in a flag bit sets the flag, a 0 clears it).



CONI PLT, Conditions In, Plotter

7 2 4 2 4				I	X	Y			
0				12 13 14	17 18				35

Read the status of the plotter into bits 30–35 of location *E* as shown.

			POWER ON	BUSY	DONE	PRIORITY INTERRUPT ASSIGNMENT			
27	28	29	30	31	32	33	34	35	

Power On is not available on all plotters.

DATAO PLT, Data Out, Plotter

7 1 4 1 4				I	X	Y			
0				12 13 14	17 18			35	

Clear Done, set Busy, and move the pen as specified by bits 30–35 of the contents of location *E* as shown (a 1 in a bit produces the indicated motion, a 0 has no effect).

RAISE PEN	LOWER PEN	$-\Delta X$ (DRUM UP)	$+\Delta X$ (DRUM DOWN)	$+\Delta Y$ (CARRIAGE LEFT)	$-\Delta Y$ (CARRIAGE RIGHT)
30	31	32	33	34	35

A CONO need be given only to change the PI assignment; DATAO places the plotter in operation by supplying plotting data. After sufficient time has elapsed for the device to carry out the specified action, the control clears Busy and sets Done, requesting an interrupt on the assigned channel.

To avoid drawing line segments shorter than one step, do not raise or lower the pen in the same DATAO that calls for *xy* motion. The consequences of specifying contradictory movements cannot be predicted.

Timing. Lowering the pen takes 60 ms, raising it takes 10 ms. The time required to move one step in either or both coordinates depends on the plotting speed as follows.

<i>Plotting speed in steps per second</i>	<i>Time per step in ms</i>
200	2.5
300	1.66
350	1.45
400	1.25
450	1.10
700	.70
900	.51

EXAMPLE. The plotting commands sent out by this program are contained six to a word in WC words beginning at location BUFFER. The interrupt routine uses one accumulator which is shared with the main program and other channels.

```

CONSZ  PLT,7           ;Wait until previous run finished as
JRST   .-1            ;indicated by no PI assignment
MOVE   T,[JSR DATA]
MOVEM  T,50           ;Set up channel 4
MOVEI  T,WC*6         ;Set up count for plotting commands
MOVEM  T,COUNT
MOVE   T,[POINT 6,BUFFER] ;Initiate byte pointer
MOVEM  T,CHARP
CONO   PLT,4          ;Assign channel
CONO   PI,2210        ;Turn on PI and channel
DATAO  PLT,PUP        ;Raise pen to trigger first interrupt
:
:
DATA:  0
SOSGE  COUNT          ;Is plot finished?
JRST   DATA1         ;Yes
MOVEM  T,TSAVE        ;Save T
ILDB   T,CHARP        ;Get next plotting command
DATAO  PLT,T          ;Plot point
MOVE   T,TSAVE        ;Restore T
JEN    @DATA

PUP:   40
TSAVE: 0
COUNT: 0
CHARP: 0

DATA1: CONO  PLT,0     ;Disconnect plotter from interrupt
        DATAO PLT,PUP ;Raise pen
        JEN    @DATA

```

The asterisk is the sign for multiplication in MACRO.

POINT is a pseudoinstruction that causes MACRO to generate a byte pointer from the three arguments that follow it. In order these arguments are the byte length in decimal, the address of the location containing the byte, and the position of the rightmost bit of the byte as the decimal number of the bit in the word. If the last argument is omitted, MACRO takes it as -1; in other words, after being incremented the pointer will point to the first byte. The left half of the pointer generated here is 440600.

Operation. On a drum plotter the supply roll is behind the drum. Bring the paper over the drum, down in front, and above and behind the pickup roll underneath the drum (use a piece of masking tape to attach the paper, or roll some onto the tube).

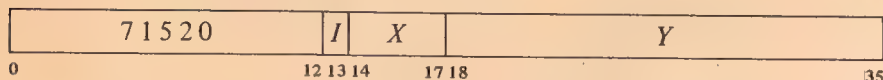
The controls are on the front [refer to the illustration on page 4-11]. To put the plotter on line simply turn on the power and the chart drive. The remaining controls are for manual operation: raising and lowering the pen, moving the carriage and drum in either direction, rapidly or single step. The switch that selects the step size on a 600-series plotter is on the back. The bed plotter has similar controls.

Lights for the plotter are the group at the right end in the bottom row on the hardcopy control indicator panel [page 4-9]. These display the status conditions and the plotting data supplied by the last DATAO. If the plotter interface is mounted in a DECTape control, there are no lights.

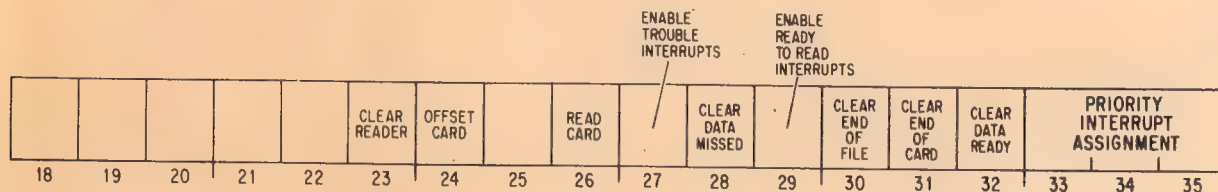
4.3 CARD READER

The card reader handles standard 12-row 80-column cards at speeds up to 1000 cards per minute (833 if power is 50 Hz). Once started, an entire card is read column by column. The reader supplies each column to the processor as twelve bits, and the program can translate in any way it wishes; the standard DEC character representations and the translation to ASCII made by the Monitor are given in Appendix B. Of course the data can simply be in binary at three columns per word (a 7 and 9 punch in the first column is the standard indication that the rest of the card contains binary data).

The interface contains a 12-bit buffer from which each column is retrieved by the processor. The reader device code is 150, mnemonic CR.

CONO CR, Conditions Out, Card Reader

Assign the interrupt channel specified by bits 33–35 of *E* and perform the functions specified by bits 23–32 as shown (in bits 27 and 29 a 1 enables the given flag to interrupt, a 0 disables it; in all other bits a 1 produces the indicated function, a 0 has no effect).

*Notes.*

- 23 Dismiss the PI assignment (assign zero); clear flags Reading Card, Data Missed, End of File, End of Card, Data Ready, Trouble Interrupt Enabled, Ready to Read Interrupt Enabled; clear the card column buffer; and disable any read command given by a CONO if the reader has not yet started the card. If any action specified by the rest of the CONO bits conflicts with these actions, the clear function has precedence.
- 24 If a card is currently being processed in the reader (Card In Reader, CONI bit 24, is 1), offset it when it is placed in the stacker. The card will actually stick out about a half inch from the rest of the stacked deck.

With the console model, off-setting a card places it in a separate stacker.

CONI CR, Conditions In, Card Reader

7 1 5 2 4	I	X	Y
0	12 13 14	17 18	35

Read the status of the reader into the right half of location *E* as shown.

TROUBLE INTERRUPT ENABLED	READY TO READ INTERRUPT ENABLED									*	*	*	*	*	*				
18	19	PICK FAILURE	PHOTO CELL ERROR	CARD MOTION ERROR	STOP	CARD IN READER	HOPPER EMPTY- STACKER FULL	READING CARD	TROUBLE	DATA MISSED	READY TO READ	END OF FILE	END OF CARD	DATA READY	PRIORITY INTERRUPT ASSIGNMENT		33	34	35

Notes.

*These bits cause interrupts.

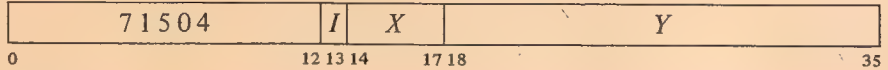
Interrupts are requested on the assigned channel by the setting of Data Ready, Data Missed, End of Card, End of File, and if enabled, Trouble and Ready to Read.

- 20 The reader has received a read command but has failed to bring in a card from the hopper.
- 21 The reader has failed to read a card properly and maintenance is probably required. The program should be dubious of any data taken from the card being read when the error occurred.
- 22 A card has failed to move properly through the reader (it has probably slipped). The program should be dubious of any data taken from the card being read when the error occurred.
- 23 Reader power is on but the reader is or soon will be unavailable to the program either because the operator has pressed the stop button or there is a trouble condition (bit 27). If Stop is set while a card is being read, the reader usually finishes it; only a power failure can stop the reader in the middle of a card.
- 24 The reader has brought a card in from the hopper and has not yet finished reading it. The program can give a CONO offset command while this bit is 1.
- 26 The reader has accepted a read command and has not yet finished reading the card.
- 27 Bit 20, 21, 22 or 25 is 1. If bit 18 is also 1, the setting of Trouble requests an interrupt on the assigned channel.
- Any condition that sets Trouble also sets Stop (bit 23) and the reader will stop at the end of the current card (of course a pick failure prevents the reader from even starting a card). Although a 1 in bit 27 does not necessarily imply an error or malfunction, it always requires operator intervention. If bit 25 is 1 it is very likely that the only trouble is the hopper is empty or the stacker is full.
- 28 The program failed to retrieve a column of data before the next column was loaded into the buffer by the reader.

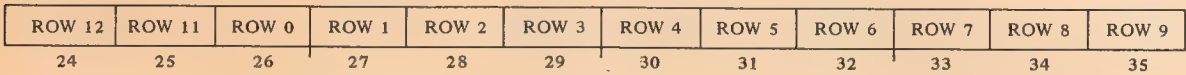
The usual procedure is to put an end-of-file card at the end of the deck rather than use the button. Actually the button can be used to signal the program for any purpose provided the reader is off line (stopped).

- 29 The reader is ready to accept a read command. If bit 19 is 1, the setting of Ready to Read requests an interrupt on the assigned channel.
- 30 The reader has stopped (probably because the hopper is empty) and the operator has pressed the end-of-file button.

DATAI CR, Data In, Card Reader



Clear Data Ready, and transfer the contents of the card column buffer into bits 24-35 of location E where the correspondence of card rows to bit positions is as shown.



If the program does not retrieve the final column and a CONO that starts a new card does not clear Data Ready, Data Missed will be set by the first column in the new card.

The program must give a CONO with a 1 in bit 26 to start every card. This read card command waits until the reader is ready, at which time Reading Card sets and the reader card cycle begins. Movement of a card in from the hopper sets Card in Reader. As each column is loaded into the buffer, Data Ready sets, requesting an interrupt on the assigned channel. The program must respond with a DATAI to transfer the column to memory and clear Data Ready. If Data Ready is still set when the next column is loaded into the buffer, Data Missed is set, requesting a second interrupt.

After all eighty columns have been read, Card in Reader goes off, clearing Reading Card and setting End of Card, which requests an interrupt. The card then moves out to the stacker, and when the device is ready to begin a new card cycle, Ready to Read goes on, but only if no new read card command has been given. If a read card command is already waiting when the reader becomes ready, it simply accepts the command and Ready to Read remains off. If no command is waiting, Ready to Read goes on, requesting an interrupt if enabled (CONI bit 19 is 1), and it goes off automatically when a new command is given.

Timing. After Reading Card sets, 18 ms elapse before Card in Reader goes on. The first Data Ready occurs 1.8 ms later. Subsequent columns are ready every 370 μs — the program must give a DATAI within 350 μs after each setting of Data Ready. Total time from first to last Data Ready is 29.2 ms. After the final Data Ready, 1.8 ms elapse before Card in Reader and Reading Card clear and End of Card sets. The program then has 9.2 ms within which to give a new CONO read card command to keep the reader going at the maximum rate. Ready to Read goes on at the end of this period if no new command appears.

If the reader operates on 50 Hz power, all times must be increased by 20 per cent.

When the last card in a deck is read, the hopper empty signal is simul-

taneous with End of Card.

Operation. The reader has a hopper and stacker capacity of 1000 cards. To load a deck, first fan the cards and jog them on the reader shelf. Turn the deck over and put the first hundred cards (about an inch of the deck) into the hopper (upper right) with the 9 edge against the back so column 1 is read first. Place the rest of the deck on top of the first part. Cards can be added to the hopper while the reader is running, but always stop the reader before removing cards from the stacker.

The reader is operated by the buttons at the left. The alternate-action POWER switch lights green when power is on. Pushing START places the reader on line so the program can read cards. Pushing STOP turns off the reader, taking it off line.

The lights at the right indicate an empty hopper, a full stacker, a pick failure, a card motion error, and a photocell output that is too weak or too strong. When one of these lights goes on the STOP light also goes on (the reader always finishes the current card before stopping). Do not attempt to reread a worn or damaged card that has caused a pick failure or motion error — duplicate it first. If any trouble light remains on after the problem is corrected press the CLEAR button; this turns off both the lights and the corresponding status signals read by a CONI. Press START to allow the program to continue reading the deck. If the trouble persists, enter it in the system log and notify maintenance personnel.

Pressing the END OF FILE button (at the right) when the reader is off line, as when the hopper is empty, sets the End of File flag. When the TEST MODE light is on, the reader processes cards off line (the test switch is behind the shelf).

Lights for the interface are in the bottom two rows on the hardcopy control indicator panel [page 4-9]. The left section of the upper row displays the contents of the card column buffer; the lights are marked by card row. The left section of the bottom row displays bits 24–35 of the status conditions. (The second light from the left is labeled HOP EMPTY, but it goes on when the hopper is empty or the stacker is full.) Of the five lights in the center, the left one is the momentary offset signal. READ is on when a read command has been given but the reader is not yet ready. The next two lights display bits 18 and 19 of the status conditions, and the last light is on while an interrupt is being requested whatever the cause.



Card Reader

Also available is a console model reader that has a 2000-card hopper and two 2000-card stackers. In use it differs from the compact model only in that offsetting a card places it in the second stacker (the one on the right), and cards can be removed from the stackers while the reader is running.

4.4 CARD PUNCH

The card punch handles standard 12-row 80-column cards at speeds up to 200 cards per minute if all eighty columns are punched, 365 cards per minute if only the first sixteen columns are punched. The processor must supply each column to the punch as twelve bits, and the program can generate this data by any procedure it wishes; the standard DEC character representations and the translation from ASCII made by the Monitor are given in Appendix B. Of course the data can simply be in binary at three columns per word (punching rows 7 and 9 in the first column is the standard procedure for indicating that the rest of the card contains binary data).

A card is taken from the hopper only when the program supplies data for the first column. In the interface is a 12-bit buffer to which the processor sends each column, but the punch has a 48-bit buffer, and it punches four columns at a time from each set of four 12-bit bytes sent through the interface. The program can send a card to the stacker after punching any number of columns. The punch device code is 110, mnemonic CDP.

CONO CDP, Conditions Out, Card Punch



Assign the interrupt channel specified by bits 33–35 of the effective conditions *E* and perform the functions specified by bits 20–32 as shown (a 1 in a bit produces the indicated function, a 0 has no effect).

		CLEAR PUNCH	OFFSET CARD		EJECT CARD	DISABLE TROUBLE INTERRUPTS	ENABLE TROUBLE INTERRUPTS	CLEAR ERROR	DISABLE END OF CARD	ENABLE END OF CARD	CLEAR END OF CARD	SET PUNCH ON	CLEAR DATA REQUEST	SET DATA REQUEST	PRIORITY INTERRUPT ASSIGNMENT		
18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35

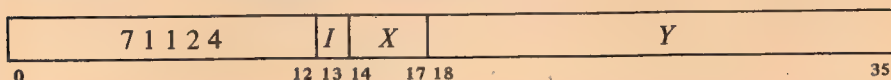
Notes.

- 20 Clear flags Trouble Interrupt Enabled, Error, End of Card Enabled, End of Card, Punch On, Busy, Data Request; clear the card column buffer. If any action specified by the rest of the CONO bits conflicts with these actions, the other bits have precedence.

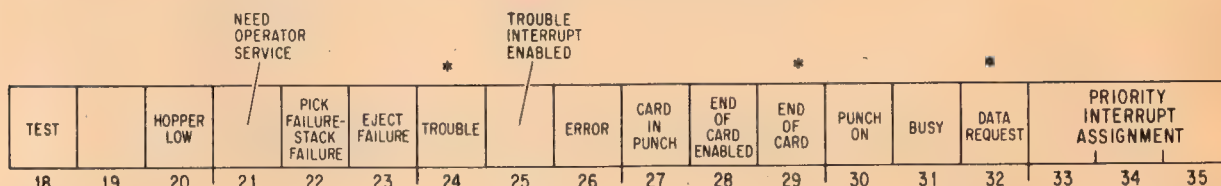
- 21 If a card is currently being processed in the punch (Card in Punch, CONI bit 27, is 1) or was ejected less than 3 ms ago, offset it when it is placed in the stacker. The card will actually stick out about a half inch from the rest of the stacked deck.
- 23 If a card is currently being processed (Card in Punch, CONI bit 27, is 1), punch whatever data is in the 4-column buffer and then eject the card. Ejection moves a card through the punch head assembly four times as fast as punching blank columns.

With the console model, offsetting a card places it in a separate stacker.

CONI CDP, Conditions In, Card Punch



Read the status of the punch into the right half of location *E* as shown.



*These bits cause interrupts

Notes.

Interrupts are requested on the assigned channel by the setting of Data Request, End of Card, and if enabled, Trouble.

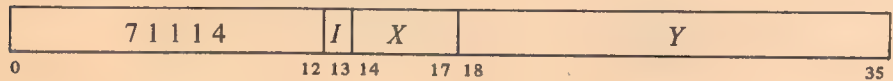
- 18 The operator has turned on the test switch, taking the punch off line.
- 20 Less than a hundred cards are left in the hopper.
- 21 The hopper is empty or the stacker or chip box is full.
- 22 The punch has received data for the first column but has failed to bring in a card from the hopper; or it has received an eject command but has failed to place the card properly in the stacker.
- 23 The punch has received an eject command but has failed to move the card out of the punch head assembly.
- 24 Bit 18, 22 or 23 is 1, or bit 21 is 1 because the hopper is empty or the stacker is full, or the operator has taken the punch off line. If bit 25 is also 1, the setting of Trouble requests an interrupt on the assigned channel.

Ordinarily a trouble condition allows the punch to finish a card but prevents it from starting another; only a power failure or the operator turning on the test switch (bit 18) can take the punch off line in the middle of a card. A full chip box does not stop the punch at all as there is actually enough room left for the chips from a whole

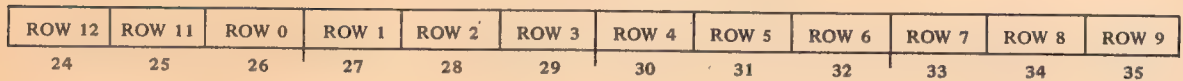
hopper full of cards. Although a 1 in bit 24 does not necessarily imply a malfunction, it always requires operator intervention. If bit 21 is 1 it is very likely that the only trouble is the hopper is empty or the stacker is full.

- 26 A column punched in a card does not agree with the data sent by the processor.
- 27 A card is in the punch head assembly. The program can give a CONO offset or eject command while this bit is 1 (the offset can also be given within 3 ms after Card in Punch clears).
- 29 Bit 28 is 1 and the program has given either an eject command or data for column 80. The setting of End of Card requests an interrupt on the assigned channel.

DATAO CDP, Data Out, Card Punch



Clear Data Request, set Punch On and Busy, and load the contents of bits 24–35 of location *E* into the interface column buffer where the correspondence of bit positions to card rows is as shown.



Setting Punch On turns on the punch motor, but only a DATAO can pick a card. Since DATAO also sets Punch On, the program can initiate punch operations while supplying data, but the usual procedure is to set Punch On while giving other initial conditions.

When the punch is ready to take a card from the hopper it sends a ready signal to the interface. This sets Data Request, which requests an interrupt on the assigned channel. To pick a card the program must respond with a DATAO, which supplies the first column, clears Data Request, and sets Punch On and Busy. The interface then sends the column to the 4-column punch buffer and clears Busy. While the punch is picking a card it also makes three more data requests to each of which the program must respond with a DATAO. When the card is properly registered in the punch head assembly, Card in Punch sets. When this flag has set and the program has supplied the first four columns, the device punches the four columns simultaneously (a 1 sent to the column buffer produces a hole in the card). The punch then continues in this fashion making four data requests for each set of four columns.

Punch On clears when the program gives an eject command. This causes the device to punch whatever is in its 4-column buffer and the card then

If the program gives a DATAO to turn on the motor, the initial ready from the punch takes the first column from the column buffer but does not set Data Request. When that flag does set, the punch is ready for the second column.

§4.4

moves out to the stacker. If the punch has already sent a ready signal, the CONO that ejects should also clear Data Request. If End of Card has been enabled by a 1 in CONO bit 28, the eject command sets it, requesting an interrupt. If no eject command has been given by the time data is supplied for column 80, End of Card sets anyway if it is enabled (producing an interrupt request), but the card remains in the punch head assembly until an eject command is given. The actual ejection of a card clears Card in Punch.

Timing. If Punch On is set when the punch motor is off, the first ready signal is delayed about 120 ms while the motor gets up to speed. When the motor is on, Card in Punch sets about 60 ms after the DATAO that sends the first column for a card. While the card is in the head assembly, punching is synchronized to a punch cycle of 11.1 ms. About 30 μ s elapse from each DATAO to the next Data Request, but after the first request the program has the full punch cycle time to supply all four columns and keep punching at the maximum rate; after that punching is delayed until the next cycle.

Giving an eject command clears Punch On and sets End of Card after 5 μ s, but Card in Punch does not clear until the card leaves the head assembly; this takes about 25 ms plus 2.8 ms for each set of four columns skipped over. After Card in Punch clears, about 30 μ s elapse before the punch indicates that it is ready to pick another card from the hopper, at which time the program should give a DATAO to pick another card at the maximum rate. (Of course the first DATAO can be given right after the eject command, and the punch will then pick another card automatically without setting Data Request for the first column.) When the final card is punched, the hopper empty signal is simultaneous with End of Card. If Punch On remains clear for about 30 seconds, the motor turns off.

Operation. The punch has a hopper and stacker capacity of 1000 cards. To load the hopper, first fan the cards and jog them on the punch shelf. Turn the deck over and put the first hundred cards (about an inch of the deck) into the hopper (upper right) with the 9 edge against the back so column 1 is punched first. Hold the right end higher so the leading edge of the bottom card rests against the picker throat, and drop the cards in place. Put the rest of the deck on top of the first part. Cards can be added to the hopper while the punch is running, but always stop the punch before removing cards from the stacker. To remove cards, push down the elevator and lift the stack out.

If the program does not eject before the punch starts punching columns 77–80, it makes another data request. The program can then supply two more columns, which will be punched in the margin of the card.

CAUTION

Any data that is given but not punched (*eg* the first column(s) when there is a pick failure) is usually lost when the punch goes off line. Hence the program should always start with the first column of a card when the punch is restarted.



Card Punch

The punch is operated by the buttons in the upper part of the panel at the right. The alternate-action POWER switch lights green when power is on. Pushing START places the punch on line so the program can punch cards. The OPERATE indicator at the lower right lights green when the punch motor is up to speed. Pushing STOP takes the reader off line but does not stop the motor; the motor is turned off only by pressing CLEAR.

The lights in the bottom row indicate an empty hopper or full stacker, a full chip box, a pick failure, an eject failure, and a stack failure. When one of these lights other than CHIP BOX goes on, the STOP light also goes on (the punch always finishes the current card before stopping). If any trouble light remains on after the problem is corrected, press the CLEAR button; this turns off both the lights and the corresponding status signals read by a CONI. Pressing CLEAR also ejects a card if one is in the head assembly, and the button glows red when clear action is required (eg when a card has gotten stuck). For a pick failure, empty the hopper, throw out the bottom card, and reload. Press START to allow the program to continue punching. If the trouble persists enter it in the system log and notify maintenance personnel.

A full chip box does not stop the punch, but once it has been stopped by some other condition (such as an empty hopper), pressing START will not place the unit on line until the box has been emptied.

At the right is a light for the Card in Punch flag. The ERROR light displays the signal that sets the Error flag; it goes off when CLEAR is pressed. The CHECKOFF light is not used. When the TEST light is on, the device punches cards off line in a test pattern (the test switch is behind the panel under the shelf).

Lights for the interface are in the second row from the bottom on the hard-copy control indicator panel [page 4-9]. The middle section of the row displays the contents of the card column buffer; the lights are marked by card row. Among the lights in the right section, PI REQ is on while an interrupt is being requested whatever the cause. The remaining lights display some of the status conditions read by a CONI.

Also available is a console model punch that has a 2000-card hopper and two 2000-card stackers. In use it differs from the compact model only in that offsetting a card places it in the second stacker (the one on the right), and cards can be removed from the stackers while the punch is running.

Appendices

APPENDIX A

INSTRUCTION AND DEVICE MNEMONICS

The illustration on the next page shows the derivation of the instruction mnemonics. The two tables following it list all instruction mnemonics and their octal codes both numerically and alphabetically. When two mnemonics are given for the same octal code, the first is the preferred form, but the assembler does recognize the second. For completeness, UUOs are listed for user mode (an asterisk indicates a UUO mnemonic recognized by MACRO for communication with the PDP-10 Time Sharing Monitor). All UUOs 000-077 are identical when the processor is not in user mode.

In-out device codes are included only in the alphabetic listing and are indicated by a dagger (†). Following the tables is a chart that lists the devices with their mnemonic and octal codes and DEC option numbers for both PDP-10 and PDP-6. A device mnemonic ending in the numeral 2 is the recommended form for the second of a given device, but such codes are not recognized by MACRO — they must be defined by the user.

<p>MOV { E e Negative e Magnitude e Swapped }</p> <p>Half word { Right Left } to { Right Left } { no effect Ones Zeros Extend sign }</p> <p>Block Transfer EXCHange AC and memory</p>	<p>ADD SUBtract MULTiply Integer MULTiply DIVide Integer DIVide</p> <p>Floating Add Floating SuBtract Floating MultiPly Floating DiVide</p> <p>Floating SScale Double Floating Negate Unnormalized Floating Add</p>
<p>use present pointer } and { LoAD Byte into AC Increment pointer } { DePosit Byte in memory Increment Byte Pointer</p>	<p>Arithmetic SHift } { ~ Logical SHift } { ~ ROTate } { Combined</p>
<p>SET to { Zeros Ones Ac Memory Complement of Ac Complement of Memory }</p> <p>AND inclusive OR } { ~ with Complement of AC with Complement of Memory Complements of Both }</p> <p>Inclusive OR eXclusive OR EQuiValence }</p> <p>to { AC AC Immediate Memory Both }</p>	<p>Jump { to SubRoutine and Save Pc and Save AC and Restore AC if Find First One on Flag and CLear it on OVerflow (JFCL 10,) on CaRrY 0 (JFCL 4,) on CaRrY 1 (JFCL 2,) on CaRrY (JFCL 6,) on Floating OVerflow (JFCL 1,) and ReSTore and ReSTore Flags (JRST 2,) and ENable PI channel (JRST 12,) }</p> <p>HALT (JRST 4.) eXeCuTe</p>
<p>SKIP if memory } JUMP if AC }</p> <p>Add One to } { memory and Skip } Subtract One from } { AC and Jump } if { never Less Equal Less or Equal Always Greater Greater or Equal Not equal }</p> <p>Compare AC { Immediate with Memory } and skip if AC</p> <p>Add One to Both halves of AC and Jump if { Positive Negative }</p>	<p>DATA } BLock } { In Out }</p> <p>CONDitions { in and Skip if { all masked bits Zero some masked bit One }</p>
<p>Test AC { with Direct mask with Swapped mask Right with E Left with E } { No modification set masked bits to Zeros set masked bits to Ones Complement masked bits } and skip { never if all masked bits Equal 0 if Not all masked bits equal 0 Always }</p>	

INSTRUCTION MNEMONICS

NUMERIC LISTING

000	ILLEGAL	132	FSC	206	MOVSM	
001	} USER : UO'S :	133	IBP	207	MOVSS	
037		134	ILDB	210	MOVN	
040		*CALL	135	LDB	211	MOVNI
041		*INIT	136	IDPB	212	MOVNM
042	} RESERVED : FOR : SPECIAL : MONITORS	137	DPB	213	MOVNS	
043		140	FAD	214	MOVMM	
044		141	FADL	215	MOVMI	
045		142	FADM	216	MOVMM	
046		143	FADB	217	MOVMS	
047	*CALLI	144	FADR	220	IMUL	
050	*OPEN	145	FADRI ▲	221	IMULI	
051	*TTCALL ▲	146	FADRM	222	IMULM	
052	} RESERVED : FOR DEC	147	FADRB	223	IMULB	
053		150	FSB	224	MUL	
054		151	FSBL	225	MULI	
055		*RENAME	152	FSBM	226	MULM
056	*IN	153	FSBB	227	MULB	
057	*OUT	154	FSBR	230	IDIV	
060	*SETSTS	155	FSBRI ▲	231	IDIVI	
061	*STATO	156	FSBRM	232	IDIVM	
062	*STATUS	157	FSBRB	233	IDIVB	
062	*GETSTS	160	FMP	234	DIV	
063	*STATZ	161	FMPL	235	DIVI	
064	*INBUF	162	FMPM	236	DIVM	
065	*OUTBUF	163	FMPB	237	DIVB	
066	*INPUT	164	FMPR	240	ASH	
067	*OUTPUT	165	FMPRI ▲	241	ROT	
070	*CLOSE	166	FMPRM	242	LSH	
071	*RELEAS	167	FMPRB	243	JFFO	
072	*MTAPE	170	FDV	244	ASHC	
073	*UGETF	171	FDVL	245	ROTC	
074	*USETI	172	FDVM	246	LSHC	
075	*USETO	173	FDVB	247		
076	*LOOKUP	174	FDVR	250	EXCH	
077	*ENTER	175	FDVRI ▲	251	BLT	
100	} UNASSIGNED : CODES :	176	FDVRM	252	AOBJP	
127		177	FDVRB	253	AOBJN	
130		200	MOVE	254	JRST	
131		201	MOVEI	25410	JRSTF	
		202	MOVEM	25420	HALT	
		203	MOVES	25450	JEN	
	UFA	204	MOVSI	255	JFCL	
	DFN	205		25504	JFOV	

A4

MNEMONICS

25510	JCRYI	333	SKIPLE	410	ANDCA
25520	JCRY0	334	SKIPA	411	ANDCAI
25530	JCRY	335	SKIPGE	412	ANDCAM
25540	JOV	336	SKIPN	413	ANDCAB
256	XCT	337	SKIPG	414	SETM
257		340	AOJ	415	SETMI
260	PUSHJ	341	AOJL	416	SETMM
261	PUSH	342	AOJE	417	SETMB
262	POP	343	AOJLE	420	ANDCM
263	POPJ	344	AOJA	421	ANDCMI
264	JSR	345	AOJGE	422	ANDCMM
265	JSP	346	AOJN	423	ANDCMB
266	JSA	347	AOJG	424	SETA
267	JRA	350	AOS	425	SETAI
270	ADD	351	AOSL	426	SETAM
271	ADDI	352	AOSE	427	SETAB
272	ADDM	353	AOSLE	430	XOR
273	ADDB	354	AOSA	431	XORI
274	SUB	355	AOSGE	432	XORM
275	SUBI	356	AOSN	433	XORB
276	SUBM	357	AOSG	434	IOR
277	SUBB	360	SOJ	434	OR
300	CAI	361	SOJL	435	IORI
301	CAIL	362	SOJE	435	ORI
302	CAIE	363	SOJLE	436	IORM
303	CAILE	364	SOJA	436	ORM
304	CAIA	365	SOJGE	437	IORB
305	CAIGE	366	SOJN	437	ORB
306	CAIN	367	SOJG	440	ANDCB
307	CAIG	370	SOS	441	ANDCBI
310	CAM	371	SOSL	442	ANDCBM
311	CAML	372	SOSE	443	ANDCBB
312	CAME	373	SOSLE	444	EQV
313	CAMLE	374	SOSA	445	EQVI
314	CAMA	375	SOSGE	446	EQVM
315	CAMGE	376	SOSN	447	EQVB
316	CAMN	377	SOSG	450	SETCA
317	CAMG	400	SETZ	451	SETCAI
320	JUMP	400	CLEAR	452	SETCAM
321	JUMPL	401	SETZI	453	SETCAB
322	JUMPE	401	CLEARI	454	ORCA
323	JUMPLE	402	SETZM	455	ORCAI
324	JUMPA	402	CLEARM	456	ORCAM
325	JUMPGE	403	SETZB	457	ORCAB
326	JUMPN	403	CLEARB	460	SETCM
327	JUMPG	404	AND	461	SETCMI
330	SKIP	405	ANDI	462	SETCMM
331	SKIPL	406	ANDM	463	SETCMB
332	SKIPE	407	ANDB	464	ORCM

NUMERIC LISTING

465	ORCMI	546	HLRM	627	TLZN
466	ORCMM	547	HLRS	630	TDZ
467	ORCMB	550	HRRZ	631	TSZ
470	ORCB	551	HRRZI	632	TDZE
471	ORCBI	552	HRRZM	633	TSZE
472	ORCBM	553	HRRZS	634	TDZA
473	ORCBB	554	HLRZ	635	TSZA
474	SETO	555	HLRZI	636	TDZN
475	SETOI	556	HLRZM	637	TSZN
476	SETOM	557	HLRZS	640	TRC
477	SETOB	560	HRRO	641	TLC
500	HLL	561	HRROI	642	TRCE
501	HLLI	562	HRROM	643	TLCE
502	HLLM	563	HRROS	644	TRCA
503	HLLS	564	HLRO	645	TLCA
504	HRL	565	HLROI	646	TRCN
505	HRLI	566	HLROM	647	TLCN
506	HRLM	567	HLROS	650	TDC
507	HRLS	570	HRRE	651	TSC
510	HLLZ	571	HRREI	652	TDCE
511	HLLZI	572	HRREM	653	TSCE
512	HLLZM	573	HRRES	654	TDCA
513	HLLZS	574	HLRE	655	TSCA
514	HRLZ	575	HLREI	656	TDCN
515	HRLZI	576	HLREM	657	TSCN
516	HRLZM	577	HLRES	660	TRO
517	HRLZS	600	TRN	661	TLO
520	HLLO	601	TLN	662	TROE
521	HLLOI	602	TRNE	663	TLOE
522	HLLOM	603	TLNE	664	TROA
523	HLLOS	604	TRNA	665	TLOA
524	HRLO	605	TLNA	666	TRON
525	HRLOI	606	TRNN	667	TLON
526	HRLOM	607	TLNN	670	TDO
527	HRLOS	610	TDN	671	TSO
530	HLLE	611	TSN	672	TDOE
531	HLLEI	612	TDNE	673	TSOE
532	HLLEM	613	TSNE	674	TDOA
533	HLLES	614	TDNA	675	TSOA
534	HRLE	615	TSNA	676	TDON
535	HRLEI	616	TDNN	677	TSON
536	HRLEM	617	TSNN	70000	BLKI
537	HRLES	620	TRZ	70004	DATAI
540	HRR	621	TLZ	70004	RSW
541	HRRI	622	TRZE	70010	BLKO
542	HRRM	623	TLZE	70014	DATAO
543	HRRS	624	TRZA	70020	CONO
544	HLR	625	TLZA	70024	CONI
545	HLRI	626	TRZN	70030	CONSZ
				70034	CONSO

INSTRUCTION MNEMONICS

ALPHABETIC LISTING

†ADC	024	BLT	251	DIVM	236
ADD	270	CAI	300	†DLS	240
ADDB	273	CAIA	304	DPB	137
ADDI	271	CAIE	302	▲ †DPC	250
ADDM	272	CAIG	307	†DSK	170
AND	404	CAIGE	305	†DTC	320
ANDB	407	CAII	301	†DTS	324
ANDCA	410	CAILE	303	*ENTER	077
ANDCAB	413	CAIN	306	EQV	444
ANDCAI	411	*CALL	040	EQVB	447
ANDCAM	412	*CALLI	047	EQVI	445
ANDCB	440	CAM	310	EQVM	446
ANDCBB	443	CAMA	314	EXCH	250
ANDCBI	441	CAME	312	FAD	140
ANDCBM	442	CAMG	317	FADB	143
ANDCM	420	CAMGE	315	FADL	141
ANDCMB	423	CAML	311	FADM	142
ANDCMI	421	CAMLE	313	FADR	144
ANDCMM	422	CAMN	316	FADRB	147
ANDI	405	†CCI	014	FADRI	145
ANDM	406	†CDP	110	FADRM	146
AOBJN	253	†CDR	114	FDV	170
AOBJP	252	CLEAR	400	FDVB	173
AOJ	340	CLEARB	403	FDVL	171
AOJA	344	CLEARI	401	FDVM	172
AOJE	342	CLEARM	402	FDVR	174
AOJG	347	*CLOSE	070	FDVRB	177
AOJGE	345	CONI	70024	FDVRI	175
AOJL	341	CONO	70020	FDVRM	176
AOJLE	343	CONSO	70034	FMP	160
AOJN	346	CONSZ	70030	FMPB	163
AOS	350	†CPA	000	FMPL	161
AOSA	354	†CR	150	FMPM	162
AOSE	352	DATAI	70004	FMPR	164
AOSG	357	DATAO	70014	FMPRB	167
AOSGE	355	†DC	200	FMPRI	165
AOSL	351	†DCSA	300	FMPRM	166
AOSLE	353	†DCSB	304	FSB	150
AOSN	356	†DF	270	FSBB	153
†APR	000	DFN	131	FSBL	151
ASH	240	†DIS	130	FSBM	152
ASHC	244	DIV	234	FSBR	154
BLKI	70000	DIVB	237	FSBRB	157
BLKO	70010	DIVI	235	FSBRI	155

ALPHABETIC LISTING

A7

FSBRM	156	HRLZI	515	JSA	266
FSC	132	HRLZM	516	JSP	265
*GETSTS	062	HRLZS	517	JSR	264
HALT	25420	HRR	540	JUMP	320
HLL	500	HRRE	570	JUMPA	324
HLLE	530	HRREI	571	JUMPE	322
HLLEI	531	HRREM	572	JUMPG	327
HLLEM	532	HRRES	573	JUMPGE	325
HLLES	533	HRRI	541	JUMPL	321
HLLI	501	HRRM	542	JUMPLE	323
HLLM	502	HRRO	560	JUMPN	326
HLLO	520	HRROI	561	LDB	135 ▲
HLLOI	521	HRROM	562	*LOOKUP	076
HLLOM	522	HRROS	563	†LPT	124
HLLOS	523	HRRS	543	LSH	242
HLLS	503	HRRZ	550	LSHC	246
HLLZ	510	HRRZI	551	†MDF	260
HLLZI	511	HRRZM	552	MOVE	200
HLLZM	512	HRRZS	553	MOVEI	201
HLLZS	513	IBP	133	MOVEM	202
HLR	544	IDIV	230	MOVES	203
HLRE	574	IDIVB	233	MOVMI	214
HLREI	575	IDIVI	231	MOVMI	215
HLREM	576	IDIVM	232	MOVMM	216
HLRES	577	IDPB	136	MOVMS	217
HLRI	545	ILDB	134	MOVN	210
HLRM	546	IMUL	220	MOVNI	211
HLRO	564	IMULB	223	MOVNM	212
HLROI	565	IMULI	221	MOVNS	213
HLROM	566	IMULM	222	MOVSI	204
HLROS	567	*IN	056	MOVSM	206
HLRS	547	*INBUF	064	MOVSS	207
HLRZ	554	*INIT	041	*MTAPE	072
HLRZI	555	*INPUT	066	†MTC	220
HLRZM	556	IOR	434	†MTM	230
HLRZS	557	IORB	437	†MTS	224
HRL	504	IORI	435	MUL	224
HRLE	534	IORM	436	MULB	227
HRLEI	535	JCRY	25530	MULI	225
HRLEM	536	JCRY0	25520	MULM	226
HRLES	537	JCRY1	25510	*OPEN	050
HRLI	505	JEN	25460	OR	434
HRLM	506	JFCL	255	ORB	437
HRLO	524	JFFO	243	ORCA	454
HRLOI	525	JFOV	25504	ORCAB	457
HRLOM	526	JOV	25540	ORCAI	455
HRLOS	527	JRA	267	ORCAM	456
HRLS	507	JRST	254	ORCB	470
HRLZ	514	JRSTF	25410		

A8

MNEMONICS

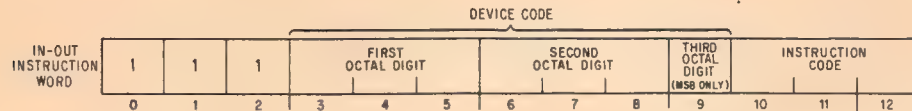
ORCBB	473	SETZM	402	TLCA	645
ORCBI	471	SKIP	330	TLCE	643
ORCBM	472	SKIPA	334	TLCN	647
ORCM	464	SKIPE	332	TLN	601
ORCMB	467	SKIPG	337	TLNA	605
ORCMI	465	SKIPGE	335	TLNE	603
ORCMM	466	SKIPL	331	TLNN	607
ORI	435	SKIPLI	333	TLO	661
ORM	436	SKIPN	336	TLOA	665
*OUT	057	SOJ	360	TLOE	663
*OUTBUF	065	SOJA	364	TLON	667
*OUTPUT	067	SOJE	362	TLZ	621
†PI	004	SOJG	367	TLZA	625
†PLT	140	SOJGE	365	TLZE	623
POP	262	SOJL	361	TLZN	627
POPJ	263	SOJLE	363	†TMC	340
†PTP	100	SOJN	366	†TMS	344
†PTR	104	SOS	370	TRC	640
PUSH	261	SOSA	374	TRCA	644
PUSHJ	260	SOSE	372	TRCE	642
*RELEAS	071	SOSG	377	TRCN	646
*RENAME	055	SOSGE	375	TRN	600
ROT	241	SOSL	371	TRNA	604
ROTC	245	SOSLE	373	TRNE	602
RSW	70004	SOSN	376	TRNN	606
SETA	424	*STATO	061	TRO	660
SETAB	427	*STATUS	062	TROA	664
SETAI	425	*STATZ	063	TROE	662
SETAM	426	SUB	274	TRON	666
SETCA	450	SUBB	277	TRZ	620
SETCAB	453	SUBI	275	TRZA	624
SETCAI	451	SUBM	276	TRZE	622
SETCAM	452	TDC	650	TRZN	626
SETCM	460	TDCA	654	TSC	651
SETCMB	463	TDCE	652	TSCA	655
SETCMI	461	TDCN	656	TSCE	653
SETCMM	462	TDN	610	TSCN	657
SETM	414	TDNA	614	TSN	611
SETMB	417	TDNE	612	TSNA	615
SETMI	415	TDNN	616	TSNE	613
SETMM	416	TDO	670	TSNN	617
SETO	474	TDOA	674	TSO	671
SETOB	477	TDOE	672	TSOA	675
SETOI	475	TDON	676	TSOE	673
SETOM	476	TDZ	630	TSON	677
*SETSTS	060	TDZA	634	TSZ	631
SETZ	400	TDZE	632	TSZA	635
SETZB	403	TDZN	636	TSZE	633
SETZI	401	TLC	641	TSZN	637

ALPHABETIC LISTING

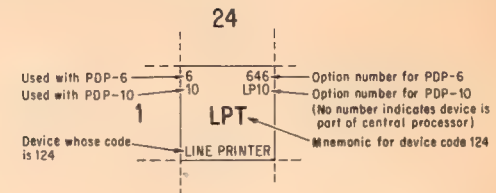
A9

*TTCALL	051	▲	*USETO	075	XOR	430
UFA	130		†UTC	210	XORB	433
*UGETF	073		†UTS	214	XORI	431
*USETI	074		XCT	256	XORM	432

		SECOND AND THIRD OCTAL DIGITS		00	04	10	14	20	24	30	34	40	44	50	54	60	64	70	74		
DEC STANDARD DEVICES	FIRST OCTAL DIGIT 0	6,10 APR CENTRAL PROCESSOR	6,10 PI PRIORITY INTERRUPT	6 DRUM PROCESSOR	167 CCI PDP-8,9 INTERFACE	10 DA10 CCI2 PDP-8,9 INTERFACE	10 DA10 ADC ANALOG-DIGITAL CONVERTER	10 AD10 ADC2 ANALOG-DIGITAL CONVERTER													
	1	6 10 PTP PAPER TAPE PUNCH	6 10 761 PTR PAPER TAPE READER	10 CP10 CDP CARD PUNCH	6 461 CDR CARD READER	6 626 TTY CONSOLE TELETYPE	6 646 LPT LINE PRINTER	6 10 340 DIS DISPLAY	6 10 340 DIS2 DISPLAY	10 XY10 PLT PLOTTER	10 XY10 PLT2 PLOTTER	10 CR10 CR CARD READER	10 CR10 CR2 CARD READER	6 165 PDP-7,8 INTERFACE				10 RC10 DSK SMALL DISK	10 RC10 DSK2 SMALL DISK		
	2	6 136 DC DATA CONTROL	6 136 DC2 DATA CONTROL	6 UTC DECTAPE	6 551 UTS DECTAPE	6 MTC MAGNETIC TAPE	6 516 MTS MAGNETIC TAPE	6 516 MTM MAGNETIC TAPE		10 DC10 DLS DATA LINE SCANNER	10 DC10 DLS2 DATA LINE SCANNER	10 RP10 DPC DISK PACK SYSTEM	10 RP10 DPC2 DISK PACK SYSTEM	10 RA10 MDF MASS DISK FILE	10 RA10 MDF2 MASS DISK FILE	6 270 DF DISK FILE					
USER SPECIAL DEVICES	3	6 DCSA DATA COMMUNICATION	6 30 DCSB DATA COMMUNICATION			10 TD10 DTC DECTAPE	10 TD10 DTS DECTAPE	10 TD10 DTC2 DECTAPE	10 TD10 DTS2 DECTAPE	10 TM10 TMC MAGNETIC TAPE	10 TM10 TMS MAGNETIC TAPE	10 TM10 TMC2 MAGNETIC TAPE	10 TM10 TMS2 MAGNETIC TAPE								
	4																				
	5																				
	6																				
7																					



DEVICE MNEMONICS



APPENDIX B

INPUT-OUTPUT CODES

The table beginning on the next page lists the complete teletype code. The lower case character set (codes 140-176) is not available on the Model 35, but giving one of these codes causes the teletype to print the corresponding upper case character. Other differences between the 35 and 37 are mentioned in the table. The definitions of the control codes are those given by ASCII. Most control codes, however, have no effect on the console teletype, and the definitions bear no necessary relation to the use of the codes in conjunction with the PDP-10 software.

The line printer has the same codes and characters as the teletype. The 64-character printer has the figure and upper case sets, codes 040-137 (again, giving a lower case code prints the upper case character). The "96"-character printer has these plus the lower case set, codes 040-176. The latter printer actually has only ninety-five characters unless a special character is "hidden" under the delete code, 177. A hidden character is printed by sending its code prefixed by the delete code. Hence a character hidden under DEL is printed by sending the printer two 177s in a row.

Besides printing characters, the line printer responds to ten control characters, HT, CR, LF, VT, FF, DLE and DC1-4. The 128-character printer uses the entire set of 7-bit codes for printable characters, with characters hidden under the ten control characters that affect the printer and also under null and delete. In all cases, prefixing DEL causes the hidden character to be printed. The extra thirty-three characters that complete the set are ordered special for each installation.

The first page of the table of card codes [*pages B6-8*] lists the column punch required to represent any character in the two DEC codes. The octal codes listed are those used by the PDP-10 software. In other words, when reading cards, the Monitor translates the column punch into the octal code shown; when punching cards, it produces the listed column punch when given the corresponding code. The remaining pages of the table show the relationship between the DEC card codes and several IBM card punches. Each of the column punches is produced by a single key on any punch for which a character is listed, the character being that which is printed at the top of the card.

INPUT-OUTPUT CODES

B2

TELETYPE CODE

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	000	NUL	Null, tape feed. Repeats on Model 37. Control shift P on Model 35.
1	001	SOH	Start of heading; also SOM, start of message. Control A.
1	002	STX	Start of text; also EOA, end of address. Control B.
0	003	ETX	End of text; also EOM, end of message. Control C.
1	004	EOT	End of transmission (END); shuts off TWX machines. Control D.
0	005	ENQ	Enquiry (ENQRY); also WRU, "Who are you?" Triggers identification ("Here is . . .") at remote station if so equipped. Control E.
0	006	ACK	Acknowledge; also RU, "Are you . . .?" Control F.
1	007	BEL	Rings the bell. Control G.
1	010	BS	Backspace; also FEO, format effector. Backspaces some machines. Repeats on Model 37. Control H on Model 35.
0	011	HT	Horizontal tab. Control I on Model 35.
0	012	LF	Line feed or line space (NEW LINE); advances paper to next line. Repeats on Model 37. Duplicated by control J on Model 35.
1	013	VT	Vertical tab (VTAB). Control K on Model 35.
0	014	FF	Form feed to top of next page (PAGE). Control L.
1	015	CR	Carriage return to beginning of line. Control M on Model 35.
1	016	SO	Shift out; changes ribbon color to red. Control N.
0	017	SI	Shift in; changes ribbon color to black. Control O.
1	020	DLE	Data link escape. Control P (DC0).
0	021	DC1	Device control 1, turns transmitter (reader) on. Control Q (X ON).
0	022	DC2	Device control 2, turns punch or auxiliary on. Control R (TAPE, AUX ON).
1	023	DC3	Device control 3, turns transmitter (reader) off. Control S (X OFF).
0	024	DC4	Device control 4, turns punch or auxiliary off. Control T (TAPE, AUX OFF).
1	025	NAK	Negative acknowledge; also ERR, error. Control U.
1	026	SYN	Synchronous idle (SYNC). Control V.
0	027	ETB	End of transmission block; also LEM, logical end of medium. Control W.
0	030	CAN	Cancel (CANCL). Control X.
1	031	EM	End of medium. Control Y.
1	032	SUB	Substitute. Control Z.
0	033	ESC	Escape, prefix. This code is generated by control shift K on Model 35, but the Monitor translates it to 175.
1	034	FS	File separator. Control shift L on Model 35.
0	035	GS	Group separator. Control shift M on Model 35.

Even Parity Bit	7-Bit Octal Code	Character	Remarks
0	036	RS	Record separator. Control shift N on Model 35.
1	037	US	Unit separator. Control shift O on Model 35.
1	040	SP	Space.
0	041	!	
0	042	"	
1	043	#	
0	044	\$	
1	045	%	
1	046	&	
0	047	'	Accent acute or apostrophe.
0	050	(
1	051)	
1	052	*	Repeats on Model 37.
0	053	+	
1	054	,	
0	055	-	Repeats on Model 37.
0	056	.	Repeats on Model 37.
1	057	/	
0	060	∅	
1	061	1	
1	062	2	
0	063	3	
1	064	4	
0	065	5	
0	066	6	
1	067	7	
1	070	8	
0	071	9	
0	072	:	
1	073	;	
0	074	<	
1	075	=	Repeats on Model 37.
1	076	>	
0	077	?	
1	100	@	
0	101	A	
0	102	B	

B4

INPUT-OUTPUT CODES

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	103	C	
0	104	D	
1	105	E	
1	106	F	
0	107	G	
0	110	H	
1	111	I	
1	112	J	
0	113	K	
1	114	L	
0	115	M	
0	116	N	
1	117	O	
0	120	P	
1	121	Q	
1	122	R	
0	123	S	
1	124	T	
0	125	U	
0	126	V	
1	127	W	
1	130	X	Repeats on Model 37.
0	131	Y	
0	132	Z	
1	133	[Shift K on Model 35.
0	134	\	Shift L on Model 35.
1	135]	Shift M on Model 35.
1	136	↑	
0	137	←	Repeats on Model 37.
0	140	`	Accent grave.
1	141	a	
1	142	b	
0	143	c	
1	144	d	
0	145	e	
0	146	f	
1	147	g	

Even Parity Bit	7-Bit Octal Code	Character	Remarks
1	150	h	
0	151	i	
0	152	j	
1	153	k	
0	154	l	
1	155	m	
1	156	n	
0	157	o	
1	160	p	
0	161	q	
0	162	r	
1	163	s	
0	164	t	
1	165	u	
1	166	v	
0	167	w	
0	170	x	Repeats on Model 37.
1	171	y	
1	172	z	
0	173	{	
1	174		
0	175	}	This code generated by ALT MODE on Model 35. ▲
0	176	~	This code generated by ESC key (if present) on Model 35, but the Monitor translates it to 175. ▲
1	177	DEL	Delete, rub out. Repeats on Model 37.

Keys That Generate No Codes

REPT	Model 35 only: causes any other key that is struck to repeat continuously until REPT is released.
PAPER ADVANCE	Model 37 local line feed.
LOCAL RETURN	Model 37 local carriage return.
LOC LF	Model 35 local line feed.
LOC CR	Model 35 local carriage return.
INTERRUPT, BREAK	Opens the line (machine sends a continuous string of null characters).
PROCEED, BRK RLS	Break release (not applicable).
HERE IS	Transmits predetermined 21-character message.

B6

INPUT-OUTPUT CODES

CARD CODES

▲ Character	PDP-10 ASCII	DEC 029	DEC 026	Character	PDP-10 ASCII	DEC 029	DEC 026
Space	040	None	None	@	100	8 4	8 4
!	041	11 8 2	12 8 7	A	101	12 1	12 1
"	042	8 7	0 8 5	B	102	12 2	12 2
#	043	8 3	0 8 6	C	103	12 3	12 3
\$	044	11 8 3	11 8 3	D	104	12 4	12 4
%	045	0 8 4	0 8 7	E	105	12 5	12 5
&	046	12	11 8 7	F	106	12 6	12 6
'	047	8 5	8 6	G	107	12 7	12 7
(050	12 8 5	0 8 4 ▲	H	110	12 8	12 8
)	051	11 8 5	12 8 4 ▲	I	111	12 9	12 9
*	052	11 8 4	11 8 4	J	112	11 1	11 1
+	053	12 8 6	12	K	113	11 2	11 2
,	054	0 8 3	0 8 3	L	114	11 3	11 3
-	055	11	11	M	115	11 4	11 4
.	056	12 8 3	12 8 3	N	116	11 5	11 5
/	057	0 1	0 1	O	117	11 6	11 6
0	060	0	0	P	120	11 7	11 7
1	061	1	1	Q	121	11 8	11 8
2	062	2	2	R	122	11 9	11 9
3	063	3	3	S	123	0 2	0 2
4	064	4	4	T	124	0 3	0 3
5	065	5	5	U	125	0 4	0 4
6	066	6	6	V	126	0 5	0 5
7	067	7	7	W	127	0 6	0 6
8	070	8	8	X	130	0 7	0 7
9	071	9	9	Y	131	0 8	0 8
:	072	8 2	11 8 2 or 11 0	Z	132	0 9	0 9
;	073	11 8 6	0 8 2	[133	12 8 2	11 8 5
<	074	12 8 4	12 8 6	\	134	11 8 7	8 7
=	075	8 6	8 3]	135	0 8 2	12 8 5
>	076	0 8 6	11 8 6	↑	136	12 8 7	8 5
?	077	0 8 7	12 8 2 or 12 0	←	137	0 8 5	8 2
Binary	7 9						
Mode Switch	12 0 2 4 6 8						
End of File	12 11 0 1						

The octal codes given above are those generated by the Monitor from the column punches. The card reader interface actually supplies a direct binary equivalent of the column punch, as listed in the following two pages.

CARD CODES

B7

Column Punch	Character	Octal	Column Punch	Character	Octal
<i>None</i>	<i>Space</i>	0000	12 9	I	4001
0	0	1000	11 1	J	2400
1	1	0400	11 2	K	2200
2	2	0200	11 3	L	2100
3	3	0100	11 4	M	2040
4	4	0040	11 5	N	2020
5	5	0020	11 6	O	2010
6	6	0010	11 7	P	2004
7	7	0004	11 8	Q	2002
8	8	0002	11 9	R	2001
9	9	0001	0 1	/	1400
12 1	A	4400	0 2	S	1200
12 2	B	4200	0 3	T	1100
12 3	C	4100	0 4	U	1040
12 4	D	4040	0 5	V	1020
12 5	E	4020	0 6	W	1010
12 6	F	4010	0 7	X	1004
12 7	G	4004	0 8	Y	1002
12 8	H	4002	0 9	Z	1001

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12	&	+	&	+	&	4000
11	-	-	-	-	-	2000
12 0				?		5000
11 0				:		3000
8 2			:	←	:	0202
8 3	#	=	#	=	#	0102
8 4	@	-	@	@	@	0042
8 5			'	↑	'	0022
8 6			=	'	=	0012
8 7			"	\	"	0006
12 8 2			¢	?	[4202
12 8 3			.	.	.	4102
12 8 4	□)	<)	<	4042
12 8 5			([(4022
12 8 6			+	<	+	4012

B8

INPUT-OUTPUT CODES

Column Punch	026 Data Processing	026 Fortran	029	DEC 026	DEC 029	Octal
12 8 7				!	↑	4006
11 8 2				:	!	2202
11 8 3	\$	\$	\$	\$	\$	2102
11 8 4	*	*	*	*	*	2042
11 8 5)	[)	2022
11 8 6			;	>	;	2012
11 8 7			⌋	&	\	2006
0 8 2			<i>See note</i>	;]	1202
0 8 3	,	,	,	,	,	1102
0 8 4	%	(%	(%	1042
0 8 5			←	"	←	1022
0 8 6			>	#	>	1012
0 8 7			?	%	?	1006
12 11 0 1				<i>End of File</i>	<i>End of File</i>	7400
12 0 2 4 6 8				<i>Mode Switch</i>	<i>Mode Switch</i>	5252
7 9				<i>Binary</i>	<i>Binary</i>	xx05

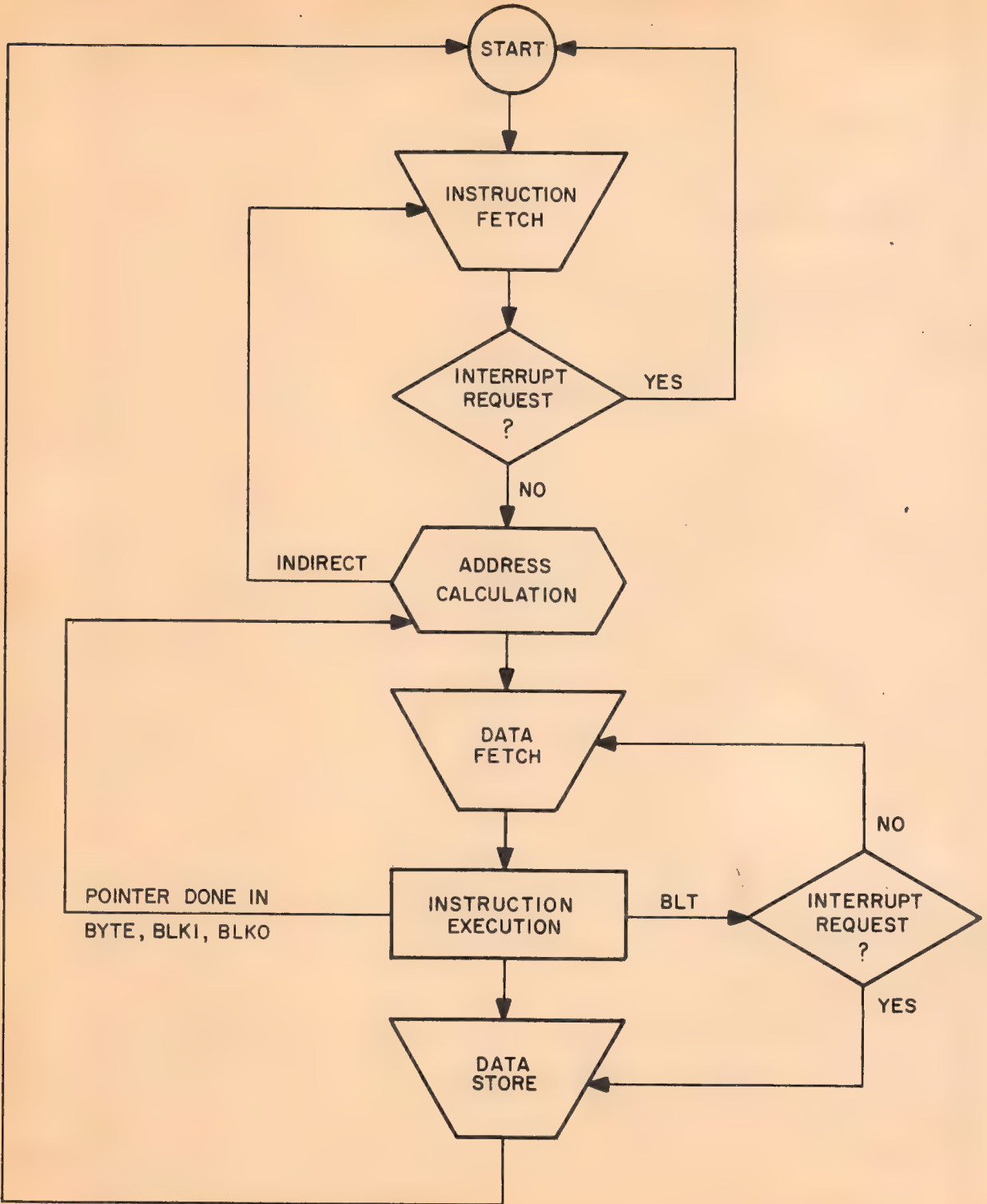
NOTE: There is a single key for the 0 8 2 punch on the 029 but printing is suppressed.

The Monitor translates the octal code for the 12 0 punch in DEC 026 to 4202 (which corresponds to a 12 8 2 punch), and the code for 11 0 to 2202 (11 8 2).

APPENDIX C

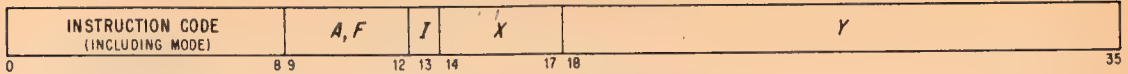
MISCELLANY

Instruction Flow Simplified	C2
Word Formats	C3
Instruction Timing Flow Chart	C4
In-out Device Bit Assignments	C6
Indicator Panels	C8
Powers of Two	C10

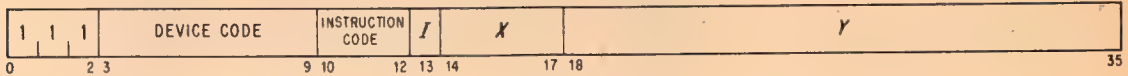


INSTRUCTION FLOW SIMPLIFIED

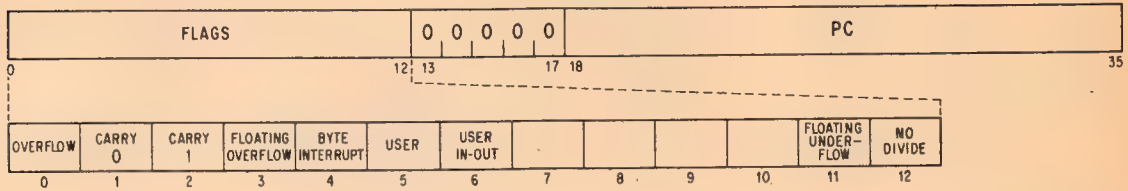
BASIC INSTRUCTIONS



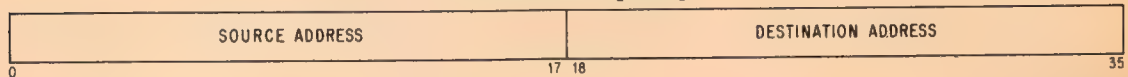
IN-OUT INSTRUCTIONS



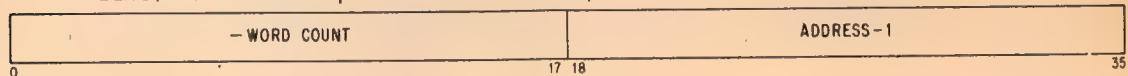
PC WORD



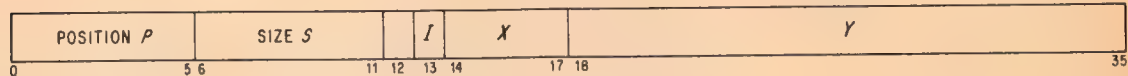
BLT POINTER [XWD]



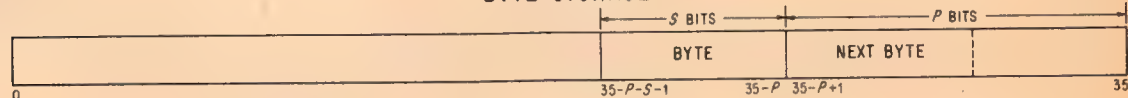
BLKI / BLKO POINTER, PUSHDOWN POINTER, DATA CHANNEL CONTROL WORD [IOWD]



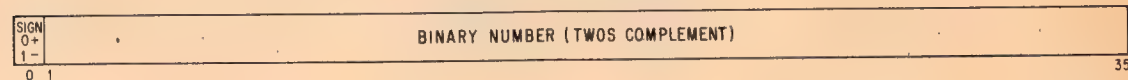
BYTE POINTER



BYTE STORAGE



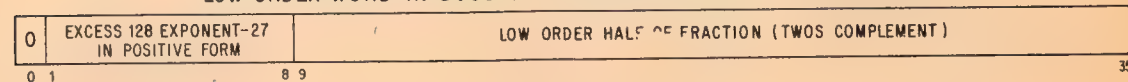
FIXED POINT OPERANDS



FLOATING POINT OPERANDS

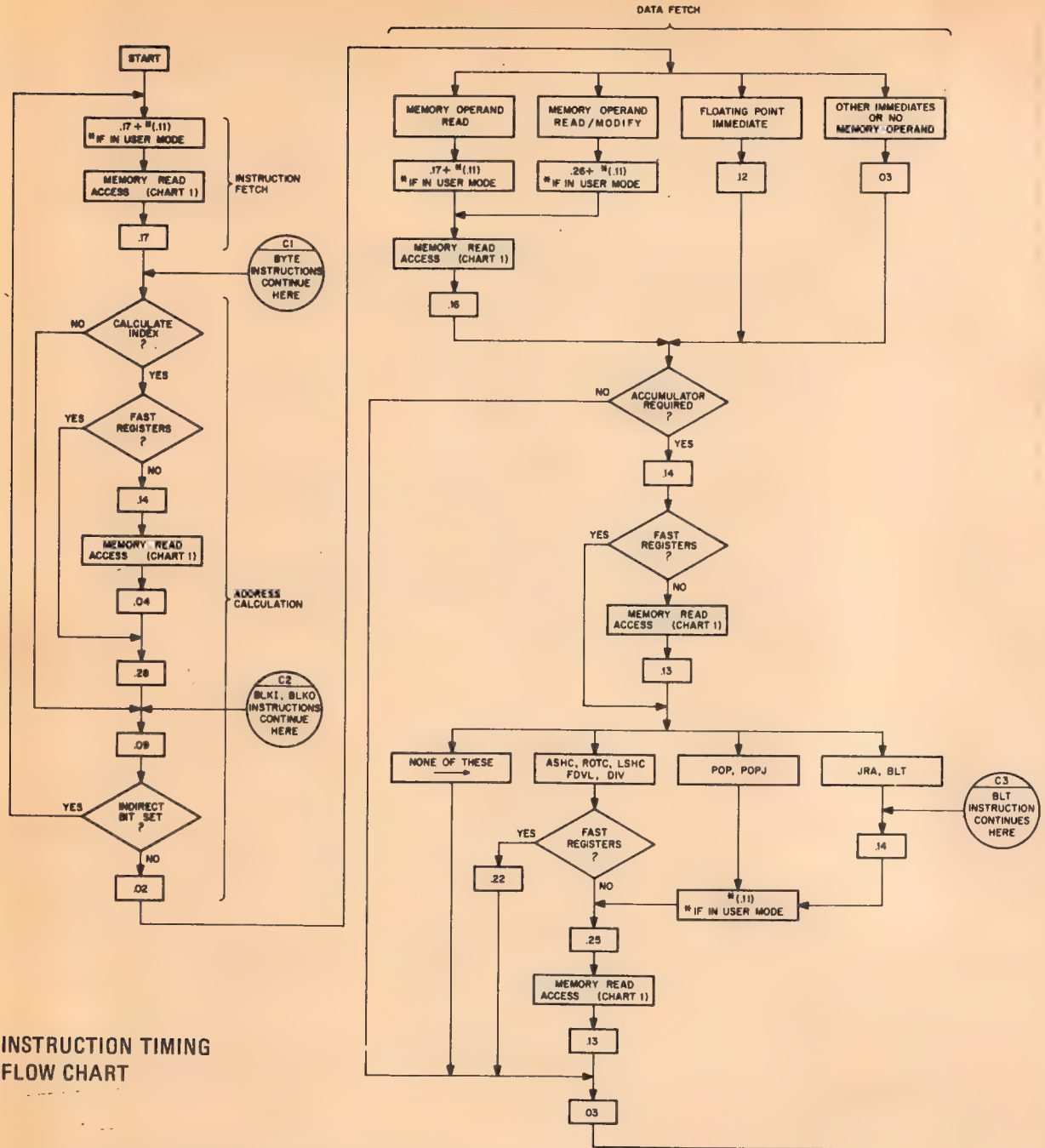


LOW ORDER WORD IN DOUBLE LENGTH FLOATING POINT OPERANDS



WORD FORMATS

C4



**INSTRUCTION TIMING
FLOW CHART**

INSTRUCTIONS THAT USE READ/MODIFY

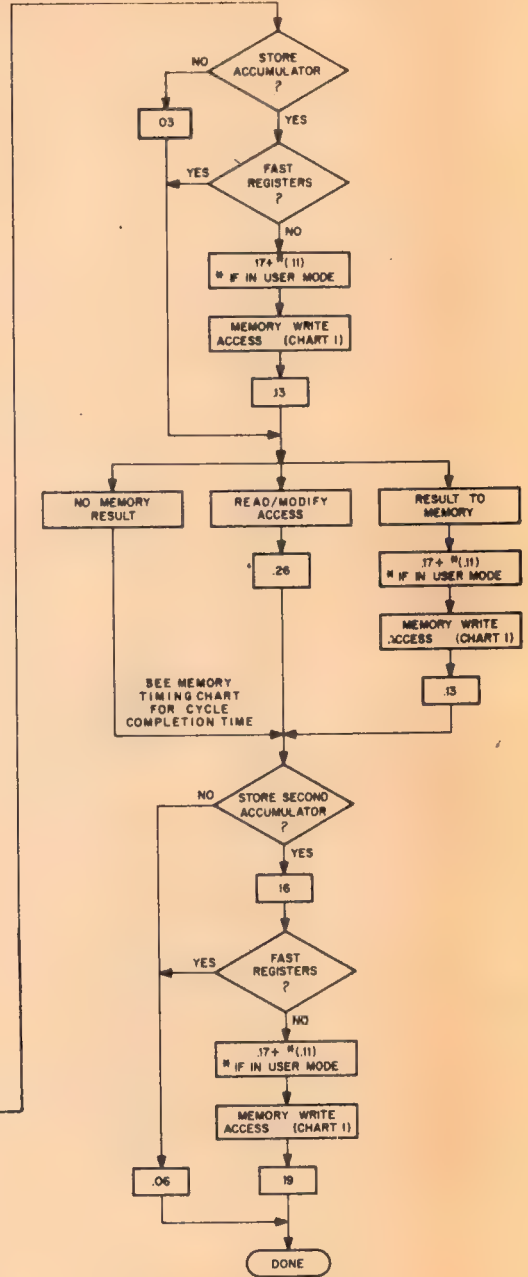
All Boolean In Memory and Both Modes Except SETZ, SETA, SETCA, SETO
 ADDM, ADDB, SUBM, SUBB
 HRRM, HRLM, HLRM, HLLM and All Halt Words In Self Mode
 MOVES, MOVNS, MOVMS, MOVSS
 ILDB, IDPB (First Time Only)
 IBP, BLKI, BLKO, DFN, EXCH
 AOS, SOS in all modes

INSTRUCTION EXECUTION

Boolean (except ANDCA, ANDCB, ORCA, ORCB), Half Words (except HLR, HLRI, HRL, HRLI), MOVE, MOVVS, EXCH, JFCL, JRST, JSP, XCT, UUO	.27	
ANDCA, ANDCB, ORCA, ORCB, HLR, HLRI, HRL, HRLI, JSR, JSA, JRA, Test class	.62	
MOVN, MOVN, ADD, SUB, AOBJP, AOBJN, CAM, CAI, SKIP, JUMP, ADJ, AOS, SOJ, SOS	.45	
PUSH, PUSHJ, POP, POPJ, DFN	.80	
JFFD	.80	+ .19 times number of leading 0s mod 18
BLT	.69	(+ .11 if User) + memory write access + .52 If not done + .09 and go to C3
IBP	.38	+ .26 if overflow word boundary
LDB, DPB	.61	+ .15 per size count Go to C1
ILDB, IDPB	.74	{ + .15 per size count } Go to C1 { + .26 if overflow }
ILDB, LDB	.45	+ .15 per position count
IDPB, DPB	.95	+ .15 per position count
Shift group	{ .39 Left } { .23 Right }	+ .15 per shift
MUL	6.02	+ .13 per transition
Average except MULI	8.36	(18 transitions for 2.34)
IMUL	6.34	+ .13 per transition
Average except IMULI	7.51	(9 transitions for 1.17)
FMP	6.39	+ .13 per transition
Average except FMPRI	8.21	(14 transitions for 1.82)
Note: Immediate mode multiplication has only half the average number of transitions		
DIV, IDIV	13.78	
FSC	1.52	+ .25 per shift to normalize
FAD, UFA	2.38	{ + .15 per shift to unnormalize } { + .25 per shift to normalize }
Average	4.33	
FSB	Same as FAD + .18	
Rounding (except divide) only when actually done	+.96	
Long mode (except divide)	+.69	
FDVR, FDV (except FDVL)	12.00	
FDVL with fast ACs	13.28	
FDVL without fast ACs	12.32 (+ .11 if User) + memory read access + .89	
COND, CONI, CONSO, CONSZ, DATA, DATAI	.12 Then wait until 4.50 has passed since last here	
CONO, CONI, DATAO, DATAI	+2.68	
CONSO, CONSZ	+2.90	
BLKO, BLKI	.80 Then turn into DATAO, DATAI end go to C2	

03

DATA STORE



MEMORY TIMING

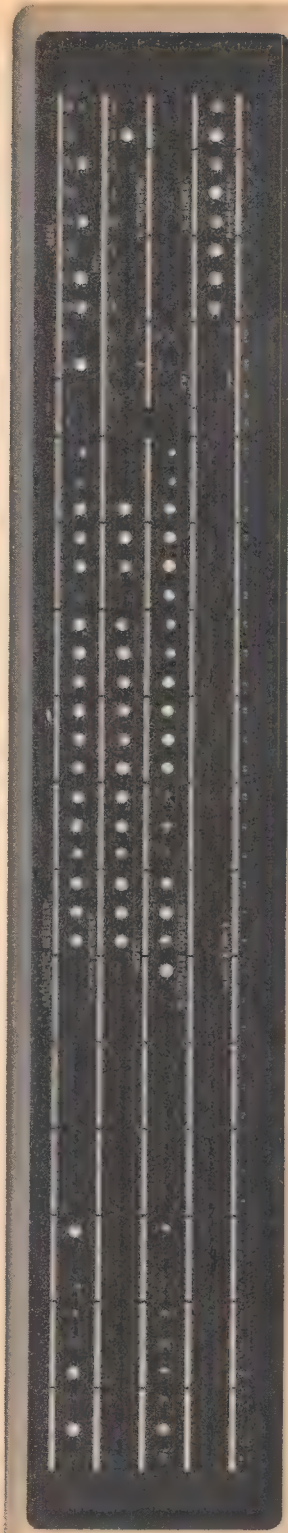
MEMORY	MA10	MB10	MB10	KM10
	SINGLE OR MULTI	SINGLE	MULTI	SINGLE (BUILT IN)
CYCLE	1.00	1.65	1.65	—
READ ACCESS	.58	.60	.70	.21
WRITE ACCESS	.20	.20	.30	.21
MODIFY COMPLETION	.35	.35	1.20	—

NOTES:
MEMORY ACCESS TIMES INCLUDE 20 FEET OF CABLE DELAY.
ALL TIMES ± 5%

DEVICE	CODE	FUNCTION	0/18	1/19	2/20	3/21	4/22	5/23	6/24	7/25	8/26	9/27	10/28	11/29	12/30	13/31	14/32	15/33	16/34	17/35				
DSK (RC10)	170	COMO	SELECT SECTOR CTR	CLEAR DISK DESIG ERROR	CLR TRACK SELECT ERROR	CLEAR DISK NOT READY	CLEAR PWR SUP FAILURE MAINT SELECT	CLR DISK PARE ERR	CLR CHAN DATA PARE ERR	CLR CHAN CONTROL PARE ERR	CLEAR NXM	CLEAR ILLEGAL WRITE	CLEAR OVER-RUN	WRITE CHAN CNTRL WD	STOP CLEAR BUSY	CLEAR DONE				PIA				
		CONI LH						PROTECT SECTOR LOWER														1'S		
		CONI RH	DATA XFER IN PROG	SEARCH ERROR	DISK DESIG ERROR	TRACK ERROR	DISK NOT READY	PWR SUPPLY FAIL	DISK PARE ERROR	CHANNEL DATA PARE ERR	CHANNEL CONTROL PARE ERR	NXM	ILLEGAL WRITE	OVER-RUN	CHAN CNTRL WD (WRITE)	BUSY	DONE					PIA		
		DATRO LH	DISK SELECT																				1'S	
		DATRO RH																					1'S	
		DATRI RH																					1'S	
DTC (TD10)	320	COMO	STOP	GO FORWARD	GO REVERSE	DELAY INHIBIT	SELECT	CLEAR SELECT # UNIT NR	TRANSPORT NUMBER	FUNCTION NUMBER											DATA PIA	FLAGS PIA		
		CONI	STOP	GO FORWARD	GO REVERSE		SELECT	DESELECT	TRANSPORT NUMBER													DATA PIA	FLAGS PIA	
		DATRO	36 BIT WORD																					
		DATRI	36 BIT WORD																					
DTS	324	COMO	PARITY ERR ENABLE	DATA MISSED	JOB DONE	ILLEGAL OPERATION	END ZONE	BLOCK MISSED														STOP ALL TRANSPORT	FUNCTION STOP	
		CONI LH	PARITY ERR ENABLE	DATA MISSED	JOB DONE	ILLEGAL OPERATION	END ZONE	BLOCK MISSED	DELAY IN PROGRESS	ACTIVE	UP TO SPEED	BLOCK NUMBER	REVERSE	DATA	FINAL DATA	CHECKSUM	IDLE	BLOCK NUMBER READ				PARITY	FUNCTION STOP	
		CONI RH	PARITY ERROR	DATA MISSED	JOB DONE	ILLEGAL OPERATION	END ZONE	BLOCK MISSED	WRITE LOCK ON	WRITE SWITCH ON	INCOMPLETE BLOCK	MARK TRACK ERROR	SELECT ERROR									PARITY	FUNCTION STOP	
		DATRO RH																					PARITY	FUNCTION STOP
DLS (DC10)	240	COMO																						
		CONI																						
		DATRO LH																						
DLS (DC10B/DC10E)	240	CONI																						
		DATRO LH																						
		DATRI RH																						
TMC (TM10)	340	COMO	UNIT	PARITY 0 = EVEN 1 = ODD	CORE DUMP																			
		CONI RH	UNIT	PARITY	CORE DUMP																			
		CONI LH																						
		DATRO LH	36 BIT WORD																					
		DATRO LH	(7 CHN: SIX 6 BIT CHARACTERS AS IN DATRI BELOW)																					
		DATRO LH	1 ST CHARACTER																					
TMS	344	COMO																						
		CONI LH																						
		CONI RH	SPORT HUNG	REWINDING	LOAD POINT	ILLEG OP	PAR ERROR	END OF FILE	END OF TAPE	READ COMPARE ERROR	RECORD LENGTH INCORRECT	DATA LATE	TAPE DONE	JOB DONE	TAPE UNIT ROT (TUR)	CHANNEL ERROR	WRITE LOCK	7-CHAN TAPE	NEXT UNIT	DATA REQ				
		DATRO LH	INITIAL CHANNEL CONTROL WORD ADDRESS																					
		DATRO LH	WRITE EVEN PARITY																					
		DATRO LH	INITIAL CHANNEL CONTROL WORD ADDRESS																					
CCI (DA10)	014	COMO																						
		CONI																						
		DATRO LH	1ST PDP 8 BYTE (A)																					
DATRO LH	36 BITS																							
DATRO LH	36 BITS																							
DATRO LH	1ST PDP 9 BYTE (A)																							
DATRO LH	2ND PDP 8 BYTE (B)																							
DATRO LH	3RD PDP 8 BYTE (C)																							
DATRO LH	2ND PDP 9 BYTE (B)																							



Indicator Panel, KA10 Arithmetic Processor, Bay 1



Indicator Panel, KA10 Arithmetic Processor, Bay 2



Indicator Panel, MB10 Core Memory (1.65 μ s)

C10

MISCELLANY

POWERS OF TWO

2^N	N	2^{-N}
1	0	1.0
2	1	0.5
4	2	0.25
8	3	0.125
16	4	0.062 5
32	5	0.031 25
64	6	0.015 625
128	7	0.007 812 5
256	8	0.003 906 25
512	9	0.001 953 125
1 024	10	0.000 976 562 5
2 048	11	0.000 488 281 25
4 096	12	0.000 244 140 625
8 192	13	0.000 122 070 312 5
16 384	14	0.000 061 035 156 25
32 768	15	0.000 030 517 578 125
65 536	16	0.000 015 258 789 062 5
131 072	17	0.000 007 629 394 531 25
262 144	18	0.000 003 814 697 265 625
524 288	19	0.000 001 907 348 632 812 5
1 048 576	20	0.000 000 953 674 316 406 25
2 097 152	21	0.000 000 476 837 158 203 125
4 194 304	22	0.000 000 238 418 579 101 562 5
8 388 608	23	0.000 000 119 209 289 550 781 25
16 777 216	24	0.000 000 059 604 644 775 390 625
33 554 432	25	0.000 000 029 802 322 387 695 312 5
67 108 864	26	0.000 000 014 901 161 193 847 656 25
134 217 728	27	0.000 000 007 450 580 596 923 828 125
268 435 456	28	0.000 000 003 725 290 298 461 914 062 5
536 870 912	29	0.000 000 001 862 645 149 230 957 031 25
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625
137 438 953 472	37	0.000 000 000 007 275 957 614 183 425 903 320 312 5
274 877 906 944	38	0.000 000 000 003 637 978 807 091 712 951 660 156 25
549 755 813 888	39	0.000 000 000 001 818 989 403 545 856 475 830 078 125
1 099 511 627 776	40	0.000 000 000 000 909 494 701 772 928 237 915 039 062 5
2 199 023 255 552	41	0.000 000 000 000 454 747 350 886 464 118 957 519 531 25
4 398 046 511 104	42	0.000 000 000 000 227 373 675 443 232 059 478 759 765 625
8 796 093 022 208	43	0.000 000 000 000 113 686 837 721 616 029 739 379 882 812 5
17 592 186 044 416	44	0.000 000 000 000 056 843 418 860 808 014 869 689 941 406 25
35 184 372 088 832	45	0.000 000 000 000 028 421 709 430 404 007 434 844 970 703 125
70 368 744 177 664	46	0.000 000 000 000 014 210 854 715 202 003 717 422 485 351 562 5
140 737 488 355 328	47	0.000 000 000 000 007 105 427 357 601 001 858 711 242 675 781 25
281 474 976 710 656	48	0.000 000 000 000 003 552 713 678 800 500 929 355 621 337 890 625
562 949 953 421 312	49	0.000 000 000 000 001 776 356 839 400 250 464 677 810 668 945 312 5
1 125 899 906 842 624	50	0.000 000 000 000 000 888 178 419 700 125 232 338 905 334 472 656 25
2 251 799 813 685 248	51	0.000 000 000 000 000 444 089 209 850 062 616 169 452 667 236 328 125
4 503 599 627 370 496	52	0.000 000 000 000 000 222 044 604 925 031 308 084 726 338 618 164 062 5
9 007 199 254 740 992	53	0.000 000 000 000 000 111 022 302 462 515 654 042 363 166 809 082 031 25
18 014 398 509 481 984	54	0.000 000 000 000 000 055 511 151 231 257 827 021 181 583 404 541 015 625
36 028 797 018 963 968	55	0.000 000 000 000 000 027 755 575 615 628 913 510 590 791 702 270 507 812 5
72 057 594 037 927 936	56	0.000 000 000 000 000 013 877 787 807 814 456 755 295 395 851 135 253 906 25
144 115 188 075 855 872	57	0.000 000 000 000 000 006 938 893 903 907 228 377 647 697 925 567 626 953 125
288 230 376 151 711 744	58	0.000 000 000 000 000 003 469 446 951 953 614 188 823 848 962 783 813 476 562 5
576 460 752 303 423 488	59	0.000 000 000 000 000 001 734 723 475 976 807 094 411 924 481 391 906 738 281 25
1 152 921 504 606 846 976	60	0.000 000 000 000 000 000 867 361 737 988 403 547 205 962 240 695 953 369 140 625
2 305 843 009 213 693 952	61	0.000 000 000 000 000 000 433 680 868 994 201 773 602 981 120 347 976 684 570 312 5
4 611 686 018 427 387 904	62	0.000 000 000 000 000 000 216 840 434 497 100 886 801 490 560 173 988 342 285 156 25
9 223 372 036 854 775 808	63	0.000 000 000 000 000 000 108 420 217 248 550 443 400 745 280 086 994 171 142 578 125
18 446 744 073 709 551 616	64	0.000 000 000 000 000 000 054 210 108 624 275 221 700 372 640 043 497 085 571 289 062 5
36 893 488 147 419 103 232	65	0.000 000 000 000 000 000 027 105 054 312 137 610 850 186 320 021 748 542 785 644 531 25
73 786 976 294 838 206 464	66	0.000 000 000 000 000 000 013 552 527 156 068 805 425 093 160 010 874 271 392 822 265 625
147 573 952 589 676 412 928	67	0.000 000 000 000 000 000 006 776 263 578 034 402 712 546 580 005 437 135 696 411 132 812 5
295 147 905 179 352 825 856	68	0.000 000 000 000 000 000 003 388 131 789 017 201 356 273 290 002 718 567 848 205 566 406 25
590 295 810 358 705 651 712	69	0.000 000 000 000 000 000 001 694 065 894 508 600 678 136 645 001 359 283 924 102 783 203 125
1 180 591 620 717 411 303 424	70	0.000 000 000 000 000 000 000 847 032 947 254 300 339 068 322 500 679 641 962 051 391 601 562 5
2 361 183 241 434 822 606 848	71	0.000 000 000 000 000 000 000 423 516 473 627 150 169 534 161 250 339 820 981 025 695 800 781 25
4 722 366 482 869 645 213 696	72	0.000 000 000 000 000 000 000 211 758 236 813 575 084 767 080 625 169 910 490 512 847 900 390 625

APPENDIX D

ALGORITHMS

All arithmetic operations on full and half words are performed in the 36-bit parallel adder. There are two sets of summand inputs to the adder, each set of 36 supplying one input to each adder stage. One set supplies the contents of AR, its complement, or zero; the other set supplies the contents of BR, its complement, or zero. Each stage also has a carry input, which is generated by the next less significant stage. Every stage has two outputs; the carry already mentioned, and a sum. The 36 sum outputs together form the sum of the two input words. The least significant stage has a carry input from the logic for performing twos complement arithmetic and incrementing by one. The negative of a number is formed at the sum outputs simply by supplying the complement of the number at one set of inputs and asserting the carry into stage 35. Adder stage 17 has extra input gating so that 1 can be added to or subtracted from both halves of AR simultaneously.

The adder produces a sum in the same way that one adds binary numbers using pencil and paper. Each adder stage has three inputs, two summand bits and a carry, and two outputs, sum and carry. The sum output of a given stage is 1 if any one or all three of the inputs are 1. The carry out is 1 if two or three of the inputs are 1. Calculations are performed as though the words represented 36-bit unsigned numbers, *ie* the signs are treated just like magnitude bits. In the absence of a carry into the sign stage, adding two numbers with the same sign produces a plus sign in the result. The presence of a carry gives a positive answer when the summands have different signs. The result has a minus sign when there is a carry into the sign bit and the summands have the same sign, or the summands have different signs and there is no carry.

Thus the program can interpret the numbers processed in fixed point arithmetic as signed numbers with 35 magnitude bits or as unsigned 36-bit numbers. A computation on signed numbers produces a result which is correct as an unsigned 36-bit number even if overflow occurs, but the hardware interprets the result as a signed number to detect overflow. Adding two positive numbers whose sum is greater than or equal to 2^{35} gives a negative result, indicating overflow; but that result, which has a 1 in the sign bit, is the correct answer interpreted as a 36-bit unsigned number in positive form. Similarly adding two negatives gives a result which is always correct as an unsigned number in negative form.

All operations discussed below have two operands, one of which is supplied to the adder from BR, which acts simply as a buffer and has no special input gating. MQ has shift gating so it can function as a low order extension of AR for handling double length operands. All actual computations take place in the single 36-bit adder, but the sum output can be placed in either AR or MQ, and all transfers to MQ from AR or BR are made through the adder. In multiplication MQ holds the multiplier and thus

controls the summation of partial products; as the multiplier is shifted out, the low order word of the product is shifted in. In division MQ supplies the low order part of the dividend to AR as the quotient is being constructed in MQ.

In any extended arithmetic operation, the requisite number of steps is counted in the 9-bit shift counter SC, which has a carry network for this purpose. SC also has a 9-bit adder for use in computations on floating point exponents and size and position calculations in byte manipulation.

FIXED POINT ALGORITHMS

Fixed point numbers are explained in detail in §1.1. For convenience let us take the computer representation of the positive number x as $+[x]$ where the brackets enclose the number in bits 1-35. Similarly the representation of $-x$ is $-[2^{35} - x]$ or $-[1 - x]$ depending on whether we are regarding numbers as integers or as proper fractions. The most negative number, -2^{35} , has the form $-[0]$, which is equivalent to the unsigned integer 2^{35} .

Addition. There are four cases of addition of two positive 35-bit numbers x and y .

- I. $x + y$
- II. $(-x) + (-y)$
- III. $x + (-y), \quad x \geq y$
- IV. $x + (-y), \quad x < y$

The operands are held in AR and BR, but it makes no difference which one is in which register. The result appears in AR. For convenience in the exposition we shall regard the numbers as proper fractions; to view them as integers, simply substitute "2³⁵" for each occurrence of "1". Since the two's complement format allows a representation for -1, either x or y may be 1 in II, and y may be 1 in IV.

I. If $x + y < 1$ the adder output placed in AR is $+[x + y]$. If $x + y \geq 1$ the carry out of stage 1 changes the sign. Consequently if the addition of two positive numbers gives a negative result, it is apparent that the sum exceeds the capacity of the register. The processor detects the overflow by checking the sign carries: there is a carry into the sign stage but none out of it. AR then contains

$$-[x + y - 1]$$

II. Ignoring the carry into the sign bit in the addition of two negatives would give

$$\begin{array}{r} -[1 - x] \\ -[1 - y] \\ \hline +[1 + 1 - x - y] \end{array}$$

If $x + y \leq 1$ the carry changes the sign and the result is

$$- [1 - x - y]$$

which is the representation of $-(x+y)$. If $x+y > 1$ there is no carry into the sign, and its absence in the presence of a carry out indicates overflow. AR contains

$$+ [1 - (x + y - 1)]$$

III. Ignoring the carry into the sign in an addition where the signs are different would give

$$\begin{array}{r} + [x] \\ - [1 - y] \\ \hline - [1 + x - y] \end{array}$$

Since $x \geq y$, it follows that $1 + x - y \geq 1$. Hence the carry changes the sign and the result is

$$+ [x - y]$$

When the operand signs are different, the magnitude of the result cannot exceed the larger operand magnitude and there can be no overflow. Since in this case the positive number is at least as large in magnitude as the negative, there is always a carry into the sign, and this added to the operand minus sign produces a carry out.

IV. The addition of numbers of differing signs where the negative has the larger magnitude gives

$$\begin{array}{r} + [x] \\ - [1 - y] \\ \hline - [1 + x - y] \end{array}$$

Since $x < y$, then $1 + x - y < 1$. Hence there are no carries associated with the sign and no overflow. The above result is the two's complement representation of $x - y$, i.e. $-(y - x)$.

Subtraction. The minuend from AC is in AR, and the subtrahend, which is either 0, E or the word from location E, is in BR. Subtraction is done directly by adding the two's complement of BR to AR. The logic supplies the complement of BR to the adder and a carry into the adder LSB.

Let x be the absolute value of the number in AR, and y the absolute value of the number in BR. There are four cases.

- I. $x - (-y)$
- II. $(-x) - y$
- III. $x - y, \quad x \geq y; \quad (-x) - (-y), \quad x \leq y$
- IV. $x - y, \quad x < y; \quad (-x) - (-y), \quad x > y$

These correspond respectively to the four cases of addition discussed previously.

Multiplication. The multiplier, 0, E or the contents of location E, is in MQ, and the multiplicand from AC is in BR. AR is clear. The 36-step procedure is as follows.

If MQ35 (the multiplier LSB) is 1, subtract BR algebraically from AR, but put the result in AR shifted one place to the right, with the LSB of the result going into MQ0, and shift MQ right so a bit of the multiplier is dropped from MQ35. Put the sign of the result in AR0 and AR1 (as though the shift followed the subtraction and did not affect the sign but did move it to AR1). If MQ35 is 0, simply shift AR and MQ right one, with AR35 going into MQ0.

In each subsequent step perform only the shift if the bits moved in and out of MQ35 on the previous step were the same. If they were different, add or subtract along with the shift: if the shift moved a 0 in and a 1 out, add BR to AR; if a 1 in and a 0 out, subtract BR from AR.

Thus the low order bits of the running sum of partial products are shifted into MQ as the multiplier is shifted out. At each step the effect of the multiplicand in BR on the partial sum in AR is one binary order of magnitude greater than in the preceding step because the partial sum was shifted right. Therefore BR can be combined directly with AR. If MQ35 is initially 0, there is no subtraction until a 1 is shifted into it. Simple shifting then continues until the next transition (from 1 to 0), following which BR is added.

The process continues in this way, subtracting at every 0-1 transition, adding at every 1-0 transition. After 35 steps MQ0-34 contains the low half of the product magnitude, and MQ35 contains the sign of the multiplier. At the final step, add or subtract as required but put the result directly into AR; shift only MQ to move the low magnitude into the correct position and make MQ0 equal to the sign of the whole product.

If the original operands were both negative and the result is also negative, set Overflow; this can occur only when -2^{35} is squared. In IMUL, if the high word is not null (*ie* if AR is neither clear nor all 1s), set Overflow; move MQ to AR for storage of the low word.

To see that this procedure results in a correct product, consider the positive binary integer

$$\begin{array}{cccccccc} 1 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{array}$$

where the decimal digits below the binary digits are the powers of 2 corresponding to the bit positions. This number is obviously equal to

$$\begin{array}{r} 100000000 \\ + \quad 111000 \\ + \quad \quad 11 \end{array}$$

Now an n -bit string of 1s whose rightmost bit corresponds to 2^k is equal to $2^{k+n} - 2^k$, or equivalently $2^k(2^n - 2^0)$, *ie* $2^n - 2^0$ is a string of n 1s and the 2^k shifts the string left k places. Hence

$$\begin{array}{rcl} 100000000 & = & 2^{8+1} - 2^8 = 2^9 - 2^8 \\ 111000 & = & 2^{3+3} - 2^3 = 2^6 - 2^3 \\ 11 & = & 2^{2+0} - 2^0 = 2^2 - 2^0 \\ \hline 100111011 & = & 2^9 - 2^8 + 2^6 - 2^3 + 2^2 - 2^0 \end{array}$$

In this last representation, each power of 2 that is subtracted corresponds to

a transition from 0 to 1 (scanning right to left), whereas each that is added corresponds to a 1-0 transition. The largest term corresponds to the transition to the sign bit, which is 0 for a positive number. The multiplication algorithm interprets the multiplier in this manner, alternately subtracting and adding the multiplicand to the partial sum in the order-of-magnitude positions corresponding to the transitions. If a multiplier of the same magnitude were negative, it would have the form

$$\begin{array}{r} 1011000101 \\ -876543210 \end{array}$$

in which the extra bit at the left represents the sign. The number is now equivalent to

$$-2^9 + 2^8 - 2^6 + 2^3 - 2^2 + 2^1 - 2^0$$

wherein opposite signs correspond to opposite transitions. The algorithm may thus use exactly the same sequence for a negative multiplier: this time the subtraction of greatest magnitude is detected by the transition to the sign bit, which is now 1.

Division. The divisor, $0, E$ or the contents of location E , is in BR. In DIV the high and low halves of the dividend from two accumulators are in AR and MQ respectively. In IDIV the one-word dividend from AC is in AR. The two types of division differ mainly in setting up the dividend; in both cases the algorithm processes a positive dividend to get a positive quotient.

In DIV if the dividend is negative ($AR0 = 1$), make it positive and set the negative dividend flag. To negate the dividend, move the low word to AR and the complement of the high word to MQ. Then move the negative of the low word back to MQ and the complement of the high word back to AR. Now the double length negative of the original dividend is in AR and MQ unless MQ is clear; in this event add 1 to AR to give the two's complement negative of the high word. Once the dividend is in positive form shift MQ left one place to close the hole between the two halves; in other words drop the low sign and get the 70-bit magnitude into AR1-35, MQ0-34.

If the IDIV dividend in AR is negative, negate it and set the negative dividend flag. Move the one word dividend in positive form to MQ and clear AR. Shift MQ left, as the algorithm operates on a double length dividend in both types of division although the high part is null in this case.

After the dividend is set up, compare the divisor with it to determine whether the division can be performed. Subtract the absolute value of the divisor from the high half of the dividend (if the divisor is positive, subtract it; if negative, add it). Since the dividend is positive, the result is also positive if the magnitude of the divisor is less than or equal to the number in AR. For a fixed fraction, the divisor is subtracted from the actual dividend and no overflow is allowed. For a fixed integer, AR is clear and the result is positive only for a zero divisor; the worst possible case is the division of $2^{35} - 1$ by 1, whose integral result can be accommodated. (Placing the one word dividend in MQ effectively multiplies it by 2^{-35} , making it the fractional part of a two word dividend with the binary point in the middle. The quotient is then a proper fraction, which is multiplied by 2^{35} simply by interpreting it as an integer.) Thus if the result of this initial subtraction is

positive, set Overflow and No Divide, and terminate the procedure so the processor goes on to the next instruction. Dividing by zero is of course meaningless. The reason for prohibiting a fractional division where the result would be greater than 1 is that it is impossible to determine the position of the binary point in the quotient. So it is up to the programmer to shift the dividend to the correct position beforehand. If the result of the initial subtraction is negative, the division can be performed and the processor goes into the division loop.

In division on paper, one subtracts out the divisor the number of times it goes into the dividend, then shifts the dividend one place to the left (or the divisor to the right) and again subtracts out. In binary computations the divisor goes into the dividend either once or not at all. Each subtraction of the divisor thus generates a single bit of the quotient. If the subtraction leaves a positive difference, *ie* if the dividend is larger than the divisor, a 1 is entered into the quotient. If the difference is negative, a 0 is entered. To compensate for subtracting too much, the hardware could add the divisor back into the dividend before going to the next subtraction step. But the PDP-10 algorithm instead shifts first and adds the divisor back in at the new position. It then continues to shift and add putting 0s into the quotient until the result again becomes positive. This procedure generates the same quotient without ever going back a step.

The hardware procedure is as follows. As each addition or subtraction is formed in the adder, put the result in AR shifted one place to the left with AR35 receiving a new bit of the dividend from MQ0, and shift MQ left bringing in a bit of the quotient at MQ35. The bit brought in is the complement of the sign from the adder: if the divisor does not go into the dividend, the resulting minus sign (1) produces a 0 quotient bit; if the divisor does go in, the plus sign gives a 1. Each step loads one bit of the quotient into MQ35, and the low half of the dividend is shifted out of MQ as the quotient is shifted in.

The first step is the test subtraction. In each subsequent step, subtract the absolute value of the divisor if the quotient bit generated in the previous step is 1, but add it back in if the quotient bit is 0. Since the divisor may have either sign, subtract it algebraically if its sign differs from the quotient bit, add it if its sign is the same.

The hardware executes 36 steps to generate 35 magnitude bits. The initial test step must give a 0, which serves as the sign since we are producing a positive quotient. In the final step put the result of the addition or subtraction directly in AR without shifting so the remainder is in the correct position, but shift MQ left putting the sign from the first step in MQ0 and bringing in the last quotient bit. (The bit dropped out of MQ0 is superfluous; it was brought into MQ35 when the hole was closed between the dividend halves.)

To complete the division we must make sure the remainder is correct and determine the correct signs of the results. Since the operations were performed on positive operands, the remainder should also be positive. A negative remainder indicates that too much has been subtracted. To correct this add the absolute value of the divisor back in. If the negative dividend flag is set, negate AR so the remainder has the sign of the original dividend.

Now move the corrected remainder to MQ and move the quotient to AR. If the negative dividend flag and the divisor sign are of opposite states, negate AR to produce the correct quotient sign. The correct quotient and remainder are now in AR and MQ ready for storage.

As an example of the way this algorithm operates, consider a division of 3-bit fixed fractions with a dividend of $+100100$ and a divisor of $+101$. By paper computation we obtain the quotient this way.

$$\begin{array}{r} .111 \\ 101 \overline{)100.100} \\ \underline{101} \\ 1000 \\ \underline{101} \\ 110 \\ \underline{101} \\ 1 \end{array}$$

Taking the processor registers to be four bits in length, AR contains 0.100 , MQ has 0.100 , and BR has 0.101 . Before starting we close the hole changing MQ to 1.000 . The sequence has four steps.

$$\begin{array}{r} 0.100 \quad 1.000 \\ -0.101 \\ \hline 1.111 \\ 1 \leftarrow 1.111 \quad 0.000 \\ +0.101 \\ \hline 0.100 \\ 2 \leftarrow 1.000 \quad 0.001 \\ -0.101 \\ \hline 0.011 \\ 3 \leftarrow 0.110 \quad 0.011 \\ -0.101 \\ \hline 0.001 \\ 4 \quad 0.001 \quad \leftarrow 0.111 \end{array}$$

The quotient is in MQ at the right, the remainder in AR at the left.

FLOATING POINT ALGORITHMS

§1.1 explains floating point numbers and §2.6 discusses the general characteristics of floating point arithmetic. Exponent computations are done in the SC adder using the exponents and signs from the floating point operands. Remember, the sign is that of the whole number, not of the exponent. Although bits 1-8 of a floating point number represent an exponent in the range -128 to $+127$, the discussion is entirely in terms of the excess 128 exponents in positive form, *ie* the set of numbers $0-255$. Computations generally use twos complement operations even though the exponent in a

ALGORITHMS

negative number is a ones complement. The SC sign bit is used to detect exponent overflow and underflow.

After exponent calculations are complete, operations on the fractions are done by the fixed point logic in AR, BR and MQ. Bits 1-8 of AR and BR are filled with null bits, 0s in a positive number, 1s in a negative. Double length operands are in AR and MQ with MQ8-35 forming a magnitude extension of AR. In almost all circumstances the logic treats AR0-35 and MQ8-35 as a single 64-bit register; in all two-word shifting AR35 is connected to MQ8 and MQ0-7 is ignored. Except in division the fixed point calculation generates a double length fraction, which is shifted arithmetically (in right shifting the sign goes into AR1; in left shifting the sign is unaffected and 0s enter MQ35). Almost all floating point instructions normalize the result, thus making use of the low order part even though the instruction may store only the high order word.

Addition, Subtraction. $E, 0$ or the word from location E is in BR, and AC is in AR. For subtraction move the negative of the subtrahend from BR to AR and move the minuend from AR to BR. This reduces subtraction to addition, so the rest of the algorithm is the same for both.

The initial objective is to determine the difference between the exponents and to determine which exponent is the larger. If the signs of the operands differ, add the exponents into SC. If the signs are the same, subtract the BR exponent from the AR exponent by adding the twos complement. Let x and y be the AR and BR exponents in positive form. The table below shows the calculations as a function of the operand signs, and the sign of the result in SC as a function both of the operand signs and the relative values of x and y .

AR+, BR+		AR+, BR-		AR-, BR+		AR-, BR-	
$+ [x]$		$+ [x]$		$- [255 - x]$		$- [255 - x]$	
$- [256 - y]$		$- [255 - y]$		$+ [y]$		$+ [1 + y]$	
<hr/>		<hr/>		<hr/>		<hr/>	
$- [256 + x - y]$		$- [255 + x - y]$		$- [255 - x + y]$		$- [256 - x + y]$	
SC+ SC-		SC+ SC-		SC+ SC-		SC+ SC-	
$x \geq y$	$x < y$	$x > y$	$x \leq y$	$x < y$	$x \geq y$	$x \leq y$	$x > y$

As can be seen from the above, if AR already contains the number with the smaller exponent, the SC and AR signs differ. Hence if the SC and AR signs are the same, switch BR and AR so the number with the smaller exponent can be shifted. If the exponents are equal, the signs may or may not be the same but it matters not whether the transfer takes place.

To control the shifting we must now get the negative of the difference between the exponents. Let d be $|x - y|$. There are four cases as a function of the SC sign and whether the AR and BR signs are equal. The second column lists the present contents of SC, the third tells what must be done to arrive at $-[256 - d]$ in SC.

SC+, AR0 = BR0	$+ [d]$		Negate SC
SC+, AR0 \neq BR0	$+ [d - 1]$		Complement SC

SC-, AR0 = BR0	-[256 - d]	Do nothing
SC-, AR0 ≠ BR0	-[255 - d]	Add 1 to SC

If $d < 64$ (indicated by a negative SC with a 0 in either SC1 or SC2) nullify AR1-8 and shift AR and MQ right d places so its bits correctly match the BR bits in order of magnitude. If $d > 64$ clear AR for its contents are of no significance.

Now move the larger exponent from BR to SC in positive form, nullify BR1-8, and add BR and AR into AR as fixed fractions. Finally enter the normalizing sequence.

This sequence first tests for a zero result. If AR and MQ8-35 are clear, bypass the rest of the procedure. If the fractional result has overflowed into AR8 (indicated by $AR0 \neq AR8$ or $AR8 = 1$ and $AR9-35 = 0$), shift right and increase the exponent by one. The number is now normalized.

Complement the exponent in SC. If the instruction is not UFA and the number is not normalized go into the normalizing loop. In each step shift the double length fraction left and add 1 to the negative exponent (decreasing its magnitude by 1). Terminate the loop when the fraction is normalized, indicated by the sign and the MSB of the fraction being different ($AR0 \neq AR9$) or the magnitude being $\frac{1}{2}$ ($AR9 = 1$ and $AR10-35 = 0$).

If the instruction specifies rounding, adjust the high fraction so it is rounded and is in twos complement form if negative. The rounding is away from zero. For a positive result the high fraction must be increased if the low fraction is greater than half the value of the high fraction LSB. In a negative result the high fraction is a ones complement, which is one greater in magnitude than the twos complement. Hence it is already rounded and should be decreased in magnitude if the low fraction is $< \frac{1}{2}$ LSB. In either case add 2^{-27} into AR if MQ8 is 1 unless MQ9-35 is clear in a negative number. A 1 in MQ8 indicates a low fraction $\geq \frac{1}{2}$ LSB in a positive number, $\leq \frac{1}{2}$ LSB in a negative number. The condition that MQ9-35 not be zero in a negative number is the case where the low fraction is exactly $\frac{1}{2}$ LSB. If the high fraction is actually changed, renormalize it. A single normalizing shift is all that is required and it occurs in only two cases: a right shift when $1 - 2^{-27}$ is rounded, a left shift when $-\frac{1}{2}$ is changed to a correct twos complement.

Once the number has been normalized (and rounded if necessary) the exponent is in negative form. Thus if the SC sign bit is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow (the maximum possible exponent overflow is 128 giving an SC contents of 777_8 , and this can occur only in division). Insert the exponent in correct form into AR1-8.

The result is now ready to store from AR unless the instruction is in long mode. To ready the double length result subtract 27 from the positive exponent in SC. Save the high word in MQ, and move the low word to AR, but only if the decreased exponent is still positive. If the sign is 1, the true exponent of the low word is less than -128, so clear AR. (Note that this condition is also true if the low exponent is > 127 , which can occur only if the high exponent is > 154 .) If the low word is nonzero, shift AR right one place to put the fraction in bits 9-35 (remember that all shift operations

use MQ8-35), clear AR0 so the low word has a positive sign even if the double length fraction is negative, and insert the low exponent in positive form in bits 1-8. Finally switch AR and MQ so the high and low words are in correct position for storage.

Scaling. The 9-bit signed scale factor from bits 18 and 28-35 of E is in SC, and AC is in AR and BR. If the floating point number being scaled is positive, simply add the sign and exponent from BR0-8 to SC; if the number is negative, add the complement of BR0-8 to SC. Let x be the exponent in positive form and let y be the absolute value of the scale factor. There are only two cases,

$$\begin{array}{r|l} +[x] & +[x] \\ +[y] & -[256-y] \\ \hline +[x+y] & +[x-y] \end{array}$$

and in either the result is in positive form in SC.

Now enter the normalizing sequence described under floating addition. Only left shifting can occur bringing 0s in from MQ. The result can be zero, and exponent overflow or underflow can occur; but there is no rounding, and at the end the one-word result is in AR ready for storage.

Multiplication. $E,0$ or the word from location E is in BR, and AC is in AR. Place the AR exponent in positive form in SC, and add the positive form of the BR exponent to it. Since both are in excess-128 code, subtract 128. Save the result in the floating exponent register FE so SC can be used to control the multiplication of the fractions.

Nullify the exponent parts of AR and BR. Move the multiplier from BR to MQ and the multiplicand from AR to BR. Clear AR. Now multiply the fractions by the same procedure given for fixed point multiplication with the following differences:

- ◆ There are only 28 steps instead of 36.
- ◆ The shift register extension of AR for the construction of the product is MQ8-35. As the multiplier is shifted out, bits of the product come in at MQ8.
- ◆ In the final step place the adder output directly into AR but do not shift MQ - the low fraction is in MQ8-34, the correct position for normalization.

Clear MQ35, move the exponent back to SC, and enter the normalizing sequence described under floating addition. If the operands are normalized, at most one left shift is needed to normalize the result.

Division. The divisor, $E,0$ or the contents of location E , is in BR. The dividend from AC is in AR. In long mode the low half of the dividend from the second accumulator is in MQ; otherwise MQ is clear.

If the dividend is negative, make it positive and set the negative dividend flag. Except in long mode, negate the dividend simply by negating AR. For long mode follow the procedure given for DIV in the second paragraph of the fixed division algorithm. With a floating point operand the left MQ shift puts the low fraction in MQ8-34.

Place the AR exponent in positive form in SC. Subtract the magnitude of the BR exponent from it by adding the negative form of the exponent (ones complement) plus 1. Since the excess-128 factors cancel in the subtraction, add 128. Save the result in the floating exponent register FE so SC can be

used to control the division of the fractions.

Nullify the exponent parts of AR and BR. Subtract the absolute value of the divisor from the high half of the dividend. If the result is positive, indicating the divisor is less than or equal to the dividend, shift AR and MQ right and increase the exponent in SC by 1. Save the adjusted exponent in FE. The shift divides by 2, so if the operands are normalized, the dividend must now be less than the divisor.

Now divide the fractions by the same procedure given for fixed point fractional division with the following differences:

- ◆ Since the dividend has already been adjusted, the test in the first step stops the division only if the divisor is zero, or is unnormalized and less than the dividend. A normalized divisor cannot cause the quotient to overflow. If the result of the initial subtraction is positive, terminate the procedure and set Floating Overflow as well as Overflow and No Divide.
- ◆ Instead of 36 steps there are only 29 if the instruction specifies rounding, otherwise 28.
- ◆ The shift register extension of AR is MQ8–35. As quotient bits are brought in at MQ35, dividend bits are supplied to AR35 from MQ8. The shifting clears MQ0–7.
- ◆ The MQ shift in the final step places a 27-bit quotient fraction in MQ9–35 or a 28-bit fraction in MQ8–35.
- ◆ As in the fixed point algorithm generate the correct signed remainder, put it in MQ, and move the quotient to AR but leave it positive.

If the instruction specifies rounding, shift AR right placing the 27-bit fraction in the correct position, and if the bit shifted out of AR35 is 1, add it back into AR35 to round the positive quotient. If the quotient is zero bypass the rest of the procedure. The remainder will also be zero except in an FDVL where the double length dividend is unnormalized and its high fraction is zero.

Complement the exponent in SC. If the instruction uses normalized operands the initial dividend adjustment guarantees that the quotient will be normalized. If it is not, shift AR left (bringing 0s into AR35) until a 1 appears in AR9, each time increasing the negative exponent by 1 (decreasing its magnitude).

Since the exponent is in negative form, if SC0 is 0, set Overflow and Floating Overflow. If SC1 is also 0, the sign bit must have been changed by decreasing the exponent, so also set Floating Underflow. Insert the exponent in correct form into AR1–8. If the negative dividend flag and the divisor sign (BR0) are of opposite states, negate AR to produce the correct quotient sign.

The quotient is now ready for storage from AR and the remaining operations are performed only for long mode. Save the quotient in BR and bring the high half of the original dividend from AC to AR. Put the dividend exponent in SC. Decrease its magnitude by 26 if the dividend was shifted right at the beginning to allow the division to be performed; otherwise decrease it by 27. Move the remainder to AR and insert the exponent in it provided the remainder is not zero and the exponent is within the proper range, –128 to 127 (the test is that the sign resulting from the exponent calculation is the same as the sign of the remainder). If the exponent is

outside that range clear AR; the assumption is that the remainder is of no significance (*ie* the exponent is too small). Move the remainder with its correct exponent from AR to MQ and put the quotient back in AR. The two words are now ready for storage.

Double Precision Division. The software routine that performs double precision floating point division and the algorithm it utilizes are given at the end of §2.11. FDVL performs the division

$$A/b = q + r2^{-27}/b$$

where q and r are the quotient and remainder. In a double precision division the divisor is of the form

$$B = b + d2^{-27}$$

Using the expansion

$$\frac{1}{x+y} = \frac{1}{x} \left[1 - \frac{y}{x} + \frac{y^2}{x^2} - \frac{y^3}{x^3} + \dots \right] \quad (y^2 < x^2)$$

and letting $x = b$ and $y = d2^{-27}$ gives

$$\frac{A}{B} = \left(q + \frac{r2^{-27}}{b} \right) \left[1 - \frac{d2^{-27}}{b} + \frac{d^2 2^{-54}}{b^2} - \frac{d^3 2^{-81}}{b^3} + \dots \right]$$

Multiplying out and gathering like terms gives

$$\frac{A}{B} = q + \frac{1}{b} (r - qd)2^{-27} - \frac{d}{b^2} (r - qd)2^{-54} + \frac{d^2}{b^3} (r - qd)2^{-81} - \dots$$

where the first two terms on the right are those in the equation at the bottom of page 2-67.

The ratio of adjacent terms is

$$\frac{T_{n+1}}{T_n} = \frac{-d2^{-27}}{b}$$

In an alternating convergent series, the error due to truncation is smaller than the first term dropped. Therefore

$$|\text{Error}| < \frac{d2^{-27}}{b} T_n$$

Since the maximum value of d is less than 1 and the minimum value of b (normalized) is $\frac{1}{2}$,

$$|\text{Error}| < T_n 2^{-26}$$

Book 2

**Assembling
the
Source Program**

MACRO-10 Assembler

CONTENTS

	Page
CHAPTER 1 INTRODUCTION	
1.1	MACRO-10 Language - Statements 1-1
1.1.1	Labels 1-2
1.1.2	Operators 1-2
1.1.3	Operands 1-2
1.1.4	Comments 1-3
1.2	Symbols 1-3
1.2.1	Symbolic Addresses 1-3
1.2.2	Symbolic Operators 1-4
1.2.3	Symbolic Operands 1-4
1.2.4	The Symbol Table 1-4
1.2.4.1	Direct Assignment Statements 1-5
1.2.5	Deleted Symbols 1-5
1.3	Numbers 1-6
1.3.1	Binary Shifting 1-7
1.3.2	Left Arrow Shifting 1-8
1.3.3	Floating-Point Decimal Numbers 1-8
1.3.4	Fixed-Point Decimal Numbers 1-8
1.3.5	Arithmetic and Logical Operations 1-9
1.3.6	Evaluating Expressions 1-10
1.3.7	Numeric Terms 1-10
1.4	Address Assignments 1-11
1.4.1	Setting and Referencing the Location Counter 1-11
1.4.2	Indirect Addressing 1-12
1.4.3	Indexing 1-12
1.4.4	Literals 1-12
1.4.4.1	Multiline Literals 1-13
1.5	Instruction Formats 1-13
1.5.1	Primary Instruction Format 1-14
1.5.2	Input/Output Instruction Format 1-15
1.6	Communication With Monitors 1-16
1.7	Operating Procedures 1-16

CONTENTS (Cont)

	Page
CHAPTER 2	
MACRO-10 ASSEMBLER STATEMENTS - PSEUDO-OPS	
2.1	Address Mode: Relocatable or Absolute 2-1
2.1.1	Relocation Before Execution - PHASE and DEPHASE Statements 2-3
2.2	Entering Data 2-3
2.2.1	RADIX Statements 2-3
2.2.2	Entering Data Under the Prevailing Radix 2-4
2.2.3	DEC and OCT Statements 2-5
2.2.4	Changing the Local Radix for a Single Numeric Term 2-5
2.2.5	RADIX50 Statement 2-6
2.2.6	EXP Statement 2-6
2.2.7	Z Statement 2-6
2.3	Input Data Word Formatting 2-7
2.3.1	BYTE Statement 2-7
2.3.2	POINT Statement - Handling Bytes 2-8
2.3.3	IOWD Statement: Formatting I/O Transfer Words 2-9
2.3.4	XWD Statement: Entering Two Half-Words of Data 2-9
2.3.5	Text Input 2-10
2.3.5.1	ASCII, ASCIZ, and SIXBIT Statements 2-10
2.3.6	Reserving Storage 2-11
2.3.6.1	Reserving a Single Location 2-11
2.3.6.2	BLOCK Statements 2-11
2.4	Conditional Assembly 2-12
2.5	Assembler Processing Statements 2-13
2.5.1	END Statements 2-13
2.5.2	PASS2 Statements 2-14
2.5.3	LIT Statements 2-14
2.5.4	VAR Statements 2-14
2.5.5	PURGE Statements 2-14
2.5.6	Listing Control Statements 2-15
2.5.7	Assembler Control Statements 2-17
2.5.7.1	REPEAT Statements 2-17
2.5.7.2	OPDEF Statements 2-18

CONTENTS (Cont)

	Page	
2.5.7.3	SYN Statements	2-19
2.5.7.4	Permanent Symbols	2-19
2.5.7.5	Extended Instruction Statements	2-19
2.5.8	Linking Subroutines	2-20
2.5.8.1	EXTERN Statements	2-20
2.5.8.2	INTERN Statements	2-21
2.5.8.3	ENTRY Statements	2-21
2.5.9	HISEG Statements	2-22
CHAPTER 3 MACROS		
3.1	Definition of Macros	3-1
3.2	Macro Calls	3-2
3.3	Macro Format	3-2
3.4	Created Symbols	3-3
3.5	Concatenation	3-5
3.6	Indefinite Repeat	3-5
3.7	Nesting and Redefinition	3-7
3.7.1	ASCII Interpretation	3-8
CHAPTER 4 ERROR DETECTION		
4.1	Teletype Error Messages	4-4
CHAPTER 5 RELOCATION		
CHAPTER 6 ASSEMBLY OUTPUT		
6.1	Assembly Listing	6-1
6.2	Binary Program Output	6-1
6.2.1	Relocatable Binary Programs - LINK Format	6-2
6.2.1.1	LINK Formats for the Block Types	6-3
6.2.2	Absolute Binary Programs	6-4
6.2.2.1	RIM10B Format	6-4
6.2.2.2	RIM10 Format	6-5
6.2.2.3	RIM Format	6-6
6.2.2.4	END Statements	6-6

CONTENTS (Cont)

Page

CHAPTER 7
PROGRAMMING EXAMPLESAPPENDIX A
OP CODES, PSEUDO-OPS, AND MONITOR I/O COMMANDSAPPENDIX B
SUMMARY OF PSEUDO-OPSAPPENDIX C
SUMMARY OF CHARACTER INTERPRETATIONSAPPENDIX D
ASSEMBLER EVALUATION OF STATEMENTS AND EXPRESSIONSAPPENDIX E
TEXT CODESAPPENDIX F
RADIX 50 REPRESENTATIONAPPENDIX G
SUMMARY OF RULES FOR DEFINING AND CALLING MACROSAPPENDIX H
OPERATING INSTRUCTIONS

ILLUSTRATIONS

6-1	General RIM10B Format	6-7
6-2	RIM10B Loader	6-8
7-1	Sample Program, CLOG	7-2
7-2	Example of Nested Macro	7-3
7-3	Two Byte Unpacking Subroutines	7-3
7-4	IRPC Example	7-4

TABLES

4-1	Error Codes	4-1
-----	-------------	-----

CHAPTER 1 INTRODUCTION

MACRO-10 is the symbolic assembly program for the PDP-10, and operates in a minimum of 5K of core memory in all PDP-10 systems. MACRO-10 is a two-pass assembler. It is completely device independent, allowing the user to select standard peripheral devices for input and output files. For example, a Teletype can be used for input of the symbolic source program, DECTape for output of the assembled binary object program, and a line printer can be used to output the program listing.

This assembler performs many useful and unique functions, making machine language programming easier, faster, and more efficient. Basically, the assembler processes the PDP-10 programmer's source program statements by translating mnemonic operation codes to the binary codes needed in machine instructions, relating symbols to numeric values, assigning relocatable or absolute core addresses for program instructions and data, and preparing an output listing of the program which includes notification of any errors detected during the assembly process.

MACRO-10 also contains powerful macro capabilities which allow the programmer to create new language elements, thus expanding and adapting the assembler to perform specialized functions for each unique programming job.

1.1 MACRO-10 LANGUAGE - STATEMENTS

MACRO-10 programs are usually prepared on a Teletype, with the aid of a text editing program, as a sequence of statements. Each statement is normally written on a single line and terminated by a carriage return-line feed sequence. MACRO-10 statements are virtually format free; that is, elements of a statement are not placed in numbered columns with rigidly controlled spacing between elements, as in punched-card oriented assemblers.

There are four types of elements in a MACRO-10 statement which are separated by specific characters. These elements are identified by the order of appearance in the statement, and by the separating, or delimiting, character which follows or precedes the element.

Statements are written in the general form:

```
label: operator    operand, operand; comments (carriage return)
```

The assembler interprets and processes these statements, generating one or more binary instructions or data words, or performing an assembly process. A statement must contain at least one of these elements and may contain all four types. Some statements are written with only one operand; but others may have many. To continue a statement on the following line, the control (CTRL) left arrow (\leftarrow), echoed as \leftarrow , is used before the carriage return-line feed sequence (\rightarrow or \downarrow). Examples of program statements are given in Chapter 7, Figures 7-1 and 7-3.

1.1.1 Labels

A label is the symbolic name, created by the source programmer to identify the statement. If present, the label is written first in a statement, and is terminated by a colon (:).

1.1.2 Operators

An operator may be one of the 366 mnemonic machine instruction codes (see PDP-10 System Reference Manual), a command to Monitor, or a pseudo-operation code which directs assembly processing. These assembly pseudo-op codes are described in this manual, and listed with all other assembler defined operators in Appendix A.

Programmers may also create pseudo-ops to extend the power of the assembly language.

An operator may be a macro name, which calls a user-defined macro instruction. Like pseudo-ops, macros direct assembly processing; but, because of their unique power to handle repetitions and to extend and adapt the assembly language, macros are considered separately (see Chapter 3). Operators are terminated with a space or tab.

1.1.3 Operands

Operands are usually the symbolic addresses of the data to be accessed when an instruction is executed, or the input data or arguments of a pseudo-op or macro instruction. In each case, the interpretation of operands in a statement depends on the statement operator. Operands are separated by commas, and terminated by a semicolon (;) or by a carriage return-line feed.

1.1.4 Comments

The programmer may add notes to a statement following a semicolon. Such comments do not normally affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The use of the following special characters should be avoided in comments: < > [].

1.2 SYMBOLS

The programmer may create symbols to use as statement labels, as operators, and as operands. A symbol contains from one to six characters from the following set:

The 26 letters, A - Z
 Ten digits, 0 - 9
 Three special characters: \$ (Dollar Sign)
 % (Percent)
 . (Period)

The first character in a symbol must not be a digit. If the first character is a period, it must not be followed by a digit. Spaces must not be embedded in symbols. A symbol may actually have more than six characters, but only the first six are meaningful to MACRO-10.

MACRO-10 accepts programs written using both upper and lower case letters and symbols. (e.g., programs written using the Teletype Model 37). Lower case letters are treated as upper case in symbols; additional special characters, and lower case letters elsewhere, are taken without change.

1.2.1 Symbolic Addresses

A symbol used as a label to specify a symbolic address must appear first in the statement and must be immediately followed by a colon (:). When used in this way, a symbol is said to be defined. A defined symbol can reference an instruction or data word at any point in the program. A symbol can be defined as a label only once. If a programmer attempts to define a symbol as a label again, the second or successive attempt is ignored and an error is indicated. The assembler recognizes only the first definition. These are legal symbolic addresses:

```
ADDR
.TOTAL
SSUM:
ABC: DEF:           (Both Labels are legal)
```

The following are illegal:

7ABC :	(First character must not be a digit.)
LAB :	(Colon must immediately follow label.)

1.2.2 Symbolic Operators

Symbols used as operators must be predefined by the assembler or by the programmer. If a statement has no label, the operator may appear first in the statement, and must be terminated by a space, tab, or carriage return. The following are examples of legal operators:

MOVE	(A mnemonic machine instruction operator.)
LOC	(An assembler pseudo-op.)
ZIP	(Legal only if defined by the user.)

1.2.3 Symbolic Operands

Symbols used as operands must have a value defined by the user. These may be symbolic references to previously defined labels where the arguments to be used by this instruction are to be found, or the values of symbolic operands may be constants or character strings. If the first operand references an accumulator, it must be followed by a comma.

```
TOTAL:  ADD AC1, TAG
```

The first operand, AC1, specifies an accumulator register, determined by the value given to the symbol AC1 by the user. The second operand references a memory location, whose name, or symbolic address is TAG. If the user has equated AC1 to 17, and the assembler has assigned TAG to the binary address, 000537, then the assembler inserts 17 in the accumulator field (bits 9 - 12) and 000537 in the address field (bits 18 - 35) of this instruction. If an accumulator is not specified, but the operator requires one, accumulator 0 is assumed by default. If an accumulator is specified by the value >17₈, the four least significant bits are used.

1.2.4 The Symbol Table

The assembler processor symbols in source program statements by referencing its symbol table, which contains all defined symbols, along with the binary value assigned to each symbol.

Initially, the symbol table contains the mnemonic op codes of the machine instructions, the Monitor I/O command mnemonics, and the assembler pseudo-op codes, as listed in Appendix A. As the source program is processed, symbols defined in the source program, as well as new symbols defined by MACRO-10 for use by this program, are added to the symbol table.

1.2.4.1 Direct Assignment Statements - The programmer inserts new symbols with their assigned values directly into the symbol table by using a direct assignment statement of the form,

```
symbol=value ;
```

where the value may be a number or expression. For example,

```
ALPHA= 5 ;
BETA= 17 ;
```

A direct assignment statement may also be used to give a new symbol the same value as a previously defined symbol:

```
BETA= 17
GAMMA= BETA
```

The new symbol, GAMMA, is entered into the symbol table with the value 17.

The value assigned to a symbol may be changed:

```
ALPHA= 7 ;
```

changes the value assigned in the first example from 5 to 7.

Direct assignment statements do not generate instructions or data in the object program. These statements are used to assign values so that symbols can be conveniently used in other statements.

1.2.5 Deleted Symbols

Sometimes a programmer may want to define a symbol in MACRO but not want to have that symbol typed out by DDT. In such a case, the programmer should define that symbol with a double equal sign:

```
FLAG== 200
```

FLAG will be assigned the value 200 and will be

- a. Fully available in MACRO.
- b. Available for type-in with DDT (assuming that symbols were loaded for the program containing FLAG).
- c. Unavailable for type-out by DDT.

This is equivalent to defining FLAG by:

```
FLAG= 200
```

and then typing

```
FLAG$K
```

to DDT

If a symbol is defined with == and declared internal, then the == will be ignored.

1.3 NUMBERS

Numbers used in source program statements may be signed or unsigned, and are interpreted by the assembler according to the radix specified by the programmer, where

$$2 \leq \text{radix} \leq 10$$

The programmer may use an assembler pseudo-op, RADIX, to set the radix for the numbers which follow. If the programmer does not use a RADIX statement, the assembler assumes a radix of 8 (octal).

The radix may be changed for a single numeric term, by using the qualifier † followed by a letter, D (for decimal), O (for octal), B (for binary), or F (for fixed-point decimal fractions). Thus,

†D10	is stored as	1010
†O10	is stored as	1000
†B10	is stored as	0010

The qualifier †L is used for bit position determination of a numeric value. †Ln generates an octal value equal to the number of 0 bits to the left of the leftmost 1, if the numeric value n were stored in a computer word.

Expression	Resultant Value	
$\uparrow L0$	44	$\overbrace{0000000000\dots0000000000}^{44_g \text{ zero bits}}$
$\uparrow L5$	41	$\overbrace{0000000000\dots0000000101}^{41_g \text{ zero bits}}$
$\uparrow L-1$	0	1111111111\dots1111111111

1.3.1 Binary Shifting

A number may be logically shifted left or right by following it with the letter B, followed by a number, n, representing the bit position in which the right-hand bit of the number should be placed. B may be any bit position 0 -35 decimal; if B is not used, B35 is assumed; n is taken as modulo 256 decimal. Thus, the number $\uparrow D10$ is stored as 000000 000012; but $\uparrow D10B32$ is shifted left three binary positions and stored as 000000 000120; and $\uparrow D10B4$ is shifted left 31 positions, so that its rightmost bit is in bit 4 and stored as 240000 000000.

Binary shifting is a logical operation, rather than an arithmetic one.

The following are legal binary shifts:

$1B0$	400000 000000
$1B17$	000001 000000
$1B35$	000000 000001
$-1B35$	777777 777777 (see explanation below)
$-1B53$	000000 777777
$-1B70$	000000 000001

Note that the following expressions are equivalent:

$$10B32 \equiv \uparrow O10B32 \equiv 10B42 - 10 \equiv 10B \langle \uparrow D \langle 42-10 \rangle \rangle \equiv 10B \langle \uparrow D42 - \uparrow D10 \rangle$$

The unary operators preceding a value are interpreted first by the assembler before the binary shift. A leading plus sign has no effect, but a leading minus sign causes the assembler to shift and then to store the 2's complement.

Binary shifting may operate on numeric terms, as defined in Section 1.3.6.

1.3.2 Left Arrow Shifting

If two expressions are combined with the operator " \leftarrow ", i.e. $\langle m \rangle \leftarrow \langle n \rangle$, the 36 bit value of expression m is shifted V bits (where V is the value of expression n) in the direction of the arrow (left) if V is positive or against the arrow if V is negative. The effective magnitude of V is that of the address of an LSH instruction.

1.3.3 Floating-Point Decimal Numbers

If a string of digits contains a decimal point, it is evaluated as a floating-point decimal number, and the digits are taken radix 10. For example, the statement,

`17.0 D` is stored as 205420 000000.

Floating-point decimal numbers may also be written, as in FORTRAN, with the number followed by the letter E, followed by a signed exponent representing a power of 10. The following examples are valid:

```
NUM1 : 17.2F-4 D
NUM2 : 3.85E2 D
NUM3 : -567.825E33 D
```

1.3.4 Fixed-Point Decimal Numbers

As shown in Section 1.3, `D` followed by a numeric term, is used to enter decimal integers.

Fixed-point decimal numbers (mixed numbers) are preceded by `F` followed by a number (not a numeric term, defined below) which normally contains a decimal point. The assembler forms these fixed-point numbers in two 36-bit registers, the integer part in the first and the fractional part in the second. The value is then stored in one storage word in the object program; the integer part to the left of the assumed binary point, the fractional part to the right.

The binary shift (`B`) operator is used to position the assumed point. The number `F123.45B8` is formed in two registers:

```
000000 000173      (the integer part)
346314 631462      (the fraction part, left-justified)
```

The `B` operator sets the assumed point after bit 8, so the integer part is placed in bits 0-8, and the fraction part in bits 9-35 of the storage word. In this case, the integer part is truncated from the left to fit the 9-bit integer field. The fraction part is moved into the 27-bit field following the assumed point and is truncated on the right. The result is,

```
173346 314631
      ↑
      (assumed point)
```

If a `B` shift operator does not appear in a fixed-point number, the point is assumed to follow bit 35, and the fractional part is lost.

Fixed-point numbers are assumed to be positive unless a minus sign precedes the qualifier:

```

000000 000173      †F123.45
000173 346314      †F123.45817
346314 631462      †F123.458-1

777777 777604      -†F123.45
777604 431463      -†F123.45817
431463 146316      -†F123.458-1

```

Negative fixed-point numbers, such as the example above, are assembled as if they were positive numbers, complemented, and then logically shifted.

1.3.5 Arithmetic and Logical Operations

Numbers and defined symbols may be combined using arithmetic and logical operators. The following arithmetical and logical operators may be used.

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
&	AND
	Inclusive OR

The assembler computes the 36-bit value of a series of numbers and defined symbols connected by arithmetic and logical operators, truncating from the left, if necessary. The following examples show how these arithmetic and logical operators are written in statements.

```

R=      65+X11-3 )
MULTI   A01+7,RHO/31 )
MOVE    A+3,BETA-5 )

```

Combinations of numbers and defined symbols using arithmetical and logical operators are called expressions.

1.3.6 Evaluating Expressions

When combining elements of an expression, the assembler first performs unary operations (leading + or then binary shifts. The logical operations are then done from left to right, followed by multiplications and divisions, from left to right. Division always truncates the fractional part. Finally, additions and subtractions are performed, left to right. All arithmetic operations are performed modulo 2^{35} .

For example, in the statement:

```
TAG: TR0 3,1+A&C)
```

The first operand field is evaluated first; the comma terminating this operand indicates that this is an accumulator. In the second operand field, the logical AND is performed first, the result is added to one, and the sum is placed in the memory address field of the machine instruction.

To change the normal order of operations, angle brackets may be used to delimit expressions and indicate the order of computation. Angle brackets must always be used in pairs.

Expressions may be nested to any level, with each expression enclosed in a pair of angle brackets. The innermost expression is evaluated first, the outermost is evaluated last. The following are legal expressions:

```
A+1/5  
<<C-D+B-29>*A-41-X>>+1
```

1.3.7 Numeric Terms

A numeric term may be a digit, a string of digits, or an expression enclosed in angle brackets. The assembler reduces numeric terms to a single 36-bit value. This is particularly useful when specifying operations such as local radix changes and binary shifts, which require single values.

For example, the tD operator changes the local radix to decimal for the numeric term that follows it. The number, 23_{10} , may be represented by

```
tD:3  
or tD<5*2+13>  
or tD<TEN*2+THREE>
```

but 23_{10} may not be written,

```
↑D100-77
```

because the ↑D operator affects only the numeric term which follows it, and in this example the second term (77) is taken under the prevailing radix, which is normally octal.

The B shift operator is preceded by a numeric term (the number to be shifted) and is followed by another term (the bit position of the assumed point). The following are legal:

```
↑F167B17
↑B10011B8
566B5
<MARK + SIGN>B<PT-XXV>
```

A bracketed numeric term may be preceded by a + or a - sign.

1.4 ADDRESS ASSIGNMENTS

As source statements are processed, the assembler assigns consecutive memory addresses to the instruction and data words of the object program. This is done by incrementing the location counter each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement may generate six storage words, incrementing the location counter by six.

The mnemonic instruction and Monitor command* statements generate a single storage word. However, direct assignment statements and some assembler pseudo-ops do not generate storage words, and do not affect the location counter. Other pseudo-ops and macros may generate many words in the object program.

1.4.1 Setting and Referencing the Location Counter

The MACRO-10 programmer may set the location counter by using the pseudo-ops, LOC and RELOC, which are described in Chapter 2. He may reference the location counter directly by using the symbol, point (.). For example, he can transfer to the second previously assigned storage word by writing:

```
JRST .-2 )
```

*The terms Monitor command (as used here) and programmed operator are synonymous.

1.4.2 Indirect Addressing

The character @ prefixing an operand causes the assembler to set bit 13 in the instruction word, indicating an indirect address. For an explanation of indirect addressing and effective address calculation, see the PDP-10 System Reference Manual, DEC-10-HGAA-D (page 1-7).

1.4.3 Indexing

If indexing is used to increment the address field, the address of the index register used is entered in parentheses, as the last part of the memory reference operand. This is normally a symbolic name defined by a direct assignment statement, or an octal number in the range 1-17, specifying 1 of the 15 index registers. However, the address of the index register may be any legal expression or expression element.

This is a symbolic, indirect, indexed, memory reference:

```
A: ADD 4,(INDEX(17))
```

NOTE

The parentheses cause the value of the enclosed expression to be interpreted as a 36-bit word with its two halves interchanged, e.g., (17) is effectively 000017000000₈.

1.4.4 Literals

In a MACRO-10 statement, a symbolic data reference may be replaced by a direct representation of the data enclosed in square brackets. This direct representation is called a literal. The assembler stores the bracketed data in its literal table, assigns an address to the first word of the data and inserts that address in the machine instruction.

A literal may be any term, symbol, expression or statement, but it must generate data. Statements which do not generate data, i.e., some pseudo-ops, such as RADIX, and direct assignment statements, may not be written as literals. Literals may be nested, up to 18 levels.

Here is a simple example. Byte instructions must reference a byte pointer word, like this:

```
LDB 4, BP )
BP: POINT 10, A+3, 14 )
```

(POINT is a pseudo-op which sets up a byte pointer word.) A literal can be used to insert the POINT statement directly. (The use of literals is also shown in Chapter 7, Figure 7-3.)

```
LDB 4, [POINT 10, A+3, 14] )
```

1.4.4.1 Multiline Literals - MACRO optionally allows multiline literals. The following is legal:

```

GETCHR: SOSG  IBUF+2                ;ANY CHARS LEFT?
        PUSHJ P,[IN      N,         ;NO, READ SOME IN
        POPJ  P,                   ;NO UNUSUAL CONDITIONS
        STATZ N,740000             ;CHECK FOR ERRORS
        JRST [MOVEI E, [SIXBIT /INPUT ERROR/]
                JRST ERRPNT]      ;PUBLISH ERROR MESSAGE
        JRST ENDFIL]              ;END OF FILE HANDLER
        ILDB  AC,IBUF+1            ;PICKUP NEXT CHAR
        POPJ  P,                   ;TRA 1,4

```

Two new pseudo-operations have been added to control whether or not this feature is available. Use of these pseudo-ops is required since

```
MOVE AC,[SIXBIT/TEXT/
```

is legal in MACRO-10, even though the closing right bracket (]) of the literal has been omitted. In normal mode, MACRO allows such an unterminated literal. However, the pseudo-op

```
MLON
```

causes the assembler to consider all code following a left bracket as part of a literal, until such time as the assembler processes a matching right bracket. Thus, carriage-return, line-feed no longer ends a literal, but rather the programmer must insert a right bracket. The pseudo-op

```
MLOFF
```

places MACRO back into the (initial) compatibility mode in which literals may occupy only a single line.

The symbol . (current location) is not changed by the use of literals:
It retains the value it had before the literal was entered.

1.5 INSTRUCTION FORMATS

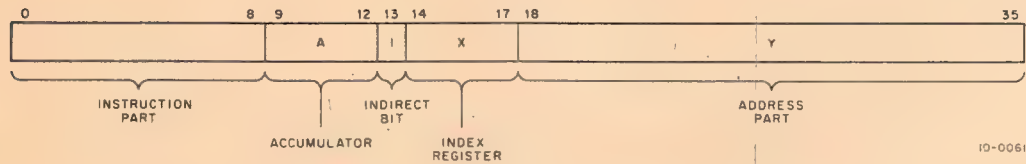
There are two types of machine instruction word formats: primary and input/output.

The 366 PDP-10 machine instructions are fully described in the PDP-10 System Reference Manual and listed alphabetically in Appendix A of this manual. Monitor I/O commands, or programmed operators, have the same formats. (See Monitor manuals.)

The primary instruction statements may have two operands: (1) an accumulator address and (2) a memory address. A memory address may be modified by indexing and indirect addressing.

1.5.1 Primary Instruction Format

After processing primary instruction statements, the assembler produces machine instructions in the general 36-bit word format shown below:

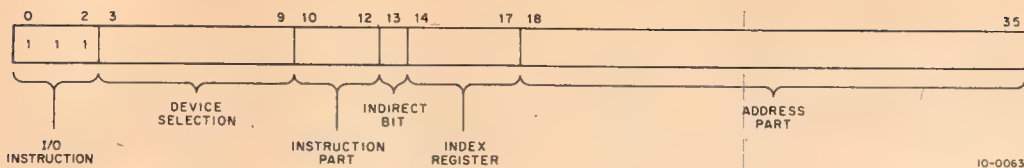


In general, the mnemonic operation code, or operator, in the symbolic statement is translated to its binary equivalent and placed in bits 0-8 of the machine instruction. The address operand is evaluated and placed in the address part (bits 18-35) of the machine instruction. The assembler assigns sequential binary addresses to each statement as it is processed by means of the location counter. Labels are given the current value of the location counter and are stored in the assembler's symbol table, where the corresponding binary addresses can be found if another instruction uses the same symbol as an address reference.

The 16 accumulators are specified by writing them (symbolically or numerically) as operands in the statement, followed by a comma. The indirect address bit is set to 1 when the character @ prefixes a memory reference. Indexing is specified by writing the index register used in parentheses immediately following the memory reference. (All PDP-10 accumulators, except accumulator 0, may be used as index registers.) Actually, expressions enclosed in parentheses (in the index register position) are evaluated as 36-bit quantities; their halves are exchanged, and then each half is added into the corresponding half of the binary word being assembled. For example, the statements

```
MOVSI AC,(1.0)           ;MOVE 1.0 TO, AC)
MOVSI AC,(SIXBIT /DSK/)
```


The format for I/O instruction words is shown below:



1.6 COMMUNICATION WITH MONITORS

Programs assembled with MACRO-10 which operate under executive control of a Monitor must use Monitor facilities for device independent I/O services. This is done by means of programmed operators (operation codes 040 through 077) such as CALL, INIT, LOOKUP, IN, OUT, and CLOSE.

Additional Monitor commands are available to allow the user program to exercise control over central processor trapping, to modify its memory allocation, and other services, which are described in the Monitor programmer's manuals.

Monitor commands are listed in Appendix A.

1.7 OPERATING PROCEDURES

Commands for loading and executing MACRO-10 are contained in the PDP-10 System User's Guide (DEC-10-NGCC-D).

CHAPTER 2
MACRO-10 ASSEMBLER
STATEMENTS - PSEUDO-OPS

Assembler statements or pseudo-ops direct the assembler to perform certain assembler processing operations, such as converting data to binary under a selected radix, or listing selected parts of the assembled object program. In this chapter, these assembler processing operations are fully described.

NOTE

The pseudo-op name must follow the rules for constructing a symbol (refer to paragraph 1.2) and must be terminated by a character other than those listed in paragraph 1.2 as valid symbolic characters. (Normally, a space or tab is used as a terminator.)

2.1 ADDRESS MODE: RELOCATABLE OR ABSOLUTE

MACRO-10 normally assembles programs with relocatable binary addresses, so that the program can be loaded anywhere in memory for execution as a function of what has been previously loaded. When desired, the assembler will also assign absolute location addresses, either for the entire program or for selected parts. Two pseudo-ops control the address mode: RELOC and LOC.

RELOC N)

This statement sets the location counter to *n*, which may be a number or an expression, and causes the assembler to assign relocatable addresses to the instructions and data which follow. Since most relocatable programs start with the location counter set to 0; the implicit statement,

RELOC 0)

begins all programs, and need not be written by the programmer who wants his program assembled with relocatable addresses.

LOC N)

This statement sets the location counter to *n*, a number or an expression, and causes the assembler to assign absolute addresses, beginning with *n*, to the instructions and data which follow. If the entire program is to be assigned absolute locations, a LOC statement must precede all instructions and data.

If *n* is not specified

```
(LOC )
```

zero is assumed initially.

If only a part of the program is to be assembled in absolute locations, the LOC statement is inserted at the point where the assembler begins assigning absolute locations. For example, the statement,

```
LOC 200)
```

causes the assembler to begin assigning absolute addresses, and the next machine instruction or data word is stored at location 200₈.

To change the address mode back to relocatable, an explicit RELOC statement is required. If the programmer wants the assembler to continue assigning relocatable addresses sequentially, he writes,

```
RELOC )
```

To switch back to the next sequential absolute assignment, he writes,

```
LOC )
```

Thus, the programmer is not required to insert a location counter value in either a LOC or RELOC statement, and unless he does, both the relocatable coding and the absolute coding will be assigned sequential addresses. This is shown in the following skeleton coding. The single quote mark is used here, and in MACRO-10 listings, to identify relocatable addresses.

<u>Location Counter</u>	<u>Program</u>	
000000'	ADD 1,X	;RELOC 0 IS IMPLICIT.
	:	
000074'	LOC 1000	;CHANGES TO ABSOLUTE, STARTING
001000	SUB 5,TOT	;WITH 001000.
	:	
001034	RELOC	;SETS LOCATION COUNTER TO 74.
000074'	ADD 2,XAT	
000075'	LOC	;SWITCHES LOCATION COUNTER
001034	EXP A-X+7	;BACK TO ABSOLUTE SEQUENCE.

When operating in the relocatable mode, the assembler determines whether each location in the object program is relocatable or absolute, using an algorithm described in Chapter 5.

2.1.1 Relocation Before Execution - PHASE and DEPHASE Statements

Part of a program can be moved into other locations for execution. This feature is often used to relocate a frequently used subroutine, or iterative loop, into fast memory (accumulators 0-17) just prior to execution.

To use this feature, the subroutine is assembled at sequential relocatable or absolute addresses along with the rest of the program, but the first statement before the subroutine contains the assembler operator, PHASE, followed by the address of the first location of the block into which the subroutine is to be moved prior to execution. All address assignments in the subroutine are in relation to the argument of the PHASE statement. The subroutine is terminated by a DEPHASE statement, which requires no operands, and which restores the location counter.

In the following example, which is the central loop in a matrix inversion, a block transfer instruction moves the subroutine LOOP into accumulators 11-16.

Relocatable Address	LOOPX:	MOVE [XWD LOOPX, LOOP]
		BLT LOOP+4
		JRST LOOP
	LOOP:	PHASE 11
Absolute Address		MOVN A (X)
		FMP MPYR
		FADM A (Y)
		SOJGE X, #-3
		JRST MAIN
		DEPHASE

The label LOOP represents accumulator 11, and the point in the SOJGE instruction represents accumulator 14.

2.2 ENTERING DATA

2.2.1 RADIX Statements

When the assembler encounters a numerical value in a statement, it converts the number to a binary representation reflecting the radix indicated by the programmer. The statement,

```
RADIX N)
```

where n is a decimal number, $2 \leq n \leq 10$, sets the radix to n for all numerical values that follow, unless another RADIX statement changes the prevailing radix or a local radix change occurs (see below).

For example, if the programmer wants the assembler to interpret his numbers as decimal quantities, then the prevailing radix must be set to decimal before he uses decimal numbers.

```
RADIX 10 )
```

The statement, RADIX 2, sets the prevailing radix to binary.

The implicit statement, RADIX 8, begins every program; if the programmer wants to enter octal numbers, this statement is not necessary.

2.2.2 Entering Data Under the Prevailing Radix

Data is entered under the prevailing radix by typing the data, followed by a carriage return:

```
765432234567 )
```

Data may be labeled and contain expressions:

```
LAB: 456+A+B / <C+D> )
```

Data may also be entered by first using a direct assignment statement to place a symbol with an assigned value in the symbol table, and then using the symbol to insert the assigned value in the object program. For example, the direct assignment statements,

```
A=2 )
B=5 )
```

cause two entries in the symbol table. The following statement enters the sum of the assigned values in the object program at symbolic address REX.

```
REX: A+B )           REX contains 000000 000007
```

The radix can also be changed locally, that is, for a single statement or a single value, after which the prevailing radix is automatically restored, as described in Section 1.3.

2.2.3 DEC and OCT Statements

To change to a local radix for a single statement, the programmer writes:

```
DEC N,N,N,...N )
```

where all of the numbers and expressions are to be interpreted as decimal numbers. The numbers or expressions following the operator are separated by commas, and each will generate a word of storage.

```
OCT N,N,N,...N )
```

Changes the local radix to octal for this statement only, and generates a word of memory for each number or expression.

The statement,

```
DEC 10,4.5,3.1416,6.03E-26,3 )
```

generates five decimal words of data.

2.2.4 Changing the Local Radix for a Single Numeric Term

To change the radix for a single number or expression, the numeric term is prefixed with †D †O, †B, or †F, as explained in Chapter 1. If an expression is used, it must be enclosed in angle brackets,

```
†D <A+B-C/200>
```

These prefixes may generate a word, or part of an instruction word. The statement,

```
TOTAL2:MOVE †D10,ABZ )
```

causes the contents of ABZ to be moved to accumulator 12_8 .

When the assembler encounters a numeric term, it forms the binary representation in a 36-bit register under the prevailing or local radix. If the quantity is a part of an instruction, it is truncated to fit in the required field.

For example, the accumulator field must have a final value in the range $0-17_8$. In the statement,

```
MOVE †D60,ABZ )
```

the assembler first interprets the accumulator address in a 36-bit register, forming the integer 000000000074: but takes only the rightmost four bits and places them in the accumulator field of the instruction, which results in the selection of accumulator 14₈.

2.2.5 RADIX50 Statement

Another radix changing statement is available, but it is used primarily in systems programming. This is RADIX50 n, sym \rangle which is used by the assembler, PDP-10 Loader, DDT, and other systems programs to pack symbolic expressions into 32 bits and add a 4-bit code field n in bits 0-3. This is explained in Appendix F of this manual. (The mnemonic SQUOZE may be used in place of RADIX50.)

2.2.6 EXP Statement

Several numbers and expressions may be entered by using the EXP statement:

```
EXP X,4, (D65,HALF,B+362-A)
```

which generates one word for each expression; five words were generated for the above example.

2.2.7 Z Statement

A zero word can be entered by using the operator, Z.

```
LABEL: Z)
```

generates a full word of all zeros at LABEL. If operands follow the Z, the assembler forms a primary machine instruction, with the operator field and other unknown fields zeroed. In the statement,

```
Z 3)
```

the assembler finds an accumulator field, but no address field, and generates this machine instruction: 000140 000000.

2.3 INPUT DATA WORD FORMATTING

2.3.1 BYTE Statement

To conserve memory, it is useful to store data in less than full 36-bit words. Bytes of any length, from 1 to 36 bits, may be entered by using a BYTE statement.

```
BYTE (N) X,X,X)
```

The first operand (n) is the byte size in bits. It is a decimal number in the range 1-36, and must be enclosed in parentheses. The operands following are separated by commas, and are the data to be stored. If an operand is an expression, it is evaluated and, if necessary, truncated from the left to the specified byte size. Bytes are packed into words, starting at bit 0, and the words are assigned sequential storage locations. If, during the packing of a word, a byte is too large to fit into the remaining bits, the unused bits are zeroed and the byte is stored left-justified in the next sequential location.

In the following statement, three 12-bit bytes are entered:

```
LABEL: BYTE (12)5,177,N)
```

This assembles at LABEL as, 0005 0177 0316, where N=316.

The byte size may be altered by inserting a new byte size in parentheses immediately following any operand. Notice that the parentheses serve as delimiters, so commas must not be written when a new byte size is inserted. The following are legal:

```
BYTE (6)5(14)NT(3)6,2,5)
```

where 5 is entered in a 6-bit byte, NT in the following 14-bit byte, 6 in the following 3-bit byte, followed by 2 and 5 in 3-bit bytes. A BYTE statement can be used to reserve null fields of any byte size. If two consecutive delimiters are found, a null field is generated.

```
BYTE (1)(),5)
```

When the assembler finds two delimiters, it assembles a null byte. In this case, 000000 000005.

To enter ASCII characters in a byte, the character is enclosed in quotation marks.

```
BYTE (7)"A")
```

Text handling pseudo-ops are discussed in Section 2.3.4. An example of the use of the BYTE statement is given in Chapter 7, Figure 7-3.

2.3.2 POINT Statement - Handling Bytes

Five machine instructions are available for byte manipulation. These instructions reference a byte pointer word, which is generated by the assembler from a POINT statement of the form,

LABEL: POINT s, address, b) (s and b are decimal)

where the first operand s is a decimal number indicating the byte size, the second operand is the address of the memory location which contains the byte, and the third operand, b, is the bit position in the word of the right-hand bit of the byte (if b is not specified, the bit position is the nonexistent bit to the left of the bit 0). The address specified in the second operand may be indirect and indexed. If the byte size is not specified, MACRO-10 assumes 36 bits.

In the following example, an LDB (load a byte from a memory location into an accumulator) and an ILDB instruction are used, along with three assembler statements. The ILDB instruction "increments" AC to look like AB, then does a load byte; the effect of the two instructions is the same.

```

000000  050000  000000  AA:    BYTE   (6)5
000001  360600  000000  AB:    POINT  6,AA,5
000002  440600  000000  AC:    POINT  6,AA

000003  135140  000001  START: LDB   3,AB
000004  134140  000002  ILDB  3,AC

```

The first statement enters the quantity 5 in a 6-bit byte at symbolic address AA which is 0. The second statement is for reference by the load byte instruction. When the LDB is executed, the machine goes to AB for the byte size, its address, and bit position. In this case, it finds that the byte size is 6 bits, the byte is located in the word AA, and the right-hand bit of the byte is in bit 5. The byte is then loaded into accumulator 3, where it looks like this: 000000 000005.

The other byte manipulation mnemonic instructions reference the byte pointer word in similar ways. The deposit byte (DPB) instruction takes a byte from an accumulator and deposits it, in the position specified by the pointer word, in a memory word.

The increment byte pointer (IBP) instruction increments the bit position indicator (the third operand in the referenced POINT word) by the byte size. This is useful when loading or depositing a string of bytes, using the same byte pointer word.

The increment and load byte (ILDB) and increment and deposit byte (IDPB) instructions increment the byte pointer word by the byte size before loading or depositing.

An example of the use of the POINT statement is given in Chapter 7, Figure 7-3.

2.3.3 IOWD Statement: Formatting I/O Transfer Words

The assembler generates I/O transfer words in a special format for use in BLKI and BLKO and all four push-down instructions. The general statement is,

```
IOWD N,M )
```

where two operands, which may be numbers or expressions, follow the IOWD operator. This statement generates one data word. The left half of the assembled word contains the 2's complement of the first operand n , and the right half-word contains the value of the second operand m , minus one. For example,

```
IOWD 6,1D256 )
```

assembles as 777772 000377.

2.3.4 XWD Statement: Entering Two Half-Words of Data

The XWD statement enters two half-words in a single storage word. It is written in the form,

```
XWD LHW,RHW )
```

where the first operand is a symbol or expression specifying the left half-word, and the second operand specifies the right half-word. Both are formed in 36-bit registers and the low order 18-bits are placed in the half-words. Three examples follow:

```
XWD A,B )
XWD SUM+2,DES+5 )
XWD START,END )
```

XWD statements are used to set up pointer words for block transfer instructions. Block transfer pointer words contain two 18-bit addresses: the left half is the starting location of the block to be moved, and the right half is the first location of the destination.

2.3.5 Text Input

The assembler translates text written in full 7-bit ASCII or 6-bit compressed ASCII. It will also format 7-bit ASCII with a null character at the end of text, if desired. These codes are listed in Appendix E.

In all three text modes, the printing symbols in the code set are translated to their binary representation. In 7-bit ASCII, five control characters are also accepted:

Horizontal Tab
 Line Feed
 Vertical Tab
 Form Feed
 Carriage Return

To translate and store a single word containing as many as five 7-bit ASCII characters, right-justified, the input characters are simply enclosed in quotation marks.

```
"AXE" )      is stored as
              0 0000000 0000000 1000001 1011000 1000101
              0  null   null   A    X    E
```

Notice that characters are right-justified, and bit 0, which is not used, is set to zero.

2.3.5.1 ASCII, ASCIZ, and SIXBIT Statements - To enter one or more words of text characters, the operators ASCII, SIXBIT, and ASCIZ are used. The delimiter for the string of text characters is the first nonblank character following the character that terminates the operator (refer to the note on page 2-1). The binary codes are left-justified. Unused character positions are set to zero (null). Text is terminated by repeating the initial delimiter. The statement,

```
ASCII "AXE" )
```

assembles as,

```
1000001 1011000 1000101 0000000 0000000 0
      A    X    E    null  null  0
```

The operator ASCIZ (ASCII Zero) guarantees a null character at the end of text. If the number of characters is a multiple of five, another all zero word is added. For example,

```
ASCIZ/"AXE"/ )
```

assembles as ,

```
0100010 1000001 1011000 1000101 0100010 0
      "   A       X       E       "
```

followed by another word of zeros .

```
0000000 0000000 0000000 0000000 0000000 0
null
```

When the full 7-bit ASCII code set is not required, the 64-character 6-bit subset may be entered, using the SIXBIT operator. Six characters are left-justified in sequential storage words. Format of the SIXBIT statement is the same as for ASCII statements. To derive SIXBIT code:

- a. Convert lower case ASCII characters to upper case characters.
- b. Add 40_8 to the value of the character.
- c. Truncate the result to the rightmost six bits.

2.3.6 Reserving Storage

The programmer can reserve single locations, or blocks of many locations for use during execution of his program.

2.3.6.1 Reserving a Single Location - The number sign (#), suffixing a symbol in an operand field, is used to reserve a single location. The symbol is defined, entered in the assembler's symbol table, and can be referenced elsewhere in the program without the number sign. For example,

```
LAB: ADD 3,TEMP#)
```

reserves a location called TEMP at the end of the program, which may be used to store a value entered at some other point in the program. This feature is useful for storing scalars, and other quantities which may change during execution.

2.3.6.2 BLOCK Statements - To reserve a block of locations, the BLOCK operator is used. It is followed by a single operand, which may be a number or an expression, indicating the number of words to be reserved. The assembler increments the location counter by the value of the operand. For

example,

```
MATRIX: BLOCK N*M
```

reserves a block of N*M words starting at MATRIX for an array whose dimensions are M and N.

2.4 CONDITIONAL ASSEMBLY

Parts of a program may be assembled, or not assembled, on an optional basis depending on conditions defined by an assembler IF statement. The general form is,

```
IF N, <.....>
```

where the coding within angle brackets is assembled only if the first operand, n, meets the statement requirement.

The IF statement operators and their conditions are listed below:

<u>Operator</u>	<u>Assemble angle-bracketed coding IF:</u>
IFE N, <...>	n=0, or blank
IFG N, <...>	n > 0
IFGE N, <...>	n = 0, or n > 0
IFL N, <...>	n < 0
IFLE N, <...>	n = 0, or n < 0
IFN N, <...>	n = 0
IF1, <...>	encountered during pass 1
IF2, <...>	encountered during pass 2

The following conditional statements operate on character strings. Arguments are interpreted as 7-bit ASCII character strings, and the assembler makes a logical comparison, character-by-character to determine if the condition is met.

The coding within the third set of angle brackets is assembled if the character strings enclosed by the first two sets of angle brackets:

IFIDN <A-Z> <A-Z>, <...>	(1) are identical
IFDIF <A-Z> <A-X>, <...>	(2) are different

These statements, IFIDN and IFDIF, are usually used in macro expansions (see Chapter 3) where one or both arguments are dummy variables.

In the following conditional statements, assembly depends on whether or not a symbol has been defined.

The coding enclosed in angle brackets is assembled if,

IFDEF SYMBOL, <...>	this symbol is defined.
IFDEF SYMBOL, <...>	this symbol is not defined.

The last pair of conditional statements is followed by a single bracketed character string, which is either blank or not blank, and which is followed by conditional coding in brackets.

The coding enclosed in the second set of angle brackets is assembled if,

IFB <...>, <...>	the first operand is blank.
IFNB <...>, <...>	the first operand is not blank.

A blank field is either an empty field or a field containing only the ASCII characters space (40_g) or tab (11_g).

2.5 ASSEMBLER PROCESSING STATEMENTS

These statements direct the assembler to perform various kinds of processing.

2.5.1 END Statements

The END statement must be the last statement in every program. A single operand may follow the END operator to specify the address of the first instruction to be executed. Normally this operand is given only in the main program; since subprograms are called from the main program, they need not specify a starting address.

```
END START)
```

When the assembler first encounters an END statement, it terminates pass 1 and begins pass 2. The END also terminates pass 2, after which the assembler automatically assembles all previously defined literals starting at the current location.*

The following processing operations can be performed at any point in the program.

*The END statement is also used to specify a transfer word in some output file formats. (See Section 6.2.2.4.)

2.5.2 PASS2 Statements

PASS2)

This statement switches the assembler to pass 2 processing for the remaining coding. Coding preceding this statement will have been processed by pass 1 only. This is used primarily for debugging, such as testing macros defined in the pass 1 portion.

The two assembly operators, LIT and VAR, are used to control assembly allocation of storage.

2.5.3 LIT Statements

LIT)

This statement causes literals that have been previously defined to be assembled, starting at the current location. If n literals have been defined, the next free storage location will be at location counter plus n. Literals defined after this statement are not affected.

2.5.4 VAR Statements

VAR)

This statement causes symbols which have been defined by suffixing with the # sign in previous statements to be assembled as block statements. This has no effect on subsequent symbol definitions of the same type.

If the LIT and VAR statements do not appear in the program, all literals and variables are stored at the end of the program.

2.5.5 PURGE Statements

The PURGE statement is used to delete defined symbols. Its general form is:

PURGE symbol, symbol, symbol)

where each operand is a user-created label, operator, or macro call which is to be deleted from the assembler's tables. The PURGE statement is normally used at the end of programs to conserve storage. Purged symbol table space is reused by the assembler.

If the programmer uses the same symbol for both a macro call and/or OPDEF and for a label, a PURGE statement deletes the macro call or OPDEF. A repeat of the symbol in the PURGE statement also purges the label. For example, the following statement purges both:

```
PURGE SOLV, SOLV )
```

The first SOLV purges the macro call; the second SOLV purges the label.

2.5.6 Listing Control Statements

As the source program statements are processed during pass 2, the program listing is normally printed on a line printer or a Teletype, depending on the listing file device specified. A sample listing is shown in Figure 7-1.

From left to right, the standard columns contain the location counter, the instruction or data in octal (divided into two 6-digit columns for easier reading), and the symbolic instruction or data, followed by comments. Relocatable object-code addresses are suffixed by a single quote mark ('), which may occur in either the left or right half.

A line printer listing always begins at the top of a page, and up to 55 lines are printed on each page. Consecutive page numbers are printed in the upper right-hand corner of each page.

Listing is suppressed within macro expansions, so that only the macro call and any succeeding lines that generate object program coding are listed.

These standard listing operations can be augmented and modified by using the following listing control statements.

```
TITLE NAME )
```

The single operand may contain up to 60 characters which will be printed on the top of each page. The first six characters of the title appear in the assembled program as the program name. If no title is given, the assembler inserts ".MAIN". The program name given in the TITLE statement is used when debugging with DDT to gain access to the program's symbol table.

```
SURTTL SUBTITLE )
```

The single operand may contain up to 40 characters. It is printed as the second line at the top of each page. If the subtitle is changed by another SUBTTL statement, the new subtitle appears in the second line of the following page.

- PAGE) This statement causes the assembler to skip to the top of the next page. (A form feed character in the input text has the same effect.)
- XLIST) This statement causes the assembler to stop listing the assembled program. The listing printout actually starts at the beginning of pass 2 operations. Therefore, to suppress all program listing, XLIST must be the first statement in the program. If only a part of the program listing is to be suppressed, XLIST is inserted at any point to stop listing from that point.
- LIST) Normally used following an XLIST statement to resume listing at a particular point in the program. The LIST function is implicitly contained in the END statement.
- LALL) This statement causes the assembler to list everything that is processed including all text, macro expansions, list control codes, and repeats, all of which are suppressed in the standard listing.
- XALL) Normally used following a LALL statement to resume standard listing with all text, macro expansions, list control codes and repeats suppressed.
- NOSYM) The assembler normally prints out the symbol table at the end of the program, but the NOSYM statement suppresses the symbol table printout.
- TAPE) This pseudo-op causes the assembler to begin assembling the program contained in the next source file in the MACRO command string. For example,

```
.R MACRO
*DSK:BINAME,LPT: +TTY:,DSK:MORE
PARAM=6
TAPE
tZ
```

would set the symbol PARAM equal to 6 and then assemble the remainder of the program from the source file DSK:MORE. Since MACRO is a 2-pass assembler, the TTY: file would probably be repeated for pass 2:

```
END OF PASS 1
PARAM=6
TAPE
tZ
```

Note that all text after the TAPE pseudo-op is ignored.

PRINTX MESSAGE) This statement, when encountered, causes the single operand following the PRINTX operator to be typed out on the TTY. This statement is frequently used to print out conditional information. PRINTX statements are also used in very long assemblies to report the progress of the assembler through pass 1.

The operand is treated as a comment and will be output on the error message media. It is not counted as an error, but if error messages are suppressed, PRINTX messages are also suppressed.

REMARK COMMENTS) The REMARK operator is used for statements which contain only comments. Such statements may also be started with a semi-colon.

2.5.7 Assembler Control Statements

2.5.7.1 REPEAT Statements - The statement

```
REPEAT N, <...> )
```

causes the assembler to repeat the coding enclosed in angle brackets n times. If more than one instruction or data word is to be repeated, each is delimited by a carriage return. For example,

```
ADDX: REPEAT 3, <ADD 6,X(4) )
      ADDI 4,1 > )
```

assembles as,

```
ADDX: ADD 6,X(4)
      ADDI 4,1
      ADD 6,X(4)
      ADDI 4,1
      ADD 6,X(4)
      ADDI 4,1
```

Notice that the label of a REPEAT statement is placed on the first line of the assembled coding. REPEAT statements may be nested to any level. The following simplified example shows how a nested REPEAT statement is interpreted.

```
REPEAT 3, <A )
REPEAT 2, <B )
      C > )
      D > )
```

assembles as,

```

[ A
  B ]
[ C
  B ]
[ C
  D ]
[ A
  B ]
[ C
  B ]
[ C
  D ]
[ A
  R ]
[ C
  R ]
[ C
  C ]
  D

```

NOTE

Brackets indicate repetition.

2.5.7.2 OPDEF Statements – The programmer can define his own operators using an OPDEF statement, which is written in the form:

```
OPDEF SYM [STATEMENT]
```

where the first operand is defined as an operator, whose function is defined by the second operand, which is enclosed in square brackets. The second operand is evaluated as a statement, and the result is stored in a 36-bit word. For example,

```
.OPDEF CAL1 [USRUU0]
```

defines CAL1 as an operator, with the value 030000 000000. CAL1 may now be used as a statement operator,

```
030140 001234 CAL1 3,1234
```

which is equivalent to,

```
030140 001234 Z 3,1234(30000)
```

When MACRO-10 encounters a user-defined operator, it assembles a single object-program storage word in the format of a primary instruction word (see Chapter 1). The defined 36-bit value is modified by accumulator, indirect, memory address and index fields as specified by the user-defined operator.

For example,

```
OPDEF CAL [MOVE 1,@SYM(2)]
CAL 1,BOL(2)
```

The CAL statement is equivalent to:

```
MOVE 2,@SYM+BOL(4)
```

In this modification the accumulator fields are added, the indirect bits are logically ORed, the memory address fields are added, and the index register addresses are added.

2.5.7.3 SYN Statements - The statement

```
SYN symbol, symbol
```

defines the second operand as synonymous with the first operand, which must have been previously defined. Either operand may be a symbol or a macro name. If the first operand is a symbol, the second is defined as a symbol with the same value. If the first is a macro name, the second becomes a macro name which operates identically. If the first is a machine, assembler, or user-defined operator, the second will be interpreted in the same manner. If the first operand in a SYN statement has been previously defined as both a label and as an operator, the second operand is synonymous with the label.

The following are legal SYN statements:

```
SYN K,X) ; IF K=5, X=5
SYN FAD,ADD)
SYN END,XEND)
```

2.5.7.4 Permanent Symbols - Redefinition of permanent symbols (e.g., device names like DIS) is permitted. Macro takes the newly defined value, but also flags the line with a "Q" warning message.

2.5.7.5 Extended Instruction Statements - For programming convenience, some extended operation codes are provided in the MACRO-10 Assembler. Primarily, these are used to replace those PDP-10 instructions where the combination of instruction mnemonic and accumulator field is used to denote a single instruction. For example:

```
JRST 4,
```

is equivalent to a halt instruction. Additionally, they are used to replace certain commonly used I/O instruction-device number combinations.

The extended instruction statements are exactly like the primary instruction statements or I/O instruction statements, except that they may not have an accumulator field or device field.

The operator field must have one of the following extended mnemonics:

Extended Instructions	Equivalent Machine Instructions	Meaning
JEN	JRST 12,	Jump and enable the PI (priority interrupt) system
HALT	JRST 4,	Halt
JRSTF	JRST 2,	Jump and reset flags
JOV	JFCL 10,	Jump on overflow and clear
JCRY0	JFCL 4,	Jump on CRY0 and clear
JCRY1	JFCL 2,	Jump on CRY1 and clear
JCRY	JFCL 6,	Jump on CRY0 or CRY1 and clear
JFOV	JFCL 1,	Jump on floating overflow
RSW	DATAI 0	Read the console switches

2.5.8 Linking Subroutines

Programs usually consist of subroutines which contain references to symbols in external programs. Since these subroutines may be assembled separately, the loader must be able to identify "global" symbols. For a given subroutine, a global symbol is either a symbol defined internally and available for reference by other subroutines, or a symbol used internally but defined in another subroutine. Symbols defined within a subroutine, but available to others, are considered internal. Symbols which are externally defined are considered external.

These linkages between internal and external symbols are set up by declaring global symbols using the operators EXTERN, INTERN, or ENTRY.

2.5.8.1 EXTERN Statements - The EXTERN statement identifies symbols which are defined elsewhere. The statement,

```
EXTERN SQRT, CUBE, TYPE )
```

declares three symbols to be external. External symbols must not be defined within the current subroutine. These external references may be used only as an address or in an expression that is to be used as an address. For example, the square root routine declared above might be called by the statement,

```
PUSHJ P, SQRT )
```

External symbols may be used in the same manner as any other relocatable symbol. Examples:

```

                EXTERN  A
200300  000003  MOVE   6, A+3
000003  000000  XWD    A+3, A
777777  777771  B=     A-7
                OPDEF  Q [XWD B+3, A-5]
777774  777773  Q
```

There are three restrictions for the use of external symbols:

- a. Externals may not be used in LOC and RELOC statements.
- b. The use of more than one external in an expression is not permitted. Thus, A-B (where A and B are both external) is illegal.
- c. An internal symbol may not be set equal to an external symbol.

2.5.8.2 INTERN Statements - To make internal program symbols available to other programs as external symbols, the operators INTERN or ENTRY are used. These statements have no effect on the actual assembly of the program, but will make a list of symbol equivalences available to other programs at load time. The statement,

```
INTERN MATRIX )
```

makes the subroutine MATRIX available to other programs. An internal symbol must be defined within the program as a label, variable, or by direct assignment.

2.5.8.3 ENTRY Statements - Some subroutines have common usage, and it is convenient to place them in a library. In order to be called by other programs, these library subroutines must contain the statement,

```
ENTRY NAME )
```

where "name" is the symbolic name of the entry point of the library subroutine.

ENTRY is equivalent to INTERN except for the following additional feature. All names in a list following ENTRY are defined as internal symbols and are placed in a list at the beginning of the library of subroutines. If the loader is in library search mode, a subroutine will be loaded if the program to be executed contains an undefined global symbol which matches a name on the library ENTRY list.

If the MATRIX subroutine mentioned before is a library subroutine, it must contain the statement,

```
ENTRY MATRIX)
```

Since library subroutines are external to programs using them, the calling program must list them in EXTERN statements.

2.5.9 HISEG Statements

```
HISEG )
```

The HISEG pseudo-op statement generates information that directs the Loader to load the current program into the high segment if the system has re-entrant (two-segment) capability. (Refer to "Block Type 3 Load Into High Segment" in paragraph 6.2.1.1 for additional information.) This pseudo-op may appear anywhere in the source program, but it is recommended that it be placed near the beginning to avoid confusion.

CHAPTER 3 MACROS

When writing a program, certain coding sequences are often used several times with only the arguments changed. If so, it is convenient if the entire sequence can be generated by a single statement. To do this, the coding sequence is defined with dummy arguments as a macro instruction. A single statement referring to the macro by name, along with a list of real arguments, generates the correct sequence.

3.1 DEFINITION OF MACROS

The first statement of a macro definition must consist of the operator `DEFINE` followed by the symbolic name of the macro. The name must be constructed by the rules for construction symbols. The macro name may be followed by a string of dummy arguments enclosed in parentheses. The dummy arguments are separated by commas and may be any symbols that are convenient--single letters are sufficient. A comment may follow the dummy argument list.

The character sequence, which constitutes the body of the macro, is delimited by angle brackets. The body of the macro normally consists of a group of complete statements.

For example, this macro computes the length of a vector:

```

DEFINE VMAG (A,B)      ;ROUTINE FOR THE LENGTH OF A VECTOR
<MOVE 0,A              ;GET THE FIRST COMPONENT
FMP 0                  ;SQUARE IT
MOVE 1,A+1             ;GET THE SECOND COMPONENT
FMP 1,1                ;SQUARE IT
FAD 1                  ;ADD THE SQUARE OF THE SECOND
MOVE 1,A+2             ;GET THE THIRD COMPONENT
FMP 1,1                ;SQUARE IT
FAD 1                  ;ADD THE SQUARE OF THE THIRD
JSR FSQRT              ;USE THE FLOATING SQUARE ROOT ROUTINE
MOVEM B                ;STORE THE LENGTH>

```

3.2 MACRO CALLS

A macro may be called by any statement containing the macro name followed by a list of arguments. The arguments are separated by commas and may be enclosed with parentheses. If parentheses are used (indicated by an open parenthesis following the macro name), the argument string is ended by a closed parenthesis. If there are *n* dummy arguments in the macro definition, all arguments beyond the first *n*, if any, are ignored. If parentheses are omitted, the argument string ends when all the dummy arguments of the macro definitions have been assigned, or when a carriage return or semicolon delimits an argument.

The arguments must be written in the order in which they are to be substituted for dummy arguments. That is, the first argument is substituted for each appearance of the first dummy argument; the second argument is substituted for each appearance of the second dummy argument, etc. For example the appearance of the statement:

```
VMAG VECT, LENGTH
```

in a program generates the instruction sequence defined above for the macro VMAG. The character string VECT is substituted for each occurrence in the coding of the dummy argument A, and the character string LENGTH is substituted for the single occurrence of B in the coding.

Statements with a macro call may have label fields. The value of the label is the location of the first instruction generated.

CAUTION

MACRO arguments are terminated only by COMMA, CARRIAGE RETURN, SEMICOLON or CLOSE PARENTHESIS (when the entire argument string was started with an open parenthesis). These characters may not be included in arguments unless < > are used. Specifically, spaces or tabs do not terminate arguments; they will be treated as part of the argument itself.

3.3 MACRO FORMAT

- a. Arguments must be separated by commas. However, arguments may also contain commas. For example:

```
DEFINE JEQ(A,B,C)
<MOVE [A]
CAMN B
JRST C>
```


If the data in location B is equal to A (a literal), the program jumps to C. If A is to be the instruction ADD2,X, the calling macro instruction would be written:

```
JEQ <ADD2,X>B,INSTX
```

The angle brackets surrounding the argument are removed, and the proper coding is generated.

The general rule is: If an argument contains commas, semicolons, or any other argument delimiters, the argument must be enclosed in angle brackets.

b. A macro need not have arguments. The instruction:

```
DATAO PTP,PUNBUF(4)
```

which causes the contents of PUNBUF, indexed by register 4, to be punched on paper tape, may be generated by the macro:

```
DEFINE PUNCH  
<DATAO PTP,PUNBUF(4)>
```

The calling macro instruction could be written:

```
PUNCH
```

PUNCH calls for the DATAO instruction contained in the body of the macro.

c. The macro name, followed by a list of arguments, may appear anywhere in a statement. The string within the angle brackets of the macro definition exactly replaces the macro name and argument string. For example:

```
DEFINE L(A,B)<3*<B-A+1>>
```

gives an expression for the number of items in a table where three words are used to store each item. A is the address of the first item, and B is the address of the last item. To load an index register with the table length, the macro can be called as follows:

```
MOVEI X,L(FIRST,LAST)
```

3.4 CREATED SYMBOLS

When a macro is called, it is often convenient to generate symbols without explicitly stating them in the call, for example, symbols for labels within the macro body. If it is not necessary to refer to these labels from outside the macro, there is no reason to be concerned as to what the labels are. Nevertheless, different symbols must be used for the labels each time the macro is called. Created symbols are used for this purpose.

Each time a macro that requires a created symbol is called, a symbol is generated and inserted into the macro. These generated symbols are of the form `..hijk`, that is, two decimal points followed by four digits. The user is advised not to use symbols starting with two points. The first created symbol is `..0001`, the next is `..0002`, etc.

If a dummy symbol in a definition statement is preceded by a percent sign (`%`), it is considered to be a created symbol. When a macro is called, all missing arguments that are of the form `%X` are replaced by created symbols. However, if there are sufficient arguments in the calling list that some of the arguments are in a position to be assigned to the dummy arguments of the form `%X`, the percent sign is overruled and the stated argument is assigned in the normal manner.

Null arguments are not considered to be the same as missing arguments. For example, suppose a macro has been defined with the dummy string:

```
(A,%B,%C)
```

If the macro were called with the argument string:

```
(OPD,) or OPD,,
```

the second argument would be considered to have been declared as a null string. This would override the `%` prefixed to the second dummy argument and would substitute the null string for each appearance of the second dummy argument in the statement. However, the third argument is missing. A label would be created for each occurrence of `%C`. For example:

```
DEFINE TYPE(A,%B)
<JSR TYPEOUT
JRST %B
SIXBIT/A/
%B:>
```

This macro types the text string substituted for `A` on the console Teletype. `TYPEOUT` is an output routine. Labeling the location following the text is appropriate since `A` may be text of indefinite length. A created symbol is appropriate for this label since the programmer would not normally reference this location. This macro might be called by:

```
TYPE HELLO
```

which would result in typing `HELLO` when the assembled macro is executed. If the call had been:

```
TYPE HELLO,BX
```

the effect would be the same. However, BX would be substituted for %B, overruling the effect of the percent sign.

3.5 CONCATENATION

The apostrophe character or single quote (') is defined as the concatenation operator and may not be used otherwise inside a macro definition. (Outside a macro definition, it is ignored except as a character in textual data.) A macro argument need not be a complete symbol. Rather, it may be a string of characters which form a complete symbol when joined to characters already contained in the macro definition. This joining, called concatenation, is performed by the assembler when the programmer writes an apostrophe between the strings to be so joined. As an example, the macro:

```
DEFINE J(A,B,C)
<JUMP 'A B,C>
```

When called, the argument A is suffixed to JUMP to form a single symbol. If the call were:

```
J (LE,3,X+1)
```

the generated code would be:

```
JUMPLE 3,X+1
```

The concatenation operator (') may be used in nested macros. However, the assembler removes the operator when it performs concatenation in first level macros, but does not remove the operator during concatenation in the second or deeper levels.

3.6 INDEFINITE REPEAT

It is often convenient to be able to repeat a macro one or more times for a single call, each repetition substituting successive arguments in the call statement for specified arguments in the macro. This may be done by use of the indefinite repeat operator, IRP. The operator IRP is followed by a dummy argument, which may be enclosed in parentheses. This argument must also be contained in the DEFINE statement's list. This argument is broken into subarguments. When the macro is called, the range of the IRP is assembled once for each subargument, the successive subarguments being substituted for each appearance of the dummy argument within the range of the IRP. For example, the single argument:

```
<ALPHA,BETA,GAMMA>
```

consists of the subarguments ALPHA, BETA, and GAMMA. The macro definition:

```
DEFINE DOEACH(A),
<IRP A
<A>>
```

and the call:

```
DOEACH<ALPHA,BETA,GAMMA>
```

produce the following coding:

```
ALPHA
BETA
GAMMA
```

An opening angle bracket must follow the argument of the IRP statement to delimit the range of the IRP. A closing angle bracket must terminate the range of the IRP. IRPC is like IRP except it takes only one character at a time; each character is a complete argument. An example of a program that uses an IRPC is given in Chapter 7, Figure 7-4.

It is sometimes desirable to stop processing an indefinite repeat depending on conditions given by the assembler. This is done by the operator STOPI. When the STOPI is encountered, the macro processor finishes expanding the range of the IRP for the present argument and terminates the repeat action. An example:

```
DEFINE CONVERT (A)
<IRP A<IFE K-A,<STOPI
CONV1 A>>>
```

Assume that the value of K is 3; then the call:

```
CONVERT<0,1,2,3,4,5,6,7>
```

generates:

```
<IRP
IFE K-0,<STOPI
CONV1 0>
IFE K-1,<STOPI
CONV1 1>
IFE K-2,<STOPI
CONV1 2>
IFE K-3,<STOPI
CONV1 3>
```

The assembly condition is not met for the first three arguments of the macro. Therefore, the STOPI code is not encountered until the fourth argument, which is the number 3. When the condition is met, the STOPI code is processed which prevents further scanning of the arguments. However, the action continues for the current argument and generates CONVI 3, i.e., a call for the macro CONVI (defined elsewhere) with an argument of 3.

3.7 NESTING AND REDEFINITION

Macros may be nested; that is, macros may be defined within other macros. For ease of discussion, levels may be assigned to these nested macros. The outermost macros, i.e., those defined directly to the macro processor, may be called first level macros. Macros defined within first level macros may be called second level macros; macros defined within second level macros may be called third level macros; etc.

At the beginning of processing, first level macros are known to the macro processor and may be called in the normal manner. However, second and higher level macros are not yet defined. When a first level macro containing second and higher level macros is called, all its second level macros become defined to the processor. Thereafter, the level of definition is irrelevant, and macros may be called in the normal manner. Of course, if these second level macros contain third level macros, the third level macros are not defined until the second level macros containing them have been called.

When a macro of level n contains a macro of level $n+1$, calling the macro results in generating the body of the macro into the user's program in the normal manner until the DEFINE statement is encountered. The level $n+1$ macro is then defined to the macro processor; it does not appear in the user's program. When the definition is complete, the macro processor resumes generating the macro body into the user's program until, or unless, the entire macro has been generated.

If a macro name which has been previously defined appears within another definition statement, the macro is redefined, and the original definition is eliminated.

The first example of a macro calculation of the length of a vector may be rewritten to illustrate both nesting and redefinition.

```
DEFINE VMAG (A,B,%C)
<DEFINE VMAG (D,E)
<JSP SJ, VL
EXP D,E>
VMAG (A,B)
```

```

        JRST %C
VL:    HRRZ 2, (SJ)
        MOVE (2)
        FMP 0
        MOVE 1,1(2)
        FMP 1,1
        FAD 1
        MOVE 1,2(2)
        FMP 1,1
        FAD 1
        JSR FSORT
        MOVEM @1 (SJ)
        JRST 2(SJ)
%C:>

```

The procedure to find the length of a vector has been written as a closed subroutine. It need only appear once in a user's program. From then on it can be called as a subroutine by the JSP instruction.

The first time the macro VMAG is called, the subroutine calling sequence is generated followed immediately by the subroutine itself. Before generating the subroutine, the macro processor encounters a DEFINE statement containing the name VMAG. This new macro is defined and takes the place of the original macro VMAG. Henceforth, when VMAG is called, only the calling sequence is generated. However, the original definition of VMAG is not removed until after the expansion is complete.

Another example of a nested macro is given in Chapter 7, Figure 7-2.

3.7.1 ASCII Interpretation

If the reverse slash (\) is used as the first character in a macro call, the value of the following symbol is converted to a 7-bit ASCII character in the current radix. If the call is

```
MAC \A
```

and if A=500 (in the current radix), this generates the three ASCII characters "500".

CHAPTER 4
ERROR DETECTION

MACRO-10 makes many error checks as it processes source language statements. If an apparent error is detected, the assembler prints a single letter code in the left-hand margin of the program listing, on the same line as the statement in question.

The programmer should examine each error indication to determine whether or not correction is required. At the end of the listing, the assembler prints a total of errors found; this is printed even if no listing is requested.

Each error code indicates a general class of errors. These errors, however, are all caused by illegal usage of the MACRO-10 language, as described in the preceding three chapters of this manual.

TABLE 4-1
ERROR CODES

<u>Error Code</u>	<u>Meaning</u>	<u>Explanation</u>
A	Argument error in pseudo-op	This is a broad class of errors which may be caused by an improper argument in a pseudo-op.
D	Multiply-defined symbolic reference error	This statement contains a tag which refers to a multiply-defined symbol. It is assembled with the first value defined.
E	External symbol error	Improper usage of an external symbol. Example: <pre>EXT: EXTERN TXT,BRT,EXT EXT CANNOT BE BOTH AN EXTERNAL AND INTERNAL SYMBOL.</pre>
L	Literal error	A literal is improper. A literal must generate 1 to 18 words. <pre>EXP [SIXBIT //]; NO CODE GENERATED.</pre>

TABLE 4-1 (Cont)
ERROR CODES

<u>Error Code</u>	<u>Meaning</u>	<u>Explanation</u>
M	Multiply-defined symbol	<p>A symbol is defined more than once. The symbol retains its first definition, and the error message M is typed out during pass 1.</p> <p>If this type of error occurs during pass 2, it is a phase error (see below).</p> <p>If a symbol is first defined as a #-sign suffixed tag, and later as a label, it retains the label definition.</p> <p>Examples:</p> <pre>A: ADD 3,X; A: MOVE ,C; M ERROR A: ADD 3,X#; X: MOVE ,C; X IS ASSIGNED THE CURRENT VALUE OF THE LOCATION COUNTER.</pre> <p>Multiple appearances of the TITLE pseudo-op (which generates both a title line and program name) are flagged as "M" (Multiple definition) errors.</p>
N	Number error	<p>A number is improperly entered.</p> <p>Examples:</p> <pre> ↑F13.33E38 (Exceeds range) ↑D15BZ (Number must fol- low B shift operator.) But ↑D15B<Z> is legal if Z is de- fined.</pre> <p>If a number contains meaningless letters or special characters, a Q error is given.</p>
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of 0.
P	Phase error	A symbol is assigned a value as a label during pass 2 different from that which it received during pass 1. In general, the assembler should generate the same number of program locations in pass 1 and pass 2, and any discrepancy causes a

TABLE 4-1 (Cont)
ERROR CODES

<u>Error Code</u>	<u>Meaning</u>	<u>Explanation</u>
p	Phase error (cont)	phase error. For example, if an assembly conditional, IF1, generates three instructions, a phase error results unless another conditional, such as IF2, generates three program locations during pass 2.
Q	Questionable	This is a broad class of possible errors in which the assembler finds ambiguous language. Example: ADD ,TOTAL SUM; SUM IS NOT NEEDED AND IS TREATED AS A COMMENT.
R	Relocation error	LOC or RELOC are used improperly. Example: LOCA; WHERE A IS NOT DEFINED.
S	Symbol format error	Usually caused by inclusion of illegal special characters. Example: SY?M: ADD 3,X;
U	Undefined symbol	A symbol is undefined.
V	Value previously undefined	A symbol used to control the assembler is undefined prior to the point at which it is first used. Causes error message in pass 1.

Error messages during pass 1 consist of two lines. The most recently used label is printed on the first line, followed by +n, where n is the (decimal) number of lines of coding between the labeled statement and the statement containing an error. The second line of the error message is a copy of the erroneous line of coding, with a letter code in the left-hand margin to indicate the type of error. If more than one type of error occurs on the same line, more than one letter is printed; but if the same type of error occurs more than once in the same line, a single letter code is printed.

During pass 2, as the listing is printed out, lines containing errors are marked by letter codes, and a total of errors found is printed at the end of the listing.

4.1 TELETYPE ERROR MESSAGES

The following error messages may be typed out on the Teletype by MACRO. Those preceded by a question mark are treated as fatal errors when running under Batch Processor (the run is terminated by BATCH.)

END OF PASS 1

Manual loading is required to start pass 2 when the input is paper tape or cards.

LOAD THE NEXT FILE

Manual loading is required if the next file is on paper tape or cards.

?COMMAND ERROR
?NO END STATEMENT
ENCOUNTERED ON INPUT FILE

Error in MACRO command string.

?CANNOT ENTER FILE XXX

PASS 1 cannot be completed because the source program is not terminated by an END statement.

?CANNOT FIND FILE XXX

?INSUFFICIENT CORE

?PDP OVERFLOW, TRY/P

?INPUT ERROR ON DEVICE DEV

?DATA ERROR ON DEVICE DEV

?DEV NOT AVAILABLE

?THERE ARE N ERRORS

This is the total number of errors detected by MACRO during assembly. These are the errors marked by letter codes on the listing. Under BATCH, if there are one or more errors the run is terminated.

CHAPTER 5
RELOCATION

The MACRO-10 assembler will create a relocatable object program. This program may be loaded into any part of memory as a function of what has been previously loaded. To accomplish this, the address field of some instructions must have a relocation constant added to it. This relocation constant, added at load time by the PDP-10 Loader, equals the difference between the memory location an instruction is actually loaded into and the location it is assembled into. If a program is loaded into cells beginning at location 1400_8 , the relocation constant k would be 1400_8 .

Not all instructions must be modified by the relocation constant. Consider the two instructions:

```
MOVEI 2,.-3
MOVEI 2,1
```

The first is used in address manipulation and must be modified; the second probably should not. To accomplish the relocation, the actual expression forming an address is evaluated and marked for modification by the Linking Loader. Integer elements are absolute and not modified. Point elements (.) are relocatable and are always modified.* Symbolic elements may be either absolute or relocatable. If a symbol is defined by a direct assignment statement, it may be relocatable or absolute depending on the expression following the equal sign (=). If a symbol is defined as a macro, it is replaced by the string and the string itself is evaluated. If it is defined as a label or a variable (#), it is relocatable.* Finally, references to literals are relocatable.*

To evaluate the relocatability of an expression, consider what happens at load time. A constant, k , must be added to each relocatable element and the expression evaluated. Consider the expression:

$$X = A + 2 * B - 3 * C + D$$

*Except under the LOC code or a PHASE code which specifies absolute addressing.

where A, B, C, and D are relocatable. Assume k is the relocation constant. Adding this to each relocatable term we get:

$$X_R = (A+K) + 2*(B+K) - 3*(C+K) + (D+K)$$

This expression may be rearranged to separate the ks, yielding:

$$X_R = A + 2*B - 3*C + D + K$$

This expression is suitable for relocation since it involves the addition of a single k. In general, if the expression can be rearranged to result in the addition of

$0*K$	The expression is legal and fixed.
$1*K$	The expression is legal and relocatable.
$N*K$	Where n is any positive or negative integer other than 0 or 1, the expression is illegal.

Finally, if the expression involves k to any power other than 1, the expression is illegal. This leads to the following conventions:

- Only two values of relocatability for a complete expression are allowed, k and 0.
- An element may not be divided by a relocatable element.
- Two relocatable elements may not be multiplied together.
- Relocatable elements may not be combined by the Boolean operators.

If any of these rules are broken, the expression is illegal and the assembled code is flagged.

If A, C, and B are relocatable symbols, then:

A+B-C	is relocatable
A-C	is fixed
A+2	is relocatable
2*A-B	is relocatable
2&A-B	is illegal

A storage word may be relocatable in the left half as well as the right half. For example:

XWD A, B

CHAPTER 6 ASSEMBLY OUTPUT

There are two MACRO-10 outputs, a binary program and a program listing. The listing is controlled by the listing control pseudo-ops, which were described in Chapter 2.

6.1 ASSEMBLY LISTING

All MACRO-10 programs begin with an implicit LIST statement. From left to right, the columns on a listing page contain:

- a. The 6-digit address of each storage word in the binary program. These are normally sequential location counter assignments. In the case of a block statement, only the address of the first word allocated is listed.
- b. The assembled instruction and data words, shown in two columns for easier reading, the 6-digit left half-word and the 6-digit right half-word. An apostrophe following either half-word indicates that the word is relocatable.
- c. The source program statement, as written by the programmer, followed by comments, if any.

If an error is detected during assembly of a statement, an error code is printed on that statement's line, near the left edge of the page. If multiple errors of the same type occur in a particular statement, the error code is printed only once; but if several errors, each of a different type, occur in a statement, an error code is printed for each error. The total number of errors is printed at the end of the listing.

The program break is also printed at the end of the listing. This is the highest relocatable location assembled, plus one. This is the first location available for the next program or for patching.

6.2 BINARY PROGRAM OUTPUT

The assembler produces binary program output in four formats. The choice depends on whether the program is relocatable or absolute, and on the loading procedure to be used to load the program for execution.

6.2.1 Relocatable Binary Programs - LINK Format

Most binary programs are output in LINK format. Like the RELOC statement, the LINK format output is implicit and is automatically produced for all relocatable MACRO-10 programs unless another format (RIM, RIM10, RIM10B) is explicitly requested. The LINK format is the only format that may be used with the Linking Loader.

The Linking Loader loads subprograms into memory, properly relocating each one and adjusting addresses to compensate for the relocation. It also links external and internal symbols to provide communication between independently assembled subprograms. Finally, the Linking Loader loads subroutines in library search mode.

Data for the Linking Loader is formatted in blocks. All blocks have an identical format. The first word of a LINK block consists of two halves. The left half is a code for the block type, and the right half is a count of the number of data words in the block. The data words are grouped in sub-blocks of 18 items. Each 18-word sub-block is preceded by a relocation word. This relocation word consists of 18 2-bit bytes. Each byte corresponds to one word in the sub-block, and contains relocation information regarding that word.

If the byte value is:

0	no relocation occurs
1	the right half is relocated
2	the left half is relocated
3	both halves are relocated

These relocation words are not included in the count; they always appear before each sub-block of 18 words or less to insure proper relocation.

All relocatable programs may be stored in LINK format, including programs on paper tape, DECtape, magnetic tape, punched cards, and disks. This format is totally independent of logical divisions in the input medium. It is also independent of the block type.

6.2.1.1 LINK Formats for the Block Types

Block Type 1 Relocatable or Absolute Programs and Data

WORD 1	The location of the first data word in the block
WORD 2	A contiguous block of program or data words
.	
.	
WORD N	(N must be less than 2000,000 octal)

Block Type 2 Symbols

	Consists of word pairs
1ST WORD	Bits 0-3 code bits
1ST WORD	Bits 4-35 radix 50 representation of symbol (see below)
2ND WORD	Data (value or pointer)
CODE 04:	Global (internal) definition
2ND WORD	Bits 0-35 value of symbol
CODE 10:	Local definition
2ND WORD	Bits 0-35 value of symbol
CODE 60:	Chained global requests:
2ND WORD	Bits 0-17 = 0
2ND WORD	Bits 18-35 pointer to first word of chain requiring definition (see Loader Manual)
CODE 60:	Global symbol additive request: (see Loader Manual)
2ND WORD	Bit 0 = 1
BIT 1	Subtract value before addition
BIT 2	Swap halves before addition
BIT 3	Rotate left 5 before addition
BIT 9	Replace left half with result in storage
BIT 10	Replace right half with result in storage
BIT 11	Replace index field with result in storage
BIT 12	Replace accumulator field with result in storage
BITS 18-35	Pointer to word requiring addition

Block Type 3 Load Into High Segment

When block type 3 is present in a relocatable binary program, the Loader loads the program into the high segment if the system has re-entrant (two-segment) capability. When used, block type 3 appears immediately after any entry blocks (type 4). This block type transmits no additional data.

Block Type 4 Entry Block

This block contains a list of radix 50 symbols, each of which may contain a 0 or 1 in the high-order code bit. Each represents a series of logical AND conditions. If all the globals in any series are requested, the following program is loaded. Otherwise, all input is ignored until the next end block. This block must be the first block in a program.

Block Type 5 End Block

This is the last block in a program. It contains one word which is the program break, that is, the location of the first free register above the program. (Note: This word is relocatable.) It is the relocation constant for the following program loaded.

Block Type 6 Name Block

The first word of this block is the program name (RADIX 50). It must appear before any type 2 blocks. The second word, if it appears, defines the length of common.

Block Type 7 Starting Address

The first word of this block is the starting address of the program. The starting address for a relocatable program may be relocated by means of the relocation bits.

Block Type 10 Internal Request

Each data word is one request. The left half is the pointer to the program. The right half is the value. Either quantity may be relocatable.

6.2.2 Absolute Binary Programs

Three output formats are available for absolute (non-relocatable) binary programs. These are requested by the RIM, RIM10 and RIM10B statements.

6.2.2.1 RIM10B Format - If a program is assembled into absolute locations (not relocatable), a RIM10B statement following the LOC statement at the beginning of the source program causes the assembler to write out the object program in TIM10B format. This format is designed for use with the PDP-10 hardware read-in feature.

The program is punched out during pass 2, starting at the location specified in the LOC statement. If the first two statements in the program are:

```
LOC 1000 )
RIM10B )
```

the assembler assembles the program with absolute addresses starting at 1000, and punches out the program in RIM10B format, also starting at location 1000. The programmer may reset the location counter during assembly of his program, but only one RIM10B statement is needed to punch out the entire program.

In RIM10B format, (see Figures 6-1 and 6-2) the assembler punches out the RIM10B Loader, (Figure 6-2) followed by the program in 17-word (or less) data blocks, each block separated by blank tape. The assembler inserts an I/O transfer word (IOWD) preceding each data block, and also inserts a 36-bit checksum following each data block as shown in Figure 6-1. The word count in the IOWD includes only the data words in the block, and the checksum is the simple 36-bit added checksum of the IOWD and the data words.

Data blocks may contain less than 17 words. If the assembler assigns a non-consecutive location, the current data block is terminated, and an IOWD containing the next location is inserted, starting a new data block.

The transfer block consists of two words. The first word of the transfer block is an instruction obtained from the END statement (See Section 6.2.2.4.) and is executed when the transfer block is read. The second is a dummy word to stop the reader.

6.2.2.2 RIM10 Format - Binary programs in RIM10 format are absolute, unblocked, and not checksummed. When the RIM10 statement follows a LOC statement in a program, the assembler punches out each storage word in the object program, starting at the absolute address specified in the LOC statement.

In order to use the Read-in-Mode switch with format, the programmer must begin with the statement:

```
IOWD N,FIRST )
```

where n is the length of the program including the transfer instruction at the end, and FIRST is the first memory location to be occupied. The last location must be a transfer instruction to begin the program, such as:

```
JRST 4,GO )
```

For example, if a program with RIM10 output has its first location at START and its last location at FINISH, the programmer may write:

```
IOWD FINISH-START+1,START )
```

NOTE

In cases where the location counter is increased but no binary output occurs (such as with BLOCK, LOC n, and LIT pseudo-ops), MACRO inserts a zero word into the binary output file for each location skipped by the location counter.

6.2.2.3 RIM Format - This format, which is primarily used in PDP-6 systems, consists of a series of paired words. The first word of each pair is a paper-tape read instruction giving the core memory address of the second word. The second word is the data word.

```
DATAI PTR,LOC
DATA WORD
```

The last pair of words is a transfer block. The first word is an instruction obtained from the END statement (See Section 6.2.2.4) and is executed when the transfer block is read. The second word is a dummy word to stop the reader.

The loader that reads this format is:

```
LOC 20
CONO PTR,60
A: CONSO PTR,10
   JRST *-1
   DATAI PTR,B
   CONSO PTR,10
   JRST *-1
B: 0
   JRST A
```

This loader is normally toggled into memory and started at location 20.

6.2.2.4 END Statements - When the programmer wants output in either RIM or RIM10B format, he may insert an instruction or starting address as the first word in the two-word transfer block by writing the instruction or address as an argument to the END statement. The second word of the transfer block is zero. In RIM10 assemblies, this argument is ignored.

If bits 0 through 8 of the instruction are zero, MACRO will insert the instruction JRST 4, 0, causing a halt when executed. The END statements

```
END SA ) OR END JRST SA )
```

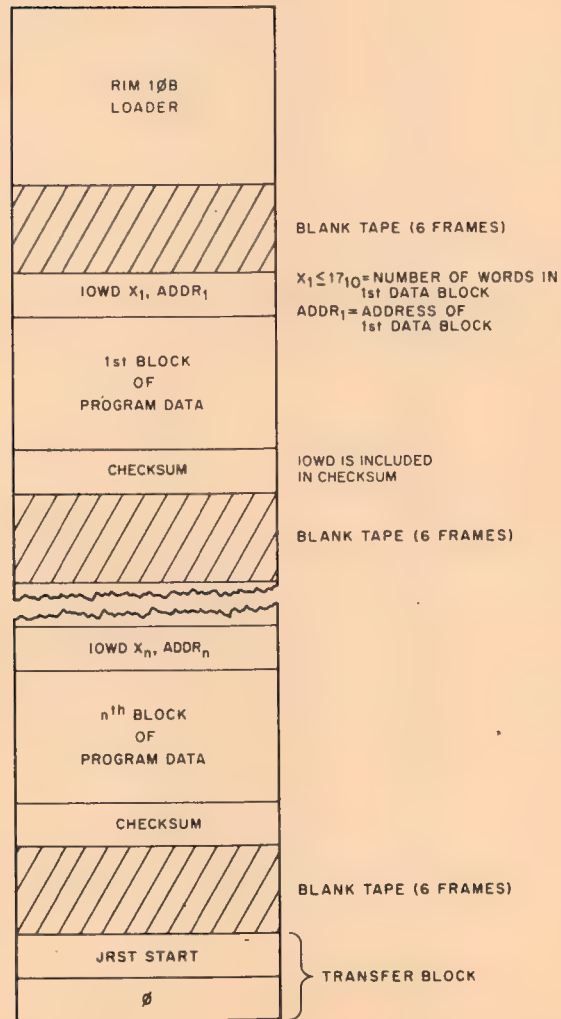
will start automatically at address SA.

Some other examples:

1st Transfer Block Word

```

END XCT@1234   XCT@1234
END Z4,SA     JRST 4,SA
END           JRST 4,0
  
```



10-0060

Figure 6-1 General RIM10B Format

```

XWD -16,0
ST:      CONO PTR,60
ST1:     HRRI A,RD+1
RD:      CONSO PTR,10
          JRST
          DATAI PTR,@TBL1-RD+1(A)
          XCT      TBL1-RD+1(A)
          XCT      TBL2-RD+1(A)
A:       SOJA A,
TBL1:    CAME CKSM,ADR
          ADD CKSM,1(ADR)
          SKIPL CKSM,ADR
TBL2:    JRST 4,ST
          AOBJN ADR,RD
ADR:     JRST     ST1
CKSM=ADR+1

```

Figure 6-2 RIM10B Loader

CHAPTER 7 PROGRAMMING EXAMPLES

A MACRO-10 routine for calculating the logarithm of a complex argument is shown in Figure 7-1. The routine begins with an ENTRY statement, identifying this library routine as CLOG (Complex Logarithm Function), and uses three external routines, ALOG, ATAN2 and CABS.

The second example, shown in Figure 7-2, contains a nested macro, SBL, and uses conditional assembly statements, which cause PIXTART and PIXOPT to be generated as either internal or external symbols, depending on the value of SBLSW. In the example, both are externals.

The third example, Figure 7-3, shows two ways of writing a byte unpacking subroutine. Both UNPACK and UNPAX use literals to set up pointer words, and load the bytes in accumulators 0 and 1. The calling sequence for UNPACK actually contains the bytes to be unpacked. For UNPAX, the calling sequence contains the address of the bytes, thus, UNPAX must refer to them indirectly.

The fourth example, Figure 7-4, demonstrates the use of the IRPC (indefinite repeat character) pseudo-op. A macro call, HEX, is made with the arguments ANS, a symbol name, and F, a hexadecimal number. The processing of the macro causes the symbol, ANS, to be assigned the converted value of the hexadecimal number, F. In this example the hexadecimal "digits", listed in ascending order, are: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F.

NOTE

Each complete program (see Figures 7-1 and 7-4) must have an END statement. All other statements may be used at the programmer's discretion; however, a TITLE statement is recommended for documentation and debugging purposes.

CLOG MACROX.H 13:46 7-APR-67 PAGE 1

TITLE CLOG
SUBTTL APRIL 7, 1967

;COMPLEX LOGARITHM FUNCTION
;THIS ROUTINE CALCULATES THE LOGARITHM OF A COMPLEX ARGUMENT
; Z = X+I*Y WITH THE FOLLOWING ALGORITHM

;LOG(Z) = LOG(ABSF(Z)) + I*THETA
;WHERE ABSF(Z) = SORT(X+2 + Y+2)
;AND THETA IS THE COMPLEX ANGLE ATAN(Y/X)

;THE ROUTINE IS CALLED IN THE FOLLOWING MANNER:
; JSA 0,CLOG
; EXP ARG
;THE REAL PART OF THE ANSWER IS RETURNED IN ACCUMULATOR A
;AND THE IMAGINARY PART IS RETURNED IN ACCUMULATOR B

ENTHY CLOG
EXTERN ALOG,ATAN2,CABS

000000 A=0
000001 B=1
000010 C=10
000011 D=11
000016 Q=16

000000	000000	000000	CLOG:	0		;ENTRY TO COMPLEX LOG ROUTINE
000001	201436	000000		MOVEI	C,C(1)	;GET ADDRESS OF COMPLEX ARGUMENT
000002	200450	000001		MOVE	D,1(C)	;GET IMAGINARY PART OF ARGUMENT
000003	200410	000000		MOVE	C,C(C)	;GET REAL PART OF ARGUMENT
000004	266700	000000		JSA	1,CABS	;CALCULATE MAGNITUDE OF Z
000005	000000	000010		EXP	C	;ADDRESS OF COMPLEX ARGUMENT
000006	266700	000000		JSA	0,ALOG	;CALCULATE LOG(ABSF(Z))
000007	000000	000000		EXP	A	;ADDRESS FOR LOG ROUTINE
000010	250000	000010		EXCH	A,C	;SWAP ANSWER WITH REAL PART
000011	266700	000000		JSA	0,ATAN2	;CALCULATE ANGLE AS ATAN(Y/X)
000012	000000	000011		EXP	D	;ADDRESS OF Y
000013	000000	000000		EXP	A	;ADDRESS OF C
000014	200040	000000		MOVE	B,A	;PUT THETA IN IMAGINARY PART
000015	200000	000010		MOVE	A,C	;RESTORE REAL PART
000016	267716	000001		JRA	A+1(Q)	;EXIT

END

THERE ARE NO ERRORS

PROGRAM BREAK IS 000017

CLOG MACROX.H 13:46 7-APR-67 PAGE 2

SYMBOL TABLE

A	000000	
ALOG	000006	EXT
ATAN2	000011	EXT
B	000001	
C	000010	
CABS	000004	EXT
CLOG	000000	INT
D	000011	
Q	000016	

SK CORE USED

Figure 7-1 Sample Program, CLOG

```

LALL
000001 PSBLSW=1
        DEFINE SBLR (A)<IRP A,<SBL A>>
        DEFINE SBL (A)<
                IFE SBLSW-PSBLSW,<INTERN A>
                IFN SBLSW-PSBLSW,<EXTERN A>>
000003 SBLSW=3
        SBLR <PIXSTART,PIXOPT>↑IRP
        SBL PIXSTART↑
                IFE SBLSW-PSBLSW,<INTERN PIXSTART>
                IFN SBLSW-PSBLSW,<EXTERN PIXSTART>↑
        SBL PIXOPT↑
                IFE SBLSW-PSBLSW,<INTERN PIXOPT>
                IFN SBLSW-PSBLSW,<EXTERN PIXOPT>↑
↑

;GENERATES PIXSTART AND PIXOPT AS EXTERNALS

```

Figure 7-2 Example of Nested Macro

```

A=1
B=3
C=10

JSP 17,UNPACK
BYTE (3)A(15)B(18)C
.
.
.
UNPACK: LDB 0,[POINT 3,0(17),2]           ;PICK UP A
        LDB 1,[POINT 15,0(17),17]        ;PICK UP B
        HRRZ 2,0(17)                     ;PICK UP C
        JRST 1(17)                       ;RETURN
.
.
.
JSP 17,UNPAX
EXP [BYTE (3)A(15)B(18)C]
.
.
.
UNPAX:  LDB 0,[POINT 3,@0(17),2]         ;PICK UP A
        LDB 1,[POINT 15,@0(17), 17]     ;PICK UP B
        HRRZ 2,@0(17)                   ;PICK UP C
        JRST 1(17)

```

Figure 7-3 Two Byte Unpacking Subroutines

.MAIN MACRO.V34 15:23 24-MAR-69 PAGE 1

```

LALL

DEFINE HEX (N,X)
N=0
IRPC X,<IFGE "X"-"A",<N=N*†D16+"X"-"A"†D10>
      IFLE "X"-"9",<N=N*†D16+"X"-"0">>

      HEX ANS,F†
000000 ANS=0
      IRPC
000017 IFGE "F"-"A",<ANS=ANS*†D16+"F"-"A"†D10>
      IFLE "F"-"9",<ANS=ANS*†D16+"F"-"0">
      †
      HEX ANS,10†
000000 ANS=0
      IRPC
000001 IFGE "1"-"A",<ANS=ANS*†D16+"1"-"A"†D10>
      IFLE "1"-"9",<ANS=ANS*†D16+"1"-"0">
000020 IFGE "0"-"A",<ANS=ANS*†D16+"0"-"A"†D10>
      IFLE "0"-"9",<ANS=ANS*†D16+"0"-"0">
      †
      HEX ANS,9ABCDEF†
000000 ANS=0
      IRPC
000011 IFGE "9"-"A",<ANS=ANS*†D16+"9"-"A"†D10>
      IFLE "9"-"9",<ANS=ANS*†D16+"9"-"0">
000232 IFGE "A"-"A",<ANS=ANS*†D16+"A"-"A"†D10>
      IFLE "A"-"9",<ANS=ANS*†D16+"A"-"0">
004653 IFGE "B"-"A",<ANS=ANS*†D16+"B"-"A"†D10>
      IFLE "B"-"9",<ANS=ANS*†D16+"B"-"0">
115274 IFGE "C"-"A",<ANS=ANS*†D16+"C"-"A"†D10>
      IFLE "C"-"9",<ANS=ANS*†D16+"C"-"0">
000002 325715 IFGE "D"-"A",<ANS=ANS*†D16+"D"-"A"†D10>
      IFLE "D"-"9",<ANS=ANS*†D16+"D"-"0">
000046 536336 IFGE "E"-"A",<ANS=ANS*†D16+"E"-"A"†D10>
      IFLE "E"-"9",<ANS=ANS*†D16+"E"-"0">
001152 746757 IFGE "F"-"A",<ANS=ANS*†D16+"F"-"A"†D10>
      IFLE "F"-"9",<ANS=ANS*†D16+"F"-"0">
      †
END

```

NO ERRORS DETECTED

PROGRAM BREAK IS 000000

Figure 7-4 IRPC Example

OP CODES, PSEUDO-OPS, AND
MONITOR I/O COMMANDS

This appendix contains a complete list of assembler defined operators including machine instruction mnemonic codes, assembler pseudo-ops, Monitor programmed operators, and FORTRAN programmed operators. These programmed operators, or user utilized operation codes are called UOO's in the list.

The notes are used to specify which pseudo-ops generate data, and which do not. Pseudo-ops which generate data may be used within literals, and in address operand fields.

The initial values given by MACRO-10 to I/O instructions and FORTRAN UOO's for which the octal op code is not shown, are also given in the notes. These may be useful in checking listings.

ASSEMBLER PSEUDO-OPS AND MONITOR COMMANDS

ASCII, pseudo-op, generates data	NLI., 031, FORTRAN UOO
ASCIZ, pseudo-op, generates data	NLO., 032, FORTRAN UOO
BLOCK, pseudo-op, no data generated	NOSYM, pseudo-op, no data generated
BYTE, pseudo-op, generates data	OCT, pseudo-op, generates data
CALL, 040, Monitor UOO	OPDEF, pseudo-op, no data generated
CALLI, 047, Monitor UOO	OPEN, 050, Monitor UOO
CLOSE, 070, Monitor UOO	OUT, 057, Monitor UOO
DATA., 020, FORTRAN UOO	OUT., 017, FORTRAN UOO
DEC, pseudo-op, generates data	OUTBUF, 065, Monitor UOO
DEC., 033, FORTRAN UOO	OUTF., 027, FORTRAN UOO
DEFINE, pseudo-op, no data generated	OUTPUT, 067, Monitor UOO
DEPHASE, pseudo-op, no data generated	PAGE, pseudo-op, no data generated
ENC., 034, FORTRAN UOO	PASS2, pseudo-op, no data generated
END, pseudo-op, no data generated	PHASE, pseudo-op, no data generated
ENTER, 077, Monitor UOO	POINT, pseudo-op, generates data
ENTRY, pseudo-op, no data generated	PRINTX, pseudo-op, no data generated
EXP, pseudo-op, generates data	PURGE, pseudo-op, no data generated
EXTERN, pseudo-op, no data generated	RADIX, pseudo-op, no data generated
FIN., 021, FORTRAN UOO	RADIX50, pseudo-op, generates data
GETSTS, 062, Monitor UOO	RELEAS, 071, Monitor UOO
HISEG, pseudo-op, no data generated	RELOC, pseudo-op, no data generated
IF1, conditional pseudo-op	REMARK, pseudo-op, no data generated
IF2, conditional pseudo-op	RENAME, 055, Monitor UOO
IFB, conditional pseudo-op	REPEAT, pseudo-op, no data generated
IFDEF, conditional pseudo-op	RERED., 030, FORTRAN UOO
IFDIF, conditional pseudo-op	RESET., 015, FORTRAN UOO
IFE, conditional pseudo-op	RIM, pseudo-op, no data generated
IFG, conditional pseudo-op	RIM10, pseudo-op, no data generated
IFGE, conditional pseudo-op	RIM10B, pseudo-op, no data generated
IFIDN, conditional pseudo-op	RTB., 022, FORTRAN UOO
IFL, conditional pseudo-op	SETSTS, 060, Monitor UOO
IFLE, conditional pseudo-op	SIXBIT, pseudo-op, generates data
IFN, conditional pseudo-op	SLIST., 025, FORTRAN UOO
IFNB, conditional pseudo-op	SQUOZE, same as RADIX50
IFNDEF, conditional pseudo-op	STATO, 061, Monitor UOO
IN, 056, Monitor UOO	STATUS, 062, Monitor UOO
IN., 016, FORTRAN UOO	STATZ, 063, Monitor UOO
INBUF, 064, Monitor UOO	STOPI, pseudo-op, no data generated
INF., 026, FORTRAN UOO	SUBTTL, pseudo-op, no data generated
INIT, 041, Monitor UOO	SYN, pseudo-op, no data generated
INPUT, 066, Monitor UOO	TAPE, pseudo-op, no data generated
INTERN, pseudo-op, no data generated	TITLE, pseudo-op, no data generated
IOWD, pseudo-op, generates data	TTCALL, 051, Monitor UOO
IRP, pseudo-op, no data generated	UGETF, 073, Monitor UOO
IRPC, pseudo-op, no data generated	UJEN, 100, Monitor UOO
LALL, pseudo-op, no data generated	USETI, 074, Monitor UOO
LIST, pseudo-op, no data generated	USETO, 075, Monitor UOO
LIT, pseudo-op, no data generated	VAR, pseudo-op, generates data
LOC, pseudo-op, no data generated	WTB., 023, FORTRAN UOO
LOOKUP, 076, Monitor UOO	XALL, pseudo-op, no data generated
MLOFF, pseudo-op, no data generated	XLIST, pseudo-op, no data generated
MLON, pseudo-op, no data generated	XWD, pseudo-op, generates data
MTAPE, 072, Monitor UOO	Z, pseudo-op, generates data
MTOP., 024, FORTRAN UOO	

MACHINE MNEMONICS AND OCTAL CODES

ADD	270	CAMGE	315	FSBRI	155	HRREM	572	MOVEM	202	SETMI	415	TLCA	645
ADDB	273	CAML	311	FSBRM	156	HRRES	573	MOVES	203	SETMM	416	TLCE	643
ADDI	271	CAML3	313	FSC	132	HRRI	541	MOVN	214	SETO	474	TLCN	647
ADDM	272	CAMN	316	HALT	254-4,	HRRM	542	MOVMI	215	SETOB	477	TLN	601
AND	404	CLEAR	400	HLL	500	HRRO	560	MOVMM	216	SETOI	475	TLNA	605
ANDB	407	CLEARB	403	HLLB	530	HRROI	561	MOVMS	217	SETOM	476	TLNE	603
ANDCA	410	CLEARI	401	HLLBI	531	HRROM	562	MOVN	210	SETZ	400	TLNN	607
ANDCAB	413	CLEARM	402	HLLBM	532	HRROS	563	MOVNI	211	SETZB	403	TLO	661
ANDCAI	411	CONI	7-24	HLLBS	533	HRRS	543	MOVNM	212	SETZI	401	TLOA	665
ANDCAM	412	CONO	7-20	HLLBI	501	HRRZ	550	MOVNS	213	SETZM	402	TLOE	663
ANDCB	440	CONSO	7-34	HLLM	502	HRRZI	551	MOVSI	205	SKIP	330	TLON	667
ANDCBB	443	CONSZ	7-30	HLLS	510	HRRZM	552	MOVSM	206	SKIPA	334	TLZA	621
ANDCBI	441	DATAI	7-04	HLLZI	511	HRRZS	553	MOVSS	207	SKIPE	332	TLZE	623
ANDCBM	442	DATAO	7-14	HLLZM	512	IBP	133	MUL	224	SKIPG	337	TLZN	627
ANDCM	420	DFN	131	HLLZS	513	IDIV	230	MULB	227	SKIPGE	335	TRC	640
ANDCMB	423	DIV	234	HLR	544	IDIVB	233	MULI	225	SKIPL	331	TRCA	644
ANDCMI	421	DIVB	237	HLRE	574	IDIVI	231	MULM	226	SKIPL	333	TRCE	642
ANDCMM	422	DIVI	235	HLREI	575	IDIVM	232	OR	434	SKIPN	336	TRCN	646
ANDI	405	DIVM	236	HLRES	577	IDP	136	ORB	437	SOJ	360	TRN	600
ANDM	406	DPB	137	HLRI	545	ILDB	134	ORCA	454	SOJA	364	TRNA	604
AOBJN	253	EQV	444	HLRM	546	IMUL	220	ORCAB	457	SOJE	362	TRNE	602
AOBJP	252	EQVB	447	HLRO	564	IMULB	223	ORCAI	455	SOJG	367	TRNN	606
AOJ	340	EQVI	445	HLROI	565	IMULI	221	ORCAM	456	SOJGE	365	TRO	660
AOJA	344	EQVM	446	HLROM	566	IMULM	222	ORCB	470	SOJL	361	TROA	664
AOJE	342	EXCH	250	HLRS	547	IOR	434	ORCBM	472	SOJLE	363	TROE	662
AOJG	347	FAD	140	HLRZ	554	IORB	437	ORCB	471	SOJN	366	TRON	666
AOJGE	345	FADB	143	HLRZI	555	IORI	435	ORCBM	472	SOS	370	TRZ	620
AOJL	341	FADL	141	HLRZM	556	IORM	436	ORCM	464	SOSA	374	TRZE	622
AOJLE	343	FADM	142	JCRY	255-6,	JCRY	255-6,	ORCM	464	SOSE	372	TRZA	624
AOJN	346	FADR	144	JCRY0	255-4,	JCRY0	255-4,	ORCMB	467	SOSG	377	TRZE	622
AOS	350	FADRB	147	JCRY1	255-2,	JCRY1	255-2,	ORCMI	465	SOSGE	375	TRZN	626
AOSA	354	FADRI	145	JEN	254-12,	JEN	254-12,	ORCMM	466	SOSL	371	TSC	651
AOSE	352	FADRM	146	JFCL	255	JFCL	255	ORI	435	SOSLE	373	TSCA	655
AOSG	357	FDV	170	JFLO	243	JFLO	243	ORM	436	SOSN	376	TSCE	653
AOSGE	355	FDVB	173	JFOV	255-1,	JFOV	255-1,	POP	262	SUB	274	TSCN	657
AOSL	351	FDVL	171	JOV	255-10,	JOV	255-10,	POPJ	263	SUBB	277	TSN	611
AOSLE	353	FDVM	172	JRA	267	JRA	267	PUSH	261	SUBI	275	TSNA	615
AOSN	356	FDVR	174	JRST	254	JRST	254	PUSHJ	260	SUBM	276	TSNE	613
ASH	240	FDVRB	177	JRSTF	254-2,	JRSTF	254-2,	ROT	241	TDC	650	TSNN	617
ASHC	244	FDVRI	175	JSA	266	JSA	266	ROTC	245	TDCA	654	TSO	671
BLKI	7-00	FDVRM	176	JSP	265	JSP	265	RSW	7-04	TDCE	652	TSOA	675
BLKO	7-10	FMP	160	JSR	264	JSR	264	SETA	424	TDEN	610	TSOE	673
BLT	251	FMPB	163	JUMP	320	JUMP	320	SETAB	427	TDN	610	TSOZ	631
CAI	300	FMPPL	161	JUMPA	324	JUMPA	324	SETAI	425	TDNA	614	TSZ	631
CAIA	304	FMPM	162	JUMPE	322	JUMPE	322	SETAM	426	TDNE	612	TSZA	635
CAIE	302	FMPR	164	JUMPG	327	JUMPG	327	SETCA	450	TDNN	616	TSZE	633
CAIG	307	FMPRB	167	JUMPG	325	JUMPG	325	SETCAB	453	TDO	670	TSZN	637
CAIGE	305	FMPRI	165	JUMPL	321	JUMPL	321	SETCAI	451	TDQA	674	UFA	130
CAIL	301	FMPRM	166	JUMPLE	323	JUMPLE	323	SETCAM	452	TDOE	672	XCT	256
CAILE	303	FSB	150	JUMPN	326	JUMPN	326	SETCM	460	TDON	676	XOR	430
CAIN	306	FSBB	153	LDB	135	LDB	135	SETCMB	463	TDZA	630	XORB	433
CAM	310	FSBL	151	LSH	242	LSH	242	SETCMI	461	TDZB	634	XORI	431
CAMA	314	FSBM	152	LSHC	246	LSHC	246	SETCMM	462	TDZE	632	XORM	432
CAME	312	FSBR	154	MOVE	200	MOVE	200	SETM	414	TDZN	636		
CAMG	317	FSBRB	157	MOVEI	201	MOVEI	201	SETMB	417	TLC	641		

APPENDIX B
SUMMARY OF PSEUDO-OPS

ASCII	Seven-bit ASCII text.
ASCIIZ	Seven-bit ASCII test, with null character guaranteed at end.
BLOCK	Reserves block of storage cells.
BYTE	Input bytes of length 1-36 bits.
DEC	Input decimal numbers.
DEFINE	defines macro
DEPHASE	Terminates PHASE relocation mode.
END	Last statement of the program.
ENTRY	Enters subroutine library.
EXP	Input expressions.
EXTERN	Identifies external symbols.

Conditional Assembly Statements

Assemble if:

IF1	Encountered during pass 1
IF2	Encountered during pass 2
IFB	Blank
IFDEF	Defined
IFDIFF	Different
IFE	Zero
IFG	Positive
IFGE	Zero, or positive
IFIDN	Identical
IFL	Negative
IFLE	Zero, or negative
IFN	Non-zero
IFNB	Not blank

<u>Format</u>	<u>Operator</u>	<u>Page</u>	<u>Type</u>	<u>Notes</u>
PRI	ANDCAB	413		
	ANDCAI	411		
	ANDCAM	412		
	ANDCB	440		
	ANDCBB	443		
	ANDCBI	441		
	ANDCBM	442		
	ANDCM	420		
	ANDCMB	423		
	ANDCMI	421		
	ANDCMM	422		
PRI	AND I	405		
	ANDM	406		
	AOBJN	253		
	AOBJP	252		
	A0J	340		
	A0JA	344		
	A0JE	342		
	A0JG	347		
	A0JGE	345		
	A0JL	341		
	A0JLE	343		
	A0JN	346		
	A0S	350		
	A0SA	354		
	A0SE	352		
	A0SG	357		
	A0SGE	355		

VAR	Assemble variables suffixed with [#]
XALL	Stop expanded listing
XLIST	Stop listing
XWD	Input two 18-bit half words
Z	Input zero word

APPENDIX C
SUMMARY OF CHARACTER INTERPRETATIONS

The characters listed below have special meaning in the contexts indicated. These interpretations do not apply when these characters appear in text strings, or in comments.

<u>Character</u>	<u>Meaning</u>	<u>Example</u>
:	Colon. Immediately follows all labels.	LABEL: Z
;	Semi-colon. Precedes all comments.	;THIS IS A COMMENT
.	Point. Has current value of the location counter.	JRST .+5 JUMP FORWARD FIVE LOCATIONS
,	Comma. General operand or argument delimiter	DEC 10,5,6 EXP A + B, C - D
	Accumulator field delimiter	MOVEI 1,TAG
	References accumulator 0. The comma is optional.	MOVEI ,TAG
	Delimits macro arguments.	MACRO (A,B,C)
!	Inclusive OR AND Multiplication Division Add (+A outputs the value of A) Subtract	Logical Operators Arithmetic Operators
1st character of text string	In ASCII, ASCIZ and SIXBIT test strings, the first non-blank character is the delimiter.	ASCII/STRING/;
B	Follows number to be shifted and precedes binary shift count.	7B2
E	Exponent. Precedes decimal exponent in floating-point numbers.	F22.1E5 EXPONENT IS 5.

()	<p>Parentheses. Use to enclose index fields.</p> <p>Enclose the byte size in BYTE statements.</p> <p>Enclose the dummy argument string in macro DEFINE statements.</p>	<pre>ADD AC1,X (7) MOVEI A,(SIXBIT/ABC/) BYTE (6) 8, 8, 7 DEFINE MAC(A,B,C)</pre>
< >	<p>Angle brackets. In an expression, enclose a numeric quantity.</p> <p>In conditional assembly statements contain a single argument, and the conditional coding.</p> <p>In REPEAT statements, contain coding to be repeated.</p> <p>In macros, enclose the macro definition</p>	<pre><A-B+500/C> IFI, <MOVE AC0, TAX> REPEAT 3, <SUB 17, TAG> DEFINE PUNCH <DATA0 PTP, PUNBUF (4)></pre>
[]	<p>Square brackets. Delimits literals.</p> <p>In OPDEF statement, contain new operator.</p>	<pre>ADD 5,[MOVEI 3,TAX] OPDEF CAL [MOVE]</pre>
=	<p>Equal sign, direct assignment</p>	<pre>SYM=6 SYM-A+B*D</pre>
"..."	<p>Quotation marks enclose 7-bit ASCII text, from one to five characters.</p>	<pre>"ABCDE"</pre>
#	<p>Number sign. Defines a symbol used as a tag. Variable.</p>	<pre>ADD 3,TAG#</pre>
'	<p>Apostrophe or single quote. Catenation character, used only within macro definitions.</p>	<pre>DEFINE MAC (A,B,C); <JUMP 'A B, C></pre>
\	<p>Reverse slash. If used as the first character in a macro call, the value of the following symbol is converted to an ASCII symbol in the current radix.</p>	<pre>MAC \ A IF A=500, THIS GENERATES THREE 7-BIT ASCII CHARACTERS, ASCII/500/</pre>
←	<p>Control left arrow. Line continuation.</p> <p>Left arrow. N-M shift n left (or right) n times.</p>	<pre>100←3=1000 100←3=10</pre>

APPENDIX D
ASSEMBLER EVALUATION OF
STATEMENTS AND EXPRESSIONS

Order of Statement Evaluation:

The following table shows the order in which the assembler searches each statement field.

<u>Label field</u>	<u>Operator field</u>	<u>Operand fields</u>
1. Symbol suffixed by colon. If colon not found no label is present.	1. Number 2. Macro/OPDEF 3. Machine operator 4. Assembler operator 5. Symbol	1. Number 2. Symbol 3. Macro/OPDEF 4. Machine Operator 5. Assembler Operator

Notice that a single symbol could be used as a label, an operator, or an operand, depending upon where it is used.

The assembler checks the operator field for a number, first, and if found, assumes that no operator is present. Likewise, if a symbol is not a macro, OPDEF, machine operator or assembler operator, the assembler will search the symbol table. If a defined symbol is found, no operator is present.

If a defined operator appears in an operand field, it must generate at least one word of data. Statements which do not generate data may not be used as part of operand expressions. If a statement used in an operand expression generates more than one word of data, only the first word generated is meaningful.

Order of Expression Evaluation:

- (Unary operator)
- ↑D, ↑O, ↑B, ↑F, ↑L
- B Shift, ← Shift
- Logical operators
- Multiply/Divide
- Add/, Subtract

At each level, operations are performed left to right.

APPENDIX E
TEXT CODES

SIXBIT	Character	ASCII 7-BIT*	SIXBIT	Character	ASCII 7-Bit*	Character	ASCII 7-Bit*
00	Space	040	40	@	100	\	140
01	!	041	41	A	101	a	141
02	"	042	42	B	102	b	142
03	#	043	43	C	103	c	143
04	\$	044	44	D	104	d	144
05	%	045	45	E	105	e	145
06	&	046	46	F	106	f	146
07	'	047	47	G	107	g	147
10	(050	50	H	110	h	150
11)	051	51	I	111	i	151
12	*	052	52	J	112	j	152
13	+	053	53	K	113	k	153
14	,	054	54	L	114	l	154
15	-	055	55	M	115	m	155
16	.	056	56	N	116	n	156
17	/	057	57	O	117	o	157
20	0	060	60	P	120	p	160
21	1	061	61	Q	121	q	161
22	2	062	62	R	122	r	162
23	3	063	63	S	123	s	163
24	4	064	64	T	124	t	164
25	5	065	65	U	125	u	165
26	6	066	66	V	126	v	166
27	7	067	67	W	127	w	167
30	8	070	70	X	130	x	170
31	9	071	71	Y	131	y	171
32	:	072	72	Z	132	z	172
33	;	073	73	[133	{	173
34	<	074	74	\	134		174
35	=	075	75]	135	~	175
36	>	076	76	†	136	~	176
37	?	077	77	←	137	Delete	177

*MACRO-10 also accepts five of the 32 control codes in 7-bit ASCII:

Horizontal Tab	011	Vertical Tab	013	Carriage Return	015
Line Feed	012	Form Feed	014		

APPENDIX F
RADIX 50 REPRESENTATION

Radix 50_8 representation is used to condense 6 character sixbit symbols into 32 bits. Let each character of a symbol be subscripted in descending order from left to right; that is, let the symbols be of the form

$$L_6L_5L_4L_3L_2L_1$$

If C_n denotes the 6-bit code for L_n , the radix 50_8 representation is generated by the following:

$$((((C_6 * 50) + C_5) * 50 + C_4) * 50 + C_3) * 50 + C_2) * 50 + C_1$$

where all numbers are octal.

The code numbers corresponding to the characters are:

<u>Code (Octal)</u>	<u>Characters</u>
00	Null character
01-12	0-9
13-44	A-Z
45	.
46	\$
47	%

APPENDIX G
SUMMARY OF RULES FOR
DEFINING AND CALLING MACROS

Assembler Interpretation:

MACRO-10 assembles macros by direct and immediate character substitutions. Whenever a macro call is encountered, in any field, the character substitution is made, the characters are processed, and the assembler continues its scan with the character following the delimiter of the last argument, except when it is delimited by a semicolon. Macros can appear any number of times on a line.

Character handling:

- a. Blanks: A macro symbol is delimited by a blank or tab and the character following the delimiter is the start of the argument string, even if it is also a blank or a tab. Other than the delimiter, blanks and tabs are treated as standard characters in the argument string.
- b. Brackets: Angle brackets are only significant in the argument fields if the first character of any field is a left angle bracket. In this case, no terminator or parenthesis tests are made between it and its matching right bracket. The matching brackets are removed from the string but the scan continues until a standard delimiter is found.
- c. Parentheses: Parentheses serve only to terminate an argument scan. They are only significant when the first character following the blank or tab delimiter is a left parenthesis. In this case, it is removed and if its matching right parenthesis is encountered prior to the normal termination of the argument scan, it is removed and the scan discontinued.
- d. Commas: When a comma is encountered in an argument scan, it acts as the delimiter of the current argument. If it delimits the last argument, the character following it will be the first scanned after the substitution is processed.
- e. Semicolons: When a semicolon is encountered in an argument scan, the scan is discontinued. If some arguments have not been satisfied, the remainder is considered to be null. It is saved, however, and will be the first character scanned after the substitution is made, normally acting as a comment flag.
- f. Carriage return: A carriage return, except when pre-empted by angle brackets (see b above) will terminate the scan similar to the semicolon. This can be circumvented, if desired, by the control left arrow key described elsewhere.

g. Back-slash: If the first character of any argument is a back-slash, it must be directly followed by a numeric term. The value of the numeric term is broken down into a string of ASCII digits of the current radix, just the reverse of a fixed-point number computation. The value is considered to be a 36-bit positive number having a value of 0 to 777777 777777. Leading zeros are suppressed except in the case of 0, in which case the result is one ASCII 0. The ASCII string is substituted and the scan continued in the normal manner (no implied terminator).

The default listing mode is XALL, in which case the initial macro call and all lines within its range which produce binary code are listed. The pseudo-op LALL will cause all lines to be listed. Substituted arguments are bracketed by 's by the assembler.

Concatenation:

The rules for concatenation are as follows:

- a. Within the outer level of angle bracket nesting one apostrophe is removed from each string of apostrophes. Thus, if a single apostrophe is encountered, it is removed; if a pair are encountered, one is removed and one left, etc.
- b. Within nested brackets, all single apostrophes are passed on to the macro processor.

Outside of macro definitions, single apostrophes are ignored except when in text strings. Therefore, MO"VEI is the equivalent of MOVEI. In any event, apostrophes will appear on the listing.

APPENDIX H
OPERATING INSTRUCTIONS

Requirements

Monitor

Minimum Core: 6K

Additional Core: Automatically requests additional core assignments from the time-sharing monitor as needed

Equipment: One input device (source program input); two output devices (machine language program output and listing output). If the listing output is to be used as input to the Cross Reference (CREF) program, it must be written on either DECtape, magnetic tape, or disk.

Initialization

`_R MACRO)`

Loads the Macro-10 Assembler into core.

`*`

The Assembler is ready to receive a command.

Commands

General Command Format

```
objprog-dev:filename.ext,list-dev:filename.ext←source-dev:filename.ext,
.....source-n )
```

objprog-dev:	The device on which the object program is to be written.			
	MTAn: (magnetic tape) DTAn: (DECtape) PTP: (paper tape punch) DSK: (disk)			
list-dev:	The device on which the assembly listing is to be written.			
	<table style="border: none;"> <tr> <td style="border: none;"> MTAn: (magnetic tape) DTAn: (DECtape) DSK: (disk) LPT: (Line printer) TTY: (Teletype) PTP: (paper tape punch) </td> <td style="border: none; vertical-align: middle; padding-left: 10px;"> } </td> <td style="border: none; vertical-align: middle; padding-left: 10px;"> Must be one of these if input to CREF.* </td> </tr> </table>	MTAn: (magnetic tape) DTAn: (DECtape) DSK: (disk) LPT: (Line printer) TTY: (Teletype) PTP: (paper tape punch)	}	Must be one of these if input to CREF.*
MTAn: (magnetic tape) DTAn: (DECtape) DSK: (disk) LPT: (Line printer) TTY: (Teletype) PTP: (paper tape punch)	}	Must be one of these if input to CREF.*		
source-dev:	The device(s) from which the source-program input to assembly is to be read.			
	MTAn: (magnetic tape) CDR: (card reader) DTAn: (DECtape) DSK: (disk) PTR: (paper tape reader) TTY: (Teletype)			

If more than one file is to be assembled from a magnetic tape, card reader, or paper tape reader, dev: is followed by a comma for each file beyond the first.

Input via the Teletype is terminated by typing CTRL Z (↑Z) to enter pass 1; the entries must be retyped at the beginning of pass 2.

filename.ext (DSK: and DTAn: only)

The filename and filename extension of the object program file, the listing file, and the source file(s).

←

The object program and listing devices are separated from the source device by the left arrow symbol.

*If /C switch is given, but no list-dev: is specified, DSK:CREF.TMP is assumed.

Disk File Command Format

DSK:filename.ext [proj,prog]

[proj,prog]

Project-programmer number assigned to the disk area to be searched for the source file(s) if other than the user's project-programmer number.

The standard protection* is assigned to any disk file specified as output.

NOTE

If object coding output is not desired (as in the case where a program is being scanned for source language errors), objprog-dev: is omitted. If an assembly listing is not desired, list-dev: is omitted.

Examples

```
._R MACRO )
*_DTA3:OBJPRG,LPT:←CDR: )
```

Assemble one source program file from the card reader; write the object code on DTA3 and call the file OBJPRG; write the assembly listing on the line printer.

```
END OF PASS 1 )
```

The source program cards must be manually re-fed for pass 2.

```
[ THERE ARE 2 ERRORS )
PROGRAM BREAK IS 002537 )
5K CORE USED )
```

Number of source errors. Size of object program. Core used by assembler.

```
*←C )
```

Return to the Monitor.

```
._R MACRO )
*_MTA3:,MTA2:←MTA1:,, )
[ THERE ARE NO ERRORS )
PROGRAM BREAK IS 003552 )
6K CORE USED )
```

Assemble the next three source files located at the present position of MTA1; write the object program on MTA3; write the listing on MTA2 for later printing.

```
*,LPT:←DTA1:FILE1,FILE2,FILE5 )
[ THERE ARE NO ERRORS )
PROGRAM BREAK IS 001027 )
6K CORE USED )
```

Assemble the source files named FILE1, FILE2, and FILE5 from DTA1; produce no object coding; write the listing on the line printer.

*Standard protection (055) designates that the owner is permitted to read or write, or change the protection of, the file while others are permitted only to read the file.

```
*->DSK:FILE1.MAC[14,12]
[THERE ARE NO ERRORS)
PROGRAM BREAK IS 000544)
5K CORE USED)
```

Scan the source program called FILE1.MAC, located in area 14, 12 on the disk, for source language errors; produce no object coding or assembly listing; print all error diagnostics on the Teletype.

*+C)

Return to the Monitor.

.R MACRO

*MTA1:,TTY:-TTY:)

Assemble a source file from the Teletype; write the object code program on MTA1 and print the assembly listing on the Teletype.

```
R:   JMP      R)
     AOS      G)
G:   JFCL)
     END)
```

} Enter the source statements

+Z)

Terminate input.

```
END OF PASS 1)
      JMP R)
```

Re-enter Teletype input.

Re-enter the first statement.

<pre>[.MAIN MACRO 10:14 20-DEC-67 PAGE1) 0 000000 000000 000001' JMP R) R: AOS G 000001 350000 000002' R: AOS G) G: JFCL) 000002 255000 000000 G: JFCL) END)</pre>	<p>Page heading.</p> <p>First assembled.</p> <p>Re-enter second.</p> <p>Second assembled.</p> <p>Re-enter third.</p> <p>Third assembled.</p> <p>Re-enter fourth.</p> <p>Fourth assembled.</p>
---	---

```
THERE IS 1 ERROR)
PROGRAM BREAK IS 000003)
```

Typeout of symbol table.

```
.MAIN  MACRO          10:14  20-DEC-67  PAGE 2)
      SYMBOL TABLE)
G      000002')
R      000001')
```

5K CORE USED)

*+C)

Return to the Monitor.

Switches are used to specify such options as:

- a. Magnetic tape control
- b. Macro call expansion
- c. Listing suppression
- d. Pushdown list expansion
- e. Cross-reference file output.

All switches are preceded by a slash (/) (or enclosed in parentheses) and usually occur prior to the left arrow.

Table 3-1
Macro-10 Switch Options

Switch	Meaning
A	Advance magnetic tape reel by one file.
B	Backspace magnetic tape reel by one file.
C	Produce listing file in a format acceptable as input to CREF; unless the file is named, CREF.TMP is assigned as the filename; if no extension is given, .TMP is assigned; if no list-dev: is specified, DSK: is assumed.
E	List macro expansions (same function as LALL pseudo-op).
L	Reinstate listing (used after list suppression by XLIST pseudo-op or S switch).
N	Suppress error printouts on the Teletype.
P	Increase the size of the pushdown list. This switch may appear as many times as desired (pushdown list is initially set to a size of 8010 locations; each /P increases its size by 8010).
Q	Suppress Q (questionable) error indications on the listing; Q messages indicate assumptions made during pass 1.
S	Suppress listing (same function as XLIST pseudo-op).
T	Skip to the logical end of the magnetic tape.
W	Rewind the magnetic tape.
X	Suppress all macro expansions (same function as XALL pseudo-op).
Z	Zero the DECTape directory.
	NOTE Switches A through C and T, W, X, and Z must immediately follow the device or file to which the individual switch refers.

Examples

```

.R MACRO)
*MTA1:,DTA3:/C+PTR:)

```

Assemble one source file from the paper tape reader; write the object code on MTA1; write the assembly listing on DTA3 in cross-reference format and call the file CREF.TMP.

```

END OF PASS 1)

```

The paper tape must be refeed by the operator for pass 2.

```

[THERE ARE 3 ERRORS)
PROGRAM BREAK IS 000401)
5K CORE USED)

```

End-of-assembly messages.

```

*DTA2:ASSEMB.ONE/Z,LPT:
-MTA4:/W,)

```

Rewind MTA4 and assemble the first two source files on it; write the object code on DTA2, after zeroing the directory, and call the file ASSEMB.ONE; write the assembly listing on the line printer.

```

[THERE ARE NO ERRORS)
PROGRAM BREAK IS 005231)
6K CORE USED)

```

```

*MTA1:/W,LPT:-MTA3:
/W,(AA),(BB) )

```

Rewind MTA1 and MTA3 and assemble files 1, 4, and 3 (in that order) from MTA3. Print the assembly listing on the line printer. Write the object code on MTA1.

```

[THERE IS 1 ERROR)
PROGRAM BREAK IS 000655)
5K CORE USED)

```

Return to the Monitor.

```

*+C)

```

```

.
```

Diagnostic Messages

Table 3-2
Macro-10 Diagnostic Messages

Message	Meaning
?CANNOT ENTER FILE filename.ext	DTA or DSK directory is full; file cannot be entered.
?CANNOT FIND filename.ext	The file cannot be found on the device specified.
?COMMAND ERROR	The last command string is in error.
?DATA ERROR ON DEVICE dev:	Output error has occurred on the device.

Table 3-2 (Cont)
Macro-10 Diagnostic Messages

Message	Meaning
END OF PASS1	This message is issued prior to pass 2 whenever the input source file is on a medium which must be manually re-entered by the operator (PTR:, CDR:, TTY:). When this message appears, the operator must re-feed the tape or cards or retype the entries.
?IMPROPER INPUT DATA	The input data is not in the proper format.
?INPUT ERROR ON DEVICE dev:	Data cannot be read.
?INSUFFICIENT CORE	An insufficient amount of core is available for assembly.
nK CORE USED	Amount of core used for this assembly.
LOAD THE NEXT FILE	Manual loading is required for the next card or paper tape file.
?NO END STATEMENT ENCOUNTERED ON INPUT FILE	The END statement is missing at the end of the source program file.
?dev: NOT AVAILABLE	The device is assigned to another user or does not exist.
?PDP OVERFLOW, TRY/P	A pushdown list overflow has occurred.
PROGRAM BREAK IS nnnnn	The highest relative location occupied by the object program produced.
?THERE ARE n ERRORS THERE ARE NO ERRORS ? THERE IS 1 ERROR	Number of source language errors found.

Error Detection

MACRO-10 makes many error checks as it processes source language statements. If an apparent error is detected, the assembler prints a single letter code in the left-hand margin of the program listing, on the same line as the statement in question.

The programmer should examine each error indication to determine whether or not correction is required. At the end of the listing, the assembler prints a total of errors found; this is printed even if no listing is requested.

Each error code indicates a general class of errors. These errors, however, are all caused by illegal usage of the MACRO-10 language.

Table 3-3
Macro-10 Error Codes

Error Code	Meaning	Explanation
A	Argument error in pseudo-op	This is a broad class of errors which may be caused by an improper argument in a pseudo-op.
D	Multiply-defined symbolic reference error	This statement contains a tag which refers to a multiply-defined symbol. It is assembled with the first value defined.
E	External symbol error	Improper usage of an external symbol. Example: EXT: EXTERN TXT, BRT, EXT EXT cannot be both an external and internal symbol.
L	Literal error	A literal is improper. A literal must generate 1 to 18 words. Example: EXP [SIXBIT //]; no code generated.
M	Multiply-defined symbol	A symbol is defined more than once. The symbol retains its first definition, and the error message M is typed out during pass 1. If this type of error occurs during pass 2, it is a phase error (see below). If a symbol is first defined as a #-sign suffixed tag, and later as a label, it retains the label definition. Examples: A: ADD 3,X; A: MOVE ,C; M error A: ADD3,X#; X: MOVE ,C; X is assigned the current value of the location counter. Multiple appearances of the TITLE pseudo-op (which generates both a title line and program name) are flagged as "M" (Multiple definition) errors.
N	Number error	A number is improperly entered. Examples: ↑F13.33E38 (Exceeds range) ↑D15BZ (Number must follow B shift operator.) But, ↑D15B<Z> is illegal if Z is defined. If a number contains meaningless letters or special characters, a Q error is given.
O	Operation code undefined	The operation field of this statement is undefined. It is assembled with a numeric code of 0.

Table 3-3 (Cont)
Macro-10 Error Codes

Error Code	Meaning	Explanation
P	Phase error	A symbol is assigned a value as a label during pass 2 different from that which it received during pass 1. In general, the assembler should generate the same number of program locations in pass 1 and pass 2, and any discrepancy causes a phase error. For example, if an assembly conditional, IF1, generates three instructions, a phase error results unless another conditional, such as IF2, generates three program locations during pass 2.
Q	Questionable	This is a broad class of possible errors in which the assembler finds ambiguous language. Example: ADD ,TOTAL SUM; SUM is not needed and is treated as a comment.
R	Relocation error	LOC or RELOC are used improperly. Example: LOCA; where A is not defined.
S	Symbol format error	Usually caused by inclusion of illegal special characters. Example: SY?M: ADD 3,X;
U	Undefined symbol	A symbol is undefined.
V	Value previously undefined	A symbol used to control the assembler is undefined prior to the point at which it is first used. Causes error message in pass 1.

Monitor Commands

Assembly of Macro source program files can be performed by use of the COMPILER, LOAD, EXECUTE, and DEBUG commands. See Table 9-1, Time-Sharing Monitor Commands, in Chapter 9 of this manual for details.

Book 3

**Communicating
with the
Monitor**

Time-Sharing Monitors

FOREWORD

This manual covers the use of the Time Sharing Monitors, which include the Multiprogramming non-disk Monitor and the Multiprogramming disk Monitor (formerly known as 10/40) and the Swapping Monitor (formerly known as 10/50).

The Single-User Monitor (formerly known as 10/20, 10/30) is covered in the manual Single User Monitor Systems.

CONTENTS

	Page
CHAPTER 1 INTRODUCTION-MONITOR CAPABILITIES	
1.1 Reentrant User-Programming Capability	1-2
1.2 Monitor Functions	1-4
1.2.1 Job Scheduling	1-4
1.2.2 Use of Swapping Space and Physical Core	1-7
1.3 User Facilities	1-8
1.4 Segments	1-11
1.5 Files	1-12
1.6 Comparison of Segments and Files	1-13
CHAPTER 2 MONITOR COMMANDS	
2.1 Console and Job Control	2-1
2.1.1 Monitor Mode and User Mode	2-2
2.2 Command Interpreter and Command Format	2-3
2.2.1 Command Names	2-3
2.2.2 Arguments	2-3
2.2.3 Login Check	2-4
2.2.4 Job Number Check	2-4
2.2.5 Core Storage Check	2-4
2.2.6 Delayed Command Execution	2-5
2.2.7 Completion-of-Command Signal	2-5
2.3 System Access Control Commands	2-5
2.4 Facility Allocation Commands	2-7
2.5 Source File Preparation Commands	2-11
2.6 File Manipulation Commands	2-15
2.6.1 Extended Command Forms	2-17
2.6.2 Compile Switches	2-21
2.6.3 Processor Switches	2-25

CONTENTS (Cont)

	Page
2.6.4 Loader Switches	2-26
2.6.5 Temporary Files	2-27
2.7 Run Control Commands	2-29
2.7.1 Additional Information on SAVE and SSAVE	2-33
2.8 Background Job Control Commands	2-36
2.9 Job Termination Commands	2-37
2.10 System Timing Commands	2-38
2.11 System Administration Commands	2-39
2.12 Monitor Diagnostic Messages	2-41
 CHAPTER 3 LOADING USER PROGRAMS	
3.1 Memory Protection and Relocation	3-1
3.2 User's Core Storage	3-3
3.2.1 Job Data Area	3-4
3.2.2 Loading Relocatable Binary Files	3-8
 CHAPTER 4 USER PROGRAMMING	
4.1 User Mode	4-1
4.2 Programmed Operators (UUO's)	4-2
4.2.1 Operation Codes 001-034	4-2
4.2.2 Operation Codes 040-077, and 000	4-3
4.2.3 Operation Codes 100-127	4-5
4.2.4 Illegal Operation Codes	4-5
4.3 Program Control	4-5
4.3.1 Starting	4-5
4.3.2 Stopping	4-5
4.3.3 Trapping	4-11
4.3.4 Timing Control	4-13

CONTENTS (Cont)

	Page
4.3.5 Identification	4-14
4.3.6 Direct User I/O	4-19
4.3.7 Segment Handling	4-21
4.4 Input/Output Programming	4-28
4.4.1 File	4-28
4.4.2 Initialization	4-35
4.4.3 Data Transmission	4-48
4.4.4 Status Checking and Setting	4-52
4.4.5 Terminating a File (CLOSE)	4-54
4.4.6 Synchronization of Buffered I/O	4-55
4.4.7 Relinquishing A Device (RELEASE)	4-55
4.5 Core Control	4-56
4.5.1 CALL AC, [SIXBIT/CORE/]	4-56
4.5.2 CALL AC, [SIXBIT/SETUWP/]	4-58
 CHAPTER 5 DEVICE DEPENDENT FUNCTIONS	
5.1 Teletype	5-2
5.1.1 Data Modes	5-4
5.1.2 DDT Submode	5-6
5.1.3 Special Programmed Operator Service	5-7
5.1.4 Special Status Bits	5-11
5.1.5 Paper Tape Input from the Teletype	5-11
5.2 Paper Tape Reader	5-12
5.2.1 Data Modes	5-12
5.3 Paper Tape Punch	5-13
5.3.1 Data Modes	5-13
5.3.2 Special Programmed Operator Service	5-14
5.4 Line Printer	5-14

CONTENTS (Cont)

	Page
5.4.1 Data Modes	5-14
5.4.2 Special Programmed Operator Service	5-15
5.5 Card Reader	5-15
5.5.1 Data Modes	5-15
5.6 Card Punch	5-16
5.6.1 Data Modes	5-16
5.6.2 Special Programmed Operator Service	5-18
5.7 DEctape	5-19
5.7.1 Data Modes	5-19
5.7.2 DEctape Block Format	5-20
5.7.3 DEctape Directory Format	5-20
5.7.4 DEctape File Format	5-22
5.7.5 Special Programmed Operator Service	5-22
5.7.6 Special Status Bits	5-26
5.7.7 Important Considerations	5-26
5.8 Magnetic Tape	5-27
5.8.1 Data Modes	5-27
5.8.2 Magnetic Tape Format	5-28
5.8.3 Special Programmed Operator Service	5-29
5.8.4 9-Channel Magtape	5-32
5.8.5 Special Status Bits	5-34
5.9 Disk	5-35
5.9.1 Data Modes	5-35
5.9.2 Structure of Files on Disk	5-36
5.9.3 User Programming for the Disk	5-42
5.10 Incremental Plotter	5-48
5.10.1 Data Modes	5-48
5.11 Display with Light Pen	5-49

CONTENTS (Cont)

	Page
5.11.1 Data Modes	5-49
5.11.2 Background	5-50
5.11.3 Display UO's	5-50
5.12 CALL AC [SIXBIT/DEVCHR/]or CALLI AC, 4	5-52
APPENDIX 1 DECTape Compatibility Between DEC Computers	A1-1
APPENDIX 2 Size of Multiprogramming Non-disk Monitor	A2-1
APPENDIX 3 Size of Swapping Monitor	A3-1
APPENDIX 4 Writing Reentrant User Programs	A4-1

LIST OF ILLUSTRATIONS

1-1 Core Management	1-3
3-1 User's Core Area	3-3
3-2 Loading User Core Area	3-8
4-1 User's Ring of Buffers	4-33
4-2 Detailed Diagram of Individual Buffer	4-34
4-3 File Protection Key	4-44

LIST OF TABLES

2-1 Monitor Command to Gain Access to the System	2-6
2-2 Monitor Commands to Allocate Facilities	2-8
2-3 Monitor Commands to Prepare Source Files	2-13
2-4 Monitor Command Diagnostic Messages	2-13

LIST OF TABLES (Cont)

	Page	
2-5	Monitor Commands to Manipulate Files	2-15
2-6	Monitor Commands to Call, Load, and Control Programs	2-30
2-7	Monitor Commands to Control Background Jobs	2-36
2-8	Monitor Command to Terminate Jobs	2-37
2-9	Monitor Commands for System Timing	2-38
2-10	Monitor Commands for System Administration	2-40
2-11	Time-Sharing Monitor Diagnostic Messages	2-41
3-1	Job Data Area Locations,	3-4
4-1	Monitor Operation Codes	4-7
4-2	CALL and CALLI Monitor Operations	4-8
4-3	Buffered Data Modes	4-30
4-4	Unbuffered Data Modes	4-30
4-5	File Status	4-35
5-1	Device Summary	5-1
5-2	PDP-10 Card Codes	5-17
5-3	DECTape Programmed Operators	5-23
5-4	MTAPE Functions	5-30
5-5	Magnetic Tape Special Status Bits	5-35

CHAPTER 1
INTRODUCTION - MONITOR CAPABILITIES

This book discusses the commands, program loading procedures, and user programming facilities available under the PDP-10 Time-Sharing Monitors - three multiprogramming, time-sharing systems designed to allow many independent user programs to share the facilities of a single PDP-10 computer. Many users can access the computer at the same time from consoles located at the computer site, at nearby offices or laboratories, or at remote points connected by telephone lines.

Operating concurrently under Monitor control, users may access available I/O devices and system software to compile, assemble, and execute their programs, or may have this sequence performed automatically for many jobs by using the batch control processor (BATCH). Real-time jobs can operate either as independent user programs or as fully integrated Monitor subroutines.

The Multiprogramming non-disk Monitor (formerly called the 10/40 Monitor) is a multiprogramming, time-sharing system which includes I/O control of all devices attached to the system, run-time selection of I/O devices, job-to-job transition, job save and restore features, and dynamic debugging facilities. All of these features are incorporated with concurrent real-time processing, batch processing, and time sharing. The Multiprogramming disk Monitor adds a comprehensive file system with both sequential and random access of shared, named files to the Multiprogramming non-disk system. The Swapping Monitor (formerly called the 10/50 Monitor) has all the features of the Multiprogramming disk system and, in addition, swaps programs between

high-speed disk and core, thereby increasing the number of users that can be accommodated simultaneously.

1.1 Reentrant User-Programming Capability

The number of users that can be handled by a given size time-sharing configuration is further increased by adding a reentrant user-programming capability to the system. This means that a sequence of instructions may be entered by more than one user process at a time. A single copy of a reentrant program may be shared by a number of users at the same time, thereby increasing system economy. All the versions of the Time-sharing Monitor normally include this reentrant capability but it may be deleted on systems lacking the dual relocation KT10A hardware option.

In a non-reentrant system, the one relocation register hardware requires that a user area be a single continuous segment of logical and physical core. Each user has a separate copy of a program even though a large part of it is the same as for other users. In a reentrant system, the two relocation register hardware allows a user area to be divided into two logical segments which may occupy non-contiguous areas in physical core. The Monitor allows one of the segments of each user area to be the same as one or more other users, so that only one physical copy of a shared segment need exist no matter how many users are using it. The Monitor normally invokes hardware write-protection for shared segments to guarantee that they are not accidentally modified.

In the PDP-10 Swapping Monitor, the reentrant capability causes the following system resources to be used more efficiently:

a) core memory, since only one copy of a shared segment exists for the entire system (Figure 1-1 illustrates this efficient

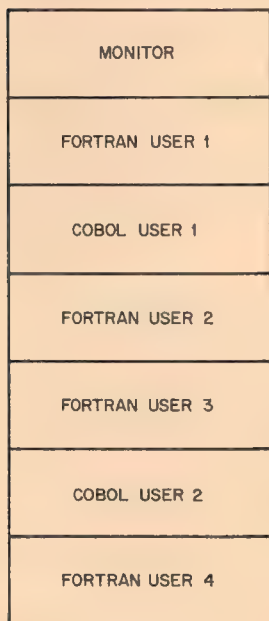
use of core memory),

b) swapping storage, since many users share the single copy of the shared segment kept in swapping storage,

c) swapping I/O channel, since a shared segment is read into core only once and is not written back onto swapping storage unless modified, and

d) file storage I/O channel, since a shared segment exists on the faster swapping storage after it has been read into core the first time from the storage device instead of being retrieved from file storage on each usage as necessary in the non-reentrant system.

NON-REENTRANT SYSTEM



REENTRANT SYSTEM

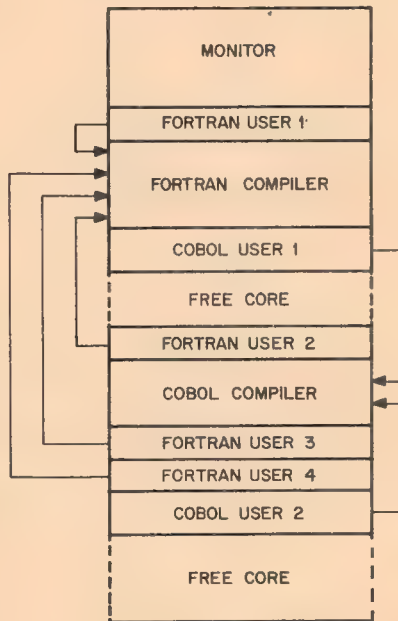


Figure 1-1

Core Management

1.2 MONITOR FUNCTIONS

The Time-Sharing Monitors act as the interface between the user and the computer so that all users are protected from one another and appear to have most resources available to themselves. The Monitors schedule multiple-user time sharing of the system, allocate available sharable resources to user programs, accept input from and direct output to all system I/O devices, and relocate and protect user programs in core memory.

The Monitors utilize the PDP-10 hardware features for memory protection, memory relocation, executive/user mode, and real-time clock to provide an advanced, third-generation, multi-programming time-sharing environment. System facilities start with a minimum configuration of 16K core and two DECTapes and can accommodate magnetic tapes, disks, drums, communication line controllers, card readers and punches, paper tape readers and punches, line printers, displays, incremental plotters, and user Teletype consoles. Other special devices, including real-time digitizers and analog converters, easily interface with the system.

Several user programs are loaded into core at once and the Time-Sharing Monitors schedule each program to run for a certain length of time, utilizing a scheduling algorithm that makes efficient use of system capabilities. The Monitors direct data flow between I/O devices and the user programs, making them device independent, and overlap I/O operations concurrently with computation for high system efficiency.

1.2.1 Job Scheduling

One of the parameters which must be specified in

creating a PDP Time-Sharing Monitor is the number of jobs which may be run simultaneously. Up to 127 jobs may be specified. Each user who accesses the system is assigned a job number. The term job is used to refer to the entire sequence of operations the user initiates from his console.

In a multiprogramming system all jobs reside in core, and the scheduler decides which of these jobs should run. In a swapping system jobs can exist on an external storage device (usually disk) as well as in core. The scheduler decides not only which job is to run but also when a job is to be swapped out onto the disk or brought back into core.

In the Swapping Monitor, jobs are retained in queues of varying priorities that reflect the status of the jobs at any given moment. Each job number possible in the system resides in only one queue at any point in time. The possible queues a job may be in include the following.

a) Run queues - for runnable jobs waiting to execute.
(There are three run queues of different levels of priorities.)

b) I/O wait queue - for jobs waiting while doing I/O.

c) I/O wait satisfied queue - for jobs waiting to run after finishing I/O.

d) Sharable device wait queue - for jobs waiting to use sharable devices.

e) Teletype wait queue - for jobs waiting for input or output on the user's console.

f) Teletype wait satisfied queue - for jobs that completed a Teletype operation and are awaiting action.

g) Stop queue - for processes that have been completed or aborted by an error and are awaiting a new command for further action.

h) Null queue - for all job numbers that are inactive (unassigned).

Each of these queues is addressed through tables.

The position of a queue's address in a table represents the priority of the queue with respect to the other queues. Within each queue, the position of a job determines its priority with respect to the other jobs in the same queue. The status of a job is changed when it is placed in a different queue.

Each job, when it is assigned to run, is given a quantum time. When this time expires, the job ceases to run and moves to a lower priority run queue. The activities of the job currently running may cause it to move out of the run queue and enter one of the wait queues. For example, when a currently running job begins input from a DECTape, it is placed in the I/O wait queue, and the input is begun. A second job is set to run while the first job's input proceeds. If the second job then decides to access a DECTape for an I/O operation, it is stopped because the DECTape control is busy, and it is put in the queue for jobs waiting to access the DECTape control. A third job is set to run. Now the input operation of the first job finishes, making the DECTape control available to the second job. The second job's I/O operation is initiated, and the job is transferred from the device wait queue to the I/O wait queue. The first job is transferred from the I/O wait queue to the highest priority run queue. This permits the first job to preempt the third job's running. When the quantum time of the first job becomes zero, it is moved into the second run queue, and the third job runs again until the second job completes its I/O operations.

Scheduling occurs at each clock tick (1/60th or 1/50th of a second) or may be forced at Monitor level between clock ticks

if the current job becomes unrunnable. The asynchronous swapping algorithm is also called at each clock tick and has the task of bringing a job from disk into core. This function is dependent upon (1) the core shuffling routine, which consolidates unused areas in core so as to make sufficient room for the incoming job, and upon (2) the swapper, which creates additional room in core by transferring jobs from core to disk. Therefore, when the scheduler is selecting the next job to be run, the swapper is bringing the job to be run after that into core. The transfer from disk to core takes place while the central processor continues computation for the previous job.

1.2.2 Use of Swapping Space and Physical Core

The reentrant capability reduces the demands on core memory, swapping storage, swapping channel, and storage channel. However, to reduce the use of the storage channel, copies of the sharable segments are kept on the swapping device. This increases the demand for swapping storage. The Monitor achieves this space-time balance dynamically by assuming that there is no shortage of swapping space. The amount of swapping space is fixed by the operator at system initialization. Thereafter, the Monitor keeps a single copy of as many sharable segments as possible in the swapping space. (The maximum number of segments that may be kept may be increased by individual installations but is always at least as great as the number of jobs plus one.) If a sharable segment is currently unused, it is called a dormant segment. If the Monitor cannot find contiguous free space on the swapping device, it will fragment the high and low segments of the user whose job is being swapped out. If swapping space runs out, the Monitor deletes a dormant segment and continues to fragment

the user's segments. If and when a deleted segment is needed again, it is retrieved from the storage device.

The Monitor keeps track of the total amount of "virtual core" assigned to all users. In computing virtual core, sharable segments count only once and dormant segments do not count at all. The Monitor does not allow more virtual core to be granted than the system has capacity to handle. When the Monitor is started the amount of unused virtual core is set equal to the amount of swapping space pre-allocated on the disk. Thus, there is always room to swap out the largest possible job in core and swap in another job.

The same techniques used in allocating swapping space are used to allocate core in both swapping and non-swapping systems. A dormant segment will stay in core until core is needed. In the swapping system, an active write-protected segment remains in core even though no one in core is using it. Some swapped-out user must be using it or else it would be dormant rather than idle.

1.3 USER FACILITIES

Users gain access to the PDP-10 Time-Sharing system from a terminal located either at the computer facility or at a spot remote from the facility but connected to it by telephone. Three levels of communication are available at the console:

- a) Monitor command level
- b) CUSP command level
- c) CUSP I/O level.

At Monitor command level, the console communicates with the Monitor Command Interpreter. The Monitor Command Interpreter

- a) provides the system with access protection,

- b) allocates and protects memory and peripherals requested by the user,
- c) provides communication with the operator for mounting of special tapes,
- d) provides run control for the user over programs stored in the system,
- e) allows the user to initiate background jobs,
- f) provides the user with job monitoring and debugging facilities, and
- g) returns facilities to the system when the job is finished using them.

Chapter 2 describes the various Monitor commands which provide each of these capabilities.

Using Monitor commands, the user at his console can call in programs from the system file. The system file contains programs for creating and editing program source files (TECO,EDITOR), for assembling or compiling program source files (MACRO,FORTRAN,BASIC, COBOL), and for loading relocatable binary files (LOADER). The usage of these and many other CUSPs (Commonly Used Systems Programs) are currently described in the System User's Guide (DEC-10-NGCC-D).

The user's console provides both a control and data path to any CUSP or other user program that the user initiates via Monitor commands. Once a particular CUSP has been called in, the user's console is at CUSP command level and the user can issue a command to the CUSP. In processing that command, the CUSP may access the user's console directly as an input or output device. This is illustrated by the following example.

*R PIP	Monitor command level. User calls CUSP named PIP, Peripheral Interchange Program.
*DSK:TEXT+TTY:	CUSP command level. User instructs PIP to create a file on the disk named TEXT using Teletype console as input medium.
THIS IS FILE TEXT	CUSP I/O level. User types input to PIP.
↑Z	↑Z causes Teletype end of file. Return to CUSP command level.
*↑C	↑C is a special character that causes return to Monitor command level.
.	The period (.) signifies return to Monitor command level.

The console is switched back to the Monitor Command Interpreter by either the program or the user. The user can exercise another dimension of control over his program by loading it with the powerful Dynamic Debugging Technique (DDT) available in the system file. Entry to DDT is through the Monitor Command Interpreter or by breakpoints in the program. While DDT is in control of the program, the user can examine intermediate results on his console and then modify his program accordingly.

The user's program communicates with the Monitor by means of PDP-10 operation codes 040 through 077. These op-codes are called UO's and are described in detail in Chapter 4. With these operation codes, the Monitor provides the program with complete device-independent I/O services. The programmer is relieved of the job of I/O programming and is freed from the dependence on the availability of particular devices at run time. In addition, the user's program may exercise control over central processor trapping, modify its memory allocation, and monitor its own running time. Provisions exist for inter-job communication and control, reentrant user programs, and, in selected cases, direct user I/O control.

1.4 SEGMENTS

A *segment* is a continuous region of the user's core area that the Monitor maintains as a continuous unit in physical core and/or as a possibly fragmented unit on the swapping device. A program or user job is composed of one or two segments. A segment may contain instructions and/or data. The Monitor determines the allocation and movement of segments in core and on the swapping device.

A *sharable segment* is a segment which is the same for many users. The Monitor keeps only one copy in core and/or on the swapping device, no matter how many users are using it. A *non-sharable segment* is a segment which is different for each user in core and/or on the swapping device.

The PDP-10's two relocation and protection registers, which divide a user's core area into two parts, permit a user program to be composed of one or two segments at any point in time. The required low segment starts at user location 0. The optional high segment starts at user location 400000 or at the end of the low segment, whichever address is greater. The low segment contains the user's accumulators, Job Data area, instructions and/or data, I/O buffers, and DDT symbols. A user's core image is composed of a low segment, which may have from 1K to 256K words, in multiples of 1K (1K = 1024_{10} words), and a high segment which may have from 0K to 128K words, also in multiples of 1K. A high segment may be sharable or non-sharable, whereas a low segment is always non-sharable. The high segment may be write-protected.

A *reentrant program* is always composed of two segments - a low segment which usually contains just data, and a high (sharable) segment which usually contains instructions and

constants. The low segment is sometimes referred to as the impure segment. The sharable high segment, if write-protected, is referred to as the pure segment.

A *one-segment non-reentrant program* is composed of a single low segment containing instructions and data. User programs written for machines with only a single relocation and protection register are always one-segment non-reentrant programs.

A *two-segment non-reentrant program* is composed of a low segment and a non-sharable high segment. This kind of program is useful when there is a requirement for two fixed-origin data areas to increase and decrease independently during execution.

1.5 FILES

A *file* is a collection of 36-bit words comprising computer instructions and/or data. A file can be of arbitrary length, limited only by the available space on the device and the user's maximum allotment of space on that device.

A *named file* is uniquely identified in the system by its filename (up to six characters in length) and extension (up to three characters in length) and by its directory name (owner's project-programmer numbers for disk, physical device name for DEctape) in which the filename and extension appear. The filename, being arbitrary, is specified by the owner, whereas the extension, usually one of a small number of standard names which identify the type of information in the file, is usually specified by the program. A named file may be written by a user program in buffered or unbuffered mode, or in both. It may be read and/or modified sequentially or randomly with buffered or unbuffered mode I/O independently of how it was written. Named files are stored on the storage device. Each named file has certain access

privileges associated with it. These privileges designate which users can read or write the file or change its access privileges. In regard to a given file, users are divided into three groups: the owner of the file, the users in his project, and the rest of the users.

A file is said to be *created* if no file by the same name existed when the file was opened for writing. A file is said to be *superseded* if another file by the same name already exists. A file is said to be *updated* when one or more blocks of the file are rewritten in place. Other users may read a disk file while a certain user is superseding it. The older version of the file is deleted only when all the readers have finished with it. Only one user may open a file for updating at a time; all other users attempting to open that file receive an error message.

1.6 COMPARISON OF SEGMENTS AND FILES

Files and segments have certain similarities and differences. Both are named, one-dimensional arrays of 36-bit words. A file can be as long as the size of disk or DECTape. A segment can be only as big as physical core. Both may be shared for reading, but only one user may supersede or update a file at a time, whereas many users share a segment for writing. When many users share the same file, each user is given his own copy of the portion of the file that he is reading. It is read into his low segment by the INPUT UWO. When many users share the same segment, each user does not have his own copy of the segment. A file exists on the storage device and portions of it may exist in different parts of the low segment of one or more users. A segment never exists on the storage device; it exists as a continuous unit only in core or on the swapping device.

2.1 CONSOLE AND JOB CONTROL

The PDP-10 time-sharing system is a multiprogramming system. This means that control is transferred rapidly among a number of programs or processes in such a way that all the processes appear to be running simultaneously. Each process is called a job. In configuring and loading a time-sharing Monitor, the system administrator sets the maximum number of jobs which his system will handle simultaneously. This number may be up to 127 jobs if the system has enough core, disk storage, processor capacity, and time-sharing consoles to handle this load.

Jobs are initiated by users typing on a time-sharing console. A console is typically any of several models of Teletype machines but may also be a CRT (cathode ray tube) with a keyboard. The console may be directly connected to the computer or may be remotely connected via a private wire or the public telephone system.

There is not necessarily a one-to-one relationship between jobs and consoles. A console must initiate a job, but the DETACH and ATTACH commands (see Table 2.7) permit a job to "float" in a state where it is not associated with a particular console. Therefore a user may control several jobs from the same console. Each job is either in the ATTACHed or DETACHed mode depending on whether a console is currently associated with that job. At any point in time, each console is attached to at most one job. The console is often referred to as being in a "detached mode," but this results from a semantic confusion. It is really

meant that the job initiated from that console is in a detached mode. By typing an appropriate command, the job may be attached to the same console or to any other console in the system.

2.1.1.1 Monitor Mode and User Mode

From the user's point of view, his console is in one of two states - monitor mode or user mode. In monitor mode, each line the user types in is sent to the Monitor Command Interpreter. The execution of certain commands (as noted in the tables below) places the console in user mode. Once the program is in user mode, the console becomes simply an input/output device for that user. In addition, user programs will use the console for two purposes. The user program will accept command strings from the console or will use the console as a direct input/output device.

Example:

monitor mode	.R PIP	monitor command
user mode	*DSK:FOO←TTY:	user program command string
user mode	THIS IS FILE FOO↑Z	user program using console as an input device
monitor mode	.R MACRO	monitor command
user mode	*TTY: ,←DSK:PROGL	user program command string
user mode		user program using console as an output device
	•	
	•	
	code	
	•	
	•	

The special character ↑C (produced by typing C with the CONTROL Key depressed) is used to stop a user program and return the console to monitor mode. There are certain commands which

cause the user program to start or continue running (as noted in the tables below) but which leave the console in monitor mode.

When the system is started, each console is in monitor mode ready for users to begin typing in commands. However, if the system becomes fully loaded (i.e., all the jobs that the system can accommodate have been initiated), then any unused consoles enter a special state where any command typed in will receive either the message "JOB CAPACITY EXCEEDED" or "X."

2.2 COMMAND INTERPRETER AND COMMAND FORMAT

Each command is a line of ASCII characters in upper and/or lower case. Spaces and non-printing characters preceding the command name are ignored. The Monitor Command Interpreter will not interpret or execute a line of comments preceded by a semicolon. Every command to the Monitor Command Interpreter must be terminated by pressing the RETURN key on the console. If the command is not understood, an error message is typed out by the Monitor and the mode is unchanged.

2.2.1 Command Names

Command names are strings from one to six letters. Characters after the sixth are ignored. Only enough characters to uniquely identify the command need be typed. In the tables which follow, the commonly used abbreviation of the command name is shown. Installations which choose to implement additional commands should take care to preserve the uniqueness of the first few letters of existing commands.

2.2.2 Arguments

Arguments follow the command name, separated from it by

a space or any printing character that is not a letter or a numeral. Argument formats are described under the associated commands.

If the Monitor Command Interpreter recognizes the command name, but a necessary argument is missing, the Monitor responds with

TOO FEW ARGUMENTS

Extra arguments are ignored.

2.2.3 Login Check (Disk Monitor Systems)

If a user who has not logged in (see Table 2.1) types a command requiring him to be logged in, the disk Monitor systems will respond with

LOGIN PLEASE

and the user's command will not be executed. Login is not required by a non-disk Monitor system.

2.2.4 Job Number Check (Non-disk Monitor Systems)

If the non-disk Monitor system recognizes a command name which requires a job number and no job number has been assigned, the Monitor assigns a job number, n, and responds with

JOB n

and a line identifying the Monitor version. The Monitor will then proceed to execute the command.

2.2.5 Core Storage Check

If the Monitor Command Interpreter recognizes a command name which requires core storage to have been allocated to the job and the job has no core, the Monitor responds with

NO CORE ASSIGNED

The user's command is not executed.

2.2.6 Delayed Command Execution

If the Monitor Command Interpreter recognizes a command that requires all devices to be inactive and the job has devices actively transmitting data to or from its core area, the execution of the command will be delayed until the devices are inactive. A command is also delayed if a job is swapped out to the disk and the command requires core residence. It will be executed when the job is returned to core.

2.2.7 Completion-of-Command Signal

Most commands are processed without delay. The completion of each command is signaled by the output of a carriage return, line feed. If the console is left in Monitor mode, a period follows the carriage return, line feed. If the console is left in user mode, any response other than the carriage return, line feed comes from the user's program. For example, all standard DEC CUSPS immediately send an asterisk (*) to the user's console to indicate their readiness to accept user-mode command strings.

2.3 SYSTEM ACCESS CONTROL COMMANDS

Access to the system is limited to authorized personnel. The system administrator provides each authorized user with a project number, a programmer number, and a password. The project and programmer numbers are octal numbers up to six digits each. The project-programmer numbers will identify not only the user but also his file storage area on the disk. The password is from one to five ASCII characters. To LOGIN successfully the project-

programmer numbers and the password typed in by the user must match the project-programmer numbers and password stored in the system accounting file (ACCT.SYS [1,1]).

Table 2-1

Monitor Command to Gain Access to the System

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
LOGIN	LOG	<p>LOGIN initializes a Monitor routine to accept the user's LOGIN data.</p> <p>The following is the procedure used to gain access to the system.</p> <p>.LOGIN</p> <p>JOB n PDP-10 4S.50F</p> <p>Job number assigned to user, followed by Monitor name and version number.</p> <p>#</p> <p>System types out number sign to indicate user should type his project-programmer number.</p> <p>proj,prog</p> <p>User types in his project-programmer number</p> <p>PASSWORD:</p> <p>System requests user to type his password. User types password followed by carriage return. To maintain password security, the Monitor will not echo the password.</p> <p>1135 8-AUG-69 TTY23</p> <p>↑C</p> <p>.</p> <p>If user entries are correct, Monitor responds with time, date, TTY number, ↑C and a period, indicating readiness to accept a command.</p>	u R D	<p>LOGIN PLEASE ?</p> <p>The user has typed a command that the Monitor cannot accept unless the user logs in.</p> <p>?INVALID ENTRY - TRY AGAIN</p> <p>An illegal project-programmer number was entered or the password did not match.</p> <p>?1+1/nK CORE VIR. CORE LEFT=0</p> <p>System core and swapping space exceeded.</p>
<p>*Characteristics:</p> <p>d = places job in detached mode L = LOGIN required (Disk Monitor)</p> <p>m = places job in Monitor mode A = no active device</p> <p>u = places job in user mode C = core required</p> <p>J = requires a job number.</p> <p>R = runs a CUSP thereby replacing previous program in user's addressing space.</p> <p>D = available only in Multiprogramming Disk and in swapping systems, not in Multiprogramming non-disk systems.</p>				

2.4 FACILITY ALLOCATION COMMANDS

The Monitor allocates peripheral devices and core memory to users upon request and protects these allocated facilities from interference by other users. The Monitor maintains a pool of available facilities from which a user can draw.

A user should never abandon a time-sharing console without returning his allocated facilities to the Monitor pool. Until a user returns his allocated facilities to the pool no other users may utilize them.

All devices controllable by the system are listed in Table 5-1. Associated with each device is a physical name, consisting of three letters and zero to three numerals to specify unit number. A logical device name may also be assigned by the user. This logical name of one to six alphanumeric characters of the user's choice is used synonymously with a physical device name in all references to the device. In writing a program, the user may use arbitrarily selected device names which he assigns to the most convenient physical devices at runtime. All references to devices in the Monitor pool are made by physical names or by assigned logical names.

When a device is assigned to a job, it is removed from the Monitor's pool of available facilities. Any attempt by another job to reference the device fails. The device is returned to the pool when the user deassigns it or kills his job.

Monitor Commands to Allocate Facilities

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
ASSIGN ¹ phys-dev log-dev	AS	To assign an I/O device to the user's job for the duration of the job or until a DEASSIGN command is given. phys-dev Any device listed in Table 5-1 ² . This argument is required. log-dev A logical name assigned by the user	m L J	dev: <u>ASSIGNED</u> The device has been successfully assigned to the job. <u>NO SUCH DEVICE</u> Device name does not exist. <u>ALREADY ASSIGNED TO JOB n</u> The device has already been assigned to another user's job. <u>LOGICAL NAME ALREADY IN USE DEVICE dev: ASSIGNED</u> The user has previously assigned this logical name to another device.
DEASSIGN ¹ dev	DEA	Returns one or more devices currently assigned to the user's job to the Monitor's pool of available devices. dev If this argument is not specified, all devices assigned to the user's job are deassigned. If this argument is specified, it can be either the logical or physical device name.	m L J	<u>NO SUCH DEVICE</u> Device name does not exist. <u>DEVICE WASN'T ASSIGNED</u> The device isn't currently assigned to this job.
REASSIGN dev job	REA	Allows one job to pass a device to a second job without going through the Monitor device pool. dev The physical or logical name of the device to be reassigned. Cannot be a user console. job The number of the job to which the device is to be reassigned.	m L J C A	<u>DEVICE dev WASN'T ASSIGNED</u> The device isn't currently assigned to this job. <u>JOB NEVER WAS INITIATED</u> The job number specified has not been initialized. <u>NO SUCH DEVICE</u> The device does not exist. <u>DEVICE CAN'T BE REASSIGNED</u> A user's console Teletype cannot be reassigned.

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
FINISH dev	F	<p>Terminates any input or output currently in progress on the device.</p> <p>dev The logical or physical name of the device on which I/O is to be terminated.</p> <p>If no name is specified, I/O is terminated on all devices assigned to the job.</p>	m L J C A	<p><u>NO SUCH DEVICE</u> Either the device does not exist or it was not assigned to this job.</p>
TALK dev	TA	<p>To allow the user to type directly to another user's console.</p> <p>dev Must be one of the following:</p> <p>CTY - Console Teletype</p> <p>TTYn - Where n can be in the range of 0 through 77.</p> <p>OPR - Operator's console (the Teletype designated as such when the Monitor was initialized).</p>	m	<p><u>BUSY</u> The console addressed is either (1) not in the Monitor mode or (2) is not positioned at the left margin.</p> <p>(OPR is never busy.)</p>
CORE n	COR	<p>To modify the amount of core assigned to the user's job.</p> <p>n = o The low and high segments disappear from the job's virtual addressing space.</p> <p>n > o Total number of 1K blocks of core to be assigned to the job from this point on.</p> <p>If n is omitted, Monitor types out the same response as when an error occurs, but does not change core assignment.</p>	m J A C	<p>10/40 Systems: m/p</p> <p>10/50 Reentrant Systems: m+n/p CORE VIR. CORE LEFT=v</p> <p>Key:</p> <p>m = number of 1K blocks in low segment</p> <p>n = number of 1K blocks in high segment</p> <p>p = maximum K per job swapping systems-max. physical user core non-swapping systems - free + dormant core</p> <p>v = number of K unassigned in core and swapping device</p>

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
RESOURCES	RES	To print out all the available devices (except TTY's) and the number of free blocks on the disk.	m	
*Refer to footnote in Table 2-1				
¹ The ASSIGN command applied to DECTapes clears the copy of the directory currently in core, forcing any directory references to read a new copy from the tape. The DEASSIGN command applied to DECTapes performs the same function. (See 5.7.7 for further details.)				
² If DTA or MTA is used, the Monitor performs a search for an available drive and then types out DTAn (or MTAn) ASSIGNED.				

Examples showing use of logical and physical names:

User types	.ASSIGN DTA,ABC	
Monitor responds	DEVICE DTA6 ASSIGNED	(successful)
User then types	.ASSIGN DTA,DEF	(find another unit)
Monitor responds	NO SUCH DEVICE	(all in use)
User then types	.ASSIGN PTP,ABC	(reserve paper tape punch)
Monitor responds	LOGICAL NAME ALREADY IN USE	(paper tape punch is reserved, but ABC still refers to DTA6 only)
	DEVICE PTP ASSIGNED	
User then types	.ASSIGN DTA1,DEF	
Monitor responds	ALREADY ASSIGNED TO JOB 2	(another user has it)

User then types	.R PIP	(request for system program PIP)
User then types	*PTP:<ABC:FOO	(command string to PIP asking that file FOO be transferred from device ABC (which is now assigned as DTA6) to device PTP (which is assigned to user)).

NOTE: The user does not type the period or the asterisk. The period is the Monitor response to the user and the asterisk is the CUSP response. The user must terminate every command to the Monitor Command Interpreter by pressing the RETURN key on the Teletype.

2.5 SOURCE FILE PREPARATION COMMANDS

The following commands call in the editing programs and cause these programs to open a specified text file for editing. Two of these commands call the TECO CUSP and two call the LINED CUSP (a disk-oriented version of EDITOR). For each editor, one command causes an existing file to be opened for changes and the other command causes a new file to be created. Each command requires a filename as its argument and may have an optional extension.

Filenames are from one to six letters or digits. All letters or digits after the sixth are ignored. A filename is terminated by any character other than a letter or digit. If a filename is terminated by a period, a filename extension is assumed to follow. A filename extension is from one to three letters or digits. It is generally used to indicate file format. The filename extension is terminated by any character other than a letter or digit.

The following are the standard meanings for file extensions:

.TMP	Temporary file
.MAC	Source file in MACRO language
.F4	Source file in FORTRAN IV language
.CBL	Source file in COBOL language (available in 1970)
.LST	Listing or CREF data
.REL	Relocatable binary file
.CMD	Command file, for @ construction
.SAV	Core dump, from SAVE command
blank	Unspecified ASCII text file

Each time one of these commands is executed the command with its arguments is "remembered" as a file on the disk. Because of this, the filename last edited may be recalled for the next edit without specifying the arguments again. For example, if the command

```
.CREATE PROG1.MAC
```

is executed, then the user may later type the command

```
.EDIT
```

instead of

```
.EDIT PROG1.MAC
```

assuming no other source file preparation command was used in the interim.

Table 2-3

Monitor Commands to Prepare Source Files

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
EDIT file.ext	ED	Runs LINED (Line Editor for Disk) and opens an already existing sequence-numbered file on disk for editing.	u L R D J	See Table 2-4'
CREATE file.ext	CREA	Runs LINED and opens a new file on disk for creation.	u L R D J	See Table 2-4
TECO file.ext	TE	Runs TECO (Text Editor and Corrector) and opens an already existing non-sequence-numbered file on disk for editing.	u L R D J	See Table 2-4
MAKE file.ext	M	Runs TECO and opens a new file on disk for creation.	u L R D J	See Table 2-4
*Refer to footnote in Table 2-1				

Table 2-4

Monitor Command Diagnostic Messages

(For File Manipulation Commands)

Message	Meaning
COMMAND ERROR	The COMPIL CUSP cannot decipher the command.
DEVICE NOT AVAILABLE	Specified device could not be initialized.
DISK NOT AVAILABLE	Device DSK: could not be initialized.

Table 2-4 (Cont)

Monitor Command Diagnostic Messages
(For File Manipulation Commands)

Message	Meaning
EXECUTION DELETED (typed by LOADER)	Errors detected during assembly, compilation, or loading prevent a program from being executed. Loading will be performed, but LOADER will EXIT to the Monitor without starting execution.
FILE IN USE OR PROTECTED	A temporary command file could not be entered in the user's UFD.
INPUT ERROR	I/O error occurred while reading a temporary command file from the disk.
LINKAGE ERROR	I/O error occurred while reading a CUSP from device SYS:.
NESTING TOO DEEP	The @ construction exceeds a depth of nine; may be due to a loop of @ command files.
NO SUCH FILE - file.ext	Specified file could not be found (may be a source file or a file required for operation of COMPIL CUSP).
NOT ENOUGH CORE	System cannot supply enough core for use as buffers or to read in a CUSP.
OUTPUT ERROR	I/O error occurred while writing a temporary command file on disk.
PROCESSOR CONFLICT	Use of + construction has resulted in a mixture of source languages.
TOO MANY NAMES or TOO MANY SWITCHES	Command string complexity exceeds table space in COMPIL CUSP.
UNRECOGNIZABLE SWITCH	An ambiguous or undefined word followed a slash (/).

2.6 FILE MANIPULATION COMMANDS

Each of the following commands performs complex functions which would require a number of commands on a less sophisticated system. The commands in Table 2-5 list the user's files and file directories and cause his source files to be compiled, loaded, and executed.

Table 2-5

Monitor Commands to Manipulate Files

Commands	Abbreviation	Explanation	Characteristics*	Monitor Messages
TYPE list	TY	<p>Directs PIP (Peripheral Interchange Program) to type contents of named source file(s) on user's Teletype.</p> <p>list A single file specification, or a string of file specifications separated by commas. A file specification is the same as that described for COMPILE, LOAD, EXECUTE, and DEBUG commands. In addition, the * construction can be used as follows:</p> <p> filename.* All files with this filename and any extension</p> <p> *.ext All files with this extension and any filename</p> <p> *.* All files</p> <p>Examples: TYPE FILEA, DTAO:FILEB.MAC,*.TMP TYPE A,DTA4:B,C[15,107]</p>	m L R D	See Table 2-4
LIST list	LI	<p>Directs PIP to list contents of named source file(s) on the line printer (LPT).</p> <p>Examples: LIST TEST.* LIST *.MAC LIST DTA4:A,B,C</p>	m L R D	See Table 2-4
DIRECT dev	DI	<p>If dev: is omitted or DSK:, directory listing of user's disk files is typed on the user's Teletype. If DTAn: is specified, directory of that DECTape is typed.</p> <p>Two switches can be used with the DIRECT command: /F List short form of directory (i.e., omit dates) /L List on line printer (LPT) instead of Teletype.</p> <p>The : may be omitted in dev.</p>	m L R D	See Table 2-4

Commands	Abbreviation	Explanation	Characteristics*	Monitor Messages																
DELETE list	DEL	Deletes one or more files from disk or DECTape. If a device name is specified, it remains in effect until changed or end of command string is reached.	m L R D	See Table 2-4																
RENAME arg	REN	<p>Changes the name of one or more files on disk or DEC tape. The arg is a pair of file specifications separated by an = sign, or a string of such pairs separated by commas:</p> <p style="text-align: center;">RENAME new1 = old1,new2 = old2,...</p> <p>Device names can be specified only with the new filename and remain in effect until changed or end of command string is reached.</p>	m L R D	See Table 2-4																
CREF	CREF	Runs CREF and lists on the line printer any CREF listing files generated by previous COMPILE, LOAD, EXECUTE, and DEBUG commands using the /CREF switch. The file containing the names of these CREF-listing files is then deleted so that subsequent CREF commands will not list them again.	m L R D	See Table 2-4																
COMPILE list	COM	<p>Produces relocatable binary file(s) for the specified program(s). The use of the MACRO assembler and/or the FORTRAN IV compiler is determined as follows.</p> <table border="0" style="width: 100%;"> <tr> <td style="width: 50%;"><u>Condition</u></td> <td style="width: 50%;"><u>Action</u></td> </tr> <tr> <td>If no .REL (binary) file</td> <td>Translate source file</td> </tr> <tr> <td>If source-file [date, time] is later than binary-file [date, time]</td> <td>Translate source file</td> </tr> <tr> <td>If other than above</td> <td>Do not translate source file; use current .REL (binary) file.</td> </tr> </table> <table border="0" style="width: 100%;"> <tr> <td style="width: 50%;"><u>Source File Extension</u></td> <td style="width: 50%;"><u>Translator Used</u></td> </tr> <tr> <td>.MAC</td> <td>MACRO assembler</td> </tr> <tr> <td>.F4</td> <td>FORTRAN IV compiler (F40)</td> </tr> <tr> <td>Other than above, or null</td> <td>"Standard processor" is used (see 2.6.2).</td> </tr> </table> <p>The list of files which may be a single file specification, or a string of file specifications separated by commas. A file specification consists of a filename (with or without an extension) and may include a device name (if the source file is not disk) or a project-programmer number (if the source file is not in the user's disk area).</p> <p>Examples: PROG1,PROG1.MAC,PROG1.F4,PROG1.XYZ,DTAO:PROG1 PROG1[10,16],PROGA,DTAO:PROGB PROGC.MAC</p> <p>(See 2.6.1, 2.6.2)</p>	<u>Condition</u>	<u>Action</u>	If no .REL (binary) file	Translate source file	If source-file [date, time] is later than binary-file [date, time]	Translate source file	If other than above	Do not translate source file; use current .REL (binary) file.	<u>Source File Extension</u>	<u>Translator Used</u>	.MAC	MACRO assembler	.F4	FORTRAN IV compiler (F40)	Other than above, or null	"Standard processor" is used (see 2.6.2).	m L R D	See Table 2-4
<u>Condition</u>	<u>Action</u>																			
If no .REL (binary) file	Translate source file																			
If source-file [date, time] is later than binary-file [date, time]	Translate source file																			
If other than above	Do not translate source file; use current .REL (binary) file.																			
<u>Source File Extension</u>	<u>Translator Used</u>																			
.MAC	MACRO assembler																			
.F4	FORTRAN IV compiler (F40)																			
Other than above, or null	"Standard processor" is used (see 2.6.2).																			

Commands	Abbreviation	Explanation	Characteristics*	Monitor Messages
LOAD list	LOA	Performs the COMPILE function for the specified program(s), then runs LOADER and loads the .REL files.	m L R D	See Table 2-4
EXECUTE list	EX	Performs the COMPILE and LOAD functions for the specified program(s) and begins execution of the loaded program.	u L R D	See Table 2-4
DEBUG list	DEB	<p>Performs the COMPILE and LOAD functions and, in addition, prepares for debugging. DDT (the Dynamic Debugging Technique program) is loaded first, followed by the user's programs with local symbols. DDT is entered on completion of loading.</p> <p>Examples: COMPILE PROGA EXECUTE DTAL:TEST.MAC DEBUG/L FILEA,FILEB,FILEC/N,FILED</p> <p>(Generate listings for FILEA,FILEB,and FILED; see 2.6.2)</p> <p>LOAD FILEA,FILEB,%60000FILEC</p> <p>(Pass origin switch to LOADER; see "Loader-Switches" 2.6.4)</p>	u L R D	See Table 2-4
*Refer to footnote in Table 2-1				

Each time a COMPILE, LOAD, EXECUTE, or DEBUG COMMAND is executed, the command with its arguments is "remembered" as a file on the disk. Because of this, the filename last used may be recalled for the next command without specifying the arguments again. (See last paragraph in Section 2.5)

2.6.1 Extended Command Forms

The commands shown in Table 2-5 are adequate for the compilation and execution of a single program or a small group of programs at one time. However, the assembly of large groups of programs, such as the FORTRAN library or the Time-Sharing

Monitor, is more easily accomplished by means of one or more of the extended command forms.

2.6.1.1 The @ File

When there are many program names and switches, they can be put into a file so that they do not have to be typed in for each compilation. This is accomplished by the use of the "@ file" construction, which may be combined with any of the commands in Tables 2-3 and 2-5.

The "@ file" must appear at any point after the first word in the command. In this construction "file" must be a filename, which may have an extension and project-programmer numbers. If the extension is omitted, a search is made for the command file with a null extension and then for a command file with the extension .CMD. The information in the command file specified is then put into the command string to replace the characters "@ file".

For example, if the file FLIST contains the string

```
FILEB,FILEC/LIST,FILED
```

then the command

```
COMPILE FILEA,FILEB,FILEC/LIST,FILED,FILEZ
```

could be replaced by

```
COMPILE FILEA,@FLIST,FILEZ
```

Command files themselves may contain the "@ file" construction to a depth of nine levels. If this indirecting process should result in files pointing in a loop, the maximum depth will rapidly be exceeded and an error message will be produced.

The following rules are used in the handling of format characters in a command file.

a) Spaces are used to delimit words but are otherwise ignored. Similarly, the characters TAB, VTAB, and FORM are treated like spaces.

b) The characters CARRIAGE RETURN, LINE-FEED, AND ALTMODE are ignored if the first non-blank character after a sequence of returns, line-feeds, and altmodes is a comma. Otherwise, they are treated either as commas by the COMPILE, LOAD, EXECUTE, and DEBUG commands or as command terminators by all the other commands appearing in Tables 2-3 and 2-5.

c) Blank lines are completely ignored since strings of returns and line-feeds are considered together.

d) Comments may be included in command files by preceding the comment with a semicolon. All text from the semicolon through the line-feed is ignored.

e) If command files are sequenced, the sequence numbers are ignored.

2.6.1.2 The "+" Construction¹

A single relocatable binary file may be produced from a collection of input source files by means of the "+" construction. For example, a user may wish to compile the parameter file, S.MAC, the switch file, FT50SB.MAC, and the file that is the body of the program, APRSER.MAC. This is specified by the following command:

```
COMPILE S + FT50SB + APRSER
```

The name of the last input file in the string is given to any output (.REL and/or .LST) files (e.g., APRSER. in the foregoing example). The source files in the "+" construction may each con-

¹Used in COMPILE, LOAD, EXECUTE, and DEBUG commands only.

tain device and extension information and project-programmer numbers.

2.6.1.3 The "=" Construction¹

Usually the filename of the binary file is the same as that of the source file, with the extension specifying the difference. This can be changed by use of the "=" construction, which allows a filename other than the source filename to be given to the output file. For example, if a binary file is desired with the name BINARY.REL from a source program with the name SOURCE.MAC, the following command is used.

```
COMPILE BINARY = SOURCE
```

This same technique may be used to specify an output name to a file produced by use of the "+" construction. To give the name WHOLE.REL to the binary file produced by PART1.MAC and PART2.MAC, the following is typed.

```
COMPILE WHOLE = PART1 + PART2
```

2.6.1.4 The "<>" Construction¹

The "<>" construction causes the programs within the angle brackets to be assembled with the same parameter file. If a + is used, it must appear before the <> construction. For example, to assemble the files LPTSER.MAC, PTPSER.MAC, and PTRSER.MAC, each with the parameter file S.MAC, the user may type

```
COMPILE S + LPTSER, S + PTPSER, S + PTRSER
```

But by using the angle brackets, the command becomes

```
COMPILE S + <LPTSER,PTPSER,PTRSER>
```

The user cannot type

```
COMPILE <LPTSER,PTPSER,PTRSER>+ S
```

¹Used in COMPILE, LOAD, EXECUTE, and DEBUG commands only.


```
COMPILE /LIST A,B,C
```

will generate listings of all three programs.

```
COMPILE A/LIST, B,C
```

will generate a listing only of program A.

```
COMPILE /LIST A, B/NOLIST, C
```

will generate listings of programs A and C.

The compile-switch "CREF" is just like "LIST", except that a cross-reference listing is generated, to be processed later by the program "CREF".

Unless the /LIST or /CREF is specified, no listing file is generated. The LIST command is used to obtain printer output of regular listing files and the CREF command to obtain printer output of CREF listing files.

Since the "LIST", "NOLIST", AND "CREF" switches are so commonly used, the switches "L", "N", and "C" are defined with the corresponding meanings, even though there are (for instance) other switches beginning with the letter "L". Thus the command

```
COMPILE /L A
```

produces a listing file "A.LST" (as well as, of course, "A.REL").

2.6.2.2 The "Standard Processor"

The "standard processor" is used to compile or assemble programs which do not have the extensions .MAC, .F4, or .REL. There are a number of switches for setting the "standard processor". If all source files are kept with the appropriate extensions, this subject can be disregarded.

If the command

```
COMPILE A
```

is executed and there is a file named "A." (that is, with a blank

extension), then "A." will be translated to "A.REL" by the "standard processor". Similarly, if the command

```
COMPILE FILE.NEW
```

is executed, the extension ".NEW", although meaningful to the user, does not specify a language, so the "standard processor" will be used. For these cases the user must be able to control the setting of the "standard processor".

The "standard processor" is FORTRAN IV at the beginning of each command string.

The "standard processor" may be changed by the following compile-switches:

MACRO	change standard to MACRO
M	same as MACRO
FORTRAN	change standard to FORTRAN IV
F	same as FORTRAN
REL	change standard to use RELocatable binary; i.e., use existing .REL files, even though a newer source file may be present. (Useful primarily in LOAD, EXECUTE, DEBUG commands).

These switches may be used as "temporary" or "permanent". For example, assume that programs A, B, and C exist on the disk, with blank extensions. Then

```
COMPILE A, B/M, C
```

will cause A and C to be translated by FORTRAN, B by MACRO.

```
COMPILE A, /M B, C
```

will cause A to be translated by FORTRAN, B and C by MACRO.

NOTE

Programs with .MAC and .F4 extensions are always translated by the extension implied, regardless of the "standard processor."

2.6.2.3 Forced Compilation

The compilation (or assembly) occurs if the source file is at least as recent as the relocatable binary file. If the binary is newer than the source, there is not normally any need to perform the translation.

There are cases, however, where such extra translation may be desirable, as for instance, when one desires a listing of the assembly. To force such an assembly, the switch "COMPILE" is provided, again in both temporary and permanent form. For example:

```
COMPILE /CREF / COMPILE A, B, C
```

will create cross-reference listing files A.LST, B.LST, and C.LST, even though current .REL files may exist. In fact, the binary files will also be recreated.

The corresponding switch "NOCOMPILE" is also provided, to turn off the forced-compile made. Note that this differs from the /REL switch which turns off even the normal compilation caused by a source file newer than the .REL file.

2.6.2.4 Library Searches

The LOADER normally performs a library search of the FORTRAN library. Sometimes it is necessary to search other files as libraries. To do this, the compile-switches "LIBRARY" and (its complement) "NOSEARCH" are provided.

These switches may be used as either "permanent" or "temporary".

For example, suppose a special library file named SPCLIB.REL were kept on device SYS at a particular installation. Then to compile and load a user program, library search the

special library, and then search the normal FORTRAN library, the following command could be used:

```
LOAD MAIN,SYS:SPCLIB/LIB
```

At this point, it should be noted that the program SPCLIB is not assembled simply because its source file is presumably not on device SYS. The COMPILE process will compile any program named in the command string, if its source is present and not older than the .REL file, unless prevented by the /REL switch.

2.6.2.5 Loader Maps

Loader maps are produced during the loading process by the compile-switch "MAP". When this switch is encountered, a loader map is requested from the Loader. The map will be written with filename MAP.MAP, in the user's disk area.

This compile-switch is the one exception to the "permanent compile-switch" rule, in that it causes only one map to be output, even though it may appear as a permanent switch.

2.6.3 Processor Switches ¹

Occasionally it is necessary to pass switches to the assembler or compiler. Recall that for each translation (assembly or compilation), a command string is sent to the translator containing three parts: the source files, a binary output file, and a listing file. If the user wishes to add switches to those files, he must do so as follows:

- a) If the "+" construction is used, group the switches according to each related source filename.
- b) Group the switches according to the three types of files (source, binary, and listing) for each source filename.

¹Used in COMPILE, LOAD, EXECUTE, and DEBUG commands only.

- c) For each source filename, separate the groups of switches by commas.
- d) Enclose all the switches for each source filename within one set of parentheses.

(SSSS)	Only source switches are present
(SSSS,BBBB)	Source and binary switches are present
(SSSS,BBBB,LLLL)	Source, binary, and listing switches are present.

- e) Place each parenthesized string immediately after the source filename to which it refers.

Examples:

DEBUG TEST(N) Suppress typeout of errors during assembly.

COMPILE OUTPUT = MTA0:(W,S,M)/L
 Rewind the magtape (W), compile the first file, produce binary output for the PDP-6(S), and eliminate the MACRO coding from the output listing (M). Output files are given the names OUTPUT.REL and OUTPUT.LST.

COMPILE/MACRO A = MTA0:(W,,Q) /L
 Rewind the magtape (W), compile the first file, and suppress Q (questionable) error indications on the listing. Note that when a binary switch is not present, the delimiting comma must appear.

COMPILE/MACRO A = MTA0:(,,Q)/L
 Compile file at current position of the tape and suppress Q error indications on the listing. Note that when the source and binary switches are not present, the delimiting comma must appear.

2.6.4 Loader Switches¹

In unusually complex loading processes, it may be necessary to pass loader-switches to the LOADER to direct its

¹Used in COMPILE, LOAD, EXECUTE, and DEBUG commands only.

operation. These are passed via the COMPILER, LOAD, EXECUTE, and DEBUG commands. These switches must be passed to the LOADER (not to the compiler or assembler). This is accomplished by the % character. The % has the same meaning as that of the / in the Loader's command string. Also, like the /, it takes one letter (or a sequence of digits and one letter) following it. Therefore, to set a program origin of 6000 for program C, the user types

```
LOAD A,B, %60000C,D
```

The most commonly used switches are:

%S Load with symbols

%nO Set program origin to n

%F Cause early search of FORTRAN library

%P Prevent FORTRAN library search

2.6.5 Temporary Files

The COMPIL CUSP decipheres the commands found in Tables 2-3 and 2-5 and constructs new commands for the CUSPS that were referenced. These new commands are written as temporary files on the disk, as are all of the Monitor-level commands. COMPIL and the other CUSPS transfer control directly to one another without requiring additional typed-in commands from the user.

Temporary filenames have the following form:

```
nnnxxx.TMP
```

where nnn is the user's job number in decimal, with leading zeros to make three digits and xxx specifies the use of the file. In the filenames listed below, job number 1 will be assumed.

2.6.5.1 001SVC.TMP

This file contains the most recent COMPILER, LOAD, EXECUTE, or DEBUG command which included arguments. It is used to remember those arguments. See section 2.6.

2.6.5.2 001EDS.TMP

This file contains the most recent EDIT, CREATE, TECO, or MAKE command which included an argument. It is used to remember that argument. See section 2.5

2.6.5.3 001MAC.TMP

This file contains commands to MACRO. It is written by COMPIL, and read by MACRO. It contains one line for each program to be assembled, and (if required) the command

NAME!

to cause MACRO to transfer control to the named CUSP ("name" may be F40, LOADER, etc.).

2.6.5.4 001FOR.TMP

This file corresponds exactly to the one described in the preceding section, except that it is read by the FORTRAN IV compiler, F40.

2.6.5.5 001PIP.TMP

This file is written by COMPIL and read by PIP. It contains ordinary PIP commands to implement the DIRECTORY, LIST, TYPE, RENAME, and DELETE commands.

2.6.5.6 001CRE.TMP

This file is written by COMPIL and read by CREF. It contains commands to CREF corresponding to each file which has produced a CREF listing on the disk.

COMPIL also reads this file, if it exists, each time a new CREF listing is generated, to prevent multiple requests

for the same file, and to prevent discarding other requests which may not yet have been listed.

2.6.5.7 001EDT.TMP

This file is written by COMPIL for each EDIT, CREATE, TECO, or MAKE command, and is read by either the LINED or TECO CUSP.

For the commands MAKE or CREATE, it contains the command

Sfile.ext (ALTMODE)

For the commands TECO or EDIT, it contains the command

Sfile.ext (RETURN) (LINEFEED)

2.7 RUN CONTROL COMMANDS

By using a run control command, the user can load core image files from retrievable storage devices (i.e., disk, DECTape, magnetic tape). These files can be retrieved and controlled from the user's console. Files stored on disk and DECTape are addressable by name. Files on magnetic tape require the user to preposition the tape to the beginning of the file.

Table 2-6

Monitor Commands to Call, Load, and Control Programs

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
RUN dev file.ext [proj,prog] core	RU	<p>To load a core image from a retrievable storage device and start it at the location specified within the file (JOBSA).</p> <p>If the program has two segments, both the low and high segments will be set up. If the high file has extension .SHR (as opposed to .HGH), the high segment will be shared. A two-segment program may have a low file extension (.LOW).</p> <p>dev The logical or physical name of the device containing the core image.</p> <p>file.ext The name of the file containing the core image; if .ext is omitted, it is assumed to be SHR + LOW, HGH + LOW, or SAV. See SAVE, SSAVE.</p> <p>[proj.prog] Project-programmer number; required only if core image file is located in a disk area other than the user's.</p> <p>core Amount of core to be assigned if different from minimum core needed to load the the program or from the core argument of the SAVE command which saved the file. Since previous core is returned, MTA must have this argument because there is no directory to tell how much core for low segment.</p>	u L J	<p>dev: <u>NOT AVAILABLE</u> The device has been assigned to another job.</p> <p><u>NO SUCH DEVICE</u> The device does not exist.</p> <p><u>nK OF CORE NEEDED</u> There is insufficient free core to load the file.</p> <p><u>NOT A DUMP FILE</u> The file is not a core image file.</p> <p><u>TRANSMISSION ERROR</u> A parity or device error occurred during loading.</p>
R file.ext core	R	<p>Same as RUN SYS: file.ext core. The R command is the usual way to run a CUSP that does not have a direct Monitor command to run it.</p>	u L J R	Same as RUN
GET dev file.ext [proj,prog] core	G	<p>Same as RUN command except that Monitor types out</p> <p style="text-align: center;">JOB SETUP</p> <p>and does not start execution.</p>	m L J A	Same as RUN

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
START adr	ST	<p>Begins execution of a program previously loaded with the GET command.</p> <p>adr The address at which execution is to begin if other than the location specified within the file (JOBSA). If adr is not specified, the starting address comes from JOBSA.</p>	u L J C A	<p><u>NO CORE ASSIGNED</u> No core was allocated to the user when the GET command was given and no core argument was specified in the GET.</p> <p><u>NO START ADR</u> Starting address was 0 because user failed to specify a starting address in END statement of source program.</p>
HALT (†C)	†C	Places the console in Monitor mode and transmits a HALT command to the Monitor Command Interpreter. Stops the job and stores the program counter in the job data area (JOBPC).	m	
CONT	CON	Starts the program at the saved program counter address stored in JOBPC by a HALT command (†C) or a HALT instruction.	u L J C	<p><u>CAN'T CONTINUE</u> The job was halted due to a Monitor-detected error and can't be continued.</p>
DDT	DD	Copies the saved program counter value from JOBPC into JOBOPC and starts the program at an alternate entry point specified in JOBDDT (beginning address of DDT as set by Linking Loader). DDT contains commands to allow the user to start or resume at any desired address	u L J C	<p><u>NO START ADR</u> DDT starting address was 0 (JOBDDT).</p>
REENTER	REE	Similar to the DDT command. Copies saved program counter value from JOBPC into JOBOPC and starts program at an alternate entry point specified in JOBREN (must be set by the user or his program).	u L J C	<p><u>NO START ADR</u> REENTER starting address was 0 (JOBREN).</p>
E adr	E	<p>Examines a core location in the user's area (high or low segment).</p> <p>adr If this argument is specified, the contents of the location are typed out in half-word octal mode. Adr is required the first time the E or D command is used.</p> <p> If adr is not specified, the contents of the location following the previously specified E adr or the location of the previous D adr are typed out.</p>	m L J C	<p><u>OUT OF BOUNDS</u> The specified adr is not in the user's core area, or the user does not have read privileges to file which initialized the high segment.</p>

Command	Abbreviation	Explanation	Characteristics*
D lh rh adr	D	<p>Deposits information in the user's core area (high or low segment).</p> <p>lh-- The octal value to be deposited in the left half of the location.</p> <p>rh The octal value to be deposited in the right half of the location.</p> <p>adr The address of the location into which the information is to be deposited.</p> <p>If adr is omitted, the data is deposited in the location following the last D adr or in the location of the last E adr.</p>	<p>m L J C</p> <p><u>OUT OF BOUNDS</u> The specified adr is not in the user's core area, or high segment is write protected and user does not have write privileges to file which initialized the high segment.</p>
SAVE dev file.ext core	SA	<p>Writes out a core image of the user's core area on the specified device. Saves any user program (reentrant, one segment non-reentrant, or two segment non-reentrant) as one or two files. Later when the program is loaded by a GET, R, or RUN command, it will be non-reentrant. If DDT was loaded with the program, the entire core area is written; if not, the area starting from zero up through the program break (as specified by JOBBF) is written.</p> <p>dev The device on which the core image file is to be written.</p> <p>file.ext The name to be assigned to the core image file. If ext is omitted and the program has only one segment, the ext is assumed to be .SAV. If ext is omitted and the program has two segments, the high segment will have extension .HGH, and the low segment will have extension .LOW.</p> <p>core Amount of core in which the program is to be run. This value is stored in the job's core area (JOBCOR) and is used by the RUN and GET commands. Specified as number of 1K blocks.</p>	<p>m L J C A</p> <p><u>n 1K BLOCKS OF CORE NEEDED</u> The user's current core allocation is less than the contents of JOBBF.</p> <p><u>DEVICE NOT AVAILABLE</u> Device dev is assigned to another user.</p> <p><u>TRANSMISSION ERROR</u> An error was detected while reading or writing the core image file.</p> <p><u>DIRECTORY FULL</u> The directory of device dev is full; no more files can be added.</p> <p><u>JOB SAVED</u> The output is completed.</p>

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
		If core is omitted, only the number of blocks required by the core image area (as explained above) is assumed.		
SSAVE dev file.ext core	SSA	Same as SAVE except that the high segment will be sharable when it is loaded with the GET command. To indicate this sharability, the high segment is written with extension .SHR instead of .HGH. A subsequent GET will cause the high segment to be sharable. Because an error message is not given if the program does not have a high segment, a user can use this command to save CUSP's without having to know which are sharable.	m L J A C	
*Refer to footnote in Table 2-1				

2.7.1 Additional Information on SAVE and SSAVE

Low segment files will be zero compressed on all devices (DTA,MTA,DSK), but high segment files will not since the high segment may be shared at the time of the command. Saved files are ordinary binary files and can be copied using the /B switch in PIP.

In order to save file space, only the high segment up through the highest location (relative to high segment origin) loaded, as specified in the LH of JOBHRL, will be written by the SAVE command. If LH is zero (high segment created by CORE or REMAP UUO) or DDT is present, the entire high segment will be written.

It is possible for most programs to be written so that only the high segment contains non-zero data. This will also save file space and I/O time with the GET command. SAVE will write the high segment (.HGH) only. The LOADER will indicate to the SAVE command that no data was loaded above the Job Data area in the low segment by setting the LH of JOBCOR to the highest location loaded in the low segment with non-zero data.

There are a number of locations in the Job Data area which need to be initialized on a GET, even though there is no other data in the low segment. The SAVE command copies these locations into the first 10_g locations of the high segment, provided it is not sharable. These 10 locations are referred to as the Vestigial Job Data area. Therefore, the LOADER will load high segment programs starting at location 400010.

To prevent user confusion, SAVE and SSAVE delete a previous file with the extension .SHR or .HGH. Therefore, SAVE deletes a file with the extension .SHR and SSAVE deletes a file with the extension .HGH. Both commands also delete a file with the extension !LOW, if the high segment was the only segment written.

The regular access rights of the saved file indicate whether a user can do a GET, R, or RUN command. These commands will assume that the user wants to execute (but not modify) the high segment independent of the access rights of the file used to initialize the segment. The Monitor will always enable the hardware user-mode write protect to prevent the user program from storing into the segment inadvertently.

To debug a reentrant CUSP which is in the system directory, the user should make a private, non-sharable copy, rather than modifying the shared version and possibly causing

harm to other users. To make a private, non-sharable copy, the following commands are used.

- a) GET SYS CUSP
- b) SAVE dev CUSP Writes a file in the user directory as non-sharable. The high segment in the user's addressing space remains sharable.
- c) GET dev CUSP Overlays the sharable program with the non-sharable one from the user's directory. Now the user can make patches while other users share the version in the system directory.

The Monitor will keep the shared and the non-shared versions separate from each other. A sharable program may be superseded into the directory by the SSAVE command. The Monitor will clear the high segment in its table of storable segments in use but will not remove the segment from the addressing space of users currently using it. Only the users doing a GET, R, or RUN command or a RUN or GETSEG UWO will have the new sharable version.

When the SAVE or SSAVE command is used to save a sharable program with only a high file, the Monitor will not modify the Vestigial Job Data area unless the user has write privileges to the file which initialized the shared segment. This prohibits unauthorized users from modifying the first 10 locations of a shared segment. This restriction does not exist if a low file is also written, since the GET command reads the low file after the high file. The real Job Data area locations are set from the low file.

2.8

BACKGROUND JOB CONTROL COMMANDS

A job is a background, or detached, job if it is not under control of a user console. Any console can initiate any number of background jobs. I/O to the console while a job is running in a background mode causes the job to stop until a console is attached.

Table 2-7

Monitor Commands to Control Background Jobs

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
PJOB	PJ	<p>Monitor responds by typing the job number to which the user's console is attached.</p> <p>10/40 System - If the console is not attached to a job, Monitor assigns a job number and types the job number and a line identifying the Monitor version.</p> <p>10/50 System - If the console is not attached to a job, Monitor responds with LOGIN PLEASE.</p>	m L J	
CSTART CCONT	CS CC	<p>Identical to the START and CONT commands, respectively, except that the console is left in the Monitor mode. To Use:</p> <ol style="list-style-type: none"> 1. Begin the program with the console in user mode. 2. Type control information to the program, then type ↑C to halt job with console in Monitor mode. 3. Type CCONT to allow job to continue running and leave console in Monitor mode. 4. Further Monitor commands can now be entered from the console. <p>Caution: These commands should not be used when the user program (which is continuing to run) is also requesting input from the console.</p>	m L J C	Same as START and CONT.
DETACH	DET	<p>Disconnects the console from the user's job without affecting the status of the job. The user console is now free to control another job, either by initiating a new job or attaching to a currently running background job.</p>	d L	

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
ATTACH job	AT	<p>Connects a console to a background job.</p> <p>job The job number of the job to which the console is to be attached.</p> <p>[proj,prog] The project-programmer number of the originator of the desired job. May be omitted if same as job to which console is currently attached. The operator (device OPR) may always attach to a job even though another console is attached, provided he specifies the proper [proj,prog].</p>	m	<p>If an error message occurs, the console remains attached to its current job.</p> <p><u>TTYn ALREADY ATTACHED</u> Job number typed is erroneous and is attached to another console, or another user is attached to the job.</p> <p><u>NOT A JOB</u> The job number is not assigned to any currently running job.</p> <p><u>CAN'T ATTACH TO JOB</u> The project-programmer number entered is not that of the originator of the desired job.</p>
*Refer to footnote in Table 2-1				

2.9 JOB TERMINATION COMMANDS

When a user leaves the system, all facilities allocated to his jobs must be returned to the Monitor facility pool so that they are available to other users.

Table 2-8

Monitor Command to Terminate Jobs

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
KJOB	K	<p>In Multiprogramming Systems: Stops all allocated I/O devices and returns them to the Monitor pool. Returns all allocated core to the Monitor pool. Returns the job number to the pool. Leaves the console in the Monitor mode. Performs an automatic TIME command.</p> <p>In Swapping Systems: All of the above procedures. In addition, if user has any files, responds with:</p> <p>CONFIRM:</p>	m A	

Command	Abbreviation	Explanation	Characteristics*
		<p>To which the user may type ↑C to abort log-out; or type one of the following: K) to kill job and delete all unprotected files; L) to list his disk directory; I) to individually save and delete files as follows:</p> <p>After each file name is listed, type: P to save and protect, S to save without protecting, or) to delete. Files with extensions.LST and .TMP will be deleted automatically.</p>	Monitor Messages
*Refer to footnote in Table 2-1			

2.10 SYSTEM TIMING COMMANDS

All system times are kept in increments of one-sixtieth or one-fiftieth of a second, depending on the power frequency of the country in which the PDP-10 is installed.

Table 2-9

Monitor Commands for System Timing

Command	Abbreviation	Explanation	Characteristics*
DAYTIME	DA	<p>Types the date followed by the time of day. Time is typed in the format.</p> <p style="text-align: center;">hh:mm</p> <p>where hh = hours mm = minutes</p>	m
TIME job	TI	<p>Types out the total running time since the last TIME command followed by the total running time used by the job since it was initialized (logged in), followed by the integrated product of running time and core size (KILO-CORE-SEC=). Time is typed in the format</p> <p style="text-align: center;">hh:mm:ss.hh</p> <p>where hh = hours mm = minutes ss.hh = seconds to nearest hundredth.</p>	m

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
		<p>Interrupt level and job scheduling times are charged to the user who was running when the interrupt or rescheduling occurred.</p> <p>job The job number of the job whose timing is desired.</p> <p> If job is omitted, the job to which the console is attached is assumed. In this case, Monitor types out the incremental running time (running time since last TIME command) as well as the total running time since the job was initialized.</p> <p> If job = 0, an approximation of the time spent core shuffling (SHFL) is printed, followed by the amount of time spent clearing core (ZCOR), the running time of the null job (NULL), the time during which one or more jobs wanted to run but were swapped out or in the process of being swapped out (LOST), and the total time system has been up (UP).</p>		
*Refer to footnote in Table 2-1				

2.11 SYSTEM ADMINISTRATION COMMANDS

The SYSTAT command permits a user to learn how heavily the system is loaded and the status of devices in the sharable device pool. The other commands in this section are restricted to system administrators only.

Monitor Commands for System Administration

Command	Abbreviation	Explanation	Characteristics*	Monitor Messages
SCHEDULE n	SCH	<p>Changes the scheduled use of the system, depending on n. This command is legal only from the operator's console. n is stored in RH of STATES word in COMMON:</p> <p>0 = regular time sharing. 1 = no further LOGINS allowed. 2 = no further LOGINS from remote TTY's.</p> <p>If n is omitted, the current value of n is printed.</p>	m	
SYSTAT	SYS	<p>Types out status of the system: system name, time of day, date, uptime, percent null time. Status of each job: job number, project-programmer number (**** if detached), TTY number, program name being run, size of low segment, state (RN = runnable, TT = TTY input wait, C = Monitor command mode) and run time. Status of high segments being used: name, directory name, size, number of users in core or on disk. Status of each assigned device: name, job number, how assigned (AS = ASSIGN command, INIT = INIT UO).</p>		
ASSIGN SYS:dev		<p>To change the systems device to device "dev." The user must be logged in under either [1,1] or [1,2].</p>	m L J	
DETACH dev	DET	<p>To assign the device "dev" to JOB 0, thus making it unavailable. The user must be logged in under [1,1].</p>	m L J	
ATTACH dev	AT	<p>To return a detached device to the Monitor pool of available devices. The user must be logged in under [1,1].</p>	m L J	
CTEST		<p>This command is used by system programmers to test extensions made to the COMPIL CUSP.</p>	L R	
*Refer to footnote in Table 2-1				

2.12 MONITOR DIAGNOSTIC MESSAGES

Once a user program has been started, a number of error conditions may arise which cause the job to revert to monitor mode. The error messages typed, and the meanings for each are summarized in the following table.

Table 2-11

Time-Sharing Monitor Diagnostic Messages

Message	Meaning
<p>The typein is typed back followed by ?)</p>	<p>The Monitor command decoder has encountered an incorrect character, such as a letter in a numeric argument. The incorrect character appears immediately before the ?. Example: User types in: CORE ABC Monitor responds: CORE A ?)</p>
<p>ADDRESS CHECK FOR DEVICE dev AT USER adr</p>	<p>Monitor has checked a user address and has found it to be too large (>C(JOBREL)) or too small (<JOBPFI). Some user addresses can be the user's accumulators while others cannot.</p> <p>One of the following addresses may be wrong: buffer buffer header dump mode command list data specified by dump mode command list insufficient core available for setting up Monitor-generated buffers.</p>
<p>BAD DIRECTORY FOR DEVICE DTAn; UUO AT USER adr</p>	<p>The DEctape directory is not in proper format or had a parity error when read. Many times this error occurs when an attempt is made to use a virgin tape.</p>
<p>DEVICE dev OK?</p>	<p>Device dev is temporarily in an inoperable state, such as LPT off-line. The user should correct the obvious condition and then type a CONT command.</p>

Time-Sharing Monitor Diagnostic Messages

Message	Meaning
ERROR IN JOB n	A fatal error has occurred in the user's job (or in Monitor while servicing the job). This typeout is normally followed by a 1-line description of the error.
HALT AT USER adr	The user program has executed a halt instruction at loc. adr. Typing CONT will resume execution at the effective address of the halt.
HUNG DEVICE dev; UO AT USER adr	A device has not generated an interrupt for a timed period and, therefore, is in need of attention.
ILLEGAL DATA MODE FOR DEVICE dev AT USER adr	The data mode specified for a device in the user's program is illegal.
ILLEGAL UO AT USER adr	An illegal UO has been executed at user location adr.
ILL INST. AT USER adr	An illegal operation code has been encountered in the user's program.
ILL MEM REF AT USER adr	An illegal memory reference has been made by the user program at adr or adr+1.
INCORRECT RETRIEVAL INFORMATION: UO AT USER adr	The retrieval pointers for a file are not in the correct format; the file is unreadable. If this typeout occurs, the user should report it on a Software Trouble Report.
INPUT DEVICE dev CANNOT DO OUTPUT; UO AT USER adr	An illegal OUTPUT UO has been executed at user location adr.
I/O TO UNASSIGNED CHANNEL AT USER adr	No OPEN or INIT was performed on the channel.
LOOKUP AND ENTER HAVE DIFFERENT NAMES: UO AT USER adr	An attempt has been made to read and write a file on the disk. However, the LOOKUP and ENTER UO's have specified different names on the same user channel. This message does not indicate a DECTape error.

Table 2-11 (Cont)

Time-Sharing Monitor Diagnostic Messages

Message	Meaning
<p>MASS STORAGE DEVICE FULL; UWO AT USER adr</p>	<p>The storage disk is full. Users must delete unneeded files before the system can proceed.</p>
<p>NON-RECOVERABLE DISC READ ERROR; UWO AT USER adr</p>	<p>Monitor has encountered an error while reading or writing a critical block in the disk file structure (e.g., the MFD or the SAT table).</p>
<p>NON-RECOVERABLE DISC WRITE ERROR; UWO AT USER adr</p>	<p>If this condition persists, the disk must be reloaded using Fail-safe after the standard location for the MFD and SAT table has been changed using the Monitor once-only dialogue.</p>
<p>NOT ENOUGH FREE CORE IN MONITOR: UWO AT USER adr</p>	<p>The Monitor has run out of free core for assigning disk data blocks and Monitor buffers. If this type-out occurs, the user should report it on a Software Trouble Report.</p>
<p>NOT FOUND</p>	<p>The program file requested cannot be found on the systems device (or on the specified device).</p>
<p>OUTPUT DEVICE dev CANNOT DO INPUT; UWO AT USER adr</p>	<p>An illegal INPUT UWO has been executed at user location adr.</p>
<p>PC EXCEEDS MEMORY BOUND AT USER adr</p>	<p>An illegal transfer has been made by the user program to user location adr.</p>
<p>SWAP READ ERROR</p>	<p>A consistent checksum error has been encountered when checksumming locations JOBDAC through JOBDAC+74 of the Job Data area during swapping.</p>

CHAPTER 3
LOADING USER PROGRAMS

3.1 MEMORY PROTECTION AND RELOCATION

Each user program is run with the processor in a special mode known as the user mode, in which the program must operate within an assigned area in core and certain operations are illegal. Since every user has an assigned area in core, the rest of core is unavailable to him; he cannot gain access to the protected area for either storage or retrieval of information.

The assigned area of each user may be divided into two segments. If this is the case, the low segment is unique for a given user and can be used for any purpose. The high segment may be used by a single user or it may be shared by many users. If the high segment is shared by other users, the program is a reentrant program. The Monitor can write-protect the high segment so that the user cannot alter its contents. This is done, for example, when the high segment is a pure procedure to be used reentrantly by many users. One high pure segment may be used with any number of low impure segments. See Chapter 1 for the distinctions between pure and impure segments. Any user program which attempts to write in a write-protected high segment is aborted and receives an error message. If the Monitor defines two segments but does not write-protect the high segment, the user has a two-segment non-reentrant program (see SETUWP UUO).

The Time-Sharing Monitor defines the size and position of a user's area by specifying protection and relocation addresses for the low and high segments. The protection address is the maximum relative address the user can reference. The relocation address is the absolute core address of the first location in the segment, as

seen by the Monitor and the hardware. The Monitor defines these addresses by loading four 8-bit registers (two 8-bit registers in PDP-10's without the KT10A option), each of which corresponds to the left eight bits of an 18-bit PDP-10 address. Thus, segments always contain an even multiple of 1024 words.

In user mode, the PDP-10 hardware automatically relocates user addresses by adding the contents of the memory relocation register in the central processor to the high-order eight bits of the user address before the address is sent to memory. The address before the addition is the relative address and after the addition is the absolute address. To determine whether a relative address is legal, its eight high-order bits are compared with the contents of the memory protection register. If the relative address is greater than the contents of the memory protection register, the Memory Protection flag is set in the central processor, and control traps to the Monitor, which aborts the user program and prints an error message on the user's console (unless the user program has instructed the Monitor to pass such interrupts to itself for error-handling). See APRENB U00, 4.3.3.1.

Some systems have only the low pair of protection and relocation registers. In this case, the user program is always non-reentrant and the assigned area comprises only the low segment.

When the Monitor schedules a user's program to run, the memory protection and relocation registers are set to the bounds of the user's allocated core area and the central processor is switched to user mode.

To take advantage of the fast accumulators, memory addresses 0-17 are not relocated, all users having access to the accumulators. Therefore, relative locations 0-17 cannot be referenced by a user's program. The Monitor saves the user's accumulators

in this area when the user's program is not running and while the Monitor is servicing a UWO from the user. See Book 1 for a more complete description of the relocation and protection hardware.

3.2 USER'S CORE STORAGE

A user's core storage consists of blocks of memory whose sizes are an integral multiple of 1024_{10} (2000_8) words. In a non-reentrant Monitor, the user's core storage is a single contiguous block of memory. After relocation, the first address in a block is a multiple of 2000_8 . The relative user and relocated address configurations are illustrated below, where P_L , R_L , P_H , and R_H are the protection and relocation addresses, respectively, for the low and high segments as derived from the 8-bit registers loaded by the Monitor. If the low segment is more than half the maximum memory capacity ($P_L > 400000$), the high segment starts at the first location after the low segment (at $P_L + 2000$). The high segment is limited to 128 K.

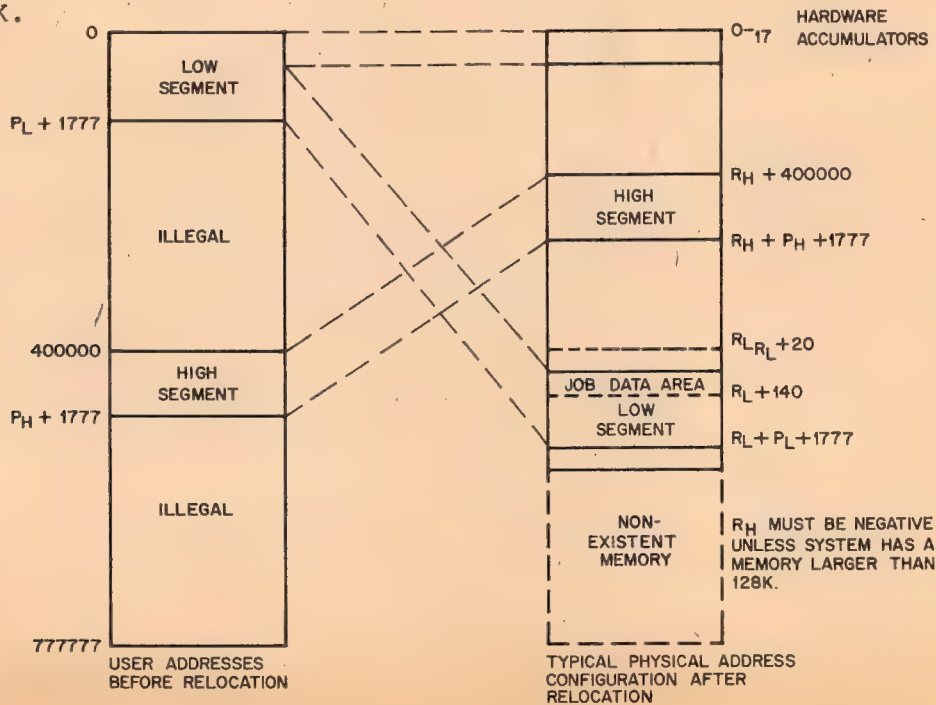


Figure 3-1

User's Core Area

There are two methods available to the user for loading his core area. The simplest way is to load a core image stored on a retrievable device (see RUN and GET, Chapter 2). The other method is to use the relocatable binary loader to link-load binary files. The user may then write the core image on a retrievable device for future use (see SAVE, Chapter 2).

3.2.1 Job Data Area

The Job Data area provides storage for specific information of interest to both the Monitor and the user. The first 140 (octal) locations of the user's core area always are allocated to the Job Data area. Locations in this area have been given mnemonic assignments whose first three characters are JOB. Therefore, all mnemonics in this manual with a JOB prefix refer to locations in the Job Data area.

Table 3-1

Job Data Area Locations
(for user-program reference)

Name	Octal Location	Description
JOBUUO	40	User's location 40 ₈ . Used for processing user UOO's (001 through 037). Op code and effective address are stored here.
JOB41	41	User's location 41 ₈ . Contains the beginning address of the user's programmed operator service routine (usually a JSR or PUSHJ).
JOBERR	42	Left half: Unused at the present. Right half: Accumulated error count from one CUSP to the next. CUSPs should be written to look at the right half only.
JOBREL	44	Left half: 0. Right half: The highest relative core location available to the user (i.e., the contents of the memory protection register when this user is running).

Table 3-1 (Cont)

Job Data Area Locations

(for user-program reference)

Name	Octal Location	Description
JOBBLT	45	Three consecutive locations where the LOADER puts a BLT instruction and a CALLI UUO to move the program down on top of itself. These locations are destroyed on every executive UUO by the executive pushdown list.
JOBDDT	74	Contains the starting address of DDT. If contents are 0, DDT has not been loaded.
JOBCN6	106	Six temporary locations used by CHAIN (FORTRAN Runtime Routine) after it releases all I/O channels. JOBCN6 is defined to be in JOBJDA.
JOBHRL	115	Left half: First relative free location in the high segment (relative to the high segment origin so it is the same as the high segment length). Set by the LOADER and subsequent GETs, even if there is no file to initialize the low segment. The left half is a relative quantity because the high segment can appear at different user origins at the same time. The SAVE command uses this quantity to know how much to write from the high segment. Right half: Highest legal user address in the high segment. Set by the Monitor every time the user starts to run or does a CORE or REMAP UUO. The word is ≥ 401777 unless there is no high segment, in which case it will be zero. The proper way to test if a high segment exists is to test this word for a non-zero value.
JOBSYM	116	Contains a pointer to the symbol table created by Linking Loader. Left half: Negative count of the length of the symbol table. Right half: Lowest register used.
JOBUSY	117	Contains a pointer to the undefined symbol table created by Linking Loader. Not yet used by DDT.
JOBSA	120	Left half: First free location in low segment (set by Loader). Right half: Starting address of the user's program.

Table 3-1 (Cont)

Job Data Area Locations
(for user-program reference)

Name	Octal Location	Description
JOBFF	121	Left half: 0. Right half: Address of the first free location following the low segment. Set to C(JOBSA) _{LH} by RESET UUU.
JOBREN	124	Left half: Unused at present. Right half: REENTER starting address. Set by user or by Linking Loader and used by REENTER command as an alternate entry point.
JOBAPR	125	Left half: 0. Right half: Set by user program to trap address when user is enabled to handle APR traps such as illegal memory, pushdown overflow, arithmetic overflow, and clock. See CALL APRENB UUU.
JOBCNI	126	Contains state of APR as stored by CONI APR when a user-enabled APR trap occurs.
JOBTPC	127	Monitor stores PC of next instruction to be executed when a user-enabled APR trap occurs.
JOBOPC	130	The previous contents of the user's program counter are stored here by Monitor upon execution of a DDT, REENTER, START, or CSTART command.
JOBCHN	131	Left half: 0 Address of first location after first FORTRAN IV loaded program. Right half: Address of first location after first FORTRAN IV Block Data.
JOBCOR	133	Left half: Highest location in low segment loaded with non-zero data. No low file written on SAVE or SSAVE if less than 140. Set by the LOADER. Right half: User argument on last SAVE or GET command. Set by the Monitor.
JOBVER	137	Left half: Zero or the programmer number of the programmer who made last identification to the program. Right half: Program version number in octal. The number is never converted to decimal. After a GET, R, or RUN command, a E command can be used to find the version number. (Digital always distributes CUSPs with the left half = 0, so customers making modifications to CUSPs should change only the left

Table 3-1 (Cont)

Job Data Area Locations

(for user-program reference)

Name	Octal Location	Description
JOBVER (Cont)	137	half. The right half will remain as a record of the Digital version.)
JOBDA	140	The value of this symbol is the first location available to the user.
NOTE		
<p>Only those JOBDAT locations of significant importance to the user are given in this table. JOBDAT locations not listed include those which are used by the Monitor and those which are unused at the present time. User programs should not refer to any locations not listed above since such locations are subject to change without notice.</p>		

Some locations in the Job Data area, such as JOBSA and JOBDDT, are set by the user's program for use by the Monitor. Others, such as JOBREL, are set by the Monitor for use by the user's program. In particular, the right half of JOBREL contains the highest legal address set by the Monitor whenever the user's core allocation changes.

JOBDAT exists in binary form in the Systems Library for loading with user programs that refer to Job Data area locations symbolically. User programs must reference locations by means of the assigned mnemonics, which are declared as EXTERNAL references to the assembler. JOBDAT is loaded automatically, if needed, during the Loader's library search for undefined global references, and the values are assigned to the mnemonics.

3.2.2 Loading Relocatable Binary Files

The relocatable binary loader (LOADER, V.47) which resides in the system file is started by the command

R LOADER core

where core is an optional argument. (See Book 5 for a description of the Loader command string.)

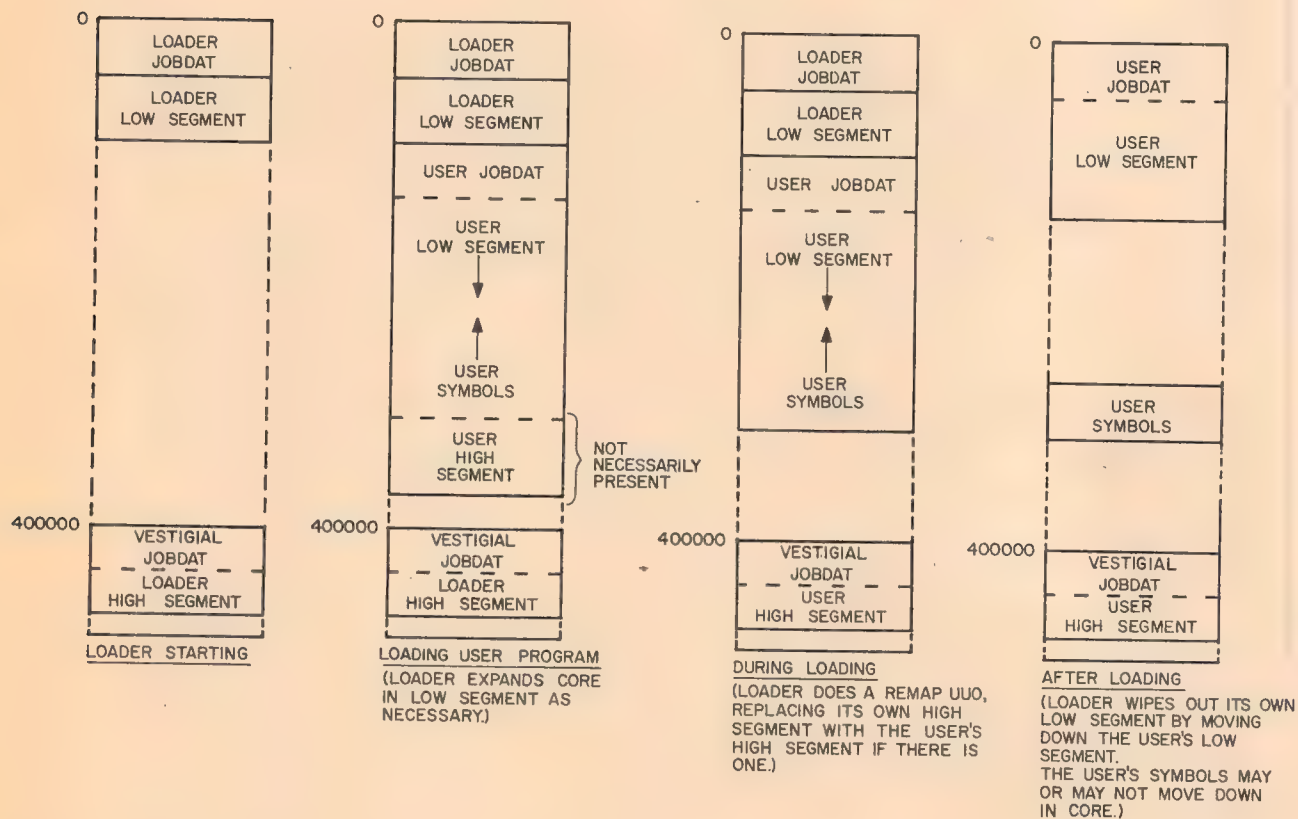


Figure 3-2

Loading User Core Area

In writing reentrant user software, an effort is made to minimize the support required to run such software on a machine having only a single relocation register. Both the source and relocatable binary files are the same for a reentrant program that must run on a non-reentrant system.

Since the Loader is reentrant, its instructions exist in the high segment. In loading two segments, both segments are data with respect to the Loader and must exist in the low segment during load time. Therefore, the following Loader variables must be duplicated for each segment:

- a) offset (the number of locations a program must be moved toward zero before it can be executed),
- b) program origin (the location assigned by the Loader to relocatable zero of a program), and
- c) location counter (the register that indicates the location of the next instruction to be interpreted).

3.2.2.1 The H Switch

A program written to be reentrant can be loaded into one segment instead of two by use of the H switch (/H). This switch is used only when a two-segment program is to be loaded into one segment. This switch is not required when a one-segment program is to be loaded into one segment.

To minimize the use of the H switch on single-register machines, the Loader will check to see if the system (i.e., hardware plus software) has a two-segment capability. If the Monitor has this capability but the machine does not, then the system does not have the two-segment capability. If the system does not have the two-segment capability, the Loader automatically loads a two-segment program into one segment, just as if the user had typed

the H switch.

To find out if the system has a two-segment capability, the Loader uses the SETUWP UWO and attempts to set the user mode write-protect bit to one. An error return indicates a single-register capability. The Loader cannot produce a two-segment program, and the Monitor cannot save a program as two segments.

If a user wants to load a program in which the low segment is longer than 400000 octal words, he can use the switch NNNNNNH. This switch changes the origin of the high segment from its initial setting of 400000 to NNNNNN where NNNNNN is larger. If NNNNNN is missing, the Loader loads everything into the low segment.

Since it is not known before load time whether a reentrant program is not going into the high segment, the code executed (including the Monitor UWO's) is the same for either case.

3.2.2.2 The HISEG Pseudo-Op

After loading, a relocatable subprogram assembled by MACRO is either put entirely in the user low segment or entirely in the user high segment. To indicate that a subprogram is to be loaded into the high segment, the HISEG pseudo-op is used. It can appear anywhere in the program although it is best to place it at the beginning since a reader of the program wants to know that the program is destined for the high segment. Near the beginning of the binary output, MACRO generates code that tells the Loader to load subprograms into the high segment. Loader Version 47 loads programs in any order. In earlier versions of the Loader, programs for the low segment must be loaded before any programs for the high segment.

3.2.2.3 The Vestigial Job Data Area

There are a few "constant" data in the Job Data area

which may be loaded by a two-segment, one-file program without using instructions on a GET command (JOB41, JOBREN, JOBVER) and there are a number of locations which the Monitor loads on a GET (JOBSA, JOBCOR, JOBHRL). The Vestigial Job Data area (the first 10 locations of the high segment) is reserved for these low segment constants. Therefore, a high segment program is loaded into 400010 instead of 400000. With the Vestigial Job Data area in the high segment, the Loader automatically loads the constant data into the Job Data area without requiring a low file on a GET, R, or RUN command, or a RUN UUU. SAVE will write a low file for a two-segment program only if the LH of JOBCOR is 140₈ or greater.

3.2.2.4 Completion of Loading

The new program code is loaded upward from an offset above the resident Loader. The program origin (i.e., the first location loaded) is 140₈, unless the user changes it by means of the assembler LOC pseudo-instruction. After completion of the loading but before exiting, the Loader does the following.

a) Sets the LH of JOBSA and the RH of JOBFF to the address of the first location above the new code area (i.e., the program break). The RH of JOBSA is set to the program starting address. This value is the last non-zero address of the assembler END pseudo-instruction to be loaded, or zero. It is used by the RUN and START commands. The LH of JOBFF is zero.

b) Sets the LH of JOBHRL to the new highest relative user address (relative to the high segment origin) in high segment, or zero if no high segment.

c) Sets the LH of JOBCOR to the highest location in the low segment that is loaded with non-zero data.

d) Uses REMAP UUU to take the top part of the low segment

which contains the user's high segment, and replaces the Loader high segment.

- e) May move symbols and reduce core, if DDT was loaded.
- f) Calls EXIT or starts up program.

If DDT was loaded by means of the D switch in the Loader command string, the RH of JOBDDT is set by the Loader to the starting address of DDT and the LH is zero. A new switch, /K, has been implemented for use with DDT. This switch moves core back to the absolute maximum needed. A /nK moves core back to nK.¹

¹In the latest version of the Loader, V.50, the /D is used to imply /B/K.

CHAPTER 4
USER PROGRAMMING

The PDP-10 central processor operates in one of three modes: executive mode, user I/O mode, or user mode. The Monitor operates in executive mode, which is characterized by the lack of memory protection and relocation (see Chapter 3) and by the normal execution of all defined operation codes. The user I/O mode is a special mode, wherein memory protection and relocation are in effect, as well as the normal execution of all defined operation codes. (This mode is not used by the Monitor, and is not normally available (see TRPSET) to the time-sharing user.) User programs are run in user mode in order to guarantee the integrity of both the Monitor and each user program.

4.1 USER MODE

The user mode of the central processor is characterized by the following features:

- a) Automatic memory protection and relocation (see Chapter 3).
- b) Trap to absolute location 40 on any of the following:
 1. Operation codes 040 through 077 and operation code 000.
 2. Input/output instructions (DATAI, DATAO, BLKI, BLKO, CONI, CONO, CONSZ, and CONSO).
 3. HALT (i.e., JRST 4,).
 4. Any JRST instruction that attempts to enter executive mode or user I/O mode.
- c) Trap to relative location 40 on execution of operation codes 001 through 037.

Since user programs run in user mode, the Monitor must

perform all input/output operations for the user, as well as any other user-requested operations that are not available in user mode. The purpose of this chapter is to describe the services the Monitor makes available to user mode programs and how a user program obtains such services.

4.2 PROGRAMMED OPERATORS (UUO's)

Operation codes 000 through 077 in the PDP-10 are programmed operators (sometimes referred to as UUO's- Unimplemented User Operators since from a hardware point of view their function is not pre-specified); some of these op-codes trap to the Monitor and the rest trap to the user program.

After the effective address calculation is complete, the contents of the instruction register, along with the effective address, are stored in user or Monitor location 40 and the instruction in user or Monitor location 41 is executed out of normal sequence. Location 41 must contain a JSR instruction to a routine to interpret the contents of location 40.

4.2.1 Operation Codes 001-037 (User UUO's)

Operation codes 001 through 037 do not affect the mode of the central processor. Thus, when executed in user mode, they trap to user location 40, which allows the user program complete freedom in the use of these programmed operators.

If a user's undebugged program accidentally executes one of these op-codes when the user did not intend to use it, the following error message is normally issued.

ERROR IN JOB n

ILLEGAL UUO AT USER 41

This message is given because the user's relative location 41

contains zero (unless his program has overtly changed it) and 000 is an illegal Monitor UUO.

4.2.2 Operation Codes 040-077, and 000 (Monitor UUO's)

Operation codes 040 through 077 and 000 trap to absolute location 40, with the central processor in executive mode. These programmed operators are interpreted by the Monitor to perform input/output operations and other control functions for the user's program.

Operation code 000 always returns the user to monitor mode with the error message:

ERROR IN JOB n

ILLEGAL UUO AT USER addr

Table 4-1 lists the operation codes 040 thru 077 and their mnemonics. Most of this chapter is a detailed description of their operation.

4.2.2.1 CALL and CALLI - Operation codes 040 through 077 limit the Monitor to 40_g operations. The CALL operation extends this set by specifying the name of the operation by the contents of the location specified by the effective address, e.g., CALL [SIXBIT/EXIT/] This provides for indefinite extendability of the Monitor operations, at the overhead cost to the Monitor of a table lookup.

The CALLI operation eliminates the table lookup of the CALL operation by having the programmer perform the lookup himself and specify the index to the operation in the effective address of the CALLI. Table 4-2 lists the Monitor operations specified by the CALL and CALLI operations.

The customer is allowed to add his own CALL and CALLI calls to the Monitor. A negative CALLI effective address

(starting with -2) should be used to specify such customer added operations.

4.2.2.2 Restriction on Monitor UWO's in Re-Entrant User Programs

There are a number of restrictions on UWO's which involve a high segment. These restrictions are to prevent naive or malicious users from clobbering other users while sharing segments and to minimize Monitor overhead in handling two-segment programs. The basic rules are as follows.

a) All UWO's can be executed from the low or high segment although some of their arguments cannot be in, or refer to, the high segment.

b) No buffers, buffer headers, or dump mode command lists may exist in the high segment for reading from or writing to any I/O device.

c) No I/O is processed into or out of the high segment except via the SAVE and SSAVE commands.

d) No STATUS, CALL or CALLI UWO allows a store in the high segment.

e) The effective address of the LOOKUP, ENTER, INPUT, OUTPUT, and RENAME UWO's cannot be in the high segment. If any one of these rules is violated, an address check error message is given (see Table 2-11).

f) As a convenience in writing user programs, the Monitor makes a special check so that the INIT UWO can be executed from the high segment, even though the calling sequence is in the high segment. The Monitor also allows the effective address of the CALL UWO (contains the SIXBIT Monitor function name) and the effective address of the OPEN UWO (contains the status bits, device name, and buffer header addresses) in the high segment.

4.2.3 Operation Codes 100-127 (Unimplemented Op Codes)

Op code 100-UJEN Dismisses realtime interrupt
from user mode (see 4.3.6.2).

Op codes 101-127 Monitor prints ILL INST AT
USER n and stops job.

4.2.4 Illegal Operation Codes

The eight input/output instructions (DATAI, etc.) and JRST instructions attempting to enter executive or user I/O mode from the user mode are interpreted by the Monitor as illegal instructions. The job is stopped and the following error message is printed on the user's console.

ERROR IN JOB n

ILL INST AT USER addr

4.3 PROGRAM CONTROL

4.3.1 Starting

All program starting is accomplished by the Monitor commands RUN, START, CSTART, CONT, CCONT, DDT, and REENTER (see Chapter 2). The starting address is either an argument of the command or stored in the user's job data area (see Chapter 3).

4.3.1.1 CALL AC, [SIXBIT/SETDDT/] or CALLI AC,2 - These cause the contents of the AC to replace the DDT starting address, which is stored in the protected job data area location, JOBDDT. This starting address is used by the Monitor command, DDT (see 3.2.2.4).

4.3.2 Stopping

Any one of the following procedures can stop a running program:

a) One ↑C from user console if user program is in a Teletype input wait; otherwise, two ↑C's from user console (see Chapter 2);

b) A Monitor detected error; or

c) Program execution of HALT, CALL [SIXBIT/EXIT/], or CALL [SIXBIT/LOGOUT/].

4.3.2.1 Illegal Instructions (700-777, JRST 10, JRST 14,) and Unimplemented Op Codes (101-127) -

Illegal instructions trap to the Monitor, stop the job, and print:

ERROR IN JOB

ILL.INST.AT USER n

Note that the program cannot be continued by typing the CONT or CCONT commands.

4.3.2.2 HALT or JRST 4, - The HALT instruction is an exception to the illegal instructions; it traps to the Monitor, stops the job, and prints:

ERROR IN JOB

HALT AT USER n

However, the CONT and CCONT commands are still valid and, if typed, will continue the program at the effective address of the HALT instruction. HALT is not the instruction used to terminate a program (see EXIT, section 4.3.2.3). HALT is useful for catching "impossible" error conditions.

Table 4-1
Monitor Operation Codes

Op Code	Mnemonic	Function
040	CALL	Operation code extension (See 4.2.2.1)
041	INIT	Initialize I/O device (See 4.4.2.2)
042		No operation
043		No operation
044		No operation
045		No operation
046		No operation
047	CALLI	Operation code extension (See 4.2.2.1)
050	OPEN	Open file (See 4.4.2.2)
051	TTCALL	Special Teletype Operations (See 5.1.3)
052		No operation
053		No operation
054		No operation
055	RENAME	Rename or delete a file (See 4.4.2.5)
056	IN	Input and Skip on error of EOF (See 4.4.3)
057	OUT	Output and skip on error of EOF (See 4.4.3)
060	SETSTS	Set file status (See 4.4.4)
061	STATO	Skip on file status one (See 4.4.4)
062	STATUS or GETSTS	Read file status (See 4.4.4)
063	STATZ	Skip on file status zero (See 4.4.4)
064	INBUF	Set up input buffer ring (See 4.4.2.3)
065	OUTBUF	Set up output buffer ring (See 4.4.2.3)
066	INPUT	Read (See 4.4.3)
067	OUTPUT	Write (See 4.4.3)
070	CLOSE	Close file (See 4.4.5)

Table 4-1 (Cont)

Monitor Operation Codes

Op Code	Mnemonic	Function
071	RELEAS	Release device (See 4.4.7)
072	MTAPE	Position tape (See 5.8.2 and 5.7.5)
073	UGETF	Get next free block number (See 5.7.5)
074	USETI	Set next input block number (See 5.7.5)
075	USETO	Set next output block number (See 5.7.5)
076	LOOKUP	Select file (See 4.4.2.4)
077	ENTER	Create file (See 4.4.2.4)
100	UJEN	Dismiss real-time interrupt (See 4.3.6.2)

Table 4-2

CALL and CALLI Monitor Operations

CALLI AC, x	CALL AC, [SIXBIT/y/]	Function
x = -2,.. ..,-n	y = Customer defined	Reserved for definition by each customer installation.
-1	LIGHTS	Displays AC in console lights
0	RESET	Reset I/O devices (See 4.4.2.1)
1	DDTIN	DDT mode console input (See 5.1.2)
2	SETDDT	Set protected DDT starting address (See 4.3.1.1)
3	DDTOUT	DDT mode console output (See 5.1.2)
4	DEVCHR	Get device characteristics (See 5.12)
5	(DDTGT)	No operation
6	(GETCHR)	Same as DEVCHR(4)

Thus, the user can set up a priority interrupt trap into his re-located core area. Upon a normal return, AC contains the previous contents of the address specified by LH of AC, so that the user program may restore the original contents of the PI location when the user is through using these UWO's. If the LH of AC is not within the range 40 through 57, an error return will be given just as if the user was not job 1.

The call is:

```

                MOVE AC, XWD N, ADR
                CALL AC, [SIXBIT/TRPSET/]
                error return
                normal return
                .
                .
                .
ADR:   JSR TRAP           ;Instruction to be stored
                        ;in exec PI location
                        ;after relocation added to it.
TRAP:  0                 ;Here on interrupt from exec.

```

The Monitor assumes that user location ADR contains either a JSR U or BLKI U, where U is a user address. Consequently, the Monitor will add the job's relocation to the contents of location U to make it an absolute IOWD. Therefore, a user should reset the contents of U before every TRPSET call.

```

                MOVEI AC, PNTR
                HRRM AC, ADR
                MOVE AC, XWD N, ADR
                CALL AC, [SIXBIT/TRPSET/]
                error return
                normal return
                .
                .
                .
ADR:   BLKI DEV,PNTR     ;Block in PNTR to be stored
                        ;in interrupt location
PNTR:  IOWD LEN,BUFFER

```

This UWO is a temporary expedient until some real-time UWO's are implemented which will not stop time sharing and which cannot crash the system.

2	FORCE	Job being forced to swap out
3	FIT	Job waiting to be fit into core
4	VIRTUAL	Amount of virtual core left in system in K (initially set to no. of K of swapping space)
5	SWPERC	LH=no. of swap read or write errors RH=error bits (bits 18-21 same as status bits) +no. of K discarded

4.3.6 Direct User I/O

The user I/O mode (bits 5 and 6 of PC word = 11) of the central processor allows running privileged user programs with automatic protection and relocation in effect. This mode provides some protection against partially debugged Monitor routines, and permits running infrequently used device service routines as a user job. Direct control by the user program of special devices is particularly important in real-time applications.

To utilize this mode, the job number must be 1.

CALL [SIXBIT/RESET/] or CALLI 0 terminates user I/O mode.

4.3.6.1 CALL AC, [SIXBIT/TRPSET/] or CALLI AC, 25 - These are privileged UWO's which may or may not stop time-sharing (stop jobs from being scheduled) and allow the user program to gain control of the interrupt locations. If the user is not job 1, an error return to the next location after the CALL will always be given and the user will remain in user mode. Time-Sharing will be turned back on. If the user is job 1, the central processor is placed in user I/O mode. Under job 1, if AC contains zero, time-sharing is turned back on if it was turned off. If the LH of AC is within the range 40 through 57, all other jobs are stopped from being scheduled and the specified executive PI location (40-57) is patched to trap directly to the user. In this case, the Monitor moves the contents of the relative location specified in the right half of AC, adds the job relocation address to the address field, and stores it in the specified executive PI location.

Bit 6=1 If clock is 50 cycle instead of 60 cycle

Set by the privileged operator command, SCHEDULE:

Bit 34=1 Means no remote LOGINS

Bit 35=1 Means no more LOGINS

20 SERIAL Serial number of PDP-10 processor
Set by MONGEN dialog

Entries in ODPTBL (once only disk parameters)

<u>Item</u>	<u>Location</u>	<u>Use</u>
0	SWPHGH	Highest logical block # in the swapping space
1	K4SWAP	K of disk words set aside for swapping
2	PROT	In-core protect time multiplies size of job in K-1
3	PROTO	In-core protect time added to above result after multiply

Entries in NSWTBL (non-swapping data)

<u>Item</u>	<u>Location</u>	<u>Use</u>
0	CORTAB	Map of physical core
-		1 bit for each K of core
7	CORTAB+7	
10	CORMAX	Size in words of largest legal user job (low seg+high seg)
11	CORLST	Byte pointer to last free block in CORTAB
12	CORTAL	Total free+dormant+idle K physical core left
13	SHFWAT	Job no. shuffler has stopped
14	HOLEF	Abs. adr. of job above lowest hole, 0 if no job
15	UPTIME	Time system has been up in jiffies
16	SHFWRD	Tot. no. of words shuffled by system
17	STUSER	Number of job using SYS if not a disk
20	HIGHJB	Highest job number currently assigned
21	CLRWRD	Total no. of words cleared by CLRCOR
22	LSTWRD	Total no. of clock ticks when null job ran and other jobs wanted to but couldn't because: <ol style="list-style-type: none"> 1. Swapped out or on way in or out 2. Monitor waiting for I/O to stop so can shuffle or swap 3. Job being swapped out because expanding core

Entries in SWPTBL (swapping data)

<u>Item</u>	<u>Location</u>	<u>Use</u>
0	BIGHOL	No. of K in biggest hole in core
1	FINISH	+Job no. of job being swapped out -Job no. of job being swapped in

Table Numbers (RH of AC) (Cont)

02	- PRJPRG	(project and programmer numbers) Index by job or segment number
03	- JBTPRG	(user program name) Index by job or segment number
04	- TTIME	(total time used) Index by job number
05	- JBTKCT	(Kilo-core ticks) Index by job number
06	- JBTPRV	(privilege bits) Index by job number
07	- JBTSWP	(job's swapping parameters) Index by job or segment number
10	- TTYTAB	(Teletype to job translation) Index by line number
11	- CNFTBL	(configuration table) Index by item number, see below
12	- NSWTBL	(non-swapping data) Index by item number, see below
13	- SWPTBL	(swapping data) Index by item number, see below
14	- JBTSGN	(high segment table) Index by job number
15	- ODPTBL	(once-only disk parameters) Index by item number, see below

Entries in CNFTBL (Configuration Table)

<u>Item</u>	<u>Location</u>	<u>Use</u>
0	CONFIG	Name of system in ASCIZ
-		
4	CONFIG+4	
5	SYSDAT	Date of system in ASCIZ
6	SYSDAT+1	
7	SYSTAP	Name of the system device (SIXBIT)
10	TIME	Time of day in jiffies
11	THSDAT	Today's date (12-bit format)
12	SYSSIZ	Highest location in the Monitor + 1
13	DEVOPR	Name of the OPR TTY console (SIXBIT)
14	DEVLST	LH is start of DDB (device-data-block) chain
15	SEGPTR	LH=-# of high segments, RH=+# of JOBS (counting NULL job)
16	TWOREG	Non-zero if system has two-register hardware and software
17	STATES	Location describing feature switches of this system in LH, and current state in RH

Assembled according to MONGEN dialog and S.MAC:

Bit 0=1 If disk system (FTDISK)
 Bit 1=1 If swap system (FTSWAP)
 Bit 2=1 If LOGIN system (FTLOGIN)
 Bit 3=1 If full duplex software (FTTTYSER)
 Bit 4=1 If privilege feature (FTPRV)
 Bit 5=1 If assembled for choice of reentrant
 or non-reentrant software at Monitor
 load time (FT2REL)

4.3.5.6 CALL AC, [SIXBIT/GETTAB/] or CALLI AC, 41 - These provide a mechanism for user programs to examine the contents of certain Monitor locations in a way which will not vary from Monitor to Monitor.

The call is:

```
CALL AC, [SIXBIT/GETTAB/]      ;OR CALLI AC, 41
error return
normal return
```

The left half of AC contains a job number or some other index to a table. Some job numbers may refer to high segments of programs by using arguments greater than the highest job number for the current Monitor. A negative LH means the current job number. The right half of AC contains a table number from the list of Monitor data tables and parameters set forth below. The entries in these tables are all globals in the Monitor subroutine COMMON. The actual values of the core addresses of these locations are subject to change and can be found in the LOADER storage map for the Monitor. The complete descriptions of these globals are found in the listing of COMMON.

An error return leaves the AC unchanged. This means that the job number or index number in the left half of AC was too high, or the table number in the right half of AC was too high, or that the user does not have the privilege of accessing that table. A skip return supplies the contents of the requested table in AC, or a zero if the table is not defined in the current Monitor.

The SYSTAT CUSP makes frequent use of these UVO's.

The list of tables and their entries is as follows, with a brief description of each.

Table Numbers (RH of AC)

```
00 - JBTSTS (job status word)
      Index by job or segment number
01 - JBTADR (job relocation and protection)
      Index by job or segment number
```

on the user's console

↑C
.

The console is left in Monitor mode ready to accept the user's first command.

Any other user program that calls these UUO's receives the error message

ILLEGAL UUO AT USER addr

The user's console is then put in Monitor mode, and the CONT and CCONT commands are not permitted.

4.3.5.4 CALL AC, [SIXBIT/PEEK/] or CALLI AC, 33 - These allow a user program to examine any location in the Monitor. Some customers may want to restrict the use of this UUO to project 1.

The call is:

```
MOVEI AC, exec address          ;TAKEN MODULO SIZE OF MONITOR
CALL AC, [SIXBIT/PEEK/]        ;OR CALLI AC, 33
```

This call returns with the contents of the Monitor location in AC. It is used by SYSTAT and could be used for on-line Monitor debugging.

4.3.5.5 CALL AC, [SIXBIT/GETLIN/] or CALLI AC, 34 - These return the SIXBIT physical name of the Teletype console that the program is attached to.

The call is:

```
CALL AC, [SIXBIT/GETLIN/]      ;OR CALLI AC, 34
```

The name is returned left-justified in the AC.

Example:

```
CTY or TTY3 or TTY30
```

This UUO is used by the LOGIN program to print the TTY name.

4.3.4.5 CALL AC, [SIXBIT/SLEEP/] or CALLI AC, 31 - These stop the job, and continue automatically after an elapsed real time of [c(AC)xclock frequency] modulo 2^{12} jiffies.

The contents of the AC are thus interpreted as the number of seconds the job wishes to sleep; however, there is an implied maximum of approximately 68 seconds (82 seconds in 50 Hz countries) or one minute.

4.3.5 Identification

4.3.5.1 CALL AC, [SIXBIT/PJOB/] or CALLI AC, 30 - These return the job number right-justified in accumulator AC.

4.3.5.2 CALL AC, [SIXBIT/GETPPN/] or CALLI AC, 24 - These return in AC the project-programmer pair of the job. The project number is a binary number in the left half of AC, and the programmer number is a binary number in the right half of AC. If the program being run is LOGIN or LOGOUT from the system device, the current project-programmer number is changed to 1,2 so that all files are accessible for reading and writing, and a skip return is given if the old project-programmer number is also logged in on another job.

4.3.5.3 CALL AC, [SIXBIT/LOGIN/] or CALLI AC, 15 - These are not available to user programmers. They are for the exclusive use of the LOGIN CUSP, which uses these operators to exit to the Monitor and to pass it certain crucial parameters (including project and programmer numbers) about the user who just successfully logged in. When the LOGIN CUSP calls these UUO's, any devices the UUO's were using are released, and the following is printed

4.3.4 Timing Control

The central processor clock, which generates interrupts at the power-source frequency (60 Hz in North America, 50 Hz in most other countries), keeps time in the Monitor. Each clock interrupt (tick) corresponds to 1/60th (or 1/50th) of a second of elapsed real time. The clock is set initially to the current time of day by console input when the system is started, as is the current date. When the clock reaches midnight, it is reset to zero, and the date is advanced.

4.3.4.1 CALL AC, [SIXBIT/DATE/] or CALLI AC, 14 - A 12-bit binary integer computed by the formula

$$\text{date} = (\text{year} - 1964) \times 12 + (\text{month} - 1) \times 31 + \text{day} - 1$$

represents the date.

This integer representation is returned right-justified in accumulator AC.

4.3.4.2 CALL AC, [SIXBIT/TIMER/] or CALLI AC, 22 - These return the time of day, in clock ticks (jiffies), right-justified in accumulator AC.

4.3.4.3 CALL AC, [SIXBIT/MSTIME/] or CALLI AC, 23 - These return the time of day, in milliseconds right-justified in accumulator AC.

4.3.4.4 CALL AC, [SIXBIT/RUNTIM/] or CALLI AC, 27 - The accumulated running time, in milliseconds, of the job whose number is in accumulator AC, is returned right-justified in accumulator AC. If the job number in AC is zero, the running time of the currently running job is returned. If the job whose number is in AC does not exist, zero is returned.

When one of the specified conditions occurs while the central processor is in user mode, the state of the central processor is Conditioned Into (CONI) location JOBCNI, and the PC is stored in location JOBTPC in the job data area (see Table 3-1). Then control is transferred to the user trap-answering routine specified by the contents of the right half of JOBAPR, after the arithmetic overflow and floating point overflow flags have been cleared. The user program must set up location JOBAPR before executing the CALL AC, [SIXBIT/APRENB/] or CALLI AC, 16. To return control to his interrupted program, the user's trap answering routine must execute a JRST 2, @ JOBTPC to restore the state of the processor.

If the user program does not enable traps, the Monitor sets the PDP-10 processor to ignore arithmetic and floating point overflow, but enables interrupts for the other error conditions in the table above. If the user program produces such an error condition, the Monitor will cause the user job to be stopped and print

ERROR IN JOB n

followed by one of the following appropriate messages:

PC OUT OF BOUNDS AT USER addr
 ILL MEM REF AT USER addr
 NON-EX MEM AT USER addr
 PDL OV AT USER addr

The CONT and CCONT commands will not succeed after such an error.

4.3.3.2 Console-Initiated Traps - Program control can be changed from the user's console by use of the ↑C, START, DDT, and REENTER commands (see Chapter 2).

is printed on the user's console, which is left in Monitor mode.

The CONT and CCONT commands cannot continue the program.

When AC is non-zero, the job is stopped but devices are not released. Instead of printing EXIT and ↑C, only the CR-LF operation is performed and a period is printed on the user's console. The CONT and CCONT commands may be used to continue the program.

4.3.2.4 CALL [SIXBIT/LOGOUT/] or CALLI 17 - All input/output devices are RELEASEd (see Section 4.4.7), and returned with the allocated core and the job number to the Monitor pool. The accumulated running time of the job is printed on the user's console, which is left in Monitor mode. This UVO is not available to user programmers. It is only for use by the LOGOUT CUSP. If a user program executes a LOGOUT UVO, the Monitor will treat it like EXIT (See 4.3.2.3).

4.3.3 Trapping

4.3.3.1 CALL AC, [SIXBIT/APRENB/] or CALLI AC, 16 - APR trapping allows a user to handle any and all traps that occur while his job is running on the central processor, including illegal memory references, non-existent memory references, pushdown list overflow, arithmetic overflow, floating point overflow, and clock flag. To enable for trapping a CALL AC, [SIXBIT/APRENB/] or CALLI AC, 16 is executed, where the AC contains the central processor flags to be tested on interrupts, as defined below:

AC Bit	Trap On
19 200000	pushdown overflow
22 20000	memory protection violation
23 10000	non-existent memory flag
26 1000	clock flag
29 100	floating point overflow
32 10	arithmetic overflow

4.3.6.2 UJEN (Op code 100) - This op code dismisses a user I/O mode interrupt if one is in progress. If the interrupt is from user mode, a JRST 12, instruction can dismiss the interrupt. If the interrupt came from executive mode, however, this operator must be used to dismiss the interrupt. The program must restore all accumulators, and execute UJEN U where user location U contains the program counter as stored by a JSR instruction when the interrupt occurred.

4.3.6.3 CALL AC, [SIXBIT/SWITCH/] or CALLI AC, 20 - These return the contents of the central processor data switches in AC. Caution must be exercised in using the data switches since they are not an allocated resource and are always available to all users.

4.3.6.4 CALL AC, [SIXBIT/SETNAM/] or CALLI AC, 43 - These are used by the LOADER. The contents of AC contain a left-justified SIXBIT program name, which is stored in a Monitor job table. The information in the table is used by the SYSTAT CUSP (See JBTPRG table under GETTAB UUU 4.3.5.6).

4.3.7 Segment Handling

4.3.7.1 CALL AC, [SIXBIT/REMAP/] or CALLI AC, 37 - These take the top part of a low segment and remap it into the high segment. The previous high segment (if any) will be removed from the user's addressing space. The new low segment will be the previous low segment minus the amount remapped.

The call is: MOVEI AC, Desired highest adr in low segment
 CALL AC, [SIXBIT/REMAP/] ;or CALLI AC, 37
 error return
 normal return

The amount remapped must be a multiple of 1K decimal words. To insure this, the Monitor will perform the inclusive OR function of 1777 and the user's request. If the argument exceeds

the length of the low segment, remapping will not take place, the high segment will remain unchanged in the user's addressing space, and the error return will be taken. The error return will also be taken if the system does not have a two-register capability. The contents of AC are unchanged. The contents of JOBREL (see Job Data area, Chapter 3) are set to the new highest legal user address in the low segment. The RH of JOBHRL will be set to the highest legal user address in the high segment (401777 or greater or 0). The hardware relocation will be changed and the user-mode write protect bit will be set.

This UVO is used by the LOADER to load reentrant programs which make use of all of physical core. Otherwise, the LOADER might exceed core in assigning more core and moving the data from the low to the high segment with a BLT instruction. The GET command also uses this UVO to do I/O into the low segment instead of the high segment.

4.3.7.2 CALL AC, [SIXBIT/RUN/] or CALLI AC, 35 - These have been implemented so that programs can transfer control to one another. Both the low and high segments of the user's addressing space are replaced with the program being called.

The call is:

```

MOVSI AC, Starting address increment
HRRI AC, Adr of six-word arg. block
CALL AC, [SIXBIT/RUN/] or CALLI AC, 35
error return (unless HALT in LH)
[normal return is not here, but to starting
address plus increment of new program]

```

The arguments contained in the six-word block are:

```

E: SIXBIT/logical device name/
   SIXBIT/filename/           ;for either or both high
                               and low files

```

SIXBIT/ext.for low file/	;if LH = 0, .LOW is assumed if high segment exists, .SAV is assumed if high segment does not exist.
0	
XWD proj. no., prog. no.	;if = 0, use current user's proj,prog
XWD 0, optional core assignment	;RH = New highest user address to be assigned to low segment. LH is ignored rather than setting high segment.

Usually a user program will specify only the first two words and set the others to zero. The RUN UWO destroys the contents of all of the user's ACs and releases all the user's I/O channels. Therefore, arguments or devices cannot be passed to the next program.

Programs on the system library (CUSPs) should be called by using device SYS with a zero project-programmer number instead of device DSK with the project-programmer number 1,1. The extension should also be 0 so that the calling user program does not need to know if the called CUSP is reentrant or not.

The LH of AC is added to and stored in the starting address (JOB SA) of the new program before control is transferred to it. ↑C followed by the START command will restart the program at the same location as specified by the RUN UWO, so that the user can start the current CUSP over again. The user is considered to be meddling with the program if the LH of AC is not 0 or 1. (See Section 4.6).

Programs which accept commands from a Teletype or a file, depending on how they were started, do so as controlled by the program calling the RUN UWO. The following convention is used with all of Digital's standard CUSPs: 0 in LH of AC means type an asterisk and accept commands from the Teletype. 1 means accept commands from a command file, if it exists; if not type an asterisk

and accept commands from the Teletype. The convention for naming CUSP command files is that the filename be of the form

```
###III.TMP
```

where III are the first three (or fewer if three do not exist) characters of the name of the CUSP doing the LOOKUP and ### is the decimal character expansion (with leading zeroes) of the binary job number. The job number is included to allow a user to run two or more jobs under the same project-programmer number. For example,

```
009PIP.TMP
039MAC.TMP
```

Decimal numbers are used so that a user listing his directory can see the same number as the PJOB command types. These command files are temporary and are, therefore, deleted by the LOGOUT CUSP. (See LOGOUT command in Chapter 2.)

The RUN UVO can give an error return with one of 13 error codes in AC if any errors are detected. Thus, the user program may attempt to recover from the error and/or give the user a more informative message on how to proceed. Some user programs do not go to the bother of including error recovery code. The Monitor detects this and does not give an error return if the LH of the error return location is a HALT instruction. If this is the case, the Monitor simply prints its standard error message for that type of error and returns the user's console to monitor mode. This optional error recovery procedure also allows a user program to analyze the error code received and then execute a second RUN UVO with a HALT if the error code indicates an error for which the Monitor message is sufficiently informative or one from which the user program cannot recover.

The error codes are an extension of the LOOKUP, ENTER, and RENAME UVO error codes and are defined in the S.MAC Monitor

file.

LOOKUP, ENTER, RENAME, RUN, GETSEG UWO Error Codes

FNFEER	0	File not found
IPPEER	1	Incorrect proj-prog no.
PRTEER	2	Protection failure or directory full on DTA
FBMEER	3	File being modified
AEFEER	4 ¹	Already existing file
NLEEER	5 ¹	Neither LOOKUP or ENTER
TRNEER	6	Transmission error
NSFEER	7	Not a saved file
NECEER	10	Not enough core
DNAEER	11	Device not available
NSDEER	12	No such device
ILUEER	13 ¹	Illegal UWO (GETSEG UWO on a one-register machine)

The Monitor does not attempt an error return to a user program after the high or low segment containing the RUN UWO has been overlaid.

In order to successfully program the RUN UWO for all size systems and for all CUSPs whose size is not known at the time the RUN UWO is coded, it is necessary to understand the sequence of operations it initiates. Assume that the job executing the RUN UWO has both a low and a high segment. (It can be executed from either segment; however, fewer errors can be returned to the user if it is executed from the high segment.)

The sequence of operations for the RUN UWO is as follows.

Does a high segment already exist with desired name?
 If yes, go to 30.
 INIT and LOOKUP file name .SHR. If not found, go to 10.
 Read high file into top of low segment by extending it. (Here the old low segment and new high segment and old high segment together may not exceed the capacity of core.)
 REMAP the top of low segment replacing old high segment in logical addressing space.
 If high segment is sharable (.SHR) store its name so others can share it.
 Always go to 40 or return to user if GETSEG UWO.

10. LOOKUP file name .HGH. If not found, go to 41 or error return to user if GETSEG UWO.

¹Not possible on RUN UWO

Read high file into top of low segment by extending it. (Here again the old low segment and new high segment and old high segment together may not exceed the capacity of core.)
 Check for I/O errors. If any, error return to user unless HALT in LH of return.
 Go to 41.

30. Remove old high segment, if any, from logical addressing space.
 Place the sharable segment in user's logical addressing space. Go to 40 or return to user if GETSEG UUU.
35. Remove old high segment, if any, from logical addressing space.
 (Go to 41)
40. Copy Vestigial Job Data area into Job Data area.
 Does the new high segment have a low file (LH JOBCOR>137)?
 If not, go to 45.
41. LOOKUP filename .SAV or .LOW or user specified extension. Error if not found. Return to user if there is no HALT in LH of error return, provided that if the CALL is from the high segment it is still the original high segment. Otherwise, the Monitor prints the error message

ERROR IN JOB n
 filename NOT FOUND, UUU AT USER addr
 and stops the job.
 Reassign low segment core according to size of file or user specified core argument, whichever is larger. Previous low segment is overlaid.
 Read low file into beginning of low segment.
 Check for I/O errors. If there is an error, print error message and do not return to user. If no errors, perform START.

45. Reassign low segment core according to larger of user's core argument or argument when file saved (RH JOBCOR).

NOTE

In order to always be guaranteed of handling the most number of errors, the cautious user should remove his high segment from high logical addressing space (use core UUU with a one in LH of AC). The error handling code should be put in the low segment along with the RUN UUU and the size of the low segment reduced to 1K. An even better idea would be to have the error handling code be written once and put in a seldom used (probably non-sharable) high segment which could be gotten in high segment using GETSEG UUU (see below) when an error return occurs to low segment on a RUN UUU.

4.3.7.3 CALL AC, [SIXBIT/GETSEG/] or CALLI AC, 40 - These have been implemented so that a high segment can be initialized from a file or shared segment without affecting the low segment. It is used for shared data segments and shared program overlays. It is also used for run-time routines such as FORTRAN or COBOL operating systems. These programmed operators work exactly like the RUN UWO with the following exceptions.

- a) No attempt is made to read a low file.
- b) The only change that is made to the low segment of the Job Data area is to both halves of JOBHRL.
- c) If an error occurs, control is returned to the location of the error return, unless the left half of the location contains a HALT instruction.
- d) On a normal return, control is returned to two locations following the UWO, whether it is called from low or high segment. It should be called from low segment unless the normal return coincides with the starting address of the new high segment.
- e) User channels 1 through 17 are not released so the GETSEG UWO can be used for program overlays, such as the COBOL compiler. Channel 0 is released because it is used by the UWO.

See steps 1 through 31 of the RUN UWO description for details of the operation of the GETSEG UWO.

4.3.7.4 CALL AC, [SIXBIT/SPY/] or CALLI AC, 42 - These are used for efficient examination of the Monitor during time sharing. Any number of K of physical core is placed into the user's logical high segment. This amount cannot be saved (no error return if tried), cannot be increased or decreased by the CORE UWO (error return taken), or cannot have the user-mode write protect bit set (error return taken).

The call is:

```

MOVEI AC, Highest physical core location
          desired
CALL AC, [SIXBIT/SPY/]          ;or CALLI AC, 42
error, return
normal return

```

Any program that is written to use the SPY UO should try the PEEK UO if it receives an error return. Some installations may restrict use of the SPY UO to certain privileged users (e.g., project 1 only).

4.4 INPUT/OUTPUT PROGRAMMING

All user input/output operations are controlled by the use of Monitor programmed operators. These are device independent, in the sense that if an operator is not pertinent to a given device, the operator is treated as a no-operation code. For example, a rewind directed to a line printer does nothing. Devices are referenced by logical names or physical names (see Chapter 2), and the characteristics of a device can be obtained from the Monitor. Properly used, these systems characteristics permit the programmer to delay the device specification for his program from program-generation until program-run time. I/O is accomplished by associating a device, a file, and a ring buffer or command list with one of a user's I/O channels.

4.4.1 File

A file is an ordered set of data on a peripheral device. Its extent on input is determined by an end-of-file condition dependent on the device. For instance, a file is terminated by reading an end-of-file gap from magnetic tape, by an end-of-file card from a card reader, or by depressing the end-of-file switch on a card reader (see Chapter 5). The extent of a file on output

is determined by the amount of information written by the OUT or OUTPUT programmed operators up through and including the next CLOSE or RELEAS operator.

4.4.1.1 Device - To specify a file, it is necessary to specify the device from which the file is to be read or onto which the file is to be written. This specification is made by an argument of the INIT or OPEN programmed operators. Devices are separated into two categories--those with no filename directory, and those with one or more filename directories.

a) Non-directory Devices - For non-directory devices, e.g., card reader, line printer, paper tape reader and punch, and user console, the only file specification required is the device name. All other file specifiers, if given, are ignored by the Monitor. Magnetic tape, which is also a non-directory device, requires, in addition to the name, that the tape be properly positioned. Even though LOOKUP is not required to read and ENTER is not required to write, it is always advisable to use them so that a directory device may be substituted for a non-directory device at run time (using the Monitor command, ASSIGN). Only in this way can user programs be truly device independent.

b) Directory Devices - For directory devices, e.g., DECTape and disk, files are addressable by name. If the device has a single file directory, e.g., DECTape, the device name and filename are sufficient information to determine a file. If the device has multiple file directories, e.g., disk, the name of the file directory must also be specified. These names are specified as arguments to the LOOKUP, ENTER, and RENAME programmed operators.

4.4.1.2 Data Modes - Data transmissions are either unbuffered or buffered. (Unbuffered mode is sometimes referred to as dump mode.) The mode of transmission is specified by a 4-bit argument to the INIT, OPEN, or SETSTS programmed operators. Table 4-3 and Table 4-4 summarize the data modes.

Table 4-3
Buffered Data Modes

Octal Code	Mnemonic	Meaning
0	A	ASCII. 7-bit characters packed left justified, five characters per word.
1	AL	ASCII line. Same as 0, except that the buffer is terminated by a FORM, VT, LINE-FEED or ALTMODE character.
2-7		Unused.
10	I	Image. A device dependent mode. The buffer is filled with data exactly as supplied by the device.
11-12		Unused.
13	IB	Image binary. 36-bit bytes. This mode is similar to binary mode, except that no automatic formatting or checksumming is done by the Monitor.
14	B	Binary. 36-bit byte. This is blocked format consisting of a word count, n (the right half of the first data word of the buffer), followed by n 36-bit data words. Checksum for cards and paper tape.

Table 4-4
Unbuffered Data Modes

15	ID	Image Dump. A device dependent dump mode.
16	DR	Dump as records without core buffering. Data is transmitted between any contiguous blocks of core and one or more standard length records on the device for each command word in the command list.
17	D	Dump one record without core buffering. Data is transmitted between any contiguous block of core and exactly one record of arbitrary length on the device for each command word in the command list.

a) Unbuffered Data Modes - Data modes 15, 16 and 17 utilize a command list to specify areas in the user's allocated core to be read or written. The effective address of the IN, INPUT, OUT, and OUTPUT programmed operators points to the first word of the command list. Three types of entries may occur in the command list.

- 1) IOWD *n*, *loc* - Causes *n* words from *loc* through *loc+n-1* to be transmitted. The next command is obtained from the next location following the IOWD. The assembler pseudo-op IOWD generates XWD *-n*, *loc-1*.
- 2) XWD 0, *y* - Causes the next command to be taken from location *y*. Referred to as a GOTO word.
- 3) 0 - Terminates the command list.

The Monitor does not return program control to the user until the command list has been completely processed. If an illegal address is encountered while processing the list, the job is stopped and the Monitor prints

ADDRESS CHECK AT USER *addr*

on the user's console, leaving the console in Monitor mode.

b) Buffered Data Modes - Data modes 0, 1, 10, 13, and 14 utilize a ring of buffers in the user area and the priority interrupt system to permit the user to overlap computation with his data transmission. Core memory in the user's area serves as an intermediate buffer between the user's program and the device. A ring of buffers consists of a 3-word header block for bookkeeping and a data storage area subdivided into one or more individual buffers linked together to form a ring. During input operations, the Monitor fills a buffer, makes the buffer available to the user's program, advances to the next buffer in the ring and fills it if it is free. The user's program follows along behind, emptying the next buffer if it is full, or waiting for the next buffer to fill.

During output operations, the user's program and the Monitor exchange roles, the user filling the buffers and the Monitor emptying them.

- 1) Buffer Structure - A ring of buffers consists of a 3-word header block and a data storage area subdivided into one or more individual buffers linked together to form a ring. The ring buffer layout is shown in Figure 4-1, and explained in the paragraphs which follow.
 - (a) Buffer Header Block - The location of the 3-word buffer header block is specified by an argument of the INIT and OPEN operators. Information is stored in the header by the Monitor in response to user execution of Monitor programmed operators. The user's program finds all the information required to fill and empty buffers in the header. Bit position 0 of the first word of the header is a flag which, if 1, means that no input or output has occurred for this ring of buffers. The right half of the first word is the address of the second word of the buffer currently in use by the user's program. The second word of the header contains a byte pointer to the current byte in the current buffer. The byte size is determined by the data mode. The third word of the header contains the number of bytes remaining in the buffer. A program may not use a single buffer header for both input and output, nor may a single buffer header be used for more than one I/O function at a time.
 - (b) Buffer Data Storage Area - The buffer data storage area is established by the INBUF and OUTBUF operators, or, if none exists when the first IN, INPUT, OUT, or OUTPUT operator is executed, a 2-buffer ring is set up. The effective address of the INBUF and OUTBUF operators specifies the number of buffers in the ring. The location of the buffer storage area is specified by the contents of the right half of JOBFF in the user's Job Data area. The Monitor updates JOBFF to point to the first location past the storage area.

All buffers in the ring are identical in structure. As Figure 4-2 shows, the right half of the first word contains the file status at the time that the Monitor advanced to the next buffer in the ring. Bit 0 of the second word of a buffer, called the use bit, is a flag that indicates

whether the buffer contains active data. This bit is set to 1 by the Monitor when the buffer is full on input or being emptied on output, and set to 0 when the buffer is empty on output or is being filled on input. The use bit prevents the Monitor and the user's program from interfering with each other by attempting to use the same buffer simultaneously. Buffers are advanced by using the UWO's and not by the user's program. The use bit in each buffer should never be changed by the user's program except by means of the UWO's. Bits 1 through 17 of the second word of the buffer contain the size of the data area of the buffer which immediately follows the second word. The size of this data area depends on the device. The right half of the first word of the data area of the buffer, i.e.,

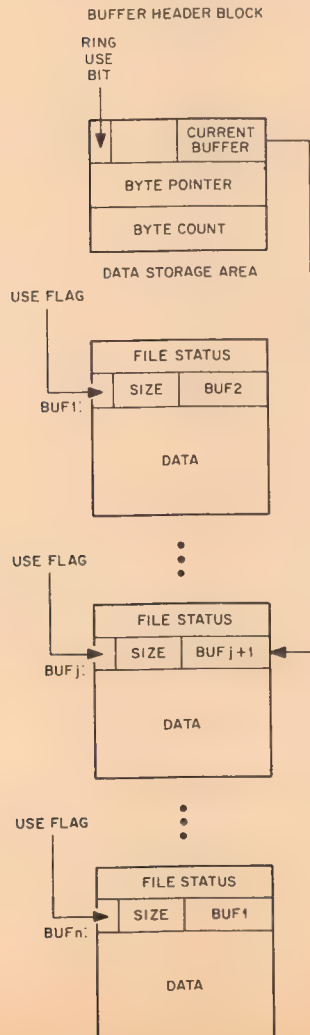


Figure 4-1
User's Ring of Buffers

the third word of the buffer, is reserved for a count of the number of words (excluding itself) that actually contain data. The left half of this word is reserved for other bookkeeping purposes, depending on the particular device and the data mode.

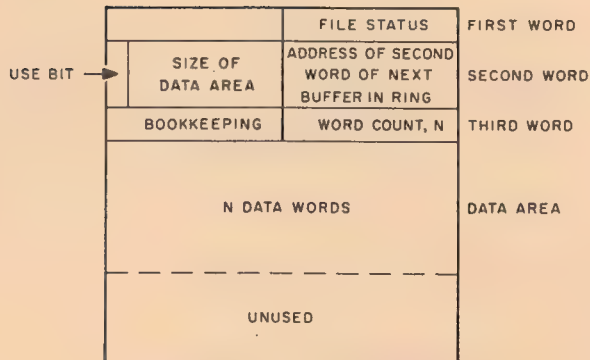


Figure 4-2

Detailed Diagram of Individual Buffer

4.4.1.3 File Status - The file status is a set of 18 bits (right half word), which reflects the current state of a file transmission. The initial status is a parameter of the INIT and OPEN operators. Thereafter, bits are set by the Monitor, and may be tested and reset by the user via Monitor programmed operators. Table 4-5 defines the file status bits. All bits, except the end-of-file bit, are set immediately by the Monitor as the conditions occur, rather than being associated with the buffer that the user is currently working on. However, the file status is stored with each buffer so that the user can determine which bufferful produced an error. A more thorough description of bits 18 through 29 is given in Chapter 5.

Table 4-5

File Status

Bit	Meaning
18	Improper mode, e.g., attempt to write on a write-locked tape.
19	Device detected error, other than hardware checksum or parity. Checksum, and/or parity error detected by hardware and/or software.
20	Data error, e.g., a computed checksum failed or invalid data was received.
21	Block too large. A block of data from a device is too large to fit in a buffer, or a block number is too large.
22	End of file.
23	Device is actively transmitting or receiving data.
24-29	Device dependent parameters. (See Chapter 5.)
30	Synchronous input. Stop the device after each buffer is filled.
31	Forces the Monitor to use the word count in the first data word of the buffer (output only). The Monitor normally computes the word count from the byte pointer in the buffer header.
32-35	Data mode. See Table 4-3 and Table 4-4.

4.4.2 Initialization

4.4.2.1 Job Initialization - The Monitor programmed operator

CALL [SIXBIT/RESET/] or CALLI 0

should normally be the first instruction in each user program.

It immediately stops all input/output transmissions on all devices without waiting for the devices to become inactive. All device allocations made by the INIT and OPEN operators are cleared, and, unless the devices have been assigned by the ASSIGN command

(see Chapter 2), the devices are returned to the Monitor facilities pool. The content of the left half of JOBSA (program break) is stored in the right half of JOBFF so that the user buffer area is reclaimed if the program is starting over. The left half of JOBFF is cleared. Any files which have not been closed are deleted on disk. Any older version having the same filename remains. The user-mode write-protect bit is automatically set if a high segment exists, whether it is sharable or not, so that a program cannot inadvertently store into the high segment.

4.4.2.2 Device Initialization

OPEN D,SPEC	INIT D,STATUS
error return	SIXBIT/ldev/
normal return	XWD OBUF,IBUF
.	error return
.	normal return
.	
SPEC:EXP STATUS	
SIXBIT/ldev/	
XWD OBUF,IBUF	

The OPEN (operation code 050) and INIT (operation code 041) programmed operators initialize a file by specifying a device, ldev, and initial file status, STATUS, and the location of the input and output buffer headers.

a) Data Channel - OPEN and INIT establish a correspondence between the device, ldev, and a 4-bit data channel number, D. Most of the other input/output operators require this channel number as an argument. If a device is already assigned to channel D, it is released. (See RELEAS in this chapter.) The device name, ldev, is either a logical or physical name, with logical names taking precedence over physical names. (See ASSIGN command, Chapter 2.) If the device, ldev, is not the system device, SYS, and is allocated to another job or does not exist, the error return is taken. If the device is the system device, SYS, the job is

put into a system device wait queue, and will continue running when SYS becomes available.

b) Initial File Status - The file status, including the data mode, is set to the value of the symbol STATUS. If the data mode is not legal (see Chapter 5) for the specified device, the job is stopped and the Monitor prints

ILL DEVICE DATA MODE FOR DEVICE dev AT USER addr,
where dev is the physical name of the device and addr is the location of the OPEN or INIT operator, on the user's console and leaves the console in Monitor mode.

c) Buffer Header - Symbols OBUF and IBUF, if non-zero, specify the location of the first word of the 3-word buffer header for output and input, respectively. Only those headers which are to be used need to be specified. For instance, the output header need not be specified, if only input is to be done. Also, data modes 15, 16, and 17 require no header. If either of the buffer headers or the 3-word block starting at location SPEC lies outside the user's allocated core area,¹ the job is stopped and the Monitor prints

ILLEGAL UO AT USER addr

(addr is the address of the OPEN or INIT operator) on the user's console, leaving the console in Monitor mode.

The first and third words of the buffer header are set to zero. The left half of the second word is set up with the byte pointer size field in bits 6 through 11 for the selected device-data mode combination.

¹Buffer headers may not be in the user's AC's. However, they may be in locations above JOBPFI. (See Table 3.1)

4.4.2.3 Buffer Initialization - Buffer data storage areas may be established by the INBUF and OUTBUF programmed operators, or by the first IN, INPUT, OUT, or OUTPUT operator, if none exists at that time, or the user may set up his own buffer data storage area.

a) Monitor Generated Buffers - Each device has associated with it a standard buffer size (see Chapter 5). The Monitor programmed operators INBUF D, n (operation code 064) and OUTBUF D,n (operation code 065) set up a ring of n standard size buffers associated with the input and output buffer headers, respectively, specified by the last OPEN or INIT operator on data channel D. If no OPEN or INIT operator has been performed on channel D, the Monitor stops the job and prints

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the INBUF or OUTBUF operator) on the user's console, leaving the console in Monitor mode.

The storage space for the ring is taken from successive locations, beginning with the location specified in the right half of JOBBF. This is set to the program break, which is the first free location above the program area, by RESET. If there is insufficient space to set up the ring, the Monitor will automatically attempt to expand the user's core allocation by 1K. If this fails, the Monitor stops the job and prints

ADDRESS CHECK FOR DEVICE ldev AT USER addr

(ldev is the physical name of the device associated with channel D and addr is the location of the INBUF or OUTBUF operator) on the user's console, leaving the console in Monitor mode.

The ring is set up by setting the second word of each buffer with a zero use bit, the appropriate data area size, and the link to the next buffer. The first word of the buffer header is set with a 1 in the ring use bit, and the right half contains the

address of the second word of the first buffer.

b) User Generated Buffers - The following code illustrates an alternative to the use of the INBUF programmed operator. Analogous code may replace OUTBUF. This user code operates similarly to INBUF. SIZE must be set equal to the greatest number of data words expected in one physical record.

```

GO:          INIT 1, 0                ;INITIALIZE ASCII MODE
             SIXBIT/MTA0/           ;MAGNETIC TAPE UNIT 0
             XWD 0, MAGBUF           ;INPUT ONLY
             JRST NOTAVL
             MOVE 0, [XWD 400000,BUF1+1] ;THE 400000 IN THE LEFT HALF
                                           ;MEANS THE BUFFER WAS NEVER
                                           ;REFERENCED.

             MOVEM 0, MAGBUF
             MOVE 0, [POINT BYTSIZ,0,35] ;SET UP NON-STANDARD BYTE
                                           ;SIZE

             MOVEM 0, MAGBUF+1
             JRST CONTIN              ;GO BACK TO MAIN SEQUENCE
MAGBUF:     BLOCK 3                  ;SPACE FOR BUFFER HEADER
BUF1:       0                        ;BUFFER 1, 1ST WORD UNUSED
             XWD SIZE+2,BUF2+1       ;LEFT HALF CONTAINS BUFFER
                                           ;SIZE, RIGHT HALF HAS
                                           ;ADDRESS OF NEXT BUFFER
             BLOCK SIZE+1            ;SPACE FOR DATA, 1ST WORD
                                           ;RECEIVES WORD-COUNT.  THUS
                                           ;ONE MORE WORD IS RESERVED
                                           ;THAN IS REQUIRED FOR DATA
                                           ;ALONE
BUF2:       0                        ;SECOND BUFFER
             XWD SIZE+2,BUF3+1
             BLOCK SIZE+1
BUF3:       0                        ;THIRD BUFFER
             XWD SIZE+2,BUF1+1       ;RIGHT HALF CLOSSES THE RING
             BLOCK SIZE+1

```

4.4.2.4 File Selection (LOOKUP and ENTER) - The LOOKUP (operation code 076) and ENTER (operation code 077) programmed operators select a file for input and output, respectively. Although these operators are not necessary for non-directory devices, it is good programming practice to always use them so that directory devices may be substituted at run time. (See ASSIGN, Chapter 2.)

```

a) LOOKUP D,E
   error return.
   normal return
   .
   .
E: SIXBIT/file/           ;filename, 1 to 6 characters.
   SIXBIT/ext/           ;filename extension, 0 to 3
                           ;characters.
   0
   XWD project number, programmer number,

```

LOOKUP selects a file for input on channel D. If no device has been associated with channel D by an INIT or OPEN UWO, the Monitor prints

```
I/O TO UNASSIGNED CHANNEL AT USER addr
```

and returns the user's console to Monitor mode. If the input side of channel D is not closed (see CLOSE, in this chapter), it is now closed. The output side of channel D is not affected. If the device associated with channel D does not have a directory, the normal return is now taken. If the device has multiple directories, e.g., disk, the Monitor searches the master file directory of the device for the user's file directory whose number is in location E+3 and whose extension is UFD. If E+3 contains zero, the project-programmer pair of the current job is used as the name of the user's file directory. If this file is not found in the master file directory, 1 is stored in bits 33 through 35 of location E+1 and the error return is taken.

The user's file directory or the device directory in the case of a single-directory device (e.g., DEctape) is searched for the file whose name is in location E and whose extension is in the left half of location E+1. If the file is not found, 0 is stored in the right half of E+1 and the error return is taken. If the device is a multiple-directory device (e.g., disk) and the file is found, but is read protected (see File Protection in this chapter),

2 is stored in the right half of location E+1 and the error return is taken. Otherwise, location E+1 through E+3 are filled by the Monitor with the following data concerning the file, and the normal return is taken.

- 1) The left half of location E+1 remains set to the filename extension.
- 2) If the device is a multiple-directory device, bits 24 through 35 of location E+1 are set to the date (in the format of DAYTIME programmed operator) that the file was last referenced.

If the device is a single-directory device, the right half of location E+1 is set to the device block number of the first block of the file.

- 3) If the device is a multiple-directory device, bits 0 through 8 of location E+2 are set to the file protection. (See "File Protection," this chapter.)
- 4) Bits 9 through 12 of location E+2 are set to the data mode in which the file was written.
- 5) Bits 13 through 23 of location E+2 are set to the time, in minutes, and bits 24 through 35 of location E+2 are set to the date (in the format of the DAYTIME programmed operator) of the file's creation, i.e., of the last ENTER or RENAME programmed operator.
- 6) If the device is a multiple-directory device, the left half of location E+3 is set to the negative of the number of words in the file, and the right half is unchanged. If the file contains more than 2^{17} words, then the left half contains the positive number of 128-word blocks in the file.

If the device is a single-directory device, location E+3 is used only for SAVED files (see Chapter 3), and contains the IOWD of the core image, i.e., the left half is the negative word length of the file and the right half is the core address of the first word minus 1.

```

b) ENTER D,E
   error return
   normal return
   .
   .
E: SIXBIT/file/           ;filename, 1 through 6
                               ;characters.
   SIXBIT/ext/           ;filename, extension, 0
                               ;through 3 characters.
   EXP<TIME>B23+DATE
   XWD project number, programmer number.

```

ENTER selects a file for output on channel D. If no device has been associated with channel D by an INIT or OPEN UUO, the Monitor prints

```
I/O TO UNASSIGNED CHANNEL AT USER addr
```

and returns the user's console to Monitor mode. If the output side of channel D is not closed (see CLOSE in this chapter), it is now closed. The input side of channel D is not affected. If the device does not have a directory, the normal return is now taken.

If the device has multiple directories, e.g., disk, the Monitor searches the master file directory of the device for the user's file directory whose name is in location E+3 and whose extension is UFD. If E+3 contains 0, the project-programmer pair of the current job is used as the name of the user's file directory. If this file is not found in the master file directory, 1 is stored in bits 33 through 35 of location E+1, and the error return is taken. Since a null filename is illegal, if the filename in location E is 0, 0 is stored in bits 33 through 35 of location E+1, and the error return is taken. The user's file directory, or the device file directory in the case of a single-directory device, such as DECTape, is searched for the file whose name is in location E and whose extension is in the left half of location E+1.

If the device is a multiple-directory device and the file is found but is being written or renamed, 3 is stored in bits 33

through 35 of location E+1, and the error return is taken. If the file is write protected (See "File Protection", this chapter), 2 is stored in bits 33 through 35 of location E+1, and the error return is taken.

If the file is found, and is not being written or renamed and is not write protected, then the file is deleted, and the storage space on the device is recovered.

On disk, this deletion of the previous version does not occur until output CLOSE time. Consequently, if the new file is aborted when partially written, the old version remains. On DECTape, the deletion must occur immediately upon ENTER to insure that space is available for writing the new version of the file.

The Monitor then makes the file entry by recording the following information concerning the file and takes the normal return.

- a) The filename is taken from location E.
- b) The filename extension is taken from the left half of location E+1.
- c) If the device is a multiple-directory device, then
 - 1) the current date is taken as the date of last reference,
 - 2) the file protection key is set to 055 (see "File Protection," this chapter),
 - 3) the current data mode is taken as the mode in which the file is to be written,
 - 4) the project number of the current job is taken as the file owner's project number, and
 - 5) if bits 13 through 35 of location E+2 are non-zero, bits 13 through 23 are taken as the time of creation, in minutes, and bits 24 through 35 are taken as the date of creation (in the format of the DAYTIME programmed operator) of the file. Otherwise, the current time and date are used.

If the device is a single-directory device, and if bits 24 through 35 of location E+2 are non-zero, they are taken as

the date of creation; otherwise, the current date is used.

4.4.2.5 File Protection and the RENAME Operator - File protection on non-directory and single-directory devices is obtained by use of the ASSIGN command (see Chapter 2). Multiple-directory devices have a master file directory for the device which contains entries for each user's file directory. File selection (see LOOKUP and ENTER in this chapter) requires specification of the name of a user's file directory and a filename within that directory. Since this permits each user to access all files on the device, a file protection scheme to prevent unauthorized references is necessary. For file protection purposes users are divided into three categories:

- a) The file owner is the user whose programmer number is the same as that in the NAME field of the user's file directory in which the file is entered. (Some installations may modify the Monitor to require both project and programmer numbers to match.)
- b) Project members are users whose project number is the same as that of the file owner.
- c) All other users.

There are three types of protection against each of the three categories of users.

- a) Protection-protection - the protection cannot be altered.

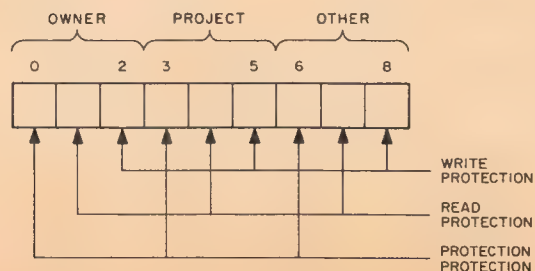


Figure 4-3 File Protection Key

- b) Read protection - the file may not be read.
- c) Write protection - the file may not be modified.

The file protection key, shown in the foregoing figure, is a set of nine bits which specify the three types of protection for each category of user. (See 5.8.2.4) When a file is created by an ENTER programmed operator, the file protection key is set to 055, indicating that the file is protection-protected and write-protected against all users except the owner. The protection key is returned by the LOOKUP D, E programmed operator in bits 0 through 8 of location E+2. It can be changed by the RENAME programmed operator. The owner's protection-protection and read-protection bits are ignored by the Monitor, thereby preventing a file from becoming inaccessible to everyone. Moreover, the owner protection-protection bit has been taken over to specify that a user wishes to protect his file from deletion when he logs off the system. This feature is handled completely by the LOGOUT CUSP.

```
RENAME D,E
error return
normal return
```

```
E: SIXBIT/file/      ;filename, 1 through 6 characters.
SIXBIT/ext/        ;filename extension, 0 through 3 characters.
EXP<PROT>B8+<TIME>B23+DATE
XWD project number, programmer number.
```

The RENAME programmed operator (operation code 055) is used to alter the filename, filename extension, and file protection key or delete a file associated with channel D on a directory device.

If no device is associated with channel D, the Monitor prints I/O TO UNASSIGNED CHANNEL AT USER addr and returns the user's console to Monitor mode. If the device is a nondirectory device, the normal return is taken. If no file is selected on channel D, 5 is stored in bits 33 through 35 of location E+1, and the error return is taken.

If the device has multiple directories, e.g., disk, the Monitor searches the master file directory of the device for the user's file directory whose name is in location E+3 and whose extension is UFD. If E+3 contains 0, the project-programmer pair of the current job is used as the name of the user's file directory. If this file is not found in the master file directory, 1 is stored in bits 33 through 35 of location E+1, and the error return is taken. The user's file directory, or the device file directory in the case of a single-directory device, is searched for the file currently selected on channel D. If the file is not found, 0 is stored in bits 33 through 35 of location E+1, and the error return is taken.

If the device is a multiple-directory device and the file is found, but is being written or renamed, 3 is stored in bits 33 through 35 of location E+1, and the error return is taken. If the file is owner write-protected or if the protection key is being modified, i.e., bits 0 through 8 of location E+2 differ from the current protection key, and the file is owner protection-protected, 2 is stored in bits 33 through 35 of location E+1, and the error return is taken.

If the new filename in location E is 0, the file is deleted, or marked for deletion, after all read references are completed, and the normal return is taken. If the filename in location E and the filename extension in the left half of location E+1 are the same as the current filename and filename extension, respectively, the protection key is set to the contents of bits 0 through 8 of location E+2, and the normal return is taken.

If the new filename in location E and/or the filename extension in the left half of location E+1 differ from the current filename and/or filename extension, the user's file directory (or the device directory) is searched for the new filename and extension, as

in LOOKUP. If a match is found, 4 is stored in bits 33 through 35 of location E+1, and the error return is taken. If no match is found, the file is changed to the new name in location E, the file-name extension is changed to the new filename extension in the left half of location E+1, the protection key is set to the contents of bits 0 through 8 of location E+2, the access date is set to the current date, and the normal return is taken.

4.4.2.6 Examples

General Device Initialization

```

INIDEV:  0:           ;JSR HERE
          INIT 3, 14   ;BINARY MODE, CHANNEL 3
          SIXBIT/DTA5/ ;DEVICE DECTAPE UNIT 5
          XWD OBUF,IBUF ;BOTH INPUT AND OUTPUT
          JRST NOTAVL  ;WHERE TO GO IF DTA5 IS BUSY

;FROM HERE DOWN IS OPTIONAL DEPENDING ON THE DEVICE AND PROGRAM
;REQUIREMENTS

          MOVE 0, JOBFF
          MOVEM 0, SV JBFF ;SAVE THE FIRST ADDRESS OF THE BUFFER
                               ;RING IN CASE THE SPACE MUST BE
                               ;RECLAIMED

          INBUF 3,4      ;SET UP 4 INPUT BUFFERS
          OUTBUF 3,1     ;SET UP 1 OUTPUT BUFFER
          LOOKUP 3, INNAM ;INITIALIZE AN INPUT FILE
          JRST NOTFND    ;WHERE TO GO IF THE INPUT FILE NAME IS
                               ;NOT IN THE DIRECTORY

          ENTER 3, OUTNAME ;INITIALIZE AN OUTPUT FILE
          JRST NOROOM     ;WHERE TO GO IF THERE IS NO ROOM IN
                               ;THE DIRECTORY FOR A NEW FILE NAME

          JRST @INIDEV    ;RETURN TO MAIN SEQUENCE

OBUF     BLOCK 3         ;SPACE FOR OUTPUT BUFFER HEADER
IBUF     BLOCK 3         ;SPACE FOR INPUT BUFFER HEADER
INNAM:   SIXBIT/NAME/    ;FILE NAME
          SIXBIT/EXT/    ;FILE NAME EXTENSION (OPTIONALLY 0),
                               ;RIGHT HALF WORD RECEIVES THE
                               ;FIRST BLOCK NUMBER
          0              ;RECEIVES THE DATE
          0              ;UNUSED FOR NONDUMP I/O
OUTNAM:  SIXBIT/NAME/    ;SAME INFORMATION AS IN INNAME
          SIXBIT/EXT/
          0
          0

```

4.4.3 Data Transmission

The programmed operators

INPUT D,E	and	IN D,E
		normal return
		error return

transmit data from the file selected on channel D to the user's core area. The programmed operators

OUTPUT D,E	and	OUT D,E
		normal return
		error return

transmit data from the user's core area to the file selected on channel D.

If no OPEN or INIT operator has been performed on channel D, the Monitor stops the job and prints

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the IN, INPUT, OUT, or OUTPUT programmed operator) on the user's console leaving the console in Monitor mode. If the device is a multiple-directory device and no file is selected on channel D, bit 18 of the file status is set to 1, and control returns to the user's program. Control always retruns to the location immediately following an INPUT (operation code 066) and an OUTPUT (operation code 067). A check of the file status for end-of-file and error conditions must then be made by another programmed operator. Control returns to the location immediately following an IN (operation code 056) and an OUT (operation code 057), if no end-of-file or error condition exists, i.e., if bits 18 through 22 of the file status are all 0. Otherwise, control returns to the second location following the IN or OUT. Note that IN and OUT UUU's are the only ones in which the error return is a skip and the normal return is not a skip.

4.4.3.1 Unbuffered (Dump) Modes - In data modes 15, 16, and 17, the effective address E of the INPUT, IN, OUTPUT, and OUT programmed operators is the address of the first word of a command list (see Section 4.4.1). Control does not return to the program until transmission is terminated or an error is detected.

Example

Dump Output

Dump input is similar to dump output. This routine outputs fixed-length records.

```

DMPINI:  0                ;JSR HERE TO INITIALIZE A FILE
          INIT 0, 16      ;CHANNEL 0, DUMP MODE
          SIXBIT/MTA2/    ;MAGNETIC TAPE UNIT 2
          0                ;NO RING BUFFERS
          JRST NOTAVL     ;WHERE TO GO IF UNIT 2 IS BUSY
          JRST @DMPINI    ;RETURN
DMPOUT:  0                ;JSR HERE TO OUTPUT THE OUTPUT AREA
          OUTPUT 0,OUTLST ;SPECIFIËS DUMP OUTPUT ACCORDING
                               ;TO THE LIST AT OUTLIST
          STATZ 0, 740000 ;CHECK ERROR BITS
          CALL[SIXBIT/EXIT/] ;QUIT IF AN ERROR OCCURS
          JRST @DMPOUT    ;RETURN
DMPDON:  0                ;JSR HERE TO WRITE AN END OF FILE
          CLOSE 0,        ;WRITE THE END OF FILE
          STATZ 0, 740000 ;CHECK FOR ERROR DURING WRITE
                               ;END OF FILE OPERATION
          CALL[SIXBIT/EXIT/] ;QUIT IF ERROR OCCURS
          RELEAS 0,       ;RELINQUISH THE DEVICE
          JRST @DMPDON    ;RETURN
OUTLST:  IOWD BUFSIZ,BUFFER ;SPECIFIES DUMPING A NUMBER OF
                               ;WORDS EQUAL TO BUFSIZ, STARTING
                               ;AT LOCATION BUFFER
          0                ;SPECIFIES THE END OF THE COMMAND
                               ;LIST
BUFFER   BLOCK BUFSIZ     ;OUTPUT BUFFER, MUST BE CLEARED
                               ;AND FILLED BY THE MAIN PROGRAM

```

4.4.3.2 Buffered Modes - In data modes 0, 1, 10, 13 and 14 the effective address E of the INPUT, IN, OUTPUT, and OUT programmed operators may be used to alter the normal sequence of buffer reference. If E is 0, the address of the next buffer is obtained from the right half of the second word of the current buffer. If E is nonzero, it is the address of the second word of the next buffer to be referenced. The buffer pointed to by E can be in an

entirely separate ring from the present buffer. Once a new buffer location is established, the following buffers are taken from the ring started at E.

a) Input - If no input buffer ring is established when the first INPUT or IN is executed, a 2-buffer ring is set up. (See INBUF, Section 4.4.2.3)

Buffered input may be performed synchronously or asynchronously at the option of the user. If bit 30 of the file status is 1, each INPUT and IN programmed operator does the following.

1. Clears the use bit in the second word of the buffer whose address is in the right half of the first word of the buffer header, thereby making the buffer available for refilling by the Monitor.
2. Advances to the next buffer by moving the contents of the second word of the current buffer to the right half of the first word of the 3-word buffer header.
3. Returns control to the user's program if an end-of-file or error condition exists. Otherwise, the Monitor starts the device which fills the buffer and stops transmission.
4. Computes the number of bytes in the buffer from the number of words in the buffer (right half of the first data word of the buffer) and the byte size, and stores the result in the third word of the buffer header.
5. Sets the position and address fields of the byte pointer in the second word of the buffer header, so that the first data byte is obtained by an ILDB instruction.
6. Returns control to the user's program.

Thus, in synchronous mode, the position of a device, such as magnetic tape, relative to the current data is easily determined. The asynchronous input mode differs in that once a device is started, successive buffers in the ring are filled at the interrupt level without stopping transmission until a buffer whose bit is 1 is encountered. Control returns to the user's program after the first buffer is filled. The position of the device relative to the data

currently being processed by the user's program depends on the number of buffers in the ring and when the device was last stopped

Example:

General Subroutine to Input One Character

```

GETCHR:      0           ;JSR HERE AND STORE PC
GETCNT:      SOSG  IBUF+2 ;DECREMENT THE BYTE COUNT
             JRST  GETBUF ;BUFFER IS EMPTY (OR FIRST CALL AFTER
             ;INIT)

GETNXT:      ILDB AC, IBUF+1 ;GET NEXT CHAR FROM BUFFER
             JUMPN AC @GETCHR ;RETURN TO CALLER IF NOT NULL CHAR1
             JRST  GETCNT ;IGNORE NULL AND GET NEXT CHAR

GETBUF:      IN      3           ;CALL MONITOR TO REFILL THIS BUFFER
             JRST  GETNXT ;RETURN HERE WHEN NEXT BUFFER IS
             ;FULL (PROBABLY IMMEDIATELY)
             JRST  ENDTST ;RETURN HERE ONLY IF ERROR OR EOF

ENDTST:      STATZ 3, 740000 ;CHECK FOUR ERROR BITS FIRST
             JRST  INERR  ;WHERE TO GO ON AN ERROR
             JRST  ENDFIL ;WHERE TO GO ON AN END OF FILE

```

b. Output- If no output buffer ring has been established, i.e., if the first word of the buffer header is 0, when the first OUT or OUTPUT is executed, a 2-buffer ring is set up (see OUTBUF, this chapter). If the ring use bit (bit 0 of the first word of the buffer header) is 1, it is set to 0, the current buffer is cleared to all 0s, and the position and address fields of the buffer byte pointer (the second word of the buffer header) are set so that the first byte is properly stored by an IDPB instruction. The byte count (the third word of the buffer header) is set to the maximum of bytes that may be stored in the buffer, and control is returned to the user's program. Thus, the first OUT or OUTPUT initializes the buffer header and the first buffer, but does not result in data transmission.

If the ring use bit is 0 and bit 31 of the file status is

¹ For some devices in ASCII mode, the item count provided will always be a multiple of five characters. Since the last word of a buffer may be partially full, user programs which rely upon the item count should always ignore null characters.

0, the number of words in the buffer is computed from the address field of the buffer byte pointer (the second word of the buffer header) and the buffer pointer (the first word of the buffer header), and the result is stored in the right half of the first data word of the buffer. If bit 31 of the file status is 1, it is assumed that the user has already set the word count in the right half of the first data word. The buffer use bit (bit 0 of the second word of the buffer) is set to 1, indicating that the buffer contains data to be transmitted to the device. If the device is not currently active i.e., not receiving data, it is started. The buffer header is advanced to the next buffer by setting the buffer pointer in the first word of the buffer header. If the buffer use bit of the new buffer is 1, the job is put into a wait state until the buffer is emptied at the interrupt level. The buffer is then cleared to 0s, the buffer byte pointer and byte count are initialized in the buffer header, and control is returned to the user's program.

Example:

General Subroutine to Output One Character

```

PUTCHR      0                ;JSR HERE AND STORE PC
            SOSG      OBUF+2 ;INCREMENT BYTE COUNT
            JRST     PUTBUF ;NO MORE ROOM (OR FIRST CALL AFTER INIT)

PUTNXT:     IDPB AC, OBUF+1 ;STORE THIS CHARACTER
            JRST     @PUTCHR ;AND RETURN TO CALLER

PUTBUF:     OUT      3        ;CALL MONITOR TO EMPTY THIS BUFFER
            JRST     PUTNXT ;RETURN HERE WHEN NEXT BUFFER IS
                        ;EMPTY (PROBABLY IMMEDIATELY)
            JRST     OUTERR ;RETURN HERE ONLY IF OUTPUT ERROR

OUTERR:     GETSTS    3,AC    ;GET THE ERROR STATUS TO LOOK AT
            .
            .
            .

```

4.4.4 Status Checking and Setting

The file status (see Table 4-5) is manipulated by the GETSTS (operation code 062), STATZ (operation code 063), STATO (operation code 061) and SETSTS (op code 060) programmed operators. In each case the

accumulator field of the instruction selects a data channel. If no device is associated with the specified data channel, the Monitor stops the job and prints,

I/O TO UNASSIGNED CHANNEL AT USER addr

(addr is the location of the GETSTS, STATZ, STATO, or SETSTS programmed operator) on the user's console leaving the console in Monitor mode.

GETSTS D,E stores the file status of data channel D in the right half and 0 in the left half of location E.

STATZ D,E skips, if all file status bits selected by the effective address E are 0.

STATO D,E skips, if any file status bit selected by the effective address E is 1.

SETSTS D,E waits until the device on channel D stops transmitting data and replaces the current file status, except bit 23, with the effective address E. If the new data mode, indicated in the right four bits of E, is not legal for the device, the job is stopped and the Monitor prints,

ILL DEVICE DATA MODE FOR DEVICE dev AT USER addr

(dev is the physical name of the device and addr is the location of the SETSTS operator) on the user's console leaving the console in Monitor mode. If the user program changes the data mode, it must also change the byte size for the byte pointer in the input buffer header (if any) and the byte size and item count in the output buffer header (if any). Changing the output item count should be done using the count already placed there by the Monitor and dividing or multiplying by the appropriate conversion factor, rather than assuming the length of a buffer.

4.4.5 Terminating A File (CLOSE)

File transmission is terminated by the CLOSE D,N (Operation code 070) programmed operator. If no device is associated with channel D or if bits 34 and 35 of the instruction are both 1, control returns to the user's program immediately.

If bit 34 is 0 and the input side of data channel D is open, it is now closed. In data modes 15, 16, and 17, the effect is to execute a device dependent function and clear the end-of-file flag, bit 22 of the file status. Data modes 0, 1, 10, 13, and 14 have the additional effect, if an input buffer ring exists, of setting the ring use bit (bit 0 of the first word of the buffer header) to 1, setting the buffer byte count (the third word of the buffer header) to 0 and setting the buffer use bit (bit 0 of the second word of the buffer) of each buffer to 0.

If bit 35 of the instruction is 0 and the output side of channel D is open, it is now closed. In data modes 15, 16, and 17, the effect is to execute a device dependent function. In data modes 0, 1, 10, 13, and 14, if a buffer ring exists, the following operations are performed.

- a) All data in the buffers that has not yet been transmitted to the device is now written.
- b) Device dependent functions are performed.
- c) The ring use bit is set to 1.
- d) The buffer byte count is set to 0.
- e) Control returns to the user after transmission is complete.

Example:

Terminating A File

```

DROPDV:      0           ;JSR HERE
             CLOSE 3,    ;WRITE END OF FILE AND TERMINATE
             ;INPUT
             STATZ 3, 740000 ;RECHECK FINAL ERROR BITS
             JRST OUTERR ;ERROR DURING CLOSE
             RELEAS 3,    ;RELINQUISH THE USE OF THE
             ;DEVICE, WRITE OUT THE DIRECTORY

             MOVE 0, SVJBFF
             MOVEM 0, JOBFF ;RECLAIM THE BUFFER SPACE
             JRST @ DROPDV ;RETURN TO MAIN SEQUENCE

```

4.4.6 Synchronization of Buffered I/O (CALL D, [SIXBIT/WAIT/]

In some instances, such as recovery from transmission errors, it is desirable to delay until a device completes its input/output activities. The programmed operators,

```
CALL D, [SIXBIT/WAIT/]and CALLI D,10
```

return control to the user's program when all data transfers on channel D have finished. This UO does not wait for a Magtape spacing operation, since no data transfer is in progress. An MTAPE D, 0 (see Section 5.7.2) should be used to wait for spacing and I/O activity to finish on Magtape. If no device is associated with data channel D, control returns immediately. After the device is stopped, the position of the device relative to the data currently being processed by the user's program can be determined by the buffer use bits.

4.4.7 Relinquishing A Device (RELEASE)

When all transmission between the user's program and a device is finished, the program must relinquish the device by performing a

```
RELEASE D,
```

RELEASE (operation code 071) returns control immediately, if no device is associated with data channel D. Otherwise, both input and output sides of data channel D are CLOSED and the

correspondence between channel D and the device, which was established by the INIT or OPEN programmed operators, is terminated. If the device is neither associated with another data channel nor assigned by the ASSIGN command (see Chapter 2), it is returned to the Monitor's pool of available facilities. Control is returned to the user's program.

4.5 CORE CONTROL

4.5.1 CALL AC, [SIXBIT/CORE/] or CALLI, ll - These provide a user program with the ability to expand and contract its core size as its memory requirements change. In order to allocate core in either or both segments, the left half of AC is used to specify the highest user address to be assigned to the high segment. If the left half of AC contains 0, the high segment core assignment is not changed. If the left half of AC is non-zero and is either less than 400000 or the length of the low segment, whichever is greater, the high segment is eliminated. If this is executed from the high segment, an illegal memory error message is printed when the Monitor attempts to return control to the illegal memory.

The error return is given if LH is greater than or equal to 400000 and if either the system does not have a two-segment capability or the user has been meddling without write access privileges (see section 4.6). A RH of 0 leaves the low segment core assignment unaffected. The Monitor clears new core before assigning it to the user, so that privacy of information is insured.

In swapping systems, these programmed operators return the maximum number of 1K core blocks (all of core minus the Monitor, unless an installation chooses to restrict the amount of core) available to the user. By restricting the amount of core available to

users, the number of jobs in core simultaneously is increased. In non-swapping systems, the number of free and dormant 1K blocks are returned. Therefore, the CORE UWO and the CORE command return the same information.

```
The call is:      MOVE AC [XWD HIGH ADR or 0, LOW ADDR or 0]
                  CALL AC, [SIXBIT/CORE/] or CALLI AC, 11
                  error return
                  normal return
```

The CORE UWO reassigns the low segment (if RH is non-zero) and then reassigns the high segment (if LH is non-zero). If the sum of the new low segment and the old high segment exceeds the maximum amount of core allowed to a user, the error return is given, the core assignment is unchanged, and the maximum core available to the user for high and low segments (in 1K blocks) is returned in the AC. In a non-swapping system, the number of free and dormant 1K blocks is returned.

If the sum of the new low segment and the new high segment exceeds the maximum amount of core allowed to a user, the error return is given, the new low segment is assigned, the old high segment remains, and the maximum core available to the user in 1K blocks is returned in the AC. Therefore to increase the low segment and decrease the high segment at the same time, two separate CORE UWO's should be used in order to reduce the chances of exceeding the maximum size allowed to a user job.

If the new low segment extends beyond 377777, the high segment shifts up into the virtual addressing space instead of being overlaid. If a long low segment is shortened to 377777 or less, the high segment shifts from the virtual addressing space to 400000 instead of growing longer or remaining where it was. If the high segment is a program, it does not execute properly after a shift unless it is a self-relocating program in which all transfer instructions are indexed.

If the high segment is eliminated by a CORE UUO , a subsequent CORE UUO in which the LH is greater than 400000 will create a new, non-sharable segment rather than reestablishing the old high segment. This segment becomes sharable after it has been a) given an extension .SHR, b) written onto the storage device, c) closed so that a directory entry is made, and d) initialized from the storage device by GET,R, or RUN commands or RUN or GETSEG UUO's. This is the same sequence which the Loader and the SAVE and GET commands use to create and initialize new sharable segments.

4.5.2 CALL AC, [SIXBIT/SETUWP/] or CALLI AC,36 - These allow a user program to set or clear the hardware user-mode write protect bit and to obtain the previous setting. It must be used if a user program is to modify the high segment.

The call is: CALL AC, [SIXBIT/SETUWP/] ; OR CALLI AC,36
 error return
 normal return

If the system has a two-register capability, the normal return will be given unless the user has been meddling without write privileges, in which case an error return will be given. This happens whether or not the program has a high segment because the reentrant software is designed to allow users to write programs for two-register machines which will run under one-register machines. Compatibility of source and relocatable binary files is therefore maintained between one-register and two-register machines.

If the system has a one-register capability, the error return (bit 35 of AC=0) is given. This allows the user program to find out whether or not the system has a two-segment capability. The user program specifies the setting of the user-mode write protect bit in bit 35 of AC (write protect =1, write privileges =0). The previous setting of the user-mode write protect bit is returned

in bit 35 of AC, so that any user subroutine can preserve the previous setting before changing it. Therefore, nested user subroutines which each set or clear the bit can be written, provided the subroutines save the previous value of the bit and restore it upon returning to its caller.

4.6 Modifying Shared Segments, and Meddling

Usually a high segment is write-protected, but it is possible for a user program to turn off the user write-protect bit or to increase or decrease a shared segment's core assignment by using the SETUWP or CORE UUO's. These are legal from the high or low segment, provided the sharable segment has not been "meddled" with unless the user has write privileges for the file that initialized the high segment. Even the malicious user can have the privilege of running such a program, although he does not have the access rights to modify the file used to initialize the sharable segment.

Meddling is defined as any of the following, even if the user has privileges to write the file which initialized the sharable segment.

- a) START or CSTART commands with an argument.
- b) DEPOSIT command in the low or high segment.
- c) RUN UUO with anything other than a 0 or 1 in LH of AC as a starting address increment.
- d) GETSEG UUO.

It is not considered meddling to do any of the foregoing with a non-sharable program. It is never considered meddling to type $\uparrow C$ followed by START (withoug an argument), CONT, CCONT, CSTART (without an argument), REENTER, DDT, SAVE, or E command.

When a sharable program is meddled with, the Monitor sets the meddle bit for the user. An error return is given when the

clearing of the user write-protect bit is attempted with the SETUWP UUU or the reassignment of core for the high segment (except to remove it completely) is attempted with the CORE UUU. An attempt to modify the high segment with the DEPOSIT command causes the message

OUT OF BOUNDS

to be printed. If the user write-protect bit was not set when the user meddled, it will be set so as to protect the high segment in case it is being shared. The command and the two UUU's are allowed in spite of meddling, if the user has the access privileges to write the file which initialized the high segment.

A privileged programmer is able to supersede a sharable program which is in the process of being shared by a number of users. Whenever a successful CLOSE, OUTPUT, or RENAME UUU is executed for a file with the same directory name and filename (previous name if the RENAME UUU is used) as the segment being shared, the segment's name will be set to 0. New users will not share the older version, but will share the newer version. This requires the Monitor to read the newly created file only once to initialize it. The Monitor deletes the older version when all users are finished sharing it.

Users with access privileges are able to write programs which access sharable data segments via the GETSEG UUU (which is meddling) and then turn off the user write-protect bit using SETUWP UUU. With DECTape, write privileges exist if it is assigned to the job (cannot be a system tape) or is not assigned to any job and is not a system tape.

When control can be transferred only to a small number of

entry points (2) which the shared program is prepared to handle, then the shared program can do anything it has the privileges to do, even though the person running the program does not have these privileges.

The ASSIGN (and DEASSIGN, FINISH, KJOB if device was previously assigned by console) command clears all shared segment names currently in use which were initialized from the device, if the device is removable (DTA,MTA). Otherwise new users could continue to share the old segment indefinitely, even if a new version were mounted on the device. Therefore, it is possible to update the library during regular time-sharing, if the programmer has the access privileges. In a DEctape system, a new CUSP tape can be mounted followed by an ASSIGN SYS command which clears segment names for the physical device but does not assign the device because everyone needs to share it.

Device Dependent Functions

This chapter explains the unique features of each standard I/O device. All devices accept the programmed operators explained in Chapter 4 unless otherwise indicated. Buffer sizes are given in octal and include two bookkeeping words. The user may determine the physical characteristics associated with a logical device name by executing a DEVCHR UUO. (See 5.12.) Table 5-1 is a summary of the characteristics of all devices.

Table 5-1
Device Summary

Physical Name	Name	Hardware Type Number	Programmed Operator	Data Modes	Buffer ¹ Size (Octal)
CTY	Console Teletype	626 Models 33, 35, 37	INPUT, IN OUTPUT, OUT	A, AL	23
TTY0, TTY1, ..., TTY77	Teletype	630, 680, or DC10	INPUT, IN OUTPUT, OUT, TTCALL	A, AL	23
PTY	Pseudo-Teletype	None	INPUT, IN OUTPUT, OUT	A, AL	23
PTR	Paper Tape Reader	760	INPUT, IN	A, AL, I, B, IB	43
PTP	Paper Tape Punch	761	OUTPUT, OUT	A, AL, I, B, IB	43
PLT	Plotter	XY 10	OUTPUT, OUT	A, AL, I, B, IB	46
LPT or LPT0, ..., LPT7	Line Printer	646, LP10	OUTPUT, OUT	A, AL, I,	34
CDR	Card Reader	461, CR10	INPUT, IN	A, AL, I, B	36

Table 5-1 (Cont.)

Device Summary

Physical Name	Name	Hardware Type Number	Programmed Operator	Data Modes	Buffer ¹ Size (Octal)
CDP	Card Punch	CP10	OUTPUT, OUT	A, AL, B, IB	35
DTA0, DTA1, ..., DTA7	DECTape	551/555, TD10/TD55	INPUT, IN OUTPUT, OUT LOOKUP ENTER MTAPE USETO USETI UGETF CALL [SIXBIT/UTPCLR/]	A, AL, I, B, IB, DR, D	202
MTA0, MTA1, ..., MTA7	Magnetic Tape	516, TM10, TU20, TU79	INPUT, IN OUTPUT, OUT MTAPE	A, AL, I, B, IB, DR, D	203
DSK	Disk	RC10	INPUT, IN OUTPUT, OUT LOOKUP ENTER RENAME USETO	A, AL, I, B, IB, DR, D	203
DIS	Display	30, 340	INPUT OUTPUT	ID	Dump only

¹Buffer sizes are subject to change and should be calculated rather than assumed by user programs. A dummy INBUF or OUTBUF may be employed for this purpose.

5.1 TELETYPE

Device Name - TTY0, TTY1, ..., TTY76, TTY77, CTY

Line number n of the Type 630 Data Communications System, Data Line Scanner DC10, PDP-8 680 System, or PDP-8/I 680I System is referred to as TTYn. The console Teletype is CTY. The Time-Sharing Monitor automatically gives the logical name, TTY, to the user's

console whenever a job is initialized.

Teletype device names are assigned dynamically. For inter-console communication by program, it is necessary for one of the two users to type DEASSIGN TTY in order to make his Teletype available to the other user's program as an output or input device. Typing ASSIGN TTYn is the only way to reassign a Teletype that has been de-assigned. Also see TALK command, Chapter 2.

Buffer Size - 23₈ words.

Two choices of Teletype routines are provided: a newer, full duplex software routine and an older, half duplex software routine. Use of the full duplex software is encouraged.

With a full duplex Teletype service, the two functions of a console, typein and typeout, are handled independently and need not be handled in the strict sense of output first and then input. For example, if two operations are desired from PIP, the request for the second operation can be typed before receiving the asterisk after completion of the first. The echo of characters typed in will disappear since the keyboard and the printing operations are independent. To stop output that is not wanted, a "Control O" is typed. Also, the command "Control C" will not stop a program instantly. Rather, the Control C will be delayed until the program requests input from the keyboard, and then the program will be stopped. When a program must be stopped instantly, as when it gets into a loop, Control C typed twice will stop the program.

Programs waiting for Teletype output will be awakened eight characters before the output buffer is empty, causing them to be swapped in sooner and preventing pauses in typing. Programs waiting for Teletype input will be awakened ten characters before the input buffer is filled, thus reducing the probability of lost typein.

5.1.1 Data Modes5.1.1.1 Full-Duplex Software A(ASCII) and AL(ASCII Line)

The input handling of all control characters is as follows.

(All are passed to program except as noted below).

000	NULL	Ignored on input, suppressed on output.
001	↑A	Echoes as ↑A. Passed to program.
002	↑B	Complements switch controlling echoing, not passed to program. Used on local-copy dataphones and TWX's.
003	↑C	The Teletype mode is switched to Monitor mode the next time input is requested by the program. Two successive ↑C's cause the mode to be switched to Monitor mode immediately.
004	↑D (EOT)	004 passed to program. Not echoed, so typing in a "Control D" (EOT) will not cause a full duplex dataphone to hang up.
005	↑E (WRU)	No special action.
006	↑F	Complements switch controlling translation of lower case letters to upper case. Used when lower case input is desired to programs. Not sent to program, but program can sense the state of this switch by the TTCALL UVO.
007	↑G (Bell)	007 passed to program, and is a break character.
010	↑H (Back-space)	Acts as a RUBOUT, unless either DDT mode or full character set mode is true, or the ↑F switch is on. In these cases, 010 is sent to the program.
011	↑I (TAB)	011 passed to program. Echoed as spaces if Teletype is a model 33 (determined by ↑P switch). Spaces are not passed to program.
012	↑J (Line-feed)	Is a break character. No other special action.
013	↑K (Vertical Tab)	013 passed to program. Echoes as four linefeeds, if a model 33. Is a break character. Linefeeds are not passed to program.
014	↑L (Form)	014 passed to program. Echoes as 8 linefeeds on a 33. Is a break character. Linefeeds are not passed to program.
015	↑M (Carriage Return)	If Teletype is in paper-tape input mode, 015 is simply passed to program. Otherwise supplies a linefeed echo, and is passed to program as a CR and LF, and is a break character (due to LF).
016	↑N	No special action
017	↑O	Suppresses output until an INPUT, or an INIT, or OPEN UVO occurs. Not passed to program. Typed as ↑O followed by carriage return-linefeed.

020	↑P	Does not appear in the input buffer. Some Teletype units (usually Models 35 and 37) have horizontal tab, vertical tab, and form feed mechanisms while other units (usually Model 33s) do not. If the user finds that his particular Teletype unit does not have these mechanisms, he should type ↑P. Otherwise, tabs will not be printed at all or spaces will be substituted for a tab depending upon the Monitor's assumption.
021	↑Q (XON)	Starts paper-tape-mode, as described above. Passed to program.
022	↑R (TAPE)	No special action.
023	↑S (XOFF)	Ends paper-tape mode, as described above. 023 is passed to program.
024	↑T (NO TAPE)	No special action.
025	↑U	Deletes input line back to last break character. Typed back as ↑U followed by carriage return-linefeed.
026	↑V	No special action.
027	↑W	No special action.
030	↑X	No special action.
031	↑Y	No special action.
032	↑Z	Acts as end-of-file on Teletype input. Echoes as ↑Z followed by carriage return-linefeed. Is a break character. Appears in buffer as 032.
033	↑[(ESC)	This is the ASCII altmode these days, but is translated to 175 before being passed to the program, unless in full character set mode (bit 29 in INIT). 175 is the 1963 altmode. Echoes as a dollar sign. Always, is a break character.
034	↑\	No special action.
035	↑]	No special action.
036	↑↑	No special action
037	↑←	No special action.
040-137		Printing characters, no special action.
140-174		"Lower case" ASCII. Translated to upper case, unless ↑F switch is set. Echoes as upper case if translated to upper case.
175 and 176		Old versions of altmode. See description of "ESC" (033).
177		RUBOUT or DELETE: A) Completely ignored if in papertape mode (XON) B) Is a break character, passed to program if either DDTmode or fullcharacter-set mode is true. C) Otherwise (ordinary case) causes a character to be deleted for each rubout typed. All the characters deleted are echoed between a single pair of backslashes. If no characters remain to be deleted, echoes as a carriage return-linefeed.

On output, all characters are typed just as they appear in the output buffer with the exception of TAB, VT, and FORM, which are processed the same as on type in.

5.1.1.2 Half-Duplex Software A(ASCII) - If, during output operations, an echo-check failure occurs (the transmitted character was not the same as the intended character), the I/O routine suspends output until the user types the next character. If that character is ↑C, the console is placed in Monitor mode immediately. If it is ↑O, all Teletype output buffers that are currently full are ignored, thus cutting the output short. All other characters cause the service routines to continue output. The user may cause a deliberate echo check by typing in while typeout is in progress. For example, to return to Monitor control mode while typeout is in progress, the user must type any character ("X", for example) until an echo check occurs and output is suspended; then and only then he types ↑C.

The buffer is terminated when it fills up or when the user types ↑Z.

5.1.1.3 Half-Duplex Software AL(ASCII Line) - Same as ASCII mode (usually preferred) with the addition that the input buffer is terminated by a CR/LF pair, FF, VT, or ALTMODE.

5.1.2 DDT Submode

To allow a user's program and the DDT debugging program to use the same Teletype without interfering with one another, the Teletype service routine provides the DDT submode. This mode does not affect the Teletype status if it is initialized with the INIT operator. It is not necessary to use INIT in order to do I/O in the DDT submode. I/O in DDT mode is always to the user's Teletype and

not to any other device.

In the DDT submode, the user's program is responsible for its own buffering. Input is usually one character at a time, but if the typist types characters faster than they are processed, the Teletype service routine supplies bufferfuls of characters at a time.

To input characters in DDT mode, use the sequence

```
MOVEI AC, BUF
CALL  AC, [SIXBIT/DDTIN/]
```

BUF is the first address of a 21-word block in the user's area. The DDTIN operator delays, if necessary, until one character is typed in. Then all characters (in 7-bit packed format) typed in since the previous occurrence of DDTIN are moved to the user's area in locations BUF, BUF+1, etc. The character string is always terminated by a null character (000). RUBOUTs are not processed by the service routine but are passed on to the user. The special control characters ↑O and ↑U have no effect. Other characters are processed as in ASCII mode.

To perform output in DDT mode, use the sequence

```
MOVEI AC, BUF
CALL  AC, [SIXBIT/DDTOUT/]
```

BUF is the first address of a string of packed 7-bit characters terminated by a null (000) character. The Teletype service routine delays until the previous DDTOUT operation is complete, then moves the entire character string into the Monitor, begins outputting the string, and restarts the user's program. Character processing is the same as for ASCII mode output.

5.1.3 Special Programmed Operator Service

TTCALL UUO is (and will always be) implemented only in the "full duplex scanner service", SCNSRF. The general form of this UUO is as follows:

OPDEF TTCALL [51B8]
 TTCALL AC, ADR

The AC field describes the particular function desired, and the argument (if any) is contained in ADR. ADR may be an AC or any address in low segment above JOB AREA (137). It may be in high segment for AC fields 1 and 3. The functions are:

AC Field	Mnemonic	Action
0	INCHRW	Input character and wait
1	OUTCHR	Output a character
2	INCHRS	Input character and skip
3	OUTSTR	Output a string
4	INCHWL	Input character, wait, line mode
5	INCHSL	Input character, skip, line mode
6	GETLIN	Get line characteristics
7	SETLIN	Set line characteristics
10	RESCAN	Reset input stream to command
11	CLRBFI	Clear typein buffer
12	CLRBFO	Clear typeout buffer
13	SKPINC	Skips if a character can be input
14	SKPINL	Skips if a line can be input
15-17	(Reserved for Expansion)	

INCHRW TTCALL 0,ADR

This command inputs a character into location ADR. ADR may be an AC or any other location in the user's low segment. If there is no character yet typed, the program waits for it.

OUTCHR TTCALL 1,ADR

This command outputs a character to the Teletype from location ADR. Only the low order 7 bits of the contents of ADR are used. The rest need not be zeroes.

If there is no room in the output buffer, the program waits until room is available. ADR may be in high segment.

INCHRS TTCALL 2,ADR

This command is similar to INCHRW, except that it skips on a successful return, and does not skip if there is no character in the input buffer; it never puts the job into a wait.

```
TTCALL    2,ADR
JRST      NONE          ;NO TYPEIN
JRST      DONE          ;CHARACTER IN ADR
```

```
OUTSTR    TTCALL    3,ADR
```

This command outputs a string of characters in ASCIIZ format:

```
TTCALL    3,MESSAGE
MESSAGE:  ASCIIZ      /TYPE THIS OUT/
ADR may be in high segment
```

```
INCHWL    TTCALL    4,ADR
```

This command is the same as INCHRW, except that it decides whether or not to wait on the basis of lines rather than characters; as such, it is the preferred way of inputting characters, since INCHRW causes a swap to occur for each character rather than each line (compare DDT and PIP input, for instance).

```
INCHSL    TTCALL    5,ADR
```

This command is the same as INCHRS, except that its decision whether to skip is made on the basis of lines rather than characters.

```
GETLIN    TTCALL    6,ADR
```

This command takes one argument, from location ADR, and returns one word, also in ADR. The argument is a number, representing a Teletype line. If the argument is negative, the line number controlling the program is assumed. If the line number is greater than those defined in the system, a zero answer is returned.

The normal answer format is as follows:

Right half of ADR:	The line number.
Left half of ADR:	Bits, as follows:
Bit	Meaning
0	Line is a pseudo-teletype.
1	Line is the CTY.
2	Line is a display console.
3	Line is a dataset data line.
4	Line is a dataset control line.
5	Line is half-duplex.

Bit	Meaning
11	A line has been typed in by the user.
12	A rubout has been typed.
13	"Control F" switch is on.
14	"Control P" switch is on.
15	"Control B" switch is on.
16	"Control Q" (paper tape) switch is on.
17	Line is in a "talk" ring.

SETLIN TTCALL 7,ADR

This command allows a program to set and clear some of the bits described for GETLIN. They may be changed only for the controlling Teletype. The bits which may be modified are bits 13, 14, 15 and 16. Example:

```

SETO    AC,0
TTCALL 6,AC
TLZ    AC,BIT 13
TLO    AC,BIT 14
TTCALL 7,AC

```

RESCAN TTCALL 10,0

This command is intended for use only by the CCL CUSP. It causes the Input Buffer to be re-scanned from the point where the last command began. Obviously, if it is executed other than before the first input, that command may no longer be in the buffer. ADR is not used, (but is address checked).

CLRBFI TTCALL 11,0

This command causes the Input Buffer to be cleared (as if the user had typed a number of "Control U's"). It is intended to be used when an error has been detected, such that a user probably would not want any commands to be executed which he might have typed ahead.

CLRBFO TTCALL 12,0

This command causes the output buffer to be cleared, as if the user had typed "CONTROL O". It should be used only rarely, since usually one wants to see all output, up to the point of an error. It is included primarily for completeness.

SKPINC TTCALL 13,0

This command skips if the user has typed at least one character. It does not skip if no characters have been typed; however, it never inputs a character. It is useful for a compute based program which wants to occasionally check for input and, if any, go off to another routine (such as FORTRAN Operating System) to actually do the input.

SKPINL TTCALL 14,0

This command is the same as SKPINC except that a skip occurs if a line has been typed.

5.1.4 Special Status Bits (Full Duplex Software Only)

An INIT or OPEN, with bit 28 a one, suppresses echoing on the Teletype. This is useful for LOGIN to eliminate the mask for the password.

5.1.5 Paper Tape Input from the Teletype (Full Duplex Software Only)

Paper tape input is possible from a Teletype equipped with a paper tape reader, controlled by the XON and XOFF characters. When commanded by the XON character, the Teletype service will read paper tapes, starting and stopping the paper tape as needed and continuing until the XOFF character is read or typed in. While in this mode of operation, any RUBOUTS will be discarded and no free line feeds will be inserted after carriage returns. Also, TABS and FORMFEEDS will not be simulated on Model 33's, to insure output of the reader control characters. In order to use paper tape processing, the Teletype with paper tape reader must be connected by a full duplex connection and only ASCII paper tapes are intended to be used.

The correct operating sequence for reading a paper tape in this way is as follows:

```
.R PIP <RETURN>
*DSK: FILE<TTY: <XON><RETURN><LINEFEED>
THIS IS WHAT IS ON TAPE
MORE OF SAME
LAST LINE
↑Z
*<XOFF>
```

5.2 PAPER TAPE READER

Device Mnemonic - PTR

Buffer Size - 43₈ words

5.2.1 Data Modes (Input Only)

NOTE: To initialize the paper tape reader, the input tape must be threaded through the reading mechanism and the FEED button depressed.

5.2.1.1 A (ASCII) - Blank tape (000), RUBOUT (377), and null characters (200) are ignored. All other characters are truncated to seven bits and appear in the buffer. The physical end of the paper tape serves as an end-of-file.

5.2.1.2 AL (ASCII Line) - Character processing is the same as for the A mode. The buffer is terminated by LINE FEED, FORM, or VT.

5.2.1.3 I (Image) - There is no character processing. The buffer is packed with 8-bit characters exactly as read from the input tape. Physical end of tape is the end-of-file indication but does not cause a character to appear in the buffer.

5.2.1.4 IB (Image Binary) - Characters not having the eighth hole punched are ignored. Characters are truncated to six bits and

packed six to the word without further processing. This mode is useful for reading binary tapes having arbitrary blocking format.

5.2.1.5 B (Binary) - Checksummed binary data is read in the following format. The right half of the first word of each physical block contains the number of data words that follow and the left contains half a folded checksum. The checksum is formed by adding the data words using 2s complement arithmetic, then splitting the sum into three 12-bit bytes and adding these using 1s complement arithmetic to form a 12-bit checksum. The data error status flag (see Table 4.5) is raised if the checksum mismatches. Because the checksum and word count appear in the input buffer, the maximum block length is 40. The byte pointer, however, is initialized so as not to pick up the word count and checksum word.

Again, physical end of tape is the end-of-file indication but does not result in putting a character in the buffer.

5.3 PAPER TAPE PUNCH

Device Mnemonic - PTP

Buffer Size - 43_g words

5.3.1 Data Modes

5.3.1.1 A (ASCII) - The eighth hole is punched for all characters. Tape-feed without the eighth hole (000) is inserted after form-feed. A rubout is inserted after each vertical or horizontal tab. Null characters (000) appearing in the buffer are not punched.

5.3.1.2 AL (ASCII Line) - The same as A mode. Format control must be performed by the user's program.

5.3.1.3 I (Image) - Eight-bit characters are punched exactly as they appear in the buffer with no additional processing.

5.3.1.4 IB (Image Binary) - Binary words taken from the output buffer are split into six 6-bit bytes and punched with the eighth hole punched in each line. There is no format control or check-summing performed by the I/O routine. Data punched in this mode is read back by the paper tape reader in the IB mode.

5.3.1.5 B (Binary) - Each bufferful of data is punched as one checksummed binary block as described for the paper tape reader. Several blank lines are punched after each bufferful for visual clarity.

5.3.2 Special Programmed Operator Service

The first output programmed operator of a file causes about two fanfolds of blank tape to be punched as leader. Following a CLOSE, an additional fanfold of blank tape is punched as trailer. No end-of-file character is punched automatically.

5.4 LINE PRINTER

Device Mnemonic - LPT

Buffer Size - 34₈ words

5.4.1 Data Modes

5.4.1.1 A (ASCII) - ASCII characters are transmitted to the line printer exactly as they appear in the buffer. See the PDP-10 System Reference Manual, for a list of the vertical spacing characters.

5.4.1.2 AL (ASCII Line) - This mode is exactly the same as A and is included for programming convenience. All format control must be performed by the user's program; this includes placing a RETURN, LINE-FEED sequence at the end of each line.

5.4.1.3 I (Image) - Same as A(ASCII) mode.

5.4.2 Special Programmed Operator Service

The first output programmed operator of a file and the CLOSE at the end of a file cause an extra form-feed to be printed to keep files separated.

5.5 CARD READER

Device Mnemonic - CDR

Buffer Size - 36₈ words

5.5.1 Data Modes

5.5.1.1 A (ASCII) - All 80 columns of each card are read and translated to 7-bit ASCII code. Blank columns are translated to spaces. At the end of each card a carriage-return/line-feed is appended. A card with the character 12-11-0-1 punched in column 1 is an end-of-file card. Columns 2 through 80 are ignored. The end-of-file button on the card reader has the same effect as the end-of-file card. As many complete cards as can fit are placed in the input buffer, but cards are not split between two buffers. Using the standard-sized buffer, only one card is placed in each buffer.

Cards are normally translated as IBM 026 card codes. If a card containing a 12-0-2-4-6-8 punch in column 1 is encountered, any following cards are translated as 029 codes (see Table 5-2

PDP-10 Card Codes) until the 029 conversion mode is turned off. The 029 mode is turned off either by a RELEASE command or by a card containing a 12-2-4-8 punch in column 1. Columns 2 through 80 of both of these cards are ignored.

5.5.1.2 AL (ASCII Line) - Exactly the same as the A mode.

5.5.1.3 I (Image) - All 12 punches in all 80 columns are packed into the buffer as 12-bit bytes. The first 12-bit byte is column 1. The last word of the buffer contains columns 79 and 80 as the left and middle bytes, respectively. The end-of-file card and the end-of-file button are processed the same as in the A mode. Cards are not split between two buffers.

5.5.1.4 B (Binary) - Card column 1 must contain a 7-9 punch to verify that the card is in binary format. Column 1 also contains the word count in rows 12-2. The absence of the 7-9 punch results in raising the IOIMPM (improper mode) flag in the card reader status word. Card column 2 must contain a 12-bit checksum as described for the paper tape reader binary format. Columns 3 through 80 contain binary data, 3 columns per word for up to 26 words. Cards are not split between two buffers. The end-of-file card and the end-of-file button are processed the same as in the A mode with a word containing 003200000000 appearing as the last word in the file.

5.6 CARD PUNCH

Device Mnemonic - DCP

Buffer Size - 35₈ words

5.6.1 Data Modes

Table 5-2

PDP-10 Card Codes

CHAR	PDP-10 ASCII	DEC 029	DEC 026	CHAR	PDP-10 ASCII	DEC 029	DEC 026
SPACE	040			@	100	84	84
!	041	1182	1287	A	101	121	121
"	042	87	085	B	102	122	122
#	043	83	086	C	103	123	123
\$	044	1183	1183	D	104	124	124
%	045	084	087	E	105	125	125
&	046	12	1187	F	106	126	126
'	047	85	86	G	107	127	127
(050	1285	084	H	110	128	128
)	051	1185	1284	I	111	129	129
*	052	1184	1184	J	112	111	111
+	053	1286	12	K	113	112	112
,	054	083	083	L	114	113	113
-	055	11	11	M	115	114	114
.	056	1283	1283	N	116	115	115
/	057	01	01	O	117	116	116
0	060	0	0	P	120	117	117
1	061	1	1	Q	121	118	118
2	062	2	2	R	122	119	119
3	063	3	3	S	123	02	02
4	064	4	4	T	124	03	03
5	065	5	5	U	125	04	04
6	066	6	6	V	126	05	05
7	067	7	7	W	127	06	06
8	070	8	8	X	130	07	07
9	071	9	9	Y	131	08	08
:	072	82	1182 or 110	Z	132	09	09
;	073	1186	082	[133	1282	1185
<	074	1284	1286	\	134	1187	87
=	075	86	83]	135	082	1285
>	076	086	1186	↑	136	1287	85
?	077	087	1282 or 120	←	137	085	82

5.6.1.1 A (ASCII) - ASCII characters are converted to card codes and punched (up to 80 characters per card). Tabs are simulated by punching from 1 to 8 blank columns; form-feeds and carriage returns are ignored. Line-feeds cause a card to be punched. All other nontranslatable ASCII characters cause a question mark to be punched. Cards can be split between buffers. Attempting to punch more than 80 columns per card causes the error bit IOBKTL to be raised. The CLOSE will punch the last partial card and then punch an EOF Card

(12-11-0-1 in column 1).

Cards are normally punched with DEC026 card codes. If bit 26 (octal 1000) of the status word is on (from INIT, OPEN, or SETSTS), cards are punched with DEC029 codes. The first card of any file indicates the card code used (12-0-2-4-6-8 punch in column 1 for DEC029 card codes; 12-2-4-8 punch in column 1 for DEC026 card codes).

5.6.1.2 AL (ASCII Line) - The same as A mode.

5.6.1.3 IB (Image Binary) - Up to 26 $\frac{2}{3}$ data words will be punched in columns 1-80. The buffer set up by the Monitor will only contain room for 26 data words. To punch a full 80-column card, the user has to set up his own buffers. Image binary will cause exactly one card to be punched for each output. The CLOSE will punch the last partial card, and then punch an EOF card (12-11-0-1 in column 1).

5.6.1.4 B (Binary) - Column 1 will contain the word count in rows 12-2. A 7-9 punch will also be in column 1. Column 2 will contain a checksum; columns 3-80 will contain up to 26 data words, 3 columns per word. Binary will cause exactly one card to be punched for each output. The CLOSE will punch the last partial card, and then punch an EOF card (12-11-0-1 in column 1).

5.6.2 Special Programmed Operator Service

Following a CLOSE, an end-of-file card is punched.

Both the first card of the file (the one that identifies the card code used) and the end-of-file card are laced in columns 2 through 80 for easy identification of files. These laced punches

are ignored by the card reader service routine.

5.7 DECTAPE

Device Mnemonic - DTA0, DTAl, ..., DTA7

Buffer Size - 202₈ words

5.7.1 Data Modes

5.7.1.1 A (ASCII) - Data is written on DECTape exactly as it appears in the buffer. No processing or checksumming of any kind is performed by the service routine. The self-checking of the DECTape system is sufficient assurance that the data is correct. See the description of DECTape format below for further information concerning blocking of information.

5.7.1.2 AL (ASCII Line) - Same as A.

5.7.1.3 I (Image) - Same as A. Data consists of 36-bit words.

5.7.1.4 IB (Image Binary) - Same as I.

5.7.1.5 B (Binary) - Same as I.

5.7.1.6 DR (Dump Records) - This mode is accepted but actually functions as dump mode 17.

5.7.1.7 D (Dump) - Data is read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is as described in

Chapter 4, "Unbuffered (Dump) Modes;" any positive number appearing in a command list terminates the list. Dump data is automatically blocked into standard-length DECTape blocks by the DECTape control. Unless the number of data words is an exact multiple of the standard length of a DECTape block (128_{10}), after each output programmed operator, the remainder of the last block written is wasted. The input programmed operator must specify the same number of words that the corresponding output programmed operator specified in order to skip over the wasted fractions of blocks.

5.7.2 DECTape Block Format

A standard reel of DECTape consists of 578 (1102_8) pre-recorded blocks each capable of storing 128 (200_8) 36-bit words of data. Block numbers which label the blocks for addressing purposes are recorded between blocks. These block numbers run from 0 to 1101_8 . Blocks 0, 1, and 2 are normally not used during time-sharing and are reserved for a bootstrap loader. Block 100_{10} (144_8) is the directory block which contains the names of all files on the tape and information relating to each file. Blocks 1_{10} through 99_{10} ($1-143_8$) and 101_{10} through 577_{10} ($145-1101_8$) are usable for data.

If in the process of DECTape I/O, the I/O service routine is requested to use a block number larger than 1101_8 or smaller than 0, the Monitor sets the Block Too Large flag (bit 21) in the file status and returns.

5.7.3 DECTape Directory Format

The directory block (block 100_{10}) of a DECTape contains directory information for all files on that tape; a maximum of 22 files can be stored on any one DECTape.

Words 0 through 82₁₀

The first 83 words of the directory contain "slots," each "slot" representing one of the 577 (blocks 1 through 1101₈ are represented in these 83 words) blocks on the DECTape. Each slot occupies five bits (seven slots are stored per word) and contains the number of the file (1-26₈) to which the block the slot represents is assigned.

Words 83 through 104₁₀

The next 22 words contain the filenames of the 22 files residing on the DECTape. Word 83 contains the filename for file #1, word 84 the filename #2, etc. Filenames are stored in 6-bit code.

Words 105 through 126₁₀

The next 22 words contain the extension names and dates of the 22 files, in the same relative order as their filenames above.

Bits 0 through 17₁₀

The extension name of the file (in 6-bit code).

Bits 18 through 23₁₀

Number of 1K blocks minus 1 needed to load the file (maximum value = 63)
This information is stored for SAVED files only.

Bits 24 through 35₁₀

The date the file was last updated, according to the formula:

$$((\text{year}-1964)*12+(\text{month}-1))*31+\text{day}-1$$
Word 127₁₀

Unused.

The message

BAD DIRECTORY FOR DEVICE DTAn: EXEC CALLED FROM USER LOC n

is produced whenever any of the following conditions are detected.

- a. A parity error while reading the directory block.
- b. No "slots" are assigned to the file number of the file.
- c. The tape block which may possibly be the first block of the file (i.e., the first block for the file encountered while searching backwards from the directory block) cannot be read.

5.7.4 DECTape File Format

A file consists of any number of DECTape blocks. Each block contains:

Word 0	Left half	The link. The link is the block number of the next block in the file. If the link is zero, this block is the last in the file.
	Right half	Bits 18 through 27: The block number of the first block of the file. Bits 28 through 35: A count of the number of words in this block which are used (maximum 177 ₈).
Words 1 through 177 ₈		Data packed exactly as the user placed in his buffer or in Dump Mode files, the next 127 words of memory. ¹

5.7.5 Special Programmed Operator Service

Several programmed operators are provided for manipulating DECTape. These allow the user to manipulate block numbers and to handle directories.

¹The Monitor compresses the user's core image by squeezing out blocks of two or more consecutive zeroes before creating the SAVed files; files with extension .SAV may be read in Dump Mode, but must be reexpanded before being run. The Monitor takes this action after input on a RUN or GET.

In addition to the operators above, INPUT, OUTPUT, CLOSE, and RELEAS have special effects. When performing nondump input operations, the DECTape service routine reads the links in each block to determine the next block to read and when to raise the end-of-file flag.

When an OUTPUT is given, the DECTape service routine examines the left half of the first data word in the output buffer (the word containing the word count in the right half). If this half contains -1, it is replaced with a 0 before being written out, and the file is thus terminated. If this half word is greater than 0, it is not changed and the service routine uses it as the block number for the next OUTPUT. If this half word is 0, the DECTape service routine assigns the block number of the next block for the next OUTPUT.

Table 5-3

DECTape Programmed Operators

Programmed Operator	Effect
USETI D, E	Sets the DECTape on device channel D to input block E next. Input operations on this DECTape must not be active because otherwise the user has no way of determining which buffer contains block E.
USETO D, E	Similar to USETI but sets the output block number. USETO waits until the device is inactive before setting up the new output block number.
UGETF D, E	Places the number of the first free block of the file in user's location E.
ENTER D, E	User's location E, E+1, E+2, and E+3, must be reserved for a directory entry. The DECTape service routine searches the directory for a filename and extension that match the contents of E and the left half of E+1. If no match is found and there is room in the directory, the service routine places the first free block number into the right half of E+1, places the date in E+2 (unless already non-zero), and places the necessary

Table 5-3 (Cont)

DECTape Programmed Operators

Programmed Operator	Effect
LOOKUP D, E error return	<p>information into the directory. If a match is found, similar actions occur, but the new entry replaces the old. If there is no room in the directory, ENTER returns to the next location. Otherwise, ENTER skips one location.</p> <p>Similar to ENTER but sets up an input file. The contents of E and E+1 are matched against the filenames and extension names in the DECTape directory. If a match is found, information about the file is read from the directory into the appropriate portions of the 4-word block beginning at E. The first block of the file is then found as follows.</p> <ol style="list-style-type: none"> 1. The first 83 words of the DECTape directory are searched in a backwards manner, beginning with the slot immediately prior to the directory block, until the first slot containing the desired file number is found. 2. The block associated with this slot is then read in and bits 18 through 27 of the first word of the block (these bits contain the block number of the first block of the file) are checked. If they are equal to the block number of this block, then this block is the first block of the file; if not, then the block with that block number is read as the first block of the file. <p>LOOKUP then skips one location. If no match is found, LOOKUP returns to the user's program at the next location.</p>
CALL D, [SIXBIT/UTPCLR/]	<p>UTPCLR clears the directory of the DECTape on device channel D. A cleared directory has zeroes in the first 83 words except in those slots related to blocks 0, 1, 2, and 100₁₀ and nonexistent blocks 1102 through 1105₈. Only the directory block (block 100) is affected by UTPCLR; the other blocks are unaffected. This programmed operator does nothing if the device on channel D is not DECTape.</p>
RENAME D, E	<p>This programmed operator is used to alter the name and extension of a file or to delete it from the DECTape. Locations E to E+3 are as in LOOKUP and ENTER. To be RENAMED a file must first be CLOSED on channel D, in order to identify for the for the RENAME UUO. RENAME then seeks</p>

Table 5-3 (Cont)

DECTape Programmed Operators

Programmed Operator	Effect
	out this file and enters the information specified in E through E+2 into the retrieval information and proper directory. If the contents of E is zero, RENAME has the effect of deleting the file. The error return is given if the new file name and extension already exist or if neither a LOOKUP nor an ENTER has been done to identify the file to be renamed.

For both INPUT and OUTPUT, block 100 (the directory) is treated as an exception case. If the user's program gives

```
USETI D, 144g
```

to read block 100, it is treated as a 1-block file.

The CLOSE operator places a '-1 in the left half of the first word in the last output buffer, thus terminating the file.

The RELEAS operator writes the copy of the directory which is normally kept in core onto block 100, but only if any changes have been made. Certain console commands, such as KJOB or CORE 0, perform an implicit RELEAS of all devices and, thus, write out a changed directory even though the user's program failed to give a RELEAS.

Two other special programmed operators are available: MTAPE D, 1 and MTAPE D, 11. MTAPE D, 1 rewinds the DECTape and moves it into the end zone at the front of the tape. MTAPE D, 11 rewinds and unloads the tape, pulling the tape completely onto the lefthand reel. These commands affect only the physical position of the tape, not the "logical" position. When either is used, the user's job can be swapped out while the DECTape is rewinding; however, the job cannot be swapped out if an INPUT or OUTPUT is done while the tape is rewinding.

5.7.6 Special Status Bits

If an attempt is made to write on a unit with the WRITE-LOCK switch on, the message

```

      DEVICE DTAn OK?
      ↑C
      .

```

is typed on the user's Teletype. When the situation has been rectified, CONT may be typed to proceed as normal.

5.7.6.1 Special DECTape Status Bits - An INIT or SETSTS to a DECTape with bit 29 ON informs the DECTape service routine that the DECTape is in nonstandard format. This implies that no file-structured operations will be performed on that tape. Blocks will be read or written sequentially; no links will be generated (output) or recognized (input). The first block to be read or written must be set by a USETI or USETO. In Dump Mode, 200₈ data words per block will be read or written (as opposed to the normal 177₈ words). No "dead reckoning" will be used on a search for a block number as the tape may be composed of blocks shorter than 200 words. The ENTER, LOOKUP, and UTPCLR UUOs are treated as no-ops. Block 0 of the tape may not be read or written in Dump Mode if bit 29 is ON, as the data must be read in a forward direction and block 0 normally cannot be read forward.

5.7.7 Important Considerations

The DECTape service routine reads the directory from a tape the first time it is required to perform a LOOKUP, ENTER, or UGETF; the directory image remains in core until a new ASSIGN command is executed from the console. To inform the DECTape service routine that a new tape has been mounted on an assigned unit, the user must use an ASSIGN command. The directory from the old tape

could be transferred to the new tape, thus destroying the information on that tape unless the user reassigns the DECTape transport every time he mounts a new reel.

5.8 MAGNETIC TAPE

Magnetic tape format is industry compatible, 7- or 9-channel 200, 556, and 800 bpi (see description below).

Device Mnemonic - MTA0, MTAl, ..., MTA7

Buffer Size - 203₈ words

5.8.1 Data Modes

5.8.1.1 A (ASCII) - Data appears to be written on magnetic tape exactly as it appears in the buffer. No processing or check-summing of any kind is performed by the service routine. The parity checking of the magnetic tape system is sufficient assurance that the data is correct. Normally, all data, both binary and ASCII, is written with odd parity and at 556 bits per inch. A maximum of 200₈ words per record is standard. The word count is not written on the tape.

5.8.1.2 AL (ASCII Line) - Same as A.

5.8.1.3 I (Image) - Same as A but data consists of 36-bit words.

5.8.1.4 IB (Image Binary) - Same as I.

5.8.1.5 B (Binary) - Same as I.

5.8.1.6 DR (Dump Records) - Standard fixed length records (128

words is the standard unless installation standard is changed with MONGEN) are read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is as described in Chapter 4, "Unbuffered (Dump) Modes." For input operations a new record is read for each word in the command list (except GOTO words); if the record terminates before the command word is satisfied, the service routine reads the next records. If the command word runs out before the record terminates, the remainder of the record is ignored. For each output command word, as many standard length records are written followed by one short record to exactly write all of the words on the tape.

5.8.1.7 D (Dump) - Variable length records are read into or written from anywhere in the user's core area without regard to the standard buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is as described in Chapter 4, "Unbuffered (Dump) Modes." For input operations a new record is read for each word in the command list (except GOTO words); if the record terminates before the command word is satisfied, the service routine skips to the next command word. If the command word runs out before the record terminates, the remainder of the record is ignored. For each output command word, exactly one record is written. See Section 4.4.1.2 for command list format.

5.8.2 Magnetic Tape Format

Magnetic tape format can be generally described as unlabelled, industry compatible format. That is, as far as the user

is concerned, the tape contains only data records and end-of-file marks which signal the end of the data set or the end of the file. Files are read from and written on the tape in a sequential manner.

An end-of-file mark consists of a record containing a 17₈ (for 7-channel tapes) or a 23₈ (for 9-channel tapes). End-of-file marks are used in the following manner.

- a. No end-of-file mark precedes the first file on a magtape.
- b. An end-of-file mark follows every file.
- c. Two end-of-file marks follow a file if that file is the last or only file on the tape.

Files are written on and read from a magtape in a sequential manner. A file consists of an integral number of physical records, separated from each other by interrecord gaps (area on tape in which no data is written). There may or may not be more than one logical record in each physical record.

5.8.3 Special Programmed Operator Service

CLOSE performs a special function for magnetic tape. When an output file is closed (both dump and nondump), the I/O service routine automatically writes two end-of-file marks and backspaces over one of them. If another file is now opened, the second end-of-file is wiped out leaving one end-of-file between files. At the end of the in-use portion of the tape, however, there appears a double end-of-file character which is defined as the logical end of tape. When an input dump file is closed, the I/O service routine automatically skips to the next end-of-file.

A special programmed operator called MTAPE provides for such tape manipulation functions as rewind, backspace record, backspace file, 9-channel tape initialization, etc. The format is

MTAPE D, FUNCTION

where D is the device channel on which the magnetic tape unit is initialized. FUNCTION is selected according to the following table:

Table 5-4

MTAPE Functions

Function	Action
0	No operation; wait for spacing and I/O to finish
1	Rewind to load point
11	Rewind and unload ¹
7	Backspace record
17	Backspace file
3	Write end of file
6	Skip one record
13	Write 3 inches of blank tape
16	Skip one file
10	Space to logical end of tape
100	Digital Compatible; 9-channel ²
101	Initialize for 9-channel tape ³

MTAPE waits for the magnetic tape unit to complete whatever action is in progress before performing the indicated function, including

¹On the 516 Control, this function is not currently implemented as such, but is treated as a Rewind function only.

²Digital Compatible mode writes (or reads) 36 data bits in five frames of a 9-track magtape. It can be any density, any parity, and is not industry compatible. This mode is in effect until a RELEAS D, or an MTAPE D, 101 is executed.

³Industry compatible 9-channel mode writes (or reads) 32 data bits per word in four frames of a 9-track magtape and ignores the last four bits of a word. It must be 800 bpi density, odd parity.

no operation (0). Bits 18 through 25 of the status word are then cleared, the indicated function is initiated, and control is returned to the user's program immediately. It is important to remember that when performing buffered input/output, the I/O service routine can be reading several blocks ahead of the user's program. MTAPE affects only the physical position of the tape and does not change the data that has already been read into the buffers.

5.8.3.1 Backspace File on Magtape - Issuing a backspace file command to a magtape unit will move the tape in the reverse direction until the tape has A) passed the end of file mark or B) reached the beginning of the tape. This means that the end of the backspace file operation will position the tape heads either immediately in front of a file mark or at the beginning of the tape.

In most cases it is desirable to skip forward over this file mark. This is decidedly not the case if you've reached the beginning of the tape; in this case giving a skip file command would indeed skip the entire first file on the tape stopping at the beginning of the second file, rather than leaving the tape positioned at the beginning of the first file.

Therefore a typical (incorrect) sequence for backspace file would be:

```
MTAPE  MT, 17      ;Backspace file
CALLI WAIT          ;*Wait for completion*
STATO  MT, 4000    ;Beginning of tape?
MTAPE  MT, 16      ;No, skip over file mark
```

Note that it is necessary to wait after the backspace file instruction in order to insure that the tape is moved to the end-of-file mark or the beginning of the tape before testing to see whether or not it is the beginning of the tape. The instruction CALLI WAIT cannot be used for this purpose; it waits only for the

completion of I/O transfer operation. (Backspace file is a spacing operation, not an I/O transfer operation.)

Instead, use the following sequence for backspace file:

```
MTAPE  MT, 17      ;Backspace file
MTAPE  MT, 0       ;Wait for completion
STATO  MT, 4000   ;Beginning of tape?
MTAPE  MT, 16     ;No, skip over file mark
```

In this case the device service routine must wait until the magtape control is free before processing the MTAPE MT, 0 command, which tells the tape control to do nothing. Thus, the service routine achieves the waiting period necessary for the completion of the previous operation and the proper positioning of the tape.

5.8.4 9-Channel Magtape

Nine-channel magtape may be written and read in two ways: normal Digital Compatible format, and industry compatible format.

5.8.4.1 Digital Compatible Mode - Digital Compatible mode is the usual mode and will allow old 7-channel user mode programs to read and write 9-channel tapes with no modification. Digital Compatible mode writes 36 data bits in five bytes of a nine track magtape. It can be any density, and parity, and is not industry compatible. The software mode is specified in the usual manner during initialization or with a SETSTS. User mode I/O is handled precisely as in the case of 7-track magtape. It is assumed that most DEC magtapes will be written and read this way.

Data Word on Tape

Tracks								
9	8	7	6	5	4	3	2	1
B0	B1	B2	B3	B4	B5	B6	B7	P
B8	B9	B10	B11	B12	B13	B14	B15	P
B16	B17	B18	B19	B20	B21	B22	B23	P
B24	B25	B26	B27	B28	B29	(B30)	(B31)	P
0	0	(B30)	(B31)	B32	B33	B34	B35	P

P = Parity
BN = Bit N in core

Data Word in Core - 5 magtape bytes/36-bit word. Parity bits are unavailable to the user. Bits are written on tape as shown in diagram, note that bits 30 and 31 get written twice and that tracks 8 and 9 of byte 5 contain 0. On reading parity bits and tracks 8 and 9 of byte 5 are ignored, the or of bits (B30) is read into bit 30 of the data word, the or of bits (B31) is read into bit 31.

5.8.4.2 Industry Compatible Mode - For reading and writing industry compatible 9-channel magtapes, an MTAPE D, 101 UUO must be executed to set the status. MTAPE D, 101 is meaningful for 9-channel magtape only and is ignored for all other devices. In the left half of the status word, bit 2 (which cannot be read by the user program) may be cleared (which returns the device to 9-channel Digital Compatible status) by a RELEAS, a call to EXIT, or an MTAPE D, 100 UUO. These MTAPE UUO's act only as a switch to and from industry compatible mode and in no other way affect I/O status, except to set the density to 800 bpi and odd parity.

On INPUT, four 8-bit bytes are read into each word in the buffer, left justified with the remaining four bits of the word containing error checking information.

On OUTPUT, the leftmost four 8-bit bytes of each word in the buffer are written out in four frames, with the remaining four rightmost bits of the word being ignored.

Data Word on Tape

Tracks								
9	8	7	6	5	4	3	2	1
B0	B1	B2	B3	B4	B5	B6	B7	B32
B8	B9	B10	B11	B12	B13	B14	B15	B33
B16	B17	B18	B19	B20	B21	B22	B23	B34
B24	B25	B26	B27	B28	B29	B30	B31	B35

Data Word in Core - four magtape bytes carry 4 8-bit bytes from data word, parity bits are obtained as shown when reading. Rightmost four bits are ignored on writing (bits 32-35).

5.8.4.3 Changing Modes - MTAPE CH, 101 automatically sets density at 800 bits (or 800 eight-bit bytes) per inch and sets odd parity. Note that buffer headers are set up when necessary by the Monitor in the usual manner according to the I/O mode the device is initialized in. Byte pointers and byte counts in buffer header will have to be changed by the user in order to operate on eight-bit bytes.

5.8.5 Special Status Bits

Special bits of the status word are reserved for selecting the density and parity mode of the magnetic tape. Table 5-4 lists the bits that are set and cleared by INIT or SETSTS.

Table 5-5

Magnetic Tape Special Status Bits.

Bit	Action
18 ¹	Improper mode. When set to one during an output operation means that the write enable ring is out.
24 ¹	I/O Beginning of Tape. The tape is at the load point.
25 ¹	I/O Tape END. The tape is at or past the end point.
26	I/O Parity. 0 for odd parity, 1 for even parity ²
27-28	I/O Density. 00 = System Standard (defined at MONGEN time) 01 = 200 bpi 10 = 556 bpi 11 = 800 bpi
29	I/O No Read Check. Suppress automatic error correction if bit 29 is a 1. Normal error correction is to repeat the desired operation 10 times before setting an error status bit.

¹These bits indicate special magnetic tape conditions and are set by the magnetic tape service routine when the conditions occur.

²Odd parity is preferred. Even parity should be used only when creating a tape to be read in BCD (Binary Coded Decimal) on another computer.

5.9 DISK

Device Mnemonic - DSK

Buffer Size - 203₈ words (of which 200₈ words are data)5.9.1 Data Modes

5.9.1.1 A (ASCII) - Data is written on the disk exactly as it appears in the buffer. Data consists of 36-bit words.

5.9.1.2 AL (ASCII Line) - Same as A.

5.9.1.3 I (Image) - Same as A.

5.9.1.4 IB (Image Binary) - Same as I.

5.9.1.5 B (Binary) - Same as I.

5.9.1.6 DR (Dump Records) - Functions exactly the same as D.

5.9.1.7 D (Dump)- Data is read into or written from anywhere in the user's core area without regard to the normal buffering scheme. Control for read or write operations must be via a command list in core memory. The command list format is as described in Chapter 4, "Unbuffered (Dump) Modes." The disk control automatically measures dump data into standard-length disk blocks of 200 octal words. Unless the number of data words is an exact multiple of the standard length of a disk block (200 words) after each command word in the command list, the remainder of that block is wasted.

5.9.2 Structure of Files on Disk

The file structure of the disk system has been designed to minimize the number of disk seeks for sequential or random accessing using either buffered or dump mode I/O. The assignment of physical space for data is performed automatically by the Monitor as logical files are written or deleted by user programs. Files may be of any length, and each user may have as many files as he wishes, as long as disk space is available. No initial estimate of file length or number of files need be given by users or their programs. Files may be simultaneously read by more than one

user at a time, thus allowing data sharing. A new version of a file may be recreated by one user while other users continue to read the old version, thus allowing for smooth replacement of shared programs and data files. Finally, one user may selectively update portions of a file, rather than creating a new one (see "General Notes," 5.9.3.3).

5.9.2.1 Addressing by Monitor - The file structure described in this section is generally transparent to the user, and a detailed knowledge of this material is not essential for effective user-mode use of the disk. There are two programs in the Time-Sharing Monitor that service the disk, DKSER and DSKINT. DKSER is the device service routine for a disk and references a disk by symbolic addressing only. This routine is essentially independent of what physical disk is attached to the system. DSKINT serves only two functions: 1) that of translating the logical addressing used elsewhere in the system to the physical addressing of the particular disk being utilized, and 2) controlling the physical disk. The Monitor can be thought of as seeing all disks in the same manner; a change of disks requires only a change in DSKINT to provide the proper software interface between the physical device and the rest of the system.

All references made herein to addresses on the disk refer to the logical or relative addresses used by the system and not to any physical addressing scheme involving records, sectors, tracks, etc., that may pertain to a particular physical device. The basic unit which may be addressed is a logical disk block which consists of 200_8 36-bit words.

5.9.2.2 Storage Allocation Table (SAT) Blocks - There is a

storage allocation table on the disk, which reflects the current status of every addressable block on the disk. These SAT blocks are contained in a file with the name "*SAT* .SYS". This file may be used by any user, but can only be modified by the Monitor. Each addressable block on the disk is represented by one particular bit within the SAT blocks.

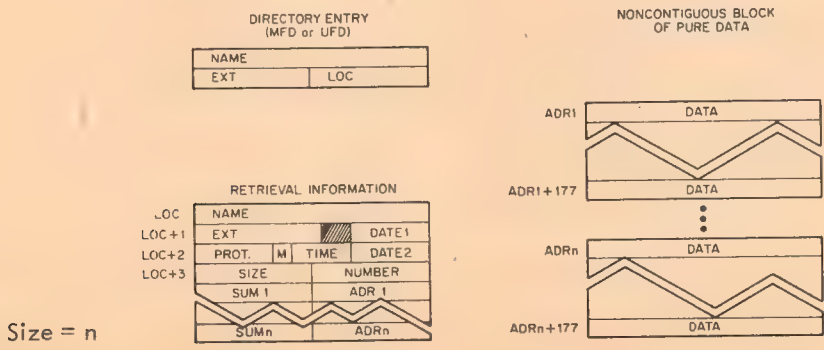
If a particular bit is on, it indicates that the corresponding block is filled with data (all blocks on the disk are filled when any information is written on them); if the bit is off, it indicates that the corresponding block is empty or available to be written on. The disk can be wiped out by zeroing the SAT blocks (which is exactly what is done when the disk is refreshed). The disk may optionally be "refreshed" whenever the Monitor is reloaded.

5.9.2.3 File Directories - There are two levels of directories on the disk; one is referenced mainly by the system and the other is referenced by individual users. There is only one higher level directory, known as the Master File Directory (MFD). One of the functions of the MFD is to serve as a directory for individual User's File Directories (UFD's). A UFD is a particular user's own directory and will contain the names of files he has written on the disk. The UFD itself is a file like any other file except that its filename is a binary number combination (project-programmer) rather than a 6-bit code and its extension is always UFD in SIXBIT. The binary combination consists of a left half, which is the project number, and a right half, which is called the programmer number. When a user is logged in under a specific project-programmer number and references the disk, he is actually referencing his own area through the UFD having his project-programmer number as its name. He may, of course, specifically code his

routine to reference files listed in the UFD's of other users or the MFD; whether he is successful or not will then depend upon the type of protection that has been specified for the file he is trying to reference.

5.9.2.4 File Format - All disk files (including MFD and UFDs) are composed of two parts: 1) pure data, and 2) information needed by the system to retrieve this data. Each data block contains exactly 200 (octal) words. If a partially filled buffer is output to the disk by a user, a full block is written with trailing zeros filling in to make 200_8 words. Word counts associated with individual blocks are not retained by the system. If such a partial block is input later, it will appear to have a full 200_8 data words.

There are three links in the chain by which the system references data on the disk. The first link is the 2-word directory entry in the UFD, which points to the Retrieval Information block(s), which in turn points to the individual pure data blocks. This chain is transparent to the user, who may look upon the directory as having 4-word entries analogous to DECTapes.



Directory Entry

- NAME - Filename in 6-bit ASCII, unless the directory is the MFD and the file is a UFD; in that case, NAME is a project-programmer number in binary.
- EXT - Filename extension in 6-bit ASCII; if NAME is a project-programmer number, EXT is UFD.
- LOC - Address of the first block on the disk that contains Retrieval Information for this file.

Retrieval Information

NAME and EXT as above; used to check hardware for possible read error, and to check against software malfunctions. (A failure to match NAME and EXT results in the message "INCORRECT RETRIEVAL INFORMATION".)

- DATE1 - In format of DATE UUO; date file last referenced (RENAME, or ENTER, or INPUT done) (bits 24-35).
- DATE2 - Same format as DATE1; date file originally created (ENTER) (bits 24-35).
- PROT. - Protection; see below (bits 0-8).
- M - Data Mode (ASCII, Binary, Dump, etc.) (bits 9-12).
- TIME - 24-hour time (in minutes) that file was originally created (bits 13-23).
- SIZE - If negative, this portion indicates the number of words in the file, where all blocks with the possible exception of the last are assumed to contain a full 200_8 words. If positive, this is a count of the number of 200_8 -word blocks contained in the file. For files of less than 2^{17} words, the negative word count is used; for larger files, the positive block count is used instead.
- NUMBER - Programmer Number.

SUM1, - Checksum; two's complement, end-around-carry, sum of
 ...SUMn data in data-block whose disk address is ADRL.

ADRL, - Address of data block (logical block number on disk).
 ...ADRn

Protection

The first nine bits of the third word of a file's retrieval information are used to specify the protection of the file. This is a necessary procedure since the disk is shared by many users, who may each desire to keep certain files from being written over, read, or deleted by other users.

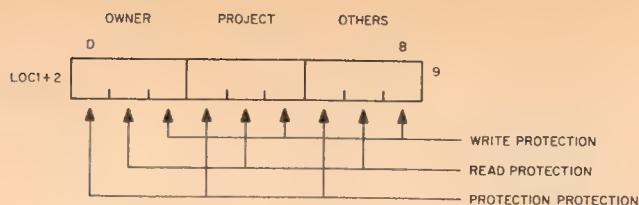
The total number of users falls into three categories:

- a. Owner of file (person whose programmer number is the same as that in the right half of the NAME field of the UFD in which the file is entered).
- b. Project members (users whose project number is the same as that in the left half of the NAME field of the UFD in which the file is entered).
- c. All other users.

There are three types of protection against each of the three categories of users:

- (1) Protection - The protection itself cannot be altered.
- (2) Read protection - The file may not be read.
- (3) Write Protection - The file may not be re-written, renamed, or deleted.

The protection mask (see above) consists of the first nine bits of the third word of retrieval information; each bit (when on) represents a particular type of protection against a specific category of user, according to the following scheme. However, owner protection-protection and owner read-protection are ignored lest the file become totally inaccessible.



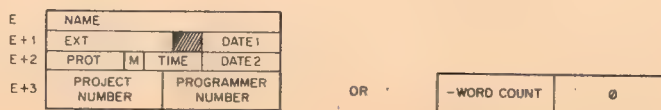
All files created with an ENTER are given the protection, 055₈ by the Monitor; if some other protection mask is desired, the the RENAME UUO must be employed by the user. (Also see Section 4.4.2.5, "File Protection".)

5.9.3 User Programming for the Disk

5.9.3.1 Format - The actual file structure of the disk is generally transparent to the user. In programming for input/output on the disk, a format analogous to that of DECTapes is used; that is, the user assumes a 4-word directory entry similar in form to the first four words of retrieval information. The UUO format is approximately the same as for DECTapes:

UUO D, E

Where UUO is an input/output programmed operator and D specifies the user channel associated with this device. E points to a 4-word directory entry in the user's program which has the following format:



(Note that E+3 differs from the fourth word of retrieval information. See Retrieval Information, 5.8.2.4 for description.)

5.9.3.2 Special Functions of Programmed Operators (UUO's) -

ENTER D,E
error return

Causes the Monitor to store away the 4-word directory entry for later entry into the proper UFD when user channel D is CLOSED or RELEASED.

NAME - The filename must be non-zero; if not, an error return results.

EXT - The file extension may be zero; if so, the Monitor will leave it zero.

DATE1 - The correct date is always filled in by the Monitor.

PROT - The protection is always supplied by the Monitor as 055. The RENAME may be used to change protection after file has been completely written and a CLOSE done.

M - The data mode is supplied by the Monitor as set by the user in the last INIT, or SETSTS UUO on channel D.

TIME, DATE2 - If both of these are 0, the Monitor supplies the current date and time as the creation date and time for the file. If either is non-zero, the Monitor will use the TIME and DATE2 supplied by the user in E+2; thus files may be copied without changing the original creation time and date.

PROJECT-NUMBER, PROGRAMMER-NUMBER - If both of these are 0, the project-number and programmer-number (binary) under which the user is logged-in is supplied by the Monitor. Otherwise the Monitor will use the project-number and programmer-number supplied by the user in E+3; however, it is generally not possible to create (ENTER) files in another

user's area of the disk, since UFDs are usually write-protected against all but the owner.

With certain types of error returns peculiar to the disk, the right half of E+1 is set to a specific number to indicate which type of error caused the return. These numbers have the following significance:

- 0 - E contained a zero file name
- 1 - E+3 contained an incorrect (or nonexistent) project-programmer number.
- 2 - File already exists, but is write-protected.
- 3 - File was being created, re-created, updated, or renamed.

No user, except an administrator with project number 1, may create a UFD, since the MFD is universally write-protected. The LOGIN CUSP (running under the administrator project number) creates a UFD for any user the first time he logs into the system.

When an ENTER is executed by the Monitor on a file that already exists, a new file by that name is written, and those bits in the SAT blocks that correspond to the blocks of the old file are zeroed when the CLOSE (or RELEAS) UUO is executed, thereby retrieving space and making it available to any other user.

LOOKUP D, E
error return

Causes the Monitor to read the appropriate UFD. If a later version of the file is being written, the old version pointed to by the UFD will be read.

NAME - The filename in SIXBIT.

EXT - The file extension in SIXBIT. A zero extension is not treated in any special manner.

DATE1, PROT, M, TIME, DATE2 are ignored. The Monitor returns these quantities to the user in E+1 and E+2.

PROJECT-NUMBER, PROGRAMMER-NUMBER - If both of these are 0, the project-number and programmer-number (binary) under which the user is logged-in is supplied by the Monitor. Otherwise the Monitor will use the project-number, programmer-number supplied by the user in E+3. Thus, it is possible to read files in other user's directories, provided that the file's protection mask permits reading. The Monitor returns the negative word count (or positive block count for large files) in the LH of E+3, 0 in RH or E+3.

The numbers placed by the Monitor in the right half of E+1 upon an error return have a significance analogous to that described for the ENTER UO:

- 0 - File was not found
- 1 - Incorrect project-programmer number in E+3
- 2 - Protection failure
- 3 - File was being created (no earlier version existed).

If the file is currently being re-created, the old file is used.

RENAME D, E
error return

This programmed operator is used to alter the name, extension, and/or protection of a file or to delete a file from the disk. Locations E through E+3 are as described above. RENAME is the only UO that

can set the protection of a file to that specified in E+2. To be RENAMED a file must first be CLOSED on channel D, in order to identify for the RENAME UUO. RENAME then seeks out this file and enters the information specified in E through E+2 into the retrieval information and proper directory. If the contents of E is zero, RENAME has the effect of deleting the file.

The error return numbers in the right half of E+1 are the same as for ENTER, with the added possibilities:

4 - Tried to RENAME file to already-existing name.

5 - Neither LOOKUP nor ENTER has been done to identify the file to be renamed.

USETO D, A

USETI D, A

These programmed operators are treated identically by the disk service routines. Their function is to notify the service routine that a particular block is to be used on the next INPUT or OUTPUT on channel D. A is a number that designates a particular block relative to the beginning of the file. If A is greater than the current size of the file (in blocks), the next OUTPUT will write a block immediately after the file; the next INPUT will cause the end-of-file flag to be set. A=1 refers to the first block of the file (i.e., block 0).

If A = 0 or if no previous LOOKUP or ENTER has been done, this UUO will set the improper mode error bit (see bit 18, Table 4-4, and Section 4.4.4).

5.9.3.3 General Notes - Three types of "writing" on the disk may be distinguished. If a user does an ENTER with a filename which did not previously exist in his UFD, he is said to be "creating" that file. If the filename did previously exist in his UFD, he is said to be superseding that file; the old version of the file stays on the disk (and is available to anyone who wants to read it) until the user does the output CLOSE (at this point, his UFD is changed to point to the new version of the file and the old version is either deleted immediately or marked for deletion later if someone is currently reading it; the space occupied by deleted files is always reclaimed in the SAT tables - see Section 5.8.2.2). Finally, if a user does a LOOKUP followed by an ENTER (the order is important) on the same filename on the same user channel, he will be able to modify selected blocks of that file, using USETO and USETI UUOs, without creating an entirely new version of it; this third type of writing is called "updating" and eliminates the need to copy a file when making only a small number of changes.

As a standard practice, user programs should read, create, and supersede (new file with same filename) files on different user channels. However, for compatibility with DECTapes, it is possible to read and create, or read and supersede, two files on the same user channel as long as all OUTPUTs and the CLOSE output are done before the LOOKUP and the first input, or vice versa. In other words, a CLOSE UUO is required between successive LOOKUPS and ENTERS unless updating is intended.

When issuing a RENAME UUO, the user must insure that the status at locations E through E+3 are as he desires them to be. Since an ENTER or LOOKUP, as well as CLOSE, must have preceded the RENAME; the contents of E through E+3 will have been altered, or filled if the E is the same for all UUO's.

CALL [SIXBIT/RESET/] - Any files which are in the process of being written, but have not been CLOSED or RELEASed, will be deleted and the space reclaimed. If a previous version of the file with the same name and extension existed, it will remain on the disk (and in the UFD) unchanged.

If the programmer wants to retain the newly created file and have the older version deleted, he must CLOSE or RELEAS the file before doing a RESET UO.

5.10 INCREMENTAL PLOTTER

Device Mnemonic - PLT

Buffer Size - 43 (octal) words

The plotter takes 6-bit characters with the bits of each character decoded as follows:

		-X	+X	+Y	-Y
Pen	Pen	Drum	Drum	Car-	Car-
Raise	Lower	Up	Down	riage	riage
				Left	Right

Do not combine pen raise or lower with any of the position functions. (For more details on the incremental plotter, see the PDP-10 System Reference Manual, DEC-10-HGAA-D.)

5.10.1 Data Modes

5.10.1.1 A (ASCII)

Five, 7-bit characters per word are transmitted to the plotter exactly as they appear in the buffer. Since the plotter is a 6-bit device, the leftmost bit of each character is ignored.

- 5.10.1.2 AL (ASCII LINE) This mode is identical to the A mode.
- 5.10.1.3 I (IMAGE) Six, 6-bit characters per word are transmitted to the plotter exactly as they appear in the buffer.
- 5.10.1.4 B (BINARY) This mode is identical to the I mode.
- 5.10.1.5 IB (IMAGE BINARY) This mode is identical to the I mode.
- 5.10.1.6 DR (DUMP RECORDS) Not available.
- 5.10.1.7 D (DUMP) Not available.

The first OUTPUT operator causes the plotter pen to be lifted from the paper before any user data is sent to the plotter. The CLOSE operator causes the plotter pen to be lifted after all user data is sent to the plotter. These two pen-up commands are the only modifications the Monitor makes to the user output file.

5.11 DISPLAY WITH LIGHT PEN (TYPE 30 and TYPE 340)

Device Mnemonic - DIS

Buffer Size - None (uses device-dependent dump mode only - 15)

5.11.1 Data Modes

5.11.1.1 ID (IMAGE DUMP - 15)

An arbitrary length area in the user area may be displayed on the scope. The command list format is as described in Chapter 4, "Unbuffered (Dump) Modes," with the addition for the Type 30 display, that, if $RH = 0$, and $LH \neq 0$, then LH specifies the intensity for the following data (4 to 13).

5.11.2 Background

The purpose of the Monitor service routine for the VR-30 is to maintain a flicker-free picture on the display during time-sharing. To do this, the picture data must be available for display at least every two jiffies. This necessitates that the display data remain in core. At present, this means that the user program must also remain in core. To minimize swapping of other programs and to make available a larger block of free core for other users, the user program is shuffled toward the top of core between pictures.

5.11.3 Display UUU's

The input/output UUU's for both displays operate as follows:

INIT D, 15	;MODE 15 ONLY
SIXBIT /DIS/	;DEVICE NAME
0	;NO BUFFERS USED
ERROR RETURN	;DISPLAY NOT AVAILABLE
NORMAL RETURN	
CLOSE D,	;STOPS DISPLAY AND
or	;RELEASES DEVICE AS
RELEAS D,	;DESCRIBED IN MANUAL

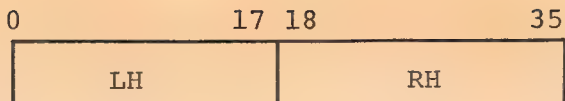
5.11.3.1 INPUT D, ADR

If a light pen hit has been detected since the last INPUT command, then C(ADR) is set to the location of last light pen hit.

If no light pen hit has been detected since last INPUT command, then C(ADR) is set to -1.

5.11.3.2 OUTPUT D, ADR

ADR specifies the first address of a table of pointers. This table is composed of pointers with the following format:



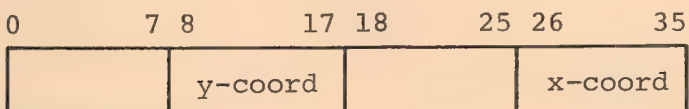
For the VR-30 Display:

If LH = 0 and RH = 0, then this is the end of the command list.

If LH \neq 0 and RH = 0, then LH is the desired intensity for the following data or commands. The intensity ranges from 4 to 13, where 4 is the dimmest and 13 is the brightest.

If LH = 0 and RH \neq 0, then RH is the address of the next pointer. Successive pointers are interpreted beginning at RH.

If LH \neq 0 and RH \neq 0, then -LH words beginning at address RH+1 are output as data to the display. The format of the data word is the following:



For the 340 Display:

If RH = 0, then this is the end of the command list.

If LH = 0 and RH \neq 0, then RH is the address of the next pointer. Successive pointers are interpreted beginning at RH.

If LH \neq 0 and RH \neq 0, then -LH words beginning at address RH+1 are output as data to the display. The format of the data word is described in the 340 programming manual.

An example of a valid pointer list for the VR-30 Display is:

	OUTPUT	D, LIST	;OUTPUT DATA
LIST:	XWD	5, 0	;POINTED TO BY LIST
	IOWD	1, A	;INTENSITY 5 (DIM)
	IOWD	5, SUBP1	;PLOT A
	XWD	13, 0	;PLOT SUBPICTURE 1
	IOWD	1, C	;INTENSITY 13 (BRIGHT)
	IOWD	2, SUBP2	;PLOT C
	XWD	0, LIST1	;PLOT SUBPICTURE 2
			;TRANSFER TO LIST 1
LIST1:	XWD	10, 0	;INTENSITY 10 (NORMAL)
	IOWD	1, B	;PLOT B
	IOWD	1, D	;PLOT D
	XWD	0, 0	;END OF COMMAND LIST

	OUTPUT	D, LIST	; OUTPUT DATA
A:	XWD	6,6	;Y= 6, X=6
B:	XWD	70,105	;Y= 70, X=105
C:	XWD	105,70	;Y= 105, X=70
D:	XWD	1000,200	;Y=1000, X=200
SUBP1:	BLOCK	5	;SUBPICTURE 1
SUB2:	BLOCK	2	;SUBPICTURE 2

An example of a valid pointer list for the 340 Display is:

	OUTPUT	D, LIST	; OUTPUT DATA POINTED ; TO BY POINTER IN LIST
LIST:	IOWD	1,A	;SET STARTING POINT TO (6,6)
	IOWD	5,SUBP1	;DRAW A CIRCLE
	IOWD	1,C	;SET STARTING POINT TO (70,105)
	IOWD	5,SUBP1	;DRAW A CIRCLE
	IOWD	1,B	;SET STARTING POINT TO (105,70)
	IOWD	2,SUBP2	;DRAW A TRIANGLE
	IOWD	0,LIST1	;TRANSFER TO LIST1
LIST1:	IOWD	1,D	;SET STARTING POINT TO ;(1000,-200)
	IOWD	5,SUBP1	;DRAW A CIRCLE
	IOWD	1,A	;SET STARTING POINT TO (6,6)
	IOWD	2,SUBP2	;DRAW A TRIANGLE
	XWD	0,0	;STOP
A:	X=6	Y=6	
B:	X=105	Y=70	
C:	X=70	Y=105	
D:	X=1000	Y=-200	
SUBP1:	BLOCK	5	;DRAW A CIRCLE
SUBP2:	BLOCK	2	;DRAW A TRIANGLE

The example shows the flexibility of this format. The user can display a subpicture by merely setting up a pointer to it. He can also display the same subpicture in many different places by setting up pointers to the subpicture, each preceded by a pointer to commands for the display to reset its coordinates.

5.12 CALL AC, [SIXBIT/DEVCHR/] or CALLI AC, 4

The user may determine the physical characteristics associated with a logical device name by executing a DEVCHR UUO. The DEVCHR UUO returns the following information in the AC

referred.

(AC) _L :	1	Device can do output
	2	Device can do input
	4	Device has a directory (DTA or DSK)
	10	Device is a TTY
	20	Device is a magnetic tape
	40	Device is available to this job or is already assigned to this job
	100	Device is a DEctape
	200	Device is a paper tape reader
	400	Device is a paper tape punch
	1000	Device has a long dispatch table (that is, UUO's other than INPUT, OUTPUT, CLOSE, and RELEAS perform real actions)
	2000	Device is a display
	4000	TTY in use as an I/O device
	10000	TTY in use as a user console (even if detached)
	20000	TTY attached to a job
	40000	Device is a line printer
	100000	Device is a card reader
	200000	Device is a disk
	400000	DEctape directory is in core (this bit is cleared by an ASSIGN or DEASSIGN command to that unit)
(AC) _R :	400000	Device assigned by a console command
	200000	Device assigned by program (INIT UUO)
Remaining Bits:		If bit 35-n contains a 1, then mode n is legal for the device.

NOTE

The mode number (0 through 17) must be converted to decimal; for example, mode 17₈ is represented by bit 35-15₁₀ or bit 20.

APPENDIX 1

DECtape Compatibility Between DEC Computers

	PDP 4	PDP 5	PDP 6	PDP 7	PDP 8	PDP 8	PDP 8/1	PDP 9	PDP 10
Read By	550& 555 TU55	552& 555 TU55	551& 555& TU55	550& 555 TU55	552& 555 TU55	TC01 & TU55	TC01 & TU55	TC01 & TU55	TC01 & TU55
PDP-4	A	D	D	A	D	D	D	D	D
PDP-5	D	A	B	C	A	A	A	A	A
PDP-6	D	A	A	C	A	A	A	A	A
PDP-7	A	C	C	A	C	C	C	C	C
Written By									
PDP-8	D	A	B	C	A	A	A	A	A
552									
PDP-8	D	A	B	C	A	A	A	A	A
TC01									
PDP-8/1	D	A	B	C	A	A	A	A	A
PDP-9	D	A	A	C	A	A	A	A	A
PDP-10	D	A	A	C	A	A	A	A	A

A = Can be done

B = Can not be done because of difference in writing checksum

C = Can be done with programmed checksum

D = Can probably be done as in (C) except that PDP-4 is too slow for calculating the exclusive or checksum in line - this must be done before writing.

NOTE: PDP-10 will not allow search to find first or last blocks when searching from the end-zone.

APPENDIX 2

Size of Multiprogramming non-disk Monitor (Reentrant 4 series, Version 50) June, 1969

There are three components to the Monitor:

- 1) Required code (4.7K)
- 2) Optional device code (0-4.4K)
- 3) Tables and buffers per job (73 words per job)

A. Required code (Assuming all features)

Lower core	96.
COMMON	409.
CLKCSS	82.
CLOCK1	367.
COMCON	1322.
CORE1	182.
DLSINT	48.
ERRCON	214.
SCNSRF	1260.
SEGCON	602.
SYSINT	78.
UUOCON	1144.

4692. words (Decimal)

B. Optional devices	Complete system
DTA	1284. +N(1)*146. N(1) = 8 2612.
MTA	452. +N(2)*9. N(2) = 2 470.
PTY	176. +N(3)*10. N(3) = 2 196.
CDR	220. 220.
CDP	308. 308.
DIS	190. 190.
LPT	100. 100.
PLT	65. 65.

Optional devices		Complete system	
PTP	167.		167.
PTR	105.		105.
	<hr/>		
	3067.	+N(1)=146.+N(2)*9.N(3)*10.	4433.

C. Tables and buffers

18. words of tables per job

55. word of TTY device data block space per job

73. words per job

Total for complete 8 user system = 4692. + 443. + 8.*73. = 9709.

WARNING: The Monitor will continue to grow despite our best efforts to prevent it. Most new features are put in with conditional assembly so that a customer can reduce this size of the Monitor by giving up some of the new features.

These sizes are subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

APPENDIX 3

Size of Swapping Monitor (Reentrant 4 series, Version 50) June, 1969

There are three components to the Monitor:

- 1) Required code (10K)
- 2) Optional device code (0-4K)
- 3) Tables and buffers per job (1K for every 8 jobs)

A. Required code (Assuming all features)

Lower core	96.
COMMON	475.
CLOCK 1	376.
COMCON	1592.
CORE1	214.
DLSINT	48.
DSKINT	130.
DSKSRB	2448.
ERRCON	211.
SCHEDB	741.
SCNSRF	1264.
SEGCON	709.
SYSINI	81.
UUOCON	1190.

10375. words (Decimal)

B. Optional devices	Complete system		
DTA	1286.	+N(1)*146.	N(1) = 8 2454.
MTA	452.	+N(2)*9.	N(2) = 2 470.
PTY	166.	+N(3)*10.	N(3) = 2 196.
CDR	220.		220.
CDP	308.		308.
DIS	191.		191.
LPT	104.		104.

Optional Devices		Complete system
PLT	80.	80.
PTP	167.	167.
PTR	105.	105.
	<hr/>	<hr/>
	3089. +N(1)=146.+N(2)*9.+N(3)*10.	4295.

C. Tables and buffers

- 21. words of tables per job
- 54. words of DSK device data block space per job
(1.5 files/job)
- 55. word of TTY device data block space per job

- 130. words per job

Total for complete 16 user system = 10375. + 3987. + 16.*130. = 16442.

WARNING: The Monitor will continue to grow despite our best efforts to prevent it. Most new features are put in with conditional assembly so that a customer can reduce this size of the Monitor by giving up some of the new features.

For a complete Swapping System (all devices):

8	JOBS	15.7K
16	JOBS	16.7K
24	JOBS	17.7K
32	JOBS	18.7K
40	JOBS	19.7K
48	JOBS	20.7K
56	JOBS	21.7K
64	JOBS	22.7K

These sizes are subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation.

Writing Reentrant User Programs

The LOADER simplification makes it somewhat more difficult to define variables and arrays. The easiest way to define them so that the resulting relocatable binary can be loaded on a one- or two-segment machine is to put them all in a separate subprogram as internal global symbols using Block 1 and Block N pseudo-ops. All other subprograms must refer to this data as external global locations. Most reentrant programs will have at least two subprograms, one for the definition of low segment locations and one for instructions and constants for the high segment. (This last subprogram must have a HISEG pseudo-op.) Since programs are self-initializing, they clear the low segment when they are started even though the Monitor clears core whenever it assigns it to a user.

Using Block 1 and Block N pseudo-ops causes the LOADER to leave indications in the Job Data area (LH of JOBCOR) so that a Monitor SAVE command will not write the low segment. This is advantageous in sharable programs for two reasons. It reduces the number of files in small DEctape directories (22 files in the maximum). Also, I/O is done only on the first user's GET that initializes the high segment but not on any subsequent user's GETs for either the high or low segment.

An Example of a Reentrant Program:

low segment subprogram:

TITLE LOW - EXAMPLE OF LOW SEGMENT SUB-PROGRAM

```

        JOBVER=137
        LOC      JOBVER
        3                ;version3
        RELOC    0
        INTERNAL LOWBEG,DATA,DATA1,DATA2,TABLE,TABLE1

LOWBEG:
DATA:   BLOCK    1
DATA1:  BLOCK    1
DATA2:  BLOCK    1

TABLE:  BLOCK    10
TABLE1: BLOCK    10
LOWEND=-1                ;last location to be cleared
        END

```

high segment subprogram:

TITLE HIGH - EXAMPLE OF HIGH SEGMENT SUB-PROGRAM

```

        HISEG
        EXTERN  LOWBEG,LOWEND
        T=1
BEGIN:  SETZM    LOWBEG                ;clear data area
        MOVEI   T,LOWBEG+1
        HRLI   T,LOWBEG
        BLT    T,LOWEND
        MOVE   T,DATA1                ;compute
        ADDI   1,1,
        MOVEM T, DATA2
        .
        .
        .
        END    BEGIN                ;starting address

```

Some reentrant programs require certain locations in the low segment to contain "constant" data which does not change during execution. Since the initialization of this data happens only once after each GET instead of after each START, programmers are tempted to place these "constants" in the same subprogram that contains the definition of the variable data locations. This action requires the SAVE command to write them out and the GET command to load them in again. Therefore the "constant" data should be moved by the programs

from the high segment to the low segment at the same time that the rest of the low segment is being initialized. The exception is when the amount of code and constants in the high segment needed to initialize the low segment constants take up too much room in the high segment. In this case, it is best to have I/O in the low segment on each GET. A rule to follow in deciding between this high segment core space and the low segment GET I/O time is to put the code in the high segment if it does not put the high segment over the next 1K boundary.

A second way of writing single save file reentrant programs has been developed in which the source file can be a single file instead of two separate ones. This is more convenient although it involves conditional assembly and therefore produces two different relocatable binaries. A number of CUSPs have been written this way.

The idea is to have a conditional switch which is 1 if a reentrant assembly and \emptyset if a non-reentrant assembly. The data is placed last in the source file following a LIT pseudo-op and consists only of Block 1 and Block N statements, along with data location tags. If a reentrant program is desired, a LOC 140 is assembled which places the data area at absolute 140 in the low segment. Because of the LOC, no other relocatable program can be loaded into the low segment. The program should be debugged as a non-reentrant program with DDT since DDT is a low segment relocatable file. The LOADER switch /B is used to protect the symbols. The usual way of assembly is reentrant so, unless already defined, the conditional switch is 1.

The program must have one location in the Job Data area when it is assembled to be reentrant so that the Monitor will know to start assigning buffers at the end of the data area in the low segment instead of at location 140. This is accomplished by chang-

ing the LH of JOBSA before the CALLI 0 (RESET) or changing the contents of JOBFF after the CALLI 0, depending on how the program reinitializes itself on errors and upon completion. The program should not change these locations if it is assembled as non-reentrant. This is so that the symbol table can be protected using the LOADER /B switch, which places the symbols next to the last program loaded and sets the LH of JOBSA appropriately higher. Therefore, this code is under control of conditional assembly.

```

TITLE DEMO - DEMO ONE SOURCE REENTRANT PROGRAM -V001
SUBTTL          T. HASTINGS          25 JUN 69
JOBVER=137
  LOC 137
  EXP 001          ;version number

  INTERN JOBVER,PURE
  EXTERN JOBSA,JOBFF

IFNDEF PURE,<PURE=1>          ;assume reentrant if PURE undefined
  IFN PURE,<HISEG>          ;tell LOADER to load in high segment
                          ;if reentrant

BEG:
IFN PURE,<
  MOVSI T,DATAE          ;only need if reentrant
  HLLM T,JOBSA          ;(not needed if two files)
                          ;set first free location in low seg,
                          ;RESET sets JOBFF from LH of JOBSA
>
  CALLI 0          ;do CALL RESET
  MOVE T,JOBFF          ;assign at least enough core for data
  CALLI T,11          ;CORE UUO
  JRST ERROR
  MOVE T,[XWD DATAB,DATAB+1] ;now clear data region
  SETZM DATAB
  BLT T,DATAE-1          ;last location cleared
  .
  .
  .
  .
  LIT          ;put literals in high seg
;DATA AREA:
IFN PURE,<LOC 140>          ;start data area at 140 in low seg
                          ;if reentrant
                          ;first location cleared every startup
DATAB:
DATA:          BLOCK 1
TABLE:          BLOCK 128
  .
  .
  .
DATAE:          END      BEG          ;define free location

```

Book 4

Editing the Source Program

**DECtape Editor
(Editor)**

**Line Editor for Disk
(LINED)**

**Text Editor and Corrector
(TECO)**

SOURCE PROGRAM PREPARATION

DECTAPE EDITOR (EDITOR)

Editor creates, adds to, or deletes from sequentially numbered source files recorded in lines of ASCII characters on a DECTape. Editor edits the source file; (the input and output files are the same). Fresh source files have editing space in each physical DECTape block. If the user has more edits for a block than will fit in it, an extra block in the DECTape is used and appropriately linked to the preceding and following logical blocks of the file. Editor provides a simple method of creating or modifying Macro or FORTRAN IV source programs.

Requirements

Minimum Core: 1K
 Additional Core: Not used
 Equipment: One DECTape unit for the reel containing the file(s) to be modified

Initialization

DECTAPE EDITOR
 * Loads the DECTape Editor program.
 Editor is ready to receive a command.

Commands

Initialize a File For Processing

<u>Command</u>	<u>Function</u>
Sn tA)	Select DECTape n and zero the directory.
Sn,filename.ext tA \$	Select DECTape n, zero the directory, and create a file called filename.ext.

<u>Command</u>	<u>Function</u>
<code>Sn, filename.ext)</code>	Select DECTape n and locate filename .ext for processing.
<code>Sn, filename.ext \$</code>	Select DECTape n and add a new file called filename .ext.

NOTE

All the above commands place Editor in the Command mode; i.e., the next typein is assumed to be one of the commands given below.

Insert a Line

<code>Innnnn)</code>	Insert the following typed line at line number nnnnn of the currently open file; nnnnn can be specified as a line sequence number or as a point (.). A point refers to the last line typed. If the line number already exists in the file, the line is replaced.
<code><u>nnnnn</u> aaaa.....a)</code>	
<code><u>nnnxx</u> \$)</code>	
<code>* -</code>	

Insert Multiple Lines

<code>Innnnn, increment)</code>	Insert the following typed lines, beginning at line number nnnnn of the currently open file; nnnnn can be specified as a line sequence number or as a point (:). Each time a line is entered, nnnnn is increased by the specified increment, and the result becomes the line number for the next insertion. Type \$ after last line insertion.
<code><u>nnnnn</u> aaaa...aaa)</code>	
<code><u>nnnxx</u> bbbb...bbb)</code>	
<code><u>nnnxx</u> \$)</code>	
<code>* -</code>	

Delete a Line

<code>Dnnnnn)</code>	Delete line number nnnnn from the currently open file; nnnnn can be specified as a line sequence number or as a point (.).
<code>* -</code>	

CommandFunctionDelete a Series of Lines

Dmmmm,nnnn)

Delete lines mmmmm through nnnn from the currently open file.

*
_Print a Line

Pnnnn)

Print line number nnnn of the currently open file; nnnn can be specified as a line sequence number or as a point (.).

nnnn aaa...aaa)

*
_Print a Series of Lines

Pmmmm,nnnn)

Print lines mmmmm through nnnn of the currently open file.

mmmm aaa...aaa)

.

.

nnnn bbb...bbb)

*
_Close the Current File

E) (End of file)

Closes the currently open file. Another file can be opened on the same or a different DECTape via an Sn command, or a return can be made to Monitor to terminate Editor.

*
_

Examples

_R EDITOR)

*S1,VECTOR \$

Select DECTape 1 and create a new file on DECTape 1 called VECTOR.

*I20,20)

Begin inserting at line sequence number 20, and increment this number by 20 each time a line is inserted. Switch to Text Mode.

```

00020 DEFINE VMAG (A,B)
00040 <MOVE 0,A)
00060 FMP 0)
00080 MOVE 1,A+1)
00100 FMP 1,1)
00120 FAD 1)
00140 MOVE 1,A+2)
00160 FMP 1,1)
00180 FAD 1)
00200 JSR FSQRT)
00220 MOVEM B)
00240 $ $)

```

Editor responds with first line sequence number.
Operator types line of coding to be inserted,
followed by a carriage return.

Typing \$ terminates insertions and returns Editor
to command mode.

*I20)

Change line number 00020.

```
00020 DEFINE VMAG (A,B,C)
```

ILR)

ILR indicates that the indexing increment has
resulted in the next line number being equal to
the line number of an already existing line (00040).
Note that the indexing increment remains as 20
until explicitly changed.

*I90)

Insert a line between lines 00080 and 00100.

```
00090 MOVE 1,C)
```

ILS)

ILS indicates that the indexing increment has
resulted in an existing line (00100) being skipped,
since the next line addressed would be 00110.

*D180)

Delete line 00180.

*P20,220)

Print lines 00020 through 00220.

```

00020 DEFINE VMAG (A,B,C)
00040 <MOVE 0,A)
00060 FMP 0)
00080 MOVE 1,A+1)
00090 MOVE 1,C)
00100 FMP 1,1)
00120 FAD 1)
00140 MOVE 1,A+2)
00160 FMP 1,1)
00200 JSR FSQRT)
00220 MOVEM B)

```

*E)

Close the currently open file.

*+C)

Return to the Monitor.

.

Diagnostic Messages

Editor Diagnostic Messages

Message	Meaning
?DDE*	Device Data Error due to a write error or WRITE LOCK switch. Editor must be restarted.
?DEC*	DECtape directory is full.
?FAU*	Filename Already Used. A filename assigned to a new file already exists on the DECtape.
?ILC*	Illegal Command.
ILR *ILS*	Insert Line Replacement, Insert Line Skip. The line sequence increment specified for the insert function will cause the next existing line to be either replaced (R) or skipped (S). This is a warning message only and does not necessarily indicate an error.
?NCF*	Not a Current File.
?NFO*	No File Open. A command requiring an active file has been given but no file is currently open.
?NLN*	Nonexistent Line Number. A print (P) or delete (D) command refers to a nonexistent line.
?UNA*	Unit Not Available. The DECtape specified in an Sn command is assigned to another job.

LINE EDITOR FOR DISK (LINED)

LINED is similar to DECTape Editor, but operates on disk files instead of DECTape files. LINED is called in by typing

```
._R LINED
```

```
*
```

LINED responds with an asterisk to indicate it is ready to accept a command string.

Commands

LINED commands are very similar to those of Editor; consequently, only a brief summary is given here.

<u>Command</u>	<u>Function</u>
S filename .ext)	Select an existing file for editing.
S filename .ext \$	Select (create) a new file for editing.
Innnnn)	Insert a line at line number nnnnn.
Innnnn ,increment)	Insert multiple lines, starting at line number nnnnn and incrementing the line number each time.
Dnnnnn)	Delete the line at line number nnnnn.
Dmmmmm ,nnnnn)	Delete lines mmmmm through nnnnn.
Pnnnnn)	Print line number nnnnn on the Teletype.
Pmmmm ,nnnnn)	Print lines mmmmm through nnnnn on the Teletype.
E)	End (close) the current file.
\$	If in Insertion Mode, ignore current text line and return to LINED command level. If in Command Mode, print the next line.

NOTE

Files are written with standard protection (055). All blocks are assumed to have integral number of lines. Use the /A switch (line blocking) with PIP to put each file on disk.

Diagnostic Messages

The diagnostic messages for LINED are similar to those for Editor. The diagnostic message ?DEC* is not included, and the following message is added:

<u>Message</u>	<u>Meaning</u>
<u>?CCL*</u>	CCL error. Error occurred while referencing CCL command file.

Monitor Commands

To call in LINED and open a new file for creation:

<u>Monitor Command</u>	<u>Equivalent CUSP Commands</u>
<u>.CREATE filename .ext</u>)	<u>.R LINED</u>)
	<u>*S filename .ext</u> \$

To call in LINED and open an existing file for editing:

<u>Monitor Command</u>	<u>Equivalent CUSP Command</u>
<u>.EDIT filename .ext</u>)	<u>.R LINED</u>)
	<u>*S filename .ext</u>

TEXT EDITOR AND CORRECTOR (TECO)*

TECO edits files recorded in ASCII characters on any standard device. It can perform simple editing functions as well as highly sophisticated search, match, and substitute operations, and operate upon arbitrary length character strings under control of commands which are themselves character strings (and contains the mechanisms necessary to exploit this recursiveness).

Requirements

Minimum Core:	4K
Additional Core:	Takes advantage of any additional core available. Each 1K additional core augments the basic 6,200+ - character buffer by 5K additional characters.
Equipment	One input device and one output device.

NOTE

TECO automatically requests more core to expand its buffer under any of the following conditions:

*PDP-10 TECO is based on PDP-1 TECO, developed at MIT by Daniel Murphy, and PDP-6 TECO, developed at MIT's project MAC by Stewart Nelson and Richard Greenblatt.

- a. An insert by way of the "I" command or "X" (Q Register) will overflow the present memory boundaries.
- b. The command acceptance routine needs more core.
- c. The total number of characters in the Data Buffer falls below 3000, and an input command from a peripheral device (other than the user console) is executed. Thus, JECO maintains a Data Buffer of at least 3000 characters.

If TECO is successful at obtaining more core, the following message will be typed:

```
*10000 <IJS >$$
[5K CORE]           nk Core, where n is
                    new core size of job
*
```

If TECO is unsuccessful at obtaining the core request, the following message is typed:

```
STORAGE CAPACITY EXCEEDED
C
*!
```

Initialization

```
_R TECO )           Loads the Text Editor and Corrector program.
*
-            TECO is ready to accept a command.
```

Basic Commands

NOTE

When typing command strings to TECO, the following points should be noted.

\$ One \$ is used to terminate the text within a command string, where applicable; two successive \$s terminate the entire command string sequence and generate a RETURN, LINE-FEED.

RUBOUT

The RUBOUT key can be used to erase the preceding typed-in character(s) of a command string. Each character erased is echoed back on the teletype (e.g., ABD RUBOUT DC...). Successive RUBOUTs can be used to erase more than one character.

N.B. To erase a carriage return (which generates RETURN, LINE-FEED), two RUBOUT's are required, one RUBOUT to erase the LINE-FEED and one to erase the RETURN.

Two successive tGs (BELL s) can be used to wipe out the entire command string currently being typed.

TECO commands in the form tx (where x is any character) can be entered by either holding down the CTRL key while striking the x key or typing up-arrow (shift N) followed by the x character. These alternatives are not true where tx is a character within a text string (such as in a Search argument); in this case, the CTRL key must be used.

A carriage return, line feed, () is ignored in a TECO command string as long as it does not appear within a particular command, such as Insert. Examples of this are given on the following pages.

Select The Input Device

Command	Function
ERdev:filename.ext [proj,prog] \$	Edit Read. Selects the input device and file (if specified). dev:* DTAn: (DECtape) PTR: (paper tape reader) DSK: (disk) MTAn: (magnetic tape) CDR: (card reader) filename.ext (DSK: or DTAn: only) [proj,prog] (DSK: only)
NOTE	
Device TTY: cannot be used here. See I (Insert) command.	
Specified only if file is located in other than user's area.	
EBdev:filename.ext \$	Edit Backup. Selects an input and file to be edited (the input device, which will also be the output device for the edited file, must be the disk*). EB is intended to be used to keep a backup of a file during a debugging session, without the user having to invent a new name for each version of the file.

*If dev: is not specified, DSK: is assumed.

CommandFunction

For example, the command string sequence

```
EBPROG1.MAC $
.
.
editing
.
.
EF $$
```

results in:

- a. Reading from file PROG1.MAC on disk
- b. Creating a new output file, which is initially given a temporary name of TECOnn.TMP, where nn is the user's job number in octal; incorporating the job number in the filename solves the problem of identifying temporary files belonging to multiple 100,100 users
- c. Performing a number of RENAMEs following the EF command so that the input file becomes PROG1.BAK, any previous PROG1.BAK file is deleted, and the new output file becomes PROG1.MAC.

An ER command can be given following an EB command and before the EF command, but an intervening EW command is illegal and results in the error message ?22 (see Table 2-5). Even though an ER command may be given, the name of the final output file is still taken from the EB command.

Select The Output Device

EWdev:filename.ext [proj,prog] \$	Edit Write. Selects the output device and file (if specified).
EZdev:filename.ext [proj,prog] \$	Edit Zero. Selects the output device and file (if specified), and rewinds the tape (if magnetic tape) or zeros the directory (if DECTape).
dev:*	DTAn: (DECTape) DSK: (disk) MTAn: (magnetic tape) PTP: (paper tape punch) LPT: (line printer)

*If dev: is not specified, DSK: is assumed.

filename.ext (DSK: or DTAn: only)
 [proj,prog] (DSK: only)

Specified only if file is located
 in other than user's area.

Terminate Output; Close File

EF	End File. Terminate output on the current output file and close the file without selecting a new output file.
EX	Exit. The EX command finishes the current edit operation by writing out all remaining pages of a file, performing an EF (End of File) command, and then exiting to the Monitor. Thus, EX is equivalent to the command string 1000000PEF (BELL)
EG	Exit and Go. The EG command executes an EX command followed by the last Monitor command (COMPILE, LOAD, EXECUTE, or DEBUG) typed by the user. (See Chapter 9.)

Magnetic Tape Positioning

EM	Edit Magtape. Rewind the currently selected input magnetic tape.
nEM	Depending upon the value of n, perform one of the following operations on the currently selected input magnetic tape.

<u>n</u>	<u>Operation</u>
1	Rewind tape to load point.
3	Write end of file.
6	Skip one record.
7	Backspace one record.
8	Skip to logical end of tape.
9	Rewind and unload tape.
11	Erase 3 in. of tape.
14	Advance tape one file.
15	Backspace tape one file.

NOTE

Throughout TECO, all numbers in command strings are interpreted as decimal.

Input Commands

<u>Command</u>	<u>Function</u>
Y	<p>Yank. Read from current input device into buffer until</p> <ol style="list-style-type: none"> a. A FORM character is read (i.e., a page has been input), or b. The buffer is more than $2/3$ full and one of the following is encountered <ol style="list-style-type: none"> (1) Line Feed (2) Form Feed c. or a point 128 characters from the end of the buffer is reached.
	<p style="text-align: center;">NOTE</p> <p>The FORM character, if read, does not enter the buffer. Any data previously residing in the buffer is destroyed. The pointer is positioned immediately before the first character in the buffer. Representative buffer size for 5K TECO:</p> <p>Total buffer capacity = approx. 11,200 characters</p> <p>$2/3$ buffer capacity = approx. 7,460 characters</p> <p>1 line-printer page = $7,200^+$ characters (120 char./line)</p> <p>(60 lines) $7,800^+$ characters (130 char./line)</p>
A	<p>Append. Read from the current input device and append the incoming data to information already residing in the buffer. Terminate reading on the same conditions as in Y.</p>

NOTE

No previous data is destroyed. The pointer is not moved.

Output Commands

PW	<p>Punch, Wait. Output the entire buffer to the selected output device, with a FORM character appended as the last character. Do not alter the contents of the buffer or move the pointer.</p>
----	--

<u>Command</u>	<u>Function</u>
P	Equivalent to a PW command, followed by a Y command (i.e., output the current contents of the buffer followed by a FORM character, and then read in more data from the input device).
np	Perform the P command n times.
m,nP	Output the m+1 through the nth character from the buffer to the current output file. Do not append a FORM character at the end. Do not alter the contents of the buffer or move the pointer.

Editing Commands

Move The Pointer

nj	Jump. Move pointer to the right of the nth buffer character and give the pointer symbol (.) the value of n. If n is omitted, set pointer in front of the first buffer character (same as 0J).
nC	Continue. Set the pointer to the right of the nth character beyond the pointer's present position (equal to .+nJ). If n is omitted, 1 is assumed.
nR	Reverse. Set the pointer to the left of the nth character prior to the pointer's present position (equal to .-n). If n is omitted, 1 is assumed.
nL	Lines. +n - Move the pointer to the right, stopping after it has passed over n LINE-FEED characters. -n - Move the pointer to the left, stopping after it has passed over n+1 LINE-FEED characters, then move to the right of the last LINE-FEED character passed over. .If n is omitted, assume 1L.

Delete Text

nD	Delete. Delete n characters. +n - Delete n characters just to the right of the pointer. -n - Delete n characters just to the left of the pointer. If n is omitted, 1 is assumed.
nK	

CommandFunction

nK	<p>Kill. +n - Move the pointer to the right, stopping after it has passed over n LINE-FEED characters. Delete all characters the pointer passes over.</p> <p>-n - Move the pointer to the left, stopping after it has passed over n+1 LINE-FEED characters, then move it to the right of the last LINE-FEED character passed over. Delete all characters between this point and the pointer's previous position.</p> <p>If n is omitted, 1 is assumed.</p>
m,nK	Delete the m+1 through the nth characters of the buffer. Set the pointer where the deletion occurred.

Insert Text

Itext... \$	Insert. Insert the text following the "I" up to, but not including, the \$ character, beginning at the current pointer position. Move the pointer to the right of the inserted material.
nI	Insert at the pointer location a character with an ASCII code of n (n must be a decimal value). Move the pointer to the right of the inserted character.
n\	Insert at the current pointer location the ASCII text representation of the decimal value of the expression n. Move the pointer to the right of the inserted text.
-> text... \$	Insert at the current pointer location a (->) character and the following text up to, but not including, the \$ character. Move the pointer to the right of the inserted text.
@I/text/	Insert at the current pointer location the text which follows. The text is delimited by a character, /, which can be any character not appearing in the text.

Type Text

NOTE

T commands do not move the pointer.

<u>Command</u>	<u>Function</u>
nT	Type. Type out the string of characters beginning at the current pointer position and terminating after the nth LINE-FEED character is encountered. +n - Typeout n lines to the right of the current pointer position. -n - Typeout n lines to the left of the current pointer position. If n is omitted, the value is assumed to be 1.
m,nT	Type out the m + 1 through the nth characters of the buffer.

Stand-Alone Examples (Elementary)

Open an Input File

ERDTA5:SOURCE.MAC	\$	Open the input file called SOURCE.MAC located on DTA5.
ERDSK:SRCE.MAC[12,24]	\$	Open the input file called SRCE.MAC located in area 12,24 on the disk.
ERPTR:	\$	Open an input file on the paper tape reader.

Open an Output File

EWDTA3:EDITED.MAC	\$	Open an output file on DTA3 and call it EDITED.MAC.
EZDTA1:DEBUG.MAC	\$	Zero the directory on DTA1, open an output file on it, and call the file DEBUG.MAC.

Read a Page

Y	Read a page into the buffer from the current input file, destroying the previous contents of the buffer.
A	Read a page into the buffer, appending the data to the end of the information currently in the buffer.

Output Data

<u>Command</u>	<u>Function</u>
PW	Output the entire buffer, followed by a FORM character.
6P	Execute the WRITE and READ cycle six times.
12,50P	Write out the 13th through the 50th characters of the buffer.

Pointer Positioning

Y18J	Read in a page of information and position the pointer after the 18th character of the buffer.
5R	Then, move the pointer left to between characters 13 and 14.

Delete Text

J19C3D or 19,22K	Move the pointer to the right of the 19th character in the buffer and then delete the next three characters to the right (characters 20, 21, and 22). Delete the 19+1 (20th) through the 22nd characters of the buffer.
------------------------	--

Insert Text

J2LITAG: MOVE 1, AMT \$	Move the pointer to a position following the second line of the buffer; insert the text TAG: MOVE 1, AMT between the second and third lines of the buffer.
69\ →	Insert the digits 69 in ASCII at the current pointer position (same as I69 or \$ or 54I57I).

NOTE

Unless a \key is present, \ is typed with a SHIFT L.

→	Insert a tab followed by the text ERROR IN JOB at the current pointer position.
---	---

CommandFunction

@I#ERDSK:PROG1#)

Insert the text ERDSK:PROG \$ at the current pointer position.

NOTE

The use of delimiters is the only method for inserting a \$ in the text.

Typing Text

3T

Type out the first three lines of the buffer.

25,100T

Type out the 25+1 (26th) through the 100th character of the buffer.

Examples (Basic)

.R TECO)

+ERDTA1:SCFILE.MAC1\$)

Open the file called SCFILE.MAC on DTA1 for input.

*EWDTA2:EDFILE.MAC1\$)

Open an output file on DTA2 and call it EDFILE.MAC.

+Y0,20T11)

Read a buffer of information from the input file and type the first 20 characters of the buffer.

aaaaa.....aaaaa)

+3LT11)

Move the pointer to the right, stopping when three LINE-FEED characters have been encountered; type the text of the fourth line in the buffer.

bbbbbb.....bbbbbb)

+1THIS IS A SAMPLE INSERT)

Insert the text THIS IS A SAMPLE INSERT between the third and fourth lines of the buffer, and position the pointer after the inserted material.

11)

+10PT11)

Write out the current buffer to the output device; read in and write out the next nine pages of data; read in the 11th page of data and position the pointer at the beginning of the buffer; type out the first line of the buffer.

cccccc.....cccccc)

+K11)

Delete this line from the file; position the pointer at the beginning of the (now) first line in the buffer.

<u>Command</u>	<u>Function</u>
*EX\$\$)	Writes out all remaining pages of the file, performs an EF (End of File) command, and exits to the Monitor.
[EXIT) ↑C)	Kill the job, deassign all devices, release core.

Advanced Commands

Search Commands

Summary

S text \$ (Search) - Searches for text in current buffer only.

N text \$ (Nonstop Search) - Searches for text through successive buffers by repeatedly writing out current buffer and reading in next buffer (similar to P command, but a form character is not inserted after outputting each buffer).

↑text \$ - Searches for text through successive buffers by repeatedly reading in new bufferful of information (Y command).

NOTES

All searches begin at the current location of the pointer.

Each search command can be preceded by the modifier characters (: and/or @).

: causes the search command to have a numeric value at completion;

0 if the search has failed (the requested text was not found) or -1 if the search was successful (the requested text was found).

@ indicates that the text to be matched is delimited by some character (same as in the @I command).

A numeric argument can appear following the modifiers (if any) but must precede the command. If the numeric argument is n, TECO searches for the nth occurrence of the text. If n is not used, the value of n is assumed to be 1.

If search is successful, the pointer is positioned to the right of the matched text. If the search fails, the pointer is positioned at the beginning of the buffer.

Use of special characters within text:

- !S - Match any separator character (any character not a letter, number, period, dollar sign, or percent symbol).
- !X - Match any (arbitrary) character. Used when the contents of some position within the text is unimportant.
- !Nx - Match any character except x.
- !Q - Takes the next character literally, even if it is one of four special characters. For example, S !Q !X \$ - Find the character !X.

See note under TECO, Basic Commands.

Search Commands Summary

Command	Action at End of Buffer	Action at End of File	Values		Typeout ? if Failure
			Success	Fail	
S	Failure	N/A	N/A	N/A	Yes
:S	Failure	N/A	-1	0	No
N	Performs a P command (but a FORM character is not inserted) and resumes search	Failure	N/A	N/A	Yes
:N	Performs a P command (but a FORM character is not inserted) and resumes search	Failure	-1	0	No
+	Performs a Y command (read only) and resumes search	Failure	N/A	N/A	Yes
:+	Performs a Y command (read only) and resumes search.	Failure	-1	0	No

Q-Register Commands

Q registers are provided for storing quantities, command strings, or buffer contents for later use. Thirty-six Q registers, labeled 0 through 9 and A through Z, are available.

<u>Command</u>	<u>Function</u>
nUi	Use. Places the numeric value n in Q-register i.
Qi	Q-register. Represents the current value in Q-register i
%i	Adds 1 to the value in Q-register i and represents the new value.
m,nXi	Xfer. Copies characters m+1 through the nth character of the buffer into Q-register i. Does not alter buffer contents or pointer.
nXi	Copies the buffer characters between the current pointer position and the nth LINE-FEED character in Q-register i
Gi	Get. Inserts the text contained in Q-register i into the buffer beginning at the current pointer location. Set the pointer to the right of the insertion.
[i	Pushes the contents of Q-register i onto the Q-register pushdown list.
]i	Pops the top entry of the Q-register pushdown list into Q-register i. The Q-register pushdown list is cleared each time two successive \$s are typed.

Macro, Iteration, and Conditional Commands

Mi	Macro. Perform the text in Q-register i as a series of commands.
<>	Iteration brackets. When > is encountered, command interpretation is sent back to <.
n <>	Perform the commands within the iteration brackets n times.
;	If not in an iteration, an error results. If most recent search failed, send command interpretation to just beyond the matching > on the right; otherwise, no effect.
n;	If not in an iteration, an error results. If the value of n is 0 or greater, send command interpretation just past the matching > to the right; otherwise, no effect.
.' tag '.	Tag definition. Tag is the name of the location in which it appears in a command string.
O tag \$	Go to the named tag, which must appear in the current macro or command string.
n"G	If n is Greater than or equal to 0, perform commands up to next '. Otherwise, skip to next '.

<u>Command</u>	<u>Function</u>
n"L	If n is Less than or equal to 0, perform commands up to next '. Otherwise, skip to next '.
n"N	If n is Not equal to 0, perform commands up to next '. Otherwise, skip to next '.
n"E	If n is Equal to 0, perform commands up to next '. Otherwise, skip to next '.
n"C	If n is a symbol Constituent (a letter, number, period, dollar sign, or percent symbol), perform commands up to next '. Otherwise, skip to next '.

NOTE

The double quotation mark (") and the single quotation mark (') symbols are matched in the same way as the left parenthesis symbol, (, and right parenthesis symbol,).

Numeric Values and Arguments in Command Strings

Many command string formats permit arguments with numeric values. The following characters may appear in a command string to develop these values in any instance where a numeric value is permissible.

0 through 9	Represent their corresponding numeric values.
B	Beginning. Equivalent to 0.
Z	Equivalent to the number of characters in the buffer.
.	Equivalent to the number of characters to the left of the current pointer position (or in other words, equal to the current pointer position).
Qi	Q-register. Equivalent to most recent numeric value placed in Q-register i.
nA	ASCII. Equivalent to ASCII value of character to right of pointer; n is used to differentiate this argument from an Append command (A) and has no other significance.
tH	Equivalent to value of elapsed time in 60ths of a second since midnight.
tF	Equivalent to the value of the console data switches.

<u>Command</u>	<u>Function</u>
tE	Has the value of the form feed switch. If, during the last Y or A command execution, data transmission was terminated by a form feed character, tE has a value of -1, otherwise, the value is 0.
tf	(On Teletype Models 33 and 35, hold down both the CTRL and SHIFT keys and type N.) Equivalent to the ASCII value of the next character in the command string; this character is not interpreted as a command.
tT	Typed Character. Stops command execution until user types a character on the Teletype; tT then becomes equivalent to the ASCII value of the character typed.
\	Equivalent to the value represented by the digits (or minus sign) immediately following the current pointer position. The value is terminated by the first nonnumeric character encountered. The pointer is positioned immediately following the value.
m+n m-n	Add Subtract } Take one or two arguments. A space is equal to +.
m*n m/n	Multiply Divide (truncates) } Take one or two arguments.
m&n	Logical AND; bitwise AND of binary representations m and n.
m#n	Logical IOR; bitwise inclusive OR of binary representations m and n.
()	Operators +, -, *, /, #, and \$ are normally performed left to right. This sequence can be overruled by use of parentheses.

NOTE

TECO does not assume that multiplication and division are always performed before addition and subtraction. Thus, to obtain the equivalent of $a + (b * c)$, one must use the parentheses; otherwise, $(a + b) * c$ is assumed.

n=	Causes the value of n to be typed out.
H	Abbreviation for B, Z. (0 through the last location of the buffer; in other words, the whole buffer).

NOTE

If a command takes two numeric arguments, a comma is used to separate them.

TECO Termination Commands

Command	Function
tC	Returns control to the Monitor without waiting for any I/O operations to finish.
tG (BELL)	Returns control to the Monitor after completing all current output requests.

Stand-Alone Examples (Advanced)

Search

J3SMOVE \$ IM \$	Within the current buffer, search for the third occurrence (3S) of the text MOVE, position the pointer immediately after it, and insert an M at that point.
------------------	---

Search for a Special Character

S†NA \$	Search for any character except A within the current buffer.
S†S \$	Search for any separator character within the current buffer.

Q-Registers, Macros, Iterations, and Conditionals

J0UN <S↓ \$;%N>QN =	Count the number of LINE-FEED characters in the buffer as follows: <ol style="list-style-type: none"> 1. Position the pointer at the beginning of the buffer (J), 2. Place 0 in Q-register N(0UN), 3. Perform a search for a LINE-FEED character (S LINE-FEED \$); if one is found, add 1 to Q-register N (;%N). Go back (< >) and repeat this cycle until the end of the buffer is reached and the test fails (;); at this point type out the contents of Q-register N(QN=).
J<SJUMPA \$; -4DIRST \$ >	Whenever JUMPA appears in the current buffer replace it with JRST.

CommandFunction

1. Position the pointer at the beginning of the buffer (J).
2. Search for JUMPA; when found, backspace the pointer four positions and delete the four characters passed over (;-4D).
3. Replace these four characters with the characters RST (IRST).
4. Repeat this routine (< >) until the test fails (end of the buffer has been reached) and exit (;) to >.

Placing a Command in a Q-Register for Later Execution

```
@I# JOUN <S!
                               $ )
;%†>QN = #HXP
```

To Execute the Command:

```
ERDTA3:FN.EX $ YMP
```

1. Insert the text 'JOUN <S! \$; %N >QN=' into the buffer (@I##)
2. Copy the contents of the buffer into Q-register P (HXP).

1. Read in a page of a file to search. (ERDTA3:FN.EX \$ Y)
2. Execute the command stored in Q-register P (MP).

Reading in Text to be Inserted in Several Places in a File and Storing it in a Q-Register

```
ERPTR: $ YHXP)
```

```
ERDTA4: TXTEDT $ )
```

```
EWDSK: TXTEDU $ )
YNCALC: $ GP)
```

```
NTOT: $ GP
```

1. Assume that the text to be inserted is on paper tape. Open an input file on the paper tape reader (ERPTR:); read the text into the buffer (Y); copy the contents of the buffer into Q-register P (HXP).
2. Open the input file to be edited and the output file to contain the edited version.
3. Read a page from the input file and initiate a search for the text CALC: . When found, insert the text stored in Q-register P at that point (GP).
4. Search for the text TOT: and, when found, insert the text stored in Q-register P after it.

Examples (Advanced)

Command	Function
<pre> .R TECO) *ERMIA1:\$EM14EM\$\$) </pre>	<p>Select MTA1 for input; rewind the tape (EM) and advance the tape one file (14EM).</p>
<pre> EZDTA1:REVFIL\$\$) </pre>	<p>Select DTA1 for output; zero the directory; open a file and call it REVFIL.</p>
<pre> *YNTAXRTSØLT) IX1\$\$) </pre>	<p>Read in the first page from the input file; search for the text TAXRT; if it cannot be found, write the buffer out, read in the next page, search again, etc.; continue this cycle until either TAXRT is found or end of file is reached. If TAXRT is found, position the pointer at the beginning of the line containing it, type the line, and place the line in Q-register 1.</p>
<pre> aaaa...TAXRT aaaa.....aaaaa </pre>	
<pre> *JNTXRTE\$ØLT: ØLT) </pre>	<p>Search the buffer for the text TXRTE; if not found, write out the buffer, read the next page, search again; continue this cycle until either TXRTE is found or end of file is reached. If TXRTE is found, position the pointer at the beginning of the line containing it, type the line, and insert the contents of Q-register 1 immediately before that line.</p>
<pre> G1\$\$) bbb...TXRTE bbb.....bbbbbb </pre>	
<pre> *NTXTEND:\$) J<SA\$;1A-47"G1A-58"L-DIB\$">) PWEF\$\$) </pre>	<p>Read pages from the input file and write them on the output file until end of file (marked by the text TXTEND;) is found. At that point, move the pointer to the beginning of the buffer (J), and search for all As in the buffer (SA); if the character following the A is a digit, 0 through 9 (ASCII codes 48₁₀ through 57₁₀), change the A to a B (IB); continue searching and modifying until end of buffer is reached; write out the last page and write end of file on the output device.</p>
<pre> *↑G\$\$) EXIT) ↑C) </pre>	<p>Return control to the Monitor after all output requests have been completed.</p>

Diagnostic Messages

TECO Diagnostic Messages

Message	Meaning
?n	<p>n is a decimal number associated with one of the list of error messages given in Table 2-5.</p> <p>TECO ignores the remainder of the command string and returns to the idle state. At this point, the user can type back ?, causing TECO to type out the command string terminated by the bad command.</p>

Error List for ?n Messages

ⁿ 10	Meaning
1	TECO attempted to read commands beyond the terminating \$\$. This error is probably due to an unterminated @I or @S command, or to an unsatisfied O command.
2	Same as 1.
3	An attempt was made to supply more than two arguments to a command, either by the use of two commas or by "H,".
4	Too many right parentheses.
5	= command with no argument.
6	U command with no argument.
7	Q,U,X, or G command specifies an illegal Q-register (i.e., other than A through Z or 0 through 9).
8	In an X command, the second argument is not greater than the first.
9	In a G command, the Q-register does not contain text.
10	In a G command, the data in the Q-register is not in correct form (this is an internal error).
11	In an Ec command (e.g., ER, EW, EF, etc.), c is illegal.
12	File not found on LOOKUP.
13	Blank filename specified for directory device.
14	Project-programmer number specified does not have UFD.

Error List for ?n Messages

n ₁₀	Meaning
15	Protection failure on disk.
16	File cannot be accessed because it is currently being written.
17	LOOKUP or ENTER returned error type 6 (not defined).
18	LOOKUP or ENTER returned error type 7 (no device).
19	Directory full on ENTER.
20	Requested I/O device not available.
21	Not assigned.
22	EW command between an EB command and its EF.
23	EM command given, but no input file open.
24	nEM command, where n is not in the range 1 to 16.
25	Internal error: EF after EB, but no input file is open.
26	Illegal character in filename.
27	Illegal character in project-programmer number.
28	Attempt to read an input page when no file has been opened for input.
29	I/O error on input device.
30	Attempt to output a page when no file has been opened for output.
31	Two arguments were supplied for an L command.
32	Attempt to move pointer beyond page.
33	A 2-argument command has its second argument less than the first argument.
34	Attempt to search for too long a character string.
35	Search command did not find the required string.
36	In an M command, the Q-register does not contain text.
37	In an M command, the data in the Q-register is not in correct form (this is an internal error).
38	Unmatched right angle bracket.

Error List for ?n Messages

n ¹⁰	Meaning
39	; encountered when not in iteration.
40	" command with no numeric argument, or "x where x is not G, L, E, N, or C.
41	This is the number typed out at the end of the ? command's dump of the command string in error. Refer to the number of the previous error.
42	A character has been encountered as an undefined command.
43	A tD command, when DDT has not been loaded with TECO.
44	Not enough core available from the Monitor.
45	A RENAME attempted with either a blank name or one already in use. Presumably due to a fault in the EB command.

Debugging Aids - As an aid in debugging macros and iterations, TECO can be set in the trace mode by typing ? as any character other than the first in a command string. When in Trace Mode, TECO types out each command as it is interpreted, interspersed with requested output. Typing a second ? in the same manner takes TECO out of Trace Mode; the ? can be typed each time it is desired to change the current mode.

The user can also type comments on his teletype sheet as he executes TECO by typing:

t Atext tA

This causes all text entered to be printed on the teletype (with the exception of terminating tA character).

NOTE

Since the terminator tA is not a command, it must be typed by holding the CTRL key down while typing A. It cannot be entered as "up arrow, A."

If DDT has been loaded along with TECO by the Linking Loader, control can be transferred to DDT by using the command tD.

Monitor Commands

To call in TECO and open a new file for creation:

Monitor Commands

```

_MAKE filename .ext
*(text input commands) $$
_EX $$

```

Equivalent CUSP Commands

```

_R TECO
*_EWDSK:filename .ext $$
*(text input commands) $$
_EX $$

```

To call in TECO and open an already existing file for creation:

Monitor Commands

```

_TECO filename .ext
*(editing) $$
_EX $$

```

Equivalent CUSP Commands

```

_R TECO
*_EBDSK:filename .ext $ Y $$
*(editing) $$
_EX $$

```


Book 5

Executing the Program On-Line

Linking Loader
(LOADER)

DDT-10

PROGRAM LOADING AND LIBRARY FACILITIES

LINKING LOADER (LOADER) (Version #043 and later)

The Linking Loader loads and links relocatable binary (.REL) programs generated by Macro-10 or FORTRAN IV preparatory to execution and generates a symbol table in core for execution under the Dynamic Debugging Technique program. It also provides automatic loading and relocation of Macro- and FORTRAN-generated binary programs, produces an optional storage map, and performs loading and library searching regardless of the input medium. Storage used by the Linking Loader is recoverable after loading.

Requirements

Minimum Core:	3K
Additional Core:	Automatically requests additional core from the Monitor as required
Equipment:	User teletype for control; one or more input devices for binary programs to be loaded; output device for loader map (optional); one system device containing library files (optional).

NOTE

The LOADER as described herein loads and links programs assembled by the Macro Assembler, or compiled by the FORTRAN Compilers. For those users who do not wish to load FORTRAN programs (which require a substantial portion of code within LOADER), a smaller version of the LOADER, called 1KLOAD (although it is actually larger than 1K), is available. 1KLOAD may be generated from the same symbolic file as LOADER by setting the parameter K to some nonzero number (e.g., K=1).

Initialization

_R LOADER core)

Loads the Linking Loader into core. The amount of core allocated is equal to 2K plus the core required by binary programs; core is optional.

*
—

Indicates that the program is ready to receive a command.

Commands

General Command Format

list-dev:filename.ext ← source-dev 1:filename.ext,dev2:...source-n \$

list-dev:

The device on which any storage maps or undefined globals are to be written.

LPT: (line printer)
TTY: (Teletype)
DTAn: (DECTape)
DSK: (disk)
MTAn: (magnetic tape)

If the Teletype is to be assumed as the output device, omit

list-dev:filename.ext ←

source-dev:

The device(s) from which the binary relocatable programs are to be loaded.

DSK: (disk)
DTAn: (DECTape)
MTAn: (magnetic tape)
PTR: (paper tape reader)

If more than one file is to be loaded from a magnetic tape, card reader, or paper tape reader, dev: is followed by a comma (or the device name or : can be repeated) for each file after the first.

filename.ext (DSK: and DTAn: only)

The filename.ext of each relocatable binary file to be loaded. If .ext is omitted, it is assumed to be .REL. If a search for filename.REL is unsuccessful, a second search for the same filename with the null extension is performed.

The filename.ext of the output listing file. If .ext is omitted, .MAP is used.

If the filename .ext of the output map file is omitted, MAPMAP.MAP is used. If only the extension is omitted, the extension MAP is used.

The storage map device is separated from the source device(s) by the left arrow symbol.

NOTES

Each time RETURN (↵) is typed, loading is performed for all files listed on that line.

Each time \$ is typed, all remaining loading, library searches, and output operations are completed, and an exit is made to the monitor.

The source device, once stated, continues as the source device until a new source device or destination device is specified, or until \$ is typed.

Files are loaded in the order they appear in the command string. The file requiring the largest COMMON area must be specified first in any loading operation.

When loading is terminated (by \$ or switches /C, /G, or /R), the following steps are executed.

- a. A FORTRAN library search is performed if any undefined globals remain (unless prevented by the /P switch).
- b. If undefined globals still remain, they are listed on the teletype or other specified listing device.
- c. The number of multiply defined globals (if any) and the number of undefined globals (if any) are printed on both the teletype and on the specified listing device (if given).
- d. A Chain file, if requested, is written.
- e. The loaded program is relocated down to the actual locations into which it is to be loaded.
- f. The message

```

LOADER x + yk core      x = low segment core;
                        y = high segment core;
                        if nonre-entrant program
                        y = 2K; if re-entrant,
                        y = program high segment
                        or Loader high segment,
                        whichever is greater

```

is printed on the Teletype.

When an automatic library search is requested by /F, /G, or \$, the following files will be searched in order:

- a. JOBDAT
- b. FORTRAN Library (LIB40 or LIB4)
- c. JOBDAT

Since JOBDAT is searched after the FORTRAN Library, it is not necessary to include it as a portion of the FORTRAN Library. It is also searched prior to the FORTRAN Library so that users who do not require FORTRAN Library subroutines do not spend the time searching the Library. (The FORTRAN Library can be named LIB40 as on the PDP-10 or LIB4 as on the PDP-6; an attempt to find LIB40 is made first; if not found, an attempt to find LIB4 follows.)

Save and Execute Commands

After loading is completed, to write the loaded program onto an output device so that it can be executed at some future date without rerunning Linking Loader:

LOADER)	Loading is completed.
EXIT)	Automatic exit to the Monitor.
↑ C)	
._SAVE dev:filename.ext core)	Write out the user's area of core onto the specified output device and, if the device is DTAn: or DSK: assign it the specified filename.ext. If .ext is omitted, .SAV is assumed.
	The value for core may be given when the user wishes to run the program in more core than it will be saved in; this might be done to gain more space for dynamic allocation of buffers.
JOB SAVED)	Save operation completed. Core is unchanged and still contains loaded program. Automatic return is made to the Monitor.
↑ C)	
._START)	Start execution of loaded program. Return is made to user's level.
EXIT)	User's program execution is completed. Automatic return is made to the Monitor.
↑ C)	
:	

Examples

._R LOADER)	Run the Linking Loader.
*DSK:MARK1,MARK3,DTA3: SUBRTE)	Load and link the .REL files MARK1 and MARK3 from the disk, .REL files SUBRTE and CALC from DTA3, and one .REL file from the paper tape reader.
*CALC,PTR: \$)	
[LOADER 6+2K CORE)	Link-loading is completed; and automatic return is made to the Monitor.
EXIT)	
↑ C)	

SAVE DSK MARKET)

Write out the user's program as an executable program on the disk and call the file MARKET.DMP. Core assigned to the user remains unchanged.

NOTE

Saving a job is optional.

[JOB SAVED)
rC)

Save process is completed; an automatic return is made to the Monitor.

START)

Begin execution of job.

[EXIT)
rC)
.

Program execution is completed; automatic return is made to the Monitor.

Switches

Switches are used to:

- Specify the types of symbols to be loaded or listed
- Set the Library Search Mode
- Load the Dynamic Debugging Technique (DDT) program
- Clear and restart Linking Loader.

All switches are either preceded by a slash (/) or enclosed in parentheses.

Linking Loader Switch Options

Switch	Meaning	Complement Switch
A	List all global symbols in storage map regardless of program length.	X†
B	(Loader feature switch DMNSW must have been set to nonzero when Loader was assembled for this switch to be available) Block transfer the loaded job's symbol table from its normal position at the top of core down to the first free location. Leave small amount of core (SYMPAT) between JOBSA and bottom of symbol table to allow for user-defined symbols. /B allows programs loaded with DDT to usefully run in as much core as is available without destroying the symbol table, and can be used with large programs which do little I/O to run in less core than needed to load and yet retain DDT and all symbols.	

Linking Loader Switch Options

Switch	Meaning	Complement Switch
nnnnC	Create Chain file; use first block data for program break; nnnn (if nonzero) is starting address. Terminate Linking Loader.	
D	Load DDT; enter Load with Symbols: Mode (S); turn off Library Search Mode (N). Terminates specification.	
E	Upon termination of loading, control will be transferred to user's program starting address (starting address of last program loaded). Equivalent to typing START following exit from Loader.	
F	Perform a library search of LIB40; exit from Load With Symbols Mode. Terminates specification.	
nnnnG	Perform an automatic search of LIB40 if any undefined globals remain (unless the /P switch is used); list any still-undefined globals; set the starting address of the program as nnnn; exit to the Monitor. Use \$, instead, if starting address to be used is the one originally specified.	
H	Load this two-segment program as a one-segment program. Use before any files are loaded.	
I	Set the loader to ignore the starting addresses in binary input.	J
J†	Set the loader to accept the starting address of this binary input program.	I
L	Enter the library search mode.	N†
M	Print the storage map and undefined globals. Terminate specification.	
N†	Turn off the Library Search Mode.	L
nnnnO	Load beginning at numeric argument (octal) if nonzero.	
P	Prevent an automatic library search.	Q†
Q†	Allow an automatic library search. Turn off the S switch.	P
NOTE		
† indicates those switches set when Loader is in its initial state.		

Linking Loader Switch Options

Switch	Meaning	Complement Switch
nnnnR	Create Chain file; use first FORTRAN IV program break; nnnn (if nonzero) is starting address. Terminate Linking Loader.	
S	Load with local symbols.	W [†]
T	Loads SYS:DDT.REL; turns on S switch; upon termination of loading transfers control to DDT for program testing. Equivalent to typing /D in command string and, then, after exit from Loader, typing DDT.	
U	List undefined global symbols on the output list device. Terminates specification.	
V	Load the reentrant FORTRAN run-time system. Use before any files are loaded.	
W [†]	Load without local symbols.	S
X [†]	Suppress listing of global symbols for zero-length programs.	A
Y	Rewind magnetic tape before use.	
Z	Clear user's core area; reset the loader to its initial state; restore the teletype; restart loading. Terminates line.	

NOTE

[†] indicates those switches set when Loader is in its initial state.

The effect of a switch on adjacently named files in the command string depends upon whether the switch is a status switch or an action switch.

Status Switches - The status switches A, I, J, L, N, O, P, Q, S, W, X set the Loader to a particular status and have an effect on the file in whose specification it appears and on any subsequently name files in the command string (unless the switch is reset). A file specification is terminated and processed when a comma, or a colon (if the previous delimiter was a colon), a RETURN, or \$ is encountered.

*DTA5:RESID/S,/M

Local symbols are loaded for this and any following files. A storage map is printed for this file.

*DTA5:RESID,/M/S

A storage map is printed for this file; however, local symbols are not loaded for this file since the /S switch appears outside the file specification

*DTA5:RESID,/S

(which is terminated by the comma). Local symbols are loaded for any following files.

Local symbols are not loaded for this file since the /S switch appears outside the file specification (which is terminated by the comma).

Action Switches - The action switches B, C, D, E, F, G, H, M, R, T, U, V, Y request an immediate or file-independent action to be performed by the Loader and are not directly related to any specific file specification(s).

Chain Feature

The Chain feature is used to segment FORTRAN programs which are too large to be loaded into core as one unit. When switch /C or /R is specified, loading is terminated and a file acceptable to the Chain program is written.

Examples: *DSK:CHNPRG +/R or *DTA1:SEGF4 +/C

If .ext is omitted for the output Chain filename, .CHN is used.

The Chain file contains:

- a. The contents to be loaded into JOB41, JOBDDT, JOBSA, JOBFF, and JOBSYM.
- b. The data, beginning from the Chain address through the top of the core area used in loading.

The Chain address is set from JOBCHN as loaded; switch /C specifies the right half and switch /R specifies the left half. Location JOBCHN is loaded as follows: the right half contains the program break of the first FORTRAN IV BLOCK DATA program; the left half contains the program break of the first FORTRAN IV program. If switch /C or /R contains a nonzero numeric argument, this becomes the starting address of the loaded program. After the Chain file has been written correctly, the messages below are output to the teletype.

```

[ CHAIN )
  EXIT )
  ↑ C )

```

Examples

/R LOADER 6

Run Linking Loader, and assign it 6K of core.

*DTA5:RESID,SUB1,SUB2,DTA3:
COMPLX

Load and link binary program files RESID.REL, SUB1.REL, and SUB2.REL from DTA5, and the file COMPLX.REL, DTA3.

```

*/F )
*/U )
[ ?000001 UNDEFINED GLOBALS )
  ? SUB4A 000153 )

*DTAS:SUB4 )

*/U )
*LPT:/M+ $ )
[ LOADER 6+2K CORE
  EXIT )
  ↑C )

```

Carriage return initiates loading.
Force a premature search of LIB40 to resolve any undefined globals up to this point.

List on the teletype (since no output device was specified in the first command line) all globals which are still undefined.

Undefined global and location containing instruction which calls it are listed.

Knowing that the undefined global is in the binary program file SUB4, the user requests that it be loaded also.

Check if undefined global has now been resolved.

All globals are defined; print storage map on the line printer and exit to the Monitor.

Use of /E Switch:

```

.R LOADER )
*DSK:PROG1,PROG2/E $ )
[ LOADER 5+2K CORE )      (Typeout from Loader)
  ...program execution occurs here...
  EXIT )
  ↑C )

```

Diagnostic Messages

Linking Loader Diagnostic Messages

Message	Meaning
?CANNOT FIND filename.ext	The filename .ext specified is not in the file directory. If no .ext is specified for a file, the file is first searched for with the name filename.REL, and if not found, is then searched for under the null filename extension.

Linking Loader Diagnostic Messages

Message	Meaning
CANNOT FIND LOADER HIGH SEGMENT	This only occurs if the REMAP UWO failed and the GETSEG UWO failed to find the LOADER high segment. It is followed by a call EXIT .LOADER will have to be restarted by the run command.
?CHAIN DEV ERROR	A device error has occurred while writing the Chain file. Chain file is terminated.
?x CHAR. ERROR IN LOADER COMMAND	An illegal character was entered in a command string.
?DIR. FULL	The file directory of the specified list device is full and cannot contain an additional file, or a null file name was specified.
EXIT	If this message appears at the beginning of the run, either insufficient core has been assigned for loading or no console is attached to the job. EXIT normally is typed at the end of the loading process (after \$ or /G) before exiting to the monitor.
?/H ILLEGAL AFTER FIRST FILE IS LOADED	/H must be the first command to LOADER. This message is followed "LOADER RESTARTED".
?ILL. COMMON a b c d SUBROUTINE test file F4 test .rel	A file other than the first contains a program which has attempted to expand the already established COMMON area. This program must be loaded first.
?ILL. FORMAT filename .ext	The input source file is in proper checksummed binary format, but not in proper link format.
?INPUT ERROR filename .ext	A READ error has occurred on an input source device. Use of that device is terminated.
LOADER RESTARTED	This is output each time the LOADER is returned to its virgin state (i.e. /Z), it usually follows another message.
?LOW SEGMENT PROGRAM; XYZ PRECEDED BY HIGH SEGMENT PROGRAM(S)	Load all low segment programs first. This message is followed by "LOADER RESTARTED".

Linking Loader Diagnostic Messages

Message	Meaning
MORE CORE NEEDED	Loader requested additional core from Monitor, but none was available.
?symbol ignored-value old-value MUL.DEF.GLOBAL filename.ext	A global symbol definition having a value different from that of a previous definition of the same symbol has been encountered. The new value is ignored and the symbol appears in the symbol table only once.
?NO CHAIN DEVICE	No device has been specified for the Chain file.
REMAP UVO FAILURE	This is followed by LOADER RESTARTED and loading must be restarted. This can only occur when loading reentrant programs.
?x SWITCH ERROR IN LOADER COMMAND	An improper switch designation has been entered in a command string.
?x SYNTAX ERROR IN LOADER COMMAND	A syntax error has been encountered in a command string.
?dev: UNAVAILABLE	Either the device does not exist or it is assigned to another job.
?UNCHAINABLE AS LOADED	The Chain address (the half of JOBCHN selected by /C or /R) is zero.
?nnnnnn UNDEFINED GLOBALS	nnnnnn undefined globals were found.
?SYMBOL TABLE OVERLAP file.ext	nnnnnn additional words (octal) are required to load everything requested in the last command string line.
?nnnnnn WORDS OF OVERLAP file.ext	

Monitor Commands

Loading of relocatable binary files can be performed by use of the LOAD, EXECUTE and DEBUG commands. LOAD performs a straightforward load process (along with any necessary translation of source files). EXECUTE is equivalent to loading with the /E switch (on termination of loading, transfer control to user's starting address). DEBUG is equivalent to loading with the /T switch (load DDT from device SYS:, turn on /S switch, and transfer control to DDT on termination of loading).

Loader switches can be passed to the Loader by prefixing them with a % symbol.

TABLE OF CONTENTS

Page

CHAPTER 1
INTRODUCTION

1.1	Loading Procedure	1-1
1.2	Learning to Use DDT	1-1

CHAPTER 2
BASIC DDT COMMANDS

2.1	Examining Storage Words	2-1
2.2	Type-Out Modes	2-1
2.3	Modifying Storage Words	2-2
2.4	Type-In Modes	2-3
2.5	Symbols	2-3
2.6	Expressions	2-4
2.7	Breakpoints	2-4
2.8	Starting the Program	2-6
2.9	Deleting Typing Errors	2-6
2.10	Error Messages	2-7
2.11	Summary	2-7

CHAPTER 3
DDT COMMANDS

3.1	Examining the Contents of a Word	3-1
3.2	Changing the Contents of a Word	3-2
3.3	Inserting a Change, and Examining the Contents of the Last Typed Address	3-3
3.4	Starting the Program	3-5
3.5	One-Time Typeouts	3-5
3.6	Symbols	3-5
3.7	Typing In	3-6
3.8	Delete	3-8
3.9	Error Messages	3-8
3.10	Upper and Lower Case (Teletype Model 37)	3-9

CHAPTER 4
MORE DDT-10 COMMANDS

4.1	Changing the Output Radix	4-1
4.2	Type Out Modes	4-1
4.3	Breakpoints	4-3
4.4	Searches	4-7
4.5	Miscellaneous Commands	4-8

CHAPTER 5
SYMBOLS AND DDT ASSEMBLY

5.1	Defining Symbols	5-1
5.2	Deleting Symbols	5-2
5.3	DDT Assembly	5-2
5.4	Field Separators	5-3
5.5	Expression Evaluation	5-4
5.6	Special Symbols	5-4

CHAPTER 6
PAPER TAPE

6.1	Paper Tape Control	6-1
-----	--------------------	-----

APPENDIX A
SUMMARY OF DDT FUNCTIONS

		A-1
--	--	-----

APPENDIX B
EXECUTIVE MODE DEBUGGING (EDDT)

		B-1
--	--	-----

APPENDIX C
STORAGE MAP FOR DDT

		C-1
--	--	-----

APPENDIX D
OPERATING ENVIRONMENT

		D-1
--	--	-----

ILLUSTRATIONS

6-1	RIM10B Block Format	6-2
-----	---------------------	-----

TABLES

3-1	Special Character Functions	3-4
-----	-----------------------------	-----

CHAPTER 1 INTRODUCTION

DDT-10 (for Dynamic Debugging Technique)* is used for on-line checkout and testing of MACRO-10 and FORTRAN programs and on-line program composition in all PDP-10 software systems.

After the user's source program has been assembled or compiled, the user's binary object program (with its symbol table) may be loaded along with DDT. DDT occupies about 2K of core.

By typing commands to DDT, the user may set breakpoints where DDT will suspend execution of his program and await further commands. This allows the user to check out his program section by section. Either before starting execution or during breakpoint stops, the user may examine and modify the contents of any location. Insertions and deletions may be done in symbolic source language or in various numeric and text modes at the user's option. DDT also performs searches, gives conditional dumps, and calls user-coded debugging subroutines at breakpoints.[†]

Symbolic on-line debugging with DDT provides a means for rapid checkout of new programs. If a bug is detected, the programmer makes changes quickly and easily and may then immediately execute the corrected section of his program.

1.1 LOADING PROCEDURE

The user loads the program to be debugged and DDT with the Linking Loader. (The /D switch commands the Loader to load DDT.) To transfer control to DDT, the user types the Monitor command,

DDT ;

After DDT responds by skipping two lines, the user may begin typing commands to DDT.

1.2 LEARNING TO USE DDT

This manual is designed to make DDT easy to use. A survey was made of several programmers who use DDT frequently, and it was learned that most debugging is done with a limited set of commands. These basic commands are described in the next chapter. When learning DDT, it is recommended that the reader concentrate on learning to use the commands in Chapter 2. If more detailed information is required, skip ahead to later chapters.

*Historical footnote: DDT was developed at MIT for the PDP-1 computer in 1961. At that time DDT stood for "DEC Debugging Tape." Since then, the idea of an on-line debugging program has propagated throughout the computer industry. DDT programs are now available for all DEC computers. Since media other than tape are now frequently used, the more descriptive name "Dynamic Debugging Technique" has been adopted, retaining the DDT acronym. Confusion between DDT-10 and another well known pesticide, dichloro-diphenyl-trichloroethane (C₁₄H₉Cl₅) should be minimal since each attacks a different, and apparently mutually exclusive, class of bugs.

After reading Chapter 2, practice debugging, using the basic commands. This may be all that will ever be needed. Read the following chapters which describe the entire command set in detail; this should be read when the basic commands are understood.

After learning the system, the Summary of Commands, listed by function in Appendix A, will be useful for quickly finding any DDT command. This summary, along with the chapter on Basic Commands, is also available in the PDP-10 Systems User's Guide (DEC-10-NGCA-D).

CHAPTER 2

BASIC DDT COMMANDS

The DDT commands most frequently used by programmers are described in this chapter. Many programs are debugged successfully using only these basic commands.

This chapter introduces the main features of DDT to the uninitiated user. Later chapters describe in detail these basic commands, less frequently used commands and other more complex options.

2.1 EXAMINING STORAGE WORDS

By using DDT, a programmer may examine the contents of any storage word by typing the address of the desired word followed immediately by a slash (/). For example, to type out the contents of a location whose symbolic address is CAT, the user types,

```
CAT/
```

DDT now types out the contents (preceded and followed by tabs) on the same line¹.

```
CAT/ MOVEM AC,DOG+21
```

The word labeled CAT is now considered to be opened, and DDT has set its location pointer to point to this address.

2.2 TYPE-OUT MODES

The preceding example showed DDT typing out the contents of location CAT as a symbolic instruction with its address field also relative to a symbol. This is the type-out mode in which DDT is initialized. It is also initialized to type all numbers in the octal radix. The user may ask DDT to re-type the preceding quantity as a number in the current radix by typing an equal sign (=). For example²,

```
CAT/ MOVEM AC,DOG+21 = 202400,,6736
```

DDT has numerous commands which reset the type-out mode permanently, temporarily, or for only one typeout. The modes that can be selected include numeric constants, floating point numbers, ASCII and SIXBIT text modes, and half-word format. Absolute or relative addressing and different radices may similarly be selected. For example, to change the current type-out mode to ASCII text, the user types the command³

```
ST
```

¹In this manual information typed out by DDT is underlined to distinguish DDT output from user-typed input.

²The two commas indicate that 202400 is in the left half of CAT, and 6736 is in the right half.

³The Teletype keys ALTMODE (ALT), PREFIX (PREFIX), or ESCAPE (ESC) are all equivalent and echo as \$.

or, to change the current type-out mode to half-word format, he types

\$H

or, to select decimal numbers in his typeouts, he types

\$10R

Using these commands (and others described in Chapter 3), a programmer may examine any location in the mode most appropriate to the information stored there. The semicolon (;) commands DDT to retype the preceding quantity in the current mode. Combining this command with a mode change gives results such as the following:

```

CAT/ MOVEM AC,DOG+21 $10R; MOVEM AC,DOG+17
or   CAT/ MOVEM AC,DOG+21 $H; 202400,,DOG+21
or   TEXT/ ANDM 1,342212(10) $T; ABCDE

```

2.3 MODIFYING STORAGE WORDS

Once a word has been opened, its contents may be changed by typing the desired new contents immediately following the typeout produced by DDT. A carriage return will command DDT to make the indicated modification and close the word. For example,

```
CAT/ MOVEM AC,DOG+21 MOVNM AC2,DOG+21)
```

The carriage return simply closes the previously examined register without opening another¹. The line feed (↓) may also be used to close a word after examining (and optionally modifying) it. The line feed also commands DDT (1) to echo a carriage return, (2) close the current word (making a modification if one was typed), (3) add one to DDT's location pointer, and (4) type out the new pointer value and the contents of that address. Thus, if a line feed had been used in the previous example, the result would be:

```

CAT/ MOVEM AC,DOG+21 MOVNM AC2,DOG+21↓
CAT+1/ AOBJN XR6,LOOPS

```

Location CAT+1 is now open and may be modified if desired.

The vertical arrow (↑) is similar to the line feed command except that the location counter is decremented by one. Therefore, if the user continued the previous example by typing ↑ the result would be

```

CAT+1/ AOBJN XR6, LOOP5↑
CAT/ MOVNM AC2,DOG+21

```

¹The carriage return command has the additional property of causing temporary type-out modes to revert to permanent mode.

Location CAT is thus displayed and shows the result of the modification made in the previous example.

The tab (→) and backslash (\) both close the current register and open the address last typed (whether typed by DDT or the user). However, tab sets DDT's location pointer (.) to this new address while backslash leaves it unaltered. A more complex example may clarify the usefulness of these commands.

```
CAT+1/ AOBJN XR6,LOOPS →|
LOOPS/ CAMGE AC2,TABL(XR6) CAMG
AC2,TABL+1(XR6)\SETZI 0=401000,0 ↓
LOOPS+1/ JUMPL AC3,FAULT JUMPL AC2,FAULT →|
FAULT/ JRST 4,FAULT
```

2.4 TYPE-IN MODES

The examples in the preceding section showed modifications made as symbolic instructions in a form identical to MACRO-10 machine language. It is also possible to enter various numbers and forms of text.

Octal values may be typed in as octal integers with no decimal point. Numeric strings with numbers following the decimal point imply decimal floating-point numbers. The E-notation may also be used on floating-point numbers. Some examples are:

Octal:	1234	7777777777	-6	0
Decimal integers:	6789	99999999.	-25.	0.
Floating-point numbers:	78.1	0.249876E-10	-4.00E+20	0.0
Incorrect formats:	76E+2	76.E+2 instead write 76.0E+2		

To enter ASCII text (up to five characters left justified in a word), type a double quote (") followed by any printing character to serve as a delimiter, then type the one to five ASCII characters and repeat the delimiter. For example:

```
"/ABCDE/           (/ is the delimiter)
"ABCDEA           (A is the delimiter)
```

Note that the mode of a quantity typed in is determined by the user's input format and is unaffected by any type-out mode settings.

2.5 SYMBOLS

The user's symbol tables are loaded by the Linking Loader when it loads programs and DDT. However, initially DDT is set to treat only global symbols (created by INTERNAL and ENTRY pseudo-ops in MACRO-10) as being defined. This means that only global symbols will be used for relative

address typeouts and, likewise, only these globals can be referenced when typing in symbolic modifications. In order to make the local symbols within a particular program available to DDT, the user types the program name (this comes from the MACRO-10 TITLE statement or the FORTRAN IV SUBROUTINE or FUNCTION statement) followed by ALTMODE and a colon (\$:). For example, the command

```
ARCTAN$:
```

will unlock the local symbols in the program named ARCTAN. This provision in DDT permits the user to debug several related subroutines simultaneously and reference the local symbol table of each independently without fear of multiply-defined local symbols. If the user's program is not titled, the command MAIN.\$: will unlock the local symbol table.

The user may also insert symbols into the symbol table. To insert a symbol with a particular value, type the value, followed by a left angle bracket (<), the symbol, and a colon (:). Some examples are

```
707<CONS: 27<X: 12.1E+2<NUMB: ADR+12<ADRX:
```

To assign a symbol with a value equal to DDT's location pointer, simply type the symbol followed by a colon. For example,

```
XFER+4/ JRST @ TABL(3) BRNCH:
```

will cause BRNCH to be defined with the value XFER+4.

2.6 EXPRESSIONS

DDT permits the user to combine symbols and numeric quantities into expressions by using the following characters to indicate arithmetic operators.

- + The plus sign indicates 2's complement addition
- The minus sign indicates 2's complement subtraction
- * The asterisk indicates integer multiplication
- ' The single quote or apostrophe indicates integer division (remainder discarded) -- slash cannot be used to indicate division since it has another use in DDT.

As usual in arithmetic expressions, the evaluation proceeds from left to right with multiplication and division performed before addition and subtraction.

2.7 BREAKPOINTS

The breakpoint facility in DDT provides a means of suspending program operation at any desired point to examine partial results and thus debug a program section by section. The simpler facts about breakpoints are presented next; the use and control of conditional breakpoints is deferred to Paragraph 4.2.

2.7.1 Setting Breakpoints

The programmer can automatically stop his program at strategic points by setting as many as eight breakpoints. Breakpoints may be set before the debugging run is started, or during another breakpoint stop. To set a breakpoint, the programmer types the symbolic or absolute address of the word at the location point in which he wants the program to stop, followed by \$B. For example, to stop when location 6004 is reached, he types,

```
6004$B
```

Breakpoint numbers are normally assigned by DDT in sequence from 1 to 8. The user may instead assign breakpoint numbers himself when he sets a breakpoint by typing,

```
$NB
```

where n is the breakpoint number ($1 \leq n \leq 8$), for example,

```
CAT+3$4B DOG+1$7B 6004$8B
```

When the programmer sets up a breakpoint he may request that the contents of a specified word be typed out when the breakpoint is reached. To do this, the address of the word to be examined is inserted, followed by two commas, before the breakpoint address. Some examples are

```
DOG,,CAT$3B AC1,,LOOP+2$B X,,6004$8B
```

2.7.2 Breakpoint Restrictions

The locations where breakpoints are set may not

- a. be modified by the program
- b. be used as data or literals
- c. be used as part of an indirect addressing chain
- d. contain the user mode Monitor command, INIT.

2.7.3 Breakpoint Type-Outs

When the breakpoint location is reached, DDT suspends program execution without executing the instruction at the breakpoint location. DDT then types the breakpoint number and the Program Counter value at the time the breakpoint is reached (this value will differ from the typed-in breakpoint address if the breakpoint is executed by an XCT instruction elsewhere in the program). The format of this typeout is as shown in the following examples:

```
$4B >> CAT+3 $7B >> DOG+1 $8B >> 6004
```

If the user requested that a specified address be examined at that breakpoint, it will be opened; for example,

```
$3B >> CAT DOG/ SOJGE 3,GOAT+6
```

2.7.4 Removing and Reassigning Breakpoints

The user may remove a breakpoint by typing,

`Ø$NB`

where n is the number of the breakpoint to be removed. For example,

`Ø$2B`

removes the second breakpoint. All assigned breakpoints are removed by typing

`$B`

The user may reassign a breakpoint without formally removing it. Thus, if he has set breakpoint No. 2 at location ADR (via the command `ADR$2B`) he may reassign No. 2 to `LOC+6` by typing `LOC+6$2B`.

2.7.5 Proceeding From a Breakpoint

Program execution may be resumed (in sequence) following a breakpoint stop by typing the proceed command, `$P`.

If the user does not wish to stop until the nth time that this breakpoint is encountered he types,

`N$P`

Then this breakpoint will be passed n-1 times before a break occurs.

2.8 STARTING THE PROGRAM

The program is started by typing

`$G`

This starts the program at the previously specified starting address in location `JOBSA`. (Typically this is the address from the `MACRO-10 END` statement.) The programmer may start at any other location by typing that address followed by `$G`. For example,

`4000$G`

starts the program at the instruction stored at location 4000. `BEGIN$G` starts the program at the symbolic location `BEGIN`.

The start command may also be used to restart from a breakpoint stop when it is not desired to continue in sequence from the point where program execution was suspended.

2.9 DELETING TYPING ERRORS

Any partially typed command may be deleted by pressing the RUB OUT key. This causes DDT to ignore any preceding (unexecuted) partial command, and DDT types `XXX`. The correct command may then be retyped.

2.10 ERROR MESSAGES

If the user types an undefined symbol which cannot be interpreted by DDT, U is typed back .
If an illegal DDT command is typed, or a location outside the user's assigned memory area is referenced
? is typed back .

2.11 SUMMARY

As was said in the beginning, these basic commands are sufficient for debugging many
programs. Complete descriptions of all DDT commands are explained in the following chapters.

CHAPTER 3

DDT COMMANDS

When DDT is initialized, it is set to type out in the symbolic instruction format with relative addresses, and to type out numbers in octal radix.

3.1 EXAMINING THE CONTENTS OF A PROGRAM STORAGE WORD

To type out the contents of a storage word, the programmer types the address, followed immediately by a slash (/). For example, to examine the contents of a word whose symbolic address is ADR, the user types,

```
ADR/
```

DDT types out the contents on the same line. In this manual, information typed out by DDT is underlined.

```
ADR/ MOVE A,CC1
```

The word labeled ADR is now considered to be opened, and DDT continues to point to this address. The point, or period, character (.) represents DDT's location pointer, and may be used to type out its contents, as in the following command.

```
./ MOVE A, CC1
```

Since we did not change the contents, they are the same, but we used the location pointer to re-examine the currently opened word. Similarly, the programmer may use the period (.) as an arithmetic expression component, such as

```
./+5/ SOJGE 2,ADR+3
```

DDT's location pointer is set to a new value by the / command when preceded by an address. For example,

```
201/ 0
```

sets the location pointer to 201. If the user types / without typing an address, the contents of the location addressed in the last typeout are typed.

```
667/ MOVE 1,6 / 0  
./ MOVE 1,6
```

Location 667 contains the instruction MOVE 1,6. The second slash displays the contents of Accumulator 6, which is zero. This does not change the location pointer, which is still pointing to location 667.

```
ADR/ MOVE A,CC1 / ADD 2,SUM+7
```

It should also be noted that the spaces which occur after DDT complete the typing of the contents of ADR are automatically produced by DDT, not the user.

The left square bracket ([)¹ has the same effect as the slash, (the address immediately preceding the [will be opened). However, [forces the typeout to be in numbers of the current radix.

```
ADR [ 11 (OCTAL)
```

```
ADR [ 9. (DECIMAL)
```

The right bracket (])^{*} has the same effect as the slash except that it forces the typeout to be in symbolic instructions.

```
ADR+23 ] MOVE 15,LIST+2
```

The exclamation point (!) works like the slash except that it suppresses type out of contents of locations until either /, [, or] is typed by the user. The LINE FEED (!) commands DDT to type out the contents of ADR+1.

```
ADR! MOVE AC,555↓ (1)
```

```
ADR+1! ) (2)
```

```
ADR/ MOVE AC,555 (3)
```

Thus, in step (1) of the example the contents of ADR are not typed out, but the address is opened to modification and MOVE AC,555 has been typed in by the user.

Step (2) of the example shows that the location pointer has been incremented by one and the contents of ADR+1 are not typed out. This is because the exclamation point is still in effect and will continue to take effect until /, [, or] is typed in by the user. In this case, the slash terminates the effect of the exclamation point.

Step (3) shows that the modification (MOVE AC,555) of ADR typed in Step (1) has been accomplished.

3.2 CHANGING THE CONTENTS OF A WORD

After a word is opened, its contents can be changed by typing the new contents following the type out by DDT, followed by a carriage return. For example,

```
ADR/ MOVE A,CC1 MOVE A,CC2 )
```

The carriage return closes the open word, but does not move the location pointer. A LINE FEED (!) command could also be used to make this modification. A LINE FEED causes a carriage return, adds

¹On Teletype Models 33 and 35 the left square bracket ([) is produced by holding the SHIFT key down and striking the K key. The right square bracket (]), is produced by holding the SHIFT key down and striking the M key.

one to DDT's location counter (moves the pointer), types out the resulting address and the contents of the new address. Thus, if we conclude our last example with a LINE FEED

```
ADR/   MOVE A,CC1  MOVE A,CC2 ↓
ADR+1/ ADD 3,CC3
```

ADR+1 is now open, and may be modified by the user.

The vertical arrow (↑)¹ works similarly, except that one is subtracted from the location pointer. The open word is closed (modified if a change is given) and the new address and contents are typed out.

```
ADR+1/ ADD 3,CC3↑
ADR/   MOVE A,CC2
```

Since the vertical arrow subtracts one from the pointer, the resulting address is ADR, and the contents now show the change made in the previous example.

3.3 INSERTING A CHANGE, AND EXAMINING THE CONTENTS OF THE LAST TYPED ADDRESS

The horizontal tab (→) causes a carriage-return line feed, then sets the location pointer to the last address typed (the new address if a modification was made) of the instruction in the register just closed. Then DDT types this new address, followed by a slash and the contents of that location, as shown below.

```
ADR5/ JRST ADR1  JRST ADR →
ADR/  MOVEM B,CC2 →
CC2/  666
```

The backslash (↘)² opens the word at the last address typed and types out the contents. However, backslash does not change the location pointer. The backslash closes the previously opened word and causes it to be modified if a new quantity has been typed in.

```
ADR/ MOVE A,CC2  JRST X \ MOVE AC,3
```

The use of the backslash accomplishes two things. First it changes ADR by replacing its contents with JRST X. Second, the backslash causes DDT to type out the contents of X, namely, MOVE AC,3. The location pointer continues to point to ADR, but now location X is open and may be modified if desired.

¹ ↑ is produced by SHIFT-N on Teletype Models 33 and 35. The backspace key may be used instead of ↑ on Teletype Model 37.

² ↘ is produced by SHIFT-L on Teletype Models 33 and 35.

If the line-feed control character and the vertical arrow were used in conjunction with the backslash, the results would be as follows.

```

ADR/   MOVEM B,CC2  MOVE A,CC1 \ 105776 ↓
ADR+1/ MOVE A,C    †
ADK/   MOVE A,CC1  \ 105776

```

The following is a summary in table form of these special control characters and their corresponding functions. For example, the chart shows that the forward slash (/) will examine the contents of an address, type out in the current mode, open the address, change the location pointer to the address just opened, but it does not cause a new quantity to be inserted in that address.

Table 3-1
Special Character Functions

Command Character	Type Out Contents	Mode	Address Opened	Change Location Pointer	Insert New Qty If New Qty Has Been Typed
/	Yes	Current		Yes ¹	No
[Yes	Numeric	Yes		
]	Yes	Symbolic			
!	No	-			
\	Yes ²	Current	Yes	No	Yes
TAB (→)	Yes ²	Current	Yes	Yes	Yes
† or backspace	Yes ²	Current	Yes	Yes (-1)	Yes
Line-feed (↑)	Yes ²	Current	Yes	Yes (+1)	Yes
Carriage return (↵)	No	None	No (closes)	No	Yes

A ? typed by DDT when examining a location indicates that the address of the location is outside the user's assigned memory area. A ? typed when depositing indicates that the location cannot be written in, because it is either outside the assigned memory area or inside DDT or inside a write-protected memory segment.

¹If a user-typed quantity preceded.

²If ! has not suppressed typeout.

3.4 STARTING THE PROGRAM

The program is started by typing

\$G

This starts the program with the instruction beginning at the user's previously specified starting address taken from location JOBSA. The programmer may start at any other instruction by typing the address of that instruction followed by \$G. For example,

4000\$G OR ADR+5\$G

starts the program at the instruction stored at location 4000 or, in the second part, at the symbolic address ADR+5. The start command may also be used to restart from breakpoints when the user does not wish to proceed to the next instruction.

3.5 ONE-TIME TYPEOUTS

These commands cause a single typeout of the opened word in the mode indicated.

3.5.1 Type Out Numeric

Although DDT is initialized to type out in symbolic mode, it is often useful to change to numeric typeout. When the programmer types the equal sign (=), the last expression typed is retyped by DDT in the current radix (initially octal). This is useful when a symbolic typeout is meaningless. Since this usually indicates that numeric data is stored in that word, the user can verify this by typing = and checking the value.

3.5.2 Type Out Symbolic

If a typeout is numeric, and the user wants to examine it in symbolic mode, he types the left arrow (-). The last typed quantity is retyped as a symbolic instruction. The address mode is determined by \$A or \$R.

3.5.3 Type Out in Current Mode

To retype a typeout in the current mode, the user types a semicolon (;). This may be used, for example, if the user has changed the typeout mode. For example,

TEXT/ ANDM 1,342212 (10) ST; ABCDE

3.6 SYMBOLS

Before DDT commands can be used to reference local symbols in the program Symbol Table, the user must type the program name as specified in the MACRO-10 TITLE statement, or the FORTRAN IV

SUBROUTINE or FUNCTION statement, followed by a dollar sign and a colon. For example,

MAIN\$:

makes the local symbols in the program called MAIN available. Since the user can debug several related subroutines simultaneously, reference to several independent symbol tables is permitted, each of which may use the same local symbols with different values. Global symbols, such as those specified in MACRO-10 INTERNAL statements, may always be referenced.

The user may insert (or redefine) a symbol in the symbol table by typing the symbol, followed by a colon. The symbol will have a value equal to the address of the location pointer (.).

X/ ADD1 3,N TAG:

causes TAG to be defined with the same value as X. All user defined symbols are global.

The user may also directly assign a value to a symbol by typing the value, a left angle bracket (<) and the symbol, terminated by a colon. This is the equivalent of a MACRO-10 direct assignment statement. Some examples are,

707 <CONS: 12.1E+2 <NUMB:
27 <X: 101 <MIL:

3.7 TYPING IN

To change or modify the contents of a word, the user may type symbolic instructions, numbers, and text characters. Type-ins are interpreted by DDT in context. That is, DDT tests the data typed in to determine whether it is to be interpreted as an instruction, a number (octal or decimal), or text. Typeout mode settings, such as \$S, \$C, and \$nR, do not affect typed input.

The user may type the following:

- a. Symbolic Instructions
- b. Numbers
 - (1) Octal integers
 - (2) Fixed-point decimal integers
 - (3) Floating-point decimal mixed numbers
- c. Text
 - (1) Up to five PDP-10 ASCII characters, left justified in a word
 - (2) Up to six SIXBIT characters, left justified in a word
 - (3) A single PDP-10 ASCII character, right justified in a word
 - (4) A single SIXBIT character, right justified in a word
- d. Symbols

Anything that is not a number or text is interpreted by DDT as a symbol.

3.7.1 Typing In Symbolic Instructions

In general, a new symbolic instruction is written for insertion by DDT, in the same way the instruction is written as a MACRO-10 source program statement. For example,

```
X/ 0 ADD AC1,DATE
```

where a space terminates the operation field, and a comma terminates the accumulator field. For example: (1) In DDT, the operation code determines the interpretation of the accumulator field. If an I/O instruction is used, DDT inserts the I/O device number in the correct place, and (2) indirect and indexed addresses are written, as in MACRO-10 statements, where @ precedes the address to set the indirect bit, and the index register specified follows in parentheses.

```
X/0 ADD 4,@NUM (17)
```

To type in two 18-bit halfwords, the left and right expressions are separated by two commas. For example,

```
X/ 0 A,,B
```

This is similar to the MACRO-10 statement

```
XWD A,B
```

3.7.2 Typing In Numbers

A typed-in number is interpreted by DDT as octal if it does not contain a decimal point.

The following examples are octal type-ins:

```
1234 -10101
772 . 777777777777
```

Fixed-point decimal integers must contain a decimal point with no digits following.

```
1234. -99. 877.
```

Floating-point numbers may be written in two formats. With a decimal point and a digit following the decimal point:

```
101.1 1234.5 999.0 -2.71828
```

Or as in MACRO-10, with E indicating exponentiation:

```
12.0E+2 77.0E+5 12.34E2 31.4159E-1
```

3.7.3 Typing In Text Characters

To type in up to five PDP-10 ASCII characters, left justified in an opened word, the user types a quotation mark, followed by any printing delimiting character, then the text characters, and terminated by the delimiting character. The following examples are legal:

```
"/TEXT/ "ABCDEFA
```

In these cases, / and A are
the delimiting characters

Lower case letters are converted to upper case. Characters outside the SIXBIT set are illegal, and DDT types a ?

To type in up to six SIXBIT characters, left justified in an opened word, the user types "\$", followed by any delimiting character, then the text characters, and terminated by repeating the delimiting character. The two examples below are SIXBIT type ins.

```
$"/DIVIDE/ $"EXXXXXE
```

To type in a single PDP-10 ASCII character, right justified in an opened word, the user types a quotation mark, followed by a single ASCII text character, then by an ALT MODE.

```
"Q$ "/$ "?$
```

To type in a single SIXBIT character, right justified in an opened word, the user types an ALT MODE, followed by a quotation mark, a single SIXBIT text character and terminated by an ALT MODE.

```
$"Q$ $"M$ $"$$
```

3.7.4 Arithmetic Expressions

Numbers and symbols may be combined into expressions using the following characters to indicate arithmetic operations.

- + The plus sign means 2's complement addition.
- The minus sign means 2's complement subtraction.
- * The asterisk means integer multiplication.
- ' The single quote means integer division with any remainder discarded. (The Slash has another function.)

Symbols and numbers are combined by +, -, *, ' to form expressions. Examples:

```
6+2
```

```
S'2.51+BASE
```

```
2*3+1
```

3.8 DELETE

Any partially typed command may be deleted by pressing the RUB OUT key. This causes DDT to ignore any preceding (unexecuted) partial command and DDT types XXX. The correct command may then be retyped.

3.9 ERROR MESSAGES

If the user types an undefined symbol which cannot be interpreted by DDT, U is typed back. If an illegal DDT command is typed, ? is typed back. Examining or depositing into a location outside

the user's assigned memory area causes DDT to type a ? Depositing in a write-protected high memory segment also results in a ? typeout.

3.10 UPPER AND LOWER CASE (TELETYPE MODEL 37)

DDT will accept alphabetic input in either upper or lower case. Lower case letters are internally converted to upper case, except when inputting text where they are taken literally as explained in Section 3.7.3.

DDT output is in upper case, except for text which is taken literally.

CHAPTER 4

MORE DDT-10 COMMANDS

This chapter describes other type-out modes, conditional breakpoints, searches and additional features. Commands are available to reset the initial settings so that numeric data can be typed out in a radix chosen by the user, in floating-point format, in RADIX50 format, as halfwords (two addresses) and as bytes of any size. The contents of a storage word may also be typed out as 7-bit PDP-10 ASCII text, or SIXBIT text characters. (See MACRO-10 Manual, Appendix 5.)

Searches can be made in any part of the program for any word, not-word (inequality), or effective address. The user specifies the instruction or data to be searched for and the limits of the search.

Breakpoints can be set conditionally, so that a program stop occurs if the condition is satisfied. In addition, a counter can be set up allowing the user to specify the number of times a breakpoint is passed before a program stop occurs.

4.1 CHANGING THE OUTPUT RADIX

Any radix (≥ 2) may be set by typing \$nR, where n is the radix for the next typeout only, and n is interpreted by DDT as a decimal value. The radix is permanently changed when the double dollar sign is used in the command \$\$nR. To change the type-out radix permanently to decimal, the user types,

```
$$10R
```

When the output radix is decimal, DDT follows all numbers with a point.

4.2 TYPE OUT MODES

When DDT-10 is loaded, the type-out modes are initialized to produce symbolic instructions with relative addresses. For numeric typeouts, the radix is initially set to octal.

These modes may be changed by the user. The duration, or lasting effect of a type-out mode change is also set by the user. Prevailing modes, which are semipermanent, are preceded by a single dollar sign. In addition, some mode changes effect only one typeout, such as the equal sign, which causes DDT to retype the last typed quantity in numeric mode.

In general, prevailing modes are changed by replacing them with another prevailing mode or by reinitializing the system. Temporary modes remain in effect until the user types a carriage return (↵), or re-enters DDT. One-time modes apply only to a single typeout.

4.2.1 Primary Type-out Modes

- \$S (OR \$\$\$)** Type out symbolic instructions. The address part interpretation is set by \$R or \$A.
 Example: `$S ADR/ ADD AC1, TABLE+3`
- \$A (OR \$\$A)** Type out the address parts of symbolic instructions, and both addresses when the mode is halfword, as absolute numbers in the current radix.
 Example: `$A ADR/ ADD 4002`
- \$R (OR \$\$R)** Type out addresses as relative addresses.
- \$.C (OR \$\$C)** Type out constants, i.e., as numbers in the current radix.
 Example: `$.C ABLE/ 254111,,4050`
- \$F (OR \$\$F)** If the output radix octal and the left half is not 0, the word will be divided into halves separated by commas. Type out the contents of storage words as floating-point numbers.
 Example: `$F X/ #0.17516230E-45`
 The number sign (#) indicates the number is unnormalized.
- \$T (OR \$\$T)** Type out as 7-bit ASCII text characters. Left-justified characters are assumed unless the leftmost character is null. If the leftmost character is null, then right-justified characters are assumed.
 Example: `$T REX/ ABCDE`
- \$6T (OR \$\$6T)** Type out as SIXBIT text characters.
 Example: `$6T HEX/ ABCDEF`
- \$\$T (OR \$\$\$T)** Type out symbols in radix 50 mode. (See MACRO-10 Manual, Appendix 6.)
 Example: `$$T 13774/ 4 CREF = 40003,,261550`
- \$H (OR \$\$H)** This command causes the typeout to be in halfwords, the left half separated from the right half by double commas. The address mode interpretation is determined by \$R or \$A.
 Example: `$A $H Z/ .4503,,4502`
 Example: `$R $H Z/ TABL+14,,TABL+13`
- \$NO (OR \$\$NO)** Type out in n-bit bytes, where n is decimal. (Use the letter O, not zero).
 Example: `$60 BYTS/ 22,23, 1, 73, 51, 46`
 As in all DDT typeouts, leading zeros are suppressed.

4.3 BREAKPOINTS

4.3.1 Setting Breakpoints

The programmer can automatically stop his program at strategic points by setting up to eight breakpoints. Breakpoints may be set before the debugging run is started, or during another breakpoint stop. To set a breakpoint, the programmer types the symbolic or absolute address of the word at the location which he wants the program to stop, followed by \$B. For example, to stop when location 4002 is reached, he types,

```
4002$B
```

If all eight breakpoints are in use, DDT will type a question mark. The user may assign breakpoint numbers when he sets a breakpoint by typing ADR \$nB, where n is the breakpoint number (1<n<8). For example,

```
SYM$3B ADR$7B
```

If n is not entered DDT will assign 1 through 8 in sequence. In the previous example, when ADR is reached, DDT types,

```
$7B >> ADR
```

indicating that the break has occurred at location ADR, and breakpoint No. 7 was encountered. The break always occurs before the instruction at the breakpoint address is executed.

If the instruction at the breakpoint location is executed by an XCT instruction, the typeout will show the address of the XCT instruction, not the location of the breakpoint. The program stops at each breakpoint address, and the programmer can then type other commands to examine and debug his program.

When the programmer sets a breakpoint, he may request that the contents of a word be typed out when a breakpoint is reached. To do this, the address of the word to be examined is inserted, followed by two commas, before the breakpoint address.

```
X,,4002$2B
```

When address 4002 is reached, DDT types out,

```
$2B>>4002 X/ ADD AC,Y+2
```

where ADD AC, Y+2 is the contents of X. Location X is left open at this point. Location 0 may not be typed out in this way because a zero argument implies no typeout.

4.3.2 Removing Breakpoints

The user may remove a breakpoint by typing,

```
0$NB
```

where n is the number of the breakpoint to be removed. Therefore,

$0\$2B$

removes the second breakpoint. All assigned breakpoints are removed by typing

$\$B$

The user may reassign a breakpoint. If he has set breakpoint No. 2 at location ADR ($ADR\$2B$), he may reassign No. 2 to $ADR+1$ by typing $ADR+1\$2B$.

4.3.3 Restrictions for Breakpoints

Breakpoints may not be set on instructions that are

- a. Modified by the program
- b. Used as data or literals
- c. Used as part of an indirect addressing chain
- d. The user mode Monitor command, INIT

A breakpoint at any other Monitor command will operate correctly, except that if the Monitor command is in error, the Monitor will type out an error and the Program Counter, but the Program Counter will be internal to DDT and meaningless to the user.

4.3.4 Restarting After a Breakpoint Stop

To resume the program after stopping at a breakpoint, the user types the proceed command,

$\$P$

The program is restarted by executing the instruction at the location where the break occurred. If the user types $n\$P$, this breakpoint will be passed $n-1$ times before a break can occur; the break will occur the n th time. If n is not specified, it is assumed to be one. If the user proceeds by typing $\$\P (or $n\$\P), the program will proceed automatically when the program breaks again. If DDT encounters an XCT loop or the Monitor command INIT when proceeding, a question mark will be typed.

Alternatively, the user may restart at any location by typing the start command,

$ADR\$G$

where ADR is any program address, or $\$G$, which restarts at the previously specified starting address in location JOBSA.

4.3.5 Automatic Restarts from Breakpoints

If the user requests DDT to type out the contents of a word and then continue program execution without stopping, he types two ALTMODES when specifying the breakpoint address.

$AC,,ADR\$B$

When ADR is encountered, the contents of AC are typed out and program execution continues. To get out of the automatic proceed mode, remove the breakpoint or reassign it with a single \$; it may be necessary to use ↑ C and DDT ↵ to get back to DDT to do this. In executive mode, hit any teletype key during the typeout.

4.3.6 Checking Breakpoint Status

The user may determine the status of a breakpoint by examining locations \$nB, \$nB+1, and \$nB+2.

\$nB contains the address of the breakpoint in the right half; the address of the location to be examined in the left half. If both halves equal zero, the breakpoint is not in use.

\$nB+1 contains the conditional breakpoint instruction. (See Paragraph 4.3.7.)

\$nB+2 contains the proceed count.

4.3.7 Conditional Breakpoints

Breakpoints may be set up conditionally in two ways. The user may provide his own instruction or subroutine to determine whether or not to stop, or he may set a proceed counter which must be equal to or less than zero in order for a break to occur.

When a breakpoint location is reached, DDT enters its breakpoint analysis routine consisting of five instructions.

SKIPE	\$NB+1	; Is the conditional break instruction 0?
XCT	\$NB+1	; No, execute conditional break instruction
SOSG	\$NB+2	; Decrement and test the proceed counter
JRST	break routine	
JRST	proceed routine	

If the contents of \$nB+1 are zero (indicating that there is no conditional instruction), the proceed counter at \$nB+2 is decremented and tested. If it is less than or equal to zero, a break occurs; if it is greater than zero the execution of the user's program proceeds with the instruction where the break occurred.

If the conditional break instruction is not zero, it is executed. If the instruction (or the closed subroutine) does not cause a program counter skip, the proceed counter is decremented and tested as above. If a program counter skip does occur, a break occurs. If the conditional instruction is a call to a closed subroutine which returns skipping over two instructions, execution of the user's program proceeds.

If the user wishes a break to occur based only on the conditional instruction, he should set the proceed counter to a large number so that the proceed counter will never reach zero.

4.3.7.1 Using the Proceed Counter - If the user wishes to proceed past a breakpoint a specified number of times, and then stop, he inserts the number of passes in \$NB+2, which contains the proceed count.

The proceed counter may be set in two ways. The first way is by direct insertion. For example,

```
$NB+2 / 0 20
```

sets the counter to 20. The second method is as follows. After stopping at a breakpoint, the proceed count may be set (or reset) by typing the count before the proceed command:

```
20$P
```

4.3.7.2 Using the Conditional Break Instruction - The user inserts a conditional instruction, or a call to a closed subroutine at \$nB+1. For example,

```
$3B+1 / 0 CAIGE ACC,15)
```

or

```
$4B+1 / 0 JSA 16, TEST)
```

When the breakpoint is reached, this instruction or subroutine is executed. If the instruction does not skip or the subroutine returns to the next sequential location, the proceed counter is decremented and tested, as explained in Paragraph 4.2.7. If the instruction skips or the subroutine returns skipping over one instruction, the program breaks. If the subroutine causes a double skip return, the program proceeds with the instruction at the breakpoint address.

Examples of Conditional Breakpoints

If address 6700 is reached and DDT's No. 4 breakpoint registers are as follows:

```
$4B / AC1,,6700
```

```
$4B+1 / CAIE AC!-100
```

```
$4B+2 / 200
```

AC1 contains 100, and DDT types

```
$4B>6700 AC1 / 100
```

Since AC1 contains 100, the compare instruction skips and the program breaks. If AC1 did not contain 100, \$4B+2 would be decremented by one and the user's program would continue running.

If the conditional break instruction transfers to a subroutine which, after the subroutine is executed, returns to the calling location +3, a break will never occur regardless of the proceed counter.

Example: If the internal DDT breakpoint registers (\$2B and \$2B+1) have the following contents, a break would not occur unless accumulator 3 contains 100.

\$2B/	<u>ADR</u>	
\$2B+1/	<u>JSR TEST</u>	(contains PC when JSR to subroutine TEST is made)
TEST/	Ø	
TEST+1/	AOS TEST	
TEST+2/	CAIE 3,100	
TEST+3/	AOS TEST	
TEST+4/	JRST @ TEST	

The subroutine TEST causes a double skip (the return is to the third instruction after the call) in DDT if accumulator 3 does not equal 100. A break will never occur at address ADR (regardless of the proceed counter) unless accumulator 3 contains 100.

4.3.8 Entering DDT from a Breakpoint

When a break occurs, the state of the user's program is saved, the JSR breakpoint instructions are removed, and the programmer's original instructions are restored to the breakpoint locations. DDT types out the number of the breakpoint and a symbol indicating the reason for the break, > for the conditional break instruction, >> for the proceed counter and the address in the user's program where the break occurred.

Example: If address ADR is reached in the user's program and DDT's breakpoint registers contain:

\$2B/	<u>ADR</u>	
\$2B+1/	<u>Ø</u>	
\$2B+2/	<u>Ø</u>	(proceed counter contains zero)

DDT stops the program and types,

\$2B >> ADR

4.4 SEARCHES

There are three types of searches: the word search, the not-word search, and the effective address search.

Searches can be done between limits. The format of the search command is,

a < b > c \$	$\left\{ \begin{array}{l} W \\ N \\ E \end{array} \right.$	Word search
		Not-word search
		Effective address search

where:

- a Is the lower limit of the search; 0 is assumed if this argument and its delimiter are not present.
- b Is the upper limit of the search. The lower numbered end of the symbol table is assumed if this argument and its delimiter are not present.
- c Is the quantity searched for.

The effective address search (E) will find and type out all locations where the effective address, following all indirect and index-register chains to a maximum depth of 64_{10} levels, equals the address being searched for.

Examples:

```
4517 <5000>X$E
```

```
INPUT <5000>700$E
```

Examples of DDT output, when searching for X in the above example, are as follows.

```
4517/  SETZM X
```

```
4721/  MOVE 2,X
```

```
5000/  MOVE 3,@ 4721  (indirectly addresses X through  
                        address 4721)
```

The word search (W) and the not-word search (N) compare each storage word with the word being searched for in those bit positions where the mask, located at \$M, has ones. The mask word contains all ones unless otherwise set by the user. If the comparison shows an equality, the word search types out the address and the contents of the register; if the comparison results in an inequality, the word search will type out nothing. The not-word search types nothing if an equality is reached. It types the contents of the register when the comparison is an inequality.

Examples:

```
INPT <INPT+10>NUM$W
```

```
INPT <INPT+10>0$N
```

```
$M/      This command types out the contents of the mask register, which is then  
         open. The contents of the mask register are ordinarily all ones unless  
         changed by the user.
```

```
N$M      Inserts n into the mask register.
```

4.5 MISCELLANEOUS COMMANDS

```
$Q      $Q represents the value of the last quantity typed.
```

```
ADR/   100  $Q+1)
```

```
ADR/   101
```

```
INST$X Causes the instruction INST to be executed.
```


Example:

JRST ADR\$X would cause the user's program to be started at ADR.

There are a number of circumstances when the user will want to zero out certain memory location(s). The following command provides this capability:

FIRST<LAST \$\$Z This command will zero out the memory locations between the indicated FIRST address and LAST address inclusively. If the FIRST address is not present, the location 0 is assumed. If the LAST address is not present, the location before the low-numbered end of the symbol table is assumed. In no case will locations 20-137 nor any part of DDT or DDT's symbol table be zeroed.

CHAPTER 5

SYMBOLS AND DDT ASSEMBLY

A symbol is defined in DDT as a string of up to six letters and numbers including the special characters period (.), percent sign (%), and dollar sign (\$). Characters after the sixth are ignored. A symbol must contain at least one letter. If a symbol contains numerals and only one letter, that letter must not be a B, D, or an E. These letters are reserved for binary-shifted and floating-point numbers.

Certain symbols can be referenced in one program from another. These symbols are called "global." Those which can only be referenced from within the same program are called "local" or "internal." Any symbol which has been defined as global by MACRO-10 (using the INTERNAL or ENTRY statements) will be considered as global by DDT-10 when it is referenced. FORTRAN subroutine entry points and COMMON block names are globals. All symbols which the user defines via DDT are considered to be global.

The user may want to reference a local symbol within a particular program. In order to do this he must first type the program name followed by \$:. Thus, if a user wishes to use a symbol local to program MIN, he types the command,

MINS:

This command unlocks the symbol table associated with MIN. The program name is that specified in the MACRO-10 TITLE statement. In FORTRAN, the program name is either MAIN, the name from the SUBROUTINE or FUNCTION statement, or DAT. for BLOCK DATA subprograms.

5.1 DEFINING SYMBOLS

There are two ways to assign a value to a symbol.

NUMERIC VALUE < SYMBOL:

This command puts SYMBOL into DDT-10's symbol table with a value equal to the specified NUMERIC VALUE. SYMBOL is any legal symbol defined or undefined.

Example:

305 < XVAR:

XVAR has now been defined to have the value 305.

TAG:

This command puts TAG into DDT-10's symbol table with a value equal to the address of the location pointer.

Example:

400 / ADD 2, 12012, X:

This puts the symbolic tag X into DDT-10's symbol

table and sets X equal to 400, the address of the last register opened.

5.2 DELETING SYMBOLS

There are times when the user will want to restrict or eliminate the use of a certain few defined symbols. The following three ways give the user of DDT-10 these capabilities.

SYMBOL \$\$K

SYMBOL is killed (removed) in the user's symbol table. SYMBOL can no longer be used for input or output.

Example

X\$\$K

This command removes the symbol X from the symbol table.

SYMBOL \$K

This command prevents DDT from using this symbol for typeout; it can still be used for typein. For example, the user may have set the same numeric value to several different symbols. However, he does not wish certain symbol(s) to be typed out as addresses or accumulators.

X/ MOVE J, SAV J\$K ← MOVE N, SAV N\$K ← MOVE AC, SAV

Since the user does not wish J to be typed out as an accumulator, he types in J\$K, followed by a left arrow to type out the contents of X again and MOVE N, SAV is typed out. He then repeats the above process until the desired result, namely AC, is typed out. Any further symbolic typeouts with the same number in the accumulator field of the instruction will type out as AC.

\$D

The last symbol typed out by DDT has \$K performed on it. The value of the last quantity output is then retyped automatically. For example,

A/ MOVE AC, LOC \$D MOVE AC, ABC+1

5.3 DDT ASSEMBLY

When improvising a program on-line to the PDP-10 on a Teletype, the user will want to use symbols in his instructions in making up the program. In this and in other situations, undefined symbols may be used by following the symbol with the number sign (#). The symbol will be remembered by DDT from then on. Until the symbol is specifically defined by the use of a colon, the value of the symbol is taken to be zero. Successive uses of the undefined symbol cause DDT to type out #. Appending # to all subsequent uses of the symbol enables the user to readily identify undefined (not yet defined by a colon) symbols.

Example:

```
MOVE 2,VALUE#
```

VALUE is now remembered by DDT and may be used further without the user appending the #. If subsequent instructions are given involving VALUE, DDT appends a # automatically to that symbol. Thus VALUE will always appear as VALUE followed by the # (until VALUE is defined).

Example:

```
START!      MOVE 2,VALUE#↓      (user types the #)
START+1!    ADDI 2, 50↓
START+2!    MOVEM 2↓ VALUE ↓
#           #
START+3!    JRST VALUE+#1↓      (DDT types #)
START+4!
```

(DDT types # after the plus sign because only at that point does DDT realize the symbol VALUE is complete.)

Undefined symbols can be used only in operations involving addition or subtraction. The undefined symbols may be used only in the address field.

Example:

```
MOVEI 2,3*UNDEF#
```

This is an illegal operation - multiplication with a symbolic tag (UNDEF) which has not previously been defined.

The question mark (?) is a command to DDT to list all undefined symbols that have been used in DDT up to that point in the program.

Example:

```
?
VALUE
UNDEF
```

5.4 FIELD SEPARATORS

The storage word is considered by DDT to consist of three fields: the 36-bit wholeword field; the accumulator or I/O device field; and the address field. Expressions are combined into these three fields by two operators:

Space The space adds the expression immediately preceding it (normally an op code) into the storage word being formed. It also sets a flag so that the expression going into the address field is truncated to the rightmost 18 bits.

Single Comma	The comma does three things: the left half of the expression is added into the storage word; the right half is shifted left 23 bits (into the accumulator field) and added into the storage word. If the leftmost three bits of the storage word are ones, the comma shifts the right half expression left one more place (I/O instructions thus shift device numbers into the device field). The comma also sets the flag to truncate addresses to 18 bits.
Double Comma	Double Commas are used to separate the left and right halves of a word whose contents are expressed in halfword mode.

The address field expression is terminated by any word termination command or character.

5.5 EXPRESSION EVALUATION

Parentheses are used to denote an index field or to interchange the left and right halves of the expression inside the parentheses. DDT handles this by the following generalized procedure.

A left parenthesis stores the status of the storage-word assembler on the pushdown list and reinitializes the assembler to form a new storage word. A right parenthesis terminates the storage word and swaps its two halves to form the result inside the parentheses. This result is treated in one of two ways:

a. If +, -, ', or * immediately preceded the left parenthesis the expression is treated as a term in the larger expression being assembled and therefore may be truncated to 18 bits if part of the address field.

b. If +, -, ', or * did not immediately precede the left parenthesis, this swapped quantity is added into the storage word.

Parentheses may be nested to form subexpressions, to specify the left half of an expression, or to swap the left half of an expression into the right half.

5.6 SPECIAL SYMBOLS

The @ sign sets the indirect bit in the storage word being formed.

Example:

```
MOVE AC,@X
```

CHAPTER 6

PAPER TAPE¹

6.1 PAPER TAPE CONTROL

\$L This command causes DDT to punch a RIM10B loader on paper tape RIM10B loader. (See *Macro-10 manual*, Chapter 6.) Thus, if the user wishes to punch out a program on paper tape he gives a \$L command first in order to get a loader punched on the same tape as the program. Later when the user wishes to read in the program from the paper tape, the hardware READ-IN feature will load the RIM10B loader into the accumulators and then the program will be loaded by the RIM10B loader.

FIRST<LAST (TAPE)² This command punches out checksummed blocks in RIM10B format on paper tape from consecutive locations between FIRST and LAST address inclusively. For example, this command will punch out a program existing in core memory in its present state of check-out for later use.

Example:

4000 < 20000 (TAPE)

FIRST<LAST \$ (TAPE) Similar to the preceding command, except that locations whose contents are zero are not punched out whenever more than two consecutive zeroes are detected.

ADR\$J This command punches a 2-word block that causes a transfer to address ADR after the preceding program has been loaded from paper tape. If ADR is not present, a JRST 4, DDT is punched as the first word.

The following succession of steps will punch a program on paper tape ready to be used as an independent entity.

- a. \$L
- b. 5000 < 20000 (TAPE)²
- c. 6000\$J (Transfer to address 6000 after program is loaded.)

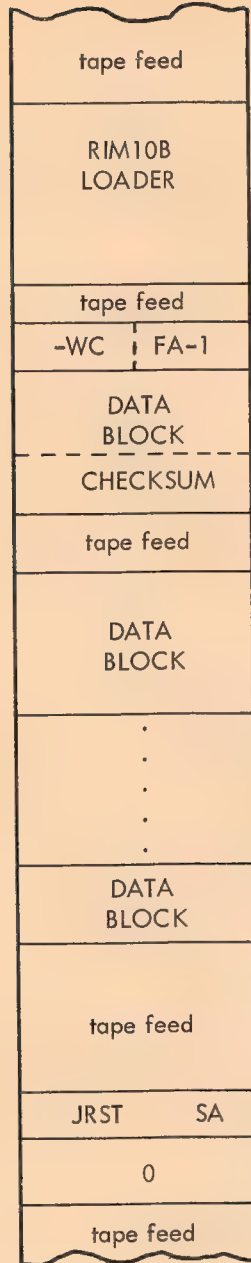
¹The paper tape functions are not available in the time-sharing user mode version of DDT.
²(TAPE) is a single control key on the Teletype, and is identical to ↑R.

Typed in:

\$L

FIRST ADDRESS <
LAST ADDRESS (TAPE)

SA\$J



Beginning of Tape

Checksum includes pointer word
WC = word count

transfer block
SA = starting address

Figure 6-1 RIM10B Block Format

APPENDIX A
SUMMARY OF DDT FUNCTIONS

Type Out Modes

To set the type-out mode to:	Type this	Sample Output(s)
Symbolic instructions	\$S	ADD 4, TAG+1 ADD 4, 4002
Numeric, in current radix	\$C	69. 105
Floating point	\$F	0.125E-3
7-bit ASCII text	\$T	PQRST
SIXBIT text	\$6T	TSRQPC
RADIX50	\$5T	4 DDTEND
Halfwords, two addresses	\$H	4002, 4005 X+1, X+4
Bytes (of n bits each)	\$NO	\$80 COULD YIELD 0, 14, 237, 123, 0

Address Modes

To set the address mode for typeout of symbolic instructions and halfwords (see examples above) to

Relative to symbolic address	\$R	TAG+1
Absolute numeric address	\$A	4005

Radix Change

To change the radix of numeric type-outs to n (for $n \geq 2$), type

\$NR	\$2R COULD YIELD
110101100000010000000000011100101100	

Permanent vs Temporary Modes

To set a temporary type-out or address mode or a temporary radix as shown in the commands above, type

\$	\$C
	\$10R

To instead set a permanent type-out or address mode or a permanent radix, in the commands above, substitute

\$\$	\$\$C
	\$\$10R

To terminate temporary modes and revert to permanent modes, or re-enter DDT, type a carriage return.

↵

Initial permanent (and temporary) modes are

\$\$\$
 \$\$R
 \$\$8R

Examining Storage Words

To open and examine the contents of any address in current type-out mode

adr/

LOC / 254020,,DDTEND

To open a word, but inhibit the type out of contents

adr!

LOC !

To open and examine a word as a number in the current radix

adr[

LOC [254020,,3454

To open and examine a word as a symbolic instruction

adr]

LOC [JRST @DDTEND

To retype the last quantity typed (particularly used after changing the current type-out mode)

\$F; #5.4999646E+11
 \$6T; 5%0 <L

Examining A Related Storage Word

To close the current open word (making any modification typed in) and to open the following related words, examining them in the current type-out mode:

To examine ADR+1

↓ (line feed)

To examine ADR-1

↑ (or backspace, on the Teletype Model 37)

To examine the contents of the location specified by the address of the last quantity typed, and to set the location pointer to this address

→| (TAB)

To examine the contents of address of last quantity typed, but not change the location pointer

\ (backslash)

To close the currently open word, without opening a new word, and revert to permanent type-out modes.

↵ (carriage return).

One-Time Only Typeouts

To repeat the last typeout as a number nber
in the current radix =

To repeat the last typeout as a
symbolic instruction (the address
part is determined by \$A or \$R) +

To type out, in the current type-out
mode, the contents of the location
specified by the address in the open
instruction word, and to open that
location, but not move the location
pointer. /

To type out, as a number, the con-
tents of the location specified by the
open instruction word and to open that
location, but not move the location
pointer. [

To type out, as a symbolic instruction,
the contents of the location specified
by the open instruction word, and to
open that word, but not move the
location pointer.]

Typing In

Current type-out modes do not affect
typing in, instead

To type in a symbolic instruction ADD AC1,@DATE(17)

To type in half words, separate the
left and right halves by two commas. 402,,403

To type in octal values 1234

To type in a fixed-point decimal
integer 99.

To type in a floating-point number 101.11
 77.0E+2

To type in up to five 7-bit PDP-10
ASCII characters, left justified,
delimited by any printing character. "/ABCDE/ (/ is delimiter)

To type in one PDP-10 ASCII character,
right justified "\$ (\$ must be ALT MODE)

To type in up to six SIXBIT characters,
left justified, delimited by any
printing character "\$"ABCDEFGA (A is delimiter)

To type in one SIXBIT character,
right justified "\$"Q\$ (\$ must be ALT MODE)

Symbols

To permit reference to local symbols within a program titled <u>name</u> , type	name\$:	MAIN.\$:
To insert or redefine a symbol in the symbol table and give it the value <u>n</u> , type	n<symbol:	14<TABL3:
To insert or redefine a symbol in the symbol table, and give it a value equal to the location pointer (<u>.</u>), type	symbol:	SYM:
To delete a symbol from the symbol table	symbol\$\$K	LPCT\$\$K
To kill a symbol for typeouts (but still permit it to be used for typing in)	symbol\$K	TBITS\$K
To perform \$K on the last symbol typed out and then to retype the last quantity	\$D	
To declare a symbol whose value is to be defined later	symbol#	JRST AJAX#
To type out a list of all undefined symbols (which were created by #), type	?	

Special DDT Symbols

To represent the address of the location pointer	. (point)
To represent the last quantity typed	\$Q
To represent the indirect address bit	@
To represent the address of the search mask	\$M
To represent the address of the saved flags, etc., (see Appendix D)	\$I
To represent the pointers associated with the nth breakpoint	\$nB

Arithmetic Operators Permitted in Forming Expressions

Two's complement addition	+
Two's complement subtraction	-
Integer multiplication	*
Integer division (remainder discarded)	' (apostrophe)

Field Delimiters In Symbolic Type-Ins

	To delimit op-code name, type one or more spaces.		
	To delimit accumulator field, type		
	To delimit two halfwords, type left,, right		
	To delimit index register	()	
	To indicate indirect addressing	@	
<u>Breakpoints</u>			
	To set a specific breakpoint <u>n</u> ($1 < n < 8$)	adr \$nB	CAR \$\$B
	To set the next unused breakpoint	adr \$B	303 \$B
	To set a breakpoint with automatic proceed	adr \$\$nB adr \$\$B	CAR \$\$B 303 \$\$B
	To set a breakpoint which will automatically open and examine a specified address, <u>x</u>	x,,adr \$nB x,,adr \$B x,,adr \$\$nB x,,adr \$\$B	AC3,,Z+6\$B AC4,,ABLE\$B AC3,,Z+6\$\$B AC4,,ABLE\$\$B
	To remove a specific breakpoint	0\$nB	0\$B
	To remove all breakpoints	\$B	\$B
	To check the status of breakpoint <u>n</u>	\$nB/	
	To proceed from a breakpoint	\$P	\$P
	To set the proceed count and proceed	n\$P	25\$P
	To proceed from a breakpoint and thereafter proceed automatically	\$\$P n\$\$P	\$\$P 25\$\$P

Conditional Breakpoints

	To insert a conditional instruction (INST), or call a conditional routine, when breakpoint <u>n</u> is reached.	\$nB+1/ \$2B+1/ <u>0</u>	INST CAIE 3,100
	If the conditional instruction does not cause a skip, the proceed counter is decremented and checked. If the proceed count ≤ 0 , a break occurs.		
	If the conditional instruction or subroutine causes one skip, a break occurs.		
	If the conditional instruction or subroutine causes two skips, execution of the program proceeds.		

Starting the Program

To start at the starting address in JOBSA	\$G	\$G
To start, or continue, at a specified address	adr \$G	LOC \$G
To execute an instruction	inst \$X	JRST 2, @JOBOPC\$X returns to program after ↑C and DDT commands

Searching

To set a lower limit (a), an upper limit (b), a word to be searched for (c), and search for that word	ac\$W	200 <250 >0\$W
To set limits and search for a not- word	ac\$N	351 <731 >0\$N
To set limits and search for an effective address	ac\$E	401 <471 >LOC+6\$E
To examine the mask used in searches (initially contains all ones)	\$M/	\$M/ -1
To insert another quantity n in the mask	n\$M	777000777777\$M

Instruction Execution

\$U
\$Y

Zeroing Memory

To zero memory, except DDT, locations 20-137, and the symbol table	\$\$Z
To zero memory locations FIRST through LAST inclusive	FIRST<LAST \$\$Z

Special Characters Used in DDT Typeouts

Breakpoint stops	
Break caused by conditional break instruction.	>
Break because proceed counter ≤ 0	>>
Undefined symbol cannot be assembled	U
Half-word type-outs	left,,right 401,,402

Unnormalized floating-point number	#1.234E+27	#1.234E+27
To indicate an integer is decimal. The decimal point is printed	\$10R 77= <u>63.</u>	
Illegal command	?	
If all eight breakpoints have been assigned	?	
RUBOUT echo	XXX	

Paper Tape Commands (Available only in EDDT)

To punch a RIM10B loader	\$L
To punch checksummed data blocks where ADR1 is the first, and ADR2 is the last location of the data	ADR1<ADR2 (TAPE) ((TAPE) is TR)
To punch a one-word block to cause a transfer to adr after the preceding program has been loaded from paper tape	adr\$J

APPENDIX B
EXECUTIVE MODE DEBUGGING (EDDT)

A special version of DDT, called EDDT, is available for debugging programs in the executive mode of the PDP-10. In general, EDDT performs the same debugging functions as user mode DDT. All of the paper tape commands are available in EDDT (those in DDT are marked by an asterisk in Chapter 5). The paper tape I/O routines in EDDT are optional at assembly time.

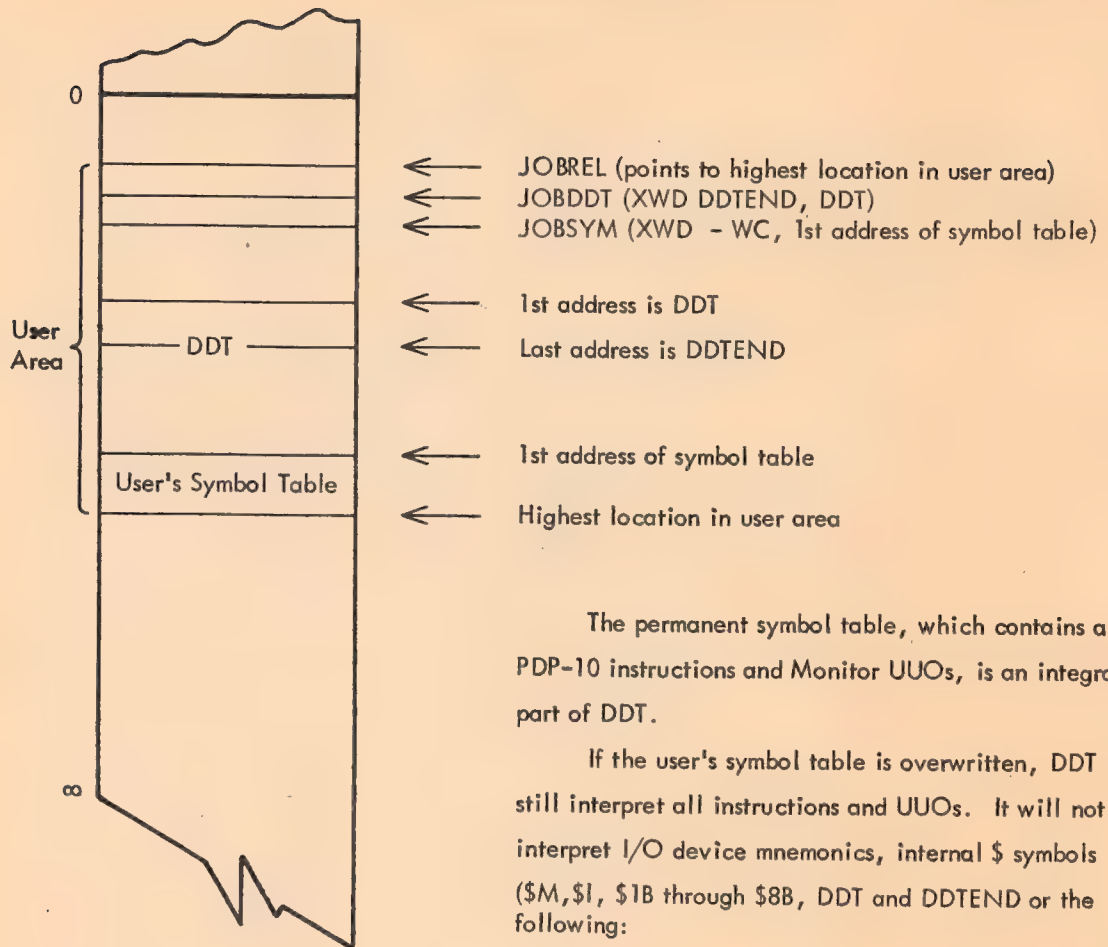
EDDT is used to debug Monitor programs, diagnostic programs, and other executive (or privileged) programs. EDDT performs its own I/O on a Teletype and controls the Priority Interrupt system. It does not check JOBREL for boundary limits as DDT does.

In EDDT the symbol table pointer is in location 36. EDDT does not check location 37, which contains the highest valid address, before address examination. If the NXM Stop switch is ON, the machine will hang up if nonexistent memory is referenced. If this happens, EDDT may be restarted by pressing START, or the CONTINUE switch may be pressed.

The first address of EDDT is DDT; the last is DDTEND.

The \$\$Z command will not zero locations 20 through 37. (In the user mode version, \$\$Z does not zero locations 20 through 137. See Section 4.5.)

APPENDIX C
STORAGE MAP FOR DDT



The permanent symbol table, which contains all PDP-10 instructions and Monitor UUOs, is an integral part of DDT.

If the user's symbol table is overwritten, DDT can still interpret all instructions and UUOs. It will not interpret I/O device mnemonics, internal \$ symbols (\$M,\$I, \$1B through \$8B, DDT and DDTEND or the following:

JOV
JEN
HALT

APPENDIX D
OPERATING ENVIRONMENT

Entering and Leaving DDT

When control is transferred to DDT, the state of the machine is saved inside DDT:

- a. The accumulators are saved.
- b.¹ The status of the priority interrupt system (the result of a CONI PI, \$I) is stored in the right half of register \$I.
- c. The central processor flags are saved in the left half of register \$I.
- d.¹ The PI channels are turned off (by a CONO PI, @\$I+1) if they have a bit in register \$I+1.
- e.¹ The Teletype PI channel is saved in the right half of register \$I+2. The teletype buffer is saved in the left half of \$I+2 but can never be restored. The character in the output buffer will have been typed on the Teletype.
- f. Then using the Monitor command DDT 2, the old PC is saved in the right half of location JOBOPC, with the flags in the left half.

When execution of a program is restarted, the following happens:

- a. The accumulators are restored.
- b.¹ Those PI channels which were on (when DDT was entered) and which have a bit equal to 1 in register \$I+1 are turned on.

$(C(\$I)_R \wedge C(\$I+1)_R) \vee 2000 \rightarrow \text{PI SYSTEM}$
(logical AND (\wedge), logical OR (\vee))

- c.¹ The Teletype PI channel is restored.

$0 \rightarrow \text{TTI DONE} \rightarrow \text{TTI BUSY} \rightarrow \text{TTO BUSY}$

TTO done is set to 1 if either TTO busy or TTO done was on when DDT was entered. Otherwise,
 $0 \rightarrow \text{TTO done}$.

- d. The processor flags are restored from the left half of register \$I.
- e. To return to a program interrupted by ↑ C, the user types:
JRST 2, @ JOBOPC\$X TO RESTORE THE PC AND FLAGS.

¹ Functions not available in the time-sharing user mode.

Loading and Saving DDT

How to load and save DDT.SAV (PDP-10) or DDT.DMP (PDP-6) in 2K of core:

<u>Instructions</u>	<u>Example</u>
Load DDT in 4K of core.	<u>.R</u> LOADER 4 DTA1:DDT,/140G (ALTMODE) LOADER <u>EXIT</u> <u>↑C</u>
Enter DDT.	<u>.ST</u>
Type out, in halfword mode, the contents of JOBSYM.	\$\$H JOBSYM/ <u>-162,,7616</u>
Open register 6 and put (JOBSYM) _{RH} into the left half of 6; put (JOBSYM) _{RH} - 4000 ₈ into the right of 6.	6! 7616,,3616
Perform a block transfer until you reach address 3777 ₈ .	BLT 6,3777\$X
Open up JOBSYM. Leave the left half as is, and change the right half to 4000 ₈ less than it was.	JOBSYM! -162,,3616
Zero memory except DDT.	\$\$Z
Open up JOBSA and check that left half = DDTEND; if not, change left half to DDTEND.	JOBSA/ 0,,DDT DDTEND,,DDT
Change back to symbol type-out mode.	\$\$S
Return to the Monitor.	↑C
Reduce core to 2K.	<u>.CORE</u> 2
Reenter DDT.	<u>.ST</u>
CHECK JOBREL.	JOBREL/ <u>3777</u>
Return to the Monitor.	↑C
Save DDT.	<u>.SAVE</u> DTA1 DDT

Explanation - The DDT saved file must be saved in 2K (minimum amount of core needed for it). Also, a starting address must be set up for DDT as location 140. To get DDT into 2K, the DDT symbol table must be moved down to the upper end of the first 2K of core. Any unused locations in DDT should be set to zero (\$\$Z) and JOBSYM should be set to the new location of the start of the DDT symbol table. Before saving the resulting file, a CORE 2 request should be given to the Monitor to ensure that DDT is saved as a 2K core image.

Book 6

Utility Programs

Peripheral Interchange Program
(PIP)

Source Compare
(SRCCOM)

Binary Compare
(BINCOM)

DECtape Utility Program
(TENDMP)

File Update Generator
(FUDGE 2)

Cross-Reference Listing
(CREF)

Global Symbol Cross Reference List
(GLOB)

PERIPHERAL INTERCHANGE PROGRAM (PIP)

PIP transfers data files from any standard I/O device to any other standard I/O device and, additionally, performs simple editing and magnetic tape control functions. PIP1, a compact version of PIP, performs a subset of PIP functions. PIP handles all data formats, and eliminates the need for a satellite computer to handle off-line data conversions.

Requirements

PIP	Minimum Core: 3K	Additional Core: 1K if disk is one of the I/O devices; any core above that required is used for extra I/O buffers.
	Equipment Handled:	DECtape, disk, magnetic tape, paper tape reader, paper tape punch, card reader, line printer, and teletype.
PIP1	Minimum Core: 1K	Additional Core: Any core greater than 1K is used for extra input buffers.
	Equipment Handled:	DECtape, disk, magnetic tape, paper tape reader, paper tape punch, card reader, line printer, and teletype.

Initialization

_R PIP) or _R PIP1)

*

Loads PIP (or PIP1) into core.

PIP is ready to receive a command; an asterisk is typed after each requested action has been completed.

Commands

General Command Format

destination-dev:filename.ext ←source-dev:filename.ext,...source-n)

destination-dev:

source-dev:

The destination device, to which the data is to be transferred; the source device(s), from which the data is to be read

NOTE

If logical device SYS (the CUSP device) is a DECtape, it must not be modified using the /R or /D switches or any other request requiring it to be initialized for input and output at the same time.

DTAn:	(DECtape)
PTR:	(paper tape reader)
PTP:	(paper tape punch)
DSK:	(disk)
CDR:	(card reader)
MTAn:	(magnetic tape)
LPT:	(line printer)
TTY: or	(Teletype)
TTYn:	

If more than one file is to be transferred from a magnetic tape, card reader, teletype, or paper tape reader, dev: is followed by a comma for each file after the first; these devices can also be followed by * or *.* to indicate all files are to be transferred.

filename.ext (DSK: and DTAn:only)

The filename and filename extension to be assigned to the file on the destination device; the filename and filename extension of the file(s) to be read from the source device.

An asterisk can be used for source files as follows.

filename.*	- Transfer all files having the specified filename.
*.ext	- Transfer all files having the specified extension.
.	- Transfer all files.
*	- Transfer all files with null extensions.

The destination descriptors and the source descriptors are separated by the left arrow symbol (←).

Disk File Descriptor Format

DSK:filename .ext [proj,prog] <protection >

[proj,prog]

Project-programmer number assigned to the disk area to be used, if other than the user's project-programmer number.

<protection >

Protection value to be assigned to the destination file. If omitted, the standard protection is assigned.

NOTE

Standard protection (055) designates that the owner is permitted to read or write, or change the protection of, the file while others are permitted only to read the file.

Standard Assumptions

Unless otherwise changed by switches, all files which are on directory devices and which have a file-name extension of .REL, .SAV, .DMP, or .CHN are copied in binary; all other files are assumed to be in ASCII line mode. Magnetic tape files, unless otherwise changed by switches, are read in odd parity and written in odd parity at 556 bpi.

Examples

<u>Command</u>	<u>Function</u>
␣R PIP ␣	Loads and starts PIP.
*LPT:←DTA1:FILE1 ␣	Transfer the file named FILE1 from DTA1 to the line printer.
LPT:←DTA1: ␣	Transfer all files with null extensions from DTA1 to the line printer.
*DTA2:FILE2←DTA1:FILE1.TMP ␣	Transfer the file named FILE1.TMP to DTA2 and give it the name FILE2.

Command	Function
<u>*DTA2:FILE3←DTA1:FILE1,FILE2</u>)	Transfer the files named FILE1 and FILE2 from DTA1 to DTA2, combining them as one file under the name FILE3.
<u>*DTA2:FILE3←DTA1:FILE1,DTA3:FILE2</u>)	Transfer the file named FILE1 from DTA1 and the file named FILE2 from DTA3 to DTA2, combining them as one file under the name FILE3.
<u>*DSK:FILE1←MTA1:</u>)	Transfer the next file from the present position of MTA1 to the user's area on the disk, call it FILE1, and assign the standard protection of 055.
<u>*DSK:FILE1<177>←MTA1:*</u>)	Transfer all files from MTA1 (starting at the current position of the read head) to the user's area on the disk, combining them into one file called FILE1, and assign protection 177.
<u>*DSK:FILE1[1,3]←MTA1:,,</u>)	Transfer the next two files from the present position of MTA1 to area 1,3 on the disk, combining them into one file called FILE1, and assign the standard protection (055).
<u>*PTP:←PTR:,,,,</u>)	Transfer five files from the paper tape reader, combining them as one file on the paper tape punch.
<u>*+C</u>)	Return to Monitor.
<u>.</u>	Dot indicates that user is at Monitor level.

Switches

Nonmagnetic-tape switches, when used, are preceded by a slash (if more than one is specified, they may be enclosed by parentheses instead) and can appear anywhere in the command string; however, if the command string contains commas, the switches must be specified prior to the first comma.

Magnetic tape switches are enclosed by parentheses and must appear immediately following the device or file to which they refer.

Switches are used to specify:

- a. Particular files for transferral or deletion;
- b. Editing;
- c. Mode of transfer;
- d. Directory manipulation (DECtape and DSK); and
- e. Magnetic tape control.

A listing of PIP switches can be obtained by typing

```
*output-dev:/Q+ )
```

where output-dev: may be either LPT: or TTY:

PIP Switch Options

Switch	Meaning		
A	Line blocking		
B	Process file in Binary mode.		
C	Suppress trailing spaces and Convert multiple spaces to tabs.		
D	Delete the file.		
E	Treat Ending (card) columns 73 through 80 as spaces.		
F	List the directory in short form for DSK: or DTAn: only. (Filenames and extensions only.) Useful when disk directory cannot be listed with /L sw.		
G	Ignore I/O errors.		
H	Process file in image binary mode.		
I	Process file in Image mode.		
L	List the directory (DSK: or DTAn: only)		
M	Magnetic tape switches. A string of one or more magnetic tape switches begins with an M and is enclosed in parentheses.		
#nA	Advance the tape n files.	E	Even parity.
#nB	Backspace the tape n files.	F	Mark End of File.
#nD	Advance the tape n records.	T	Skip to logical end of Tape.
#nP	Backspace the tape n records.		
2	200 bpi density.		
5	556 bpi density.		
8	800 bpi density.	U	Rewind tape and Unload.
A	Advance tape one file.	W	Rewind the tape.
B	Backspace tape one file.		

NOTE

MTA switches always apply to the device or file immediately preceding the switches. MTA switches should be used only in specific situations. For a more detailed treatment, see PIP Programmer's Reference Manual.

Examples

.R PIP)

DSK:/X←DTA1:.*)

DSK:(DX)←DTA1:FILE1,.REL)

*MTA2:/S←CDR:)

*LPT:/P←DTA1,FILE1)

*DTA2:FILE1/I←PTR:)

*TTY:/L←DTA1:)

[nnnn FREE BLOCKS LEFT)
filename.ext no. blocks creation date

⋮

*DTA1:/Z←)

*MTA2:(M8E)←MTA1:(ME8))

*MTA2:(MW)←)

*LPT:←MTA1:(M2W),(MA),,,)

*MTA1:(M#4A)←CDR:)

*↑C)

⋮

Load and run PIP.

Transfer all files from DTA1 to DSK, keeping them separate and retaining their filenames.

Transfer all files, except FILE1 and any files with the extension .REL, from DTA1 to DSK, keeping them separate and retaining their filenames.

Transfer a file from the card reader to MTA2 and add sequence numbers.

Take FILE1 (a FORTRAN output print file), interpret the carriage control characters, and print the file using specified carriage control.

Initialize both DTA2 and the paper tape reader in image mode and transfer one file from the paper tape reader to DTA2, calling it FILE1.

List the directory of DTA1 on the teletype.

Zero the directory of DTA1.

Transfer a file from MTA1 to MTA2 in 800 bpi, even parity mode.

Rewind MTA2.

Set MTA1 to 200 bpi, odd parity, rewind the tape, and transfer the first, third, fourth, and fifth files to the printer.

Advance MTA1 four files before transferring a file from the card reader.

Return to the Monitor.

Diagnostic Messages

PIP Diagnostic Messages

Message	Type	Meaning
?4K NEEDED	S	4K is not currently available but is needed when a disk is present in the system.
?DECTAPE I/O ONLY	S	I/O device for copy block 0.(/U) must be a DECTape.
?DEVICE dev:DOES NOT EXIST	I/O	Either device name has been misspelled or there is no such device.
?DEVICE dev:NOT AVAILABLE	I/O	The device has been assigned to another job.
?DIRECTORY FULL	FR	There is no room for an entry in a DECTape directory.
DISK DIRECTORY READ	I/O	This message is nonfatal if the /G switch is used; otherwise, it is fatal and is preceded by a ?. A second message follows (see Table 6-3).
?DISK OR DECTAPE INPUT REQUIRED	I/O	This command requires a directory device for input.
?DTA TO PTP ONLY	RIM	DTA input and PTP output must be specified for /Y.
FAILURE DURING (/X,/Z,/D,/R) REQUEST	S	Each file requested does exist, but one or more was unavailable for processing. This message is never fatal.
?FILE filename.ext ILLEGAL EXTENSION	RIM	Extension for /Y request must be .RMT, .RTB, or .SAV.
?FILE filename.ext ILLEGAL FORMAT	RIM	1. Zero-length file; or 2. Requisite job data info not available; or 3. Block overlaps previous block (RIM 10) or 4. EOF found when data was expected, or 5. A pointer word was expected but not found in the source file.
?filename.ext (3) FILE WAS BEING MODIFIED	FR	Disk file named is currently being processed by another job.
?filename.ext (0) FILE WAS NOT FOUND	FR	Filename.ext not found during LOOKUP.

PIP Diagnostic Messages

Message	Type	Meaning
?filename.ext (0) ILLEGAL FILE NAME	FR	Indicates that 1. No filename was specified for DTA output file; or 2. A reject occurred on a /R request for disk file; or 3. Illegal filename was specified for a /R request on DTA.
?filename.ext (1) NO SUCH PROJECT-PROGRAMMER NUMBER	FR	The project-programmer number specified for a DSK file is incorrect.
INPUT DEVICE dev: FILE filename.ext	I/O	This message is nonfatal if the /G switch is used; otherwise, it is fatal and is preceded by a ?. A second message follows (see Table 6-3).
?LINE TOO LONG	S	A line >140 characters was detected in the source file.
?LOAD POINT BEFORE END OF (MB) OR (MP) REQUEST	S	Load point on a magnetic tape file has been reached before the tape has been backspaced the number of files or records specified in (M#nB), (M#nP).
?NO BLOCK 0 COPY	C	/U given but PIP assembled without provision for this.
?NO FILE NAMED filename.ext	FR	No such file found during PIP directory search.
?NO FILE NAMED QPIP	S	The data file for the /Q switch is not available.
OUTPUT DEVICE dev: FILE filename.ext	I/O	Followed by a second message (see Table 6-3).
?PIP COMMAND ERROR	C	1. Illegal format for command string; or 2. Nonexistent switch requested; or 3. Filename.ext other than * (or *.*) requested for a non-directory device; or 4. The illegal switch combination RX.
?filename.ext (2) PROTECTION FAILURE	FR	Same as FAILURE DURING ... message except that the processing halts.
?filename.ext (4) RENAME FILENAME ALREADY EXISTS	FR	Tried to rename file with already existing name.
?filename.ext (5) RENAME ERROR	FR	LOOKUP or ENTER not done.
?filename.ext (6)	I/O	Error not yet defined.
?filename.ext (7)	I/O	Error not yet defined.

PIP Diagnostic Messages

Message	Type	Meaning
?TERMINATE /X MAX of 999 FILES PROCESSED	S	The /X switch specified for nondirectory device source files has processed the maximum number of files (999).
?TOO MANY REQUESTS FOR dev:	C	Conflicting parity/density requests have been given for a magnetic tape.
?TRY PIP		During a PIP1 run, a switch or function which is not present in PIP1 has been requested.
NOTES		
<p>All fatal diagnostic messages are preceded by a question mark (?).</p> <p>Message types are:</p> <p>C Command string error FR File reference error I/O I/O error RIM Readin Mode specification error S Other types of errors.</p>		

Table 6-3
Secondary PIP I/O Diagnostic Messages

Message	Device	Meaning
BINARY DATA INCOMPLETE	PTR	Length of block disagrees with word count (nonfatal if the /G switch has been specified).
BLOCK TOO LARGE	DTA	DTA link number >1101 ₈ .
CHECKSUM OR PARITY ERROR	All	Read or write error (nonfatal if the /G switch has been specified).
INPUT BUFFER OVERFLOW	All except DTA	Block too large for buffer (nonfatal if the /G switch has been specified).
DEVICE ERROR	All	The data control unit has detected the loss of data (nonfatal if the /G switch has been specified).

Secondary PIP I/O Diagnostic Messages

Message	Device	Meaning
PHYSICAL EOT	MTA	The end of tape has been reached (nonfatal if the /G switch has been specified).
WRITE (LOCK) ERROR	DTA,DSK,MTA	Attempt has been made to write on a write-locked file.
7-9 PUNCH MISSING	CDR	Binary card lacks 7-9 punch (nonfatal if the /G switch has been specified).

1K Version of PIP (PIP1) Limitations

The following limitations apply to PIP1:

- a. Z and MW requests ignore all source devices.
- b. B switch included since REL, SAV, DMP, and CHN files are not automatically copied in 36-bit bytes.
- c. Error messages assume all I/O devices are DECtape.
- d. Neither project-programmer numbers nor protection can be specified for disk files.
- e. The * cannot be used for filenames or extensions.
- f. SAV files cannot be successfully copied with PIP1.

Monitor Commands

The following Monitor commands perform PIP-type operations.

<u>Desired Result</u>	<u>Monitor Command</u>	<u>Equivalent CUSP Commands</u>
To type the contents of a file on the TTY.	_TYPE dev:filename.ext)	_R PIP) *TTY: +dev:filename.ext)
To list the contents of a file on the line printer.	_LIST dev:filename.ext)	_R PIP) *LPT: +dev:filename.ext)
To type the directory of a device on the TTY.	_DIRECT dev:)	_R PIP) *TTY: +dev:/L)
To list the directory of a device on the line printer.	_DIRECT dev:/L)	_R PIP) *LPT: +dev:/L)

<u>Desired Result</u>	<u>Monitor Command</u>	<u>Equivalent CUSP Commands</u>
To delete a file.	<u>_DELETE</u> dev:filename.ext	<u>_R PIP</u>) *dev:filename.ext/D+)
To rename a file.	<u>_RENAME</u> dev:newfn =oldfn	<u>_R PIP</u>) *dev:newfn/R+oldfn)

NOTE

If dev: is omitted in the Monitor commands, DSK: is assumed.

FILE UPDATE GENERATOR (FUDGE2)

FUDGE2 updates files containing one or more relocatable binary programs, and permits the user to manipulate individual programs within program files.

Requirements

Minimum Core: 2K

Additional Core: Dynamically allocates its buffers to utilize as much core as is made available.

Equipment: Two input devices, one for the master file and one for the transaction file; one output device for the updated file. The input device(s) and output device can be the same device (DSK:). The two input devices can be the same DECTape.

Initialization

._R FUDGE2)

*

-

Loads the File Update Generator program.

FUDGE2 is ready to receive a command.

<programe,.....> (DSK: and
DTAn: only)

The filename.ext of the program file containing the programs to be used in performing additions or replacements to the master file.

If no .ext is given, .REL is assumed.

Program names must be specified in the same relative order in which they appear in the file.

Program names are grouped together within angle brackets <> and are separated by commas.

If it is desired to append, replace, insert, or extract all programs within a file, only the filename.ext need be specified.

Program names cannot be specified for the output file.

The new output file is separated from the master and transaction files by the left arrow symbol (←).

Command Codes

The function to be performed by FUDGE2 is selected by including one of the following command codes at the end of the command string. Command codes are enclosed within parentheses (or preceded by a slash) and one (and only one) must appear in every command string.

FUDGE2 Command Codes

Command	Meaning
A	Append one or more programs from the transaction file(s) to the master file and write out the new file. The command string is as follows: new-file ← master-file,transaction-file,.....(A) \$
D	Delete one or more programs from the master file and write out the new file. The files (and programs) to be deleted are listed after master-dev:. The command string is as follows: new-file ← master-file<file(s) to be deleted>(D) \$
E	Extract the specified files (and programs) from one or more input files and create a new output file. If program names are not specified for a file, the entire file is extracted. The command string is as follows: new-file ← masterfile<file(s) to be extracted>(E) \$

FUDGE2 Command Codes

Command	Meaning
I	<p>Insert programs from one or more transaction files onto the master file and write out the new file. The programs from the transaction file(s) are inserted immediately before the specified programs on the master file. The command string is as follows:</p> <p>new-file ← master-file<file(s) to be inserted before>, transaction-file(s) (I) \$</p>
L	<p>List all relocatable programs within a file and print the listing on the output device, which must be either TTY: or LPT: The command string is as follows:</p> <p>listing-device ← file(L) \$</p>
R	<p>Replace the named program(s) on the master file with the named program(s) from the transaction file, and write out the new file. The command string is as follows:</p> <p>new-file ← master-file<file(s) to be replaced>, transaction-file<replacement file(s)>(R) \$</p>
<p>NOTE</p> <p>Only one operation can be specified per command string. Thus, to delete a file and replace some other one, two command strings are required.</p>	

Examples

<pre><u>R</u> FUDGE2) <u>*LPT</u>:←DTA1:LIB40(L) \$)</pre>	<p>List all relocatable programs (.REL) from the file LIB40, located on DTA1 on the line printer.</p>
<pre><u>*DTA2</u>:LIB4AA ←DTA1:LIB40<EXP.2>(D)\$)</pre>	<p>Delete the program EXP.2 from the file LIB40 on DTA1; write the new file on DTA2 and call it LIB4AA.REL.</p>
<pre><u>*DSK</u>:LIB4BB←D7A2:LIB4AA <EXP.3,EXP.3C>,) D7A1:F1 <EXP.3A,EXP.3B>(R) \$)</pre>	<p>Replace programs EXP.3 and EXP.3C located in file LIB4AA on D7A2, with programs EXP.3A and EXP.3B in File F4 on D7A9; write out the new LIB4AA file on disk and call it LIB4BB.</p>
<pre><u>*PTP</u>:←DSK:LIB4BB,DTA4:SCIENC<COSRTE>/A \$)</pre>	<p>Append the program COSRTE, located in file SCIENC on DTA4, to the file LIB4BB on disk; write out the updated LIB4BB file on the paper tape punch.</p>

```
*DTA1:NFILE←DSK:MFILE<M1,M2,M3,M4> )
```

```
DTA3:TFILEA<TA1,TA2> )
DTA4:TFILEB<TB1,TB2>/I $ )
```

```
*DTA1:NFILE←DSK:MFILE<M1,M2,M3,M4> )
DTA3:TFILEA )
DTA4:TFILEB/I $ )
```

```
*+C )
```

Insert into MFILE the programs TA1 and TA2 from TFILEA and TB1 and TB2 from TFILEB. Create NFILE with the following order:
TA1,M1,TA2,M2,TB1,M3,TB2,M4

Insertion is on a one-to-one basis. If there are more programs to be inserted than specified programs before which they are to be inserted, the extra files are ignored.

However, in this example (where TFILEA and TFILEB contain the programs TA1 and TA2 and TB1 and TB2, respectively) create an NFILE with the following order:
TA1,TA2,M1,TB1,TB2,M2,M3,M4

Return to the Monitor.

Switches

Switches are used to manipulate file directories and to position magnetic tape. They are either preceded by a slash or enclosed in parentheses and can appear anywhere in the command string.

FUDGE2 Switch Options

Switch	Meaning
B	Backspace magnetic tape one file.
K	Advance magnetic tape one file.
W	Rewind magnetic tape.
Z	Clear directory of destination device (DTAn: only).

Examples

```
*R FUDGE2 )
*DTA2:TESTA←MTA1:(WK),MTA2: :(ZA) $ )
```

Clear the directory of DTA2; rewind MTA1 and advance the tape one file; append the first two program files from MTA2 to the second file on MTA1 and write out the resultant file on disk, calling it TESTA.

```
*+C )
```

Return to the Monitor.

Diagnostic Messages

FUDGE2 Diagnostic Messages

Message	Meaning
?CANNOT DO I/O AS REQUESTED	Input cannot be performed on one of the devices specified for input (it is an output only device) or output cannot be performed on the device specified for output.
?DEVICE ERROR ON OUTPUT DEVICE	A write error has occurred on the output file.
?DIRECTORY FULL ON OUTPUT DEVICE	No more files can be added to the file directory on the output device (the directory is full).
?ENTRY BLOCK TOO LARGE, PROGRAM xxxxxx	The entry block of program xxxxxx is too large for the FUDGE2 entry table, which allows for 32 entry names. FUDGE2 can be reassembled with a larger table.
?FUDGE SYNTAX ERROR	The command string is illegal (e.g., the left arrow was omitted, a program name was specified for the output file, or some meaningless command was entered).
?x IS AN ILLEGAL CHARACTER	An illegal character has been encountered in the command string.
?x IS AN ILLEGAL SWITCH	An illegal or otherwise meaningless switch has been encountered in the command string.
?dev NOT AVAILABLE	The device either does not exist or has been assigned to another job.
?NOT ENOUGH ARGUMENTS	An insufficient number of files of one type or another has been specified.
?dev filename.ext progname NOT FOUND	Either the filename.ext or the program name was not found on the device (or in the file) specified. If a program name is printed, this may indicate that the program names in the command string appear in a sequence different from their sequence within the file; thus, the program may actually exist in the named file but was missed because of the incorrectly entered sequence in the command string.
?PROGRAM ERROR WHILE RESETTNG MASTER DEVICE	Either FUDGE2 cannot find the master device or cannot find the program name on the master device.

FUDGE2 Diagnostic Messages

Message	Meaning
?TOO MANY FILE NAMES OR PROGRAM NAMES	More than 40 program names or file names were given in a command string. Break the job into several segments and rerun.
?TRANSMISSION ERROR ON INPUT DEVICE dev	A transmission error has occurred while reading data from device dev.
?UNEQUAL NUMBER OF MASTER AND TRANSACTION PROGRAMS	An unequal number of master and transaction programs (or files) has been specified with a Replace request.

CROSS-REFERENCE LISTING (CREF) (VERSION CREF.V32 AND LATER)

CREF produces a sequence-numbered assembly listing followed by one to three tables, one showing cross references for all operand-type symbols (labels, assignments, etc.), another showing cross references for all user defined operators (macro calls, OPDEFs etc.), and another (if the proper switch is specified) showing the cross references for all op codes and pseudo-op codes (MOVE, XALL, etc.). A number sign (#) appears on the definition line of all symbols. The input to CREF is a modified assembly listing file created during a Macro-10 assembly or FORTRAN IV compilation when the /C switch is specified in the command string.

CREF provides an invaluable aid for program debugging and modification.

Requirements

Minimum Core:	2K
Additional Core:	Takes advantage of any additional core available, as necessary.
Equipment:	One input device (normally disk) which contains the modified assembly listing file; one output device (normally the line printer) for the listing.

Initialization

<code>.R CREF</code>	Loads the Cross-Reference Listing program into core.
<code>*</code>	The program is ready to receive a command.

Commands

General Command Format

<code>output-dev: +input-dev:filename.ext</code>	
<code>output-dev:</code>	The device on which the assembly listing and cross-reference tables are to be printed (LPT: is assumed if device is not specified).
<code>input-dev:</code>	The device on which the modified assembly listing was written during Macro-10 assembly (DSK: is assumed if device is not specified).
<code>filename.ext (DSK: or DTAn:only)</code>	The filename and filename extension of the modified assembly listing file (CREF.LST is assumed if filename.ext is not specified).
<code>+</code>	The output device and the input device are separated by the left arrow symbol.

Disk File Command Format

<code>DSK:filename.ext [proj,prog]</code>	
<code>[proj,prog]</code>	Project-programmer number assigned to the disk area to be searched for the source file if other than the user's project-programmer number.

Examples

<pre> .R MACRO) *PTP: ,/C+DTA1:TXCALC) THERE ARE NO ERRORS) PROGRAM BREAK IS 003771) 7K CORE USED) *+C) .R CREF) *) *+C) . </pre>	<p>Loads the Macro-10 Assembler into core.</p> <p>Assembles the program TXCALC from DTA1; writes the object program coding on the paper tape punch; writes a modified assembly listing on DSK: (assumed) and assigns it the filename CREF.LST.</p> <p>Return to the Monitor.</p> <p>Loads CREF into core.</p> <p>Selects the default assumptions of:</p> <table border="0" style="margin-left: 20px;"> <tr> <td>output-dev:</td> <td>LPT:</td> </tr> <tr> <td>input-dev:</td> <td>DSK:</td> </tr> <tr> <td>filename.ext</td> <td>CREF.LST</td> </tr> </table> <p>Return to the Monitor.</p>	output-dev:	LPT:	input-dev:	DSK:	filename.ext	CREF.LST
output-dev:	LPT:						
input-dev:	DSK:						
filename.ext	CREF.LST						

Switches

Switches are used to specify such options as magnetic tape control and list selection. All switches are preceded by a slash (/).

CREF Switch Options

Switch	Meaning
A	Advance magnetic tape reel by one file. /A may be repeated.
B	Backspace magnetic tape reel by one file. /B may be repeated.
K	Kill listing of references to basic symbols (labels, assignments, etc.).
M	Suppress listing of references to user-defined operators (Macro calls, OPDEFs, etc.).
O	Allow listing of references to machine and pseudo-operation codes (MOVE, XALL, etc.).
R	Requests (by typing out RESTART LISTING AT LINE:) the line number at which the listing is to Restart. (Such action might be necessary if the line printer ran out of paper, or jammed, etc.) The user types the line number followed by a carriage return.
S	Suppress program listing (list only the selected tables).

CREF Switch Options

Switch	Meaning
T	Skip to logical end of magnetic Tape.
W	ReWind magnetic tape.
Z	Zero the DECTape directory (DECTape must be output only).

Examples

```
.R CREF )
```

Loads CREF into core.

```
*/M←MTA1:/W )
```

Rewind MTA1 and process the first file, listing only the cross references for operand-type symbols (labels, assignments, etc.).

```
*DTA5:SAVE1/Z← )
```

Process the file named CREF.LST in the user's area of disk; write the program listing and operand-type cross references on DTA5 and call the file SAVE1.

```
*↑C )
```

Return to Monitor

```
.
```

Diagnostic Messages

CREF Diagnostic Messages

Message	Meaning
?dev NOT AVAILABLE	Device is assigned to another job.
?CANNOT ENTER FILE fnme.ext	DTA or DSK directory is full; file cannot be entered.
?CANNOT FIND FILE fnme.ext	The file cannot be found on the device specified.
?COMMAND ERROR	Error in last command string entered.
?DATA ERROR DEVICE dev:	READ or WRITE error.
?ERROR READING COMMAND FILE	Disk data error while reading nnnCRE.TMP (see below).
?IMPROPER INPUT DATA	Input data not in CREF format.
?INPUT ERROR ON DEVOCE dev:	READ error has occurred on the device.
?INSUFFICIENT CORE	Additional core is required for execution but none is available from Monitor.

Monitor Commands

CREF-format listing files generated by COMPILE, LOAD, EXECUTE, and DEBUG commands (using the /CREF switch) can be printed on the line printer by typing

```
._CREF >
```

The CREF command will print out all listing files that are specified in the CCL command file, nnn CRE.TMP (where nnn is the user's job number). After completion of this operation, nnn CRE.TMP is deleted to preclude the listing files being listed again by the next CREF command.

GLOBAL SYMBOL CROSS REFERENCE LIST (GLOB) VERSION #002 OR LATER

GLOB reads multiple binary program files produced by Macro and F40 and generates an alphabetic cross-referenced list of all global symbols encountered.

Requirements

Minimum Core: 2K
 Additional Core: Requests additional core from the Monitor as required.
 Equipment: An input device for each binary file to be scanned for global symbols and one or more listing devices for output.

Initialization

<code>._R GLOB)</code>	Loads the Global Cross-Reference Listing program.
<code>*</code>	The program is ready to receive a command.
<code>-</code>	

Commands

Input Command

`dev:filename.ext,...filename.ext,dev:filename.ext,...filename.ext,...)`

<code>dev:</code>	The device(s) containing the binary program files to be scanned.
	MTAn: (magnetic tape)
	DTAn: (DECtape)
	DSK: (disk)
	PTR: (paper tape reader)

filename.ext (DSK: and DTAn: only)

The filename and filename extension of each binary program which resides on either disk or DECtape.

Output Command

dev: ←\$

dev:

The device on which the global symbol listing is to be printed.

LPT: (line printer)

TTY: (Teletype)

Other output devices can be specified if desired.

More than one output command can be given if it is desired to produce several types of listings on several different devices. Each new output command is typed after the previous request has been completed.

Examples

.R GLOB)

*DSK:F1,F2,DTA3:CALC1,CALC5)

The binary program files to be scanned are F1 and F2 on DSK, and CALC1 and CALC5 on DTA3.

*LPT: ← \$)

All global symbols in these programs are to be listed on the printer. Printed with each symbol are its value, the name of the program in which it was defined, and the names of all the programs in which it was referenced (i.e., declared external).

*↑C)

Return to the Monitor.

.

Switches

The switches available in GLOB are used to determine the types of global symbols to be listed on each of the specified output devices. If no switches are typed, all global symbols are listed. There are also three separate switches (L, M, and X) which act independently.

All switches are either preceded by a slash or enclosed in parentheses and can appear anywhere in the output command string. However, only the most recently specified switch (except L, M, or X, which always take effect) is in effect at any given time.

GLOB Switch Options

Switch	Meaning
A	All global symbols are to be listed (assume if no switch is given).
E	List erroneous (multiply defined or undefined) symbols only.
F	List fixed (nonrelocatable) symbols only.
L	Turn on Library Search Mode (that is, only scan programs if they contain globals previously defined and not yet satisfied).
M	Turn off Library Search Mode.
N	List only those symbols which are never referred to.
R	List relocatable symbols only.
S	List multiply specified (i.e., symbols defined in more than one program, but with non-conflicting values) only.
X	Omit printing of listing title when output is other than TTY. Include printing of listing title when output is TTY.

NOTE

Normally, the title is printed on all devices except the Teletype.

Examples

```

.R GLOB
*DTA1:TEST1.REL,SUBRTE,DSK:ARITH1,
*SCIENC,RETEST

```

The binary programs to be scanned are files TEST1.REL and SUBRTE on DTA1, and ARITH1, SCIENC, and RETEST on disk.

```

*LPT:+/R $

```

List only relocatable symbols on the printer.

```

*TTY:+/E $

```

Printer listing is completed. Enter command to print all erroneous symbols on the Teletype.

```

U EXTSYM      SUBRTE

```

(U = Undefined; EXTSYM is the undefined symbol; SUBRTE is the program in which EXTSYM appears.)

```

*↑C

```

Return to the Monitor.

Diagnostic Messages

GLOB Diagnostic Messages

Message	Meaning
?COMMAND SYNTAX ERROR	An illegal command string has been entered.
?DESTINATION DEVICE ERROR	An I/O error has occurred on the output device.
?DIRECTORY FULL	No more files can be added to the directory of the output device.
?dev NOT AVAILABLE	The device either does not exist or has been assigned to another job.
?filename.ext NOT FOUND	The filename.ext cannot be found in the directory on the device specified.
?TABLE OVERFLOW - CORE UUO FAILED TRYING TO EXPAND TO xxx	GLOB requested additional core from the Monitor, but none was available.

GLOB Error Flags

Flag	Meaning
M	Multiply defined symbol (all values are shown).
N	Never referred to (i.e., was not declared external in any of the binary programs).
S	Multiply specified symbol (i.e., defined in more than one program, but with non-conflicting values). In the listing, the name of the first program in which the symbol was found is followed by a plus sign.
U	Undefined symbol.

SOURCE COMPARE (SRCCOM) (VERSION 013)

SRCCOM compares, line by line, two versions of a source file coded as lines of ASCII characters and outputs any differences.

Requirements

Minimum Core: 2K

Additional Core: The minimum core allows for comparing files with minimal differences. SRCCOM automatically requests more core from the Monitor when it needs it. Major differences can usually be handled in 3K, but for comparing two completely different files, enough core is required to store all of both files simultaneously.

Equipment: User teletype for control; two input devices for the two files to be compared; one output device for listing the differences.

Initialization

```

_R SRCCOM
*

```

Loads the Source Compare routine.

Source Compare is ready to receive a command.

Commands

General Command Format

`list-dev:filename.ext ←input 1-dev:filename.ext , input2-dev:filename.ext`

list-dev: The device on which the differences are to be listed.

LPT:	(line printer)	(Any device
TTY:	(teletype)	that can output
MTAn:	(magnetic tape)	ASCII characters)
DTAn:	(DECtape)	
DSK:	(disk)	

input -dev: The devices on which the two source files to be compared are located.

MTAn:	(magnetic tape)	(Any device
DTAn:	(DECtape)	that can input
DSK:	(disk)	ASCII characters)
PTR:	(paper tape reader)	

`filename.ext` (DSK: and DTAn: only)

The filename and extension of either of the input source files.

The filename and extension to be assigned to the output list file. (SRCCOM.LST is assumed if no filename is specified.)

The output device is separated from the input source file devices by the left arrow symbol.

Default Conditions

TTY: is assumed as the output device if no other device is specified.

DSK: is assumed as both input devices if no other devices are specified.

A dot is necessary in filename #2 to explicitly indicate a null extension if the extension for filename #1 is not null.

Example:

LPT: ←DRAn:FORSE.MAC,DSK:FORSE.(FORSE has a null extension.)

The filename and extension for input file #1 is assumed to be the filename and extension for filename #2 unless another filename or extension is specified.

The filename and extension for the output file is assumed to be SRCCOM.LST unless another filename or extension is specified.

Switches

Switches are used to specify the manner in which the comparison is to be done. All switches consist of a single character preceded by a slash (/), anywhere in the command string.

Source Compare Switches

Switch	Meaning
/B	Enables the comparing of Blank lines. Normally blank lines are completely ignored.
/C	Comments (all text on a line after a semicolon) are ignored. /C will not cause a line consisting entirely of a comment to become a blank line which will be ignored. /S is also implied.
/S	Spacing (spaces and tabs) is ignored.
/n	(n=1, 2, ..., 9) A match consists of n lines. (n is normally 3) n successive lines must be found identical in the two input files for a match in the two files to be found. When a match is found, all differences between the current match and the previous match are listed. The first line of the match is also listed to make the location in the file easier to find.

Examples

```

.R SRCCOM )
*LPT:←DTA2:SOURCE.001,DTA3:
  SOURCE.002 )

```

Compare the source file SOURCE.001 on DTA2 with the source file SOURCE.002 on DTA3 and list all differences on the line printer.

```

*LPT:←DSK:TRY1,DSK:TRY2 )

```

Compare the two files, TRY1 and TRY2, both of which are on the disk, and list the differences on the printer.

```

*↑C )

```

Return to the Monitor.

Example of Source Compare Output

SRCCOM output	File 1 input	File 2 input
	page 1	page 1
FILE 1) FILE #1	FILE #1	FILE #2
FILE 2) FILE #2		
1)1† FILE #1	A	A
††1) A	B	B
****	C	C
2)1† FILE #2	D	G
† 2) A	E	H
*****	F	I
1)1† D	G	J
1) E	H	1
1) F	I	2
1) G	J	3
****	K	
2)1† G	L	
*****	M	
1)1† K		
1) L		
1) M		
1)2† N		
****	page 2	page 2
2)1† 1	N	N
2) 2	O	O
2) 3	P	P
2)2† N	Q	Q
*****	R	R
1)2† W	S	S
****	T	T
2)2† 4	U	U
2) 5	V	V
2) W	W	4
*****	X	5
	Y	W
	Z	X
		Y
		Z

†These numbers in the SRCCOM listing are page numbers referring to the input files.

††A line identical to both input files is listed to help find the location of the differences within the two files.

Diagnostic Messages

Source Compare Diagnostic Messages

Message	Meaning
?2K CORE NEEDED AND NOT AVAILABLE	SRCCOM needs 2K to initialize IO devices and the core is not available from the Monitor
?BUFFER CAPACITY EXCEEDED AND NO CORE AVAILABLE	The buffer is not large enough to handle the number of lines required for looking ahead and no more core is available from the Monitor.
?COMMAND ERROR	Error in last command string entered.
?DEVICE dev:NOT AVAILABLE	One of the input devices cannot be initialized; generally, the device either does not exist or has been assigned to another job.
?FILE 1 READ ERROR	An error has occurred on the first input device specified in the command.
?FILE 2 READ ERROR	An error has occurred on the second input device specified in the command.
?INPUT ERROR - filename .ext FILE NOT FOUND	The specified filename cannot be found.
?NO DIFFERENCES ENCOUNTERED	No differences were found between the two source files.
?OUTPUT DEVICE ERROR	An error has occurred on the output device.
?OUTPUT INITIALIZATION ERROR	The output device cannot be initialized; the device either does not exist or has been assigned to another job, the device is not an output device or the filename could not be entered on the device.

BINARY COMPARE (BINCOM)

BINCOM compares, word by word, two versions of a binary (.REL) program file and outputs any differences.

Requirements

Minimum Core: 1K if output device is other than DTAn:, MTAn:, or DSK:; otherwise 2K.

Additional Core: See Minimum Core.

Equipment: Two input devices for the two files to be compared; one output device for listing the differences. Both input files can be on disk.

Initialization

```

_R BINCOM )      Loads the Binary Compare routine.
*               Binary Compare is ready to receive
-               a command.

```

Commands

General Command Format

```
list-dev:filename.ext ←input1-dev:filename.ext , input2-dev:filename.ext )
```

list-dev: The device on which the differences are to be listed.

```

LPT:    (line printer)
TTY:    (teletype)
MTAn:   (magnetic tape)
DTAn:   (DECTape)
DSK:    (disk)

```

If list-dev:filename.ext ←is omitted, TTY: is assumed.

input1-dev: The devices on which the two binary files to be compared are located

```

DTAn:   (DECTape)
DSK:    (disk)
CDR:    (card reader)
PTR:    (paper tape reader)
MTAn:   (magnetic tape)

```

filename.ext (DSK: and DTAn: only)

The filename and extension of either of the input binary files.

The filename and extension to be assigned to the output list file.

Binary Compare Diagnostic Messages

Message	Meaning
?FILES BEING COMPARED ON SAME INPUT DEVICE	Files cannot be compared from the same input device unless that device is DSK:.
?INPUT ERROR filename.ext NOT FOUND	The file specified could not be found on the input device.
NO ERRORS ENCOUNTERED	No differences were found between the two binary program files.
?OUTPUT INITIALIZATION ERROR	The file cannot be entered.

Error Differences

Whenever a difference is encountered between the two files being compared, a line is printed on the listing device in the following format:

```
octal loc.      file1-word      file2-word      XOR of both words
```


TENDMP is a utility program, used to save and restore core images on DECtape. It is compatible with the PDP-10 time-sharing system's directory-structured DECtape format, and with the format of a SAVE file.

TENDMP operates in executive mode only, and will run on a PDP-6 or a PDP-10 computer with either a TD-10 or 551/136 DECtape system. (If a PDP-10 is used, the KE-10 Extended Order Code option is required.)

1. TENDMP FUNCTIONS

TENDMP has the following functions:

- 1) Selection of DECtape unit.
- 2) Listing of directory of DECtape.
- 3) Loading a program into core.
- 4) Zeroing of directory.
- 5) Merging a program from tape into core, leaving other areas untouched.
- 6) Dumping nonzero regions of core onto tape.
- 7) Deleting a particular file from a tape.
- 8) Specifying a starting address to be saved with a core image.
- 9) Starting a program loaded from tape.

All of the above functions can be performed by commands from the console Teletype, or by calling TENDMP as a subroutine and providing a command string in core.

2. COMMANDS¹

2.1 Unit Selection

To select a unit, type n $\text{\textcircled{\$}}$ where n is a number from 0 through 7 (unit 0 means 8, as usual).

¹In the command formats which follow, certain conventions apply.

- a. The symbol $\text{\textcircled{\$}}$ represents the ALTMODE character (ASCII code 033, 175, or 176). ALTMODE echoes back as a dollar sign (\$).
- b. Alphabetic characters in commands and filenames can be typed in either upper or lower case; they are considered to be in upper case.
- c. Filenames consist of a 6-character name and a 3-character extension. A space (not a period) separates the name from the extension. All printing characters are legal in filenames. If the space and extension are omitted, a null extension is assumed. SAVE files created by Monitor normally have the extension "SAV."
- d. The character RUBOUT or DELETE erases the entire command currently being typed.

This command causes TENDMP to read the directory block from DECtape unit n into core, and defines that unit as the current unit. This unit and directory are remembered in core, and need only be specified once during a sequence of operations, perhaps including running other programs, if the storage in core from 37177 through 37757 is not disturbed.²

2.2 Listing a File Directory

To list the file directory of the current unit, type

F(\$)

The name and extension of each file will be listed.

2.3 Loading and Starting a Program

To load and start a program, type the filename followed by a carriage return. If the extension is null, it may be omitted.

Examples:

file ext ↵

or

file ↵

This command causes core to be cleared from location 40 through location 37176, the program to be read from tape, and the program to be started at the address which was specified when the program was dumped (see below).

2.3.1 Variations on the Above Loading - To load a program without starting it, that is with control remaining in TENDMP, type:

L(\$file) ↵

This command clears core from 40 through 37176, and loads the program into core.

The clearing of core can be inhibited, and a program merged with existing core, as follows:

M(\$file) ↵

This causes only those areas for which information is present on tape to be modified. For instance,

M(\$EDDT SAV) ↵

would allow EXEC DDT to be used to examine a current core image, or to share core with a maintenance or diagnostic program.

Location 40 is cleared before the merge is done.

²All addresses given in this document assume the 16K version of TENDMP. Translation of these figures for other versions is a simple matter.

2.3.2 Starting the Program - The command to go to the current starting address (that last specified by loading or merging a program or set by the operator (see below)) is:

C(\$)

2.4 Zeroing a Directory

To clear the directory of the current unit, type

Z(\$)

To put a clear directory on a virgin DECTape, type

n(\$)
Z(\$)

The first command is necessary to specify the current unit, and it will read the contents of the directory block into core, even though the contents may be unspecified. However, the Z(\$) command will discard that information, create a legitimate, clear directory in core, and write it on the tape.

A clear directory reserves blocks 1, 2, and 144 (octal) with file code 36, and clears all filenames.

The last word in the directory, which is the tape identifier, is not cleared. If a tape does not have an identifier, one may be added manually by depositing it in location 37376, and then performing a Z(\$), D(\$), or K(\$) command.

2.5 Dumping a Program Onto Tape

When a program is dumped, a starting address is dumped with it. Therefore, a starting address must be specified before dumping.

The current starting address is set each time a program is loaded from tape; it may be left alone if the same address is to be dumped. Otherwise, the command

n(\$)

where n is the new current starting address is used.

Since this is the same format as the unit selection command, the restriction is imposed that the starting address must be greater than seven. In fact, it should be 20 or greater, since the accumulators are all used by TENDMP, and are not dumped with a file.

Dumping a file is accomplished as follows:

D(\$) file ext ↵

This causes the contents of core, from location 20 through 37174, to be dumped with the name "file ext" on the current unit, and with the current starting address. Sequences of two or more zeroes are omitted

from the file. If a file of that name already exists, it is superceded. The correct core size information is put into the directory, but the contents of the date field are unpredictable. The directory is then written onto the tape. Control remains in TENDMP.

2.6 Deleting a File from the Tape

To delete a specific file, type

K\$ file ext

The specified file will be deleted (killed) and the directory will be written back out onto tape.

3. DIAGNOSTIC MESSAGES

As core space in TENDMP is at a premium, there is only one error indication - the Teletype bell is rung. The user should then be able to diagnose the error by examining the last command typed and checking the list of possible errors given below.

- a. The directory may not have been read in from the DECTape as yet, or it may have been clobbered after being read in.
- b. The filename ext. specified in a K\$, L\$, or M\$ command cannot be found on the current DECTape.
- c. There is no room either in the directory or on the DECTape to dump another file. In the latter case, the directory in core may be in an intermediate state, so that rereading the directory from tape by an n\$ command is advisable.
- d. There are either no units or there is more than one unit dialed to the current unit number.
- e. The write-lock switch was on for a D\$, K\$, or Z\$ command.
- f. A tape read or write error occurred.
- g. The tape has run into the end zone (position the tape manually if this is the case).

4. TENDMP VERSIONS

The user should be aware that there are several versions of TENDMP and should check the label of the paper tape version to be used for the following parameters:

- a. Which DECTape control is used - TD-10 or 551.
- b. What memory size is handled. 16K or 32K are the usual cases.
- c. What paper tape format is used - HRI (Hardware Readin Mode - also called RIM10B) or RIM (Readin Mode - read by a special subroutine located in the shadow accumulators on the PDP-6). HRI is preferred for PDP-10.

The creation of these versions is explained in "Assembly Instructions" below.

5. ASSEMBLY INSTRUCTIONS

Assembling a copy of TENDMP from the source is straightforward. The output is normally in RIM10B format, and is, of course, absolute.

The normal assembly is for a 16K machine, and for the TD-10 DECtape control.

Conditional assemblies are provided to modify the above standards.

The normal assembly command string to MACRO is:

```
*PTP:,LPT: ← TTY:,DTAx:TENDMP.MAC ↵
```

To modify the standards, add some of the following at assembly time:

```
MODE=1 ;for 551 control version.
```

MODE is normally zero, representing a TD10 TENDMP.

```
CORE=1 ;for an 8K version
```

```
CORE=4 ;for a 32K version
```

CORE is normally 2, and is the number of 8K blocks of core for this version of TENDMP.

```
DEFINE RIM10B RIM ;for PDP-6 paper tape format
```

Example:

```
*PTP:,LPT: ← TTY:,DTAx:TENDMP.MAC ↵
CORE=4 ↵
↑Z
END OF PASS 1
↑Z
```

6. STORAGE ALLOCATION

TENDMP, when loaded, occupies the upper end of available memory. The figures below are given for a 16K memory; translation of these figures for other core sizes should be obvious.

<u>Locations</u>	<u>Contents</u>
37175, 37176	Cleared
37177 through 37376	Directory of the current DECtape
37377 through 37757	Actual coding and temporary storage areas
37760 through 37776	Area reserved for command string (not modified)
37777	Byte pointer to the ASCII command string which may be in locations 37760 through 37776

In addition to these locations, TENDMP also modifies the contents of all accumulators (location 0 through 17).

Since the actual code occupies locations 37400 through 37757, TENDMP can fit into two DECTape blocks and can therefore be located in and bootstrapped from blocks 0 and 1 of a properly prepared DECTape, using the Hardware Readin feature of the PDP-10/TD-10.

TENDMP's starting address is 37400.

7. CALLING TENDMP AS A SUBROUTINE

TENDMP can be called from a program by the following procedure:

Place a series of commands, in ASCII format, in the reserved core area (37760-37776), omitting line feeds after carriage returns. Place a byte pointer to this command string in location 37777 such that an ILDB 37777 will retrieve the first character. Transfer to 37401. The commands will be executed. If the last command does not cause a transfer out of TENDMP, a RUBOUT should be the last character. This causes a carriage return line feed to be typed out, and control to be switched to the Teletype.

If an error occurs during these commands, the Teletype bell is rung, and control switches immediately to the Teletype.

TENDMP Command Summary

Operation	Command	Comments
Select a DECTape unit, read in its directory, and designate it as the current DECTape.	n(\$)	n must be in the range 0 to 7.
Zero the directory of the current DECTape. ¹	Z(\$)	
Dump nonzero areas of core and the current starting address onto the current DECTape. ¹	D (\$ file ext.)	The core image file is assigned the name "file ext."
Specify a new starting address prior to dumping or before giving the G \$ command.	n(\$)	n is at least greater than 7 but should be greater than 17 ₈ .
Clear core, load a program from the current DECTape, and start it.	file ext.)	"file ext" is the name of the core image file to be loaded.
Clear core, load a program from the current DECTape, but do not start it.	L(\$ file ext.)	
¹ The new directory is written onto the DECTape.		

TENDMP Command Summary

Operation	Command	Comments
Merge a program from the current DECTape; leave the remainder of core undisturbed.	M(\$) file ext ↵	
Go to the current starting address.	G(\$)	
Delete (kill) a file from the current DECTape. ¹	K(\$) file ext ↵	
List the file directory of the current DECTape.	F(\$)	
¹ The new directory is written onto the DECTape.		

CREDITS:

The basic structure of TENDMP was suggested by MACDMP, a program written at the Project MAC PDP-6 installation at MIT. This guidance is gratefully acknowledged.

ADDENDUM

Post Assembly Instructions to Generate a Self Starting TENDMP

When a paper tape TENDMP is assembled with MACRO (version 008) it is not self starting, i.e., after reading the paper tape the operator must depress the "CONTINUE" switch to commence operation at the CTY.

In order to generate a self starting version from the assembler output:

1. Load a monitor with exec DDT
2. Then load assembler paper tape of TENDMP (either 16K or 32K)
3. Enter exec DDT (start at 141) and type

```
$L           : $ = ALTMODE
37400<37757↑R   ↑R = Control R
37400$J
```

The above instructions pertain to 16K TENDMP, for the 32K version the first digit of each number will be 7 instead of 3.

APPENDIX A

PDP-10 Equipment List

	Processor and Processor Options		
KA10	<p>ARITHMETIC PROCESSOR: central processing unit for all PDP-10 systems with floating point and byte manipulation instructions and including:</p> <ul style="list-style-type: none"> - 300 character/second photoelectric paper tape reader - 50 cps paper tape punch - 10 cps console teleprinter, LT35A (LT37 furnished when available) - functional operator console - multiplexed Input/Output Processor (IOP) with seven levels of priority interrupt - real time clock 	MD10E	CORE MEMORY EXPANSION MODULE: 32,768 words, 1.80 μ s cycle time. Up to three may be added to each MD10.
		BS10A	ADDITIONAL MEMORY CABLE SETS: for the MD10.
		DF10	DATA CHANNEL: permits data transfers between high speed devices and core memory. It will service up to eight high speed devices such as RC10, RB10, and RP10.
			Disk Systems
		RC10	SWAPPING DISK CONTROL: provides control for up to 4 RD10 disk files. It connects to the DF10 data channel which provides a direct path to memory and requires at least one RD10.
KM10	FAST REGISTERS: sixteen 36-bit integrated circuit registers used as multiple accumulators and/or index registers and for highly iterative program loops. Replaces the first 16 locations of main core memory.	RD10	SWAPPING DISK FILE: stores 512,000 36-bit words. Average latency time, 17 ms. Transfer rate is 13.3 μ s per 36-bit word. The RD10 provides high speed swapping of programs directly in and out of core memory in timesharing systems. The RD10 can also be used for user file storage. Up to 4 RD10's can be connected to one RC10 disk control unit.
KT10A	DUAL MEMORY PROTECTION & RELOCATION REGISTERS: provide multiprogramming hardware for automatic protection and relocation of reentrant and non-reentrant code. (Required for time-sharing.)	RB10A	STORAGE DISK FILE (dual positioning): stores from 20,971,520 to 104,857,600 36-bit words in multiples of 8,388,608 words. Average access time is 190 ms. Transfer rate ranges from 22.5 μ s to 72 μ s per word depending on zone being accessed. Dual head positioning arms permit overlapping of data transfer and seek operations. Basic unit includes six disks and RA10 Disk Control.
	Core Memory		
	<i>Core memories are available in various sizes and speeds. Each memory stores 36 data bits plus a parity bit and each operates asynchronously with respect to the central processor and channel, establishing its own independent timing cycle.</i>		
MA10	CORE MEMORY: 16,384 words, 1.00 μ s cycle time. Each is supplied with one memory port with cables. Up to three MC10 Additional Memory Access Ports may be added, allowing access to a total of four processors and/or channels.	RB10C	ADDITIONAL DISK: a maximum of 20 disks (each with a capacity of 4,194,304 words) can be added to the basic RB10. (Please specify in even multiples of two RB10C's.)
MA10A	CORE MEMORY: 8,192 words, 1.00 μ s cycle time. Each is supplied with one memory port with cables. Up to three MC10 Additional Memory Access Ports may be added, allowing access to a total of four processors and/or channels.	RP10	DISK PACK CONTROL: provides control of up to eight RP02 Disk Pack Drives. Requires the DF10 data channel which provides a direct path to memory. Also requires at least one RP02.
MC10	ADDITIONAL MEMORY ACCESS PORT: provides the additional cables and logic to connect an additional processor/channel to a MA10, MA10A, or MB10 memory port.	RP02	DISK PACK DRIVE: The RP02 provides storage for up to 5,196,800 36-bit words on interchangeable disk packs. Average access time is 62.5 ms, including 12.5 ms average rotational latency. Transfer rate is 15 μ s/word. Requires RP10 Control. Includes one RP02P pack.
MD10	CORE MEMORY: 32,768 words, 1.80 μ s cycle time. Supplied with four memory access ports and a memory cable set for one of these ports. Up to three additional BS10A memory cable sets are optional.	RP02P	DISK PACK: Pack for RP02 Disk Pack Drive.

Magnetic Tape Systems

TD10	DECtape CONTROL: provides control for up to eight TU55 DECtape transports. Requires at least two TU55 transports. (One TD10 control is required with every PDP-10 system.)
TU55	DECtape UNIT: reads and writes magnetic tape at a 15K characters/second rate. (Tapes are 3½ in. diameter, 260 ft. long and ¾ in. wide.) Tape units are bi-directional and redundantly recorded. Each tape has a directory, allowing random access to user files. (Two DECtape units are required per PDP-10 system.)
TM10A	MAGNETIC TAPE CONTROL: controls up to eight tape transports. Permits reading either 7 or 9 channel (or combination of both) industry standard tape.† Requires at least one DEC magnetic tape unit of the types shown below. Magnetic tape unit types may be intermixed on a single control.
TM10B	MAGNETIC TAPE CONTROL: same as TM10A but provides for data channel operation. Requires a DF10 data channel.
TM10C	TM10B MODIFICATION KIT: provides necessary components for converting a TM10A Magnetic Tape Control to a TM10B.
TU20A	MAGNETIC TAPE UNIT: reads and writes 9-channel USASI standard† magnetic tape at 45 inches/second and a density of 800 bits/inch.
TU20B	MAGNETIC TAPE UNIT: reads and writes 7-channel industry standard tape at 45 inches/second and densities of 200, 556, and 800 bits/inch (36K characters/second).
TU30A	MAGNETIC TAPE UNIT: reads and writes 9-channel USASI standard† magnetic tape at 75 inches/second and density of 800 bits/inch (60K characters/second).
TU30B	MAGNETIC TAPE UNIT: reads and writes 7-channel industry standard tape at 75 inches/second, and densities of 200, 556 and 800 bits/inch (60K characters/second).

Input/Output Devices

Punched Card Equipment

CR10A	CARD READER: reads 80-column punched cards at 1,000 cards/min (800 cards/min in systems using 50 Hz power). Card Hopper and stacker capacities are 1,000 cards.
-------	---

CP10A CARD PUNCH: punches cards at a rate of 200 cards/min when punching in all 80 columns. A maximum rate of 365 cards/min is possible when only the first 16 columns are punched. Card Hopper and stacker capacities are 1,000 cards.

Line Printers

		Characters	Lines/Minute	Columns/Line
LP10A	LINE PRINTER	64	300	132
LP10C	LINE PRINTER	64	1,000	132
LP10D	LINE PRINTER	96	600	132
LP10E	LINE PRINTER	128	500	132

Plotters

XY10 PLOTTER CONTROL: interface for Cal-Comp 500 and 600 series digital incremental plotters.

XY10A PLOTTER AND CONTROL

	Cal Comp Plotter Model	Step Size	Speed (Steps/Minute)	Width (Inches) Paper	
XY10(565)		0.01 inches	18,000	12	
		0.005 inches	18,000		
		0.1 mm.	18,000		
XY10B	PLOTTER AND CONTROL	XY10(563)	0.01 inches	12,000	31
			0.005 inches	18,000	
			0.1 mm.	18,000	

Data Communication Equipment

Data Line Scanner

Data Line Scanner provides on-line servicing of up to 64 communication lines. Accommodates any device which uses eight level serial teletype code at speeds up to 100 kilobaud. Full duplex with local copy, and half duplex data modes are available on each line serviced.

DC10A CONTROL UNIT: the scanner and control unit for the DC10 communication controller provides 4 units of cabinet space and power supplies for various combinations of line equipment.

DC10B 8-LINE GROUP UNIT: provides teletype interface for up to 8 local lines, full duplex. May be used with duplex or full duplex with local copy data sets. When used with data sets, communications must be established, maintained, and terminated manually, unless DC10E Expanded Data Set Control Units are provided. Requires one unit of cabinet space in a DC10A or DC10F.

†USASI X3.22-1968 Recorded Magnetic Tape for Information Interchange.

- DC10C 8-LINE TELEGRAPH RELAY ASSEMBLY: provides conversion from local to long lines using full or half-duplex facilities. Requires two units of cabinet space in a DC10A or DC10F.
- DC10D TELEGRAPH POWER SUPPLY: the standard line voltage supply used with DC10C (120V dc at 2 amperes). No additional cabinet space required.
- DC10E EXPANDED DATA SET CONTROL: provides expanded control of eight data sets in the DC10 system. Requires two units of cabinet space in a DC10A or DC10F.
- DC10F EXPANDER CABINET: provides eight units of cabinet space and power supplies for expansion beyond capacity of DC10A.

680/I Data Communication System

680/I Data Communication System provides on-line servicing of up to 63 communication lines. System handles 8 level serial teletype code at speeds of 110, 150, or 300 baud. Terminals may be local or remote via modems (data sets). To configure a 680/I system, determine the number of lines, both local and data set. Add to the basic communication system enough M750 dual serial line adapters for the total line capacity. (The maximum number of lines is 63.) A 680/I system must include one DC68A. If there are any local teletypes, a DC08B is required. If there are more than 48 local teletypes, a second DC08B is required. Each data set line requires one 689LM. If there are 1 to 32 data set lines, one 689AG is required. If there are more than 32 data set lines, a second 689AG is required.

- DC68A BASIC COMMUNICATION SYSTEM: includes hardware common to any 680/I system for PDP-10 use. Additional options listed below are required to implement a specific number of local or data set lines. The DC68A basic system includes one DA10 PDP-8/PDP-10 interface, one PDP-8/I-D computer (rack mounted with 4K of memory with MP8/I parity option, and an ASR33 teleprinter), one DW08A negative bus adapter, one DL8/I serial line adapter, one DC08A serial line multiplexor, and DC08Y clocks for 110, 150, and 300 baud.
- M750 DUAL SERIAL LINE ADAPTER: implements two full duplex channels in the basic communication system. One unit is required for every two local or data set lines.
- DC08B LOCAL LINE PANEL: accommodates up to 48 local terminals suitable for direct 680/I connection.

689AG MODEM INTERFACE: provides control interface and mounting space for up to 32 689LM's.

689LM MODEM INTERFACE AND CONTROL: provides complete interfacing to and control of one BELL 100 series modem (data set) or equivalent.

Teletypes and Terminals

For Local DC10 Use

- LT33A TELEPRINTER: 33TS machine (KSR33, friction feed).
- LT33B TELEPRINTER: 33TY machine (ASR33, sprocket feed, automatic reader control XON/XOFF feature).
- LT35A TELEPRINTER: VSL312HF machine (KSR35, sprocket feed).
- LT37AC TELEPRINTER: KSR37, sprocket feed, 60 Hz Operation only. Also suitable for use with Bell System 103-type data set or equivalent.

For Local 680/I Use

- LT33C TELEPRINTER: 33TS machine (KSR33, friction feed).
- LT33H TELEPRINTER: 33TY machine (ASR33, sprocket feed, automatic reader control XON/XOFF feature).
- LT35C TELEPRINTER: VSL312HF machine (KSR35, sprocket feed).

Display Systems

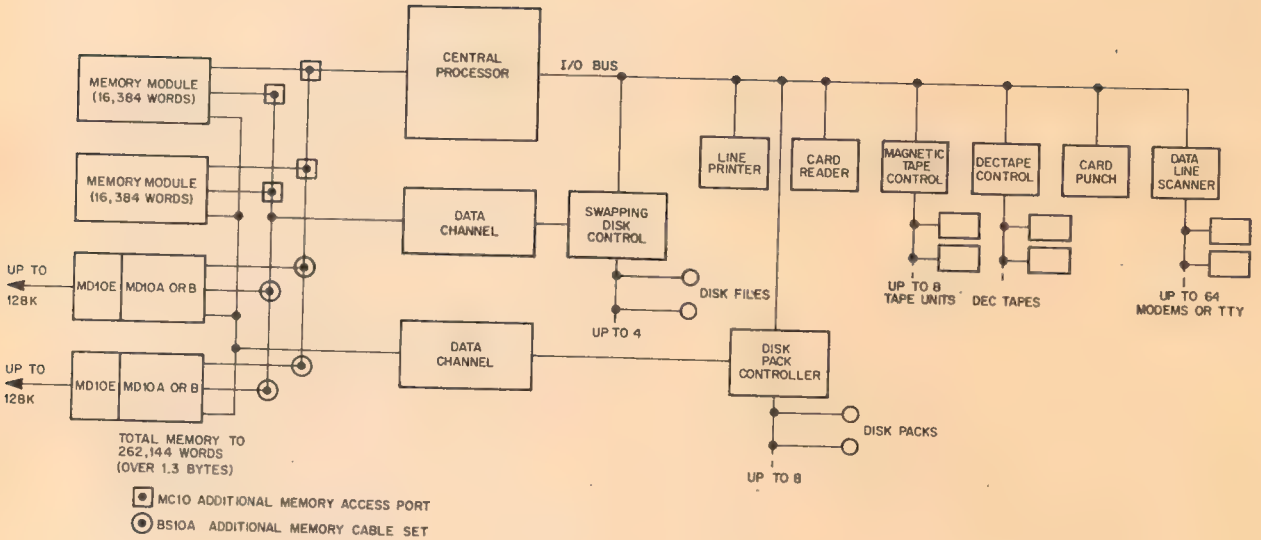
- 346/340B PRECISION INCREMENTAL CRT DISPLAY: plots points, lines, vectors, and characters on a 9 $\frac{3}{8}$ in. square raster of 1,024 points along each axis. 1 $\frac{1}{2}$ μ s is required per point in vector, increment, and character modes. Random point plotting rate of 35 μ s. A 370 high-speed Light Pen is included.
- 342B CHARACTER GENERATOR for 346/340B
- 348/VR30 PRECISION POINT PLOTTING DISPLAY: operates at a maximum plotting rate of 20 KC or one point every 50 μ s on a 9 $\frac{3}{8}$ in. x 9 $\frac{3}{8}$ in. display area. Number of addressable points along each axis is 1024. A 370 high-speed Light Pen is included.
- VP10 POINT PLOTTING DISPLAY CONTROL: operates at either of two maximum plotting rates. Low rate is 10 KC (one point every 100 μ s). High rate is 50 KC (one point every 20 μ s). Number of addressable points along each axis is 1024. Control interfaces to a customer supplied oscilloscope (Tektronix Type RM503 or equivalent) or to a CRT display.
- 370 HIGH SPEED LIGHT PEN: for use with VP10.

Miscellaneous

DA10 PDP-8 or PDP-9 to PDP-10 INTERFACE GP10

GENERAL PURPOSE INTERFACE TO PDP-10 I/O BUS: includes cabinet, two 728 power supplies, one 844 power control, indicators, end panels, fan, convenience outlet with fans, and BS10A/15 ft. cable set. Logic provides a status register, device decoding, read-in gating and line buffering.

DK10 PROGRAMMABLE REAL TIME CLOCK: unit is supplied with a crystal oscillator which provides a resolution of 10 μ s.



TYPICAL PDP-10 SYSTEM CONFIGURATION

Appendix B
PDP-10 Software

Table B-1 shows the DEC-supplied system software (CUSPs) currently available to PDP-10 users.

Table B-1
PDP-10 Software

Name of CUSP	Comment
AID	See description in PDP-10 User's Bookshelf in Appendix C.
BASIC	See PDP-10 User's Bookshelf in Appendix C.
BATCH	See PDP-10 User's Bookshelf in Appendix C.
BINCOM	Documented in this handbook.
*CHKPNT	Saves current charge file and initiates a new one.
CODE	See PDP-10 User's Bookshelf (supplementary documents) in Appendix C..
*COMPIL	Documented in this handbook.
COPY	See PDP-10 User's Bookshelf (supplementary documents) in Appendix C.
CREF	Documented in this handbook.
*DD10	Loads system to disk from DECTape.
DDT	Documented in this handbook.
*DSKLST	Snapshot of disk.
EDITOR	Documented in this handbook.
*FAILSAFE	Saves the contents of disk on magtape and later restores these contents back onto the disk.

Table B-1 (cont)

PDP-10 Software

Name of CUSP	Comment
*FILDDT	Debugging aid for the Monitor.
FORTRAN IV (F40)	See PDP-10 User's Bookshelf in Appendix C.
FUDGE2	Documented in this handbook.
GLOB	Documented in this handbook.
*LINED	Documented in this handbook.
LOADER	Documented in this handbook.
*LOGIN	Documented in this handbook.
*LOGOUT	Documented in this handbook.
MACRO	Documented in this handbook.
*MONEY	Lists charges of computer users.
PIP	Documented in this handbook.
PIPl	Documented in this handbook.
*PRINT	Queues files for LPT.
*PRINTR	Prints selected system files.
*REACT	Alters system accounting file (login numbers and codes).
SRCCOM	Documented in this handbook.
*STACK	See PDP-10 User's Bookshelf in Appendix C.
*SYSTAT	Snapshot of time-sharing system.
TECO	Documented in this handbook.
TENDMP	Documented in this handbook.

*Currently available in disk systems only

pdp10 user's bookshelf

A Bibliography of PDP-10 Programming Documents

OCTOBER, 1969

Software documents in this bibliography can be obtained from Digital Sales Offices or by sending a written request (with check or money order) to *Program Library, Digital Equipment Corporation, Maynard, Massachusetts 01754*. The following key, which indicates the current status of software manuals and their relationship to preceding editions, is designed to help the reader determine whether the present content of a given manual meets his needs.

- (1) *New* signifies that the manual is being published for the first time (designated by a box).
- (2) *Major Revision* signifies that new capabilities and/or changed procedures have been incorporated in the manual (designated by an asterisk).
- (3) *Minor Revision* signifies that the manual remains essentially the same as its predecessor.
- (4) Manuals that are unchanged since the last bibliography are shown with only the date of publication after the title.

PDP-10 System Reference Manual **Minor Revision** **August, 1969**

An indexed programmer's handbook that describes the PDP-10 processor and the basic instruction repertoire. Following an introduction to the PDP-10's central processor structure, general word format, memory characteristics, and assembler source-programming conventions, this manual presents the specific instruction format, mnemonic and octal op codes, functions, timing formulas, and examples of each of the basic instructions. Several helpful appendices, including mnemonic op code tables, algorithms and timing charts, complete the manual.

Order No. DEC-10-HGAC-D \$5.00

***Time-Sharing Monitors:**

Multiprogramming Monitor (10/40) **Major Revision** **August, 1969** **Swapping Monitor (10/50)**

A complete guide to the use of the PDP-10's two powerful, real-time, multiprogramming, time-sharing Monitors. Both Monitors schedule multiple-user time sharing of the system, allocate facilities to programs, accept input from and direct output to all system I/O devices, and relocate and protect user programs in storage. This manual details user interaction with the Monitors, from both a programming and operating viewpoint, and contains several quick-reference tables of commonly used Monitor commands and parameters, as well as examples of user coding.

Order No. DEC-T9-MTZA-D \$3.00

AID (Algebraic Interpretive Dialogue) **October, 1968**

A 'hands-on' guide to the use of AID at the Teletype console. AID, a PDP-10 version of JOSS¹, is an on-line system which provides each user with a personal computing service utilizing a conversational algebraic language. This manual describes the use of the Teletype, the syntax and general rules governing the AID language, and each of the AID commands, with appropriate examples.

Order No. DEC-10-AJBO-D \$3.00

¹JOSS is a trademark and service mark of the RAND Corporation for its computer program and services using that program.

Single-User Monitor Systems **November, 1968**

A complete guide to the use of the Single-User Monitor, which performs fast job-to-job sequencing, provides I/O service for all standard devices, and is upward compatible with the Time-Sharing systems. This manual contains the same type of helpful information as the Time-Sharing manual described above.
Order No. DEC-10-MKDO-D \$2.00

Batch Processor (Batch) and Job Stacker (Stack) **May, 1969**

An indexed manual containing all information required to prepare and run user jobs under control of the Batch Processor in either a single-user or time-sharing environment. Batch supervises the sequential execution of a series of jobs with a minimum of operator attention, yet allows the operator to interrupt, skip, repeat, or prematurely terminate one or more of the jobs in the series at any time. Job Stacker is used in conjunction with Batch to (1) transfer job files to the Batch input device and stack them there for subsequent input to Batch, (2) transfer Batch output job files from the Batch output device to some other device, (3) list job file directories, (4) delete job files, and (5) list directories with selective file deletion or transfer.

Order No. DEC-10-MBAC-D \$1.00

***System User's Guide** **Major Revision, Available** **August, 1969**

A fact-filled operations guide designed for handy reference at the user's Teletype console. Contains the basics of Teletype usage and complete operating procedures for all Commonly Used Service Programs (CUSP'S). Includes complete write-ups on DEctape Editor, Advanced BASIC, LINED, CCL (Concise Command Language), and Linking Loader. A typical chapter includes a brief description of the program, its operating environment, initialization procedures, command string formats, special switches, diagnostic messages, and in-depth examples. The manual is tab-indexed for the user's convenience.

Order No. DEC-10-NGCC-D \$10.00

COBOL LANGUAGE **August, 1969**

A reference manual designed to aid the user in writing COBOL programs for the PDP-10. Each COBOL language element is accorded a detailed treatment that explains and demonstrates its use in a variety of programming contexts. The four major divisions of a COBOL program and their conventional formats are clearly described and effectively illustrated. Other subjects given extended coverage in this manual are the COBOL library, COBOL reserved words, and the CALL procedure. Each chapter contains numerous examples of the efficient use of the components of a COBOL program. Indexed.

Order No. DEC-10-KC1A-D \$6.00

TECO (Text Editor and Corrector) Minor Revision, August, 1969

This programmer's reference manual describes the powerful context editor for the PDP-10. Editing is done on a character, line or variable character string basis. Describes more than 30 commands for inserting, deleting, appending, searching for, and displaying text.

Order No. DEC-10-ETEC-D \$1.50

FORTRAN IV September, 1968

This manual describes statements and features of FORTRAN IV on the PDP-10. Includes descriptions of library functions, calling library subroutines from the Science Library, and the FORTRAN IV operating System. An appendix contains language differences for those using the small (5.5K) PDP-10 FORTRAN Compiler.

Order No. DEC-10-AFCO-D \$2.00

ADVANCED BASIC Minor Revision, August, 1969

A valuable guide to the BASIC® commands needed for a more efficient expression of scientific, business, and educational problems. The manual contains complete tutorial explanations of these additional features: (1) matrix computations; (2) alphanumeric information handling; (3) program control and storage facilities; (4) program editing capabilities; (5) formatting of Teletype output; and (6) documentation and debugging aids.

Order No. DEC-10-KJZB-D \$3.50

PIP (Peripheral Interchange Program) November, 1968

Explains how PIP is used to transfer data files between standard peripheral devices. Shows how command strings are written, describes switches available for optional functions, techniques for handling file directories, error messages and other features.

Order No. DEC-10-PPCO-D \$1.00

Science Library and Fortran Utility Subprograms October, 1968

A general reference manual covering Science Library arithmetic function and utility subprograms and FORTRAN IV nonmathematical utility subprograms (e.g., CHAIN, PDUMP, DATE, TIME). A functional description followed by the calling sequence, list of external subprograms called, entry points, and subprogram length, is given for each subprogram. In addition, the type of argument(s) and result, a description of the algorithm used, and a discussion of the accuracy of the algorithm are given for each function. Appendices contain information on error analyses, double-precision format and input conversion, a bibliography, and average run times.

Order No. DEC-10-SFLC-D \$4.00

MACRO-10 Assembler Minor Revision, October, 1969

The programmer's reference manual for the PDP-10 assembly system. Explains format of statements, use of pseudo-operations, and coding of macro instructions which make MACRO-10 one of the most powerful assemblers available.

Order No. DEC-10-AMZA-D \$3.00

PDP-10 Reference Card May, 1968

A handy pocket-sized guide to instruction mnemonics, hard-

ware and software (Monitor system) word formats, and instruction codes.

Order No. DEC-10-J 00 A-D \$0.25

PDP-10 Interface Manual May, 1968

A complete guide to the process of interfacing any type of experimental apparatus, special purpose I/O devices, or other user-constructed items to the PDP-10 system. This manual details user time-sharing, I/O bus, console, memory bus, and channel bus requirements and provides other information useful to both the novice experimenter and the advanced logic designer.

Order No. DEC-10-HIFB-D \$10.00

DDT-10 (Dynamic Debugging Technique) Minor Revision, April 1969

This reference manual describes the dynamic debugging program used for on-line checkout and testing of MACRO-10 and FORTRAN programs. The commands of DDT are grouped so that they can be used easily and effectively by both the uninitiated user and the experienced programmer. Included in the appendices is an informative summary of all DDT functions.

Order No. DEC-10-CDDC-D \$1.00

The following supplementary documents are also available from the Program Library.

Concise Command Language (CCL) for the PDP-10		
Time-Sharing Monitors	DEC-10-RWDA-D	\$1.00
CHAIN (Reads CHAIN Files into Core and Links Them to Resident Programs)	DEC-10-LOVB-D	1.00
PDP-10 ASCII/BCD Code Conversion Program (CODE)	DEC-10-YNZA-D	1.00
PDP-10 DEctape Copy Program (COPY)	DEC-10-RPTA-D	1.00
FAILSAFE—Disk Save and Restore Program	DEC-10-YPDA-D	1.00
FORTTRAN IV Software Maintenance Memos	DEC-10-KF1A-D	1.00
GLOB (Global Symbol Cross-Reference List)	DEC-10-YRZA-D	1.00
LINED—A Line Editor for PDP-10 Disk Files	DEC-10-EZDA-D	1.00
Linking Loader V.27	DEC-10-LLZA-D	1.00
MACRO V.24 Addendum (Supplements MACRO-10 Assembler Manual)	DEC-10-AMBO-DN	No Charge
FORTTRAN IV Utility Subprograms (RELEASE, MAGDEN, BUFFER, IFILE, and OFILE)	DEC-10-FIYB-D	1.00
TENDUMP—DEctape Utility Program	DEC-10-LZYC-D	1.00
PDP-8 Scan 680 for PDP-10	DEC-10-RSCB-D	1.00
DC08A/689AG Data Line Scanner for PDP-10	DEC-10-RWVA-D	1.00
Software Manual Update, August 1969 (Insert Pages for Updating PDP-10 Software Documents)	(No Order No.)	1.00

MASTER INDEX/GLOSSARY

Page numbers are those which appear in boldface at the top center of each page.

Absolute address:

An address that is permanently assigned by the machine designer to a storage location. See Monitor 354.

Absolute binary programs, 250

Absolute coding:

Coding that uses machine instructions with absolute addresses.

AC, 20

Access:

See random access, remote access, serial access.

Access time:

(1) The time interval between the instant at which data are called for from a storage device and the instant delivery is started.

(2) The time interval between the instant at which data are requested to be stored and the instant at which storage is started.

(3) See page 15

Accumulator, 9, 15, 354

ADD, 17, 45

Address:

(1) An identification, as represented by a name, label, or number, for a register, location in storage, or any other data source or destination such as the location of a station in a communication network.

(2) Loosely, any part of an instruction that specifies the location of an operand for the instruction.

Address assignments, 205-207

indexing, 206

indirect, 206

literals, 206

location counter, 205, 361

Address break, 98, 106, 107

Address format:

The arrangement of the address parts of an instruction.

Address mode,

absolute, 211

relocatable, 211

Addressing, 9, 48

AID, 635

Algorithm:

A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps, e.g., a full state-

ment of an arithmetic procedure for evaluating SINX to a stated precision. Contrast with heuristic.

(1) fixed point, 176-181

(2) floating point, 181-186

Allocation:

See storage allocation.

Allocation of devices, 315

Alphabet:

(1) An ordered set of all the letters and associated marks used in a language.

(2) An ordered set of letters used in a language, e.g., the 128 characters of the USASCII alphabet.

Alphanumeric:

Pertaining to a character set that contains both letters and digits and usually other characters such as punctuation marks. Synonymous with alphameric.

AND, 38

ANDCA, 38

ANDCB, 39

ANDCM, 38

AOBJN, 59

AOBJP, 59

AOJ, 62

AOS, 63

APR, 91, 97, 101

AR (address register), 8

Argument:

An independent variable, e.g., in looking up a quantity in a table, the number, or any of the numbers that identifies the location of the desired value.

Arithmetic and logical operations, 203

shift, 44, 49

testing, 59-64

Arithmetic testing, 59-64

Array:

An arrangement of elements in one or more dimensions.

AS (address switch register), 7, 8

ASCII:

Same as USASCII.

Standard, 220, 240

ASCIZ, 220

ASH, 42, 59

ASHC, 42, 50

Assemble:

To prepare a machine language program from a symbolic language program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses. See MACRO-10.

Assembler:

- (1) A computer program which accepts symbolic code and translates it into machine instructions, item for item. See MACRO-10.
- (2) evaluation of statements and expressions, 267
- (3) interpretation of macros, 271

Assembler statements, 211

- allocation of storage, 224
- control statements, 227
- processing, 223

Assembling TENDMP, 625

Assembly

- listing, 247
- output, 247

ASSIGN command, 316, 348

ASSIGN SYS command, 348, 425

Asynchronous:

The PDP-10 hardware does not rely on an internal clock to indicate by signal that one operation has been executed before beginning a second operation.

ATTACH command, 345, 348

B operator (binary shift), 202

Background Job Control Monitor Commands

- ATTACH job, 345
- CCONT, 344
- CSTART, 344
- DETACH, 344
- PJOB, 344

Background processing:

The automatic execution of lower priority computer programs when higher priority programs are not using the system resources.

Base:

- (1) A reference value.
- (2) A number that is multiplied by itself as many times as indicated by an exponent.
- (3) Same as radix.

Base address:

A given address from which an absolute address is derived by combination with a relative address. See memory protect, virtual memory.

BASIC (Advanced), 636

Batch, 635

Batch processing:

Pertaining to the technique of executing a set of computer programs such that each is completed before the next program of the set

is started.

Loosely, the execution of computer programs serially.

Bell character:

A communication control character intended for use when there is a need to call for human attention. It may activate alarm or other attention devices. Abbreviated BELL.

Benchmark problem:

A problem used to evaluate the performance of computers relative to each other.

Binary:

- (1) Pertaining to a characteristic or property involving a selection, choice, or condition in which there are two possibilities.
- (2) Pertaining to the numeration system with a radix of two.
- (3) See 89, 111, 112, 116
- (4) arithmetic, see 10.

Binary code:

A code that makes use of exactly two distinct characters, usually 0 and 1.

Binary-coded decimal notation:

Positional notation in which the individual decimal digits expressing a number in decimal notation are each represented by a binary numeral, e.g., the number twenty-three is represented by 0010 0011 in the 8-4-2-1 type of binary-coded decimal notation and by 10111 in binary notation. Synonymous with BCD.

Binary Compare, 618-620

- commands, 618-619
- diagnostic messages, 619-620
- initialization, 618
- on CUSP, 634
- requirements, 618

Binary digit:

In binary notation, either of the characters, 0 or 1. Abbreviated bit.

Binary program output

- absolute, 247, 250
- relocatable, 248

Binary shifting, 201

BINCOM, see binary compare

Bit:

- (1) A binary digit.
- (2) See parity bit.
- (3) Position determination, see 200.

Bits, file status, 398

BLK1, 88, 190, 193

BLK0, 88, 193

BLOCK, 221

Block:

- (1) A set of things, such as words, characters, or digits

- handled as a unit.
- (2) A collection of contiguous records recorded as a unit. Blocks are separated by interblock gaps and each block may contain one or more records.
- (3) A group of bits, or binary digits, transmitted as a unit over which an encoding procedure is generally applied for error-control purposes.
- Block gap:**
An area on a data medium used to indicate the end of a block or record. Synonymous with interblock gap.
- Block I/O, 88**
- Block length:**
A measure of the size of a block, usually specified in units such as records, words, computer words, or characters.
- Block transfer:**
The process of transmitting one or more blocks of data where the data are organized in such blocks. See 28.
- Block types, 249-251**
- BLT, 28**
- Boolean, 35**
- Bootstrap:**
A technique or device designed to bring itself into a desired state by means of its own action, e.g., a machine routine whose first instructions are sufficient to bring the rest of itself into the computer from an input device. See 15.
- BR (buffer register), 9**
- Branch:**
- (1) A set of instructions that are executed between two successive decision instructions.
 - (2) To select a branch as in (1).
 - (3) A direct path joining two nodes of a network or graph.
 - (4) Loosely, a conditional jump.
- Branchpoint:**
A place in a routine where a branch is selected.
- Breakpoint:**
A place in a routine specified by an instruction, instruction digit, or other condition, where the routine may be interrupted by external intervention or by a monitor routine. See DDT-10 for use of breakpoints in debugging.
- Buffer:**
A routine or storage device used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transmitting data from one device to another. See 127, 128.
- Buffer header, 401**
- Buffer structure, 396**
- Buffers**
Monitor generated, 402
user generated, 403
- Bug:**
A mistake or malfunction.
- Busy (I/O), 89, 112, 116, 117, 119, 128, 134, 140, 142**
- BYTE, 217**
- Byte:**
- (1) An aggregate of bits whose size lies between that of a word and that of a single bit. On the PDP-10 the byte size is controlled by the programmer.
 - (2) manipulation, 33-35
 - (3) size, altering, 217
size manipulation, 218
 - (4) pointer, 33
 - (5) unpacking subroutine, 257
- Byte interrupt, 73, 75, 104**
- CAI, 60**
- CAL, 229**
- Calculating the logarithm of a complex argument, 256**
- Call:**
- (1) To transfer control to a specified closed subroutine.
 - (2) In communications, the action performed by the calling party, or the operations necessary in making a call, or the effective use made of a connection between two stations.
 - (3) Synonymous with cue.
- CALL and CALLI Monitor operations, 372**
- Calling sequence:**
A specified arrangement of instructions and data necessary to set up and call a given subroutine.
- Calls, macro (see macro calls)**
- CAM, 61**
- Card codes, 162**
- Card in punch, 141**
- Card punch, 140-144, 442**
codes, 162-164
data modes, 442
interrupts, 141, 142
operation, 144
timing, 143
- Card reader, 136-140, 441**
card codes, 443
codes, 162-164
data modes, 441
interrupts, 137, 138
operation, 139
timing, 138
- Carries, 44**
- Carry 0, 44, 63, 64, 73**
- Carry 1 44, 63, 64, 73**
- CCONT command, 344, 375, 376**
- CDP (card punch), 140-142**

- Central processing unit:
 A unit of a computer that includes the circuits controlling the interpretation and execution of instructions. Synonymous with main frame.
- Central processor, 102-109
 indicators, 102
 operating keys, 105
 operating switches, 107
- CHAIN, 636
- Chained list:
 A list in which the items may be dispersed but in which each item contains an identifier for locating the next item to be considered.
- Chaining search:
 A search technique in which each item contains an identifier for locating the next item to be considered.
- Changing the local radix, 215
- Channel:
 (1) A path along which signals can be sent, e.g., data channel, output channel.
 (2) A partially autonomous portion of the PDP-10 which can overlap I/O transmission while computations proceed simultaneously.
- Character:
 A letter, digit, or other symbol that is used as part of the organization, control or representation of data. A character is often in the form of a spatial arrangement or adjacent or connected strokes.
- Character(s) (MACRO-10)
 interpretations, 265
 strings, 198
- Character codes, 48
- Character handling in macros, 271
- Character string:
 A string consisting solely of characters.
- Check bit:
 A binary check digit, e.g., a parity bit.
- Check character:
 A character used for the purpose of performing a check.
- Check sum, 114, 115
- CHKPNT, 634
- CLEAR (see SETZ), 36
- Clear:
 To place one or more storage locations into a prescribed state, usually zero or the space character. Contrast with set.
- Clock:
 (1) A device that generates periodic signals used for synchronization.
 (2) A device that measures and indicates time.
 (3) A register whose content changes at regular intervals in such a way as to measure time.
 (4) See 98, 107 (interrupt)
- Click - hardware option programmable, 632
- CLOG (sample MACRO program), 256
- CLOSE programmed operator, 418
- Closed subroutine:
 A subroutine that can be stored at one place and can be connected to a routine by linkages at one or more locations. Contrast with open subroutine.
- COBOL (COmmon Business Oriented Language):
 A business data processing language.
- COBOL language
 manual, 635
- CODE, ASCII-BCD conversion program, 636
- Code:
 (1) A set of unambiguous rules specifying the way in which data may be represented, e.g., the set of correspondences in the standard code for information interchange.
 (2) To represent data or a computer program in a symbolic form that can be accepted by a data processor.
- Code set:
 A finite and complete set of representations defined by a code.
- Codes
 error, 241
 text, 269
- Collating sequence:
 An ordering assigned to a set of items, such that any two sets in that assigned order can be collated.
- Command execution, 313
- Command format
 command arguments, 311
 command names, 311
- Command language:
 A source language consisting primarily of procedural operators, each capable of involving a function to be executed.
- Commands, DDT
 breakpoints, 544-545, 559-563
 changing output radix, 557
 defining symbols, 566
 deleting symbols, 567
 miscellaneous, 564-565
 modify storage, 542, 549-551
 searches, 563-564
 typein, 554
 typeouts, 552, 557-558
- Commands, TECO
 conditional, 514, 517
 delete text, 510
 editing, 507

- insert text, 510
- I/O, 506
- iteration, 514, 517
- macro, 514, 517
- magnetic tape positioning, 505
- numeric values and arguments, 515
- opening an I/O file, 509
- output data, 510
- pointer positioning, 510
- Q-Register, 513, 517, 518
- read a page, 509
- search, 512, 513, 517
- select I/O device, 503- 04
- termination, 517
- typing text, 511
- Commands, TENDMP, 626
- Comments field, 17, 197
- COMMON (Subroutine), 380
- Communication hardware options
 - control unit, 630
 - data line scanner, 630
 - expanded cabinet, 631
 - expanded data set control, 631
 - telegraph power supply, 631
 - telegraph relay, 631
- Communications system model 680/I
 - basic system, 631
 - dual serial line adapter, 631
 - local line panel, 631
 - modem interface, 631
- Communication with Monitors, 210
- COMPIL, 335, 634
- Compile:
 - To prepare a machine language program from a computer program written in another programming language by making use of the overall logic structure of the program, or generating more than one machine instruction for each symbolic statement, or both, as well as performing the function of an assembler.
- Compiler:
 - A computer program more powerful than an assembler. A compiler accepts symbolic code which it then translates and expands.
 - Examples in PDP-10 systems:
 - FORTRAN and COBOL.
- COMPILE command, 324
- Compile switches, 329
- Complement, 10, 37, 38, 39, 40
- Concatenation:
 - (1) The joining of two strings of characters to produce a longer string often used to create symbols in macro defining. See 237
 - (2) of macros, 272
- Conditional assembly, 222
- Conditional jump:
 - A jump that occurs if specified criteria are met.
- Configuration Table entries, 381
- Configuration for PDP-10, 632
- CONI, 87, 88, 89, 376
- CONO, 86, 89, 90, 92
- Conservation
 - memory, 217
 - storage, 220
- CONSO, 88
- Console,
 - data transfers, 91
 - user's, 309
- CONSZ, 87, 88
- CONT
 - command, 339, 375, 376
 - instruction, 105
- Context switching:
 - The saving of key registers prior to switching between jobs, as in in time sharing.
- Control characters, 430
- Control count, 27, 31
- COPY, 636
- CORE command, 317
- Core control, 420
- Core memory hardware options
 - additional access port, 629
 - cable sets, 629
 - data channel, 629
 - expansion module, 629
- Core storage check, 312
- Counter:
 - A device such as a register or storage location used to represent the number of occurrences of an event.
- CPA (see APR), 97
- CPU:
 - Central Processing Unit
- CR (card reader), 136-138
- Create:
 - A file is created when it has been opened for writing, written upon, and closed for the first time. Only one user may be creating the file at a time. A segment is created by the CORE or REMAP UUU. Logically, GET, R, and RUN commands also do core UUU's.
- CREATE command, 321
- Created symbols, 235
- CREF command, 324
- CREF, see cross reference listing
- CRE.TMP, 336
- Cross reference listing, 604-608
 - commands, 605-606
 - diagnostic messages, 607
 - initialization, 605
 - monitor commands, 608
 - requirements, 605
 - switches, 606-607
- CRT display:
 - Cathode ray tube display.
- CSTART command, 344
- CTEST command, 348
- Current address, 17
- CUSP (Commonly Used Systems Programs, e.g., FORTRAN, PIP, etc.)
- CUSP command level, 303, 304

- CUSP I/O level, 303, 304
- Cylinder:
 A disk can be considered to be a set of cylinders with one cylinder corresponding to each position of the disk arms.
-
- D command, 340
- D switch, 364
- Data bank:
 A comprehensive collection of libraries of data. For example, one line of an invoice may form an item, a complete invoice may form a record, a complete set of such records may form a file, the collection of inventory control files may form a library, and the libraries used by an organization are known as its data bank.
 Synonymous with data base.
- Data blocks, 252
- Data channel, 400
- DATAI, 87, 88, 90
- Data missed, 136, 137, 138
- Data modes
 buffered, 394, 413
 unbuffered, 394, 413
- DATAO, 87-90
- Data ready, 136-138
- Data request, 140-143
- Data transmission, 412
- DAYTIME command, 346
- DDT (Dynamic Debugging Technique):
 A program used for on-line testing and debugging of object programs, 304
 command, 339
 submode, 432
- DDT-10, 537-582
 assembly, 567
 breakpoints 544-545, 559-563
 commands, see commands, DDT
 defining symbols, 566
 deleting symbols, 566
 deleting typing errors, 546, 555
 EDDT, 579
 entering and leaving, 581
 error messages 547, 555
 expressions, 544
 field separators, 568, 569
 learning to use, 539
 loading and saving, 582
 loading procedure, 539
 paper tape control, 570
 proceed counter, 562
 special character functions, 551, 569
 starting the program, 546, 552
 storage map, 580
 symbols, 543, 552, 566-569
 type in modes, 543, 553
 type out modes, 541, 552
 upper and lower case, 551
- DD10, 633
- DEASSIGN command, 316
- Debug:
 To detect, locate, and remove mistakes from a routine or malfunctions from a computer. Synonymous with troubleshoot. See 224
- DEBUG, Monitor command, 325
- Debugging CUSPs, 342
- DEC, Macro-10 pseudo-op, 215
- Decimal print routine, 83
- Decision table:
 A table of all contingencies that are to be considered in the description of a problem, together with the actions to be taken. Decision tables are sometimes used in place of flowcharts for problem description and documentation.
- Decode:
 To apply a set of unambiguous rules specifying the way in which data may be restored to a previous representation, i.e., to reverse some previous encoding.
- DEctape:
 A DEC development of convenient, pocket-sized reels of random access magnetic tape.
 block format, 446
 compatibility between DEC computers, 481
 data modes, 445
 directory format, 446
 file format, 448
 programmed operators, 449
- DEctape control, 630
- DEctape Editor, 493-497
 commands, 493-495
 diagnostic messages, 497
 examples, 495
 initialization, 493
 requirements, 493
- DEctape unit, 630
- DEFINE, 233
- Defined symbol, 197
 deletion, 224
- Defining and calling macros, 271
- DELETE command, 324
- Deleted symbols, 199
- Deleting file from tape, 624
- DEPHASE, 213
- DEPOSIT, 106
- DEPOSIT NEXT, 106
- DETACH command, 344
- DETACH dev command, 344, 348
- Device code, 17
- Device dependent functions, 427
- Device names
 logical, 315, 318
 physical, 315, 318
 redefining, 229
- Device requirements (MACRO-10), 195
- Device summary, 427
- Devices

- directory, 393
 - non-directory, 393
 - Devices, allocation of, 315
 - DFN, 55
 - Direct addressing, 13, 16
 - Direct assignment statements, 199
 - DIRECT command, 323
 - Directory device:
 - A storage retrieval device such as disk or DECTape which contains a file describing the layout of stored data (programs and other files).
 - Directory name:
 - (1) "Project-programmer number" pair which uniquely identifies a directory.
 - (2) The device name in the case of DECTape or magtape.
 - Directory, zeroing a, 623
 - Disk, 461
 - data modes, 461
 - structure of files, 462
 - user programming, 468
 - Disk hardware options
 - additional disk, 629
 - disk pack control, drive, 629
 - storage file, 629
 - swapping control, 629
 - swapping file, 629
 - Dismissing an interrupt, 93
 - Display system hardware options
 - character generator, 631
 - high-speed light pen, 475, 476, 631
 - precision incremental CRT, 631
 - precision point plotting, 631
 - DIV, 47
 - Done (I/O), 89, 112, 116, 119, 121, 128, 134
 - Dormant segment:
 - Description of a sharable high segment kept on swapping space and possibly core which is in no user's addressing space.
 - Double equal sign, 199
 - Double length numbers, 11
 - Double precision:
 - (1) Pertaining to the use of two computer words to represent a number.
 - (2) floating point, 85
 - DPB, 34, 218
 - DS (register), 7
 - DSKLST, 634
 - Dump:
 - A listing of all variables and their values or a listing of the values of all locations in core.
 - Dumping program onto tape, 623
-
- EDS.TMP, 336
 - EDT.TMP, 337
 - Effective address:
 - (1) The actual address used, that is the specified address as modified by any indexing or indirect addressing rules.
 - (2) see 13, 43, 49, 51, 72, 79, 86, 96
 - (3) MACRO-10, 206
 - END, 223, 252
 - End block, 250
 - End of card, 136-143
 - End of file, 137
 - End of transmission block character (ETB)
 - A communication control character used to indicate the end of a block of data where data are divided into blocks for transmission purposes.
 - Entering data, 214
 - changing local radix, 215
 - two half words, 219
 - under prevailing radix, 214
 - ENTER programmed operator, 403
 - ENTER (UUO), 318
 - ENTRY, 231
 - Entry block, 249
 - EOT:
 - The end of transmission character.
 - EQV, 41
 - Error codes (MACRO-10), 241
 - A, D, E, L, 241
 - M, N, O, P, 242
 - Q, R, S, U, V, 243
 - Error detection, 241
 - Error message:
 - An indication that an error has been detected. See 127, 129
 - ETX:
 - The end of text character.
 - EXAMINE NEXT, 105
 - EXAMINE THIS, 106
 - Excess 128 code, 11
 - EXCH, 27
 - EXECUTE command, 325
 - Executive mode, 365
 - EXP, 216
 - Exponent overflow, underflow, 53-59
 - Expressions, 203
 - evaluating, 204
 - nested, 204
 - priority of operations, 204
 - relocatable, 245
 - Extended instructions, 229
 - EXTERN, 231
 - External symbol, 230
-
- Facility allocation Monitor commands
 - ASSIGN, 316
 - CORE, 317
 - DEASSIGN, 316
 - FINISH, 317
 - REASSIGN, 316

- RESOURCES, 318
TALK, 317
- FAD, 56
- FADR, 53
- FAILSAFE, 634, 636
- Fast memory, 9
- FDV, 58
- FDVR, 54
- Field:
In a record, a specified area used for a particular category of data, e.g., a group of card columns used to represent a wage rate or a set of bit locations in a computer word used to express the address of the operand.
- FILDDT, 634
- File:
A collection of related records treated as a unit. In the PDP-10, a named or unnamed collection of 36 bit words (instructions and/or data). Length is not restricted by size of core. One of the uses of files is to initialize segments when they are created with instructions and/or data. See 392
owner, 408
protection, 408
protection key, 409
selection, 403
status bits, 398
- File extension:
1 to 3 alphanumeric characters usually chosen by the program to describe the class of information in file.
extensions, 319
list of, 320
- File manipulation Monitor commands
COMPILE, 324
CREF, 324
DEBUG, 325
DELETE, 324
DIRECT, 323
EXECUTE, 325
LIST, 323
LOAD, 325
RENAME, 324
TYPE, 323
- File, Monitor handling of
comparison with segments, 307
created, 306
names, 306
superseded, 307
updated, 307
- Filename:
1 to 6 alphanumeric characters chosen by the user to identify the file. See 319
- File structured device:
A device on which data is given names and arranged into files; the device also contains directories of these names.
- File update generator, 597, 603
commands, 598-599
diagnostic messages, 602-603
initialization, 597
requirements, 597
switches, 601
- Files (temporary)
CRE.TMP, 336
EDS.TMP, 336
EDT.TMP, 337
FOR.TMP, 336
MAC.TMP, 336
PIP.TMP, 336
SVC.TMP, 335
- FINISH command, 317
- Fixed point, 10
arithmetic, 44-50, 64
decimal numbers, 202
double length, 44
- Flag:
(1) Any of various types of indicators used for identification.
(2) A character that signals the occurrence of some condition, such as the end of a word.
(3) restoration, 77
- Flags, 71, 75, 77, 95, 97, 104, 105, 115
address break, 98, 106, 107
binary, 89, 111, 112, 116
busy (I/O), 89, 112, 116, 117, 119, 128, 134, 140, 142
byte interrupt, 73, 75, 104
card in punch, 141
carry 0, 44, 63, 64, 73
carry 1, 44, 63, 64, 73
clock, 98, 107
data missed, 136, 137, 138
data ready, 136, 137, 138
data request, 140-143
done (I/O), 89, 112, 116, 119, 121, 128, 134
end of card, 136, 143
end of file, 137
error, 127, 129
floating overflow, 51-58
floating underflow, 51, 52, 53, 56, 74, 104
interrupt enables, 136
memory protection, 98, 100
no divide, 51, 54, 58, 74, 104
nonexistent memory, 98, 106, 108
overflow, 44, 49, 51, 52, 53, 54, 56, 58, 63, 64, 72, 98
parity error, 94, 95, 97, 107
power failure, 97
punch on, 140, 142, 143
pushdown overflow, 31, 80, 81, 98, 104
reading card, 136, 138
ready to read, 136-138
stop, 137
tape, 112-114
trap offset, 98
trouble, 136, 137, 140, 141

user, 73, 101
 user in-out, 74, 86, 96, 98, 104
 Floating overflow, c.51-58, 73, 98
 Floating point representation,
 (1) A numeration system in which each number, as represented by a pair of numerals, equals one of those numerals times a power of an implicit fixed positive integer base where the power is equal to the implicit base raised to the exponent represented by the other numeral. See 11
 (2) arithmetic, 50-59, 64
 (3) decimal numbers, 202
 (4) double length, 12, 85
 Floating underflow, 51-56, 74, 104
 FMP, 58
 FMPR, 54
 Foreground processing:
 The automatic execution of high priority programs that have been designed to preempt the use of the computing facilities.
 Formats, 167
 Format characters, rules for handling, 327
 FOR.TMP, 336
 FORTRAN
 (FORMula TRANslating system):
 A language primarily used to express computer programs by arithmetic formulas.
 FORTRAN IV source programs
 creating or modifying, 493, 497
 FSB, 57
 FSBR, 53
 FSC, 52
 FUDGE2, see File Update Generator
 Full duplex software, 429
 Full word data transmission, 27-32
 Functions, device dependent, 427
 Functions (TENDMP), 621

GET command, 338
 GLOB, see global symbol cross reference list
 Global request:
 Request to the Loader to link a global symbol to a program. A global request points to the last reference in the program at which the global symbol was used. Each reference in the program points to the previous reference to the requested global. Such a chain is terminated by a non-relocatable zero address in the program. Chained globals are restricted to references appearing in the address part of a storage word. Symbolic references to the AC or index fields cannot be chained. Locations containing

global symbol references must not be loaded into twice, as unpredictable loader actions may result.
 Global symbol:
 Any symbol accessible to other programs. See 230
 Global symbol cross reference list:
 609-612
 commands, 609, 610
 diagnostic messages, 612
 initialization, 609
 requirements, 609
 switches, 610, 611

Half word data transmission, 20=27
 HALT, instruction, 77, 100, 230
 HALT command, 339, 370
 Handling bytes, 218
 Hardcopy equipment, 123-144
 Hardware:
 Physical equipment, as opposed to the computer program or method of use, e.g., mechanical, magnetic, electrical, or electronic devices
 Contrast with software (2).
 Heuristic:
 Pertaining to exploratory methods of problem solving in which solutions are discovered by evaluation of the progress made toward the final result. Contrast with algorithm.
 High segment:
 (1) In the PDP-10 that segment of the user's core which generally contains pure code and which can be shared by other jobs; usually write protected. (e.g., FORTRAN compiler).
 (2) Block load into, 249
 HISEG pseudo-op, 312
 HISEG statements, 232
 HLL, 20, 21
 HLLE, 22
 HLLO, 22
 HLLZ, 21
 HLR, 20, 25
 HLRE, 26
 HLRO, 26
 HLRZ, 26
 Hollerith:
 Pertaining to a particular type of code or punched card utilizing 12 rows per column and usually 80 columns per card.
 HRL, 20, 22
 HRLE, 23
 HRLO, 23
 HRLZ, 23
 HRR, 20, 24
 HRRE, 25
 HRRO, 25
 HRRZ, 24

H switch (Loader), 361

I, 13

IBP, 34, 218

Identification, 378

IDIV, 47

Idle segment:

A sharable high segment which no users in core are using, however, at least one swapped-out user is using, else it would be a dormant segment.

IDPB, 34, 219

IF, 222

IFDIF, 222

IFIDN, 222

ILDB, 34, 218

Immediate mode addressing:

Process through which the right half of a word gives the operand and not the address.

Impure code:

That code which is modified during the course of a run, e.g., data tables.

Impure segment, 99

IMUL, 46

Indefinite repeat, 237

Indexing, 206, 208

Index registers, 9, 13, 14, 15, 16, 27, 79

Indicators, 102

MEMORY STOP, 104

PI ON, 104

PROGRAM STOP, 104

RUN, 103

USER MODE, 104

Indicator panels, 172

Indirect address:

A single instruction address that is at once the address of another address. The second address is the specific address of the data to be processed. If the second address is also an indirect address, it is known as second-level indirect addressing, and so on to other levels.

Indirect addressing, 3, 14, 16, 49, 51, 77, 206

Information retrieval:

The methods and procedures for recovering specific information from stored data.

INIT (UUO), 368

Initialization,

Buffer, 402

Device, 400, 411

Job, 399

Initialize:

To set counters, switches, and addresses to zero or other starting values at the beginning of, or at prescribed points in, a computer routine.

In-out bit assignments, 170

In-out devices, 156, 170

Input-output, see I/O

Input data word formatting, 217

INPUT (UUO), 368

Instruction:

A statement that specifies an operation and the values or locations of its operands. In this context, the term instruction is preferable to the terms command or order which are sometimes used synonymously.

instructions (illegal), 370

Instructions,

arithmetic testing, 59

byte, 34

fixed point, 45

floating point, 52

without rounding, 55

with rounding, 53

full word, 27

half word, 21

in-out, 86

jump, 74

logic, 36

logical testing, 66

move, 29

pushdown, 31, 80

shift, 43, 49

rotate, 43

Instruction flow, 106

Instruction formats, 12-14, 207

input-output, 209

primary, 210

Instruction times, 19

Interactive time-sharing:

Denotes response between the computer system such as the PDP-10 time-sharing system in which many users at Teletypes can develop and execute programs simultaneously.

Interface, hardware options,

to PDP-10 interface, 632

to PDP-10 I/O bus, 632

Interleaving:

To insert segments of one program into another program so that the two programs can, in effect, be executed simultaneously; e.g., a technique used in multi-programming.

Interlock, 64

INTERN, 231

Internal request, 250

Internal symbol:

A symbol generating a global definition which can be used to satisfy all global requests for that symbol. See 230

Interpreter:

A routine such as a Command String Interpreter that translates and stores each source language statement before translating and storing the next one.

Interpretive compiler:

A routine which, as the computation progresses, translates a stored

program expressed in some machine-like pseudo code into machine code and performs the indicated operations, by means of subroutines, as they are translated. (e.g., AID)

Interrupt, 91, 96

(1) A temporary disruption of the normal operation of a routine by a special signal from the computer, e.g., for I/O purposes.

(2) channel, 117

(3) dismissing, 93

(4) instructions, 94

(5) requests, 92

(6) starting, 92

Interrupt enabled, 136

I/O device hardware options,

card punch, 630

card reader, 630

line printer, 630

plotter, 630

I/O (Input/Output),

(1) Input or output or both.

(2) See 78, 86-91

(3) codes, 157-169

(4) instruction format, 209

I/O instructions, 369

IOR, 39

IOWD, 219

IR (index register), 8

IRP, 237

IRPC, 238

example, 258

JCRY, 75, 230

JCRYO, 75, 230

JCRY1, 75, 230

JEN, 77, 230

JFCL, 75

JFFO, 74

JFOV, 75, 230

Job:

A specified group of tasks prescribed as a unit of work for a computer. By extension a job usually includes all necessary computer programs, linkages, files and instructions to the Monitor.

See 299, 309

attached mode, 309

detached mode, 309

number check, 312

termination Monitor command,

KJOB, 345

Job data area:

The first 140 octal locations of a user's core area. This area provides storage for items used by both the Monitor and the user program. See page 356

JOBAPR, 358, 376

JOBBLT, 357

JOBCHN, 358

JOBCN6, 357

JOBCNI, 358, 376

JOBCOR, 358

JOBDA, 359

JOBDDT, 357

JOBERR, 356

JOB41, 356

JOBFF, 358

JOBHRL, 357

JOBOPC, 358

JOBREL, 356

JOBREN, 358

JOBSA, 358

JOBSYM, 357

JOBTPC, 358, 376

JOBUSY, 357

JOBUUO, 356

JOBVER, 358

JOV, 75, 230

JRA, 79

JRST, 76, 77, 230

JSA, 79

JSP, 76, 78

JSR, 75, 78

JUMP, 61

Jump:

A departure from the normal sequence of executing instructions, synonymous with transfer (1).

Justify:

(1) To adjust the printing positions of characters on a page so that the lines have the desired length and that both the left and right hand margins are regular.

(2) By extension, to shift the contents of a register so that the most or the least significant digit is at some specified position in the register. Contrast with normalize.

K

(1) An abbreviation for the prefix file, i.e., 1000 in decimal notation.

(2) In automatic data processing, loosely, two to the tenth power, 1024 in decimal notation.

Keys, 105

KJOB command, 345

K switch, 364

Labels, 196, 197

LALL, 226

Latency:

The time delay while waiting for a rotating memory to reach a given location as desired by the user. The average latency is one half the revolution time.

LDB, 34, 218

Leader:

The blank section of tape at the

- beginning of a reel or fanfold of tape.
- Least significant bit, 48
- Library subroutines, 231
search mode, 248
- Line Editor for Disk, 499-500
commands, 499
diagnostic messages, 499
initialization, 499
Monitor commands, 500
- LINED
See Line Editor for Disk
- Line printer:
A device that prints all characters of a line as a unit.
Contrast with character printer.
- Line printer, 123-131
data modes, 440
instructions, 125
operation, 129
output format, 124
printing speed, 125
- LINK, 248
- Linking Loader:
This routine loads programs into the user's area of memory, properly relocating each one and adjusting addresses to compensate for relocation. It also links (i.e., provides the main program with the correct address of each referenced subprogram, etc.) internal and external symbols to provide communication between independently assembled programs. It also loads subroutines in library search mode. See 245, 248, 526
chain feature, 533
commands, 527-530
diagnostic messages, 534
initialization, 527
Monitor commands, 536
requirements, 526
switches, 530-533
- Linking subroutines, 230
- LIST, 226, 247
- LIST command, 323
- List:
(1) An ordered set of items.
(2) See chained list, pushdown list, pushup list.
(3) To printout a listing on the line printer or Teletype.
- Listing control, 225, 226
suppression, 225
- List processing:
A method of processing data in the form of lists. Usually, chained lists are used so that the logical order of items can be changed without altering their physical locations.
- LIT, 224
- Literals, 206
multilined, 207
nested, 206
- Load:
In programming, to enter data into storage or working registers.
- LOAD command, 325
- Loader:
Program which attaches pieces of programs together which may have been created separately previous to the run. See Linking Loader, 360
completion of loading, 363
H switch, 361
loading user programs, 356
reentrant, 361
switches, 334
- Loading User Programs, 356
- LOC, 211
- Local radix, 215
changing, 215
- Location counter, 205, 208, 311
- Logarithm of a complex argument, 256
- Logic, 35
- Logical device name:
The name used in ASSIGN commands, 315
- Logical operations, 35-44, 72, 201
- Logical shift, 43, 49, 50
- Logical testing and modification, 65-72
- Logic operator:
A logic operator each of whose operands and whose result have one of two values.
- Login:
The number and the process with which a user identifies himself to a system. It then accepts him as a valid user and assigns him appropriate system resources.
LOGIN, 635. See inside front cover.
Login check, 312
LOGIN command, 314
LOGIN CUSP, 378
- LOGOUT CUSP, 635, 375
LOGOUT UUO, 375
- LOOKUP (UUO), 368, 403
- Loop:
A sequence of instructions that is executed repeatedly until a terminal condition prevails.
- Low segment:
In the PDP-10 that segment of core containing the job data area and I/O buffers, unique and accessible to the user. It is often used to contain the program, but will be used only for data tables, etc. if the user is working with a shared program, such as a system CUSP.
- LPT (line printer), 123, 126
- LSH, 42, 43
- LSHC, 42, 43

-
- MA (memory address), 8
- Machine language:
A language that is used directly by a machine.
- Machine Mnemonics, 260
- Macro calls, 234, 271
format, 234
nested, 239
- Macro:
An instruction in a source language which is equivalent to a specified sequence of machine instructions.
- Macros
calls, 234, 271
concatenation, 272
created symbols, 235
definition, 233, 271
format, 234
indefinite repeat, 237
nesting, 239
redefining, 239
- MACRO-10
creating or modifying programs, 493, 497
diagnostics, 278, 279
error codes, 280-281
operating instructions, 273
- MACRO-10 assembler, 196
definition, 195
device requirements, 195
- MACRO-10, entering data, 213
changing local radix, 215
under prevailing radix, 214
- MACRO-10 statements, 195
assembler, 196
comments, 197
elements, 195
error codes, 241-243
format, 195
labels, 196
operands, 196
operators, 196
relocatable program, 245
Teletype error messages, 244
- MACRO-10, symbols
addresses, 197
deleted, 199
operators, 198
operands, 198
table, 198
- MAC.TMP, 336
- Magnetic Tape, 453, 457
backspace file, 457
data modes, 453
format, 454
MTAPE, 455
9-channel Magtape, 458
- Magnetic Tape Hardware Options
control, 630
modification kit, 630
units, 630
- MAKE command, 321
- Marginal check:
(1) A preventive maintenance procedure in which certain operating conditions, such as supply voltage or frequency, are varied about their nominal values in order to detect and locate incipient defective parts.
(2) Panel, 103
- Mask:
(1) A pattern of characters that is used to control the retention or elimination of portions of another pattern of characters.
(2) A filter.
(3) 14, C.65-71, 83, 86
- Mass storage:
Secondary storage with a large capacity. On a PDP-10, usually a large disk.
- Matrix:
(1) In mathematics, a two-dimensional rectangular array of quantities. Matrices are manipulated in accordance with the rules of matrix algebra.
(2) In computers, a logic network in the form of an array of input leads and output leads with logic elements connected at some of their intersections.
(3) By extension, an array of any number of dimensions.
- Meddling, 423
- Memory, 14-15
- Memory access time, 15
- Memory allocation, 15
- Memory conservation, 217
- Memory protection:
An arrangement for preventing access to certain areas of storage, e.g., Monitor, for purposes of reading or writing. See 97-100
and allocation, 353
flag, 354
- MEMORY STOP, 104
- Merge:
To combine items from two or more similarly ordered sets into one set that is arranged in the same order.
- Message:
An arbitrary amount of information whose beginning and end are defined or implied
- MI (Memory Indicators), 8
- Mnemonic symbol:
(1) A symbol chosen to assist the human memory, e.g., an abbreviation such as "mpy" for "multiply". See 16, 147
(2) Alphabetic, 152
(3) Derivation, 148
(4) Device, 156
(5) Numeric, 149

Mode:

- (1) A method of operation, e.g., binary mode, interpretive mode, alphanumeric mode.
- (2) The characteristic of a quantity being suitable for integer or for floating point computation.
- (3) Method of card reading and punching, i.e., Hollerith code, which interprets each column as a six-bit alphanumeric character or transcription mode, which interprets each punch as a binary one (1) and each non-punch as a binary zero (0).

Modem (MODulator-DEMulator):

A device that modulates and demodulates signals transmitted over communication facilities.

Modes, 19

- arithmetic testing, 59, 60
- fixed point, 45
- floating point, 50, 52, 56
- half word, 21
- logic, 35, 36, 41
- logical testing, 65
- move, 29
- paper tape punch, 89, 115
- readin, 90, 114
- user, 99

MONEY, 635

Monitor:

The specific program which schedules and controls the operation of several related or unrelated routines, performs overlapped I/O and allocates resources so that the computer's time is efficiently used. Also provides context switching in 9 time-shared environment. See 99, 101

Monitor command diagnostic messages, 321, 349

Monitor command interpreter, 311

Monitor commands

extended

- <> construction, 328
- = construction, 328
- + construction, 327
- @ file, 326

functions, 298

interpreter, 302, 304

level, 302, 304

summary, 259

(see inside back cover of this handbook)

Monitor locations, examining, 390

Monitor mode, 310

Monitor operation codes, 371

Monitor UVO's, 367

restriction in reentrant programs, 368

Move instructions, 28

MOVE, 29, 32

MOVM, 30

MOVN, 29

MOVS, 29

MQ (multiplier-quotient register), 9

MUL, 46

Multiline literals, 207

Multiprocessing:

Pertaining to the simultaneous execution of two or more computer programs or sequences of instructions by a computer or computer network. Loosely, parallel processing.

Multiprogramming:

- (1) A technique which allows scheduling in such a way that more than one job is in an executable state at one time.
- (2) Disk Monitor, 295
- (3) Non-disk Monitor, 295, 483

Name block, 250

Negative fixed point numbers, 203

Nesting:

- (1) Including a routine or block of data within a larger routine or block of data.
- (2) Macros, 239, 257
- (3) Subroutines, 80

New symbol, 199

No-Divide, 51, 54, 58, 74, 104

Non-Directory device:

A device such as mag tape or paper tape which does not contain a file describing the layout of stored data (programs and other files).

Nonexistent memory, 98, 106, 108

Non-Reentrant program

one segment, 306

two segment, 306, 353

Non-Reentrant system, 296

Non-Sharable segment:

A segment for which each user has his own copy. Non-sharable segments never have names even if initialized from a file; they may be created by CORE or REMAP UVO.

No-Op:

(1) An instruction that specifically instructs the computer to do nothing, except to proceed to the next instruction in sequence.

(2) 65, 66, 68, 70, 72

Normalization:

(1) This term refers to the positioning of data, left justified with respect to the binary point.

(2) 51, 53, 57, 59, 61

NOSYM, 226

Null character:

A control character that serves to accomplish media fill or time fill e.g., in USASCII the all zeroes character (not numeric zero).

Null characters may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence, but control of equipment or the format may be affected. Abbreviated NUL.

Numbers, 200-205

arithmetic and logical operations, 203

binary shifting, 201

evaluating expressions, 204

fixed-point decimal, 202

floating-point decimal, 202

terms, 204

Number system, 10-12

Numeric terms, 204

NXM STOP, 108

Object code:

(1) Output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

Object program:

(1) The program which is the output of an automatic coding system, usually in machine language ready for execution.

OCT, 215

Octal codes, 260

Octal-to-Decimal conversion, 83

Offset:

(1) The number of locations toward zero a program must be moved before it can be executed. (See LDRBLT description in the Monitor manual.) See 361, 363

One's complement:

In the binary number system this complement is formed by setting each bit to the opposite value. See 10

On-Line:

(1) Pertaining to equipment or devices under direct control of the central processing unit.

(2) Pertaining to a user's ability to interact with a computer.

OP codes, 259

OPDEF, 228

Open subroutine:

A subroutine that must be re-located and inserted into a routine at each place it is used. Synonymous with direct insert subroutine. Contrast with closed subroutine.

OPEN (UUO), 368

Operand:

That which is operated upon. An operand is usually identified by an address part of an instruction.

See 196

Operating keys, 105

CONT, 105

DEPOSIT NEXT, 106

DEPOST, 106

EXAMINE NEXT, 106

EXAMINE THIS, 106

READ IN, 105

RESET, 105

XCT, 106

START, 105

STOP, 105

Operating instructions (MACRO-10), 273

procedures, 210

Operating switches, 107

FM ENB, 102, 109

FP TRP, 109

MA TRP OFFSET, 109

MI PROG DIS, 108

NXM STOP, 108

PAR STOP, 108

REPT, 108

REPT BYP, 108

SHIFT CNTR MAINT, 109

SING CYCLE, 107

SING INST, 107

Operation

card reader, 139

line printer, 129

plotter, 135

processor, 103

punch, 115

reader, 111

Teletype, 117

Operation codes, illegal, 369

Operator:

(1) In the description of a process, that which indicates the action to be performed on operands.

(2) See unimplemented user operator (UUO), programmed operator.

(3) User defined, 228

OR (See IOR), 39

ORCA, 39

ORCB, 40

ORCM, 40

Order of expression evaluation, 267 (TECO), 516

OUTPUT (UUO), 368

Overflow:

That portion of the result of an operation that exceeds the capacity of the intended unit of storage. See 44, 49, 51, 63, 64, 72, 98

Overlay:

The technique of repeatedly using the same blocks of internal storage during different stages of a program. When one routine is no longer needed in storage, another routine can replace all or part of it.

- Pack:**
To compress data in a storage medium by taking advantage of known characteristics of the data in such a way that the original data can be recovered, e.g., to compress data in a storage medium by making use of bit or byte locations that would otherwise go unused.
- PAGE**, 226
- Paper tape punch**, 115-117, 439
data modes, 439
operation, 116
timing, 116
- Paper tape reader**, 111-115, 438
data modes, 438
operation, 113
readin mode, 114
timing, 112
- Parentheses**, 206, 233
- Parity bit:**
A binary digit appended to an array of bits to make the sum of all the bits always odd or always even.
- Parity check:**
(1) A check that tests whether the number of ones (or zeroes) in an array of binary digits is odd or even. Synonymous with odd-even check. See 48, 49, 118
- Parity error**, 94, 107
- PAR STOP**, 108
- Pass:**
One cycle of processing a body of data.
- PASS2**, 224
- Password**, 313
- Patch:**
To modify a routine in a rough or expedient way.
- PC (program counter)**, 7, 72
- Peripheral equipment:**
In a data processing system, any unit of equipment, distinct from the central processing unit, which may provide the system with outside communication.
- Peripheral Interchange Program**, 585-598
commands, 586
diagnostic messages 592, 593
initialization, 585
Monitor commands, 596
requirements, 585
switches, 586-591
- Permanent symbols, redefining**, 229
- PHASE**, 213
- PI**, 91-94, 95, 97
- PI ON**, 10^d
- PIP**, See Peripheral Interchange Program
- PIP.TMP**, 336
- PJOB command**, 344
- Plotter**, 131-135, 474
data modes, 474
instructions, 133
operation, 135
timing, 134
- PLT (Plotter)**, 133, 134
- POINT**, 218
- Pointer:**
The location containing an address rather than data and which the user plans to use to implement indirect addressing.
(2) Byte, 35
(3) I/O block, 88
- POP**, 31, 32
- POPJ**, 81
- Postmortem dump:**
A static dump used for debugging purposes; performed at the end of a machine run.
- Power failure**, 97
- Powers of two**, 174
- Prevailing radix**, 214
- Primary instruction statement**, 208
- PRINT**, 635
- Printer**, see "line printer".
- PRINTR**, 635
- PRINTX**, 227
- Priority interrupt:**
The interrupt that usurps control of the computer program or system and jumps the sequencing to another device, program, program step, or to the device that generates the interrupt signal. See 15, 28, 33, 35, 51, 86, 88, 113
conditions, 94
dismissing an interrupt, 93
interrupt requests, 92
starting an interrupt, 92
timing, 95
- Priority of operations**, 204
- Processor conditions**, 96-98
- Processor hardware options**
arithmetic processor, 629
dual memory protection, 629
fast registers, 629
relocation registers, 629
- Processor modes**, 365
- Processor (standard)**, 330
- Processor switches**, 333
- Program:**
(1) A series of actions proposed in order to achieve a certain result.
(2) To design, write and test a program as in (1).
- Program break:**
The length of a program; the first location not used by a program (before relocation); the relocation constant for the following

- program (after relocation). See 247
- Program control, 72-81
- Program library:
A collection of available computer programs and routines.
- Programmed operators (UO's):
PDP-10 instructions which instead of doing computation, cause a jump into the Monitor system at a predetermined point. The Monitor interprets these entries as commands from the user to perform specified operations. See 210, 366 DECTape, 449
- Programming conventions, 16
- Program origin:
The location assigned by the Loader to relocatable zero of a program. See 361, 363
- Program starting, 369
- PROGRAM STOP, 104
- Project members, 408
- Project-Programmer numbers, 313
- Protected location:
A storage location reserved for special purposes in which data cannot be stored without undergoing a screening procedure to establish suitability for storage therein. See 99
- Protection address, 353, 355
- Protection register, 305, 354
- Pseudo code:
A code that requires further translation prior to execution.
- Pseudo-Op:
(1) An operation that is not part of the computer's operation repertoire as realized by hardware; hence an extension of the set of machine operations.
(2) In MACRO-10, directions for assembly operations.
(3) See 211, 259, 261
- PTP (paper tape punch), 115, 116
- PTR (paper tape reader), 111, 112
- Punch on, 140-143
- Pure code:
Code which is never modified in the process of execution. Hence it is possible to let many users share the same copy of a program. This technique is used by many of the CUSP's. See 99
- PURGE, 224
- PUSH, 31, 32
- Pushdown list:
(1) A list that is constructed and maintained so that the item to be retrieved is the most recently stored item in the list, i.e., last in, first out. See 30, 31
(2) Subroutines containing, 81, 84
- Pushdown overflow, 31, 80, 81, 98, 104
- PUSHJ, 80, 81
- Pushup list:
A list that is constructed and maintained so that the next item to be retrieved and removed is the oldest item still in the list, i.e., first in, first out.
-
- Quantum time, 300
- Queue:
An ordered line waiting for service. See 299
-
- Radix:
In positional representation, that integer, if it exists, by which the significance of the digit place must be multiplied to give the significance of the next higher digit place. For example, in decimal notation, the radix of each place is ten; Synonymous with base. See 200, 213
- RADIX, 213
- RADIX Statement, 213
- RADIX50 statement, 216
- Representation, 270
- Random access:
A device in which the access time is effectively independent of the location of the data. Synonymous with direct access device.
- R Command, 338
- REACT, 635
- READ IN key, 105
- Read-in feature, 25D
- Reading Card, 136, 138
- Readin mode, 90, 114
- Ready to read, 136, 138
- Real time:
(1) Pertaining to the actual time during which a physical process transpires.
(2) Pertaining to the performance of a computation during the actual time that the related physical process transpires in order that results of the computation can be used in guiding the physical process.
- REASSIGN command, 316
- Record:
A collection of related items of data, treated as a unit.
- Redefining macros, 239
- REENTER command, 339
- Reentrant code:
See pure code.
- Re-entrant program:
A two-segment program composed of a sharable and non-sharable segment. See 296, 305, 353, 487
- Reentrant System, 296, 361
- Register, 49

Relative address:

The number that specifies the difference between the absolute address and the base address.

RELEASE programmed operator, 419

RELOC, 211

Relocatable object program, 245, 248

block formats, 249

conventions, 246

Relocate:

In computer programming to move a routine from one portion of storage to another and to adjust the necessary address references so that the routine, in its new location, can be executed. See 99

Relocation address, 353, 355

Relocation before execution, 213

Relocation constant:

The number added to every relocatable reference within a program. The relocation constant is the relocated breakpoint of the previous program. See 246

Relocation Register, 296, 305, 354

REMARK, 227

Remote access:

Pertaining to communication with a data processing facility by one or more stations that are distant from that facility.

Remote station or terminal:

Data terminal equipment for communicating with a data processing system from a location that is time, space, or electrically distant.

RENAME command, 324

RENAME (UUO), 368

REPEAT, 227

Reserving storage, 221

blocks, 221

single location, 221

Response time:

The time which elapses between generation of an inquiry at a terminal and the receipt of a response at the terminal.

Restore, 77

REPT, 108

REPT BYP, 108

RESET, 105

RESOURCES command, 318

Result, 50

RIM format, 252

RIM10 format, 251

RIM10B format, 250, 253, 254

ROT, 42, 43

Rotate, 42

ROTC, 42, 44

Rounding, 52, 59

RSW (See DATAI APR), 91, 230

RUN command, 338

Run control Monitor commands,

CONT, 339

D, 340

DDT, 339

E, 339

GET, 338

HALT, 339

R, 338

REENTER, 339

RUN, 338

SAVE, 340

SSAVE, 341

START, 339

RUN instruction, 103

SAVE command, 340, 341, 368

Scaling, 51

SCHEDULE command, 348

Scheduler:

A section of the Time-Sharing Monitor which determines the sequence of time allotments to users.

Science Library and FORTRAN Utility Subprograms, 636

Serial access:

(1) Pertaining to the sequential or consecutive transmission of data to or from storage.

(2) Pertaining to the process of obtaining data from, or placing data into, storage where the time required for such access is dependent upon the location of the data most recently obtained or placed in storage. Contrast with random access.

Service routine:

A routine in general support of the operation of a computer, e.g., an input-output, diagnostic, tracing, or monitoring routine. Synonymous with utility routine.

SETA, 36

SETCA, 37

SETCM, 37

SETM, 37

SETO, 36

SETZ, 36

Sharable segment:

A segment which can be used by several users at the same time.

Shared code:

Pure code residing in the high segment of user's core.

Shift:

A movement of data to the right or left.

Shift and rotate, 42, 44, 50, 201

Shuffling, 301

Sign bit:

A binary digit occupying the sign position. See 10,49

Significance, 51

Simulate:

(1) To represent certain features of

- the behavior of a physical or abstract system by the behavior of another system.
- (2) To represent the functioning of a device, system, or computer program by another, e.g., to represent one computer by another, to represent the behavior of a physical system by the execution of a computer program, to represent a biological system by a mathematical model.
- SIXBIT, 220
- SKIP, 62
- Software:
- A set of computer programs, procedures, rules, and associated documentation concerned with the operation of a data processing system e.g., compilers, monitors, editors, utility programs. Contrast with hardware.
- SOT, 63
- SOS, 64
- Source Compare, 613, 617
- commands, 614
- diagnostic messages, 617
- initialization, 613
- requirements, 613
- switches, 615
- Source language:
- The language from which a statement is translated.
- Source preparation monitor commands
- CREATE, 321
- EDIT, 321
- MAKE, 321
- TECO, 321
- Source program:
- A program written in a symbolic or algebraic language designed for ease of expression.
- Source word, 20
- Square brackets, 206
- SQUOZE, 216
- See RADIX50
- SRCCOM
- See Source Compare
- SSAVE command, 341, 368
- Stack, 635
- START command, 339
- START instruction, 105
- Starting address, 250
- Static dump:
- A dump that is performed at a particular point in time with respect to a machine run, frequently at the end of a run.
- Status bits
- (See entry for individual devices)
- Status checking and setting, 416
- STATUS (UUO), 368
- STOP, 37
- STOP, 105
- STOPI, 238
- Storage device:
- The PDP-10 device used to store named files by the GET, R, or RUN commands. If the file is marked as sharable (extension= "SHR"), the Monitor will give the segment the same name as the file. This is the only way that a segment can be shared.
- Storage
- conserving, 217, 220
- reserving, 221
- Storage allocation (TENDMP), 625
- Storage I/O channel, 297
- SUB, 45
- Subroutines, 78
- entry point, 78
- frequently used, 213
- library, 231
- linking, 230
- multiple entry, 79, 81
- nesting, 81
- non-reentrant, 78, 100
- two byte unpacking, 257
- SUBTTL, 225
- SVC.TMP, 335
- Swapping:
- The movement of program sections between core and secondary storage.
- I/O channel, 297
- Monitor, 295, 485
- Space, 301
- Storage, 297, 301
- Swapping device:
- Secondary storage suitable for swapping usually a high speed drum or disk.
- Switches, 107
- for compilation listings, 329
- for forced compilation, 332
- for library searches, 332
- for loader maps, 333
- Switches used with monitor commands
- Compile switches, 329
- Loader switches, -34
- Processor switches, 333
- Symbol, 197, 200
- created, 235
- external, 230
- format for block, 249
- global, 230
- internal, 230
- Symbol table, 198, 208
- (1) A dictionary of names used in a program. For example, see MACRO-10.
- (2) direct assignment, 199
- Symbolic address, 197
- (1) An address expressed in symbols convenient to the programmer.
- (2) data reference, 206
- (3) expressions, 216
- (4) operands, 198
- (5) operators, 198
- Symbolic location name, 17
- SYN, 229
- Syntax:

- (1) The structure of expressions in a language.
 - (2) The rules governing the structure of a language.
 - SYS (Device), 387
 - SYSTAT command, 348
 - SYSTAT CUSP, 380, 635
 - System access monitor command
 - LOGIN, 314
 - System administration monitor commands
 - ASSIGN SYS, 348
 - ATTACH dev, 348
 - CTEST, 348
 - DETACH dev, 348
 - SCHEDULE, 348
 - SYSTAT, 348
 - System configuration, 632
 - System timing monitor commands
 - DAYTIME, 346
 - TIME, 346
-
- Table:
 - (1) A collection of data in which each item is uniquely identified by a label, by its position relative to the other items, or by some other means.
 - (2) Search technique, 84.
 - Table numbers (RH of AC), 380
 - Tag:
 - One or more characters attached to an item or record for the purpose of identification.
 - TALK command, 317
 - TAPE, 226
 - Tape, 112, 114
 - TDC, 69
 - TDN, 68
 - TDO, 69
 - TDZ, 69
 - TECO,
 - See Text Editor and Corrector
 - TECO command, 321
 - Teletype, 117, 122
 - codes, 158, 161
 - input, 119
 - output, 119
 - timing, 119
 - Teletype error messages, 244
 - Teletype model 37, 197
 - Teletypes and terminals hardware options
 - DC10 Teleprinters, 631
 - 680I Teleprinters, 631
 - TENDMP
 - assembling, 625
 - calling TENDMP as a subroutine, 626
 - command summary, 626
 - definition, 621
 - diagnostic messages, 624
 - functions, 621
 - self-starting, 8
 - storage allocation, 625
 - versions, 624
 - Testing macros, 224
 - Text codes, 269
 - Text Editor and Corrector, 501
 - commands, see commands, TECO
 - debugging aids, 522
 - diagnostic messages, 520-522
 - initialization, 502
 - monitor commands, 523
 - order of operator evaluation, 516
 - Text input, 220
 - entering characters, 220
 - TIME command, 346
 - Time quantum:
 - That portion of time given to a specific time shared user.
 - Timing,
 - card reader, 138
 - control, 377
 - interrupt, 95
 - line printer, 125
 - plotter, 134
 - punch, 143
 - reader, 112
 - Teletype, 119
 - TITLE, 225
 - TLC, 68
 - TLN, 67
 - TLO, 68
 - TLZ, 67
 - Track:
 - The portion of a moving storage medium, such as a drum, tape, or disk, that is accessible to a given reading head position.
 - Transfer block, 251
 - Trap:
 - An unprogrammed conditional jump to a known location, automatically activated by hardware with the location from which the jump occurrence recorded. See 15
 - Trap offset, 98
 - TRC, 66
 - TRN, 66
 - Trapping, 375
 - console-initiated traps, 376
 - TRO, 67
 - Trouble, 136, 137, 140, 141
 - TRPSET call, 384
 - TRZ, 66
 - TSC, 70
 - TSN, 70
 - TSO, 71
 - TSZ, 70
 - TTCALL UUU, 433
 - TTY (teletype), 117, 119
 - Two byte unpacking subroutines, 257
 - Two's complement arithmetic:
 - Subtraction is performed by means of adding the two's complement of one number to the number it is to be subtracted from. Two's complement is formed by adding one to the one's complement of the given binary number. See 10, 55, 83, 201
 - TYPE command, 323

UFA, 55

Unary operators, 201

Underflow:

Pertaining to the condition that arises when a machine computation yields a nonzero result that is smaller than the smallest nonzero quantity that the intended unit of storage is capable of storing. Contrast with overflow.

Unimplemented operations, 15, 82

Update:

A file is updated when opened for reading and writing, one or more blocks are rewritten in place, and the file closed. Only one user may be updating the file at a time.

USASCII (USA Standard Code for Information Interchange:)

The standard code, using a coded character set consisting of 7-bit coded characters (* bits including parity check), used for information interchange among data processing communication systems, and associated equipment. The USASCII set consists of control characters and graphic characters. Synonymous with ASCII.

User, 73, 101

User defined operator, 228

User facilities, 1-8, 302

User I/O Mode, 365, 383

User In-out, 74, 86, 96, 98, 100, 101, 104

User Mode:

A hardware defined state of the PDP-10 computed during which all instructions executed normally except that all IO and HALT instructions cause immediate jumps into the Monitor. This makes it possible to prevent the user from interfering with any other user or with the operation of the Monitor. Memory protection and relocation are in effect so that the user can modify only his own area of core. See 104, 310, 353, 365.

User program:

All of the code running under control of the Monitor in an addressing space of its own.

programming, see 100

UUO:

Unimplemented User Operator. See program operator.

Monitor, 367

User, 366

See 15, 82, 304

VAR, 224

Vestigial job data area:

The first 10 octal locations of the high segment used to contain data for initializing certain locations in the job data area. See 362, 343

Virtual core:

That amount of core space which the user appears to be able to use. Usually handled by a program which allows the currently referenced parts of the program to be in core at one time, with additional information being brought off storage as needed. See 302

Word formats, 167

Write protect, 99

X, 13

XALL, 226

XCT, 74, 106, 107

XLIST, 226

XOR, 40

XWD, 219

Y, 13

Z, 216

MAIN OFFICE AND PLANT

146 Main Street, Maynard, Massachusetts 01754 • Telephone: From Metropolitan Boston: 646-8600 • Elsewhere: (617) 897-5111 • TWX: 710-347-0212 Cable: Digital Mayn. Telex: 94-8457

UNITED STATES

NORTHEAST

NORTHEAST OFFICE:

15 Lunda Street, Waltham, Massachusetts 02154
Telephone: (617)-891-1030 & 1033

WALTHAM OFFICE:

146 Main Street, Maynard, Massachusetts 01754
Telephone: (617)-891-6310 & 6315

CAMBRIDGE/BOSTON OFFICE:

899 Main Street, Cambridge, Massachusetts 02139
Telephone: (617)-491-6130 TWX: 710-320-1167

ROCHESTER OFFICE:

130 Allens Creek Road, Rochester, New York 14619
Telephone: (716)-461-1700 TWX: 510-253-3078

CONNECTICUT OFFICE:

1 Prestige Drive, Meriden, Connecticut 06450
Telephone: (203)-237-8441 TWX: 710-461-0054

MID-ATLANTIC—SOUTHEAST

MID-ATLANTIC OFFICE:

U.S. Route 1, Princeton, New Jersey 08540
Telephone: (609)-452-9150 TWX: 510-685-2338

NEW YORK OFFICE:

Suite #1

71 Grand Avenue, Palisades Park, New Jersey 07650
Telephone: (201)-941-2016 or (212)-594-6955
TWX: 710-992-8974

NEW JERSEY OFFICE

1259 Route 46, Parsippany, New Jersey 07054
Telephone: (201)-335-3300 TWX: 710-987-8319

PRINCETON OFFICE:

Route One and Emmons Drive,
Princeton, New Jersey 08540
Telephone: (609)-452-2940 TWX: 510-685-2337

LONG ISLAND OFFICE:

1919 Middle Country Road
Centereach, L.I., New York 11720
Telephone: (516)-585-5410 TWX: 510-228-6505

PHILADELPHIA OFFICE:

1100 West Valley Road, Wayne, Pennsylvania 19087
Telephone: (215)-687-1405 TWX: 510-668-4461

WASHINGTON OFFICE:

Executive Building
7100 Baltimore Ave., College Park, Maryland 20740
Telephone: (301)-779-1100 TWX: 710-826-9662

MID-ATLANTIC—SOUTHEAST (cont.)

CHAPEL HILL OFFICE:

P.O. Box 1186, Chapel Hill, North Carolina 27514
Telephone: (919)-929-4095 TWX: 510-920-0763

HUNTSVILLE OFFICE:

Suite 41 — Holiday Office Center
3322 Memorial Parkway S.W., Huntsville, Ala. 35801
Telephone: (205)-881-7730 TWX: 810-726-2122

ORLANDO OFFICE:

Suite 232, 6990 Lake Ellenor Drive, Orlando, Fla. 32809
Telephone: (305)-851-4450 TWX: 810-850-0180

ATLANTA OFFICE:

Suite 116, 1700 Commerce Drive, N.W.,
Atlanta, Georgia 30318
Telephone: (404)-351-2822 TWX: 810-751-3251

KNOXVILLE OFFICE:

Digital Equipment Corporation
5731 Lyons View Dr., S.W., Knoxville, Tenn. 37919
Telephone: (615)-588-6571 TWX: 810-583-0123

CENTRAL

CENTRAL OFFICE:

1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2560 TWX: 910-686-0655

PITTSBURGH OFFICE:

400 Penn Center Boulevard,
Pittsburgh, Pennsylvania 15235
Telephone: (412)-243-8500 TWX: 710-797-3657

CHICAGO OFFICE:

1850 Frontage Road, Northbrook, Illinois 60062
Telephone: (312)-498-2500 TWX: 910-686-0655

ANN ARBOR OFFICE:

230 Huron View Boulevard, Ann Arbor, Michigan 48103
Telephone: (313)-761-1150 TWX: 810-223-6053

MINNEAPOLIS OFFICE:

Digital Equipment Corporation
15016 Minnetonka Industrial Road
Minnetonka, Minnesota 55343
Telephone: (612)-935-1744 TWX: 910-576-2818

CLEVELAND OFFICE:

Park Hill Bldg., 35104 Euclid Ave.
Willoughby, Ohio 44094
Telephone: (216)-946-8484 TWX: 810-427-2608

INTERNATIONAL

CANADA

CANADIAN OFFICE:

Digital Equipment of Canada, Ltd.
150 Rosamond Street, Carleton Place, Ontario
Telephone: (613)-257-2615 TWX: 610-561-1651

OTTAWA OFFICE:

Digital Equipment of Canada, Ltd.
120 Holland Street, Ottawa 3, Ontario
Telephone: (613)-725-2193 TWX: 610-562-8907

TORONTO OFFICE:

Digital Equipment of Canada, Ltd.
230 Lakeshore Road East, Port Credit, Ontario
Telephone: (416)-278-6111 TWX: 610-492-4306

MONTREAL OFFICE:

Digital Equipment of Canada, Ltd.
640 Cathcart Street, Suite 205, Montreal, Quebec
Telephone: (514)-861-6394 TWX: 610-421-3690

EDMONTON OFFICE:

Digital Equipment of Canada, Ltd.
5531-103 Street
Edmonton, Alberta, Canada
Telephone: (403)-434-9333 TWX: 610-831-2248

EUROPEAN HEADQUARTERS

Digital Equipment Corporation International-Europe
81 Route De L'Aire
1227 Carouge / Geneva, Switzerland
Telephone: 42 79 50 Telex: 22 683

GERMANY

COLOGNE OFFICE:

Digital Equipment GmbH
5 Koeln, Bismarckstrasse 7, West Germany
Telephone: 52 21 81 Telex: 841-888-2269
Telegram: Fltp Chip Koeln

MUNICH OFFICE:

Digital Equipment GmbH
8000 Muenchen 19, Leonrodstrasse 58
Telephone: 516 30 54 TELEX: 841 524226

ENGLAND

READING OFFICE:

Digital Equipment Co. Ltd.
Arkwright Road, Reading, Berkshire, England
Telephone: Reading 85131 Telex: 84327

MANCHESTER OFFICE:

Digital Equipment Co. Ltd.
13/15 Upper Precinct, Walkden
Manchester, England M28 5az
Telephone: 061-790-4591/2 Telex: 668666

LONDON OFFICE:

Digital Equipment Co. Ltd.
Bilton House, Uxbridge Road, Ealing, London W.5.
Telephone: 01-579-2781 Telex: 84327

FRANCE

PARIS OFFICE:

Equipment Digital S.A.R.L.
233 Rue de Charenton, Paris 12, France
Telephone: 344-76-07 TWX: 21339

BENELUX

THE HAGUE OFFICE:

(serving Belgium, Luxembourg, and The Netherlands)
Digital Equipment N.V.
Koninginnegracht 65, The Hague, Netherlands
Telephone: 635960 Telex: 32533

SWEDEN

STOCKHOLM OFFICE:

Digital Equipment Aktiebolag
Vretenvagen 2, S-171 54 Solna, Sweden
Telephone: 08 98 13 90 TELEX: 170 50 Digital S
Cable: Digital Stockholm

SWITZERLAND

SWITZERLAND OFFICE:

Digital Equipment Corporation S.A.
81 Route De L'Aire
1227 Carouge / Geneva, Switzerland
Telephone: 42 79 50 Telex: 22 683

CENTRAL (cont.)

ST. LOUIS OFFICE:

Suite 110, 115 Progress Pky., Maryland Heights,
Missouri 63042
Telephone: (314)-872-7520 TWX: 910-764-0831

DAYTON OFFICE:

3101 Kettering Blvd., Dayton, Ohio 45439
Telephone: (513)-299-7377 TWX: 810-459-1676

DALLAS OFFICE:

1625 W. Mockingbird Lane, Suite 309
Dallas, Texas 75235
Telephone: (214)-638-4880

HOUSTON OFFICE:

3417 Milam Street, Suite A, Houston, Texas 77002
Telephone: (713)-524-2861 TWX: 910-881-1651

WEST

WESTERN OFFICE:

560 San Antonio Road, Palo Alto, California 94306
Telephone: (415)-328-0400 TWX: 910-373-1266

ANAHEIM OFFICE:

801 E. Ball Road, Anaheim, California 92805
Telephone: (714)-776-6932 or (213)-625-7669
TWX: 910-591-1189

WEST LOS ANGELES OFFICE:

2002 Cotner Avenue, Los Angeles, California 90025
Telephone: (213)-479-3791 TWX: 910-342-6999

SAN FRANCISCO OFFICE:

560 San Antonio Road, Palo Alto, California 94306
Telephone: (415)-326-5640 TWX: 910-373-1266

ALBUQUERQUE OFFICE:

6303 Indian School Road, N.E.
Albuquerque, N.M. 87110
Telephone: (505)-296-5411 TWX: 910-989-0614

DENVER OFFICE:

2305 South Colorado Blvd., Suite #5
Denver, Colorado 80222
Telephone: 303-757-3332 TWX: 910-931-2650

SEATTLE OFFICE:

1521 130th N.E., Bellevue, Washington 98004
Telephone: (206)-454-4058 TWX: 910-433-3006

SALT LAKE CITY OFFICE:

431 South 3rd East, Salt Lake City, Utah 84111
Telephone: (801)-328-9838 TWX: 910-925-5834

ITALY

MILAN OFFICE:

Digital Equipment S. p. A.
Corso Garibaldi, 49, 20121 Milano, Italy
Telephone: 872 748, 872 694, 872 394 Telex: 33615

AUSTRALIA

SYDNEY OFFICE:

Digital Equipment Australia Pty. Ltd.
75 Alexander Street, Crows Nest, N.S.W. 2065, Australia
Telephone: 439-2566 Telex: AA20740

Cable: Digital, Sydney

MELBOURNE OFFICE:

Digital Equipment Australia Pty. Ltd.
60 Park Street, South Melbourne, Victoria, 3205
Telephone: 69-6142 Telex: AA30700

WESTERN AUSTRALIA OFFICE:

Digital Equipment Australia Pty. Ltd.
643 Murray Street
West Perth, Western Australia 6005
Telephone: 21-4993 Telex: AA92140

BRISBANE OFFICE:

Digital Equipment Australia Pty. Ltd.
139 Merivale Street, South Brisbane
Queensland, Australia 4101
Telephone: 44047 Telex: AA40616

JAPAN

TOKYO OFFICE:

Rikei Trading Co., Ltd. (sales only)
Kozato-Kaikin Bldg.
No. 18-14, Nishishimbashi 1-chome
Minato-Ku, Tokyo, Japan
Telephone: 5915246 Telex: 7814208

Cable: Digital, Tokyo

Digital Equipment Corporation International (engineering and services)

Fukuyoshicho Building, No. 2-6, Roppongi 2-Chome,
Minato-Ku, Tokyo
Telephone No. 585-3624 Telex No.: 0242-2650

.....	TIME-SHARING MONITORS
.....	Contents
.....	Introduction
.....	System Overview
.....	Monitor Commands
.....	SYSTEM REFERENCE MANUAL
.....	Loading User Programs
.....	Central Processor
.....	User Programming
.....	Basic I/O Equipment
.....	Device Dependent Functions
.....	Hardcopy Equipment
.....	EDITOR
.....	Instruction and Device Mnemonics
.....	LINED
.....	I/O Codes
.....	TECO
.....	MACRO-10 ASSEMBLER
.....	LOADER
.....	Statements
.....	DDT
.....	Pseudo-Ops
.....	PIP
.....	Macros
.....	FUDGE2
.....	Error Detection
.....	CREF
.....	Relocation
.....	GLOB
.....	Assembler Output
.....	SRCCOM, BINCOM
.....	Summary of Machine Mnemonics, Assembler Pseudo-ops, Monitor UUO's
.....	TENDMP
.....	Operating Procedures
.....	Appendices
	A. Equipment List
	B. List of Systems Programs
	C. Bookshelf
.....	Index/Glossary

MONITOR COMMANDS

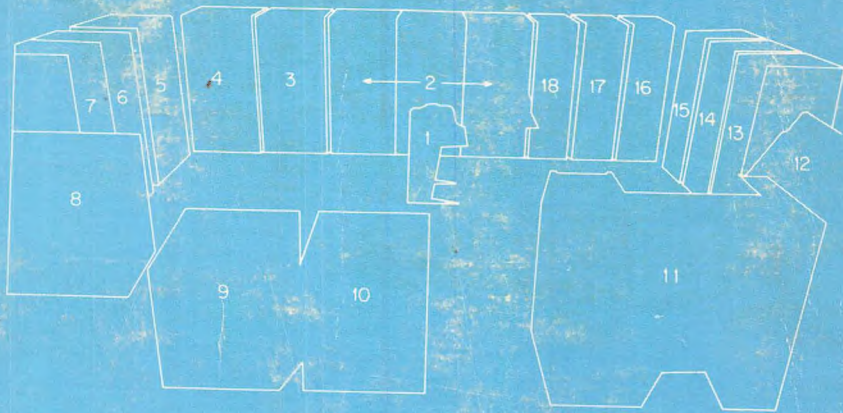
NAME	ABBREVIATION	ARGUMENTS				
		1	2	3	4	5
ASSIGN	AS	dev	ldev			
ASSIGN SYS		dev				
ATTACH	AT	dev				
ATTACH	AT	job	proj prog			
CCONT	CC					
COMPILE	COM	list				
CONT	CON	lh	rh	adr		
CORE	COR					
CREATE	CREA					
CREF	CREF					
CSTART	CS	dev				
CSTART	CS	list				
CTEST		list				
D(deposit)	D	dev				
DAYTIME	DA	dev				
DDT	DD	adr				
DEASSIGN	DEA	file	ext			
DEBUG	DEB	list				
DELETE	DEL	dev				
DETACH	DET	dev	file	ext	proj prog	core
DIRECT	DI					
E(examine)	E					
EDIT	ED	list				
EXECUTE	EX	list				
FINISH	F					
GET	G	file	ext			
HALT	↑ C					
KJOB	K	file	ext	core		
LIST	LI	dev	job			
LOAD	LOA					
LOGIN	LOG	arg				
MAKE	M					
PJOB	P	dev	file	ext	proj prog	core
R	R	dev	file	ext	core	
REASSIGN	REA	n				
REENTER	REE	dev	file	ext	core	
RENAME	REN	adr				
RESOURCES	RES					
RUN	RU	core				
SAVE	SA	file	ext			
SCHEDULE	SCH					
SSAVE	SS					
START	ST					
SYSTAT	SYS					
TALK	TA	dev				
TECO	TE	file	ext			
TIME	TI	job				
TYPE	TY	list				

Key:

adr	octal address	lh rh	octal value of left and right half words
core	decimal number of 1K blocks	proj prog	project-programmer numbers
dev	physical device name	list	a single file specification or a string of file specifications
ldev	logical device name		
ext	filename extension	arg	a pair of file specifications or a string of pairs of file specifications
file	filename		
job	job number assigned by Monitor	n	scheduled use of the system.

See Book 3, Chapter 2 for further explanation of commands.

These abbreviations are accurate and unique as of now, but their accuracy and uniqueness may be changed in the future by the addition of new commands.



- | | |
|------------------------------|---|
| 1. Console Teletype | 10. Disk Pack Unit* |
| 2. Central Processor | 11. Line Printer |
| 3. 16K, 1.0 μ sec Memory | 12. Card Reader |
| 4. 16K, 1.0 μ sec Memory | 13. Magnetic Tape Transport |
| 5. 16K, 1.0 μ sec Memory | 14. Magnetic Tape Transport |
| 6. Data Channel | 15. Magnetic Tape Control |
| 7. Swapping Disk Control | 16. Communications System |
| 8. Swapping Disk | 17. Line Printer/Card Reader Control |
| 9. Disk Pack Unit* | 18. DECTape Control and 3 DECTape Units |

*Disk Pack Control Not Shown

