PART I
PDP-15/10 SOFTWARE SYSTEM
INTRODUCTION

1.1 GENERAL

Software for the PDP-15/10 consists of the COMPACT and the BASIC I/O Monitor Software Systems, which are designed to operate in a paper tape (or card) environment.

1.1.1 COMPACT System

COMPACT is a complete programming system for the PDP-15/10 and includes a symbolic assembler, text editor, debugging routine, various utility routines, and a library of mathematical routines. With COMPACT software, the user can prepare symbolic programs on-line using the Text Editor, assemble them using the CAP-15 Assembler, and execute them under the control of the debugging routine (ODT). The utility and mathematical routines in the system can be incorporated into user programs as required. With the addition of a DECtape transport and control unit, users can use the FAST-15 System to store and retrieve frequently used system and user programs. In addition, users can convert PDP-8 programs (PAL-D, PAL-III, or MACRO-8) to PDP-15 assembly language with the aid of 8TRAN.

1.1.2 Basic I/O Monitor System

The more sophisticated Basic I/O Monitor Software System, available to users with expanded PDP-15/10 Computers (Refer to Table 1-1), is a complete system for the preparation, compilation, assembly, debugging, and operation of relocatable programs.

Powerful system programs include FORTRAN IV, FOCAL, a sophisticated macro assembler (MACRO-15), an on-line debugging system (DDT), an on-line editor (EDIT-15), and a peripheral interchange program (PIP). A versatile and flexible input-output programming system (IOPS) frees the user from the need to create device handling subroutines and from the concerns of device timing. The Basic I/O Monitor is upward compatible with the Advanced and Background/Foreground Monitors of the PDP-15 series; thus, all programs prepared for the Basic I/O Monitor can be run using the Advanced or Background/Foreground Monitors. In addition, users having PDP-8 programs (PAL-D, PAL-III or MACRO-8) may convert them to PDP-15 assembly language with the aid of 8TRAN.

1.1.3 Hardware Requirements

Minimum hardware configurations applicable to the COMPACT and Basic I/O Monitors are provided in Table 1-1.

1.2 COMPACT SYSTEM PROGRAMS

The COMPACT System consists of the programs described in the following paragraphs.

1.2.1 Assembler

The PDP-15/10 Assembler (CAP-15) is a two-pass assembler that requires less than 3K of core memory. It processes source programs to produce an executable binary code. The Assembler makes machine language programming easier, faster, and more efficient. It permits the programmer to use mnemonic symbols to represent instruction operation codes, locations, and numeric data. The programmer can direct the Assembler's processing by means of a powerful set of pseudo operations. An output listing that shows the programmer's source coding, as well as the binary object code produced by the Assembler, can be obtained.

1.2.2 Text Editor

The Text Editor provides for the creation/modification of source programs and other ASCII text material. Commands issued from the teletype direct the Editor to bring a group of lines from the input device to an internal buffer. The user can then, by means of additional commands, examine, delete, and change the contents of the buffer, and insert new text at any point in the buffer. When a block of lines has been edited, it is punched on the output device.

The Editor is most frequently used to modify PDP-15/10 source programs, but it can also be used to edit any symbolic text. The Editor operates with either high-speed or low-speed paper tape devices, and occupies approximately 2000_{10} locations of core memory. Any additional memory is used for buffers.

Table 1-1. PDP-15/10 Hardware Configurations

PDP-15/10 Hardware Configuration Applicable Software	Basic 4K System		Expanded 8K System*	
	ASR 33	ASR 33 DECtape	ASR 33 High-speed Reader/punch	ASR 33 High-speed Reader/punch DECtape
COMPACT	✓	✓	✓	✓
COMPACT FAST 15	—	✓	—	✓
BASIC I/O MONITOR	—	—	✓	✓

*Additional options include the CR03B Card Reader, the 647D or 647F Line Printer, and the TU20, TU20A, TU30, and TU30A Magnetic Tape Transports.

1.2.3 **Debugging Routine**

Octal Debugging Technique (ODT) is a debugging aid that allows the user to conduct an interactive, on-line debugging session using octal numbers and teletype commands. When errors are found, the programmer can correct them on-line and execute the program immediately to test the correction. Thus, ODT can be used to compose a program on-line and check it out as composition progresses. Manual operation of console controls is not required to operate ODT; all functions are initiated by typing commands on the teletype.

1.2.4 **Utility Routines**

Utility routines in the COMPACT Software System include a FAST-15 system for DECtape handling, a hardware readin mode (HRM) punch routine, paper-tape handling routines, teletype I/O routines, an octal dump routine, and a memory scan routine. These routines are briefly described in the following paragraphs.

1.2.4.1 **FAST-15 System** – FAST-15 (Fast Acquisition of System Tape) is a loading system used to retrieve frequently used programs from DECtape and to create system tapes. The main advantages of the system are speed and ease of access. Equipment required for use of FAST-15 includes a Type TC02 DECtape Control Unit and a Type TU55 DECtape Transport.

The FAST-15 system tape, as distributed by Digital Equipment Corporation, contains commonly used system programs such as the Symbolic Editor, the CAP-15 Assembler, and ODT. Since these programs can be called from DECtape with only a small bootstrap, paper tape handling is eliminated. FAST-15 is not restricted to system programs; it also can be employed very conveniently for frequently accessed user programs.

1.2.4.2 **HRM Puncher** – The Hardware Readin Mode (HRM) Puncher is a self-relocating, paper tape dump program. It can be loaded by means of the PDP-15/10 Hardware Readin (HRI) facility into any block of core memory. When loaded, the HRM Puncher relocates itself and punches out an area of memory, specified by the user, in the HRI format.

1.2.4.3 **Paper Tape Handling Routines** – The paper tape handling routines include PTLIST, a paper tape list and PTDUP, a paper tape duplicator. PTLIST can be used to read an ASCII coded paper tape from either the high-speed or low-speed paper tape reader, and provide a character-by-character listing on the teletype. PTDUP can be used to duplicate/verify ASCII or binary tapes using the high-speed paper tape reader and punch.

1.2.4.4 **Teletype Input/Output Routines** – The teletype input/output routines include the Teletype I/O Conversion (TICTOC) Package and the Decimal and Octal Print Package.

TICTOC is used to read 8-bit ASCII code from the teletype and pack it as a 6-bit trimmed ASCII code, and vice versa. Routines in the package fall into three main categories: input, output, and formatting.

The decimal and octal print routines are subroutines that can be used to dump the accumulator in either signed-decimal or octal format.

1.2.4.5 **Octal Dump Routine** – The octal dump routine allows the user to obtain hard copy or paper tape output showing the contents of any register or set of registers that he specifies. The user specifies the registers that are to be dumped via the teletype keyboard.

1.2.4.6 **SCAN Routine** – SCAN is a small program used to scan areas of memory for a particular bit configuration. The user specifies the start and stop address for the area to be scanned, the bit configuration to look for, and the bit positions to be tested. When scanning the area, if a match is found, the address of the match and the matching word are printed.

1.2.5 Mathematical Routines

The COMPACT Software System mathematical routines are grouped into four major packages: Integer Arithmetic, Trigonometric Functions, Floating Point, and Floating Point I/O. The Integer Arithmetic routines allow PDP-15/10 users, without the EAE option, to write programs using simulated multiply and divide instructions. The Trigonometric Package provides the user with a large repertoire of trigonometric functions in both single- and double-precision. The Floating Point Package allows the user with the capability of inputting and outputting decimal data in floating point format. All floating point data transfers are handled through a software accumulator.

1.2.6 8TRAN

8TRAN is used to translate programs written in Pal III, Pal-D, or MACRO-8 assembly language to MACRO-15 assembly language for assembly and execution in the PDP-15 Basic Software environment. The translator produces a straightforward translation that clearly indicates the parts of the translated program to be reviewed in light of the PDP-15's greater word length and more powerful instruction set. It does not simulate the PDP-8 or produce directly executable code.

A full description is contained in PDP-15 8TRAN Manual (DEC-15-ENZA-D).

1.3 BASIC I/O MONITOR

The PDP-15/10 Basic I/O Monitor System simplifies the handling of input/output functions and facilitates the creation, debugging, and use of PDP-15/10 programs. It allows overlapped input/output and computation, as well as simultaneous operation of a number of asynchronous peripheral devices, while freeing the user from the need to create device handling subroutines. The Monitor, operating in conjunction with the Input/Output Programming System (IOPS), provides a complete interface between the user's programs and the peripheral hardware.

The Monitor accepts I/O commands from the system or user programs and supervises their execution. By calling upon the device manipulation routines of IOPS, it provides for simultaneous I/O and computation.

The Basic I/O Monitor contains:

a. Routines for its own initialization and control

b. Tables to allow communication between the Monitor, system programs, user programs, and the Input/Output Program System

c. The CAL Handler, which is used to dispatch to the appropriate Monitor and I/O subroutines

d. Device handlers for the teletype and clock.

The Monitor resides in lower core and occupies about 880_{10} locations.

1.3.1 Input/Output Programming System (IOPS)

The Input/Output Programming System (IOPS) consists of an I/O control routine and individual hardware device handling subroutines that process file and data level commands to the devices. These handlers exist for all standard PDP-15/10 peripherals (see Part II, Chapter 4).

The I/O control routine accepts user program commands and transfers control to the appropriate device handlers. These device handlers are responsible for transferring data between the program and I/O devices, for initiating the reading or writing of files, for the opening and closing of files, and for the performance of all other functions peculiar to a given hardware device. They are also responsible for ignoring functions which they are incapable of handling (for example, trying to rewind a card reader, or skipping files on a non-file-oriented device). All device handlers operate either with or without the Automatic Priority Interrupt (API) option.

1.3.2 System Programs

The Basic I/O Monitor, in addition to the IOPS, is complemented by the following system programs:

a. FORTRAN IV Compiler, Object Time System, and Science Library

b. MACRO-15 Assembler

c. FOCAL

d. Dynamic Debugging Technique (DDT) Program

e. Text Editor Program (EDITOR)

f. Peripheral Interchange Program (PIP)

g. Linking Loader (LOADER)

h. 8TRAN, PDP-8 to PDP-15 Translator

i. Chain Builder Program (CHAIN)

j. Chain Execute Program (EXECUTE)

k. PUNCH-15

1.3.2.1 **FORTRAN IV Compiler** – The PDP-15 FORTRAN IV compiler is a two-pass system that accepts statements written in the FORTRAN IV language and produces a relocatable object program capable of being loaded by the Linking Loader. It is completely compatible with USA FORTRAN IV, as defined in USA Standard X3.9-1966, with the exception of the following features, which were modified to allow the compiler to operate in 8192 words of core storage:

a. Complex arithmetic is not legal.

b. Adjustable array dimensions are not allowed at source level, but can be implemented by calling dimension-adjustment subroutines.

c. Blank Common is treated as Named Common except when object program is used in chaining.

d. The implied DO feature is not included for the DATA statement.

e. Maximum of 5 characters in Hollerith constants.

f. Specification statements must be strictly positioned and ordered.

The FORTRAN IV compiler operates with the PDP-15/10 program interrupt or API facilities enabled. It generates programs that operate with the program interrupt or API enabled and can work in conjunction with assembly language programs that recognize and service real-time devices. Subroutines written in either FORTRAN IV or MACRO-15 assembly language can be loaded with and called by FORTRAN IV main programs. Comprehensive source language diagnostics are produced during compilation, and a symbol table is generated for use in on-line debugging with DDT.

The PDP-15 FORTRAN IV Compiler, Object Time System, and Science Library are described fully in the FORTRAN IV Manual (DEC-15-KFZA-D).

1.3.2.2 **FOCAL** – The FOCAL (Formulating On-Line Calculations in Algebraic Language) compiler is an on-line interactive (conversational) system for non-programmers which makes use of standard mathematical notation and short imperative English command statements to solve user problems.

Arithmetic capabilities of FOCAL permit single commands to generate random numbers and to evaluate standard functions including: square root, absolute value, sign, integer, and natural exponent of any number; sine, cosine, arctangent, and Naperian log value.

With FOCAL, user defined mathematical operations are executed by a single command. In addition, specific hardware, text format, and error codes can be defined to facilitate user needs.

The FOCAL compiler is described fully in the PDP-15 FOCAL Manual (DEC-15-KJZA-D).

1.3.2.3 **MACRO-15** – The MACRO-15 Assembler provides PDP-15/10 users with highly sophisticated macro generating and calling facilities within the context of a symbolic assembler. MACRO-15 is described in detail in the MACRO-15 Assembler Manual (DEC-15-AMZA-D). Some of the prominent features of MACRO-15 include:

a. The ability to -

(1) define macros

(2) define macros within macros (nesting)

(3) re-define macros (in or out of macro definitions)

(4) call macros within macro definitions

(5) have macros call themselves (recursion)

b. Conditional assembly based on the computational results of symbols or expressions.

c. Repeat functions.

d. Boolean manipulation.

e. Optional octal and symbolic listings.

f. Two forms of radix control (octal, decimal) and two text modes (ASCII and 6-bit trimmed ASCII).

g. Global symbols for easy linking of separately assembled programs.

h. Choice of output format: relocatable, absolute binary (check summed); or full binary capable of being loaded via the hardware READIN switch.

i. Ability to call input/output system macros that expand into IOPS calling sequences.

1.3.2.4 **Dynamic Debugging Technique (DDT) Program** – DDT provides on-line debugging facilities within the PDP-15/10 Basic Software System, enabling the user to load and operate his program in a real-time environment while maintaining strict control over the running of each section. DDT allows the operator to insert and delete breakpoints, examine and change registers, patch programs, and search for specific constants or word formats.

The DDT-15 breakpoint feature allows for the insertion and simultaneous use of up to four breakpoints, any one (or all) of which may be removed with a single keyboard command. The search facility allows the operator to specify a search through any part or all of an object program with a printout of the locations of all registers that are equal (or unequal) to a specified constant. This search feature also works for portions of words as modified by a mask. With DDT-15, registers may be examined and modified in either instruction format or octal code, and addresses may be specified in symbolic relative, octal relative, or octal absolute. Patches may be inserted in either source language or octal.

DDT-15 is described more fully in the PDP-15 Utility Program Manual (DEC-15-YWZA-D).

1.3.2.5 **Text Editor Program** – The Text Editor of the PDP-15/10 Basic I/O Monitor provides the ability to read alphanumeric text from an input device (paper tape reader), to examine and correct it, and to write it on an output device. It can also be used to create new symbolic programs.

The Editor operates on lines of symbolic text delimited by carriage return (CR) or ALT MODE characters. These lines can be read into a buffer, selectively examined, deleted or modified, and written out. New text may be substituted, inserted, or appended.

For further details on the Text Editor, refer to the PDP-15 Utility Programs Manual (DEC-15-YWZA-D).

1.3.2.6 **Peripheral Interchange Program (PIP)** – The primary function of PIP is to facilitate the manipulation and transfer of data files from any input device to any output device. It can be used to segment or combine files, perform code conversions, and copy tapes.

Directions for the use of PIP-15 can be found in the PDP-15 Utility Programs Manual (DEC-15-YWZA-D).

1.3.2.7 **Linking Loader** – The Linking Loader loads any PDP-15, FORTRAN IV, or MACRO-15 object program which exists in relocatable format (or absolute format if pseudo-ops .ABS and .FULL are not used). Its tasks include loading and relocation of programs, loading of called subroutines, retrieval and loading of implied subroutines, and building and relocation of the necessary symbol tables. Its operation is discussed in the PDP-15 Utility Program Manual (DEC-15-YWZA-D).

1.3.2.8 **8TRAN** – 8TRAN is used to translate programs written in Pal III, PAL-D or MACRO-8 assembly language to MACRO-15 assembly language for assembly and execution within the PDP-15 Basic Software environment. The Translator produces a straightforward translation that clearly indicates the parts of the translated program requiring review, in light of the PDP-15's greater word length and more powerful instruction set. It does not simulate the PDP-8 or produce directly executable code. A full description is contained in PDP-15 – 8TRAN Manual (DEC-15-ENZA-D).

1.3.2.9 **Chain Builder and Execute Programs** – The Chain Builder and Execute programs provide the user with a capability for program segmentation which allows for multiple core overlap of executable code and certain types of data areas. A more complete description of the Chain Builder and Execute programs is given in the Utility Programs Manual (DEC-15-YWZA-D).

1.3.2.10 **PUNCH-15** – PUNCH-15 is a utility program used to output selected areas of core memory onto paper tape for use with the PDP-15 Basic I/O Monitor. The primary applications of PUNCH-15 in the Basic I/O Monitor environment are:

a. System program modification

b. +.DAT SLOT reassignment

c. Production of an executable user program core load on a single paper tape. This is particularly useful when that core load consists of a main program and several subroutines/library routines, the constant reloading of which may be time consuming.

The PDP-15 Utility Programs Manual (DEC-15-YWZA-D) describes PUNCH-15 more fully.

PART II
COMPACT SOFTWARE

COMPACT Assembler
(CAP-15)

COMPACT
Text Editor

Octal Debugging
Technique

COMPACT
Utility Routines

COMPACT
Mathematical Routines

APPENDICES

PART II
COMPACT SOFTWARE

CHAPTER 1
COMPACT ASSEMBLER (CAP-15)

1.1 INTRODUCTION

The CAP-15 Assembler processes symbolic source programs to produce a binary code that can be executed by the PDP-15/10 computer. It normally processes source programs in two passes and requires 3K of core memory. The Assembler makes machine language programming for the PDP-15/10 easier, faster, and more efficient. It permits the programmer to use mnemonic symbols to represent instruction operation codes, locations, and numeric quantities; thus, the programmer can easily refer to any point in his program without knowing actual machine locations.

The programmer can direct the Assembler's processing by means of a powerful set of pseudo-operation (pseudo-op) instructions. These pseudo-ops can be used for program control, to reserve blocks of core storage, to set the radix for numerical interpretation by the Assembler, to handle strings of text characters in 6-bit ASCII code, to define symbolic addresses, and many other functions that are described in detail in Section 1.4.

An output listing, showing both the programmer's source coding and the object program produced by the Assembler, can be printed. This listing includes all the symbols used by the programmer and their assigned values. If assembly errors are detected, erroneous lines are marked with meaningful letter codes, which can be interpreted by referring to Section 1.5.4. Operating procedures are found in Section 1.5.

The Assembler normally processes a source program in two passes; that is, it reads the same source program twice, producing a printed listing and/or outputting the object code during the second pass. (However, if the listing and binary output utilize the same device, then a third pass is required to produce the binary output.) Assembler coding for the two passes is resident in memory at the same time. PASS1 and PASS2 are almost identical in operation. The main function of PASS1 is to identify locations that are to be assigned symbols, and to construct a symbol table. PASS2 uses the information derived by PASS1 (and left in memory) to produce the final listing object code output.

1.2 HARDWARE REQUIREMENTS AND OPTIONS

The Assembler operates with PDP-15/10 systems having at least 4K of core memory and a console Teletype (ASR33). With the addition of a high-speed paper tape reader/punch, the user can significantly decrease assembly time and also have the capability of simultaneously producing a listing and binary output during PASS2.

1.3 ASSEMBLY LANGUAGE ELEMENTS

1.3.1 Program Statements

A single statement may be written on a 72-character teletype line, in which case a carriage return delimits the statement. Since a carriage return is not a printable character, it is represented in this manual as ↵

```
STATEMENT↵
```

Several statements may be written on a single line, separated by semicolons:

```
STATEMENT;STATEMENT;STATEMENT↵
```

In this case, the statement line ends with a carriage return character, but semicolons are used as internal statement delimiters. Thus, if a statement is followed by another statement on the same line, it must end with a semicolon.

A statement may contain up to four fields separated by a space, spaces, or tab characters. These four fields are the label (or tag) field, the operation field, the address field, and the comments field. Because the space and tab characters are not printed, the space is represented by ␣ , and the tab by ⇥ in this manual. Tabs are normally set ten spaces apart on most teletype machines, and are used to line the fields up in columns in the source program listing.

The basic statement format is as follows:

```
LABEL ⇥ OPERATION ⇥ ADDRESS ⇥ /COMMENTS↵
```

where each field is delimited by a tab or space, and each statement is terminated by a semicolon or carriage return. The comments field is preceded by a tab (or space) and a slash (/).

An assembly statement may have an entry in any or all of the four fields. The following forms are acceptable:

```
TAG↵
TAG ⇥ OP↵
TAG ⇥ OP ⇥ ADDR↵
TAG ⇥ OP ⇥ ADDR ␣ comments↵
TAG ⇥ OP ␣ comments↵
TAG ⇥   ⇥ ADDR↵
TAG ⇥   ⇥ ADDR ␣ comments↵
TAG ⇥ comments↵
    ⇥ OP↵
    ⇥ OP ⇥ ADDR↵
    ⇥ OP ⇥ ADDR ⇥ comments↵
    ⇥ OP ⇥ comments
    ⇥  ⇥ ADDR↵
    ⇥  ⇥ ADDR ⇥ comments↵
/comments↵
     ⇥comments↵
```

Note that when a label field is not used, its delimiting tab is written, except for lines containing only comments. When the operation field is not used, its delimiting tab is written if an address field follows, except in label only and comments only statements.

A *label* (or tag) is a symbolic address created by the programmer to identify the statement. When a label is processed by the Assembler, it is said to be defined. A label can be defined only once. The *operation code* field may contain a machine mnemonic instruction code, a pseudo-op code, a number, or a symbol. The *address field* may contain a symbol, number, or expression which is evaluated by the Assembler to form the address portion of a machine instruction. In some pseudo-operations, this field is used for other purposes, as explained later in this manual. *Comments* are usually short explanatory notes that the programmer adds to statement as an aid in analysis and debugging. Comments do not affect the object program or assembly processing; they are merely printed in the program listing. Comments must be preceded by a slash (/); the slash must be preceded by one of the following:

a. Space
b. Tab
c. Semicolon
d. Carriage return

1.3.2 **Symbols**

A symbol consists of a string of alphabetic or alphanumeric characters (including periods and percent signs). The first character of a symbol *must* be a letter, a period or a percent sign. The first character of a symbol in the label field must *not* be a digit. A period can not be used alone as a symbol in the label field. The letter X, if used alone, can only appear in an address field (see 1.3.4.4).

The following symbols are legal:

MARK 1	. . 1234	.A
A%	%50.99	.%
P9.3	INPUT	.5A

The following symbols are illegal:

TAG:1	L @ B1	The colon (:) and @ are illegal characters.
5ABC		The first character may not be a digit.
X		See section 1.3.4.4.

Only the first six characters of a symbol are meaningful to the Assembler, but the programmer may use more for *his own* information. If he writes,

SYMBOL1
SYMBOL2
SYMBOL3

as the symbolic labels on three different statements in his program, the Assembler will recognize only SYMBOL and type error flags on the lines containing SYMBOL1, SYMBOL2 and SYMBOL3 because to the Assembler they are duplicates of SYMBOL.

1.3.2.1 **Evaluation of Symbols** – When the Assembler encounters a symbol during processing of a source language statement, it evaluates the symbol by reference to two tables: The user's symbol table and the permanent symbol table. The user's symbol table contains all symbols defined by the user. The user defines symbols by using them as labels or by direct assignment statements (see Section 1.3.2.2). A label is defined when first used, and cannot be redefined. When a label is defined by the user, it is given the current value of the Location Counter (see Section 1.3.4).

The Assembler has, in its permanent symbol table, definitions of the symbols for all of the PDP-15/10 memory reference instructions, operate instructions, index and limit register instructions, and some input/output transfer instructions. (See Appendix B for a complete list of these instructions.) Both the permanent symbol table and the user's symbol table reside in storage in alphanumeric sequence. The permanent symbol table symbols can be used in the operation field of a statement without prior definition by the user.

Example

→| LAC ␣ A↵	When the LAC symbol appears in the operation field of a statement, the Assembler treats it as an op code rather than a symbolic address. It has a value of 200000_8, which is taken from the operation code definition in the permanent symbol table.

The user can use instruction mnemonics (see Appendix B) or the pseudo-instruction mnemonics code (see Appendix C) as symbol labels. For example,

```
DZM →| DZM ␣ Y↵
```

where the label DZM is entered in the user's symbol table and given the current value of the Location Counter, and the op code DZM is given the value 140000 from the permanent symbol table. The user must be careful, however, in his usage of these dual purpose (field dependent) symbols. Symbols in the operation field will be interpreted as either instruction codes or pseudo-ops, rather than as a symbolic label, if they are in the permanent symbol table. In the following example, several symbols with values have been entered in the user's symbol table and the permanent symbol table. The sample coding shows how the Assembler uses these tables to form object program storage words.

User Symbol Table		**Permanent Symbol Table**	
Symbol	*Value*	*Symbol*	*Value*
TAG1	100	LAC	200000
TAG2	200	DAC	040000
DAC	300	JMP	600000

If the following statements are written, the following code is generated by the Assembler

```
                  .
                  .
TAG1 →|  DAC  →|  TAG2↵                    040200
                  .
                  .
TAG2 →|  LAC  →|  DAC↵                     200300
```

```
DAC    →| JMP    →| TAG1↵                        600100
            .
            .
            .
       →| TAG1↵                                  000100
            .
            .
            .
       →|          →| DAC↵                       000300
```

1.3.2.2 **Direct Assignment Statements** – The programmer may define a symbol directly in the user's symbol table by means of a direct assignment statement, written in the form:

```
SYMBOL = n↵

        or

SYM1 = SYM2↵
```

where n is any number or expression. There should be no spaces before the symbol or between the symbol and the equal sign. (The operation field is assumed to be to the right of the equal sign, unless it is followed by a space or a tab, in which case the address field is assumed.) The Assembler enters the symbol in the user's symbol table, along with the assigned value. Symbols entered in this way can be redefined. The following are legal direct assignments:

```
Z = 28; A = 1; B = 2↵
```

A symbol can also be assigned a symbolic value:

```
A = 4↵
B = A↵
```

The symbol B is given the value 4. Direct assignment statements do not generate storage words in the object program. In general, it is good programming practice to define symbols before using them in statements which generate storage words. The Assembler will interpret the following sequence without trouble.

```
Z = 5↵
Y = Z↵
V = Y↵
  →| LAC ␣ V ␣ /SAME AS LAC 5
```

A symbol may be defined after use. For example,

```
  →| LAC ␣ Y↵
Y = 1↵
```

This is called a forward reference, and is resolved properly in PASS 2. When first encountered in PASS 1, the LAC Y statement is incomplete because Y is not yet defined. Later in PASS 1, Y is given the value 1. In PASS 2, the Assembler finds that Y = 1 in the symbol table, and forms the complete storage word.

Since the Assembler operates in two passes, only one-step forward references are allowed. The following forward reference is illegal:

```
    ⇥LAC ␣Y↵
Y = Z↵
Z = 1↵
```

In the listing, during PASS1, the line which contains Y = Z will be printed with an A error code indicating a direct assignment error.

1.3.2.3 **Undefined Symbols** – If any symbols remain undefined at the end of PASS1 of assembly, they are automatically defined as the addresses of successive registers following the end of the program (i.e., following the highest program counter value encountered by the Assembler). All statements that referenced the undefined symbol will be flagged as undefined. One memory location is reserved for each undefined symbol with the initial contents of the reserved location being unspecified.

Example

Location Counter	Source Statement	Generated Code	Comments
100	⇥LAC UNDEF1↵	200104	Flagged as an error
101	⇥LAC UNDEF3	200106	
102	⇥LAC UNDEF4	200107	
103	⇥LAC UNDEF2	200105	
	⇥.END		

1.3.3 Numbers

1.3.3.1 **Integer Values** – An integer is a string of digits, with or without a leading sign. Negative numbers are represented internally in two's complement form. The range of integers is as follows.

Unsigned	$0 \rightarrow 262143_{10}$	(777777_8)	e.g., 2^{18} -1
Signed	$0 \rightarrow 131071_{10}$	(377777_8)	e.g., $\pm\ 2^{17}$-1

An octal integer* is a string of digits (0-7), signed or unsigned. If a non-octal digit is encountered (8 or 9) it will be flagged as a numerical error.

*Initiated by usage of .OCT pseudo-instruction and is also the initial assumption if no radix control pseudo-instruction was encountered (see Section 1.4.4).

Example

Coded Value	Generated Value (Octal)	Comment
-5	777773	two's complement
3347	003347	
3779	000000	error

A decimal integer* is a string of digits (0-9), signed or unsigned.

Example

Coded Value	Generated Value (Octal)	Comment
-8	777770	two's complement
256	000400	

1.3.3.2 **Expressions** – Expressions are strings of symbols and numbers separated by arithmetic or Boolean operators. Expressions represent unsigned numeric values ranging from 0 to 2^{18} -1. All arithmetic is performed in unsigned integer arithmetic (two's complement), modulo 2^{18}. Division by zero is regarded as division by one and results in the original dividend; this condition will not be regarded as an error. Fractional remainders are ignored. The value of an expression is calculated by substituting the numeric value for each element (symbol) of the expression and performing the specified operations.

The following are allowable operators to be used with expressions:

Character		Function
Name	Symbol	
Plus	+	Addition (two's complement)
Minus	-	Subtraction (convert to two's complement and add)
Asterisk	*	Multiplication (unsigned)
Slash	/	Division (unsigned)

*Initiated by usage of .DEC pseudo-op (see Section 1.4.4).

Character		Function	
Name	Symbol		
Ampersand	&	Logical AND	Boolean
Exclamation point	!	Inclusive OR	
Back slash	\	Exclusive OR	
Comma	,		

Operations are performed from left to right (i.e., in the order in which they are encountered). For example, the assembly language statement A + B*C + D/E - F * G is equivalent to the following algebraic expression

(((((A + B)* C) + D)/E) - F)* G.

Example

Assume the following symbol values:

Symbol	Value (Octal)
A	000002
B	000010
C	000003
D	000005

The following expressions would be evaluated according to the rules above.

Expression	Evaluation (Octal)	
A+B -C	000007	
A/B+A * C	000006	(The remainder of A/B is lost)
B/A -2* A -1	000003	
A & B	000000	
C+A & D	000005	
B * D/A	000024	
B*C/A*D	000074	

The expression (A + B) * (C + D) cannot be represented in one source statement. To circumvent this problem, at least one of the members of the expression should be defined by a direct assignment statement (see Section 1.3.2.2).

Examples

```
        E = A + B↵

        C + D *E                    /Represents (A + B) * (C + D)↵
```

or

```
        E = A + B↵

        F = C + D↵

        E * F                       /Represents (A + B) * (C + D)↵
```

1.3.4 **Address Assignments**

As source program statements are processed, the Assembler assigns consecutive memory locations to the storage words of the object program. This is done by reference to the Location Counter, which is initially set to zero, and incremented by one each time a storage word is formed in the object program. Some statements, such as machine instructions, cause only one storage word to be generated, incrementing the Location Counter by one. Other statements, such as those used to enter data or text, or to reserve blocks of storage words, cause the Location Counter to be incremented by the number of storage words generated.

1.3.4.1 **Referencing the Location Counter** – The programmer may directly reference the Location Counter by using the period symbol (.) in the address field. He can write

```
        →| JMP ␣ .-1↵
```

which will cause the program to jump to the storage word the address of which was previously assigned by the Location Counter. The Location Counter can be set to another value by using the .LOC pseudo-op, as described in 1.4.8.

1.3.4.2 **Direct Addressing** – Direct Addressing occurs when bits 4 and 5 of the Memory Reference Instruction word (see Figure 1-1) are set to 0. The machine action defined by the operation code field is applied to the operand specified by the 12-bit address field. The 12-bit address field allows addressing of any location of the 4,096 word memory page. Access to locations outside the current page is gained via execution of indexed or indirect address instructions.

1.3.4.3 **Indirect Addressing** – To specify an indirect address, which can be used only in memory reference instructions, the programmer writes an asterisk (*) immediately following the operation field symbol. This sets the Defer Bit (bit 4) for the storage word (see Figure 1-1) and the contents of the address field point to a location (within the current page) which contains the 15-bit effective address.

The error flag S results if an asterisk (*) suffixes a non-memory reference instruction.

Two examples of legal indirect addressing are as follows:

```
→| TAD* →| A↵
→| LAC* →| B↵
```

The following example is illegal.

→| CLA*↵ Indirect addressing cannot be specified in non-memory reference instructions.

NOTE

A symbol followed by an asterisk and by another symbol is interpreted as a multiplication operation (e.g., LAC*SYMBOL).

0 3	4	5	6 17
Operation Code	*	X	Address Field

X= Indexed Address Indicator

*= Indirect Address Indicator

Figure 1-1. Memory Reference Instruction Format

1.3.4.4 **Indexed Addressing** – Indexed Addressing is specified by bit 5 (indirect bit) of the Memory Reference Instruction word (see Figure 1-1) being set to 1. Indexed Addressing can be either Diiect or Indirect. For Direct Indexed Addressing, bit 4 is 0 and bit 5 is 1. In this mode, the effective address is taken as the sum of the 12-bit address field (bits 6-17) and the contents of the Index Register. The Index Register is an 18-bit hardware register which may contain a signed 2's complement number.

Indirect Indexed Addressing is specified by bits 4 and 5 being set to 1. In this mode of addressing, the effective address (EFA) is taken as the contents of the *location specified* by the sum of the 12-bit address field and the Index Register.

Example

```
                      LAC*  →| 30,X
                              /C(30) + C(XREG) = EFA
Contents                      /37 + 20 = EFA
 C(30) = 37                   /57 = EFA
 C(XREG) = 20                 /C(57) = AC
 C(57) = 101                  /101 = AC
```

Using a comma X(,X) in the address field causes bit 5 of the current address value to be exclusively ORed with 10000. For example, if bit 5 of thc previous address value was 1, this operation sets it to 0. The Assembler takes this into consideration when the operation field value and the address field value are combined (see 1.3.6.2).

In all cases when X is used, the value 10000 is used to perform the operation specified.* Standard usage of X is shown below.

Example

```
                              A = 50
                                 LAC  →| A,X

                      Storage word generated   210050
```

Using X to denote Index Register usage causes the following restrictions:

a. X cannot be used as a TAG.

b. X may not be used more than once in an expression.

c. X can only appear in an address field.

d. X cannot be used with a .DSA statement.

1.3.4.5 **Literals** – Symbolic data references in the operation and address fields may be replaced with direct representation of the data enclosed in parentheses**. This inserted data is called a literal. The Assembler sets up the address link, therefore one less statement is necessary in the source program.

The following examples show how literals may be used, and their equivalent statements. The information contained within the parentheses, whether a number, symbol, expression, or machine instruction, is assembled and assigned consecutive memory locations after the highest location used by the program (including registers reserved for undefined symbols). The address of the generated word will appear in the statement that referenced the literal. Duplicate literals are stored only once so that many uses of the same literal in a given program result in only one memory location being allocated for that literal. Nested literals are not allowed and result in an L error diagnostic at assembly time.

*The symbol X should only be used for indexing purposes. After the expression has been evaluated, bit 5 will be set to 1 regardless of the expression result.

**The opening parenthesis [( ] is mandatory while the closing parenthesis [ )] is optional. The operation field is assumed to be to the right of the opening parenthesis [( ] unless it is followed by a space or tab, in which case the address field is assumed.

Usage of Literal	Equivalent Statements
⇥ ADD ␣ (1)	⇥ ADD ␣ ONE ONE ⇥ 1
⇥ LAC ␣ (TAG	⇥ LAC ␣ TAGAD TAGAD ⇥ TAG
⇥ LAC ␣ (DAC ⇥ TAG)	⇥ LAC ␣ INST INST ⇥ DAC ⇥ TAG
⇥ LAC ␣ (JMP ⇥ .+2)	HERE ⇥ LAC ␣ INST INST ⇥ JMP ␣ HERE+2

The following sample program illustrates how the Assembler handles literals.

Location Counter	Source Statement	Generated Code
	⇥ .LOC ␣ 100	
100	TAG1 ⇥ LAC ␣ (100)	200110
101	⇥ DAC ␣ 100	040100
102	⇥ LAC ␣ (JMP ␣ .+5)	200111
103	⇥ LAC ␣ (TAG1)	200110
104	⇥ LAC ␣ (JMP ␣ TAG1)	200112
105	⇥ LAC ␣ (JMP ␣ TAG2)	200112
	TAG2=TAG1	
106	⇥ LAC ␣ (JMP)	200113
107	DAC ⇥ LAC ␣ (DAC ⇥ DAC)	200114
	⇥ .END	

Location Counter	Source Statement	Generated Code
	Generated Literals	
110		000100
111		600107
112		600100
113		600000
114		040107

1.3.5 Statement Fields

1.3.5.1 **Label Field** – If the user wishes to assign a symbolic label to a statement, to facilitate references to the storage word generated by the Assembler, he may do so by beginning the source statement with any desired symbol. The symbol must be terminated by a space (spaces) or tab, or a statement terminating semicolon, or carriage return.

Examples

TAG ␣ any value↵

TAG ␣ any value↵

TAG ⇥ any value↵

TAG;

TAG↵

These examples are equivalent to coding

TAG ⇥ 0↵

in that a word of all 0s will be output with the symbol TAG associated with it.

Symbols used as labels are defined in the user's symbol table with a numerical value equal to the present value of the Location Counter. A label is defined only once; if it was previously defined by the user, the current definition of the symbol will be flagged as a multiple definition error. All references to a multiply-defined symbol will be made to the first value encountered by the Assembler.

Example

Location Counter	Statement	Storage Word Generated	Notes
100	A ⇥ LAC ⇥ B	200103	error, multiple definition first value of A referenced
101	A ⇥ LAC ⇥ C	200104	
102	⇥ LAC ⇥ A	200100	

Location Counter	Statement	Storage Word Generated	Notes
103	B →| 0	000000	
104	C →| 0	000000	

Anything more than a single symbol to the left of the label-field delimiter is an error; it will be flagged and ignored. The following statements are illegal.

TAG+1 →| LAS↵
LOC* 2 →| RAR↵

Redefinition of certain symbols can be accomplished by using direct assignments; that is, the value of a symbol can be modified. If an Assembler permanent symbol or user symbol (which was defined by a direct assignment) is redefined, the value of the symbol can be changed without causing an error message. If a user symbol, which was first defined as a label, is redefined by either a direct assignment or by using it again in the label field, it will cause an error.

Example

Coding	Generated Value (Octal)	Comments
A=3		sets current value of A to 3
→| LAC →| A	200003	
→| DAC →| A	040003	
A=4		redefines value of A to 4
→| LAC →| A	200004	
* B →| DAC →| A	040004	
B=A		illegal usage; a label cannot be redefined
→| DAC →| B	040105	
PSF=700201		to redefine possibly incorrect permanent symbol definition.

*Assume that this instruction will occupy location 105.

1.3.5.2 **Operation Field** – Whether or not a symbol label is associated with the statement, the operation field must be delimited on its left by a space(s) or tab. If it is not delimited on its left, it will be interpreted as the label field. The operation field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic modulo 2^{18}. In the operation field, machine instruction op-codes and pseudo-op mnemonic symbols take precedence over identically named user defined symbols. The operation field must be terminated by one of the following characters:

→| or ␣ (s)	field delimiters
↲ or ;	statement delimiters

Examples

```
TAG  →| ISZ↲
     →| .+3 ␣ ↲
   ␣ CMA!CML↲
     →| TAG/5+TAG2;  →| TAG3↲
```

1.3.5.3 **Address Field** – The address field, if used in a statement, must be separated from the operation field by a tab, or space(s). The address field may contain any symbol, number, or expression which will be evaluated as an 18-bit quantity using unsigned arithmetic, modulo 2^{18}. If op-code or pseudo-op code symbols are used in the address field, they must be user defined, otherwise they will be undefined to the Assembler and cause an error message. The symbol X *cannot* be user defined (see section 1.3.4.4). The address field must be terminated by one of the following characters:

→| or ␣ (s)	field delimiters
↲ or ;	statement delimiters

Examples

```
     TAG2  →| DAC  →| .+3  →| /COMMENT↲

           →|       →| TAG2/5 + 3 ␣ (s)↲

           →| ISZ  →| TAG2, X + 4↲

→| JMP  →| BEGIN↲

→| TAD  →| A;  →| DAC  →| B↲
```

In the last example, a tab or space(s) is required after the semicolon so the Assembler can interpret DAC as being the operation field rather than the label field.

An error condition will exist if the bank and page bits (3, 4 & 5) of the address do not match the bank and page bits of the bank currently being assembled into and the extended memory bits of the address are not zero.

Examples

Location (octal)	Instruction	Comments
1000	→| LAC ␣ 100	will not cause error messages
1001	→| DAC ␣ 101	

Location (octal)	Instruction	Comments
1002	⇥ JMS ␣ 250	will not cause error messages
1003	⇥ ISZ ␣ 40146	will cause a bank error message code B (see 1.3.5.3)
1004	⇥ LAC ␣ 10100	will cause a bank error message code (see 1.3.5.3)

1.3.5.4 **Comments Field** – Comments may appear anywhere in a statement. They must begin with a slash (/) immediately preceded by a

␣ (s)	space(s)
⇥	tab
↵	carriage return (end of previous line)
;	semicolon

Comments are terminated only by a carriage return.

Examples

```
␣ (s) / THIS IS A COMMENT↵

TAG1 ⇥ LAC ␣ /after the ; is still a comment↵

/THIS IS A COMMENT↵

 ⇥ RTR ␣ /COMMENT↵

 ⇥ RTR; ⇥ RTR; /THIS IS A COMMENT↵
```

Observe that ⇥ A/COMMENT↵ is not a comment, but rather an operation field expression. A line that is completely blank; that is, there is no data between two sets of ↵ (s), will be treated as a comment by the Assembler.

1.3.6 Statement Evaluation

When the Assembler evaluates a statement, it examines the first character position in the label, operation, and address fields for a numeric character. (Comments fields are ignored.) If the first character of a field is numeric, the contents of the whole field is treated as a number.

1.3.6.1 **Numbers** – Numbers are not field dependent. When the Assembler encounters a number (or expression) in the operation or address fields (a number in the first character position of the label field is illegal), it uses those values to form the storage word. The following statements are equivalent:

```
⇥ 200000 ␣ 10↵

⇥ 10 + LAC ↵

⇥ LAC ␣ 10↵
```

All three statements cause the Assembler to generate a storage word containing 200010.

A statement can consist of a number or expression which generates a single 18-bit storage word; for example,

```
⇥ 23; ␣ 45; ␣ 357; ␣ 62↵
```

This group of four statements generates four words interpreted under the current radix.

Zero words are generated by statements containing only labels. For example,

```
A; B; C; D; E↵
```

generates five words set to zero, which can be referenced by the labels defined.

1.3.6.2 **Word Evaluation** – When the Assembler encounters a symbol in a statement field, it determines the value of the symbol by referring to the user's symbol table and the permanent symbol table, according to the priority list shown below. The value of a storage word is computed by combining the 18-bit operation field quantity with the 18-bit address field quantity, in the following manner.

```
[(OPERATION FIELD + (ADDRESS FIELD & 017777))] = Value of Word
        0-17                0-17
```

If the operation code is one of the 9-bit operators (see Appendix B), the word value is computed as follows.

```
[(OPERATION FIELD + (ADDRESS FIELD & 777))⊨ Value of Word
        0-17                0-17
```

Extensive error checking is performed on the address field value to ensure correct results. If the page bit of the address field is different from that of the program counter, the line is flagged with a 'B' error code at assembly time, indicating a page error. Page bits are always set to 0 if the address value is legal and Index Register usage is not specified.

Example:

Error Flag	LOC	Object Code		Source Code
				.LOC 100
B	00100	210000	A	LAC B /ADDRESS OF B ON A DIFFERENT PAGE
				.LOC 10000
B	10000	040100	B	DAC A /ADDRESS OF A ON A DIFFERENT PAGE

The LAW instruction and the 9-bit operators check the low order 5 bits (or 9 bits) of the address value for validity by ANDing off the low order bits and checking the result for 0 or for equality with the 'AND' value.

[address field & 760000] = 760000 or 000000

OR

IF A 9 BIT OPERATOR

[address field & 777000] = 777000 or 000000

Example

```
        LAW -1              /The value for -1 is 777777.
                            /When this value is ANDed
                            /with 760000, the result
                            /is 760000 and valid.
```

If the ADDRESS FIELD and 760000_8 does not equal 760000_8 or 000000, any erroneous results produced are flagged by the Assembler. This validity check is performed only if an operation field and an address field are present.

If the ADDRESS FIELD ANDed with 760000_8 does not equal 760000_8 or 000000, any erroneous results produced are flagged by the Assembler. This validity check is performed only if an operation field and an address field are present.

If the instruction is a Memory Reference instruction, the low order 5 bits of the address value are examined to make sure they are not set*. If any of the 5 bits are set, the line is flagged with an E error code during assembly.

```
        →| 2  →| 5  →| /GENERATES 000007
```

The value of a symbol depends on whether it is in the label field, or the address field. The Assembler attempts to evaluate each symbol by running down a priority list, depending on its field as shown.

Label Field	**Operation Field**	**Address Field**
Current value of Location Counter	1. Pseudo-op	1. User symbol table, (including direct assignments)
	2. Direct assignment in user symbol table.	2. Undefined
	3. Permanent symbol table	
	4. User symbol table	
	5. Undefined	

This means that if a symbol is used in the address field, it must be defined in the user's symbol table; otherwise, it is undefined.

In the operation field, pseudo-ops take precedence and cannot be redefined. Direct assignments allow the user to redefine machine op codes, as shown in the example below.

*Does not include LAW instruction.

Example

```
        DPOSIT = DAC↵
```

The user can use machine instruction codes and Assembler pseudo-op codes in the label field and refer to them later in the address field.

1.4 PSEUDO OPERATIONS

In the discussion of symbols in the previous chapter, it was stated that the Assembler has, in its permanent symbol table, definitions of the symbols for all the PDP-15 memory reference instructions, operate instructions, and some IOT instructions (listed in Appendix B) which may be used in the operation field without prior definition by the user. Also contained in the permanent symbol table are a class of symbols called pseudo-operations (pseudo-ops) which, instead of generating instructions or data, direct the assembler on how to proceed with the assembly.

By convention, the first character of every pseudo-op symbol is a period (.). This convention is used to prevent the programmer from inadvertently using, in the operation field, a pseudo-instruction symbol as one of his own. *Pseudo-ops may be used only in the operation field.*

1.4.1 **Program Control (.END)**

One pseudo-op that must be included in every source program is .END, which must be the last statement in the program. This statement marks the physical end of the source program and may also contain the location of the first instruction in the object program to be executed at run-time.

The .END statement is written in the general form

```
        ⇥.END ␣ START↵
```

where START may be a symbol, number, or expression whose value is the address of the first program instruction to be executed.

If a starting address is not present in an .END statement, the program will halt after being loaded and the user must manually start his program.

The following are legal .END statements:

```
        ⇥.END ␣ BEGIN + 5↵

        ⇥.END ␣ 200↵

        ⇥.END↵
```

1.4.2 **Program Segments (.EOT)**

If the input source program is physically segmented, each segment except the last must terminate with an .EOT (end-of-tape) statement. The last segment must terminate with an .END statement. For example, if the input source program is prepared on three different tapes, the first two are terminated by .EOT statements, and the last by an .END statement. The .EOT statement is written without label and address field, as follows.

```
        ⇥.EOT↵
```

1.4.3 **Reserving Storage Words In The Object Program**

The programmer may reserve blocks of storage words, or single words, for use during program execution.

1.4.3.1 **Reserving Blocks of Storage (.BLOCK)** – .BLOCK reserves a block of memory equal to the value of the expression contained in the address field. If the address field contains a numerical value, it will be evaluated according to the radix in effect. The symbolic elements of the expression must have been defined previously; otherwise, phase errors (see Section 1.5.5) might occur in PASS2.

The user may reference the first location in the block of reserved memory by defining a symbol in the label field. The initial contents of the reserved locations are unspecified.

Label Field	Operation Field	Address Field
User Symbol	.BLOCK	Predefined Expression

Examples

```
    BUFF ⇥.BLOCK ␣ 12
         ⇥.BLOCK ␣ A-B+65↵
```

1.4.3.2 **Reserving Single Storage Words** – Storage words set to zero are set up as

```
    A ⇥0; ⇥0; ⇥0↵
```

In this way, three words are set to zero, starting at A. Storage words set to zero are also set up by statements containing only labels:

```
    A; B; C; D; E↵
```

1.4.4 **Radix Control (.OCT and .DEC)**

The initial radix (base) used in all number interpretation by the Assembler is octal (base 8). In order to allow the user to express decimal values, and then restore to octal values, two radix setting pseudo-ops are provided.

Pseudo-op Code	Meaning
.OCT	Interpret all succeeding numerical values in base 8 (octal)
.DEC	Interpret all succeeding numerical values in base 10 (decimal)

These pseudo-instructions must be coded in the operation field of a statement. All numbers are decoded in the current radix until a new radix control pseudo-instruction is encountered. The programmer may change the radix at any point in a program.

Source Program		Generated Value (Octal)	Radix in Effect	
⇥ LAC	⇥ 100	200100	8	initial value is assumed to be octal
⇥ 25		000025	8	
⇥ .DEC				
⇥ LAC	⇥ 100	200144	10	
⇥ 275		000423	10	
⇥ .OCT				
⇥ 76		000076	8	
⇥ 85		000000	error	

1.4.5 Text Handling (.SIXBT)

Label Field	Operation Field	Address Field		
SYMBOL	.SIXBT	delimiter	character string	delimiter

.SIXBT denotes 6-bit trimmed ASCII characters, which are formed by truncating the leftmost bit of the corresponding 7-bit ASCII character. The Assembler converts the characters to their appropriate numerical equivalent (see Appendix A). The characters are packed, three per word left justified, with unused bits set to zero.

0 5	6 11	12 17
1st character	2nd character	3rd character

Spaces or tabs prior to the text delimiter are ignored. Any printing character may be used as the text delimiter, except ↵ . The text delimiter must be present on both ends of the text string, otherwise, the user may get more characters than desired; however, ↵ may be used to terminate the test string. After a carriage return, a new .SIXBT pseudo-op is necessary to continue with 6-bit text.

Examples

Source	Generated Code
⇥.SIXBT ⇥ /ABCDE/↵	010203
	040500
TAG ⇥.SIXBT ⇥ -/125/-↵	576162
	655700
⇥.SIXBT ⇥ /ABCD↵	010203
	040000

1.4.6 **Defining A Symbolic Address (.DSA)**

.DSA (define symbol address) is used in the operation field when it is desired to create a word composed only of an address field. It is especially useful when a user symbol is also an instruction or pseudo-op symbol.

Label Field	Operation Field	Address Field
User Symbol	.DSA	Any expression

Examples

```
JMP ⇥ LAC ⇥ TAG

    ⇥ .DSA ⇥ JMP

    ⇥       ⇥ JMP
```

Equivalent methods of defining the user symbol JMP to be in the address field.

1.4.7 **Conditional Assembly (.IFDEF, .IFUND and .ENDC)**

It is often useful to assemble some parts of the source program on an optional basis. This is done by means of conditional assembly statements of the form:

```
⇥ .IF... ⇥ expression
```

The pseudo-op may be one of the two conditional pseudo-ops shown below, and the address field may contain any symbol or expression. If the condition is satisfied, that part of the source program starting with the statement immediately following the conditional statement and up to the .ENDC (end conditional) pseudo-op is assembled. If the condition is not satisfied, this coding is not assembled (it is treated as comments).

The two conditional pseudo-ops (sometimes called IF statements) and their meanings are shown below.

Conditional Pseudo-op	Assemble IF x is:
.IFDEF x	defined (present) in user symbol table
.IFUND x	undefined (not present) in user's symbol table

In the following sequence, the pseudo-op .IFDEF is satisfied, and the source program coding between .IFDEF and ENDC is assembled.

```
Y = 5↵
 ⇥ .IFDEF ⇥ Y↵
 ⇥ LAC ␣ A↵
 ⇥ DAC ␣ B↵
 ⇥ .ENDC↵
```

Conditional statements may be nested. For each IF statement there must be a terminating .ENDC statement. If the outermost IF statement is not satisfied, the entire group is not assembled. If the first IF is satisfied, the succeeding coding is assembled. If another IF is encountered, however, its condition is tested, and the succeeding coding is assembled only if the second IF statement is satisfied. Logically, nested IF statements are like AND circuits. If the first, second and third conditions are satisifed, then the coding that follows the third nested IF statement is assembled.

Example

```
⇥ .IFDEF ␣ W↵            conditional 1 initiator
⇥ LAC  ⇥ TAG↵
⇥ .IFUND ␣ Y↵            conditional 2 initiator
⇥ DAC  ⇥ TAG1↵
⇥ .ENDC↵                 conditional 2 terminator
⇥ .IFDEF ␣ Z↵            conditional 3 initiator
⇥ DAC  ⇥ TAG2↵
⇥ .ENDC↵                 conditional 3 terminator
⇥ .ENDC↵                 conditional 1 terminator
```

IF statements may be activated by means of a strip of tape which precedes the regular source tapes. The strip tape may take on the following form:

Strip Tape

```
A = 0                    /Causes A and B to be
B = 0                    /entered into the user's symbol table.
 ⇥ .EOT                  /Terminates the strip tape
```

Source Program

```
→| .IFDEF  →| A

        .

        .                          /This coding would be assembled.

        .

→| .ENDC

→| .IFUND  →| B

        .

        .                          /This coding would not be assembled.

        .

   →| .ENDC

   →| .END
```

Since the purpose of the strip tape is to enter definitions into the user's symbol table, it is only necessary to have the strip tape read in PASS1 of assembly.

1.4.8 Setting The Location Counter (.LOC)

Label Field	Operation Field	Address Field
Not Used	.LOC	Predefined symbolic expression, or number

The .LOC pseudo-op sets or resets the Location Counter to the value of the expression contained in the address field. The symbolic elements of the expression must have been defined previously; otherwise, phase errors might occur in PASS2. The .LOC pseudo-op may be used anywhere and as many times as required. If the .LOC pseudo-op is not used, the assembler assumes location 0 as the starting point of the program.

Examples

Location Counter	Instruction
100	→| .LOC ␣ 100
100	→| LAC ␣ TAG1

Location Counter	Instruction
101	→| DAC ␣ TAG2
102	→| LOC ␣
102	A →| LAC ␣ B
103	→| DAC ␣ C
107	→| .LOC ␣ A+5
107	→| LAC ␣ C
110	→| DAC ␣ D
111	→| LAC ␣ E
112	→| DAC ␣ F

1.4.9 Listing Control (.XLIST and .LIST)

The following assembler listing controls are effective only when a listing was requested in the command string (Section 1.5.2).

The .XLIST statement causes the assembler to stop listing the assembled program. The listing printout actually starts at the beginning of PASS2; therefore, to suppress all of the program listing, .XLIST must be the first statement in the program. If only a part of the program listing is to be suppressed, the .XLIST statement can be inserted at any point to stop listing from that point.

The .LIST statement, which is normally used following an .XLIST statement, causes the assembler to resume the listing at the point at which it is encountered.

1.4.10 Object Output Control (.FULL)

Label Field	Operation Field	Address Field
Not Used	.FULL	Not Used

The .FULL pseudo-op causes hardware readin-mode binary output to be produced (see Section 1.5.4.3 for a description of the normal binary output). It must appear before any coding (except comments), otherwise, it will be flagged and ignored. The program is assembled as unchecksummed binary code and each physical record of output contains nothing other than 18-bit binary storage words generated by the Assembler. The Assembler will cause the address of the .END statement to contain a punch in channel 7, thereby allowing the output to be loaded via hardware readin mode. If no address is specified in the .END statement, a halt (rather than a jump) will be output as the last word.

The following specific restrictions apply to programs assembled in .FULL mode output

.LOC	Should be used only at the beginning of the program.
.BLOCK	May be used only if no literals appear in the program, and must immediately precede .END.
	Undefined symbols may be used if no literals appear in the program.
	Literals may be used only if the program has no undefined symbols.

1.4.11 **Size of Program (.SIZE)**

When the Assembler encounters .SIZE, it outputs, at that point, the address of the last location plus one occupied by the object programs. This is normally the length of the object program (in octal).

Label Field	Operation Field	Address Field
User Symbol	.SIZE	Not Used

Example

```
Generated Code      Source Code

   00100                                   →| .LOC   →| 100↵
   00100              000105               →| .SIZE↵
   00101              200103               →| LAC    →| A↵
   00102              040104               →| DAC    →| B↵
   00103              000000            A  →| 0↵
   00104              000000            B  →| 0↵
                      000000               →| .END↵
```

1.5 OPERATING PROCEDURES

1.5.1 **Loading Procedures**

The loading procedure depends on whether the Teletype paper tape reader or the high-speed paper tape reader is used as the loading device.

1.5.1.1 **Teletype Reader as Loading Device** – Place paper tape of the hardware readin low-speed binary loader (see Appendix D) into the teletype reader. Engage the start switch on the reader. Enter 7700 into the ADDRESS switches (17700 if computer has 8K of memory). Press I/O RESET and then press READIN. When the computer halts (AC=777777), disengage the start switch, place the Assembler binary tape into keyboard reader, engage the start switch, and press START.

1.5.1.2 **High-Speed Reader as Loading Device** – Place paper tape of the Hardware Readin High-speed Binary Loader (see Appendix D) into the high-speed reader. Enter 7720 into the ADDRESS switches (17720 if computer has 8K of memory). Press I/O RESET and then press READIN. When computer halts (AC = 777777) place the Assembler binary tape into high-speed reader and press START.

1.5.2 **COMMAND String**

After the Assembler has been loaded into memory, it will type out a sequence of messages. The user's responses to these messages indicate the options and devices that are to be used for the current assembly.

1.5.2.1 **Binary Option** – The first message typed is *BIN -. If no binary output is desired, the user responds with carriage return (↵).

If binary output is desired, the user types L↵ if the binary device is the low-speed punch, or H↵ if the binary device is the high-speed punch. If any other character is typed it is ignored and the message (*BIN -) is repeated.*

1.5.2.2 **Listing Option** – The second message typed is *LST-. If no listing is desired, the user responds with ↵. If a listing is desired, the user types L↵ to produce the listing on the teleprinter, or H↵ to produce the listing on the high-speed punch. If any other character is typed, it is ignored and the message (*LST-) is repeated.*

1.5.2.3 **Symbol Table Option** – The third message typed is *SMB-. If no symbol table output is desired, the user responds with↵ . If symbol table output is desired, the user types L↵ to produce the symbol table on the teleprinter, or types H↵ to produce the symbol table on the high-speed punch. If any other character is typed it is ignored and the message (*SMB-) is repeated.*

1.5.2.4 **Source Input Device** – The last message typed is *SRC-. If the source program is to be read from the keyboard reader, the user types L ↵. If it is to be read from the high-speed reader, the user responds with H↵. If H or L is not specified, or if any other character is typed, it is ignored and the message (*SRC-) is repeated.* When the carriage return is typed, the Assembler starts reading from the input device.

1.5.2.5 **Error Printout** – Normally, all errors that are encountered by the Assembler are output to the listing device and to the Teletype, if it is not the listing device. If no error printout is desired on the Teletype, it can be suppressed by typing N in addition to the normally typed character on any of the four command string messages prior to typing the carriage return.

*Note, that the character just prior to the carriage return (↵) is the one accepted by the Assembler, and therefore, if it is desired to change it, it may be retyped before the↵ is typed.

Examples

(All characters underlined are typed by the Assembler)

*BIN - H↲	Binary on high-speed punch.
*LST - N↲	No listing, and suppress error printout on Teletype.
*SMB - L↲	Symbol printed on Teletype.
*SRC - H↲	Source input from high-speed reader.
*BIN - F↲	Illegal request, message repeated.
*BIN - L↲	Binary output on Teletype punch.
*LST - L↲	Listing on Teletype.
*SMB - L↲	Symbol table printout on Teletype.
*SRC - L↲	Source input from keyboard reader.

NOTE

Normally, the Assembler requires two passes to assemble any program: however, if the binary output device and the listing device are the same device, then three passes are required. PASS2 produces the listing and PASS3 produces the binary object code. PASS2 always produces the symbol table, if requested. When assembly is completed, the Assembler returns to the command string processor for additional assemblies.

1.5.3 **Continuation and Termination Control**

Two control characters are available to control Assembler processing: CTRL C and CTRL P. They are typed by holding the CTRL key and striking either C or P. The Assembler echoes these characters as ↑C or ↑P.

CTRL C may be typed when it is desired to prematurely terminate assembly and return to the beginning of assembly. The Assembler echoes ↑C and returns to the command string processor. CTRL C may also be used to return to the beginning of the command string if it is desired to change the options before starting the assembly. (Note that the keyboard reader start switch must be disengaged before typing CTRL C.)

Examples

(All characters underlined are typed by the Assembler)

```
TAG1    00045:00200 ↑C

*BIN-
```

A multiple definition error occurred and the user prematurely terminated the assembly by typing CTRL C.

```
*BIN-↵
*LST- H↵
*SMB- ↑C↵
*BIN- L↵
```

CTRL C was typed to change the binary request from no binary output to binary output on the Teletype.

Two pseudo-ops control the termination of a segment or total source input to assembly. They are .EOT and .END (see Sections 3.1 and 3.2).

When .EOT is encountered by the Assembler it outputs EOT ↑P on the Teletype. It then waits for the user to load the next tape into the tape reader and type CTRL P to continue.

When .END is encountered and another pass is required, the Assembler types END OF PASS ↑P. When the tape is reloaded in the reader for the next pass, the user types CTRL P to proceed.

1.5.4 **Assembly Output**

1.5.4.1 **Symbolic Listing** – If the user requests a symbolic listing, via the command string, the Assembler will produce an output listing on the requested output device. (Teletype or high-speed punch)

The body of the listing will be formatted as follows.

```
Error Flags     Location     Object Code     Source Statement

   xxx           xxxxx         xxxxxx          x           x
```

where

Error Flags = Errors encountered by the Assembler (see Section 1.5.5)

Location = Location assigned to the binary code.

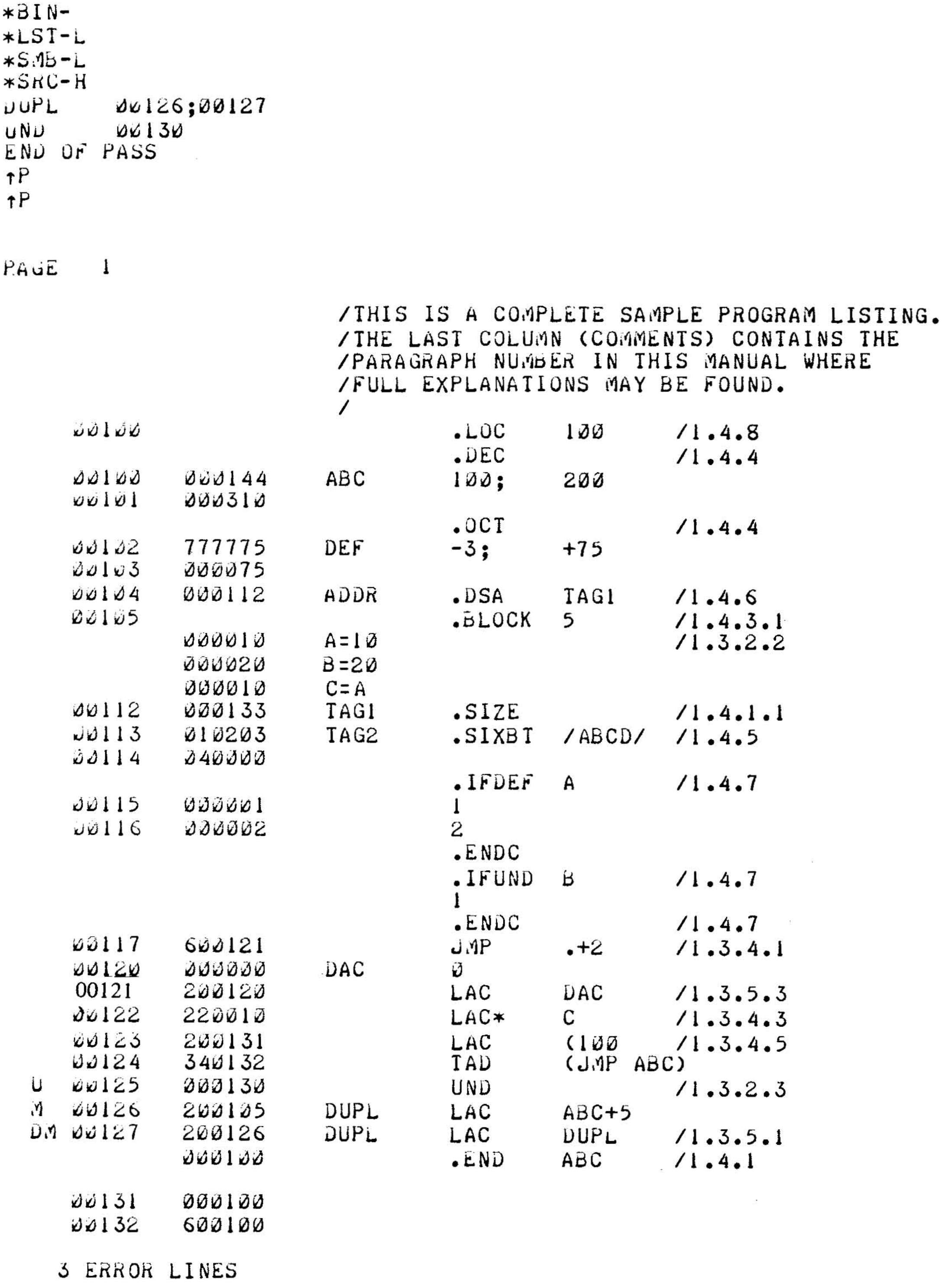
Object Code = The contents of the location (in octal)

NOTE

> Locations and object codes assigned for literals are listed following the program.

When 56 lines have been encountered or when a form feed is encountered, the Assembler precedes the following output with a new page number. In the case of a form feed the Assembler also outputs three up arrows (↑↑↑)

preceding the new page number, to indicate that a form feed caused the new page. At the end of the assembly listing will be an error line count indicating any errors encountered during assembly.



```
*BIN-
*LST-L
*SMB-L
*SRC-H
DUPL      ØØ126;ØØ127
UND       ØØ13Ø
END OF PASS
↑P
↑P


PAGE    1

                              /THIS IS A COMPLETE SAMPLE PROGRAM LISTING.
                              /THE LAST COLUMN (COMMENTS) CONTAINS THE
                              /PARAGRAPH NUMBER IN THIS MANUAL WHERE
                              /FULL EXPLANATIONS MAY BE FOUND.
                              /
    ØØ1ØØ                               .LOC      1ØØ      /1.4.8
                                        .DEC               /1.4.4
    ØØ1ØØ   ØØØ144       ABC            1ØØ;      2ØØ
    ØØ1Ø1   ØØØ31Ø
                                        .OCT               /1.4.4
    ØØ1Ø2   777775       DEF            -3;       +75
    ØØ1Ø3   ØØØØ75
    ØØ1Ø4   ØØØ112       ADDR           .DSA      TAG1     /1.4.6
    ØØ1Ø5                               .BLOCK    5        /1.4.3.1
            ØØØØ1Ø       A=1Ø                              /1.3.2.2
            ØØØØ2Ø       B=2Ø
            ØØØØ1Ø       C=A
    ØØ112   ØØØ133       TAG1           .SIZE              /1.4.1.1
    ØØ113   Ø1Ø2Ø3       TAG2           .SIXBT    /ABCD/   /1.4.5
    ØØ114   Ø4ØØØØ
                                        .IFDEF    A        /1.4.7
    ØØ115   ØØØØØ1                      1
    ØØ116   ØØØØØ2                      2
                                        .ENDC
                                        .IFUND    B        /1.4.7
                                        1
                                        .ENDC              /1.4.7
    ØØ117   6ØØ121                      JMP       .+2      /1.3.4.1
    ØØ12Ø   ØØØØØØ       DAC            Ø
    00121   2ØØ12Ø                      LAC       DAC      /1.3.5.3
    ØØ122   22ØØ1Ø                      LAC*      C        /1.3.4.3
    ØØ123   2ØØ131                      LAC       (1ØØ     /1.3.4.5
    ØØ124   34Ø132                      TAD       (JMP ABC)
U   ØØ125   ØØØ13Ø                      UND                /1.3.2.3
M   ØØ126   2ØØ1Ø5       DUPL           LAC       ABC+5
DM  ØØ127   2ØØ126       DUPL           LAC       DUPL     /1.3.5.1
            ØØØ1ØØ                      .END      ABC      /1.4.1

    ØØ131   ØØØ1ØØ
    ØØ132   6ØØ1ØØ

  3 ERROR LINES
```

1.5.4.2 **Symbol Table Output** – After the assembly listing has been typed, the Assembler will output a symbol table (if requested) which lists all user defined symbols. There will be two symbol lists; the first will be an alphabetically ordered list of the symbols and the second will be a list in numerical value order. The symbol table listing is useful in tracing or debugging a program for which the programmer does not have a complete assembly listing (symbols defined by labels will have a value of 5 octal digits and symbols defined by direct assignments will have a value of 6 octal digits).

Sample Symbol Table Listing

```
PAGE    2

A       000010
ABC      00100
ADDR     00104
B       000020
C       000010
DAC      00120
DEF      00102
DUPL     00126
TAG1     00112
TAG2     00113
UND      00130




PAGE    3

A       000010
C       000010
B       000020
ABC      00100
DEF      00102
ADDR     00104
TAG1     00112
TAG2     00113
DAC      00120
DUPL     00126
UND      00130
```

1.5.4.3 **Object Program Output** – If the user requests binary output, the normal object code produced by the Assembler is a binary paper tape which can be loaded at run time by either of the hardware readin binary loaders (see Appendix D). The format of the binary output is as follows:

Block Heading – (three binary words)

WORD 1 Starting address to load the block body which follows.

WORD 2 Number of words in the block body (two's complement).

WORD 3 Checksum of block body (two's complement). It also includes Word 1 and Word 2 of the block heading.

Block Body – (n binary words)

The block body contains the binary data to be loaded under block heading control.

Starting Block – (two binary words)

WORD 1	Locations to start execution of program. It is distinguished from the block heading by having bit 0 set to 1 (negative).
WORD 2	Dummy word

If the value of the expression of the .END statement is equal to zero, the provided loader halts before transferring control to the object program, thereby allowing manual intervention by the user. (See Section 1.4.10 for an alternate form of binary output.)

1.5.5 **Error Detection and Flagging**

The Assembler examines each source statement for possible errors. The statement which contains the error will be flagged by one or several letters in the left-hand margin of the line. The following table shows the error flags and their meanings.

Flags	Meaning
A	Error in direct symbol table assignments ignored (see Section 2.5.1).
B	Memory bank error (see Section 1.3.5.3).
D	The statement contains a reference to a multiply-defined symbol. It is assembled with the first value defined.
E	Erroneous results may have been produced. Will also occur on undefined .END value (see Section 1.3.5.3).
L	Literal phase error; literal encountered in PASS2 does not equal any literal encountered in PASS1.
M	An attempt is made to define a symbol which has already been defined. The symbol retains its original value.
N	Error in number usage.
O	Operand error. Non-Memory Reference Instruction has an address value.
P	Phase error; PASS1 value does not equal PASS2 value of a symbol. PASS1 value will be used.

Flag	Meaning
Q	Questionable line
S	Symbol error; an illegal character was encountered and ignored.
T	TAG or LABEL error. 1. Unrecognizable character in TAG field. 2. A period used alone in a TAG field. 3. A TAG begins with a number. 4. X is used as a TAG.
U	An undefined symbol was encounterd.
X	Illegal use of Index Register. 1. X occurs in a .DSA statement. 2. X occurs more than once in an expression. 3. X occurs in a TAG or OP-Code field.

In addition to flagging error lines, the Assembler, during PASS1, will print the following conditions.

Condition	Example	
Multiple definitions	ABC	00100; 00125
Direct assignment forward references	A = B	
Undefined symbols	UNDF	06255

The following condition will cause assembly to be terminated prematurely in PASS1.

Message	Cause
TABLE OVERFLOW	Too many symbols and/or literals

1.5.6 Internal Operations

1.5.6.1 **Symbol Table Capacity** – The Assembler occupies approximately 3000_{10} memory locations, leaving about 1090_{10} registers free for symbols and literals. (The Assembler determines the physical size of memory of the computer; therefore, if it is an 8K machine, 5190_{10} registers are available for table space.) Each symbol defined by the user requires three memory locations and each literal requires one memory location. For a 4K PDP-15/10 this means that about 360_{10} symbols (or 300_{10} symbols and 190_{10} literals) may be used before overflow occurs.

1.5.6.2 **Halts** – Normally, the Assembler does not halt for any reason except if an unknown program interrupt occurs. If this happens the machine halts with the status word in the AC. In order to clear the condition the user must deposit, in the location the program counter is pointing to, the instruction to clear the flag that caused the unknown interrupt, and then press CONTINUE.

CHAPTER 2
COMPACT TEXT EDITOR

2.1 INTRODUCTION

The PDP-15/10 Compact Text Editor provides for the creation and/or modification of source programs and other ASCII text material. Commands issued from the Teletype direct the Editor to bring a group of lines from the input device to an internal buffer. The user may then, by means of additional commands, examine, delete, and change the contents of the buffer, and insert new text at any point in the buffer. When a block of lines has been edited, it is punched on the paper tape reader.

Editor operation codes are divided into two basic categories: control instructions and Editor commands. Control instructions determine whether the Editor is to be used to create new ASCII material (input level) or to modify existing text (edit level). Within the edit level there are four Editor command classes: I/O requests, pointer manipulation, editing requests, and examination requests.

The Editor is most frequently used to modify PDP-15/10 source programs, but it also can be used to edit any symbolic text. The Editor operates with either high-speed or low-speed (ASR33) paper tape devices, and occupies approximately 2000_{10} locations of core memory. Any additional memory is used for buffers.

Appendix E provides a concise summary of Editor commands. Appendix F contains a simple procedure for creating ASCII text using the Editor. Appendix G contains examples of Editor operation, written for the user who is not familiar with the Editor, but wishes to edit an ASCII tape immediately. The user should at least read 2.3, Operating Procedures, before attempting to use the Editor as described in Appendix G. A more comprehensive, annotated example of an actual editing session is contained in Appendix H.

2.2 FUNCTIONAL DESCRIPTION

2.2.1 Input Format

The following paragraphs describe the control levels, operation code formats, and data modes for the Editor.

2.2.1.1 **Control Levels** – The PDP-15/10 Compact Editor operates on one of two control levels: input or edit. The input level is used to create new text material; on this level the Editor interprets lines from the Teletype as text to be added to an open block. Instructions are available to conveniently change the control level. The edit level is used to modify existing text; on this level the Editor accepts and acts upon control words and data strings to bring in lines of text, to change, delete, or replace the line currently in the work area, and to insert single or multiple lines after the current line.

2.2.1.2 **Operation Code Format** – The format for all Editor operations consists of the operation code, followed by a ␣ (space), followed by arguments where applicable. The space is a blank delimiter which is considered by the Editor to be a part of the command itself, not part of the argument string which follows the command. Legal abbreviations are indicated with square brackets in this manual. Certain commands (e.g., FIND and RETYPE) require the presence of arguments. Others (DELETE, NEXT) may take explicit arguments at the option of the user. Optional arguments are given in parentheses. For a description of the command language see 2.4.

2.2.1.3 **Data Mode** – The Editor can accept input from a maximum of two devices* in addition to the keyboard. The first device normally holds previously prepared text upon which changes are to be carried out. The second, the subsidiary input device, is usually the medium through which additional, previously prepared text is inserted in the object text.

Data from the input device is made available for editing in block form. A user-specified portion of the input is held in a core buffer for editing until the user requests that the contents of the buffer be added to the output text. Commands to the Editor are performed on that portion of the text currently in the buffer. Lines may be accessed repeatedly until the buffer is emptied by the user. The lines of text in the buffer are made available for modification by manipulating a software pointer (see Figure 2-1).

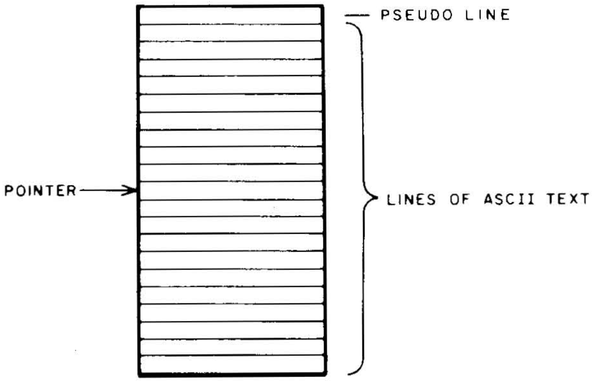


Figure 2-1. Line Buffer and Software Pointer

2.2.2 **Output Format**

The teleprinter is used by the Editor to echo user requests, to make responses to those requests, and to print error messages. Edited text is punched out on either the low-speed (ASR33) or the high-speed paper tape punch (if available and requested by the user during the initialization sequence). Edited source programs are punched in a form that is ready to be read by the Assembler; parity is not punched; channel 8 8 is always punched (see Figure 2-2).

*The low-speed (ASR33) paper tape reader, and a high-speed paper tape reader (if available).

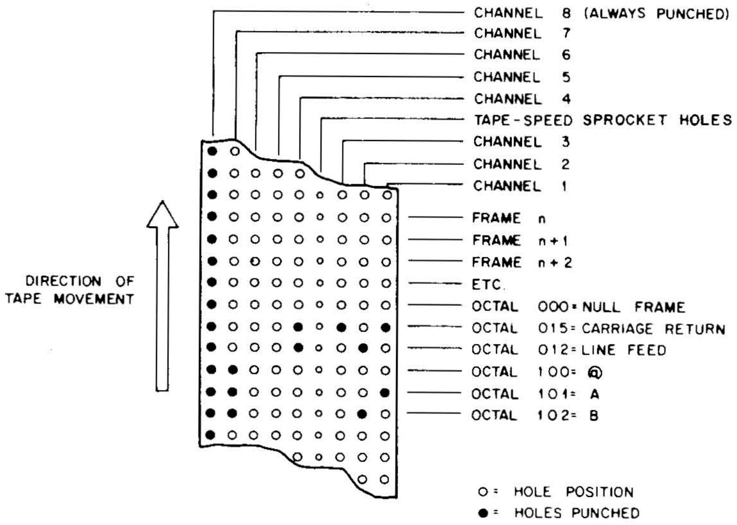


Figure 2-2. PDP-15/10 ASCII Tape Format

2.3 OPERATING PROCEDURES

2.3.1 Loading Procedure

Prior to loading the Editor, proceed as follows to generate leader:

Turn Teletype switch to OFF LINE.

Press punch switch ON.

Press HERE IS key several times to generate leader.

Press punch switch OFF.

Turn Teletype switch to ON LINE.

The loading procedure used depends on whether the low-speed (ASR33) or high-speed paper tape reader is used as the loading device.

2.3.1.1 **Low-Speed Reader** – The low-speed reader loading procedure is as follows:

Place paper tape containing low-speed, hardware readin binary loader in the Teletype reader.

Turn reader switch to ON

Set ADDRESS Switches to 7700 (17700 for 8K systems).

Press I/O RESET and READIN. The computer will halt (AC=777777).

Turn reader switch to OFF and place binary tape of Editor in Teletype reader.

Turn reader switch to ON and press START. The Editor will be loaded into memory and will type EDITOR on the Teletype.

2.3.1.2 **High-Speed Reader** – The high-speed reader loading procedure is as follows:

Place paper tape containing high-speed, hardware readin binary loader in the high-speed paper tape reader.

Set ADDRESS Switches to 7720 (17720 for 8K systems).

Press I/O RESET and READIN. The computer will halt (AC=777777).

Place binary tape of Editor in high-speed reader and press START. The Editor will be loaded into memory and will type EDITOR on the Teletype.

2.3.1.3 **Loader Halts** –

AC = 777777	Program loaded.
AC = non-zero	Checksum error on last block loaded. Reposition tape to blank frame prior to beginning of last block and press START to read again. To ignore error, press CONTINUE.

2.3.2 Initialization

The Editor always begins with control at the edit level and assumes that the user wishes to modify some existing text. When first loaded, or when restarted, the Editor types EDITOR, followed by the I/O initialization request sequence (the underlined portion is typed by the Editor).

Select input device:	INTXT*↵ H, L↵	User specifies high-speed (H) or low-speed (L) paper tape device, ↵ = carriage return.
Select subsidiary device:	GETXT*↵ H, L↵	
Select output device:	OUTXT*↵ H,L	

The user's response to INTXT* initializes the Editor to handle the READ command. If it is L↵ ,the low-speed reader on the Teletype is used to read ASCII text into the block buffer for editing. If the response is H↵ , the Editor expects to find the ASCII text on the high-speed paper tape reader. Similarly, the user's response to GETXT* initializes the Editor to handle the GET command by selecting the high- or low-speed reader as the subsidiary device. (The user may specify that the same device be used for both READ and GET.) The response to OUTXT* selects the output device, initializing the Editor to handle the WRITE command on the high- or low-speed punch. To obtain a listing of the edited text, the user must specify the ASR33 punch as the output device by responding to OUTXT* with L↵.

The carriage return is the signal to the Editor to process each step in the initialization. The Editor interprets only the letter immediately preceding the carriage return to be the desired device selection (i.e., HXL↵ is equivalent to L↵).

If that letter is not an H or L, the Editor assumes that a mistake has been made and repeats the initialization message. If the user's response specifies the device that is not ON, or is not even part of the system, the Editor goes into an indefinite loop, the first time it attempts to use that device, waiting for a response from a device that is not available. In that event, the Editor may be restarted (press STOP, set the Address switches to 22_8, and press START) and reinitialized.

2.3.3 **Operation**

All text in the buffer is available for editing until a WRITE command is issued or until each individual line is deleted. Using a block buffer has the advantage of rapid correction of command errors. If the user finds that he has typed the wrong command, he can immediately correct it since the buffer has not yet been added to the output file.

If the ASR33 is the output device, the punch switch must be OFF except when actually outputting edited text (see Paragraph 2.3.3.2). This avoids contamination of the output tape by command echoing and typing of Editor commands. The Teletype punch must be specified as the output device to obtain a listing.

2.3.3.1 **Editor Break (CTRL P)** – Frequently, having made a mistake in his command string, the user may wish to stop processing and reissue his command. When the user types the break character CTRL P (formed by depressing the CTRL key while striking P) during command processing, the normal instruction sequence is interrupted as soon as processing of the current line has been completed.

Control is transferred from the command processor to the edit command decoder. The line after the line that was being processed when CTRL P was typed is left in the work area as the current line for examination or modification. The Editor then awaits a new command from the keyboard.

The break character (echoed as ↑P) results in program restart when the Editor is waiting for a command. On input level, the break character results in a transfer of control to the edit level (see paragraph 2.4.1.2).

2.3.3.2 **Editor Continue** – If the INTXT* device is the Teletype, the Editor halts after a READ. The user must turn the Teletype reader switch to OFF and then press CONTINUE. Similarly, if the GETXT* device is the Teletype reader, the Editor halts after a GET to allow the user to turn the reader switch to OFF and press CONTINUE. If the OUTXT* device is the Teletype punch, the Editor halts twice in the execution of a WRITE. After the command-terminating carriage return, the Editor halts to allow the user to turn on the punch, thus avoiding contamination of his output tape. When punching stops, the Editor halts and waits for the user to press CONTINUE as a sign that the Teletype punch has been turned off, and messages may be typed safely.

2.3.3.3 **Editor Recovery** – If a paper tape reader out-of-tape condition occurs during the execution of a command, the user can recover by typing CTRL R (formed by depressing the CTRL key while striking R).

2.3.3.4 **Using the Erase and Kill Characters** – The Editor allows the use of two keyboard characters for correction of the line currently being typed by the user. The RUBOUT key (erase character) results in the deletion of the immediately preceding character. The Editor echoes a backslash (\) for each RUBOUT typed. CTRL U ("kill line" character) results in deletion of the entire line typed so far. CTRL U is formed by depressing the CTRL key while striking the letter U. The Editor echoes an at sign (@) each time CTRL U is typed.

2.3.3.5 **Editor Restart** – To restart the Editor at the beginning of the initialization sequence, the user should press STOP, set the Address switches to 22_8, and press I/O RESET and START.

If, during command processing (especially FIND and LOCATE), the Editor attempts to move the current-line pointer past the end of the block buffer, it is assuming the user has made a mistake and types

```
END OF BUFFER REACHED BY:
```

followed by the command string. The user must issue a TOP request if further modifications to the current block are required.

If the user requests SIZE␣n, where n is greater than the number of full-length lines that the block buffer can hold, the Editor types

```
CAPACITY WARNING
```

The user may respond with CTRL U to proceed as though the SIZE ␣ n request were never issued, or he can type a carriage return to ignore the warning and continue. If the user types a carriage return, the command is processed and the Editor's buffer capacity may be exceeded.

CAPACITY WARNING is also typed when the Editor calculates that execution of a command will result in more than SIZE or the internal parameter SYSMAX (whichever is greater) lines in the buffer. Again, the user has the option of killing the command (CTRL U) or of processing it anyway (carriage return). The Editor calculates the number of lines in the buffer without regard to their length. For example, with a 4K system, the Editor could easily hold 30 lines of CAP-15 assembly language instructions without comments. Thus

```
>SIZE␣ 30
CAPACITY WARNING
>READ
>
```

would be extremely dangerous if the user intends to add extensive commands throughout his code with the APPEND command. Since the APPEND command does not change the number of lines in the buffer, the user would receive no additional capacity warnings unless he attempted to use INSERT, INPUT, or GET.

2.3.4 Error Recovery

Operator command errors are detected by the Editor. The message

```
NOT A REQUEST:
```

is typed, followed by the command string in error. The user should then retype his command in the correct form.

2.4 COMMAND LANGUAGE

2.4.1 Control Commands

Editor control commands consist of three commands that cause the Editor to enter the input level, and one command to enter the edit level. (Control is initially at the edit level). These commands are described in the following paragraphs.

2.4.1.1 **Transfer from Edit to Input Level** – Any one of the following three commands causes the Editor to transfer control from the edit level to the input level.

a. Carriage return (↵) typed as the first character on a line.

b. The INSERT command (see Section 2.4.4.4) with no arguments. In this case the current line is added to the output before the control level is changed.

c. The OVERLAY (␣ n) command (see Section 2.4.4.7). In this case, n lines (or the current line only if n is omitted) are deleted from the buffer before the control level is changed.

2.4.1.2 **Transfer from Input to Edit Level** – A carriage return typed as the first character of a line when operating in the input level causes the Editor to transfer control from input level to edit level. The user can also change the control level by typing CTRL P (formed by depressing the CTRL key while striking P). When control has been transferred to the edit level, the Editor awaits the next command from the Teletype. The line after the line that was being processed when CTRL P was typed is left in the work area as the current line, ready for examination or modification.

2.4.2 Editor Commands

2.4.2.1 **SIZE [S] (␣ n)** – Set the total number of lines that will occupy a buffer to n. The SIZE command may be issued at any time, and takes effect when the next group of lines is loaded into the buffer via a READ command. The integer variable n is initially set to 20_{10} for a 4K system, or 55_{10} for an 8K system, and must always be set greater than 1.

2.4.2.2 **READ ↵** – Reads sequential lines from the input device, loading them into the buffer as they are encountered, until the number of lines in the buffer is equal to the argument specified in the SIZE request. The pointer is set to the first line of the buffer when the operation is complete (see Figure 2-3). The READ request will not be accepted if any lines remain in the current buffer. The buffer must have been cleared by DELETE requests or a WRITE command.*

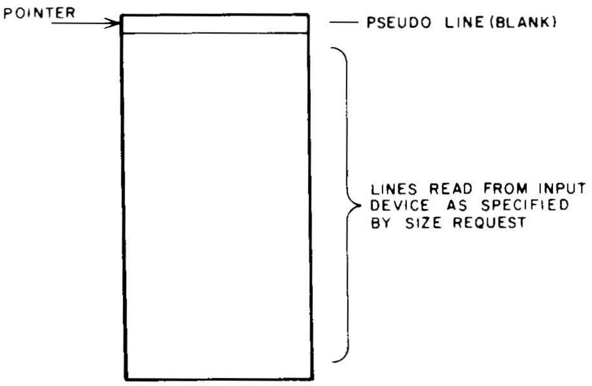


Figure 2-3. Line Buffer After READ Command

*If the input device runs out of tape, the user may *terminate the processing of this command* by typing a CTRL R on the Teletype.

If more than one block of text is to be read, the READ command inputs the first line of the next block into an intermediate buffer. This means that if the user changes tapes on the input device between READs, he will find the "next line" from the previous tape at the top of the block buffer, followed by SIZE-1 lines from the new tape. This does not occur if GET is used to input the new tape into the block buffer, since GET does not use an intermediate buffer.

NOTE

> If the input device is the low-speed paper tape reader, the Editor halts at the beginning of a READ to allow the user to turn on the reader. Also, the Editor halts at the completion of a READ to allow the user to turn off the reader; the user should press CONTINUE to proceed.

2.4.2.3 **GET [G] (␣ n) ↲** – The next n lines from the *subsidiary* input device are added to the buffer below the current line. When command processing is complete, the nth line read is left in the work area as the current line. If n is omitted, it is assumed to be 1. The pointer remains at the last line read (see Figure 2-4).

2.4.2.4 **RENEW ↲** – The contents of the block buffer are written on the output device and a new block is read into core.*

2.4.2.5 **WRITE ↲** – Punches the current contents of the block buffer on the OUTXT device, and clears the buffer.*

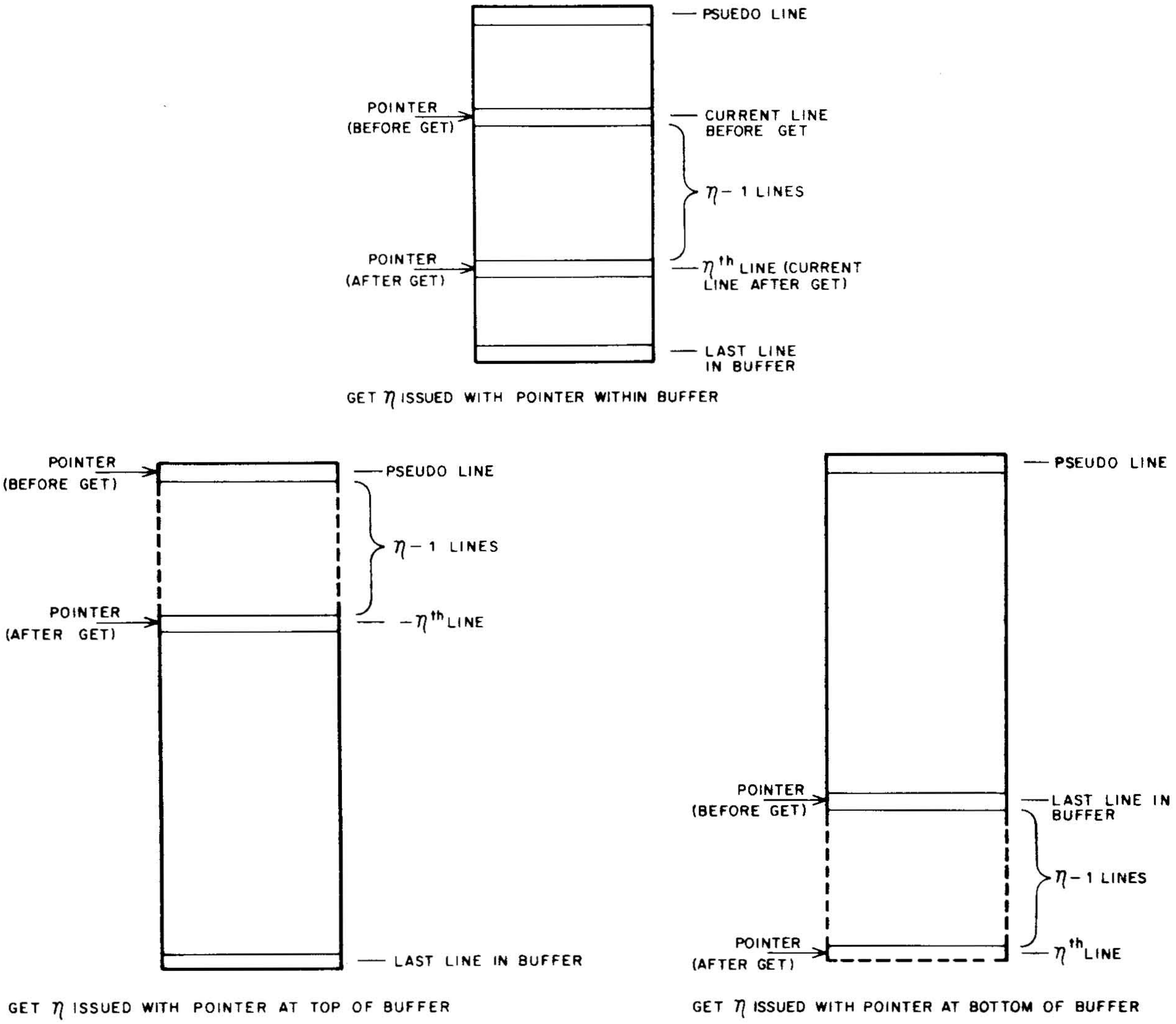
2.4.2.6 **CLOSE ↲** – This command must be preceded by a WRITE request. The remainder of the input file is then written on the output device.*

2.4.3 Pointer Manipulation

The pointer is a software device that places the current line in a work area to facilitate editing. After a page of text has been read, the pointer is positioned at the top of the block buffer to allow the user to insert text *before* the first line in that block. This is done by keeping a pseudo line in the buffer which is never punched out. Thus, the first line of the text in a block is the "second" line in the buffer. Consequently, the first line of the block of text in the buffer is shifted one position to the right when examined with the PRINT command. This "space" is not printed. The following paragraphs describe commands that enable the user to manipulate the pointer.

2.4.3.1 **TOP [T] ↲** – This command moves the pointer to the beginning of the edit block buffer. The first line of the buffer becomes the current line (see Figure 2-5).

*If the output device is the low-speed punch, the Editor Continue feature provides protection against output contamination (see paragraph 2.3.3.2).



GET η ISSUED WITH POINTER WITHIN BUFFER

GET η ISSUED WITH POINTER AT TOP OF BUFFER

GET η ISSUED WITH POINTER AT BOTTOM OF BUFFER

NOTE: If GET n attempts to exceed the maximum line buffer size, CAPACITY WARNING will be typed. The user may type a carriage return (↵) to ignore the warning and continue, or he may type CTRL U to proceed as though the command were never issued.

Figure 2-4. Line Buffer After GET Command

NOTE

If the input device is the low-speed paper tape reader, the Editor halts at the beginning of a GET to allow the user to turn on the reader. Also, the Editor halts at the completion of a GET to allow the user to turn off the reader; the user should press CONTINUE to proceed.

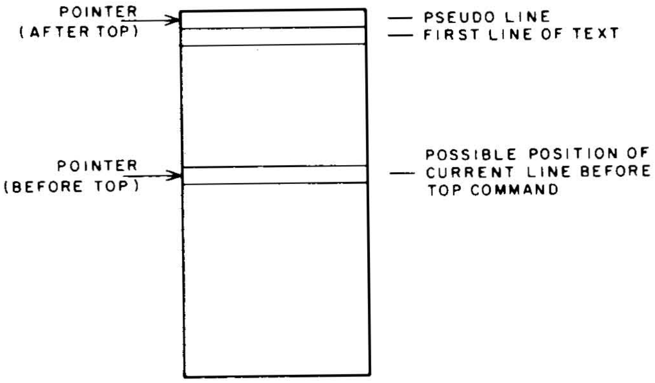


Figure 2-5. Line Buffer After TOP Command

2.4.3.2 **NEXT [N] (⌴ n)↵** – The pointer is moved past the next n lines, beginning with the line currently in the work area. Line N+1 is brought into the work area for modification (see Figure 2-6). If omitted, n is assumed to be 1. If the command results in the pointer moving past the last line of buffer, the error message

END OF BUFFER REACHED BY:
NEXT n

is printed and the pointer is moved to the top of the buffer.

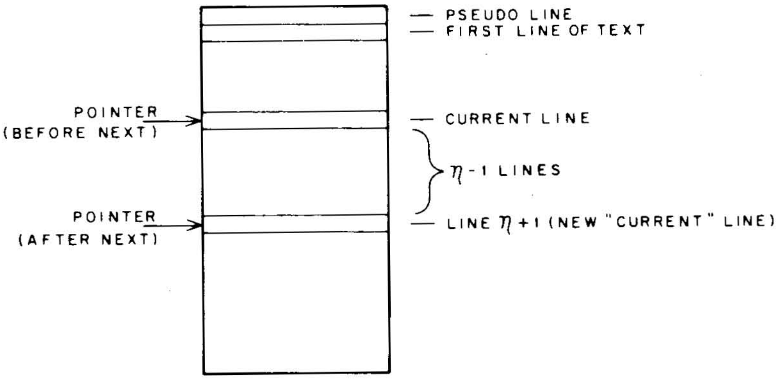
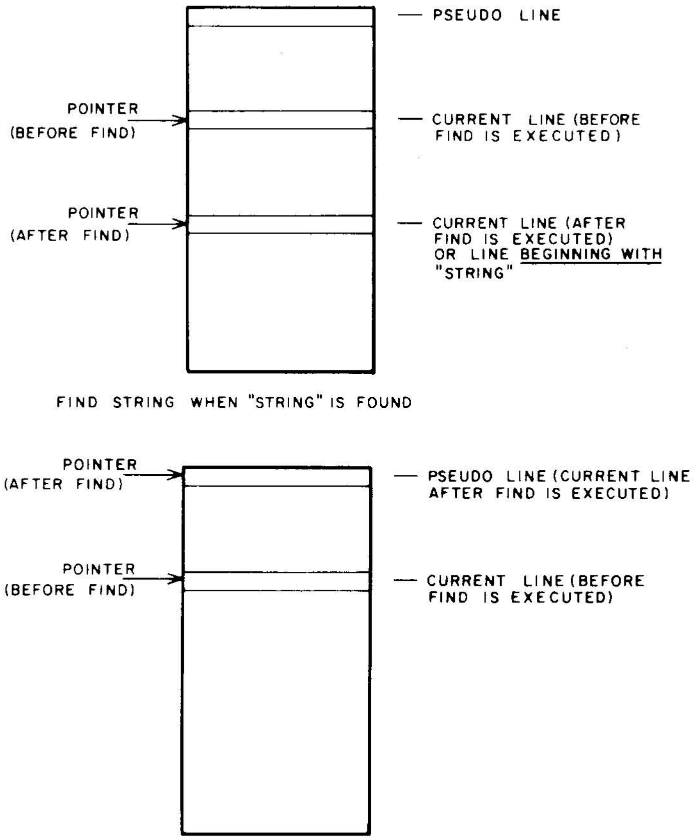


Figure 2-6. Line Buffer After NEXT Command

2.4.3.3 **FIND [F] ⌴ string↵** – This command searches the buffer for the next line that *begins with* the character group "string". The search begins with the line following the current line. If the search is successful, the line beginning with "string" is brought into the work area (see Figure 2-7). If the search is unsuccessful, the END OF BUFFER message is printed and the pointer is moved to the top of the buffer. "String" may contain any number of characters.

2.4.3.4 **LOCATE [L] ⌴ string↵** – This command searches the buffer for the next occurrence of a line that *contains* the character group "string." The search begins with the line following the current line. If the search is successful, the line containing "string" is brought into the work area (see Figure 2-8). If the search is unsuccessful, the END-OF-BUFFER message is printed and the pointer is moved to the top of the buffer. "String" may contain any number of characters.


PSEUDO LINE
POINTER (BEFORE FIND)
CURRENT LINE (BEFORE FIND IS EXECUTED)
POINTER (AFTER FIND)
CURRENT LINE (AFTER FIND IS EXECUTED) OR LINE BEGINNING WITH "STRING"
POINTER (AFTER FIND)
PSEUDO LINE (CURRENT LINE AFTER FIND IS EXECUTED)
POINTER (BEFORE FIND)
CURRENT LINE (BEFORE FIND IS EXECUTED)

FIND STRING WHEN "STRING" IS FOUND

FIND STRING WHEN "STRING" IS NOT FOUND

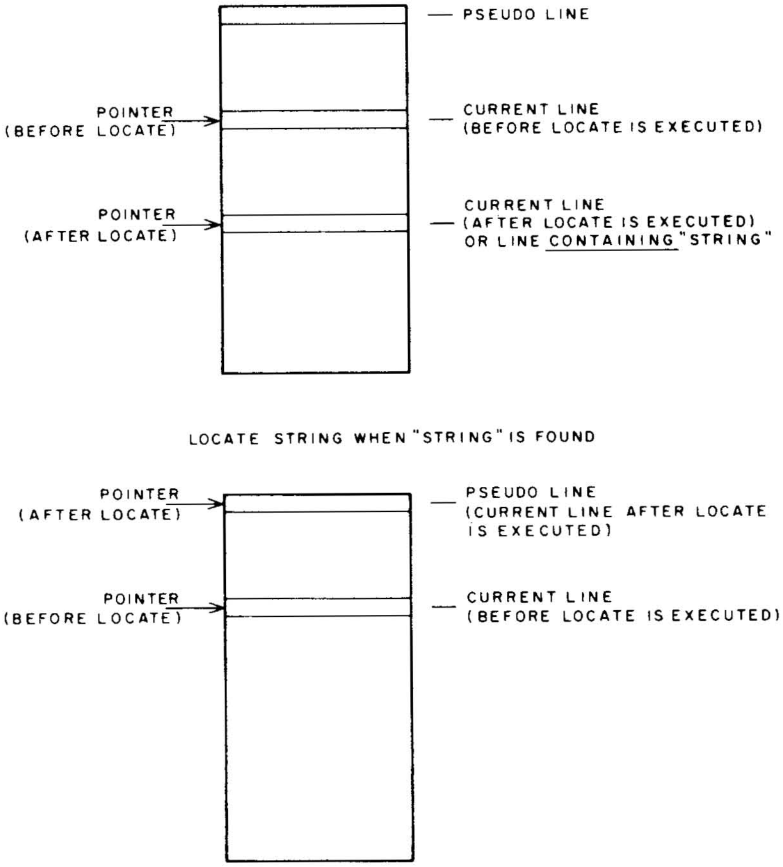
Figure 2-7. Line Buffer After FIND Command

2.4.3.5 **BOTTOM [B] ↵** – This command moves the pointer to the beginning of the last line in the buffer. The last line is typed on the Teletype (see Figure 2-9).

2.4.3.6 **SEARCH ␣ string** – The entire input tape is searched for the next occurrence of a line beginning with the character group "string." This command must be preceded by a WRITE to clear the block buffer. If the search is successful, the line beginning with "string" is brought into the work area with the remainder of the buffer left empty for inputting new text or large inserts for the subsidiary input device using the GET command.

NOTE

If the output device (OUTXT*) is the Teletype punch (L), the Editor will halt at the beginning of a SEARCH to allow the user to turn on the punch. The user should press CONTINUE to proceed with the SEARCH. When the line beginning with "string" has been found, the Editor will again halt to allow the user to turn off the punch. Again, the user should press CONTINUE to proceed.


PSEUDO LINE
POINTER (BEFORE LOCATE)
CURRENT LINE (BEFORE LOCATE IS EXECUTED)
POINTER (AFTER LOCATE)
CURRENT LINE (AFTER LOCATE IS EXECUTED) OR LINE CONTAINING "STRING"
POINTER (AFTER LOCATE)
PSEUDO LINE (CURRENT LINE AFTER LOCATE IS EXECUTED)
POINTER (BEFORE LOCATE)
CURRENT LINE (BEFORE LOCATE IS EXECUTED)

LOCATE STRING WHEN "STRING" IS FOUND

LOCATE STRING WHEN "STRING" IS NOT FOUND

Figure 2-8. Line Buffer After LOCATE Command

2.4.4 **Editing Requests**

The following paragraphs describe commands that enable the user to accomplish his actual editing task.

2.4.4.1 **RETYPE [R] ␣ line ↵** – The character string "line" replaces tne current line. The new line is left in the work area and may be modified.

2.4.4.2 **APPEND [A] ␣ string ↵** – "String" is added to the current line following the last character preceding the terminating carriage return. Thus, to add a comment to the current line

```
JMS GETNUM ↵
```

the command might be

```
A ␣ ⇥ /Get decimal argument. ↵        NOTE: ⇥ indicates a tab
```

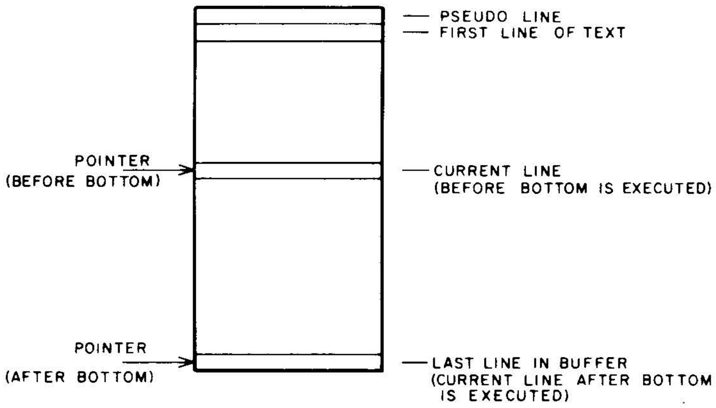

PSEUDO LINE
FIRST LINE OF TEXT
POINTER (BEFORE BOTTOM)
CURRENT LINE (BEFORE BOTTOM IS EXECUTED)
POINTER (AFTER BOTTOM)
LAST LINE IN BUFFER (CURRENT LINE AFTER BOTTOM IS EXECUTED)

Figure 2-9. Line Buffer After BOTTOM Command

The new current line would be:

JMS GETNUM ⇥ /Get decimal argument.↵

If "string" is absent, the current line is unchanged. When the current line *is* changed, the new line is left in the work area and may be modified.

2.4.4.3 **CHANGE [C] ␣ qstring1qstring2q↵** – In the current line, the first character group (string1) which matches that occurring between the first pair of delimiting characters (q's in this case) is replaced by the character group (string2) appearing between the second pair of delimiting characters. The delimiting characters are chosen by the user and can be any character (including blank) that does not appear in either of the character strings. Both "string1" and "string2" may contain any number of characters, including zero. If VERIFY is ON (see paragraph 2.4.5.2), the program will print the new current line on the Teletype when the request change has been accomplished. The new line is left in the work area and may be modified.

Examples

Current Line:	NXTLIN	JMS TYPOUT	/PRNT THE LINE.

a. In the comment, spell PRINT properly, JMS TYPOUT /PRINT THE LINE.

REQUEST:	CHANGE␣/RN/RIN/↵		
NEW LINE:	NXTLIN	JMS TYPOUT	/PRINT THE LINE.

b. Make the "JMS" a "JMP".

REQUEST:	CHANGE ␣ XSXP*X↵		
NEW LINE:	NXTLIN	JMP* TYPOUT	/PRINT THE LINE.

c. Delete the "T" in the tag,

```
REQUEST:          C ␣ /T//⤸
NEW LINE:         NXLIN               JMP* TYPOUT          /PRINT THE LINE.
```

2.4.4.4 **INSERT [I] ␣ line** – The character string "line" is inserted below the current line. The pointer is stepped to place "line" in the work area and "line" becomes the new current line (see Figure 2-10). The program remains at the edit level when the request processing is completed.

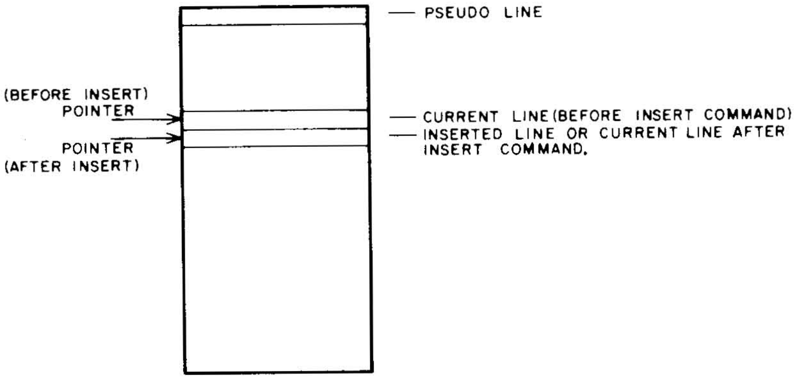

PSEUDO LINE
(BEFORE INSERT) POINTER
POINTER (AFTER INSERT)
CURRENT LINE (BEFORE INSERT COMMAND)
INSERTED LINE OR CURRENT LINE AFTER INSERT COMMAND.

Figure 2-10. Line Buffer After INSERT Command

2.4.4.5 **GET [G] (␣ n)⤸** –This command adds n lines from subsidiary input device to buffer below the current line (see Section 2.4.2.3). If the subsidiary device runs out of tape, the user may terminate processing of this command by typing CTRL R on the Teletype.

2.4.4.6 **DELETE [D] (␣ n)⤸** – The next n lines, including the current line, are deleted from the buffer. The line following the last line deleted becomes the current line (see Figure 2-11). If n is omitted, only the current line is deleted. If n is large enough to cause the pointer to pass the end of the buffer, the END-OF-BUFFER message is printed and the pointer is moved to the top of the buffer.

2.4.4.7 **OVERLAY [O] (␣ n)⤸** – Starting with the current line, n lines (or the current line only, if n is omitted) are deleted from the buffer and the control level is changed to input level. When control returns to edit level, the pointer will be positioned at the last line typed by the user.

2.4.5 Examination Requests

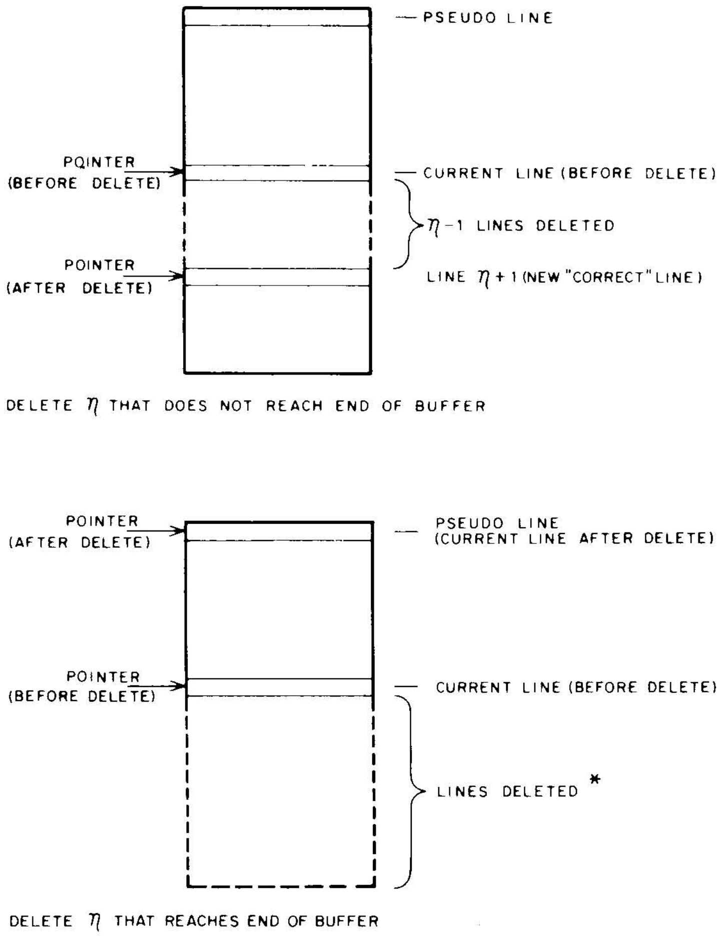
The following paragraphs describe commands that allow the user to examine text within the block buffer.

2.4.5.1 **PRINT [P] (␣ n)⤸** – The next n lines from the buffer, including the current line, are printed on the Teletype. The pointer is left at the last line printed (see Figure 2-12); n is assumed to be one if omitted.

If, as a results of the request, the pointer moves past the last line of the buffer, the error message

```
END OF BUFFER REACHED BY:
PRINT n
```

is printed and the pointer is moved to the top of the buffer (see Figure 2-12).



DELETE η THAT DOES NOT REACH END OF BUFFER

DELETE η THAT REACHES END OF BUFFER

Figure 2-11. Line Buffer After DELETE Command

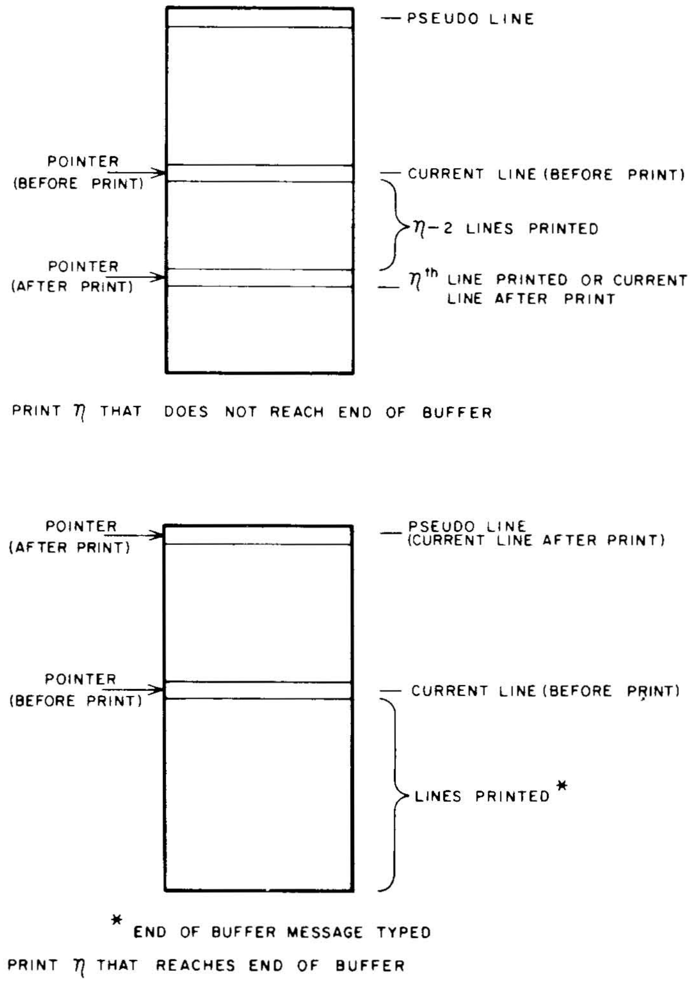
2.4.5.2 **VERIFY [V] ␣ $\begin{pmatrix}\textbf{ON}\\\textbf{OFF}\end{pmatrix}$** – Set Editor response according to the parameter. When VERIFY is ON, text lines are printed in response to certain editing requests, as follows:

a. The line brought into the work area as a result of a FIND or LOCATE request is printed.

b. The last line of the buffer, brought in by the BOTTOM request, is printed.

c. The new line resulting from a CHANGE request is printed.

When VERIFY is OFF, only error messages are printed. After the Editor is loaded initially, VERIFY is ON. The verify request without arguments is equivalent to requesting VERIFY ON.


PSEUDO LINE
POINTER (BEFORE PRINT)
CURRENT LINE (BEFORE PRINT)
$\eta - 2$ LINES PRINTED
POINTER (AFTER PRINT)
η^{th} LINE PRINTED OR CURRENT LINE AFTER PRINT
POINTER (AFTER PRINT)
PSEUDO LINE (CURRENT LINE AFTER PRINT)
POINTER (BEFORE PRINT)
CURRENT LINE (BEFORE PRINT)
LINES PRINTED *
* END OF BUFFER MESSAGE TYPED

PRINT η THAT DOES NOT REACH END OF BUFFER

PRINT η THAT REACHES END OF BUFFER

Figure 2-12. Line Buffer After PRINT Command

2.4.5.3 **BRIEF␣** $\left(\begin{matrix}\textbf{ON}\\ \textbf{OFF}\end{matrix}\right)$↵ – Set Editor verification according to the parameter. BRIEF ON results in the abbreviated printing of the current line when responding to edit requests. An attempt is made to print only the tag, operation code, and address fields of lines brought into the word area as a result of the FIND, LOCATE, and BOTTOM requests. The printing of the new line resulting from a CHANGE request is terminated at the last newly inserted character. BRIEF is initially set to OFF. The setting of this indicator is of no consequence when VERIFY is OFF. The brief request without arguments is equivalent to BRIEF ON.

CHAPTER 3
OCTAL DEBUGGING TECHNIQUE

3.1 INTRODUCTION

Octal Debugging Technique (ODT) is a debugging aid that allows the user to conduct an interactive, on-line debugging session using octal numbers and Teletype commands. The program, which is self-contained and completely independent, is designed to run on a basic 4K or 8K PDP-15/10 with an ASR33 and will operate with PI or API enabled. ODT is written in CAP-15 Assembly Language and may be assembled along with the user programs with which it is to run. All symbols that are internal to ODT begin with a percent sign (%). Thus, if the user avoids beginning his own symbols with this character, multiple definition of symbols will not occur when ODT is assembled with the user program. However, ODT would normally be assembled separately from the user's programs and then loaded into memory only when desired. Standard versions which are delivered in object program form may be used for the debugging of any programs as long as enough memory is available to load ODT at the proper locations.

Using ODT, the programmer can conduct a debugging session, and when errors are found, correct them on-line and execute the program immediately to test the correction. Thus, ODT can be used to compose a program on-line and check it out as composition progresses. Manual operation of console controls is not required to operate ODT; all functions are initiated by typing commands on the Teletype.. The source coding for ODT is so designed that many features can be removed by defining parameters at assembly time. This will be of prime interest to users who are checking out large programs and wish to conserve core storage.

Appendix I provides a convenient listing of the Teletype code in octal form. Appendix J is a concise listing of ODT commands.

3.2 GENERAL DESCRIPTION

All input to ODT is initiated through the console Teletype keyboard. In general, the user types commands to ODT which control the following operations:

Starting the user program at any point.

Stopping and subsequently continuing the user program at selected points called breakpoints.

Examining and/or modifying the AC and Link at a breakpoint.

Examining and/or modifying any memory locations (registers).

Searching user defined areas for registers with specified bit configurations.

Punching out areas of memory to be loaded later for more debugging and/or execution.

Initializing buffer areas to any desired value.

Selection of hardware options such as a high-speed or low-speed paper tape punch.

There are two outputs from ODT: teleprinter and paper tape punch (either high-speed punch or ASR33 punch). The teleprinter is used by ODT to type all information concerning register contents and error indications. This output is very useful since it contains the user's commands as well as ODT responses, which can be analyzed off-line if necessary. The paper tape punch is used to save or dump the user program, or portions of it, as the user desires. The tapes generated when dumping a program are fully checksummed and can be loaded by the PDP-15/10 Loader. The format of this tape is shown in Figure 3-1. Each memory word consists of three data frames of six bits each. The program start address is distinguished from the other load addresses by having bit 0 set to 1.

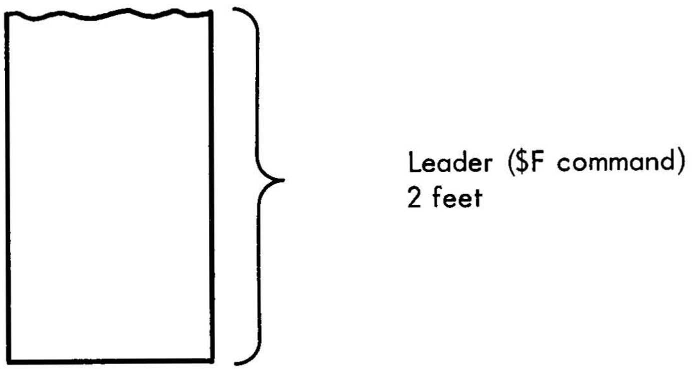


Figure 3-1. PDP-15/10 Loader Format

3.2.1 Operational Organization

ODT can be loaded and used in four basic ways:

It can be loaded immediately following the user program, with control being given to ODT after it is loaded.

It can be loaded after the user program has been running.

It can be loaded before the user program(s) and initiated by manual control after the user program(s) has been running.

It can be loaded by itself and used as a stand-alone program.

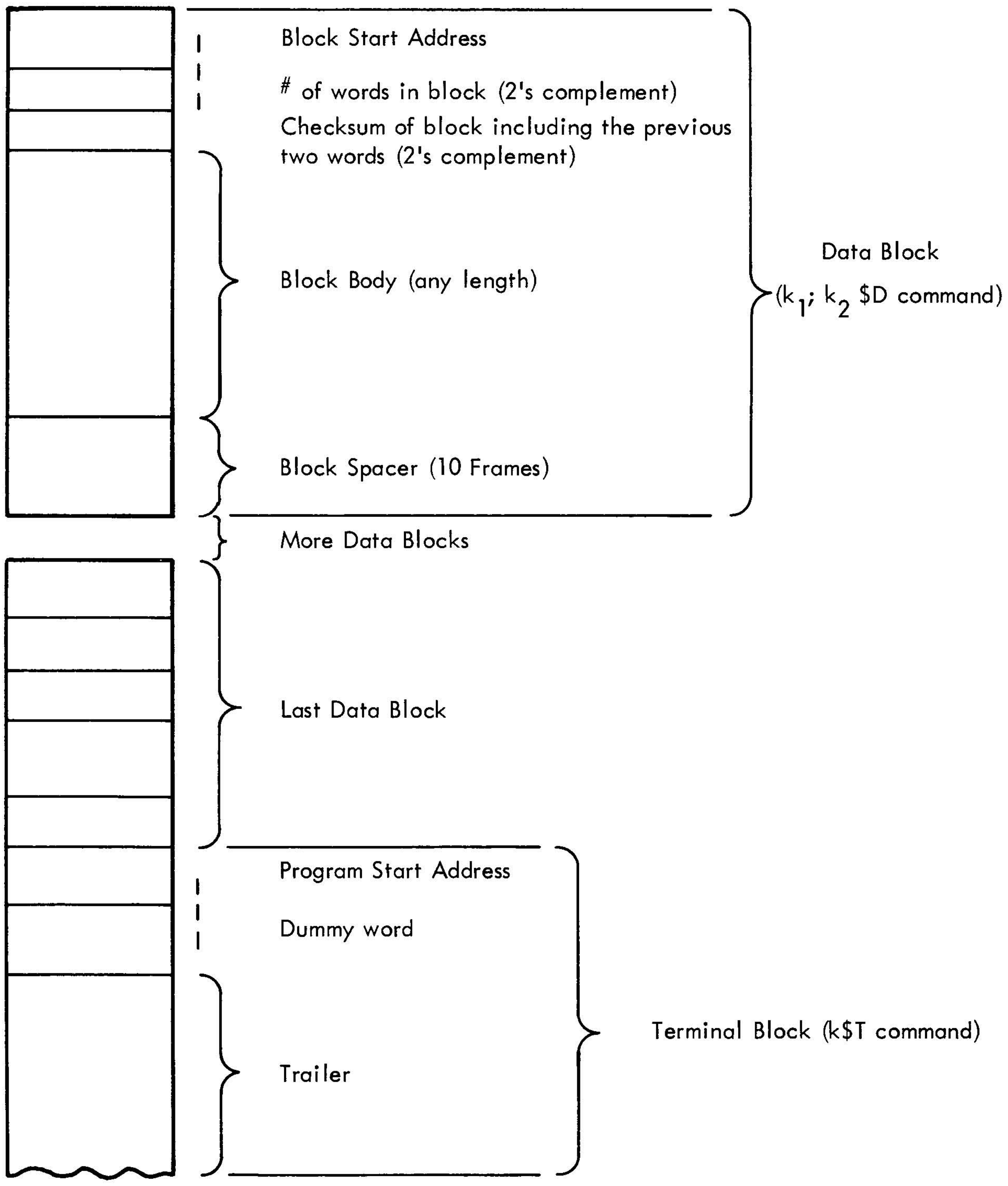


Figure 3-1. PDP-15/10 Loader Format (cont)

In any case, ODT is initiated by starting the processor at location %ODT. The distributed version is loaded and initiated at location %ODT (6000 for 4K and 16000 for 8K). However, the user may assemble versions of ODT which are loaded at any location in memory. Also, by choice of .END statements, ODT may be initiated automatically by the loader or may require manual initiation at location %ODT. Once initiated, the user has control over running and stopping his program with keyboard commands. If ODT breakpoints are requested, the following modifications are made to the user program:

Each instruction that the user designates as a breakpoint is replaced with JMS* AUTOX. This instruction gives control to ODT when the breakpoint is encountered, and is removed when a breakpoint is executed so that the user will see his own instructions if he examines these locations or dumps memory.

An auto-index register, the address of which may be examined and/or modified by using the $V command, is used by ODT and may not be modified or referenced by the user program.

If the PI (program interrupt) is used, ODT must, during breakpoints, intercept all TTY interrupts and pass all other to the user. Therefore the instruction at location 1 is replaced with the instruction JMP %INT. Location 1 is restored when a dump is performed but it is not restored during other ODT operations.

3.2.2 **Functional Organization**

The internal organization of ODT is almost totally modularized into independent subroutines. In general, the internal structure consists of major functions, as follows:

Initialization

TTY Character Input/Output

TTY Message Output

PI Intercept

Command Decode

Command Execution

3.2.2.1 **Initialization** – This function is performed when ODT is started at location %ODT. The tasks performed include disabling the entire system (similar to an I/O reset), resetting miscellaneous registers internal to ODT, clearing all breakpoint requests, clearing hardware and software flags, and initiating the command decode function.

3.2.2.2 **Teletype Character Input/Output** – All teletype communication is performed a character at a time; that is, the input and output is not buffered. This means that the user must not type commands unless ODT is waiting for one. If characters are typed while a requested operation is in progress, they will not be seen by ODT and will not be echoed on the teleprinter.

3.2.2.3 **Teletype Message Output** – This function is performed by repeated use of the teletype character output routine. The functions performed include the typing of addresses, register contents, operational messages and error messages.

3.2.2.4 **PI Interrupt** – If the PI is enabled, the Teletype will interrupt when it is ready for service. Therefore, during a breakpoint, ODT must be able to handle the Teletype I/O interrupts. This function is performed with the help of the PI Intercept (%INT) routine. This routine is entered, instead of the users interrupt program, on every program interrupt which occurs while ODT has control. If the interrupt is not from the Teletype, control continues to the user trap program.

3.2.2.5 **Command Decoder** – The functions performed by the command decoder include:

Reading the keyboard.

Assembling and saving all octal numbers typed by the user.

Detecting the command character.

Allocating control to the proper command execution subroutines.

Detection of illegal numbers and commands.

3.2.2.6 **Command Execution** – In general, each command has a separate entry point into the command execution routines. These routines are entered from the command decoder. They exit either to the user's program ($G, $C) or back to the command decoder to interpret the next command.

3.3 COMMAND LANGUAGE

The following conventions are used throughout this section and the remainder of this chapter:

$ represents either the ALT MODE key or the dollar sign ($) key ($ is echoed).

k represents an octal number of six or less digits. When used to specify an address, k is five or less octal digits.

n represents a single octal digit.

↵ represents carriage return

↓ represents line feed.

↑ (up arrow) is formed by depressing the shift key and typing N.

Any underlined text in the examples refers to that which is typed by ODT.

Δindicates a blank or space.

Priority levels are specified as follows:

Highest	0			
↑	1		Hardware API levels.	
	2			
	3			Optional
	4			
	5		Software API levels.	
	6			
	7			
↓	8	←	PI level	
Lowest	9	←	Program level	

When a command is typed by the user, ODT responds in one of the following ways:

If the command does not require a typed response, ODT will respond with a carriage return and line feed (↵↓) to indicate that the command has been accepted. On this type of command, the user should not type anything on the keyboard until after ODT has typed the↵ ↓. Anything typed after the command and before the↵ ↓ will be ignored; in fact, it won't be echoed on the teleprinter.

On commands that request the contents of a register to be displayed, ODT performs the typed response but does not follow with the↵↓; this allows modification and/or other commands to be typed on the same line.

On commands that cause multiple words, messages, or multiple lines to be typed, ODT terminates the output with↵ ↓↓.

Illegal or incorrect commands are ignored, and ODT responds with ?↵ ↓.

3.3.1 Register Examination and Modification

The following paragraphs describe all ODT register examination and modification commands.

3.3.1.1 **Open a Register (k/,/)** – The command k/ causes ODT to open register k. This means that ODT fetches the contents of register k, types the contents on the teleprinter, and then leaves the register open for modification. Once a register is opened, the user may close it, or he may type a new octal value for the register and then close it. ODT allows only one register to be open at any particular time. Thus, another register-opening command will close the open register.

The / command causes the last referenced register to be opened. This allows the user to open and close register N, perform some non-register-opening commands (possibly including the execution of his program), and then type / to have register N opened again.

Example

400/002066↵↓	Open and close register 400
39?↵↓	9 is illegal
3765/012643 17/001326↵↓	Open and close register 3765 and then open register 17
/001326↵↓	Re-examine register 17
0/001472	Open register 0

3.3.1.2 **Close Register (↵)** – Typing a carriage return (↵) closes any open register. If a register is open and the user types k↵ the value k will be stored in the open register and the register will be closed.

Example

```
300/000206↵↓              Register 300 unchanged
300/000206 3333↵↓         Register 300 changed
/003333↵↓                 Verifies contents of location 300
0/001472 279?↵↓           Open register 0; Register 0 is not changed
```

If a register is open, typing another register-opening command will close it.

Example

```
300/003333 426/ 010000    Closes 300 and opens 426
```

3.3.1.3 **Close Register and Open Next (↓)** – The line feed (↓) works the same as the carriage return (↵) except that after modifying (optional) and closing the open register, the next sequential register is opened and its location and contents are printed. Note that on the 4K machine, location 0 follows 7777.

Example

```
300/003333 126↓↵          Change contents of location 300 to 126_8
00301/700301              and open 301.
```

Note that ODT types addresses as five digits and register contents as six digits with *no* leading zero suppression.

3.3.1.4 **Close Register and Open Previous (↑)** – The up arrow (↑) acts like the line feed (↓) except that the register previous to the currently open register is opened.

Example

```
30/700314↑↵↓              Open register 30. The ↑ causes
0027/700002               ODT to open register 27.
```

3.3.1.5 **Close Register and Interpret Contents as Address (←)** – The left arrow (←) closes (optionally modifies) the currently open register. It then interprets the contents of the register as a memory referencing instruction. ODT then opens the register referenced by the address portion of that instruction. The indirect bit is not tested so that if the register contained the instruction LAC* 200 the following would occur:

```
326/220200←↵↓             Ignore the indirect bit and
00200/nnnnnn              open register 200.
```

3.3.1.6 **Open AC and Link Register ($A)** – Typing SA at any time opens and displays the register holding the current AC value. The value may be modified if desired.

Typing ↓ opens the next register which holds the Link status in bit 17. This may also be modified.

3.3.1.7 **Display PI and API Status ($J)** – Typing $J causes ODT to sample and print the current PI and API status registers. Since both of these may be running, the values may be constantly changing.

Example

```
$J↵↓
PI/nnnnnn↵↓
API/nnnnnn↵↓↓
```

The PI is sampled with an IORS instruction and the API is sampled with the RPL instruction. Definitions of the bits in these registers may be found in the PDP-15 Reference Manual.

3.3.2 Execute Instruction k (k$X)

The instruction k will be executed. The saved AC and Link are restored before execution and saved again after. As long as k is not a CAL, JMP or JMS, control will remain in ODT. On a JMS or CAL, control returns to ODT on the instruction following the JMS or CAL, even if control is transferred to a routine that normally contains a breakpoint. The JMP A instruction will be executed as though the command k$G were given, where k is address A. All other instructions, except for the DBR instruction and the multiply/divide instructions of the EAE class, may be executed in this manner. Note that a skip instruction that actually skips causes the current breakpoint return address to be incremented by one.

ODT responds with ↵↓ as soon as the $X command is accepted. When ODT is ready to accept another command, it types another carriage return – line feed sequence.

Example

```
140012$X↵↓                          User executes a DZM 12.
↵↓
```

3.3.3 Setting User Start Address

3.3.3.1 **Open User Start Address ($Z)** – Typing $Z opens a register internal to ODT and types its contents. The user can then type the start address of his program followed by . Then the command $G (see Section 3.3.3.2) will transfer control to the address specified. (The initial content of the $Z register is unspecified.)

Example

```
$Z000000 200↵↓                      Set start address equal to 000200.
```

3.3.3.2 **Transfer to User Start Address or Address k (G, kG)** – This command is used to start the user program at any location. The command k$G starts the program at location k. The command $G starts the program as specified by the $Z register.

The normal use of $G is to initially start the user program. If breakpoints have been specified, they are set before control goes to the user program. The AC and link are unspecified unless the user has used the $A command to set particular values. The AC and link may also be specified by executing AC modifying instructions with the $X command. The status of the PI and API are not altered by a $G command.

Another use of $G is to restart the user program at any point after a breakpoint has been encountered. That is, instead of continuing after the breakpoint with a $C, the user can start at some other location by typing the $G command. The user's AC and link are restored for a $C command. Again, the PI and API are not disturbed.

3.3.4 **Using Breakpoints**

ODT allows the user to specify up to four breakpoints at selected locations in his program (except location 1). The breakpoint facility of ODT provides a means of suspending the program operation at any desired point and then examining the status of the program through the use of the other ODT commands.

3.3.4.1 **Setting Breakpoints (kB, knB)** – To set a breakpoint at location k, the user types k$nB, where n is 1, 2, 3, or 4. If breakpoint n is already set at some location, it is moved to location k. The command k$B will assume breakpoint 1.

3.3.4.2 **Removing Breakpoints ($nB, $B)** – To remove breakpoint n, the user can type $nB where n is 1, 2, 3 or 4. To remove all breakpoints, the user can type $B.

NOTE

The above described commands do not immediately alter the breakpoints. The mechanism is set so that the desired changes occur at the next $G or $C command.

Example

$B⤸↓	Remove all breakpoints
300$4B⤸↓	Set breakpoint 4 at location 300.
1760$B⤸↓	Set breakpoint 1 at location 1760.

3.3.4.3 **Continue from Breakpoint (C, kC)** – The $C command causes the user's instruction at the breakpoint to be executed and control to be returned to the user program. The k$C command does the same thing except that a loop count is set so that the break does not occur until the kth time it is encountered. This command may be given only while a breakpoint is in progress. Note that even though the loop count may not be satisfied, ODT is still entered and has control for about 200 μs on every pass through the breakpoint. During this time, re-entrancy may occur (see paragraph 3.3.4.9). Note that $C is the same as 1$C.

3.3.4.4 **Kill PI and API During Breaks ($K)** – Typing $K causes the PI and API to be disabled during all breakpoints. Their status is saved and restored at the $C command. The $K command stays in effect until the $U command is given.

3.3.4.5 **Allow PI and API During Breaks ($U)** – Typing $U re-enables the PI and API after having been disabled by the $K command. This is the normal operating mode for ODT.

3.3.4.6 **Vary Autoindex ($V)** – The operation of breakpoints requires the use of one autoindex register. The user may not use this register if ODT is used. Normally, ODT uses location 17 but the user may change this with the $V command. Typing $V opens a register internal to ODT, which holds the address of the autoindex register. The user may alter this value to any number in the range 10 through 17. This alteration, if performed, should be accomplished before the first $G command is given to ODT. The user should also note that the autoindex register is used by ODT even if breakpoints are not specified. Starting ODT at location %ODT will reset the autoindex location to 17. Restarting ODT at %REST will leave the autoindex specification unaltered.

3.3.4.7 **Open Re-entrant PC List ($Y)** – This command, which is really only meaningful after a re-entrancy error, opens the first breakpoint return address. Typing ↓ opens the succeeding PC values. The list is terminated by an IOT class instruction (op code 70). Note that the values in this list are the addresses + 1 of the active breakpoints.

3.3.4.8 **Breakpoint Operation** – When, during execution, the user program encounters the location of the breakpoint, control is transferred immediately to ODT. ODT saves the user program AC and link. The status of the teleprinter flag is also saved. From this point on, until a $G or $C command is given, the user does not have control of the Teletype. ODT then replaces all break instructions and types the breakpoint number and location as follows:

```
↵↓ $Bn Δk↵ ↓
```

where n is the breakpoint number and k is the address of the breakpoint.

ODT is now in control and is waiting for the user to type commands. The user can examine and change any registers. The current AC and link may be examined and modified by using the $A command. The user should note that the instruction at the breakpoint has not been executed. It will not be executed until a $C command is given. The user can remove and/or move any breakpoints at this time.

At a breakpoint, while ODT has control, the status of the PI and API is not altered (except for short periods of time as described below). Thus, a breakpoint at priority n causes ODT to be entered and executed at that priority. This means all processing at that priority level and below will stop.

However, all processing at priority levels above the level of the breakpoint will continue. It may be that the user will want all processing to stop at a breakpoint. This may be accomplished in two ways. The user may set the breakpoint in a section of code that is executed at the highest priority. If this cannot be done, the user can type the command $K to "kill" PI and API processing at the next breakpoint. The PI and API status is saved and then restored on the $C command. The $K command can be typed at any time that ODT has control; it stays in effect until the $U command is typed and vice versa.

3.3.4.9 **Breakpoint Restrictions** – The following restrictions apply when using breakpoints:

a. Breakpoints may not be placed at the following types of instructions:

   (1) Instructions that are modified by the user program.

(2) Instructions that are executed out of line by an XCT.

(3) Instructions or data that are not actually executed, such as instructions used as arguments following a JMS.

(4) An instruction that uses indirect addressing and is the first indirect following a DBR instruction. Breakpoints on other indirect instructions are not restricted.

(5) An instruction in an area operating in the extend mode.

(6) On any of the exit sequence instructions of a PI or API routine. For example, a breakpoint must not be set on any of the instructions .

```
ION
DBR
JMP*   0
```

in the exit of a PI routine.

b. If a breakpoint is placed on a CAL instruction, the breakpoint will occur as normal. However, when the $C command is given, the breakpoint will be removed before control returns to the user. Thus, a breakpoint on a CAL instruction may only be executed once. A breakpoint may be set on any of the other memory referencing instructions (op codes 04 through 60), any operate instruction (op code 74), and the EAE instructions (op code 64). The IOT instructions are discussed in the next paragraph.

c. There is one IOT class instruction that cannot be at a location where a breakpoint is set; this is the DBR instruction. Generally, this instruction requires that it be followed by a JMP* referencing the location holding the PC, link and extend mode status. This will not occur if the DBR instruction is at the breakpoint, since many ODT instructions will be executed after the DBR is executed and before the user regains control.

If the $K command is in effect, breakpoints should not be set at the IOT instructions listed below. If a breakpoint is set at these instructions, the results will be as indicated.

IORS — The PI bit will always indicate that the PI is disabled.

IOF — No operation

ION — Could cause ODT to fail if any breakpoints are defined within an interrupt routine.

SPI — SPI will operate correctly if the API was off when the break was executed; however, if the API was enabled, then ODT raises the priority to level 0 (highest priority). This may cause SPI not to skip when the user expects it to do so.

ISA — If the API was disabled when the break was executed and if the ISA is enabling the API, ODT may fail if any breakpoints are defined within the API level routines. If the API was enabled when the break was executed, then the API will be enabled and active at level 0 when the ISA is executed. ODT will then DBK. Thus, if the user's ISA has requested an interrupt at levels 4-7, the ISA will work. If the ISA has turned off the API, the ISA will operate correctly. But if the ISA has tried to raise the priority to levels 1-7, the ISA will operate as a no-op.

RPL — If the API was off when the break was executed, RPL will work. Otherwise, RPL will read a word into the AC which indicates that the API is enabled and level 0 is active.

d. There are several timing considerations for setting breakpoint instructions.

(1) If the user is operating with no PI or API, then setting a breakpoint merely stops his program. Hence, all I/O in progress will be terminated until the $C command is given.

If a XX$C command is given, the breakpoint is not executed again until the XXth occurence. The user must realize that on every pass over the breakpoint, ODT is entered, the user instruction is executed by ODT, and then control is returned to the user program. Therefore, the instruction at the breakpoint, which normally takes from 1.5 to 12 μs to execute, will actually take about 200 μs to execute.

(2) If the use is running with PI and API enabled, then several additional timing restrictions must be considered.

(a.) Assume the user inserts breaks only at the program level (that is, at non-PI, API levels). When a breakpoint is encountered, independent of the status of the iteration loop count, the PI is disabled for short periods of less than 80 μs and the API is temporarily raised to level 0 (highest priority) for periods of less than 50 μs.

(b.) Assume that the user inserts multiple breakpoints on the same priority level. When the breakpoint is encountered at level n, ODT will operate at that level (except for the short intervals, described in paragraph (2) (a) above where ODT operates at level 0). Therefore as long as the break is in progress, which could be several minutes, all interrupts on that priority and below will be inhibited.

(c.) Several possibilities exist if the user places breakpoints on more than one level. In general, ODT is designed such that only one breakpoint may be in progress at any given time. However, it is possible that a breakpoint is executed at level N and then, before ODT can remove the other breakpoints, another breakpoint is encountered at a level M, where M has a higher priority than level N. ODT always detects this situation, but there is no way to recover to the extent that the user can continue his program. However, ODT will do the following:

Disable PI, API

Print the message ODT REENTERED
Halt

The user may press CONTINUE to restart ODT or he may manually restart ODT at location %REST (16011 on the distributed version). Thus, the user may examine, with ODT commands, the status of the program. In particular, he may use the $Y (see below) command to open up the first location of list of PC values of all breakpoints which are now active. By stepping down the list, he may determine which breaks occurred and in which order. The list may have from two to four entries, but is terminated with an IOT class instruction (op code 70). At first glance, it may appear that if a break is in progress, then the next break which interrupts the currently active break will cause ODT to terminate. However, the following may occur:

A breakpoint instruction is executed at some level. This saves the PC (return address which is the breakpoint address + 1).

A higher priority level becomes active and takes control away from ODT, possibly before ODT can even save the AC. Of course, this may occur at any time before ODT removes the other breakpoints.

Another break is executed on the higher level. This instruction will save the PC in the location following the PC of the interrupted breakpoint. If this breakpoint routine (in ODT) executes one more instruction, then ODT will detect the re-entrancy and the re-entrant error process will be initiated. However, this second breakpoint may be interrupted by a higher level request before it executes its first instruction (following the breakpoint instruction). Of course, the higher level routine may contain another breakpoint; etc.

In any case, the following can be guaranteed:

In no case will the PC value of an executed breakpoint be lost.

Since there are a maximum of four breakpoints, either some breakpoint will get far enough to detect the re-entrancy or the breakpoint at the highest level will be executed. This means all other breaks are locked out so that the re-entrancy will be detected.

On multiple entries, all AC values after the first will be lost. The first AC may or may not be lost.

Thus, in case of re-entrancy, ODT will not collapse and the user will be able to determine what happened. The user may not issue a $C command. The $G must be used to restart the program.

3.3.5 **Searching Operations**

ODT has the capability to search a specified area of memory for any bit configuration in any specified bit position. When a match is found, ODT types the location and the contents of the memory register. The search is performed between a low and high limit with the following algorithm:

a. Get the memory word

b. Mask with the search mask (see below)

c. Compare with *k* from the k$W command

d. If equal, print location address and unmasked contents. If not equal, go to next memory register.

The following command sequence is used to set the mask, low limit, high limit, and initiate the search:

$M	Open mask register for modification
↓	Open low limit register
↓	Open high limit register
k$W	Search for *k*

The mask and limits must be set before the k$W command is given. Otherwise, whatever is in these locations will be used.

Note that a core dump between the low and high limits is performed if the user specifies a mask of 0 and then types 0$W ($W is the same as a 0$W).

Example

Suppose the user wants to locate all JMS instructions between location 200 and 1500. The command sequence would be as follows:

```
$M 077336 740000↓↵          User opens and changes the mask.

07745/00100 200↓↵           The line feed on previous line opens low limit

07746/007000 1500↵↓         Opens and changes high limit.

100000$W↵↓                  User types JMS op code and then

00300/102030↵↓              ODT starts the search

01476/126750↵↓↓             There are two JMS instructions

                            JMS    2030 at location 300 and

                            JMS*   6750 at location 1476.
```

NOTE

The address of low-limit and high-limit registers may vary depending upon the assembled version of ODT.

3.3.6 **Initialize Buffers ($I)**

This command is used to load a specified value into any contiguous block of memory. The user defines the block to be loaded by setting start and stop addresses into the low and high limits of the word search routine. This means the user must first open the mask with $M and then type a line feed (↓) to open the low limit. Once the start and stop addresses have been set up, typing k$I causes the value k to be loaded into the buffer. Typing $I is the same as typing 0$I.

Example

```
$M 740000↓↵              User opens MASK and then types ↓ to open low limit. Change to 433.
                         Open high limit and change to 624. Fill buffer with 400000.
07745/000200 433↓↵

07746/007000 624↵↓

400000$I↵↓↓
```

NOTE

> The address of low-limit and high-limit registers may vary depending upon the assembled version of ODT.

3.3.7 **Paper Tape Output**

The following paragraphs describe the ODT paper tape output commands.

3.3.7.1 **High-Speed/Low-Speed Punch Available ($H, $L)** – Typing $H indicates to ODT that all punching is to be performed on the high-speed punch. Typing $L indicates to ODT that all punching is to be performed on the low-speed punch (ASR Teletype). Since the low-speed punch and the teleprinter are physically connected, the user always has the punch turned off until after a command to punch has been given to ODT. ODT halts to allow the user to turn the punch on and off at the correct times. The user operates the CONTINUE switch on the operator's console to resume ODT operations.

3.3.7.2 **Feed Leader ($F)** – The $F command causes about 2 feet of leader to be punched either on the high- or low-speed punch (depending on whether $H or $L has been seen by ODT). For the low-speed punch, ODT halts to allow the user to turn on the punch. The user then operates the CONTINUE lever. When the punching is complete ODT halts again so that the punch may be turned off before the next command is typed. The punch must be off when the keyboard is used or else the commands will be punched as they are typed. The halts are not performed when the high-speed punch is used.

NOTE

> The initiation of any punching permanently disables the PI and API. The user's program may not be continued with $C. The program must be restarted with a $G.

3.3.7.3 **Dump Block of Memory (k_1 ; k_2 $D)** – This command causes the dumping of all locations between k_1 and k_2, inclusive. If the low-speed punch is used, halts occur before and after punching. If k_2 is not specified, an error is assumed.

All dumping is performed in the PDP-15/10 Loader format (see Figure 3-1). The main feature is that noncontiguous blocks may be punched, each with a separate checksum. After the last block there is a special start block that tells the loader to stop loading and transfer control to a specified address. This terminal block is punched by using the k$T command. As many blocks as desired may be dumped before the k$T is issued.

3.3.7.4 **Terminate Punching (k$T)** – The k$T command causes the terminal block (see paragraph 3.3.7.3) to be punched. If $T is typed, the loader halts when loading is complete. If k$T is typed, the loader gives control to location k when loading is completed.

3.4 OPERATING PROCEDURE

This section contains paragraphs describing the ODT loading procedure, start-up procedure, conditional assembly, and error recovery.

3.4.1 Loading Procedure

ODT can be loaded in several different ways. Since the user will have the symbolic source tape of ODT, he may assemble it into any convenient, contiguous block of memory. The pseudo-op .LOC must be supplied by the user. ODT would not, in general, be assembled below location 101; ODT may not, under any circumstances, be assembled below location 21. ODT may either be assembled and loaded with the user program, or assembled separately and loaded with the user program. It could even be loaded after the user program has been running. ODT may be assembled in the .FULL mode if desired.

ODT will also be supplied as a binary tape that can be loaded by the PDP-15/10 Loader. Assuming that the loader is already in memory, proceed as follows:

Press IO RESET

Set the Address switches to 7700 or 7720 (17700 or 17720, if 8K). Note that these two addresses are for the two versions of the Loaders; the first one for the low-speed and the second for the high-speed paper tape reader.

Press START. ODT will be initiated automatically by the loader unless a checksum error occurs. The distributed binary version loads at location 6000 (16000 if 8K) and runs almost up to the PDP-15/10 Loader. The addresses for starting and restarting are:

%ODT = 6000	4K systems or	%ODT = 16000	8K systems
%REST = 6011		%REST = 16011	

3.4.2 Start-Up Procedure

Several start-up procedures may be used with ODT. The distributed binary version starts automatically when loaded. User-assembled versions are started in a manner dependent upon the .END statement used. The source tape will not have an .END statement. Instead, it will have the .EOT pseudo operation. When the .EOT is encountered the user may continue with another segment of his program or he may insert a segment consisting of the .END statement. In any case, the .END determines the start-up procedure as follows:

.END %ODT - ODT will start automatically when the binary tape is loaded.

.END X - The user program will start at location X when the binary tape is loaded. The user will have to manually give control to ODT at location %ODT.

.END - The loader will halt when loading is complete. The user must manually start at %ODT when he is ready.

If ODT is in control initially, the user program is initiated with the $G command. Normally, control will then stay in the user program until a breakpoint is executed.

In general, when ODT is started manually, an IO reset should be performed first.

When ODT is initiated, it types ↵ ↓↓ to indicate that it is ready to accept commands.

3.4.3 Conditional Assembly

The ODT source program is designed such that certain features may be deleted under the control of assembly parameters. At assembly time, a parameter tape may be read before the ODT source tape in pass 1. This tape contains definitions of parameters that allow the user to generate a particular version of ODT. The parameter tape need not be read on pass 2; however, reading it on pass 2 will provide a list of all defined parameters for the assembly.

A parameter is either "defined" or "undefined." Defined means that the parameter name appears in the Assembler symbol table. "Undefined" means that it does not appear in the symbol table. None of the parameters are defined in the ODT source code. If the source tape is assembled without a parameter tape, all of the parameters listed below will be undefined; thus a version with all features will be produced. To define a parameter, %%J for example, include the assignment statement

```
        %%J=0↵ ↓
```

on the parameter tape. The parameter tape should be terminated by the .EOT pseudo-op. A complete description of how to define a parameter is given in Paragraph 1.4.7.

The following paragraphs list the parameters available for ODT and their effect.

3.4.3.1 **%%J** – If undefined, the command $J (display PI and API status) is available. If defined, $J is not available, thus saving approximately 21 (17_{10}) locations.

3.4.3.2 **%%X** – If undefined, the $X (execute instruction k) command is available. If defined, the $X command is not available, thus saving about 46 (38_{10}) locations.

3.4.3.3 **%%MULD** – If undefined, there is no restriction on setting a breakpoint at a multiply or divide EAE class instruction. However, if the user's PDP-15/10 does not have the EAE option or if the user does not want to set breakpoints at multiply or divide EAE instructions (the other EAE instructions are not restricted), then 25 (21_{10}) locations may be saved by defining %%MULD

3.4.3.4 **%%WM** – If undefined, the word searching commands $M, $W, and the $I command are available. If defined, these commands are not available, thus saving about 54 (44_{10}) locations. If the user defines %%WM, the utility program SCAN may be used to perform the word searching or scanning function.

3.4.3.5 **%%KU** – If undefined, the commands $K and $U are available to control the PI and API at breakpoints. If defined, these commands are not available, saving about 23 (19_{10}) locations. In this case, $U is always in effect.

3.4.3.6 **%%DFT** – If undefined, the commands $H, $L, $D, $F, and $T are available to dump segments of memory in the PDP-15/10 Loader format on either the high-speed ($H) or low-speed ($L) paper tape punches. If defined, these commands are not available, saving about 213 (139_{10}) locations. When these commands are not available, dumping may be performed, in the HRI format, by the HRM Puncher utility program.

3.4.3.7 **%%V** – This parameter controls the $V command for varying the autoindex number. If %%V is undefined, the $V command is available. But if %%OB (see below) is defined, ODT will not need an autoindex register and the

$V command will not be available even if %%V is undefined. If %%V is defined, then the $V command is not available. In this case, ODT assumes register 17*. If %%V is defined and %%OB is undefined, approximately 30 (24_{10}) locations are saved. But if %%OB is defined, these 30 locations are included in the locations saved.

3.4.3.8 **%%OB** – This parameter controls the number of breakpoints allowed in ODT. As described in the paragraph about the $B command (see paragraph 3.3.4.1), there are four breakpoints available. This is the case when %%OB is undefined.

If %%OB is defined, there is only one breakpoint available. This has several consequences which are listed below. It is assumed that the reader is familiar with the description of the breakpoint commands and restrictions described in paragraph 3.3.4.

a. The command $B removes the breakpoint. (If $nB is typed, ODT ignores the n).

b. The command k$B sets the breakpoint at location k. If it was already set somewhere else, it is moved to location k. (If k$nB is typed, ODT ignores the n).

c. When the breakpoint is encountered, the message printed is "↵↓ $B Δ k↵↓" where k is the address of the break.

d. The re-entrancy problem does not exist when only one breakpoint is available.

e. The command $Y (open re-entrant PC list) is not available.

f. No autoindex register is used; thus, the command $V (vary autoindex) is not available.

g. Approximately 214 (140_{10}) locations are saved.

3.4.3.9 **%AUTOX** – The default assumption is 17 for which autoindex register to use. The parameter %AUTOX may be defined to change the default assumption; that is, if %AUTOX is undefined, then register 17 is used unless changed by the $V command. If %AUTOX is defined as

%AUTOX=XX

where XX is an octal number in the range of 10-17, then the default assumption will be autoindex register XX.

3.4.4 **Error Recovery**

3.4.4.1 **Runaway Program** – If the user program does not execute a breakpoint, ODT will not regain control. The following sequence of operations may be performed to give ODT control:

Depress PROGRAM STOP.

Depress IO RESET

*The assumed autoindex register may be changed by a parameter definition. (See %AUTOX)

Set the address of %REST in the Address switches (6011 or 16011 on the distributed version).

Depress START

This will cause ODT to restore the breakpoint instructions and transfer control to the command decoder. The $G command must be used to start the user's program.

3.4.4.2 **Re-Entrancy** – The problem of re-entrancy on breakpoints is discussed in Section 3.3.4.9. ODT always detects the re-entrancy and prints an error message. At this time the user cannot determine if the PI and/or API were enabled when the re-entrancy occurred. However, the other PI and API status bits may be examined with the $J command. The $Y command is used to open the first entry of a list of PC values with the following meanings:

PC_1 = Adr + 1 of first breakpoint to occur
PC_2 = Adr + 1 of next breakpoint to occur
.
.
.
PC_n = Adr + 1 of nth breakpoint to occur (maximum of four)
70XXXX = End of PC list

After the first value has been printed (by the $Y), each successive value may be printed by using the ↓ command. The end of the list is indicated by an IOT class instruction (70XXXX).

After a re-entrancy error, the user must again specify all breakpoints before restarting his program with the $G command.

3.4.4.3 **Breakpoint Entry Error** – In the event of a software error in the user program or in ODT, it is possible that control could come to ODT as though a breakpoint had been executed. ODT would not be able to determine which breakpoint was executed; ODT would, therefore, proceed as follows:

Disable PI, API
Print BAD BREAK ENTRY
Halt

ODT can be continued by pressing CONTINUE or by manually starting at location %REST. Since, after a software failure, the user program or ODT may be destroyed, the user should reload both his program and ODT.

Another situation, which is not a software error but an operational error, will cause the above error procedure to occur. This is the execution of the instruction at the breakpoint address, out of line, with an XCT. If this is the case, reloading is not necessary but the user's program may be restarted only with the $G command.

3.4.4.4 **Command Error** – If a command is illegal or contains illegal characters, ODT ignores the command and responds with ?↵ ↓. If, while typing a command, the user changes his mind, he can cause ODT to ignore the command in the following ways:

Type an 8 or 9.

Type two ALT MODEs (or two $s)

Type CTRL

Type an illegal command character (e.g., "E").

3.5 ODT ASSEMBLY INSTRUCTIONS

ODT will be delivered both as an object program tape and as a source tape. The object program tape is loaded and initiated at location %ODT (6000 for 4K and 16000 for 8K). This version includes all ODT commands defined in this manual.

The source tape is provided to allow the user freedom to assemble ODT at different locations, in the .FULL or PDP-15/10 Loader format, and with certain features and options deleted or changed.

Paragraph 3.4.3 described all of the conditional features of ODT. The largest version of ODT (no parameters defined except possibly for %AUTOX) is approximately 950_{10} locations. The smallest version (all parameters defined except possibly %AUTOX) is approximately 530_{10} locations.

The source tape has no .LOC statement and does not have an .END statement. If the user assembles ODT by itself a .LOC and .END must be provided on separate tapes (segmented source tapes are described in paragraph 1.4.2 (Part II)). The main reason for leaving off the .LOC and .END is to allow the user freedom to change the .LOC or .END or follow ODT with user programs without requiring an editing of the ODT source tape.

The following shows the coding for the tapes necessary to assemble ODT at location 200 in the .FULL mode with automatic initiation and with the dumping and word searching commands deleted.

Tape 1

```
   %%DFT  = 0↵↓
   %%WM   = 0↵↓
⇥ .EOT↵↓
```

Tape 2

```
⇥ .FULL↵↓
⇥ .LOC ⇥ 200↵↓
  .EOT    ↵ ↓
```

Tapes 3, 4, 5

Supplied ODT source tape (segmented into three pieces).

Tape 6

```
⇥ .END⇥%ODT↵ ↓
```

On pass 1 of the assembly, tapes 1 through 6 must be read by the Assembler. On successive passes, it is not necessary to include tape 1; however, doing so will cause the parameter assignments to appear on the assembly listing. All tapes other than the parameter tape must be read on all passes.

CHAPTER 4
COMPACT UTILITY ROUTINES

4.1 INTRODUCTION

This section provides descriptions of all utility routines supplied with the PDP-15/10 COMPACT Software System. These routines include a FAST-15 system for DECtape handling, a hardware readin mode (HRM) punch routine, paper-tape handling routines, Teletype I/O routines, an octal dump routine, and a scan routine used to search core memory.

4.2 FAST-15

4.2.1 General Description

FAST-15 (Fast Acquisition of System Tape) is a loading system for use in the PDP-15/10 COMPACT Software System to retrieve frequently used programs from DECtape and to create system tapes. The main advantages of the system are speed and ease of access.

The equipment required for use of FAST-15 includes a basic PDP-15/10 with 4,096 words of core memory, one Type TC02 DECtape Control unit, and one Type TU55 DECtape Transport.

The FAST-15 system tape, as distributed by Digital Equipment Corporation, contains commonly used system programs such as the Symbolic Editor, the CAP-15 Assembler, and ODT. Since these can be called from DECtape with only a small bootstrap, paper-tape handling is eliminated. This results not only in a significant time savings, but also in increased reliability. FAST-15 is by no means restricted to systems programs; it can be employed very conveniently for frequently accessed user-created programs. This chapter contains complete directions for use of the FAST-15 system tape, as well as directions for adding user programs to the system.

4.2.2 The FAST System

The FAST System includes four programs: the FAST Loader, the High Writer, the Low Writer, and the Reader.

In normal use, once the FAST Loader and FAST Writer have prepared a DECtape for system use, only the FAST Reader need be used; this program is commonly designated FAST.

4.2.2.1 **FAST Loader** – The FAST Loader writes a table of contents or directory onto block 1 of a certified DECtape. The directory consists of 18 three-word entries; one entry for each of the 18 accumulator (AC) switches. Each directory entry specifies three parameters for the program to be stored and retrieved, under the control of an AC switch, as follows:

The first location in memory occupied by the program (load point),

The number of locations allocated in memory,

The starting location of the program.

4.2.2.2 **FAST Writer** – The FAST Writer transfers a program from core memory to DECtape as specified by the table in the first DECtape block. Program selection is determined by the leftmost AC switch in the 1 position. The FAST Writer exists in both high and low versions. The high version, which occupies location 7600 - 7777_8, is normally used. However, it is permissable for programs read by the FAST Reader to overlay the first 100_8 locations of the Reader (e.g., locations 7600 - 7677_8). So that such programs may be written on a FAST DECtape, a low version of the FAST writer, which occupies locations 100 - 300_8, is used.

4.2.2.3 **FAST Reader** – The FAST Reader transfers a program from the DECtape to the computer memory in the locations specified by the table in the first DECtape block. Program selection is determined by the leftmost AC switch in the 1 position.

4.2.3 The FAST Loader

The FAST Loader writes a table of contents (directory) of predetermined programs onto the first block of certified DECtape. This table determines the order of programs on DECtape and can be modified by the user. This table, which is located at the end of the FAST Loader, contains the following information for all programs:

Its first location in memory (load point).

The total number of words to be loaded.

Its starting address.

One program or core image is assigned to each of the 18_{10} accumulator switches. To each switch, 32_{10} DECtape blocks are assigned which allows 17600_8 words to be written and/or read. The number of blocks assigned to each switch is not variable, but the number of words read or written is variable. Switch 17 has 30_{10} blocks assigned. This allows 17000_8 words to be written instead of 17600_8.

To load the directory onto the first block of the DECtape, the user should first modify the table of contents in the FAST Loader as necessary. This is most conveniently done by using the Symbolic Editor and punching a revised source tape. Using the CAP-15 Assembler, prepare a binary object tape. Select the DECtape for unit 1-WRITE. Load the FAST Loader using the PDP-15/10 Loader; it will stop with all 1s in the AC. Press CONTINUE to execute the writing of the directory on block 1. If an error occurs, the Loader will return to the beginning and halt with all 1s in the AC; otherwise, it will halt with the AC clear (all 0s).

4.2.4 The FAST Writer

This description assumes that the DECtape which is about to be prepared is ready; that is, the FAST Loader has been appropriately modified and the directory has been written onto block 1 of the DECtape.

To transfer a program from memory to the DECtape:

Load the program into memory.

Place the desired DECtape on a transport and select unit 1-WRITE.

Press I/O RESET.

Place the Writer in the paper tape reader, set the ADDRESS switches to 7600 (17600 for 8K systems), and press START*. The Writer will stop with all 1s in the AC.

Set the DATA (AC) switches to select the desired program. Selection is controlled by the leftmost switch that is up; all switches to the right of that one are ignored. (No switches up is equivalent to switch 0 up).

Press CONTINUE to execute the transfer. If no error occurs, the computer will halt with the AC clear. If an error occurs, the Writer will return to the beginning and stop with all 1s in the AC.

4.2.5 The FAST Reader

The FAST Reader (commonly designated FAST) occupies memory locations 7600 - 7777_8 (17600 - 17777_8 for 8K systems). FAST destroys the 66_8 locations immediately preceding the first location of FAST before the program is read from the DECtape. However, the program read-in may overlay all of these 66_8 locations. If several programs are to be loaded through repeated use of FAST, only the last program to be loaded may overlay the 66_8 locations mentioned above since each call to FAST destroys these locations.

4.2.5.1 **Standard FAST System Tape** – The FAST System DECtape (prepared for system program retrieval) distributed by Digital Equipment Corporation, allocates the first five DATA (AC) switches as follows:

Leftmost Switch Up	Program
0	Symbolic Tape Editor
1	CAP-15 Assembler
2	ODT (with all commands defined)

*If the area of core to be written will overlay the Writer, then the user must read in the low version of the Writer and set the Address switches to 100_8.

Leftmost Switch Up	Program
3	Paper Tape Lister
4	Paper Tape Duplicator

The user can, however, prepare different system DECtapes and working program DECtapes by the method described in paragraphs 4.2.3, 4.2.4, and 4.2.6.

All of the standard system programs as distributed by DEC are initiated automatically by FAST after they are loaded. All system programs, except for ODT, are not normally loaded into memory with user programs; thus, their memory requirements are of no concern. ODT is loaded with user programs. This program is an overlay of FAST*, and must be loaded by FAST after other programs have been loaded. The memory requirement for ODT is exactly the same as the object program distributed on paper tape.

Program	Load Address	Start Address	Size
ODT	6000	6000 (%ODT) 6011 (%REST)	1665

4.2.5.2 **The FAST Start** – These instructions are for use with DECtapes (similar to the one distributed by DEC) that have been properly prepared.

Place the FAST System DECtape on a transport and select unit 1-WRITE LOCK.

Assuming FAST is already in memory, set the ADDRESS switches to 7600 (17600 for 8K systems) and press START (this is known as FAST Start). The computer will halt with all 1s in the AC.

If FAST is not in memory, perform the following:

Press I/O RESET.

Place the FAST binary tape in the paper tape reader.

Set the ADDRESS switches to 7600 (17600 for 8K systems) and press START.

The computer will halt with all 1s in the AC.

Set the AC switches to select the program desired, according to the table of contents (written onto block 1 by the FAST loader) associated with the DECtape.

*The first 100_8 locations of FAST can be overlayed by a program read from DECtape.

Press CONTINUE. The DECtape will rewind to block 1, search forward to the appropriate block, and transfer the program. If no error occurs, control will transfer to the start of the program. If an error occurs, the computer halts with all 1s in the AC. Check for an incorrect setting of the DECtape switches. The DECtape status flags can be examined for possible malfunctions.

If the program just loaded did not overlay any part of FAST, then FAST is ready for the procedure beginning with "Set the AC switches". However, if the program did overlay FAST, FAST must be reloaded.

NOTE

> The program selected is determined by the leftmost AC switch that is up; the positions of other switches to the right are ignored by FAST. No switches up is equivalent to switch 0 up.

4.2.6 Example Writing A FAST System Tape

This example shows how to add a program to the standard system DECtape. The program to be added uses 426 locations beginning with location 2000. The start address is 2010. The binary tape is in the CAP-15 Loader format.

4.2.6.1 **Preparing the Directory** – The standard systems tape has a directory with five programs assigned to switches 0 through 4. This example will use switch 5 for the new program. The directory may be changed in two ways. The best method is to edit the source tape of the FAST Loader and assemble the new FAST Loader as described in paragraph 4.2.3. This always generates a hard copy of the new directory. The other method, which may be faster but is prone to errors, is to load the FAST Loader into memory, manually alter the proper directory entries and then execute the FAST Loader. In any case, the first 6 entries in the directory are *initially* as follows:

```
TABLE        TABLE        /DUMMY LOCATION
ZERO             1        /SWITCH 0: SYMBOLIC TAPE EDITOR
              7477
                22        /STARTING ADDRESS
ONE              1        /SWITCH 1 PDP-15/10 ASSEMBLER
              7477
                22
TWO          16000        /SWITCH 2 ODT-15
              1700
             16000        /OVERLAYS FAST


THREE            1        /SWITCH 3 PAPER TAPE LISTER
              1000
                22
FOUR             1        /SWITCH 4 PAPER TAPE DUPLICATOR
              2000
                22
```

```
FIVE          1          /SWITCH 5 UNASSIGNED
           7477
             22
```

The standard system programs will remain as they are. Therefore, the entries for switches 0 through 4 need not be changed. The entry for switch 5 will be changed to

```
FIVE       2000          /PROGRAM LOADED AT 2000
            426          /USES 426 WORDS
           2010          /INITIATE AT 2010
```

When the directory has been properly prepared, it may be written on DECtape as described in paragraph 4.2.3.

4.2.6.2 **Writing the Program on DECtape** – To add the program to the system DECtape, proceed as follows:

Place the prepared DECtape on unit 1 - WRITE.

Place the binary tape containing the program in the paper tape reader.

Set the ADDRESS switches to 7720 (or 7700) and press IO RESET and then START.

Set the ADDRESS switches to 7600 (17600 for 8K systems), place the HRI tape for the FAST Writer in the paper tape reader, and press READIN.

Set AC switch 5 up (all others down).

Press CONTINUE. The DECtape will rewind to block 1 and then will search forward to the proper block and write out the program.

If no errors occur, the Writer will halt with 0s in the AC. If it halts with all 1s, an error has occurred. The program will be written again if the user presses CONTINUE.

When the DECtape stops and the program halts with the 0 in the AC, the writing is complete. To check that the program may be read again, proceed as follows:

Change the DECtape unit to unit 1 - WRITE LOCK.

Load the FAST Reader by placing the tape in the paper tape reader. Set the Address switches to 7600 (17600 for 8K systems) and press READIN.

Set AC switch 5 up. Press CONTINUE to read the program. Control should be transferred to location 2010 when loading is complete. If an error occurs, the reader will halt with all 1s in the AC. In this case, press CONTINUE to read program A again.

4.2.7 Assembling The High And Low Writers

As mentioned previously, the FAST Writer exists in two versions. The high version occupies locations 7600 - 7777$_8$ (17600 - 17777$_8$ for 8K systems) and the low version occupies locations 100 - 277$_8$. There is only one source tape for the two versions of the Writer. The source code is conditionalized so that if the symbol LOW is defined, the low version is produced at assembly time; if undefined, the high version is produced. (See Part II, paragraph 1.4.7, for a complete description of conditional assemblies.) Thus, if the source tape is assembled by itself, the high version is produced since LOW is undefined. The low version is produced by preceding the source tape with a parameter tape containing the following two statements.

```
    LOW = 0↵

⇥.EOT↵
```

4.3 HRM PUNCHER

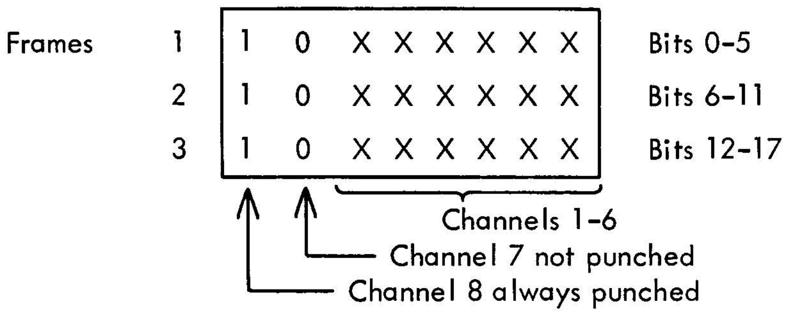
4.3.1 General Description

The Hardware Readin Mode (HRM) Puncher is a self-relocating dump program, written in CAP-15 Assembly Language. It can be loaded by means of the PDP-15/10 hardware readin (HRI) facility (*see PDP-15 Reference Manual*) into any block of memory. Once loaded, the HRM Puncher relocates itself and punches out a block of contiguous memory locations, specified by the user, in the HRI format. The HRM Puncher operates anywhere in up to 8K of memory.

4.3.2 Output Format

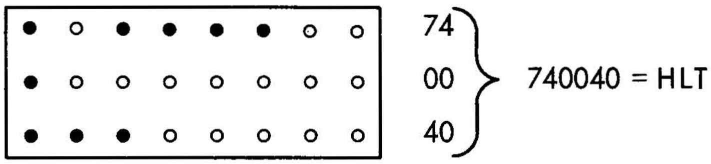
Binary output is punched in the HRI format which consists of the data words followed by the hardware readin word.

The data words correspond to consecutive memory words from the start address through the stop address. Each word is punched as three frames. Each frame has channel 8 punched and channel 7 not punched. Channels 1 through 6 contain six bits of the data word. The three frames correspond to the memory word as follows:



```
Frames  1 | 1 0 X X X X X X |  Bits 0-5
        2 | 1 0 X X X X X X |  Bits 6-11
        3 | 1 0 X X X X X X |  Bits 12-17
```

Following all the data words is the HRI word which halts the processor. This word is different than the data words in that channel 7 of the third frame is punched. The HRI word is punched as follows:



The HRI reads this word and executes the instruction, i.e., the processor halts.

4.3.3 **Functional Description**

The HRM Puncher is self-relocating and self-initializing. It uses several locations external to itself. These are:

Location 0

Locations 7766-7777, see below; (17766-17777 for 8K).

The source tape may be assembled under the control of two parameters to get four versions of the HRM puncher. The parameters are defined (or left undefined) at assembly time by means of a parameter tape (See Paragraph 1.4.7). The parameters are:

SLOW	Defined for low-speed punch (ASR). Undefined for high-speed punch.
V2	Defined to punch leader/trailer. Undefined to punch no leader or trailer.

The four versions are described below. The distributed object program, in the .FULL mode, is for the low-speed punch and produces leader/trailer.

4.3.3.1 **High-Speed Punch With No Leader/Trailer** – For this version, both parameters are undefined. The leader and trailer should be provided manually. This version requires 73_8 locations for the puncher, plus locations 0 and 7770-7777 for pointers and temporary storage.

4.3.3.2 **High-Speed Punch With Leader/Trailer** – This version is assembled with V2 defined and SLOW undefined. The memory requirements are 115_8 locations for the puncher and locations 0 and 7766-7777.

4.3.3.3 **Low-Speed Punch (ASR) With No Leader/Trailer** – This version is assembled with only SLOW defined. The memory requirements are 102_8 locations for the puncher and locations 0 and 7770-7777. The leader and trailer should be provided manually.

4.3.3.4 **Low-Speed Punch With Leader/Trailer** – Both V2 and SLOW are defined to assemble this version. The memory requirements are 124_8 locations for the puncher and locations 0 and 7766-7777.

4.3.3.5 **HRM Puncher** – The HRM puncher is divided into two almost equal parts. The first part performs the relocation and initialization (which includes the reading of the switches to set the start and stop addresses for the dump). The second part performs the actual dumping process.

4.3.4 Operating Procedure

The following paragraphs contain procedures for loading and starting the HRM puncher.

4.3.4.1 **Loading Procedure** – Proceed as follows to load the HRM puncher:

Press IO RESET.

Place the HRM puncher in the reader.

Set the address of where the HRM puncher is to be loaded in the ADDRESS switches.

Press the READIN key.

The load address must not be 0. The address must be selected so that the HRM puncher does not overlay locations 7766-7777. The size of the puncher is defined for the various versions in Section 4.3.3.

4.3.4.2 **Start-up Procedure** – The HRM puncher can be started and/or restarted at any time after it has been loaded. The procedure is:

Manually generate leader if necessary.

Press IO RESET.

Initiate at the selected load address by pressing START. The program will halt immediately.

Load the start address for the dump into the AC switches. Press CONTINUE. The program will halt immediately.

Load the stop address for the dump into the AC switches (this address must be greater than the start address). Press CONTINUE.

When punching is complete, the program will halt. Manually generate the trailer if necessary.

To repeat, or to punch out another area, repeat the procedure from the beginning.

4.4 PAPER TAPE HANDLING ROUTINES

4.4.1 Paper Tape Lister (PTLIST)

The Paper Tape Lister (PTLIST) is used to read an ASCII-coded paper tape from either the high-speed or low-speed paper tape reader, and to provide a character-by-character listing on the Teletype. Carriage return and line feed characters must be punched on the tape if these operations are to take place (they are not handled automatically by PTLIST). If a tab is encountered by PTLIST, it is converted to the appropriate number of spaces. Each tab stop is assumed to be every tenth print position.

When the program has been loaded by the PDP-15/10 Loader, it will type the following message on the Teletype:

PLEASE READY THE INPUT DEVICE AND SET THE AC SWITCH.

If input is to be from the low-speed (ASR) paper-tape reader, the user should set the reader switch to the ON position, set the AC switches to 400000, and depress the CONTINUE switch to start the listing. The program enters a wait loop when the reader runs out of tape. Inserting more input will cause the reader to continue. Typing CTRL U will cause the program to terminate*.

If input is to be from the high-speed paper tape reader, the user should set the AC switches to 0 and depress the CONTINUE switch to start the listing. The listing will terminate when the reader runs out of tape. At termination, the program types whatever is left in the input buffer and halts at location 21. Press CONTINUE to restart.

4.4.2 **Paper Tape Duplicator (PTDUP)**

The Paper Tape Duplicator (PTDUP) is used to duplicate and/or verify ASCII or binary paper tapes using the *high-speed paper tape reader and punch*. The program can also be used to punch a title on a tape that is being duplicated.

After the program has been loaded using the PDP-15/10 Loader, it prints the following message:

SWITCH? (M, V, or D)

The user should type the letter of the function that he wishes to perform, as described in the following paragraphs.

NOTE

At the end of the job the program halts at location 21. Press CONTINUE to restart.

M (Master Tape Duplicator) – This switch allows the user to type in a title and have that title punched in readable format preceding his duplicated tape. The following characters have a special meaning while typing in a title line.

ASCII Character	Action
212 LINE FEED	Punch a title line and return for more input.
215 CARRIAGE RETURN	Punch a title line and return for more input.
377 RUB OUT(S)	Ignore the previous character(s)

*CTRL U is formed by depressing the CTRL key while striking the U key.

ASCII Character	Action
375 ALT MODE	Punch title line and begin duplicating the tape (acts like D switch after it punches the title).
225 CTRL U	Ignore whatever is typed on this line and begin to type a new line.

D (Duplicate a Tape) – The following message is printed:

PARITY

The user should type Y if he wants even parity to be generated. Any other character will cause the paper tape image to be duplicated as is. If the high-speed reader runs out of tape, the program will terminate when it has completed punching the input buffer.

NOTE

The tape being duplicated must have at least one inch of trailer. The last ten frames are not duplicated.

V (Verify a Tape) – A parity check is performed on each frame. Frames not having even parity cause PARITY ERROR to be printed. The program will stop with the reader positioned at one frame past the frame in error. Depressing CONTINUE will cause the program to continue verifying.

At the end of duplication the input frame count and output frame count are printed in octal, as follows:

```
INPUT FRAME COUNT xxxxxx
OUTPUT FRAME COUNT xxxxxx
```

If the input and output frame counts are unequal, an error has occurred. The user should then restart the job.

NOTE

If the punch runs out of tape, the programs halts with all 1s in the AC. Refill the punch and continue.

The following message is always printed when verification has been completed.

```
PARITY ERROR = nnnnnn
```

where nnnnnn indicates the number of error frames in octal.

4.5 TELETYPE INPUT/OUTPUT ROUTINES

The Teletype Input/Output routines include the Teletype I/O Conversion (TICTOC) and the Decimal and Octal Print packages. Each of the routines in these packages is described in the following paragraphs. Each package should be assembled along with the program with which it is to run. The Teletype must be initialized by the user program to enable use of the Teletype I/O routines. This is accomplished by the statement TLS+10 at the beginning of the user program (see PDP-15 Reference Manual).

4.5.1 **Teletype I/O Conversion (TICTOC) Package**

The Teletype Input/Output Conversion (TICTOC) Package is used to convert 8-bit ASCII code to a 6-bit trimmed ASCII code, and vice versa. Formatting facilities are also available. Routines in the package fall into three main categories: input, output, and formatting. The routines in each of these categories are described in the following paragraphs, along with input and output formats and character sets.

NOTE

> When using output and formatting routines, ensure that all characters have been printed before halting. This can be accomplished by means of the "skip if teleprinter flag set" instruction TSF (see PDP-15 Reference Manual).

4.5.1.1 **Input Routines** – The input routines include %TIC and %TIC1. %TIC is used to input a string of characters from the Teletype and pack them three to a word into memory. The first, second, and third characters (in order of arrival) are packed into the left, middle, and right sections of a word. Subroutine %TIC requires two arguments for execution; the calling sequence is as follows:

```
%TIC        (PC-1)       LAW          (Stop Character)
            (PC)         JMS          %TIC
            (PC + 1)     .DSA         (Buffer Area)
            (PC + 2)     (Return)
```

Stop Character may be the trimmed ASCII (see Section 5.1.4) or 8-bit ASCII of any character. %TIC will only look at the rightmost six bits of the Stop Character. When this character is typed, %TIC stores it with the rest of the text and returns to the calling program at location (PC + 2). *Buffer Area* is the address of the first location of a block of storage into which the incoming text is to be packed. When the Stop Character (terminating character) is encountered, it is packed and the rest of the word filled out with zeros, if necessary. The user is not allowed to use the character @ as part of his text.

Four teletype keys have special meaning for %TIC:

LINE FEED (ASCII 212) – causes %TIC to ignore what has been typed and start over again. The input buffer address is reinitialized to receive the new text, and %TIC outputs a carriage-return and line feed to the Teletype. The tab count is set to 0.

AT SIGN (@ ASCII 300) – delimits the text externally. The @ performs the same function as the Stop Character. The rightmost six bits are stored with the rest of the text and, if necessary, the word is filled out with zeros. Return is made to location PC + 2 of the calling program (see calling sequence).

TAB KEY (ASCII 211) – TICTOC keeps a tab count. When the TAB key is struck, the Teletype spaces to the next TAB stop. The spaces are stored in the input buffer with only the rightmost six bits being packed. The tab count is then set to 0.

CARRIAGE RETURN (ASCII 215) – The tab count is cleared and a Teletype carriage return and line feed are executed. No data is packed and %TIC keeps listening. %TIC will not stop listening until either the Stop Character or an @ (at sign) has been typed. %TIC uses subroutine %TIC1 to get an 8-bit ASCII character. %TIC restores the AC and link before returning. Refer to Section 5.1.4 for list of valid %TIC characters.

%TIC1 Calling sequence:

```
(PC)        JMS         %TIC1       /Subroutine Call

(PC + 1)    (Return)                /Return with 8-bit ASCII
                                    character in AC.
```

Subroutine %TIC1 inputs a single 8-bit ASCII character from the Teletype. %TIC1 uses subroutine %TOC1 to echo the character that was typed on the keyboard.

4.5.1.2 **Output Routines** – The output routines include %TOC, %TOC1, and %TDIG.

%TOC – Subroutine %TOC is used to type out a string of text in the same format as described for %TIC (three 6-bit characters per word). Each 6-bit set is tested for being less than 40_8. If the 6-bit set is less than 40_8, then 300_8 is added to it to form an 8-bit ASCII character. 200_8 will be added to the 6-bit set if it is 40_8 or greater. When the 8-bit character has been built, %TOC will use subroutine %TOC1 to print it. The calling sequence is as follows:

```
(PC - 1)    LAW         (Stop Character)

(PC)        JMS         %TOC

(PC + 1)    .DSA        (Buffer Address)

(PC + 2)    (Return)
```

The AC and link are restored by subroutine %TOC. The buffer address is the address of the first word of the block of storage containing the text. The last character in the typing string is the Stop Character, which is not typed. As in %TIC, the rightmost six bits of the Stop Character are the only bits considered. If an at sign (@) is encountered, it will act as if it were the Stop Character.

%TOC1 – Subroutine %TOC1 outputs an 8-bit ASCII character to the teleprinter. A tab count is kept by %TOC1 which is used by subroutines %TABIT and %TIC. No assumption is made concerning the position of the teleprinter. %TOC1 types one character at the current position of the Teletype. The 8-bit character is not examined in any way before printing on the Teletype. The calling sequence is as follows:

```
(PC)        JMS         %TOC1       /Call with 8-bit character
                                    in AC.

(PC + 1)    (Return)                /Return with AC and link
                                    unchanged.
```

%TDIG – Subroutine %TDIG is used to type a single digit. The calling sequence is as follows:

```
(PC)          JMS          %TDIG          /AC must already be loaded.

(PC + 1)      (Return)                    /AC and link unchanged.
```

The AC must be set before calling %TDIG. The leftmost fourteen bits are stripped from the AC. The four bits left are added to 260_8 to form an 8-bit ASCII character. %TDIG uses subroutine %TOC1 to print the character.

4.5.1.3 **Formatting Routines** – The formatting routines include %CARR, %TABIT, and %SPACE. The calling sequence and description of each routine follows.

%CARR – Calling sequence:

```
(PC)          JMS          %CARR          /Subroutine call

(PC + 1)      (Return)
```

A carriage return (ASCII 215) and line feed (ASCII 212) are printed. The tab count is then set to 0.

%TABIT – Calling sequence:

```
(PC)          JMS          %TABIT         /Subroutine call

(PC + 1)      (Return)
```

The Teletype spaces to the next tab stop. The tab count is then set to 0. This count is kept automatically. %TIOCN, the number of spaces in a tab, is normally assembled as 8 (decimal), but may be altered by the user.

%SPACE – Calling sequence:

```
(PC)          JMS          %SPACE         /Subroutine call

(PC + 1)      (Return)
```

A space (ASCII 240) is printed on the Teletype. The accumulator and link are unchanged by calling the formatting routines.

4.5.1.4 **Input Format and Character Set** – The input characters read from the Teletype buffer by %TIC1 and %TIC are 8-bit ASCII. They are converted to 6-bit ASCII by stripping off the high-order 2-bits. The character set is as shown in Appendix I.

4.5.1.5 **Output Format and Character Set** – The 6-bit code is expanded to 8-bit ASCII by adding 200_8 or 300_8 to it. If the 6-bit number is less than 40_8, 300_8 is added; otherwise, 200_8 is added (see Appendix A).

4.5.2 **Decimal And Octal Print Package**

The Decimal and Octal Print Routines (%DIP, %OPT, %OPS, and %OPZ) are subroutines which dump the accumulator in either signed-decimal or octal mode.

4.5.2.1 **Decimal Integer Print (%DIP)** – This routine prints the signed decimal equivalent of an 18-bit binary number. The binary number to be printed must be loaded into the AC before calling %DIP. Insignificant 0s (leading 0s) are not printed; the routine types a space in place of each leading zero. In the case of a negative number, before typing the first significant figure, the routine types a minus sign. If the number is positive, the sign is omitted (space) and understood to be plus.

4.5.2.2 **Octal Print Subroutines (%OPS, %OPT, %OPZ)** – The Octal Print subroutines type the contents of the accumulator as an octal number, suppressing nonsignificant leading zeros if desired. Two methods of suppression are available through two entry points.

a. %OPS - The (JMS %OPS) entry causes leading zeros to be suppressed; the printed number occupying only the number of spaces needed to print all significant digits (left justified).

b. %OPT - The (JMS %OPT) entry causes leading zeros to be suppressed by blanks; the printed number is right justified in six spaces.

c. %OPZ - The (JMS %OPZ) entry causes no zero suppression to take place.

4.5.2.3 **Operation** – Input to the subroutines is provided in the accumulator by the user.

The subroutines are called directly:

```
(PC - 1)        LAC             NUM

(PC)            JMS             %DIP/%OPZ/%OPS/%OPT

(PC + 1)        (Return)
```

Control is returned to the user at location PC + 1.

The link and AC are restored upon return to the calling program.

The character set for the decimal and octal print routines is as follows.

%DIP: -,0,1,..9,Space

%OPS/%OPT 0,1,...7,Space

%OPZ: 0,1,..7

Examples

Input (AC)	**%DIP**	**%OPT**	**%OPZ**	**%OPS**
777777	-1	777777	777777	777777
377777	131071	377777	377777	377777
400000	-131072	400000	400000	400000

Input (AC)	%DIP	%OPT	%OPZ	%OPS
000000	0	0	000000	0
400055	-131027	400055	400055	400055
000055	45	55	000055	55

4.6 OCTAL DUMP ROUTINE

4.6.1 General Description

The Octal Dump Program allows the user to obtain either Teletype hard copy or paper tape output showing the contents of any register or set for registers that he specifies. The user specifies which registers are to be dumped via the Teletype keyboard as described in the following paragraph.

4.6.2 Input Format And Character Set

To obtain the dump of all memory locations between registers A (represented by xxx..) and B (represented by yyy..) where A ⩽ B, the user must type xxx..-yyy..-. A minus sign must be placed after each address; there is no need to type leading zeros in the address.

Examples

To dump from 10_8 to 25_8, type: 10-25-

To dump from 100_8 to 1377_8, type: 100-1377-

To obtain the contents of a single register, type the address twice, followed by a minus sign each time, as usual.

Examples

To obtain 10, type: 10-10-

To obtain 7725, type: 7725-7725-

To obtain the contents of all registers from address A up to the end of memory (7777 in a 4K machine, 17777 in an 8K machine) type address A followed by a slash.

Example

1500/

To print out memory locations between two registers A and B, type the smallest register first and the largest second, each followed by a minus sign.

Example

If A < B, type: A - B -
If A > B, type: B - A -

If the user types 6 characters and the sixth character is not a slash (/), or minus (-) the program will interpret this as an illegal character. The highest possible address is 7777_8 (or 17777_8 for an 8K machine).

Always place a minus sign after each address. If an illegal (not octal) character is typed, the program does not recognize it and types back a question mark (?) followed by a carriage return and line feed.

The input character set includes the numbers 0,1,...7, and the special characters / and -.

4.6.3 **Output Format And Character Set**

The output format appears as follows

```
LLLLL   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx

LLLLL   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx   xxxxxx
```

where the LLLLLs represent a five-character octal address in increments of 10_8, and the xxxxxx's represent the contents (in octal) of locations LLLLL to LLLLL + 7_8.

Example

```
00100   123014   530126   430765   110476   104572   271246   312467   764213

00110   453123   693530   511430   610110   222104   631221   745311   113457
```

If all locations in any line or group of lines of at least 10_8 locations are equal, the output would appear as follows.

```
00120   123456   711111   740040   000000   000000   123123   740040   740040

        740040   omitted

00160   740040   740040   740040   740040   740040   123123   600000   000000
```

The output character set includes only the digits 0 through 7_8.

4.6.4 **Operation**

DUMP is normally loaded starting at location 7300 (17300 for an 8K machine). The user can change this starting address by reassembling the DUMP program, preceded by a strip tape. The strip tape satisfies conditional assembly statements in the DUMP program by defining a start address. This is accomplished by setting the symbol LOW to the desired starting address.

Example

```
LOW = 22↵

 ⇥.EOT↵
(DUMP program follows)
```

The above example would cause the DUMP program to be assembled starting at location 22. If the symbol LOW is not defined, the default start address is location 7300 (17300 for an 8K machine).

After loading the program starts automatically. It types a carriage return/line feed and prints the following message:

```
        OUTPUT DEVICE =
```

The user must respond with H if the output is to go to the high-speed paper tape punch. Any other character will cause the program to assume the teleprinter as the output device. The program then outputs a carriage return/line feed and goes into a wait loop. The user should then type his parameters.

After the user's parameters have been satisfied, the program restarts automatically with a carriage return/line feed and waits for more parameters. If the user wishes to change the output device, he must restart the DUMP program at the original start address.

4.7 SCAN ROUTINE

4.7.1 General Description

SCAN is a small (100_8 locations) program used to scan areas of memory for a particular bit configuration. The user specifies the start and stop address for the area to be scanned, the bit configuration to look for, and the bit positions to be tested (e.g., a mask). SCAN then scans the area. When a match is found, the address of the match is printed along with the unmasked matching word. Proper selection of the operating parameters allows SCAN to be used as a dump. SCAN operates in either 4K or 8K without reassembly.

4.7.2 Output Format

The output of SCAN is a sequence of addresses and their contents printed on the Teletype. When SCAN is initiated, it prints a carriage return (↵) and a line feed (↓). As matches are found, the addresses and contents are printed as:

```
        AAAAA/XXXXXX↵↓
```

where AAAAA is the address (5 octal digits) and XXXXXX is the unmasked contents (6 octal digits). After the last line has been printed, another carriage return/line feed sequence is printed and SCAN halts.

Example

The following is an example of a SCAN output to search an area for all JMS instructions:

	Remarks
↵↓	
00200/100600↵↓	JMS 600 at location 200.

	Remarks (cont)
00377/100670↵↓	JMS 670 at location 377.
↵↓	END of SCAN.

4.7.3 **Functional Description**

SCAN is a self-contained and self-initializing program. Once loaded, it may be started and restarted at any time. SCAN never alters any locations outside of itself. SCAN consists of a main program and three subroutines. The actual scan is accomplished by the main program. All output is performed by the three closed subroutines. The SCAN process is performed for all locations between the beginning address and the end address as follows:

Fetch the contents of the core location.

Mask* it with the contents of MASK (specified by user).

Compare the result with the contents of WORD (specified by user).

If not equal, repeat for next core location. If the compare is equal, print the address and the unmasked contents.

In general, SCAN operates at I/O speed. If no matches are detected, a 4K area is scanned in the time it takes to print two carriage return/line feed sequences.

SCAN may be used as a dump or as a search for any particular bit configurations. For example, the user may:

Locate all instructions with a particular op code (such as all JMSs or all XCTs).

Locate all references to a particular address.

Locate all indirect references.

4.7.4 **Operating Procedure**

The following paragraphs describe the loading procedure and start-up procedure for the SCAN program.

4.7.4.1 **Loading Procedure** – SCAN is loaded by the hardware readin (HRI) facility of the PDP-15/10. The procedure is as follows.

Load the paper tape into the reader.

Set the address of BEGIN in the Address switches. This address is 7700 (17700 for 8K) on the supplied binary tape.

Depress the IO RESET key.

*"Mask" indicates an 18-bit Boolean AND function.

Depress the READIN key.

When loading is complete, the processor will halt.

4.7.4.2 **Start-Up Procedure** – The following procedure can be used at any time to start or restart SCAN. The purpose of this process is to set four parameters for SCAN. These are, in the order that they are set:

Location			
4K	**8K**		
7701	17701	WORD	/Bit configuration to look for.
7702	17702	MASK	/Specifies which bits to test. /(1 to test, a 0 to not test.)
7703	17703	BEGLOC	/First location to test.
7704	17704	ENDLOC	/Last location to test.

Initially, the first three values are 0 and the last is 7777. Once a value has been set, it does not have to be set again for another run if it is desired to use the same value again.

The Start-up procedure is as follows:

Press IO RESET key.

Set the beginning address of SCAN into the ADDRESS switches. This is 7700 for 4K or 17700 for 8K.

Press EXAMINE to store the address in the AR register.

Set the desired value for WORD into the AC switches. Operate the DEPOSIT NEXT key. This sets the value from the AC switches into memory.

NOTE

If it is not desired to change the present value of the parameter, step d may be replaced with the operation EXAMINE NEXT.

Repeat the preceding step for MASK, BEGLOC, and ENDLOC.

Operate the START key. This starts SCAN at location BEGIN.

To search for all references to 7200 between locations 500 and 1500:

(WORD) = 7200
(MASK) = 7777
(BEGLOC) = 500
(ENDLOC) = 1500

To dump all of the area between the BEGLOC and ENDLOC, specify WORD = 0 and MASK = 0. This causes a match, and associated printout, to occur at every location.

If a restart is desired and no parameters are to be set, only the first three steps and the sixth step need be performed. After the first three steps have been performed, the sixth step may be performed at any time to start the program.

CHAPTER 5
COMPACT MATHEMATICAL ROUTINES

5.1 INTRODUCTION

All mathematical routines in the PDP-15/10 COMPACT Software System library are described in this section. These routines are grouped in four major packages: Integer Arithmetic, Trigonometric Functions, Floating Point, and Floating Point I/O. Table 5-1 on page 5-19 provides a concise listing of their important characteristics. Each package, if used, must be assembled and loaded with the user program. The size and number of tapes furnished for each package are as follows.

Package	Size (decimal)	Number of Tapes
Integer Arithmetic	184	4
Trigonometric Functions	760	2
Floating Point	574	1
Floating Point I/O	480	1

Separate tapes are supplied for each of the four Integer Arithmetic routines. Trigonometric Functions are on two tapes - one single precision and one double precision. Each of the above tapes is terminated with a .EOT pseudo-op. All symbols that are internal to the math routines begin with a percent sign (%) so that the user can avoid using them as his own symbols.

If any of the trigonometric functions are not required by a user program, the user can specify that they are not to be assembled by means of a strip tape. The strip tape contains symbols that are used in conditional assembly statements, and is ended with a .EOT pseudo-op. This procedure can save a great deal of core space, and symbol table space, for users who do not require all of the trigonometric functions. The following symbols (set equal to 0) should be included on the strip tape, as desired, to specify that a function should not be assembled.

Single-Precision Functions

Function	Size (decimal)	Symbol
SQRT	34	%NSQRT
SIN	7	%NSIN
COS	11	%NCOS
%SIN (internal routine)	55	(both of the above)
EXP	69	%NEXP
ALOG	11	%NALOG
ALOG 10	11	%NLG10
%LOGS (internal routine)	51	(both of the above)
ATAN	50	%NATAN
TANH	33	%NTANH
POLY	32	%NPOLY

Double Precision Functions

DSQRT	35	%NDSQ
DSIN	7	%NDSIN
DCOS	12	%NDCOS
%DSIN (internal routine)	69	(both of the above)
DEXP	86	%NDEXP
DLOG	12	%NDLG
DLOG 10	12	%NDL10
%DLOGS (internal routine)	59	(both of the above)
DATAN	90	%NDTAN
DPOLY	35	%NDPOL

For example, if a user program did not require the functions SQRT, TANH, and ALOG 10, and wished to conserve core space, he would assemble a strip tape containing the following symbols before assembling his own program and the trigonometric functions.

```
%NSQRT=0↵

%NTANH=0↵
```

```
%NLG10=0↵

        ⇥.EOT↵
```

Assuming that the double-precision trigonometric functions were not required, the following functions would then be assembled.

SIN

COS

%SIN

EXP

ALOG

%LOGS

ATAN

POLY

The same procedure applies to the Floating Point Package and to Floating Point I/0. Both the single precision and double precision sections of the Floating Point Package require that the general floating point section be assembled with them. However, if either the single precision or double precision section of the Floating Point Package is not required by a user program, the user can delete them by including the symbols %NSING or %NDOUB on the strip tape. In like manner, if either the floating point input (FLIP) program or the floating point output (FLOP) program is not required in the Floating Point I/0 Package, the symbols %NFLIP or %NFLOP, respectively, should be included on the strip tape.

5.2 INTEGER ARITHMETIC SIMULATION

The Integer Arithmetic Simulation routines include multiply (MULT), logical multiply (LMUL), divide (DIV), and logical divide (LDIV). The purpose of these routines is to allow PDP-15/10 users to program using simulated multiply and divide instructions. A description of each routine is given in the following paragraphs. The reader is referred to Table 5-1 (Page 5-19) for a tabular summary of all COMPACT mathematical routines. Included in the summary for each routine is the name, mnemonic, calling sequence, function, errors, accuracy, timing, storage requirements, and pertinent comments.

5.2.1 Multiply Routines (MULT/LMUL)

The purpose of MULT is to multiply a signed 18-bit multiplicand by a signed 18-bit multiplier and produce a signed 36-bit product. Return is made with the low-order product in the AC and the high-order product in location %MHIGH.

The purpose of LMUL is to multiply a logical (unsigned) 18-bit multiplicand by a logical 18-bit multiplier and produce a logical 36-bit product. Return is made to the calling program with the low-order product in the AC and the high-order product in location %LMHY.

The calling sequence for the multiply routines is as follows.

```
PC-1        LAC         Multiplier

PC          JMS         MULT (or LMUL)

PC+1        LAC         Multiplicand

PC+2        (Return)
```

The algorithm used by the multiply routines is as follows: the least significant bit of the multiplier is tested; if it is equal to 1, the multiplicand is added to the developing product, which is then shifted right one bit position; if it equals 0, no addition is made before the shift. The process is repeated until all the bits of the multiplier, in order from least significant to most significant, have been processed.

Example

For this example, assume that register MP1 is the multiplier, MP2 is the multiplicand, and that the product will be developed in registers MP3 and MP1, combined. Each of these registers is 5 bits long for purposes of illustration. The multiplier (MP1) is equal to 9, and the multiplicand (MP2) is equal to 4.

MP3	MP1	MP2	Comments
00000	01001	00100	Initial state of registers. The least significant bit of MP1 is tested.
00100	01001	00100	Since it is a 1, the contents of MP2 are added to MP3.
00010	00100	00100	The combined contents of MP3 and MP1 are shifted right one position. The least significant bit of MP1 is tested.
00001	00010	00100	Since it is a 0, no addition takes place, the combined contents of MP3 and MP1 are shifted right one position, and the least significant bit of MP1 is tested again.
00000	10001	00100	Since it is a 0, no addition takes place, the combined contents of MP3 and MP1 are shifted right one position, and the least significant bit of MP1 is tested again.
00100	10001	00100	Since it is a 1, the contents of MP2 are added to the contents of MP3.
00010	01000	00100	The combined contents of MP3 and MP1 are shifted right one position and the least significant bit of MP1 is tested again.
00001	00100	00100	Since it is a 0, no addition takes place, the combined contents of MP3 and MP1 are shifted right one position, and the multiplication is complete (with a product of 36 in locations MP3 and MP1 combined).

5.2.2 **Divide Routines (DIV/LDIV)**

The purpose of DIV is to divide a signed 36-bit dividend by a signed 18-bit divisor. The signed 18-bit quotient that is developed is returned to the AC. The remainder, signed the same as the dividend, is returned in location %REM. When the magnitude of the divisor is equal to or less than that of the high-order dividend, no division takes place since the quotient cannot be expressed by an 18-bit signed integer. In this case, the program exits with the link bit set to 1. If division takes place, the link bit is set to 0 prior to exit.

The purpose of LDIV is to divide a logical (unsigned) 36-bit dividend by an unsigned 18-bit divisor. The 18-bit quotient that is developed is returned to the AC. The remainder is returned in location %LREM. When the high-order dividend is greater than or equal to the divisor, division does not take place, and the link bit is set to 1. If division takes place, the link bit is set to 0 prior to exit.

The calling sequence for the divide routines is as follows.

PC-1	LAC	Dividend (high order)
PC	JMS	DIV (or LDIV)
PC+1	LAC	Dividend (low order)
PC+2	LAC	Divisor
PC+3	(Return)	

The algorithm used by both division routines is binary long division, where the quotient is determined by a subtraction process. Unlike decimal long division, where a single quotient digit can be one of 10 numbers, in binary long division the quotient digit is either 1 or 0. To determine this digit, the divisor is subtracted from the dividend and if the remainder is negative, the quotient digit is 0, the remainder is ignored, the divisor is moved one place to the right with respect to the dividend, and the process is repeated. If the remainder is positive, the quotient is 1 and the remainder is used as the next dividend, as in ordinary long division. In either case, the divisor is moved one place to the right and the next bit from the original dividend is included with the new dividend. The following example demonstrates the algorithm. The divisor is equal to 5, and the dividend is equal to 45_8.

Example

```
Step 1           0
           101 /100101
                101
               -111
```

Result of division is negative; therefore, quotient is 0. Disregard remainder. Move divisor one place to right.

```
Step 2            01
           101 /100101
                 101
                +100
```

Result of division is positive; therefore, quotient is 1. Retain remainder as new dividend and bring down next digit from dividend.

Step 3

```
         011
  101 /100101
       101
       1000
        101
        +11
```

Result of subtraction is positive; therefore, quotient is 1. Retain remainder as new dividend and bring down next digit.

Step 4

```
         0111
  101 /100101
       101
       1000
        101
         111
         101
        +010
```

Final quotient is 111 with a remainder of 010.

In implementing the algorithm, the divide routines rotate the dividend left instead of moving the divisor right. No division occurs if the high-order dividend is greater than or equal to the divisor. This eliminates situations where the high-order dividend is divisible by at least one (which, if allowed to continue, would produce erroneous results).

NOTE

DIV does not check for the overflow condition that will occur when the high-order dividend is zero, the low-order dividend is greater than 377777 (bit zero is set), and the divisor is equal to 1.

5.3 TRIGONOMETRIC FUNCTIONS

Detailed algorithms for all trigonometric functions in the PDP-15/10 COMPACT Software System Mathematical Library are described in this chapter. Most of the functions are computed by methods of approximation as described by Cecil Hastings in his book *Approximation for Digital Computers.*

The trigonometric routines must be assembled along with the user program. The Floating Point Package (paragraph 5.4) must also be assembled to enable use of the trigonometric routines. Execution time for the trigonometric routines is greatly reduced if the EAE option is available since multiplication and division can then be accomplished by the hardware.

The reader is referred to Table 5-1 for a tabular summary of all PDP-15/10 mathematical routines. Included in the summary for each routine is the name, mnemonic, calling sequence, function, errors, accuracy, timing, and pertinent comments.

5.3.1 Square Root (SQRT, DSQRT)

The calling sequence for the square root routines is as follows.

```
PC             JMS                SQRT (or DSQRT)
```

```
PC+1          .DSA                ARG (+400000 if indirect)

PC+2          (Error Return)

PC+3          (Normal Return)
```

If the argument (ARG) is negative, return is made to the error return (PC+2) with the argument in the floatin accumulator and the AC set equal to 1; otherwise, return is made to location PC+3 with the square root of th argument in the floating accumulator.

A first-guess approximation of the square root of the argument is obtained as follows.

If the exponent (EXP) of the argument is odd:

$$P_0 = .5\left(\frac{EXP-1}{2}\right) + ARG\left(\frac{EXP-1}{2}\right)$$

If the exponent (EXP) of the argument is even:

$$P_0 = .5\left(\frac{EXP}{2}\right) + ARG\left(\frac{EXP - 1}{2}\right)$$

Newton's iterative approximation is then applied three times for SQRT or four times for DSQRT.

$$P_{i+1} = 1/2\left(P_i + \frac{ARG}{P_i}\right)$$

5.3.2 **Sine And Cosine (SIN, COS, DSIN, DCOS)**

The calling sequence for the sine and cosine routines is as follows.

```
PC            JMS                 SIN (or COS, DSIN, DCOS)

PC+1          .DSA                ARG (+400000 if indirect)

PC+2          (Error Return)

PC+3          (Normal Return)
```

If the integer portion of the product (ARG*$\pi/2$) is too large (i.e., the exponent of the product is greater than 21_8), return is made to the error return (PC+2) with the AC set equal to 3. Otherwise, return is made to location PC+3 with the sine or cosine of the argument in the floating accumulator.

The argument is converted to quarter circles by multiplying by $2/\pi$. The low two bits of the integral portion determine the quadrant of the argument and produce a modified value of the fractional portion (F) as follows.

Low 2 Bits	**Quadrant**	**Modified Value (Z)**
00	I	F
01	II	1-F
10	III	-F
11	IV	-(1-F)

Z is then applied to the following polynominal expression:

$$\sin x = \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$$

where n = 4 for SIN and COS, and n = 6 for DSIN and DCOS. The values of C are as follows.

SIN, COS	**DSIN, DCOS**
$C_1 = .157080 \times 10^1$	$C_1 = .157079633 \times 10^1$
$C_3 = -.645964 \times 10^0$	$C_3 = -.645964097 \times 10^0$
$C_5 = 796897 \times 10^{-1}$	$C_5 = .796926260 \times 10^{-1}$
$C_7 = -.467377 \times 10^{-2}$	$C_7 = -.468175300 \times 10^{-2}$
$C_9 = .151484 \times 10^{-3}$	$C_9 = .160438400 \times 10^{-3}$
	$C_{11} = -.359518435 \times 10^{-5}$
	$C_{13} = .544652850 \times 10^{-7}$

The argument for COS and DCOS routines is adjusted by adding $\pi/2$. The sine function is then used to compute the cosine as follows.

$$\cos x = \sin \left(\frac{\pi}{2} + x \right)$$

5.3.3 Exponential (EXP, DEXP)

The calling sequence for the exponential routines is as follows.

PC	JMS	EXP (or DEXP)
PC+1	.DSA	ARG (+400000 if indirect)
PC+2	(Error Return)	
PC+3	(Normal Return)	

If the integer portion of the product ARG*$\log_2 e$ is too large (i.e., the exponent of the product is greater than 21_8), return is made to the error return (PC+2) with the AC set equal to 3. Otherwise, return is made to location PC+3 with the exponential of the argument in the floating accumulator.

The function e^X is calculated as

$$_2x \log_2 e$$

which will have an integral portion (I) and a fractional portion (F). Then

$$e^X = (2^I)(2^F)$$

where

$$2^F = \left(\sum_{i=0}^{n} C_i F^i \right)^2$$

and n = 6 for EXP, and n = 8 for DEXP. The values of C are as follows.

EXP	DEXP
$C_0 = .100000 \times 10^1$	$C_0 = .100000000 \times 10^1$
$C_1 = .346574 \times 10^0$	$C_1 = .346573590 \times 10^0$
$C_2 = .600566 \times 10^{-1}$	$C_2 = .600566267 \times 10^{-1}$
$C_3 = .693801 \times 10^{-2}$	$C_3 = .693801368 \times 10^{-2}$
$C_4 = .601130 \times 10^{-3}$	$C_4 = .601133075 \times 10^{-3}$
$C_5 = .416700 \text{ v } 10^{-4}$	$C_5 = .416670330 \times 10^{-4}$
$C_6 = .240977 \times 10^{-5}$	$C_6 = .240678700 \times 10^{-5}$
	$C_7 = .119610000 \times 10^{-6}$
	$C_8 = .518000000 \times 10^{-8}$

5.3.4 Natural And Common Logarithms (ALOG, ALOG10, DLOG, DLOG10)

The calling sequence for the logarithm routines is as follows.

```
PC                 JMS                 ALOG (or ALOG10, DLOG, DLOG10)
```

```
PC+1          .DSA               ARG (+400000 if indirect)

PC+2          (Error Return)

PC+3          (Normal Return)
```

If the argument is less than or equal to zero, return is made to the error return (PC+2) with the AC set equal to 2. Otherwise, return is made to location PC+3 with the result in the floating accumulator.

The exponent of the argument is saved as one greater than the integral portion of the result. The fractional portion of the argument is considered to be a number between 1 and 2. Z is computed as follows:

$$Z = \frac{x - \sqrt{2}}{x + \sqrt{2}}$$

then

$$\log_2 x = 1/2 + \left(\sum_{i=0}^{n} C_{2i+1} \; Z^{2i+1} \right)$$

where n = 2 for ALOG and ALOG10, and n = 3 for DLOG and DLOG10.

The values of C are as follows.

ALOG & ALOG10	**DLOG & DLOG10**
$C_1 = .288539 \times 10^1$	$C_1 = .288539007 \times 10^1$
$C_3 = .961471 \times 10^0$	$C_3 = .961800762 \times 10^0$
$C_5 = .598979 \times 10^0$	$C_5 = .576584342 \times 10^0$
	$C_7 = .434259751 \times 10^0$

Finally, $\log_e x = (\log_2 x)(\log_e 2)$ for ALOG and DLOG

and $\log_{10} x = (\log_2 x)(\log_{10} 2)$ for ALOG10 and DLOG10.

5.3.5 Arc Tangent (ATAN, DATAN)

The calling sequence for the arc tangent routines is as follows.

```
PC            JMS                ATAN (or DATAN)
PC+1          .DSA               ARG (+400000 if indirect)
PC+2          (Return)
```

Return is made to location PC+2 with the arc tangent of the argument in the floating accumulator. There are no error conditions.

For x less than or equal to 1, Z = x, and

$$\text{arc tangent } x = \left(\sum_{i=0}^{n} C_{2i+1} \; Z^{2i+1} \right)$$

where n = 8 for ATAN, and n = 3 for DATAN. For x greater than 1, Z = 1/x, and

$$\text{arc tangent } x = \pi/2 - \left(\sum_{i=0}^{n} C_{2i+1} \; Z^{2i+1} \right)$$

where n = 7 for ATAN, and n = 3 for DATAN. The values of C are as follows.

ATAN	DATAN
$C_1 = .999999 \times 10^0$	$C_1 = .999215000 \times 10^0$
$C_3 = -.333299 \times 10^0$	$C_3 = -.321181900 \times 10^0$
$C_5 = .199465 \times 10^0$	$C_5 = .146276600 \times 10^0$
$C_7 = -.139085 \times 10^0$	$C_7 = -.389929000 \times 10^{-1}$
$C_9 = .964200 \times 10^{-1}$	
$C_{11} = -.559099 \times 10^{-1}$	
$C_{13} = .218612 \times 10^{-1}$	
$C_{15} = -.405406 \times 10^{-2}$	

To get full 34-bit accuracy in DATAN, the tangent of the first approximation is taken, and the small angle theory is then used to minimize the error angle as follows:

$$\arctan x = P - \frac{\tan (P - x)}{1 + x \tan P}$$

where P is the result of the first approximation.

5.3.6 **Hyperbolic Tangent (TANH)**

The calling sequence for the hyperbolic tangent routine is as follows.

```
PC          JMS               TANH

PC+1        .DSA              ARG (+400000 if indirect)

PC+2        (Error Return)

PC+3        (Normal Return)
```

If the integer portion of the product x * $\log_2 e$ is too large (i.e., the exponent of the product is greater than 21_8), return is made to the error return (PC+2) with the AC set equal to 3. Otherwise, return is made to location PC+3 with the hyperbolic tangent of the argument in the floating accumulator.

The function

$$\tanh \; |x| = \left(1 - \frac{2}{1 + e^{2|x|}}\right)$$

e^x is computed as $2^{x \log_2 e}$ which will have an integral portion (I) and a fractional portion (F). Then,

$$e^x = (2^I)(2^F)$$

where $$2^F = \left(\sum_{i=0}^{n} C_i F^i\right)^2$$ and n = 6.

The values of C are as follows.

$C_0 = .100000 \times 10^1$

$C_1 = .346574 \times 10^0$

$C_2 = .600566 \times 10^{-1}$

$C_3 = .693801 \times 10^{-2}$

$C_4 = .601130 \times 10^{-3}$

$C_5 = .416700 \times 10^{-4}$

$C_6 = .240977 \times 10^{-5}$

5.3.7 **Polynomial Evaluation (POLY, DPOLY)**

The calling sequence for the polynomial evaluation routine is as follows.

```
PC          JMS               POLY (or DPOLY)

PC+1        .DSA              PLIST

PC+2        (Return)
```

The value of Z must be loaded in the floating accumulator prior to calling POLY or DPOLY. PLIST refers to a list of constants stored in contiguous locations within the calling program as follows:

PLIST	-N	/2's complement of number or terms
	C_n	/Last term
	C_{n-1}	
	.	.
	.	.
	.	.
	C_0	/First term

Return is made to location PC+2 with the result in the floating accumulator. There are no error conditions.

The polynomial

$$x = \left(\sum_{i=0}^{n} C_{2i+1} Z^{2i+1} \right)$$

is evaluated as follows.

$$x = Z(C_0 + Z^2 (C_1 \ldots \qquad \ldots + Z^2 (C_n Z^2 + C_{n-1})))$$

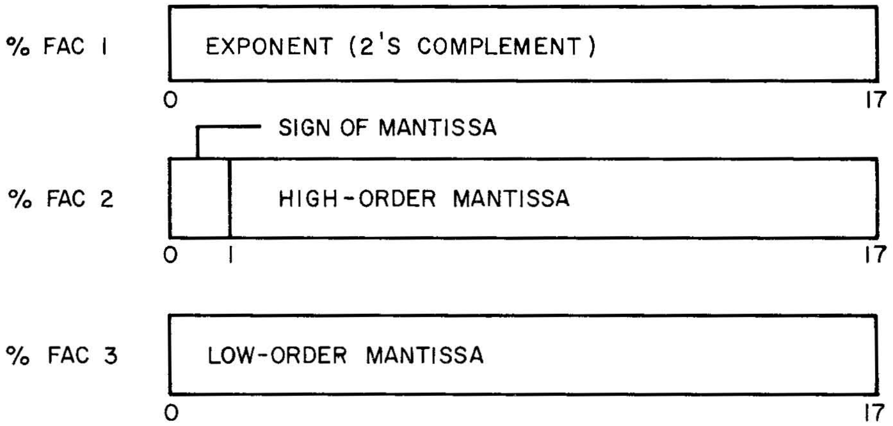
5.4 FLOATING POINT PACKAGES

The purpose of the Floating Point Package is to allow PDP-15/10 users to program for floating-point data using simulated floating-point instructions. The Floating Point Package must be assembled and loaded with the user program at run time. All simulated floating-point instructions exist as subroutines within this package and must be called as such by the user program.

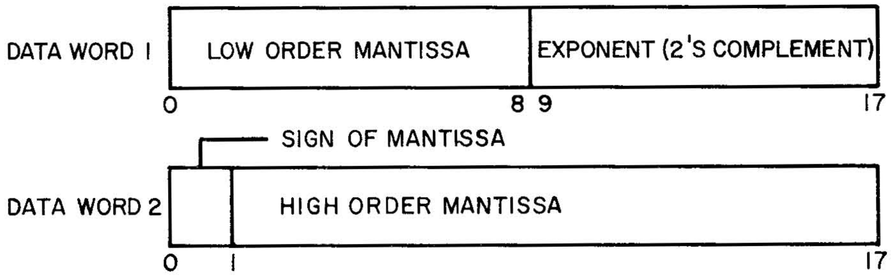
The reader is referred to Table 5-1 for a tabular summary of all PDP-15/10 mathematical routines. Included in the summary for each routine is the name, mnemonic, calling sequence, function, errors, accuracy, timing, storage requirements, and pertinent comments. The following paragraphs provide a brief description of the floating-point accumulator, and single-precision, double-precision, and general floating-point operations.

5.4.1 Floating Point Accumulators

The floating accumulator is a software accumulator which is included in the Floating Point Package. It is a three-word accumulator, %FAC1 being the label of the first word, %FAC2 the second, and %FAC3 the third. Floating-point data is stored in the floating accumulator in the following format; negative mantissae are indicated by the setting of bit 0 of word %FAC2.



Used by both single- and double-precision routines, this format is also the general format of double-precision numbers. Single-precision numbers have a different format and must be converted before and after use in the floating accumulator. This conversion is taken care of automatically by the floating arithmetic load and store routines FLAC and FDAC (see Table 5-1). The format of single-precision numbers is as follows.



5.4.2 **Single Precision Operations**

Single-precision routines perform floating-point operations on double-word quantities as described in paragraph 5.4.1. The arithmetic operations are performed with one operand in the floating accumulator and the other operand taken from storage. The computed result is developed in the floating accumulator, and is accurate to 26 bits.

All routines in the single-precision floating-point package are summarized in Appendix A. In general, the calling sequence for arithmetic routines is as follows:

```
JMS           SUBR

.DSA          ARG2

(Return)
```

where ARG2 is the address of the first location of the argument from storage. The single-precision load and store routines (FLAC and FDAC) must be used to load ARG1 into the floating accumulator prior to calling an arithmetic routine, and to store the result upon return.

5.4.3 Double Precision Operations

Double-precision routines perform floating-point operations on triple-word quantities as described in paragraph 5.4.1. The arithmetic operations are performed with one operand in the floating accumulator and the other operand taken from storage. The computed result is developed in the floating accumulator, and is accurate to 34 bits.

All routines in the double precision floating point package are summarized in Appendix A. The calling sequence is the same as for single-precision routines except that the double-precision load and store routines (DLAC and DDAC) must be used for loading and storing the floating accumulator.

5.4.4 General Floating Point Operations

General floating-point routines perform the actual computation for arithmetic operations set up by the individual single- and double-precision routines. All arithmetic operations are performed with one operand in the floating accumulator and the other in the held accumulator (%HAC1 - %HAC3). All floating-point operations are performed on normalized operands.

All general floating point routines are summarized in Table 5-1. The calling sequence for these routines is simply

```
PC          JMS          SUBR

PC+1        (Return)
```

except for the unfloat routine %FUNF. This routine is called as follows:

```
PC          JMS              %FUNF.

PC+1        (Error Return)

PC+2        (Normal Return)
```

where the error return (PC+1) is taken if there is an attempt to unfloat a number that has an integer portion too large for an 18-bit word. If the error return is taken, the AC is set equal to 3. Otherwise, return is made to PC+2 with the result in the AC.

5.5 FLOATING POINT I/O

5.5.1 General Description

The floating-point input/output (FLIO) program is designed to allow the user to input and output decimal data in floating-point format. The program is designed to operate with either the high-speed paper tape reader/punch or the ASR33 Teletype. All data transfers are handled through the floating accumulator. FLIO should be assembled and loaded along with the Floating Point Package (FPOINT) and the user program. The user program must initialize (select) the device used by FLIO; for example, to select teleprinter output, "TSL+10" must appear at the beginning of his program (See PDP-15 Reference Manual).

For input, the user program must specify the device as the only argument when calling FLIP (floating-point input). The input field is delimited by:

Tab	211
Line Feed	212
Carriage Return	215
Slash	257

The second number of the exponent will also delimit the field. Spaces and illegal characters are ignored until the first digit is encountered. If the decimal number is too large, the rightmost extra digits are ignored until the exponent or a delimiting character is typed.

For output, the user program must specify the device and the number of digits to be output when calling FLOP (floating-point output). Output takes place at the current device position; that is, spacing and other formatting functions are not automatically performed.

The following represents device assignments for both input and output.

ASR33 Teletype	000000
High-Speed Reader/Punch	400000

The calling sequence for FLIP is as follows.

```
JMS              FLIP          /Subroutine call
xxxxxx                         /Device specification
Next Instr.                    /Return
```

The calling sequence for FLOP is as follows.

```
JMS              FLOP          /Subroutine call
xxxxxx+n                       /Device specification plus no.
                               /of digits to be output (octal)
Next Instr.                    /Return
```

NOTE

Again, for both input and output, data transfers are processed through the floating accumulator.

5.5.2 **Input Format And Character Set**

The input format is flexible. The normal form is ± .DDDD... ± EE where the Ds are decimal digits representing the mantissa (a maximum of nine digits) and the Es are decimal digits representing the exponent (two digits). Any meaningful variation of the normal form is permitted on input. If the decimal point is omitted, the number is assumed to be integral. If more than one decimal point is in the input stream, the last one to appear is the one that will be used. A plus or minus sign after at least one number of the mantissa field will signal the program that the next two digits are exponent digits.

The input character set is as follows.

+

-

Space

0, 1. . . .9

/

E

Tab

Carriage Return

Line Feed

Examples

123.123E4	=	1231230.
.123123-4	=	.0000123123
-1.23.123	=	-123.123
+123123	=	123123.

5.5.3 **Output Format And Character Set**

The output format is rigid. The user can specify as many digits to be output as he wishes between one and nine (inclusive).* After the last mantissa is typed/punched, a plus is output if the exponent is positive and a minus sign is output if the exponent is negative. The output character set includes the following.

+

.

-

0, 1. . .9

Space

*If the argument to FLOP is 0, an "x" is printed, if the argument specifies more than nine decimal digits, an "x" is printed along with the first nine digits.

Examples

```
.12312300+04

.123-4

.1+10
```

All output is rounded. For example,

```
JMS                                     FLOP

6
```

indicates that the user wishes to output six decimal digits. If his internal number is .123456789E12, his output would be rounded as follows.

```
.123457+E12
```

Table 5-1. Summary of PDP-15/10 COMPACT Mathematical Routines

Routine Name	Mnemonic	Calling Sequence	Function	Errors*	Accuracy Bits	Timing*** Non-EAE	EAE	Storage (Decimal)	Comments
Integer Arithmetic								*184*	Signed high-order product in location %MHIGH.
Multiply	MULT	LAC Multiplier JMS MULT/LMUL LAC Multiplicand	I*J	None	17	447 µs	N.A.	55	Low-order product in AC. Unsigned high-order product in location %LMHY.
Logical Multiply	LMUL		I * J	None	18	404 µs	N.A.	29	Low-order product in AC.
Divide	DIV	LAC Dividend (high order) JMS DIV/LDIV	I/J	I > J, Link set	17	483 µs	N.A.	64	Signed remainder in location %REM.
Logical Divide	LDIV	LAC Dividend (low order) LAC Divisor	I/J	I > J, Link set	18	414 µs	N.A.	36	Unsigned remainder in location %LREM.
Trigonometric Functions								*760*	Any trigonometric function can be deleted by conditional assembly, see Chapter 1
Square Root:									
SP Square Root	SQRT	JMS SQRT/DSQRT	$x^{1/2}$	#1	26	8.488 ms	3.522 ms	34	
DP Square Root	DSQRT	.DSA Address of arg. (Error Return) (Normal Return)	$x^{1/2}$	#1	34	13.907 ms	5.585 ms	35	
Sine:									
SP Sine	SIN	JMS SIN/DSIN	sin (x)	#3	26	11.460 ms	4.814 ms	7	Calls POLY and %SIN**
DP Sine	DSIN	.DSA Address of arg. (Error Return) (Normal Return)	sin (x)	#3	34	17.481 ms	9.441 ms	7	Calls DPOLY and %DSIN**
Cosine:									
SP Cosine	COS	JMS COS/DCOS	cos (x)	#3	26	11.895 ms	5.269 ms	11	Calls POLY and %SIN**
DP Cosine	DCOS	.DSA Address of arg. (Error Return) (Normal Return)	cos (x)	#3	34	18.135 ms	9.894 ms	12	Calls DPOLY and %DSIN**
Exponential:									
SP Exponential	EXP	JMS EXP/DEXP	e^x	#3	26	14.471 ms	7.944 ms	69	
DP Exponential	DEXP	.DSA Address of arg. (Error Return) (Normal Return)	e^x	#3	34	27.107 ms	9.377 ms	86	
Natural Logarithm:									
SP Natural Logarithm	ALOG	JMS ALOG/DLOG	$\log_e x$	#2	26	11.099 ms	5.817 ms	11	Calls POLY and %LOGS**
DP Natural Logarithm	DLOG	.DSA Address of arg. (Error Return) (Normal Return)	$\log_e x$	#2	34	17.928 ms	6.461 ms	12	Calls DPOLY and %DLOGS**

ee footnotes at the end of table.

Table 5-1. Summary of PDP-15/10 COMPACT Mathematical Routines (cont)

Routine Name	Mnemonic	Calling Sequence	Function	Errors*	Accuracy Bits	Timing***		Storage (Decimal)	Comments
						Non-EAE	EAE		
Trigonometric Functions (cont)									
Common Logarithm:									
SP Common Logarithm	ALOG10	JMS ALOG10/DLOG10	$\log_{10} x$	#2	26	11.099 ms	5.817 ms	11	Calls POLY and %LOGS**
DP Common Logarithm	DLOG10	.DSA Address of arg.	$\log_{10} x$	#2	34	17.928 ms	6.461 ms	12	Calls DPOLY and %DLOGS**
		(Error Return)							
		(Normal Return)							
Arc Tangent:									
SP Arc Tangent	ATAN	JMS ATAN/DATAN	$\tan^{-1}(x)$	None	26	14.115 ms	7.736 ms	50	Calls POLY
DP Arc Tangent	DATAN	.DSA Address of arg.	$\tan^{-1}(x)$	None	34	54.540 ms	24.075 ms	90	Calls POLY and %DSIN**
		(Normal Return)							
Hyperbolic Tangent	TANH	JMS TANH	tanh(x)	#3	26	17.910 ms	9.737 ms	33	
		.DSA Address of arg.							
		(Error Return)							
		(Normal Return)							
Polynomial Evaluation:									
SP Poly. Eval.	POLY	JMS POLY/DPOLY		None	N.A.	****	****	32	Called by SIN, COS, ALOG ALOG10, and ATAN
DP Poly. Eval.	DPOLY	.DSA PLIST	$x = \sum_{i=0}^{n} C_{2i+1} Z^{2i+1}$	None	N.A.	*****	*****	35	Called by DSIN, DCOS, DLOG DLOG 10, and DATAN
		(Normal Return)							
		. .							
		. .							
		. .							
		PLIST -N /2's comp. of no. of terms							
		C_n /Last term							
		C_n† C_{n-1}							
		.							
		.							
		.							
		C_0 /First term							
Floating Point								*574*	Either SP or DP Arithmetic can be deleted by conditional assembly. If either of these is
SP Arithmetic:		*Arg. 1 (FL.AC)* *Arg. 2*							
Add	FAD	Augend Addend JMS SUBR	A+B → FL.AC	None	26	415.5 µs	393 µs		
Subtract	FSUB	Minuend Subtrahend .DSA Arg. 2	A-B → FL.AC	None	26	513 µs	457.5 µs		
Multiply	FMPX	Multiplicand Multiplier	A*B → FL.AC	None	26	1.482 ms	421.5 µs		

See footnotes at the end of table.

Table 5-1. Summary of PDP-15/10 COMPACT Mathematical Routines (cont)

Routine Name	Mnemonic	Calling Sequence			Function	Errors*	Accuracy Bits	Timing***		Storage (Decimal)	Comments
								Non-EAE	EAE		
Divide	FDVD	Dividend	Divisor	JMS SUBR .DSA Arg. 2	R/B → FL.AC	None	26	1.986 ms	481.5 μs	93	assembled, the General Floating Point tape must also be assembled. See Chapter 1.
Reverse Subtract	FSBR	Subtrahend	Minuend		B-A → FL.AC	None	26	510 μs	445.5 μs		
Reverse Divide	FDVR	Divisor	Dividend		B/A → FL.AC	None	26	2.211 ms	556.5 μs		
Load	FLAC		Address		Arg. → FL.AC	None	N.A.	105 μs	105 μs		
Store	FDAC	Value	Address		FL.AC → Arg.	None	N.A.	79.5 μs	79.5 μs		
DP Arithmetic		*Arg. 1 (FL.AC)*	*Arg.2*								
Add	DFAD	Augend	Addend		A+B → FL.AC	None	34	463.5 μs	385.5 μs		
Subtract	DSUB	Minuend	Subtrahend		A-B → FL.AC	None	34	516 μs	450 μs		
Multiply	DMPY	Multiplicand	Multiplier		A*B → FL.AC	None	34	2.031 ms	411 μs		
Divide	DDVD	Dividend	Divisor	JMS SUBR	A/B → FL.AC	None	34	2.336 ms	471 μs		
Reverse Subtract	DSBR	Subtrahend	Minuend	.DSA Arg. 2	B-A → FL.AC	None	34	519 μs	435 μs	98	
Reverse Divide	DDVR	Divisor	Dividend		B/A → FL.AC	None	34	2.840 ms	696 μs		
Load	DLAC		Address		Arg. → FL.AC	None	N.A.	94.5 μs	94.5 μs		
Store	DDAC	Value	Address		FL.AC → Arg.	None	N.A.	69 μs	69 μs		
General Operations:											
Negate Hardware AC	%ANEG.	JMS	%ANEG.		-(AC)→AC	None	N.A.	7.5 μs	7.5 μs		
Negate Floating AC	%FNEG	JMS	%FNEG.		-(FL.AC)→FL.AC	None	N.A.	12 μs	12 μs		
Exchange Floating AC with Held AC	%SWITCH	JMS	%SWITCH		(FL.AC)⇄(Held AC)	None	N.A.	57 μs	57 μs		
General Floating Multiply	%FM	JMS	%FM.		(FL.AC)*(Held AC)	None	34	2.1 ms	216 μs		Product in Floating Accumulator
Get Address	%FG.	JMS	%FG.			None	N.A.	38 μs	38 μs		
Gen. Floating AC	%FA.	JMS	%FA.		(FL.AC)+(Held AC)	None	34	207 μs	201 μs	383	Sum in Floating Accumulator
Normalize Floating AC	%FNOR.	JMS	%FNOR.			None	N.A.	363 μs	60 μs		
Hold Floating Add	%FH.	JMS	%FH.		(FL.AC)→(Held AC)	None	N.A.	21 μs	21 μs		
Sign Control	%FS.	JMS	%FS.			None	N.A.	42 μs	42 μs		
Round and Insert Sign	%FIR.	JMS	%FIR.			None	N.A.	39 μs	39 μs		
Gen. Floating Divide	%FD.	JMS	%FD.		(Held AC)/(FL.AC)	None	34	2.1 ms	276 μs		Quotient in Floating Accumulator
Float	%FLOT.	JMS	%FLOT.		(AC)→(FL.AC)	None	N.A.	405 μs	95 μs		
Unfloat	%FUNF.	JMS	%FUNF.		(FL.AC)→(AC)	#3	N.A.	46 μs	96 μs		
		(Error Return)									
		(Normal Return)									
Floating Point I/O										*480*	Either FLIP or FLOP can be deleted by conditional assembly, see Chapter 1.
Floating Point Input	FLIP	JMS FLIP	/Subr call		Input FL.PT.	******	N.A.	N.A.	N.A.	181	
		xxxxxx	/Device								

See footnotes at the end of table.

Table 5-1. Summary of PDP-15/10 COMPACT Mathematical Routines (cont)

Routine Name	Mnemonic	Calling Sequence		Function	Errors*	Accuracy Bits	Timing***		Storage (Decimal)	Comments
							Non-EAE	EAE		
Floating Point Output	FLOP	JMS FLOP xxxxxx	/Subr.call /Device + no. of digits to be output	Output FL.PT.	None	N.A.	N.A.	N.A.	183	Device is specified as: 000000 for TTY 400000 for high-speed paper-tape reader punch

*For errors designated as #n, the error number is stored in the AC, and the Error Return is taken. Errors result in attempts to 1) take the square root of a negative number, 2) take the Log_x of an argument <0, and 3) unfloat a number, the integer portion of which is too large for an 18-bit word.

**Internal routines. %SIN and %DSIN called by SIN/COS, DSIN/DCOS, and DATAN. %LOGS and %DLOGS called by ALOG/ALOG 10, DLOG/DLOG10. Storage requirements for %SIN, %DSIN, %LOGS, and %DLOGS are 55, 69, 51, and 59 locations, respectively.

***Timings indicated are estimated and represent average-to-worst-case times.

****Non-EAE = 2.862 ms + 1.581C; EAE = 1.128 ms + .896C. C = number of coefficients.

‘****Non-EAE = 4.322 ms + 2.482C; EAE = 1.076 ms + .876C. C = number of coefficients.

‘****Flip will ignore any characters after the first nine significant digits, and will continue to do so until a delimiting character is read.

APPENDIX A
CHARACTER SET

Printing Character	7-bit ASCII	6-bit Trimmed ASCII**	Printing Character	7-bit ASCII	6-bit Trimmed ASCII**
@†	100	00††	(Space)	040	40
A	101	01	!	041	41
B	102	02	"†	042	42
C	103	03	#†	043	43
D	104	04	$†	044	44
E	105	05	%	045	45
F	106	06	&	046	46
G	107	07	,†	047	47
H	110	10	(	050	50
I	111	11	)	051	51
J	112	12	*	052	52
K	113	13	+	053	53
L	114	14	,	054	54
M	115	15	-	055	55
N	116	16	.	056	56
O	117	17	/	057	57
P	120	20	0	060	60
Q	121	21	1	061	61
R	122	22	2	062	62
S	123	23	3	063	63
T	124	24	4	064	64
U	125	25	5	065	65
V	126	26	6	066	66
W	127	27	7	067	67
X	130	30	8	070	70
Y	131	31	9	071	71
Z	132	32	:†	072	72
[†	133	33	;	073	73
\	134	34	<†	074	74
]†	135	35	=	075	75
↑†	136	36	>†	076	76
←†	137	37	?†	077	77
Null (Blank)	000				
Horizontal Tab	011				
Line Feed	012				
Vertical Tab	013				
Form Feed	014				
Carriage Return	015				
Rubout	177				

Notes: All other characters are illegal and are flagged and ignored

† = Illegal as source except in a comment or text; Ignored by %TOC and %TOC1.

†† = Ignored by %TOC and %TOC1.

** = .SIXBT pseudo-op text value.

APPENDIX B
PERMANENT SYMBOL TABLE

Memory Reference

CAL	000000
DAC	040000
JMS	100000
DZM	140000
LAC	200000
XOR	240000
ADD	300000
TAD	340000
XCT	400000
ISZ	440000
AND	500000
SAD	540000
JMP	600000

Operate

OPR	740000
NOP	740000
CMA	740001
CML	740002
OAS	740004
RAL	740010
RAR	740020
IAC	740030
HLT	740040
XX	740040
SMA	740100
SZA	740200
SNL	740400
SML	740400
SKP	741000
SPA	741100
SNA	741200
SZL	741400
SPL	741400
RTL	742010
RTR	742020
SWHA	742030
CLL	744000
STL	744002
CCL	744002
RCL	744010
RCR	744020
CLA	750000

Operate (Cont)

CLC	750001
LAS	750004
LAT	750004
GLK	750010
LAW	760000

I/O States

IOT	700000
IORS	700314
CAF	703302

Interrupt

IOF	700002
ION	700042

Teletype Keyboard

KSF	700301
KRB	700312
KRS	700322

Teletype Teleprinter

TSF	700401
TCF	700402
TLS	700406

Paper Tape Reader

RSF	700101
RCF	700102
RSA	700104
RRB	700112
RSB	700144

Paper Tape Punch

PSF	700201
PCF	700202
PSA	700204
PSB	700244

Index and Limit Register Instructions

AAS	720000
AAC	723000
AXS	725000
CLLR	736000
AXR	737000

CLX	735000
PAL	722000
PAX	721000
PLA	730000
PLX	731000

Index and Limit Register Instructions (cont)

PXA	724000
PXL	726000

Mode Switching and Listing Instructions

EBA	707724
DBA	707722
SBA	707721

Index Register

X	10000

APPENDIX C
SUMMARY OF PSEUDO-OPS

	Paragraph	Format	Function
.BLOCK	1.4.3.1	label →| .BLOCK →| exp↵	Reserves a block of storage words equal to the expression. If a label is used, it references the first word in the block.
.DEC	1.4.4	→| .DEC↵	Sets prevailing radix to decimal.
.DSA	1.4.6	label →| .DSA ␣ exp↵	Defines a user symbol which is to be used only in the address field.
.END	1.4.1	→| .END ␣ START↵	Must terminate every source program. START is the address of the first instruction to be executed.
.ENDC	1.4.7	→|.ENDC↵	Terminates conditional coding in .IF statements.
.EOT	1.4.2	→|.EOT↵	Must terminate physical program segments, except the last, which is terminated by .END.
.FULL	1.4.10	→|.FULL↵	Produces hardware readin binary tapes.
.IFDEF .IFUND	1.4.7	→|.IFxxx ␣ exp↵	If a condition is satisfied, the source coding following the .IF statement, and terminating with an .ENDC statement, is assembled.
.LIST	1.4.9	→| .LIST↵	Resume printing of assembly listing.
.LOC	1.4.8	→|.LOC ␣ exp↵	Sets the Location Counter to the value of the expression.
.OCT	1.4.4	→|.OCT↵	Sets the prevailing radix to octal. Assumed at start of every program.
.SIXBT	1.4.5	label →| .SIXBT ␣ /text/↵	Input text strings in 6-bit trimmed ASCII, with first character as delimiter.
.SIZE	1.4.11	label →| .SIZE↵	The Assembler outputs the address of last location plus one occupied by the object program.
.XLIST	1.4.9	→| .XLIST↵	Suppress printing of assembly listing.

APPENDIX D
PDP-15/10 HARDWARE READIN BINARY LOADERS

```
/
/PDP-15/1Ø HARDWARE READIN LOADERS
/
/DEFINING %LOW PRODUCES THE LOW SPEED VERSION
/OTHERWISE, THE HIGH SPEED VERSION IS PRODUCED.
/
/LOW SPEED READER VERSION:
/HARDWARE READIN TO 77ØØ (177ØØ IF 8K). WHEN IT HALTS,
/PLACE BINARY PROGRAM TAPE IN LOW SPEED READER
/AND PRESS START, WITH BANK/PAGE MODE SWITCH IN PAGE POSITION.
/
/IIGH SPEED READER VERSION:
/HARDWARE READIN TO 772Ø (1772Ø IF 8K). WHEN IT HALTS,
/PLACE BINARY PROGRAM TAPE IN HIGH SPEED READER
/AND PRESS START, WITH BANK/PAGE MODE SWITCH IN PAGE POSITION.
/
/LOADER HALTS:
/
/AC=777777 - PROGRAM LOADED.
/AC=NONZERO - CHECKSUM ERROR ON LAST BLOCK LOADED.
/       REPOSITION TAPE AT BLANK FRAME PRIOR TO
/       BEGINNING OF LAST BLOCK AND PRESS START
/       TO REREAD.
/       TO IGNORE ERROR, PRESS CONTINUE.
/
CAF=7Ø33Ø2
RSF=7ØØ1Ø1
RSB=7ØØ144
RRB=7ØØ112
KSF=7ØØ3Ø1
KRB=7ØØ312
KRS=7ØØ322
        .FULL
SKPFLG=RSF
RDSLCT=RSB
RDBFR=RRB
        .LOC 1772Ø
        .IFDEF %LOW
SKPFLG=KSF
RDSLCT=KRS
RDBFR=KRB
        .LOC 177ØØ
        .ENDC
        CAF                     /CLEAR FLAGS
LDNXBK  DZM LDCKSM              /CHECKSUMMING LOCATION
        JMS LDREAD              /GET A WORD
        DAC LDSTAD
        SPA                     /BLOCK HEADING-LOADING ADDRESS
        JMP LDXFR               /START BLOCK
        JMS LDREAD
        DAC LDWDCT              /WORD COUNT (2'S COMPLEMENT)
        JMS LDREAD
LDNXWD  JMS LDREAD
        DAC* LDSTAD             /LOAD DATA INTO
        ISZ LDSTAD              /MEMORY
        ISZ LDWDCT              /FINISHED LOADING
```

```
        JMP LDNXWD              /NO
        TAD LDCKSM              /ADD INTO CHECKSUM
        SZA
        HLT                     /CHECKSUM ERROR
        JMP LDNXBK
LDXFR   DAC LDWDCT
        ISZ LDWDCT
        JMP* LDSTAD             /EXECUTE START ADDRESS
        CLC!HLT                 /MANUALLY START USER PROGRAM
LDREAD  Ø
        .IFDEF %LOW
        LAW -3
        DAC LDCTR
        DZM LDTMP
        .ENDC
        TAD LDCKSM
        DAC LDCKSM
LDRDA   RDSLCT
        SKPFLG
        JMP .-1                 /WAIT FOR READER
        RDBFR                   /READ BUFFER
        .IFDEF %LOW
        TAD LDMSK               /BINARY FRAME
        SPA!CLL                 /YES
        JMP LDRDA               /NO
        TAD LDTMP
        ISZ LDCTR
        SKP!RTL
        .ENDC
        JMP* LDREAD
        .IFDEF %LOW
        RTL                     /ACCUMULATE 3 FRAMES
        RTL                     /INTO 1 BINARY WORD
        DAC LDTMP
        JMP LDRDA
LDCTR   Ø                       /PACK COUNTER
LDTMP   Ø                       /BINARY WORD
LDMSK   777600                  /BINARY FRAME MASK
        .ENDC
LDCKSM  Ø                       /CHECKSUM
LDSTAD  Ø                       /LOADING/STARTING ADDRESS
LDWDCT  Ø                       /WORD COUNT
        .END
```

APPENDIX E
SUMMARY OF EDITOR COMMANDS

Transfer from Edit Level to Input Level

>↵	Carriage return as first character.
>I↵	INSERT command with no arguments.
>O(␣↵n)	OVERLAY command.*

Transfer from Input Level to Edit Level

↵	Carriage return as first character.
CTRL P	(echoes ↑P)

Text Input/Output

S(␣ n) ↵	(SIZE) Sets number of lines to occupy buffer.
READ ↵	Reads sequential lines from input device; number of lines specified by SIZE command.
G(␣ n) ↵	(GET) Adds next n lines from subsidiary device to the buffer.
RENEW ↵	Writes contents of buffer and reads new block.
WRITE ↵	Punches contents of buffer on output device and clears buffer.
CLOSE ↵	Writes remainder of input file on output device; must be preceded by a WRITE command.

Pointer Manipulation

T↵	(TOP) Moves pointer to top of buffer.
N(␣ n)↵	(NEXT) Moves pointer past next n lines.
F ␣ string ↵	(FIND) Searches buffer to find next line that *begins with* string.
L ␣ string ↵	(LOCATE) Searches buffer to find next line that *contains* string.
B↵	(BOTTOM) Moves pointer to final line in buffer.

*n lines are deleted from the buffer before the control level is changed.

SEARCH ␣ string↵	Searches input file for next line that begins with string.

Editing Requests

R ␣ line↵	(RETYPE) Replaces the current line with "line."
A ␣ string↵	(APPEND) Adds "string" to current line.
C ␣ qstring1qstring2q↵	(CHANGE) Replaces string1 with string2.
I ␣ line↵	(INSERT) Inserts "line" after current line.
G(␣ n)↵	(GET) Adds n lines from subsidiary input device after current line.
D(␣ n)↵	(DELETE) Deletes next n lines from buffer.
O(␣ n)↵	(OVERLAY) Deletes next n lines from buffer and transfers control to input level.

Examination Requests

P(␣ n)↵	(PRINT) Prints next n lines on Teletype.
V ␣ $\begin{pmatrix}\text{ON}\\ \text{OFF}\end{pmatrix}$ ↵	(VERIFY) Causes text lines to be printed in response to certain commands.
B ␣ $\begin{pmatrix}\text{ON}\\ \text{OFF}\end{pmatrix}$ ↵	(BRIEF) Causes abbreviated printing of VERIFY lines.

APPENDIX F
CREATING ASCII TEXT WITH THE EDITOR
(Using ASR33 Reader/Punch)

This appendix contains procedures intended for the user who is not familiar with the Editor, but wishes to create ASCII text on paper tape immediately. The basic procedure includes loading the paper tape containing the Editor into core memory, completing an initialization sequence, typing the desired text on the Teletype, and punching out the text (which has been held in the Editors buffer). It is assumed that the low-speed (ASR33) paper tape reader and punch are to be used. Appendix D contains detailed examples of basic Editor operation, and Appendix E contains a comprehensive Editor demonstration.

To punch leader:

1. Turn Teletype switch to OFF LINE.

2. Press Teletype punch switch ON.

3. Press HERE IS key several times to generate leader.

4. Press Teletype punch switch OFF.

5. Turn Teletype switch ON LINE.

To load the Editor:

6. Place the tape containing the low-speed, hardware readin binary loader in the Teletype reader.

7. Press Teletype reader switch ON.

8. Set ADDRESS switches to 7720 (17720 for 8K systems).

9. Press I/O RESET and READIN. The computer will halt (AC = 777777).

10. Press reader switch OFF and place binary tape of Editor in Teletype reader.

11. Press reader switch ON and press START. The Editor will be loaded into memory (this takes about 3-1/2 minutes using the ASR33 reader).

To initialize the Editor:

When loading is complete, the Editor types

```
EDIT-15
INTXT*
```

12. You type: L↲
    The Editor types: GETXT*

13. You type: L↵
The Editor types: OUTXT*

14. You type: L↵
The Editor types: EDIT
followed by: >

15. You type a carriage return following the right angle bracket, and the Editor responds

INPUT

To create your ASCII text:

16. Just type it in. Each line should consist of less than 72 characters followed by a carriage return. If you make a mistake, don't worry about it now. Make a note of it and, when you have finished creating your text, you can edit it using basic Editor commands illustrated in Appendix D. The Editor will store your ASCII text in a 20-line buffer (55-line buffer for 8K systems). When you have typed enough lines to fill the buffer or when you have finished creating your text (whichever occurs first), proceed with the following steps.

To punch out the contents of the buffer:

17. Type a carriage return as the first character on a line. The Editor will respond

EDIT
>

18. You type WRITE↵ following the right angle bracket.

19. Press Teletype punch switch ON.

20. Press CONTINUE. The Editor will punch out the buffer. When punching stops

21. Press the Teletype punch switch OFF.

22. Press CONTINUE. The Editor will respond with

>

23. If you wish to continue creating text, return to step 15. If you are finished creating text, return to steps 1 through 5 to generate trailer in the same manner that you generated leader.

APPENDIX G
EXAMPLES OF EDITOR OPERATION

This appendix is intended for the user who is not acquainted with the Editor, but has an ASCII tape that he wishes to modify. Basic editing can be accomplished using only two I/O commands (READ and WRITE) and six editing commands (INSERT, RETYPE, DELETE, PRINT, NEXT, and TOP). Most editing jobs can be accomplished using these simple commands; the following editing session illustrates their use.

(1) To get things started, load the Editor into memory and initialize it by following the procedure in paragraph 2.3.

(2) Make sure the Teletype reader switch is OFF.

(3) Place the tape containing the ASCII text to be edited in the Teletype reader.

(4) Type READ↵

(5) The Editor will halt. Turn the Teletype reader switch ON.

The Editor will read 20_{10} lines (55_{10} for 8K systems) into a block buffer and halt. Turn the Teletype reader switch OFF and press CONTINUE. The Editor will type a right angle bracket (>) in the left margin. This is the signal that the Editor has finished processing your last command (READ) and is ready to accept your next command. You can now perform the following editing operations.

To obtain a listing of the text in the Editor's buffer:

a. Type TOP↵
   This moves the current-line pointer to the beginning of the buffer.
   The Editor types >

b. Type PRINT ␣ 22↵
   This commands the Editor to print 22 lines, starting with the current line. The number 22 includes the 20 lines in the buffer, one pseudo line at the beginning of the buffer, and one extra line to make sure you get everything.
   The Editor types:

```
(Line 1 of original text)
         .
         .
         .
(Line 20 of original text)
END OF BUFFER REACHED BY:
PRINT 22
>
```

To insert a line at the top of the buffer (before line 1 or original text):

a. Type TOP↵
   The Editor types >

*If an out-of-tape condition occurs during the READ (e.g., the ASCII text is less than 20_{10} lines), you can continue by typing CTRL R.

b. Type INSERT ␣ (Line to be added before line 1)

To verify this change:

a. Type TOP↵
The Editor types >

b. Type PRINT ␣ 3↵
The Editor types:

(Line 1 to be added)
(Line 1 of original text)
>

To change line 5 instead of verifying as above:

a. Type NEXT ␣ 4↵
This moves the current-line pointer from line 1 of the original text to line 5 of the original text.
The Editor types >

b. Type PRINT↵
This will allow you to verify that the current-line pointer is in fact pointing at the line you wish to modify.
The Editor types:

(Line 5 of original text)
>

c. Type RETYPE ␣ (New version of line 5)↵
The Editor types >

To add four lines after line 7:

a. Type NEXT ␣ 2
This moves the current-line pointer to line 7.
The Editor types >

b. Type INSERT↵
The Editor types INPUT
You type:

(Line 1 to be inserted)↵
(Line 2 to be inserted)↵
(Line 3 to be inserted)↵
(Line 4 to be inserted)↵

The Editor types:

↵
EDIT↵
>

To verify your insertion:

a. Type TOP↵
The Editor types >

b. Type NEXT ␣ 8↵
(You typed 8 as the parameter for NEXT since you wish to move the pointer down through seven lines of original text plus the line you added at the top.)
The Editor types >

c. Type PRINT ␣ 6↵
The Editor types:

```
(Line 7 of original text)
(Line 1 to be inserted)
          .
          .
          .
(Line 4 to be inserted)
(Line 8 of original text)
>
```

To delete lines 11, 13, and 14:

a. Type NEXT ␣ 3↵
The Editor types >

b. Type PRINT↵
(To verify the position of the current-line pointer.)
The Editor types:

```
(Line 11 of original text)
>
```

c. Type DELETE↵
The Editor types >
(The current-line pointer is now positioned at line 12, the line after the line deleted.)

d. Type NEXT↵
The Editor types >

e. Type DELETE ␣ 2↵
The Editor types >

Now you have added 5 lines to the original 20 lines read, deleted 3 lines, and modified 1 line. You now have 22 lines in your buffer. To verify that this block is as you want it, issue a TOP command followed by a PRINT ␣ 24 command.

To punch out the edited block:

a. Type WRITE↵
The Editor halts. Turn on the Teletype punch and press CONTINUE.
The Editor punches out the block buffer (tabs will not appear in the listing) and halts.

Turn off the Teletype punch and press CONTINUE.
The Editor types >

You now have the option of returning to step 4 to fill the buffer and edit more previously prepared text, or using the INSERT command to fill the buffer. In either case, if the buffer is filled again, a WRITE must precede further READs. If starting a new editing job, type CTRL P to clear all buffers before beginning.

APPENDIX H
EDITOR DEMONSTRATION

This appendix provides a comprehensive example of an actual editing session. Before attempting to follow the example in this appendix, the user should perform the operations outlined in Appendices F and G, and read sections 2.3 and 2.4 to familiarize himself with the Editor. The first page shows the editorial changes that are to be made to an ASCII tape. The next several pages show hardcopy output of the editing session, and the last page shows an assembly listing of the newly edited file. The following comments are keyed to the two pages containing the hard-copy output of the editing session.

1) After initializing the Editor to READ text from the high-speed paper tape reader, to GET subsidiary input from the Teletype reader, and to WRITE edited text on the high-speed punch, the user must READ a block of text before editing can begin.

2) User types a carriage return to ignore CAPACITY WARNING.

3) The user has misspelled a command. It must be retyped in order to be executed. After the command has been executed, PRINT is used to verify the results.

4) The use of command abbreviations saves time.

5) The user should have simply typed–

```
I ␣ ISZ PACK /BUMP TO "TO" ADDRESS
```

instead of changing from edit level to input level, typing the line, followed by another carriage return to get back to edit level.

6) Verification is truncated with BRIEF ON, and prints out only up to the character changed. This saves time on long editing jobs.

7) Verification can be eliminated by the confident user.

8) The end of the buffer has been reached. The user must now issue a WRITE followed by a READ to continue.

9) The editing task is now complete. The user issues a WRITE to finish punching out his edited tape.

Input File Marked For Editing

```
                         LEFT-ADJUSTED   NON-HEADERED
/SUBROUTINE PACK, 7-BIT CHARS TO IOPS ASCII.   ON RETURN, AC HOLDS TOTAL
                                               WORDS OCCUPIED BY PACKED
                                               ARRAY. A WORD OF ALL 1's
                                               MUST TERMINATE THE INPUT
                                               (UNPACKED) ARRAY
/CALL:    JMS     PACK
/         FROM    / START OF INPUT ARRAY.
/         TO      / START OF OUTPUT ARRAY.
PACK      Ø
          LAC *   PACK               /GET FROM ADRESS.  D
ISZ PACK  DAC     PFROM              /GIVE TO FROM POINTER.
          LAC*    PACK               /GET ADDRESS OF TO ARRAY.
          DAC     PTO                /GIVE TO OUTPUT POINTER.
          ~~DAC~~     PLBH               /SAVE AS START ADDRESS.
          ~~ISZ~~     PACK               /BUMP TO RETURN.
PLOOP1    LAW     17773              /SET UP
          DAC     PK5CHR             /5 CHARACTER COUNTER
PLOOP2    LAC*    PFROM              /GET NEXT WORD IN INPUT ARRAY.
     SAD  ~~SAC~~     ~~(-1)~~ ENDCHR          /TERMINATOR?
          ~~SKP~~                        ~~/NO, SKIP.~~
          JMP     PCLOS              /YES, GO CLOSE OUTPUT ARRAY.
          ISZ     PFROM              /POINT TO NEXT WORD.
          ~~ISZ     PFROM              /POINT TO NEXT WORD.~~
          DAC     PWRD3              /SET UP TO ROTATE.
PLOOP7    JMS     PRAL7
          ISZ     PK5CR H            /5 CHARS IN Q ?
          JMP     PPOOP2             /NO, GET ANOTHER.
          LAC     PWRD2              /WORD PAIR COMPLETE.
          RAL!CLL                    /CLEAR PAIR BIT 35.
          DAC     PWRD2
          LAC     PWRD1              /GET FIRST WD OF PAR.  IR
          RAL~~!CLL~~                    /BIT Ø OF WD 2.
          DAC*    PTO                /INSERT FIRST WD IN OUT ARRAY.
          ISZ     PTO                /BUMP OUT ADDRESS.
          JMP     PLOOP1  OO         /GO SET UP NEXT PAIR.
PCLOS     LAW     17773              /MAKE SURE PAIR IS COMPLETE.
DZM PWRD3 SAD     PK5CHR
          JMP     PLOOP7             /INCOMPLETE PAIR.
ENDCHR LAW-1  ~~CLA!CMA~~                 /FORM WORD PAIR COUNT.
TAD* PACK     ~~TAD     PLBH~~            /START ADDRESS.
          CMA
          TAD     PTO                /LESS END ADDRESS.
ISZ PACK  JMP*    PACK               /RETURN TO CALLER.
PFROM Ø
PTO   Ø
PK5CHR Ø
          .END
```

Hard-Copy Output of Editing Session

```
EDIT-15/10

INTXT*
H
GETXT*
L
OUTXT*
H
EDIT
>READ
>FIND /SUBROUT
/SUBROUTINE PACK, 7-BIT CHARS TO IOPS ASCII.
>OVERLAY 1
INPUT
/SUBROUTINE PACK, 7-BIT LEFT-ADJUSTED CHARS TO NON-HEADERED IOPS
/ASCII.  ON RETURN, AC HOLDS TOTAL WORDS OCCUPIED BY PACKED ARRA

CAPACITY WARNING

/A WORD OF ALL 1'S MUST TERMINATE THE INPUT (UNPACKED) ARRAY.

CAPACITY WARNING


EDIT
>LOCATE FROM
 /      FROM
>APPEND         /START OF INPUT ARRAY.
>NEXT
>APEND          /START OF OUTPUT ARRAY.
NOT A REQUEST:
 APEND          /START OF OUTPUT ARRAY.
>APPEND         /START OF OUTPUT ARRAY.
>PRINT 1
 /      TO              /START OF OUTPUT ARRAY.
>L LAC
        LAC     PACK            /GET FROM ADRESS.
>CHANGE LAC/LAC*/
        LAC     PACK            /GET FROM ADRESS.
>CHANGE /LAC/LAC*/
        LAC*    PACK            /GET FROM ADRESS.
>C /ADR/ADDR/
        LAC*    PACK            /GET FROM ADDRESS.
>NEXT 1
>INSERT
CAPACITY WARNING

INPUT
        ISZ PACK              /BUMP TO "TO" ADDRESS.

CAPACITY WARNING
```

Hard-Copy Output of Editing Session (cont)

```
EDIT
>PRINT
        ISZ PACK        /BUMP TO "TO" ADDRESS.
>BRIEF ON
>C ./ ./.
        ISZ PACK
>PRINT
        ISZ PACK        /BUMP TO "TO" ADDRESS.
>L PLBH
        DAC     PLBH
>BRIEF OFF
>PRINT
        DAC     PLBH            /SAVE AS START ADDRESS.
>DELETE 2
>PRINT
 PLOOP1 LAW     17773           /SET UP
>N 1
>A      /5-CHARACTER COUNTER.
>L (
        SAC     (-1             /TERMINATOR?
>VERIFY OFF
>C /SAC/SAD/
>C /8\(-1/ENDCHR/
>V ON
>PRINT
        SAD     ENDCHR          /TERMINATOR?
>N
>D
>N
>P
        ISZ     PFROM           /POINT TO NEXT WORD.
>D
>P
        ISZ     PFROM           /POINT TO NEXT WORD.
>F PL
END OF BUFFER REACHED BY:
F PL
>W
>READ
>F PL
 PLOOP7 JMS     PRAL7
>N
>RETYPE ISZ PK5CHR      5\/5 CHARS IN?
>N
>C ,LP,PL,
        JMP     PLOOP2          /NO, GET ANOTHER.
>N 2
>CHANGE /L/L!CLL
        RAL!CLL                 /CLEAR PAIR BIT 35.
```

```
>N 2
>P
        LAC     PWRD1           /GET FIRST WD OF PAR..
>C /R./IR/
        LAC     PWRD1           /GET FIRST WD OF PAIR.
>L RAL
        RAL!CLL                 /BIT Ø OF WD 2.
>C /!CLL//
        RAL             /BIT Ø OF WD 2.
>L JMP
        JMP     PLØØP1          /GO SET UP NEXT PAIR.
>CHANGE /ØØ/OO/
        JMP     PLOOP1          /GO SET UP NEXT PAIR.
>N
>
INPUT
        DZM PWRD3               /FILL PAIR WITH ZEROES.


EDIT

>L !
        CLA!CLL         /FORM WORD PAIR COUNT
>R ENDCHR       LAW -1  /FORM WORD PAIR COUNT.
>N
>R      TAD* PACK       /START ADDRESS.
>L PTO
        TAD     PTO             /LESS END ADDRESS.
>INSERT ISZ PACL\K
CAPACITY WARNING

>BOTTOM
        .END
>OVERLAY
INPUT
PFROM   Ø

CAPACITY WARNING
PTO

CAPACITY WARNING
PK5CHR  Ø

CAPACITY WARNING
        .END

CAPACITY WARNING

EDIT
>WRITE
```

Edited Output File

```
PLEASE READY THE INPUT DEVICE AND SET THE AC SWITCH

/SUBROUTINE PACK, 7-BIT LEFT-ADJUSTED CHARS TO NON-HEADERED IOPS
/ASCII.  ON RETURN, AC HOLDS TOTAL WORDS OCCUPIED BY PACKED ARRAY.
/A WORD OF ALL 1'S MUST TERMINATE THE INPUT (UNPACKED) ARRAY.
/CALL:  JMS     PACK
/       FROM            /START OF INPUT ARRAY.
/       TO              /START OF OUTPUT ARRAY.
PACK    Ø
        LAC*    PACK            /GET FROM ADDRESS.
        DAC     PFROM           /GIVE TO FROM POINTER.
        ISZ PACK        /BUMP TO "TO" ADDRESS.
        LAC*    PACK            /GET ADDRESS OF TO ARRAY.
        DAC     PTO             /GIVE TO OUTPUT POINTER.
PLOOP1  LAW     17773           /SET UP
        DAC     PK5CHR  /5-CHARACTER COUNTER.
PLOOP2  LAC*    PFROM           /GET NEXT WORD IN INPUT ARRAY.
        SAD     ENDCHR          /TERMINATOR?
        JMP     PCLOS           /YES, GO CLOSE OUTPUT ARRAY.
        ISZ     PFROM           /POINT TO NEXT WORD.
        DAC     PWRD3           /SET UP TO ROTATE.
PLOOP7  JMS     PRAL7
        ISZ PK5CHR      /5 CHARS IN?
        JMP     PLOOP2          /NO, GET ANOTHER.
        LAC     PWRD2           /WORD PAIR COMPLETE.
        RAL!CLL                 /CLEAR PAIR BIT 35.
        DAC     PWRD2
        LAC     PWRD1           /GET FIRST WD OF PAIR.
        RAL                     /BIT Ø OF WD 2.
        DAC*    PTO             /INSERT FIRST WD IN OUT ARRAY.
        ISZ     PTO             /BUMP OUT ADDRESS.
        JMP     PLOOP1          /GO SET UP NEXT PAIR.
PCLOS   LAW     17773           /MAKE SURE PAIR IS COMPLETE.
        DZM PWRD3               /FILL PAIR WITH ZEROES.
        SAD     PK5CHR
        JMP     PLOOP7          /INCOMPLETE PAIR.
ENDCHR  LAW -1          /FORM WORD PAIR COUNT.
        TAD* PACK       /START ADDRESS.
        TAD     PLBH            /START ADDRESS.
        CMA
        TAD     PTO             /LESS END ADDRESS.
        ISZ PACK
        JMP*    PACK            /RETURN TO CALLER.
PFROM   Ø
PTO
PK5CHR  Ø
        .END
```

APPENDIX I
ASCII CHARACTER SET

MODEL 33ASR TELETYPE CODE (ASCII) IN OCTAL FORM

Character	8-Bit Code (in Octal)	Character	8-Bit Code (in Octal)
A	301	!	241*
B	302	"	242*
C	303	#	243*
D	304	$	244
E	305	%	245*
F	306	&	246*
G	307	'	247*
H	310	(	250*
I	311	)	251*
J	312	*	252*
K	313	+	253*
L	314	,	254*
M	315	–	255*
N	316	.	256*
O	317	/	257
P	320	:	272*
Q	321	;	273
R	322	<	274*
S	323	=	275*
T	324	>	276*
U	325	?	277
V	326	@	300*†
W	327	[	333*
X	330	\	334*
Y	331	]	335*
Z	332	↑	336
0	260	←	337
1	261	Leader/Trailer	200**
2	262	Line-Feed (↓)	212†
3	263	Carriage-Return(↵)	215†
4	264	Space (Δ)	240
5	265	Rub-out	377**
6	266	Blank (Null)	000**
7	267	ALT MODE ($)	373
8	270*	Tab	211†
9	271*		

*Not recognized by ODT.

**Ignored by ODT.

†Have special meaning to TIC & TOC.

APPENDIX J
ODT COMMAND SUMMARY

Register Examination and Modification

k/	Open register k.
/	Open last referenced register.
↵	Close register.
k↵	Store k and close register.
↓	Close register and open next register.
↑	Close register and open previous register.
←	Close register and interpret contents as address.
$A	Open AC and Link registers.
$J	Display PI and API status.

Execute Instruction

k$X	Execute instruction k.

Setting User Start Address

$Z	Open user start address.
$G	Start program at address specified in $Z register.
k$G	Start program at location k.

Breakpoint Commands

k$B	Set breakpoint 1 at location k.
k$nB	Set breakpoint n at location k.
$B	Remove all breakpoints.
$nB	Remove breakpoint n.
$C	Execute instruction at breakpoint and continue with user program.
k$C	Execute instruction at breakpoint and continue with user program when the breakpoint has been encountered for the kth time.
$K	Disable PI and API during all breakpoints.
$U	Enable PI and API during all breakpoints.
$V	Vary autoindex (must be between 10 and 17).
$Y	Open re-entrant PC list (after re-entrancy error).

Searching Operations

$M	Open mask register for modification.
k$W	Search for bit configuration k.

Initialize Buffers

$I	Initialize (set) block of memory locations to zero.*
k$I	Initialize (set) block of memory location to k.*

Paper Tape Output

$H	Select high-speed punch.
$L	Select low-speed punch.
$k_1 ; k_2 \ \$D$	Dump (punch) all locations between k_1 and k_2, inclusive.
$T	Punch terminal block; loader will halt when loading complete.
k$T	Punch terminal block; loader will transfer to location k when loading complete.

*User must first specify high and low limits of block by means of $M command.

**If low-speed punch ($L), ODT halts to allow user to turn-on punch.

PART III
Basic I/O MONITOR

The PDP-15 Monitor Environment

User Program Commands (System Macros)

Basic Input Output Monitor Function

I/O Device Handlers

APPENDICES

PART III
BASIC I/O MONITOR

CHAPTER I
THE PDP-15 MONITOR ENVIRONMENT

1.1 MONITOR FUNCTIONS

The Basic I/O Monitor greatly simplifies the task of programming I/O functions by providing an interface between system or user programs and the external world of I/O devices. Upward compatibility exists between the Basic I/O Monitor and the other monitors of the PDP-15 Software System; programs written to operate under control of the Basic I/O Monitor will also operate, without modification, under control of the Advanced and Background/Foreground Monitors. The Monitors, by means of the Input/Output Programming System (IOPS) and Program Interrupt (PI) or Automatic Priority Interrupt (API), allow simultaneous operation of multiple I/O devices along with overlapping computations.

Certain features such as the general Monitor environment, data handling, and logical/physical I/O device associations, are common to all three Monitors. Detailed information on the Advanced and Background/Foreground Monitors will be found in manuals DEC-15-MR2A-D and DEC-15-MR3A-D respectively.

1.1.2 General I/O Communication

A general communication required to accomplish an I/O task is the same for all three Monitor systems (see Figure 1-1). A system or user program initiates an I/O function by means of a Monitor command (system macro), which is interpreted by a CAL handler within the Monitor as a legitimate I/O call. (See the PDP-15 Reference Manual DEC-15-BRZA-D for a description of the CAL instruction.) The I/O call includes a logical I/O device number as one of its arguments. The Monitor establishes the logical/physical I/O device association by means of its Device Assignment Table (.DAT). When this has been accomplished, the Monitor passes control to the appropriate device handler subroutine to initiate the I/O function and returns control to the system or user program. The system or user program retains control until an interrupt (PI or API) occurs, at which time it relinquishes control to the device handler to perform and/or complete the specified I/O function. Computations or other processing can be performed by the system or user program while waiting for an interrupt. This feature allows the programmer to make optimum use of available time.

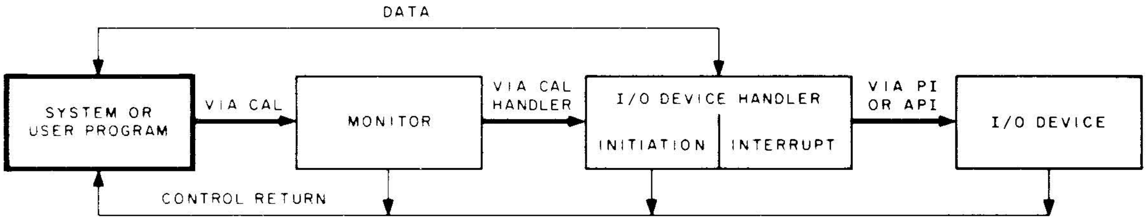


Figure 1-1. General I/O Communication in Monitor Environment

1.1.3 Command, Control, and Data Flow

Figure 1-2 provides a more detailed representation of the Monitor environment, with emphasis on command, control, and data flow. As shown, the user can initiate a command via the Teletype which is interpreted by a Command Processor within the system program (or user program if so designed).

Each system or user program must internally set up line buffers (except when using Dump mode, discussed later) to be used in transmitting data to or from the external environment. Each line buffer of n words consists of a two-word header (referred to as a header word pair) and n -2 words of data. The system or user program can exercise control on output by modifying the header word pair, or it can verify on input by examining the header word pair. The use of line buffers is discussed in more detail later in this chapter.

Monitor I/O commands (system macros) are written as part of the system or user program. In FORTRAN IV source programs, these commands are in the form of READ and WRITE statements (refer to the PDP-15 FORTRAN IV Manual, DEC-15-KFZA-D). These statements are translated by the compiler into the proper calling sequences for the FORTRAN Object Time System which provides the required Monitor calls at execution time. In MACRO-15 source programs, Monitor I/O commands are written as system macros within the system or user program. These system macros are expanded at assembly time and include a CAL initiated Monitor call that contains the logical device number as one of the arguments.

At execution time, Monitor calls are processed by the CAL Handler within the Monitor. Non-I/O functions are then further processed by the Monitor Control routine and I/O functions are processed by the I/O Control routine (see Figures 1-2 and 1-3). A complete description of each of these commands is given in Chapter 2. If the original command involved is an I/O function, the I/O control routine checks the Device Assignment Table to associate the logical I/O device (specified by the system macro) to a physical I/O device. The logical/physical device associations can be modfied either by reassembly or with the aid of utility program PUNCH 15.

When the logical/physical I/O device association has been established, the Monitor passes control to the appropriate I/O device handler which initializes itself, initiates I/O, and returns control to the system or user program. As mentioned previously, the system or user program retains control until the specified device causes an interrupt (PI or API). At this point, it relinquishes control to the device handler to continue or complete the specified I/O operation.

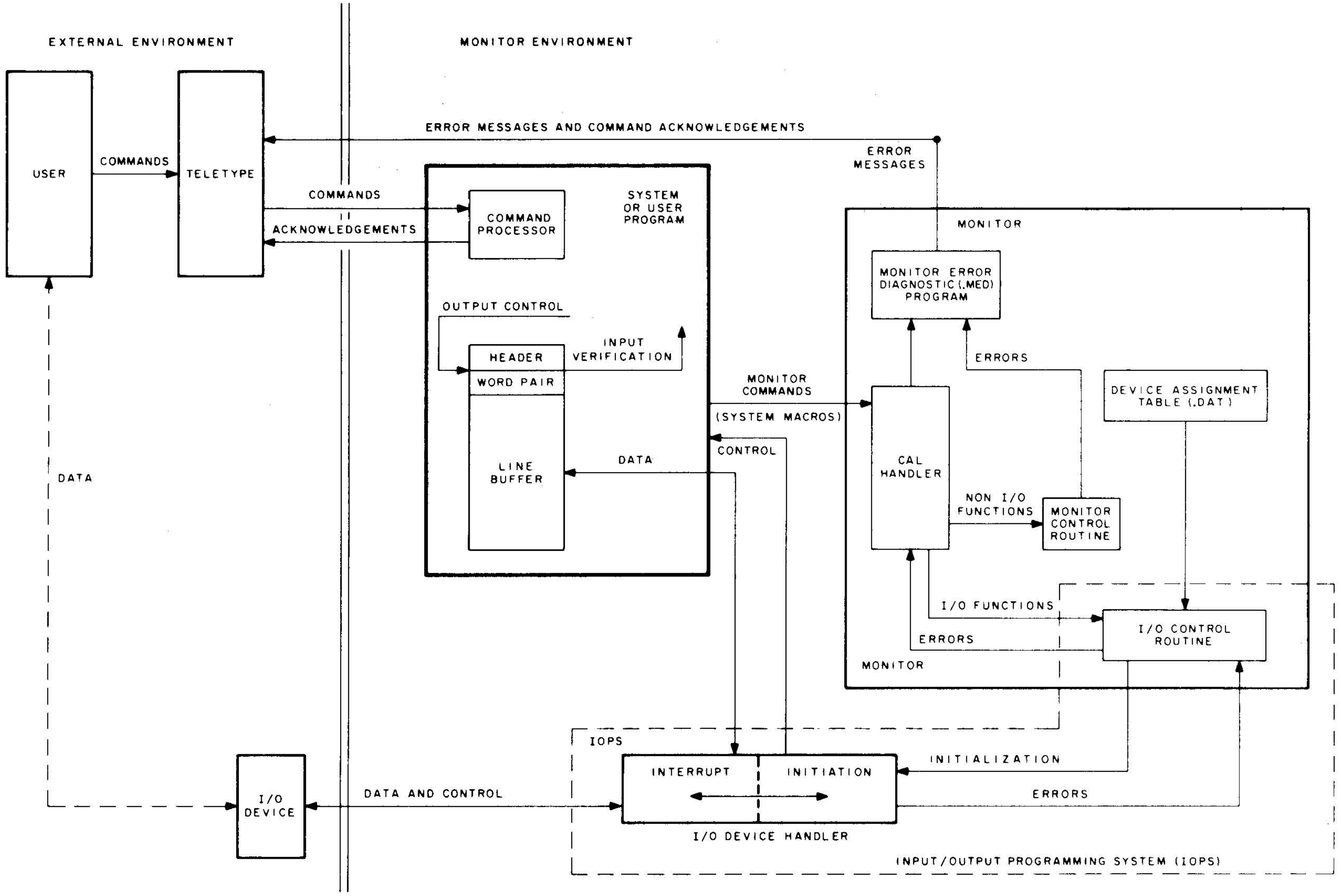


Figure 1-2. Command, Control, and Data Flow in the Basic I/O Monitor Environment

	Function Code	Command
Functions processed by I/O control routine	1	.INIT
	2	.DLETE, .RENAM, and .FSTAT
	3	.SEEK
	4	.ENTER
	5	.CLEAR
	6	.CLOSE
	7	.MTAPE
	10	.READ and .REALR
	11	.WRITE and .REALW
	12	.WAIT and .WAITR
	13	.TRAN
Functions processed by monitor control routine	14	.TIMER
	15	.EXIT
	16	.SETUP
	17	.IDLE and .IDLEC
	20	.RLXIT

Figure 1-3. Monitor Commands and Function Codes

NOTE

.INIT, .READ, .WRITE, .WAIT, .WAITR, .CLOSE, .TIMER, and .EXIT are recognized by all three Monitors.

.SEEK, .ENTER, .FSTAT, .RENAM, .DLETE, .TRAN, .CLEAR, and .MTAPE are recognized by the Advanced and Background/Foreground Monitors (and are ignored by the Basic I/O Monitor).

.SETUP is used by the Monitors in setting up the I/O skip chain and API channel registers (see Chapter 4).

.IDLE, .IDLEC and .RLXIT are recognized by the Background/Foreground Monitor only.

In either case, control is returned to the system or user program at the point where it was interrupted. The system or user program, by means of a .WAIT system macro (described in Chapter 2), can determine whether an input or output operation has been completed. If the transfer of data from or to the system or user program line buffer has been completed, program execution continues; if the transfer has not been completed, control is returned to the .WAIT macro.

Additional buffering is provided by the individual device handlers as required. All device handlers are non-resident in the sense that only those handlers required by the system or user program are loaded into core.

1.2 LINE BUFFERS

As mentioned in the preceding general description of the Monitor environment, each system or user program must internally set up line buffers to be used in transmitting data to or from the external environment. An exception to this rule is when data is transmitted in the Dump mode (described in paragraph 1.3.1.4). Each line buffer of n words (always even) should be set up to consist of a two-word header (termed a header word pair) followed by n-2 words of data as shown in Figure 1-4.

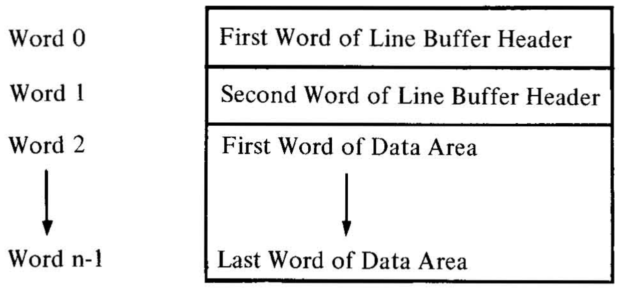


Figure 1-4. Line Buffer Structure

A system or user program should contain at least one line buffer for each device that is to be used. This buffer is used to set up output lines before transmittal to an output device, or to receive input lines from the associated input device. The Monitor accepts commands (system macros) from system or user programs to initiate input to the line buffers and to write out the contents of line buffers. Complete descriptions of these commands are given in Chapter 2. Line buffers are internal to, and must be defined by, each system or user program. The header word pair within a line buffer is detailed in Figure 1-5. The .BLOCK pseudo operation may be used to reserve space for a line buffer. A tag is required to allow referencing by individual .READ and .WRITE macros. For example:

```
            .DEC

LINEIN      .BLOCK 52       /creates 52-word line
                            /buffer named LINEIN.

LINOUT      .BLOCK 52       /creates 52-word line
                            /buffer named LINOUT.
```

Before output, the user must set the appropriate word pair count in bits 1 through 8 of word zero in the line buffer if they have not already been set by a device handler on input. This count overrides the word count passed to IOPS by

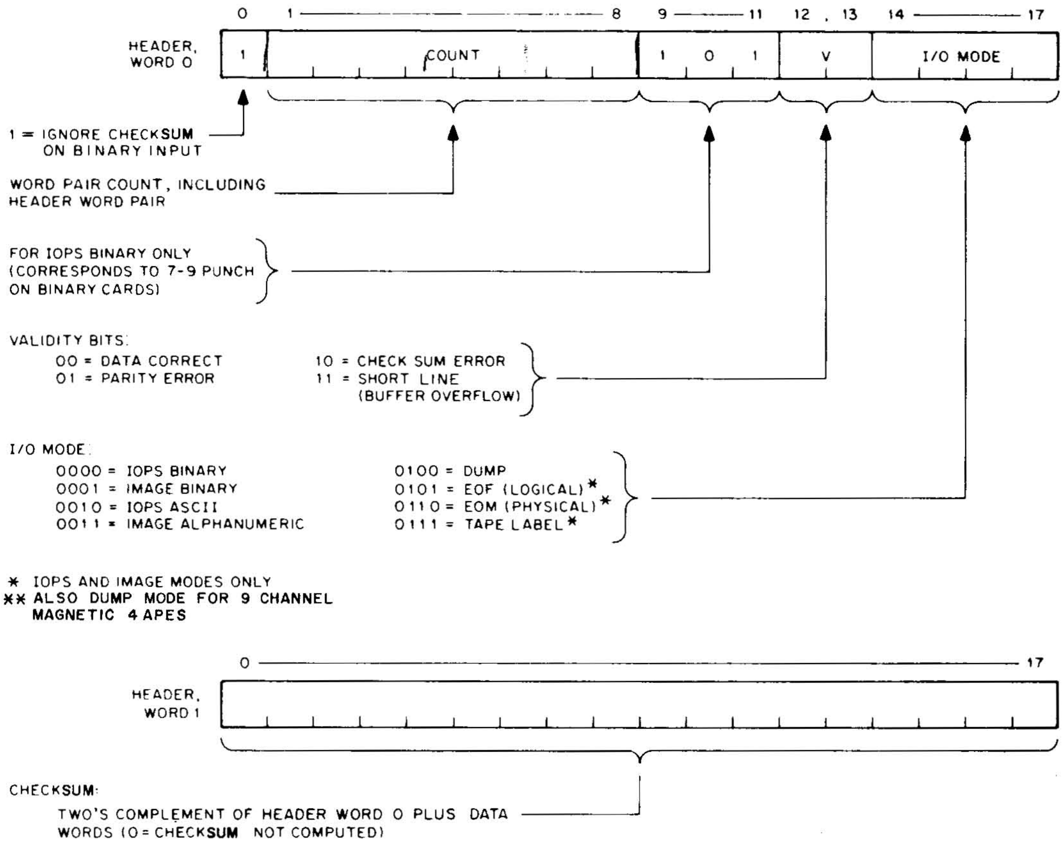

0
1
8
9
11
12
13
14
17
HEADER, WORD 0
1
COUNT
1 0 1
V
I/O MODE
1 = IGNORE CHECKSUM ON BINARY INPUT
WORD PAIR COUNT, INCLUDING HEADER WORD PAIR
FOR IOPS BINARY ONLY (CORRESPONDS TO 7-9 PUNCH ON BINARY CARDS)
VALIDITY BITS:
00 = DATA CORRECT
01 = PARITY ERROR
10 = CHECK SUM ERROR
11 = SHORT LINE (BUFFER OVERFLOW)
I/O MODE:
0000 = IOPS BINARY
0001 = IMAGE BINARY
0010 = IOPS ASCII
0011 = IMAGE ALPHANUMERIC
0100 = DUMP
0101 = EOF (LOGICAL)*
0110 = EOM (PHYSICAL)*
0111 = TAPE LABEL*
* IOPS AND IMAGE MODES ONLY
** ALSO DUMP MODE FOR 9 CHANNEL MAGNETIC 4APES
0
17
HEADER, WORD 1
CHECKSUM:
TWO'S COMPLEMENT OF HEADER WORD 0 PLUS DATA WORDS (0 = CHECKSUM NOT COMPUTED)

Figure 1-5. Format of Header Word Pair

the .WRITE macro. (The word count must still be specified in the .WRITE macro for each data mode; however, it only has meaning in Dump mode since there is no header word pair.) In IOPS binary mode (discussed in Paragraph 1.3.1.2), bits 9 through 11 should be set to 101 if the output will ultimately be on cards. The checksum word, the second word in the header, need not be set by the user since checksums are computed by IOPS.

Before input, the user should not be concerned with the header word pair since they will be set by IOPS to enable the user to determine what has happened after input has terminated.

On input, the word count specified in the .READ macro is used by IOPS to determine the maximum number of locations to be occupied by the data being read. If the word count is exceeded before input is terminated, or if there is a parity or checksum error, IOPS sets the appropriate validity bits in header word 0 to indicate the error.

After input, the user should check the validity bits in word 0 of the line buffer header to determine if the data was read without error. If multiple errors are detected, priority is given to a parity error over a checksum error. IOPS ignores checksum errors on binary input if bit 0 of word 0 of the line buffer header is set to 1. IOPS sets the I/O mode bits (bits 14 through 17 of word 0 of the line buffer header) to: 6 (0110_2) if it senses a physical end-of-medium (such as end-of-tape in the paper-tape reader), or 5 (0101_2) if it senses a logical end-of-file during an IOPS binary read.

When choosing a word count (that is, the maximum line buffer size) to specify in system macros, both the set of possible devices and the mode of data transmission must be considered. The maximum line buffer sizes (including 2-word header) for standard peripheral devices, along with applicable data modes, are listed in Table 1-1.

Table 1-1. Maximum Line Buffer Sizes

Device	Maximum Line Buffer Size	Data Modes*	Notes
PR (paper tape reader)	52_{10}	All	34_{10} sufficient only if mode 2. Headers accepted for mode 0; generated for modes 1, 2 & 3.
PP (paper tape punch)	52_{10}	All	34_{10} sufficient only if mode 2. Headers output only for mode 0.
TT (Teletype)	34_{10}	2 & 3 only	Allows for 80_{10} characters. Headers generated on input. Headers not output on output.
CD (card reader)	36_{10}	2 only	Headers generated on input.
LP (line printer)	52_{10}	2 only	Allows for 125_{10} characters. No headers output.
MT0-7 (magnetic tape)	255_{10}	0, 2 & 5	Modes 0 and 2 allow for several line buffers (logical records) per physical block.

*See paragraph 1.3 below.

1.3 DATA MODES

The Input/Output Programming System allows data transmission to or from a system or user program in five different modes.

Mode	*Code**
IOPS Binary	0
Image Binary	1
IOPS ASCII	2
Image Alphanumeric	3
Dump	4
9-Channel Dump	5

*Bits 14 through 17 of Header Word 0, specified by system macro and set by IOPS.

1.3.1 **IOPS Modes**

The two IOPS data modes include IOPS ASCII and IOPS binary as shown in Figure 1-6 on paper tape, and described in the following paragraphs.

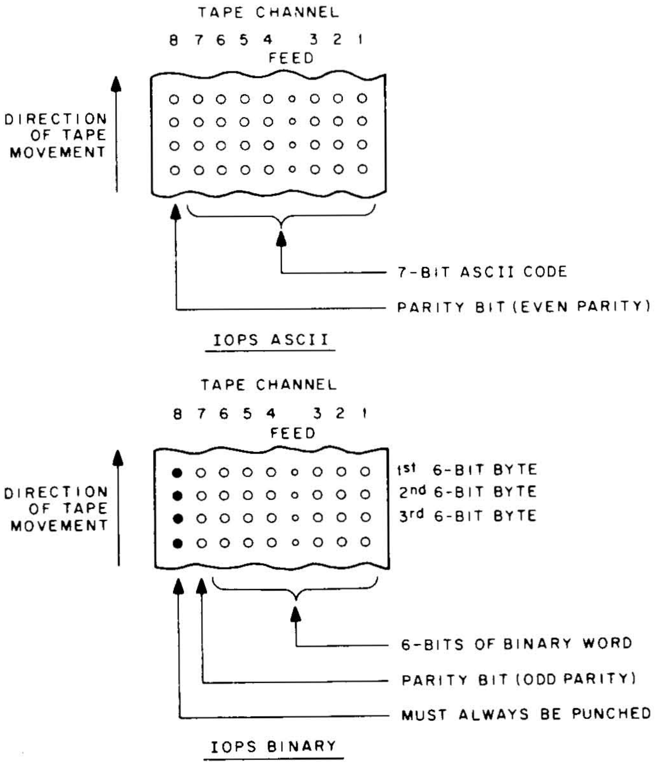


Figure 1-6. IOPS Mode Data on Paper Tape

1.3.1.1 **IOPS ASCII** – 7-bit ASCII is used by IOPS to accommodate the entire 128-character revised ASCII set *(Appendix A)*. All alphanumeric data, whatever its original form on input (ASCII, Hollerith, etc.) or final form on output, is converted internally and stored as 5/7 ASCII. "5/7 ASCII" refers to the internal packing and storage scheme. Five 7-bit ASCII characters are packed in two contiguous locations as shown in Figure 1-7 and can be stored as binary data on any bulk storage device. Input requests involving IOPS ASCII should be made with an even word count to accommodate the paired input.

ASCII data is input to or output from IOPS ordinarily, via the Teletype or paper tape, although it may exist in 5/7 ASCII form on any mass storage device. IOPS ASCII is defined as a 7-bit ASCII character with even parity in the eighth (high order) bit, in keeping with USA standards. IOPS performs a parity check on input of IOPS ASCII data prior to the 5/7 packing. On output, IOPS generates the correct parity.

Non-parity IOPS ASCII occurs in data originating at a Model 33, 35, or 37 teletype, without the parity option. This data always appears with the eighth (high order) bit set to 1. Apart from parity checking, the IOPS routines handle IOPS ASCII and non-parity IOPS ASCII data identically.

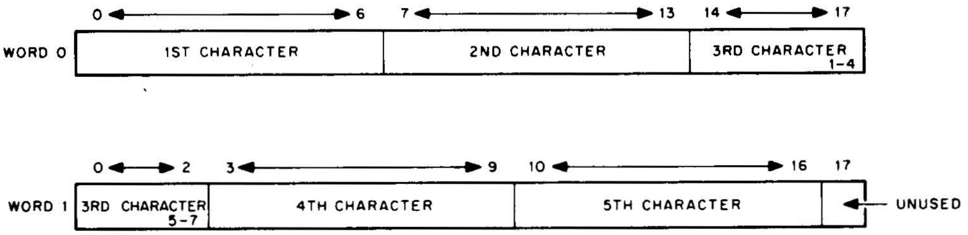


Figure 1-7. 5/7 ASCII Packing Scheme

An alphanumeric line consists of an initial form control character (line feed, vertical tab, or form feed), the body of the line, and a carriage return (CR) or ALT MODE. CR (or ALT MODE) is a required line terminator in IOPS ASCII mode. Control character scanning is performed by some device handlers for editing or control purposes (see paragraph 4.4 for effects of control characters on specific devices).

1.3.1.2 **IOPS Binary** – IOPS Binary data is blocked in an even number of words, with each block preceded by a two-word header. On paper tape (see Figure 1-6), IOPS binary uses six bits per frame, with the eighth channel always set to 1, and the seventh channel containing the parity bit (odd parity) for channels 1 through 6 and channel 8. The parity feature supplements the checksumming as a data validity provision in paper tape IOPS binary.

1.3.1.3 **Image Modes** – Image Mode data is read, written, and stored in the binary or alphanumeric form of the source or terminal device, one character per word, as shown in Figures 1-8 and 1-9. No conversion, checking, or packing is permitted, and character scanning is generally omitted.

1.3.1.4 **Dump Mode** – Dump mode data is always binary. Dump mode is used to output from or load directly into any core memory area, bypassing the use of line buffers. Each dump mode statement has arguments defining the core memory area to be dumped. Dump mode is normally used with bulk storage devices, although it is also possible to use it with paper tape output and input.

1.3.1.5 **9-Channel Dump Mode** – This mode is always binary and is exclusively used with 9 channel magnetic tape transports. This mode is designed to take advantage of all 8 data bits in each frame of the tape and thus insure maximum thruput for each word transferred. A full discussion of this data mode is provided in paragraph 4.4.6.

1.3.2 Input/Output Data Mode Terminators

Input/output terminators for each of the data modes are summarized in Table 1-2.

1.4 SYSTEM TABLES

System tables used by each of the monitor systems include the Device Assignment Table (.DAT), and the System Communication Table (.SCOM). These tables are discussed in the following paragraphs.

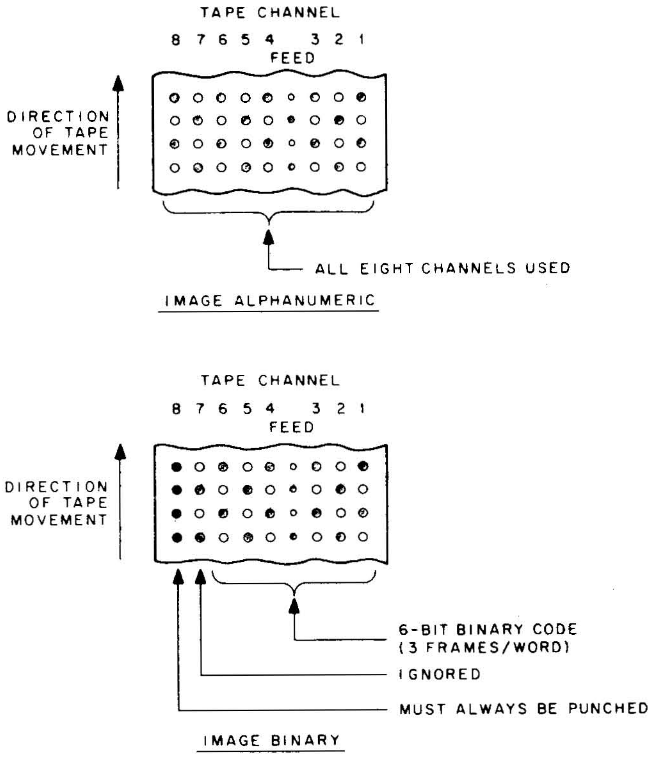


Figure 1-8. Image Mode Data on Paper Tape

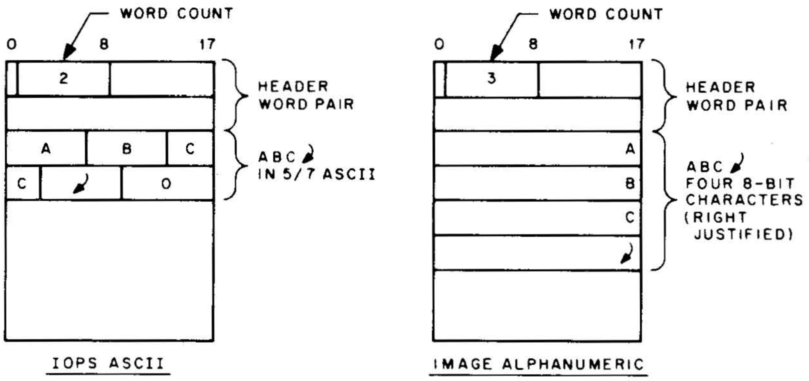


Figure 1-9. IOPS ASCII and Image Alphanumeric Data in Line Buffers

1.4.1 Device Assignment Table (.DAT)

Both FORTRAN IV and MACRO-15 coded user programs, as well as the system programs, specify I/O operations with commands to *logical* I/O devices. One of the Monitor's functions is to relate these logical units to *physical*

Table 1-2. Input/Output Data Mode Terminators

	IOPS ASCII	IOPS Binary	Image Alphanumeric	Image Binary	Dump	9-Channel Dump
INPUT	Carriage Return	Word Pair Count	Word Count	Word Count	Word Count	
	ALT MODE	End of Medium	End of Medium	End of Medium	End of Medium	
	Word Pair Count**	Word Count*	End of File**	End of File**	End of File**	Word Count
	End of Medium	End of File**				
	Word Count*					
	End of File**					
OUTPUT	Carriage Return					
	ALT MODE	Word Pair Count	Word Pair Count	Word Pair Count	Word Count	Word Count
	Word Pair Count*** (except on teletype)					

*A short Line Indicator will be placed by IOPS into the validity bits (12 and 13) of Header Word 0 if the maximum size of the line buffer is reached before an End-of-File.

**Bulk storage only.

***If the word pair count is not greater than 1, the output line is ignored. If the word pair count is greater than 1, it has no effect and a carriage return or ALT MODE are the only legal line terminators.

devices. To do this, the Monitor contains a Device Assignment Table (.DAT) which has "slot" numbers that correspond directly to logical I/O device numbers. Each .DAT slot *contains* the physical device unit number (if applicable) along with a pointer to the appropriate device handler.

All I/O communication in the Monitor environment is accomplished by the logical/physical device associations provided by the Device Assignment Table (see paragraph 3.3.2).

1.4.2 System Communication Table (.SCOM)

The System Communication Table (.SCOM) provides a list of registers that can be referenced by the Monitor, IOPS, and system programs. A complete list of .SCOM entries, and the purpose of each, is given in Table 1-3. The System Communication Table begins at location 100_8.

1.5 SPECIFYING DEVICES USED TO LINKING LOADER

When writing a MACRO-15 program that uses Monitor commands (system macros), it is necessary to use the .IODEV pseudo operation somewhere in the program to specify to the Linking Loader which .DAT slots are to be used. The .IODEV pseudo-op causes a code to be generated that is recognized by the Linking Loader and used to load device handlers associated with specified .DAT slots. FORTRAN IV programs cause the compiler to generate this code based

Table 1-3. System Communication Table (.SCOM) Entries

Word	Purpose
.SCOM	First free register below resident portion of System Bootstrap (constant)
.SCOM + 1	First free register above resident Monitor (constant)
.SCOM + 2	First free register
.SCOM + 3	Last free register
.SCOM + 4	Hardware options available: Bit 0 1 = API Bit 1 1 = EAE Bit 2 1 = TTY is 35/37 Bit 3 1 = Non-resident Monitor in core Bit 6 1 = 9-channel, 0 = 7-channel Magnetic tape Bit 7 1 = Page Mode, 0 = Bank Mode
.SCOM + 5	System program starting location
.SCOM + 6	User starting location (bits 3 through 17), and: Bit 0 1 = DDT Load Bit 1 1 = G Load Bit 2 1 = No-symbol-table Load
.SCOM + 7-11_8	Device numbers of Linking Loader's devices. These are used to avoid loading user handlers already in core for the Loader itself.
.SCOM + 12-15_8	Transfer vectors associated with API software level channel registers 40 through 43_8.
.SCOM + 16	Contains PC on keyboard interrupts.
.SCOM + 17	Contains AC on keyboard interrupts.
.SCOM + 20	Bit 0 1 = 12K, 20K or 28K System Bit 3-17 = First Free Register in extra 4K Page

on the units specified in READ and WRITE statements. If a variable is used in a FORTRAN program to specify an I/O unit, handlers will be loaded for all positive .DAT slots that have handlers assigned. The .IODEV pseudo-op has the following form:

```
.IODEV 3, 5, 6
```

where the MACRO-15 program containing this statement can use .DAT slots 3, 5, and 6. An error message is generated if a slot called for by a program is unassigned. The MACRO-15 Assembler Manual (DEC-15-AMZA-D) provides a full discussion of pseudo-ops.

CHAPTER 2
USER PROGRAM COMMANDS (SYSTEM MACROS)

User program commands or system macros that apply to the Basic I/O Monitor are described in this chapter for convenient reference. Because of the upward compatability of Monitor systems, all I/O Monitor commands (system macros) are also used in the Advanced and Background/Foreground Monitor environments.

NOTE

When executing a system macro, the monitor makes no attempt to save the user's accumulator and link bit.

2.1 BASIC I/O MONITOR COMMANDS (SYSTEM MACROS)

The following commands are available for use in programs that are to operate in the Basic I/O Monitor environment. Each command is described in detail in the paragraphs that follow.

Name	Purpose
.INIT	Initializes the device and device handler.
.READ	Transfers data from the device to the line buffer.
.WRITE	Transfers data from the line buffer to the device.
.WAIT	Checks availability of the user's line buffer and waits if busy.
.WAITR	Checks availability of the user's line buffer, and provides transfer address for busy return.
.CLOSE	Terminates use of a file.

Name	Purpose
.TIMER	Calls and uses real-time clock.
.EXIT	Returns control to the Monitor.

2.2 INIT (INITIALIZE)

FORM: .INIT a, F, R

VARIABLES:

a = Device Assignment Table (.DAT) slot number (in octal radix)

F = File Type: $\begin{cases} 0 = \text{Input File} \\ 1 = \text{Output File} \end{cases}$

R = User Restart Address* (should be in every .INIT statement)

EXPANSION:

LOC	CAL + F_{7-8} + a_{9-17}	
LOC + 1	1	/The CAL handler will place the unit number (if /applicable) associated with .DAT slot *a* into bits /0 through 2 of this word.**
LOC + 2	R	
LOC + 3	n	/Maximum size of line buffer associated with .DAT /slot *a* for example, 255_{10} for DECtape.***

DESCRIPTION: The macro .INIT causes the device and device handler associated with .DAT slot *a* to be initialized. .INIT must be given prior to any I/O commands referencing .DAT slot *a*; a separate .INIT command must be given for each .DAT slot referenced by the program. Each initialized .DAT slot constitutes an open file to the device handler and must be .CLOSEd. Since a .DAT slot may refer to only one type of file (input or output), only one file type specification (0 or 1) may be made in an .INIT statement. If a .DAT slot first references an input file, then an output file (or vice versa), a second .INIT command must be executed to change the transfer direction prior to the actual data transfer command.

2.3 READ

FORM: .READ a, M, L, W

VARIABLES:

a = .DAT slot number (octal radix)

M = Data mode $\begin{cases} 0 = \text{IOPS Binary} \\ 1 = \text{Image Binary} \\ 2 = \text{IOPS ASCII} \\ 3 = \text{Image Alphanumeric} \\ 4 = \text{Dump Mode} \end{cases}$

L = Line Buffer address

W = Line buffer word count (decimal radix), including the two-word header

*Has meaning only for .INIT commands referencing slots used by Teletype (the last .INIT command encountered for any slot referencing the keyboard or teleprinter takes precedence). When the user types CTRL P, control is transferred to R. For example, the Linking Loader takes advantage of this feature to restart the system when a new medium has been placed in the input device.

**Has no direct effect upon the user's program, but should be noted so that no attempt will be made to use LOC + 1 as a constant.

***Size is returned by the handler so that the program, in a device-independent environment, can use it to properly set up line buffers.

EXPANSION:	LOC	CAL + M_{6-8} + a_{9-17}	
	LOC + 1	10	/CAL Handler will place unit number (if applicable) /into bits 0 through 2.
	LOC + 2	L	
		.DEC	/Decimal radix
	LOC + 3	-W	

DESCRIPTION: The .READ command is used to transfer the next line of data from the device assigned to .DAT slot *a* to the line buffer in the user's program. In the operation, M defines the mode of the data to be transferred; L is the address of the line buffer; and W is the number of words in the line buffer (including the two-word header).

Since I/O operations and internal data transfers may proceed asynchronously with computation, a .WAIT command must be used after a .READ command before the user attempts to use the data in the line buffer or to read another line into it.

When a .READ (non-dump mode) has been completed, the program should interrogate bits 12 through 13 of the first word of the line buffer header to ascertain that the line was read without error. Bits 14 through 17 should be checked for end-of-medium and end-of-file conditions.

2.4 .WRITE

FORM: WRITE a, M, L, W

VARIABLES: a = .DAT slot number (octal radix)

M = Data mode
- 0 = IOPS Binary
- 1 = Image Binary
- 2 = IOPS ASCII
- 3 = Image Alphanumeric
- 4 = Dump Mode

L = Line buffer address

W = Line buffer word count (decimal radix), including the two-word header

EXPANSION:	LOC	CAL + M_{6-8} + a_{9-17}	
	LOC + 1	10	/CAL Handler will place the unit number (if appli- /cable) associated with .DAT slot *a* into bits /0 through 2.
	LOC + 2	L	
		DEC	/Decimal radix
	LOC + 3	-W	

DESCRIPTION: .WRITE is used to transfer a line of data from the user's line buffer to the device associated with DAT slot *a*.

WAIT must be used after a .WRITE command, before the line buffer is used again, to insure that the transfer to the device has been completed.

On non-bulk storage devices, headers are output along with the data in IOPS binary mode only (bit 9 and 11 of header word 0 should be set to 1). On bulk storage devices, headers are output along with the data in all modes except dump mode. In image modes, the header space cannot be used for data, even though the headers are not written out. The word pair count in the header takes precedence over maximum size (or word count) in all modes and must be inserted by the user.

For both .READ and .WRITE macros, dump mode causes the transfer of the specified core area to or from one record on magnetic or paper tape. One or more blocks on DECtape or disk may be occupied by a single dump command. A subsequent .WRITE in dump mode will utilize the unfilled portion of the last block.

2.5 .WAIT

FORM: WAIT a

VARIABLES: a = .DAT slot number (octal radix)

EXPANSION:

	LOC	CAL + a_{9-17}	
	LOC + 1	12	/The CAL Handler will place the unit number (if /applicable) associated with .DAT slot *a* into bits /0 through 2.

DESCRIPTION: .WAIT is used to detect the availability of the user's line buffer (being filled by .READ or emptied by .WRITE). If the line buffer is available, control is returned to the user immediately after the .WAIT macro expansion (LOC + 2). If the transfer of data has not been completed, control is returned to the .WAIT macro. .WAIT must also be used after the .TRAN command.

2.6 .WAITR

FORM: WAITR a, ADDR

VARIABLES: a = .DAT slot number (octal radix)

ADDR = Address to which control is passed if line buffer is not available for use.

EXPANSION:

	LOC	CAL + 1000_8 + a_{9-17}	
	LOC + 1	12	/The CAL Handler will place the unit number (if /applicable) associated with .DAT slot *a* into bits /0 through 2.
	LOC + 2	ADDR	

DESCRIPTION: .WAITR is also used to detect the availability of the user's line buffer. If the buffer is available, control is returned to the user immediately after the .WAITR macro expansion (LOC + 3). If the transfer of the data has not been completed, however, control is given to the instruction at ADDR. It is the user's responsibility to return to the .WAITR to again check the availability of the buffer.

2.7 .CLOSE

FORM:	.CLOSE a		
VARIABLES:	a = .DAT slot number (octal radix)		
EXPANSION:	LOC	CAL + a_{9-17}	
	LOC + 1	6	/The CAL Handler will place the unit (if applicable) /associated with .DAT slot *a* into bits 0 through 2.

DESCRIPTION: When action has been initiated (.INIT or .SEEK or .ENTER) on a file (whether the device is file-oriented or not) this action must be terminated by a .CLOSE command.

On input, it is assumed that the user is finished with the file when the .CLOSE macro is used, so the file is closed. On output, all associated output is allowed to finish and then an EOF (end-of-file) line is output before the file is finally closed. If *a* refers to a file-oriented device, any earlier file of the same name and extension, as currently referenced, is deleted from its directory after the new file is written.

2.8 .TIMER

FORM:	TIMER	n,C	
VARIABLES:	n = Number of clock increments (decimal radix)		
	C = Address of subroutine to handle interrupt at end of interval		
EXPANSION:	LOC	CAL	
	LOC + 1	14	
	LOC + 2	C	
		.DEC	/Decimal radix
	LOC + 3	-n	

DESCRIPTION: .TIMER is used to set the real-time clock to n increments and to start it. Each clock increment represents 1/60s for 60 Hz systems and 1/50s for 50 Hz systems.

C + 1 is the location to which control is given when the Monitor services the clock interrupt. The coding at C should be in subroutine form; for example,

```
C           0               /C + 1 is reached via JMS
            DAC SAVEAC
            . ⎫ Must not contain any Monitor CALs
            . ⎬
            . ⎭ in Basic or Advanced Software Systems
            LAC C           /Restore Link
            RAL
            LAC SAVEAC      /Restore AC
XIT         JMP* C
```

so that control will return to the originally-interrupted sequence when the interval-handling routine has been completed. The Monitor automatically reenables the interrupt system before transferring control to C + 1. If the user wishes to initiate another interval at the completion of the previous interval in the subroutine specified to .TIMER, he may do so as follows:

```
            LAC  (desired interval in 2's complement)
            DAC* (7
            LAC C           /Restore Link
            RAL
            LAC SAVEAC      /Restore AC
            CLON            /Turn on clock
            JMP* C
```

2.9 .EXIT

```
FORM:           .EXIT

EXPANSION:      LOC         CAL
                LOC + 1     15
```

DESCRIPTION: .EXIT provides the standard method for returning to the Monitor after completion of a system or user program. In the Basic I/O Monitor environment, it causes a program halt; in the Advanced Monitor environment, it causes the non-resident monitor to be reloaded. When the reloading process has been completed, the Monitor types

```
MONITOR
$
```

on the teleprinter, indicating that it is ready to accept the next command. In the Background/Foreground Monitor environment, the effect of the .EXIT depends upon whether it occurs in a BACKGROUND or a FOREGROUND job.

CHAPTER 3
BASIC INPUT/OUTPUT MONITOR FUNCTIONS

3.1 GENERAL

The Basic I/O Monitor simplifies the programming of input and output functions in the basic paper-tape environment. It serves as an interface between the system and user programs and the external world of device hardware, relying upon the routines and capabilities of the Input/Output Programming System (IOPS) to relieve the programmer of writing his own device and data handling subroutines. The I/O Monitor allows simultaneous operation of many I/O peripherals and overlapped computation. Since upward compatibility exists between the Monitor systems, user programs that are written to operate under control of the I/O Monitor will also operate, without modification, under control of the Advanced and Background/Foreground Monitors.

The Input/Output Monitor is designed to take advantage of the Automatic Priority Interrupt (API) if it is present on the system. Both the I/O skip chain for the Program Interrupt Control (PIC) and the API channels are set up to handle all devices which have been requested by the user. All unused channels are tied to an error routine to detect spurious interrupts.

The reader is referred to Chapter 1 for a general discussion of the Monitor environment, and to Chapter 2 for detailed descriptions of user program commands (system macros) available in the I/O Monitor environment.

3.2 PROGRAMMING EXAMPLE

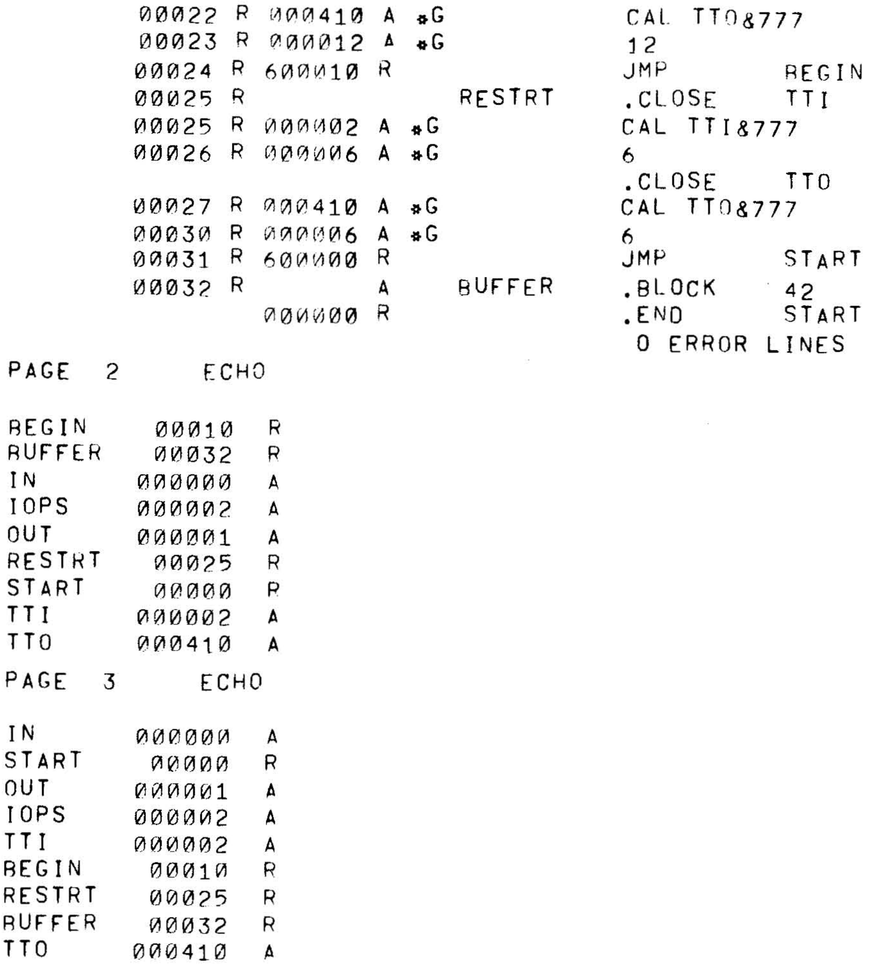
The following example illustrates the use of system macros with MACRO-15 programs in the Basic I/O Monitor environment. The example inputs a line of data from the Teletype keyboard, and outputs the same line of data to the Teletype. The arguments used by the system macros are given symbolic names (via MACRO-15 direct assignment statements) to facilitate recall for the programmer, and to change the arguments easily, if desired. Note the use of the pseudo-ops. .TITLE, .IODEV, .BLOCK, and .END, in addition to the system macros. The assembly listing that follows the example shows how the system macros are expanded at assembly time. (The reader may wish to compare these expansions with the system macro descriptions in Chapter 2.)

```
         .TITLE ECHO
 TTI=2
 TTO=41Ø
 OUT=1
 IN=Ø
 IOPS=2
        .IODEV  2,4
START   .INIT   TTO,OUT,RESTRT          /INITIALIZE TELETYPE OUTPUT
        .INIT   TTI,IN,RESTRT           /AND INPUT
BEGIN   .READ   TTI,IOPS,BUFFER,34      /INPUT IOPS ASCII FROM TELETYPE
        .WAIT   TTI                     /WAIT UNTIL INPUT COMPLETE
        .WRITE  TTO,IOPS,BUFFER,34      /OUTPUT SAME DATA ON TELETYPE
        .WAIT   TTO                     /WAIT UNTIL OUTPUT COMPLETE
        JMP     BEGIN                   /LOOP TO INPUT AGAIN
RESTRT  .CLOSE  TTI                     /TERMINATE INPUT
        .CLOSE  TTO                     /TERMINATE OUTPUT
        JMP     START                   /RETURN TO REINITIALIZE
BUFFER  .BLOCK  42                      /SET UP TELETYPE BUFFER (34 DEC)
        .END    START                   /END OF ECHO PROGRAM
```

ASSEMBLY LISTING:

```
PAGE   1      ECHO

                                              .TITLE ECHO
                 ØØØØØ2  A     TTI=2
                 ØØØ41Ø  A     TTO=41Ø
                 ØØØØØ1  A     OUT=1
                 ØØØØØØ  A     IN=Ø
                 ØØØØØ2  A     IOPS=2
                                              .IODEV     2,4
         ØØØØØ R               START          .INIT      TTO,OUT,RESTRT
         ØØØØØ R ØØ141Ø  A *G                 CAL+OUT*1ØØØ TTO&777
         ØØØØ1 R ØØØØØ1  A *G                 1
         ØØØØ2 R ØØØØ25  R *G                 RESTRT+Ø
         ØØØØ3 R ØØØØØØ  A *G                 Ø
                                              .INIT      TTI,IN,RESTRT
         ØØØØ4 R ØØØØØ2  A *G                 CAL+IN*1ØØØ TTI&777
         ØØØØ5 R ØØØØØ1  A *G                 1
         ØØØØ6 R ØØØØ25  R *G                 RESTRT+Ø
         ØØØØ7 R ØØØØØØ  A *G                 Ø
         ØØØ1Ø R               BEGIN          .READ      TTI,IOPS,BUFFER,34
         ØØØ1Ø R ØØ2ØØ2  A *G                 CAL+IOPS*1ØØØ TTI&777
         ØØØ11 R ØØØØ1Ø  A *G                 1Ø
         ØØØ12 R ØØØØ32  R *G                 BUFFER
                                              .DEC
         ØØØ13 R 777736  A *G                 -34
                                              .WAIT      TTI
         ØØØ14 R ØØØØØ2  A *G                 CAL TTI&777
         ØØØ15 R ØØØØ12  A *G                 12
                                              .WRITE     TTO,IOPS,BUFFER,34
         ØØØ16 R ØØ241Ø  A *G                 CAL+IOPS*1ØØØ TTO&777
         ØØØ17 R ØØØØ11  A *G                 11
         ØØØ2Ø R ØØØØ32  R *G                 BUFFER
                                              .DEC
         ØØØ21 R 777736  A *G                 -34
                                              .WAIT      TTO
```



```
        ØØØ22 R ØØØ41Ø A *G            CAL  TTO&777
        ØØØ23 R ØØØØ12 A *G            12
        ØØØ24 R 6ØØØ1Ø R               JMP        BEGIN
        ØØØ25 R               RESTRT   .CLOSE     TTI
        ØØØ25 R ØØØØØ2 A *G            CAL  TTI&777
        ØØØ26 R ØØØØØ6 A *G            6
                                       .CLOSE     TTO
        ØØØ27 R ØØØ41Ø A *G            CAL  TTO&777
        ØØØ3Ø R ØØØØØ6 A *G            6
        ØØØ31 R 6ØØØØØ R               JMP        START
        ØØØ32 R        A      BUFFER   .BLOCK     42
                ØØØØØØ R               .END       START
                                        0 ERROR LINES
PAGE  2       ECHO

BEGIN     ØØØ1Ø   R
BUFFER    ØØØ32   R
IN       ØØØØØØ   A
IOPS     ØØØØØ2   A
OUT      ØØØØØ1   A
RESTRT    ØØØ25   R
START     ØØØØØ   R
TTI      ØØØØØ2   A
TTO      ØØØ41Ø   A
PAGE  3       ECHO

IN       ØØØØØØ   A
START     ØØØØØ   R
OUT      ØØØØØ1   A
IOPS     ØØØØØ2   A
TTI      ØØØØØ2   A
BEGIN     ØØØ1Ø   R
RESTRT    ØØØ25   R
BUFFER    ØØØ32   R
TTO      ØØØ41Ø   A
```

3.3 OPERATING THE BASIC I/O MONITOR SYSTEM

The reader is referred to the PDP-15/10 Users Guide (DEC-15-GG1A-D) for detailed operating procedures for system programs in the I/O Monitor environment. The following PDP-15 Software System manuals contain additional detailed information on system programs.

Manual	Document Number
Utility Programs	DEC-15-YWZA-D
MACRO-15 Assembler	DEC-15-AMZA-D
FORTRAN IV	DEC-15-KFZA-D
FOCAL	DEC-15-KJZA-D
8TRAN	DEC-15-ENZA-D

This section contains descriptions of loading programs, device assignments, and error detection and handling.

3.3.1 Loading Programs in the Basic I/O Monitor Environment

In the paper tape system, each system program accompanied by the necessary I/O device handlers and an appropriate version of the Monitor, is punched on a separate paper tape in absolute format. See Figure 3-1 for memory maps of the Basic I/O Monitor System.

The eleven system tapes supplied are:

FORTRAN IV

FOCAL-15

MACRO-15

PIP-15

Text Editor

Linking Loader

DDT (without Patch File capabilities)

DDT (with Patch File capabilities)

8TRAN (PDP-8 to PDP-15 Translator)

CHAIN

EXECUTE

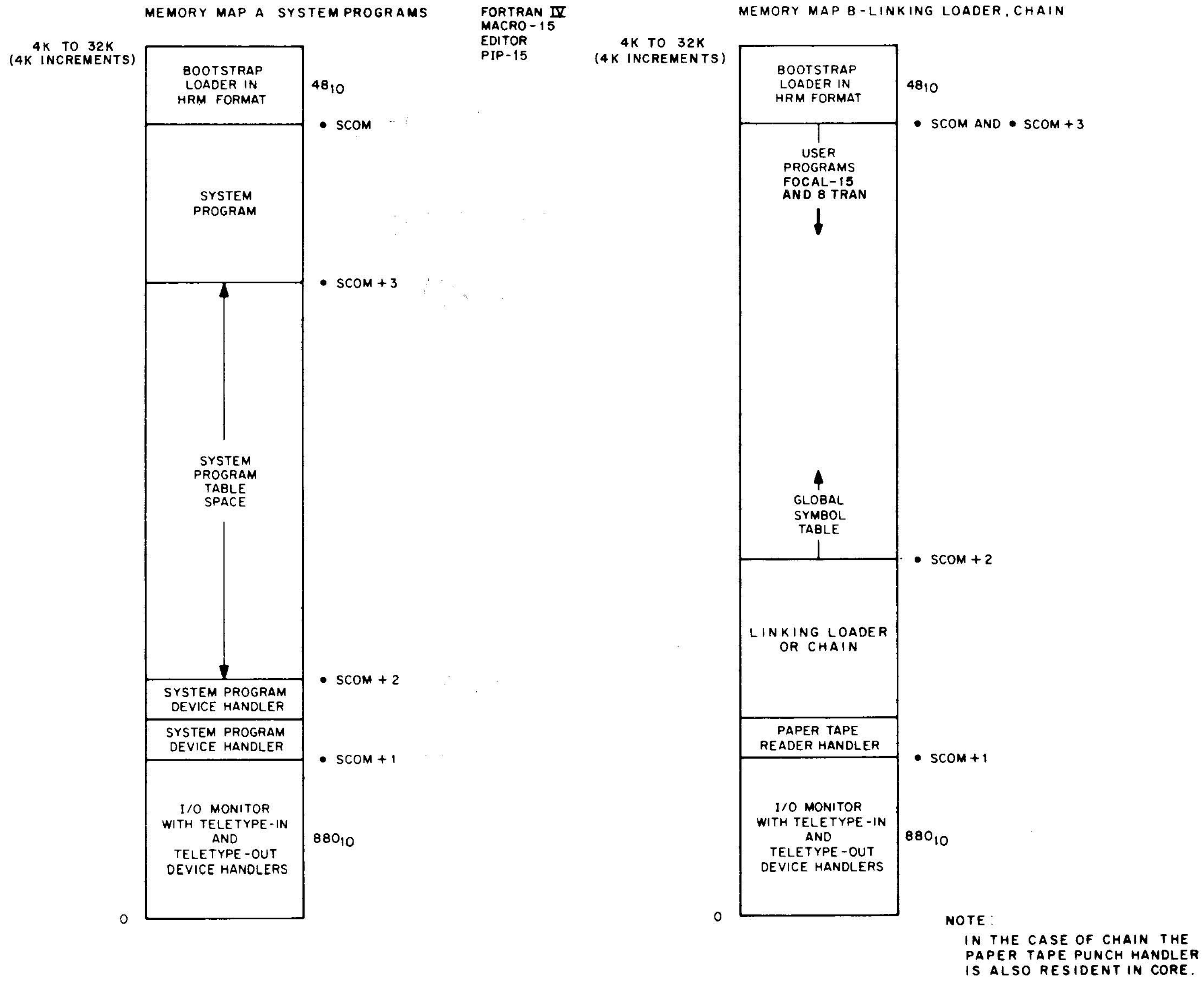
In addition, the utility program PUNCH-15 which provides the ability to dump an executable core load and .ABS loader onto paper tape, is provided with all paper tape systems.

At the beginning of each tape is a Bootstrap Loader in hardware READIN mode. By setting the starting address of the Loader on the console ADDRESS switches to 17720 (8K), depressing I/O RESET, and then depressing the READIN switch, these system tapes may be loaded.

Since the tapes also contain appropriate versions of the Basic I/O Monitor and the necessary I/O device handlers, the system programs listed above can be loaded, ready for operation, in a single step.

Once the system program has been loaded and takes control, the individual system program operating procedures come into use. (See PDP-15/10 Users Guide, DEC-15-GG1A-D.)

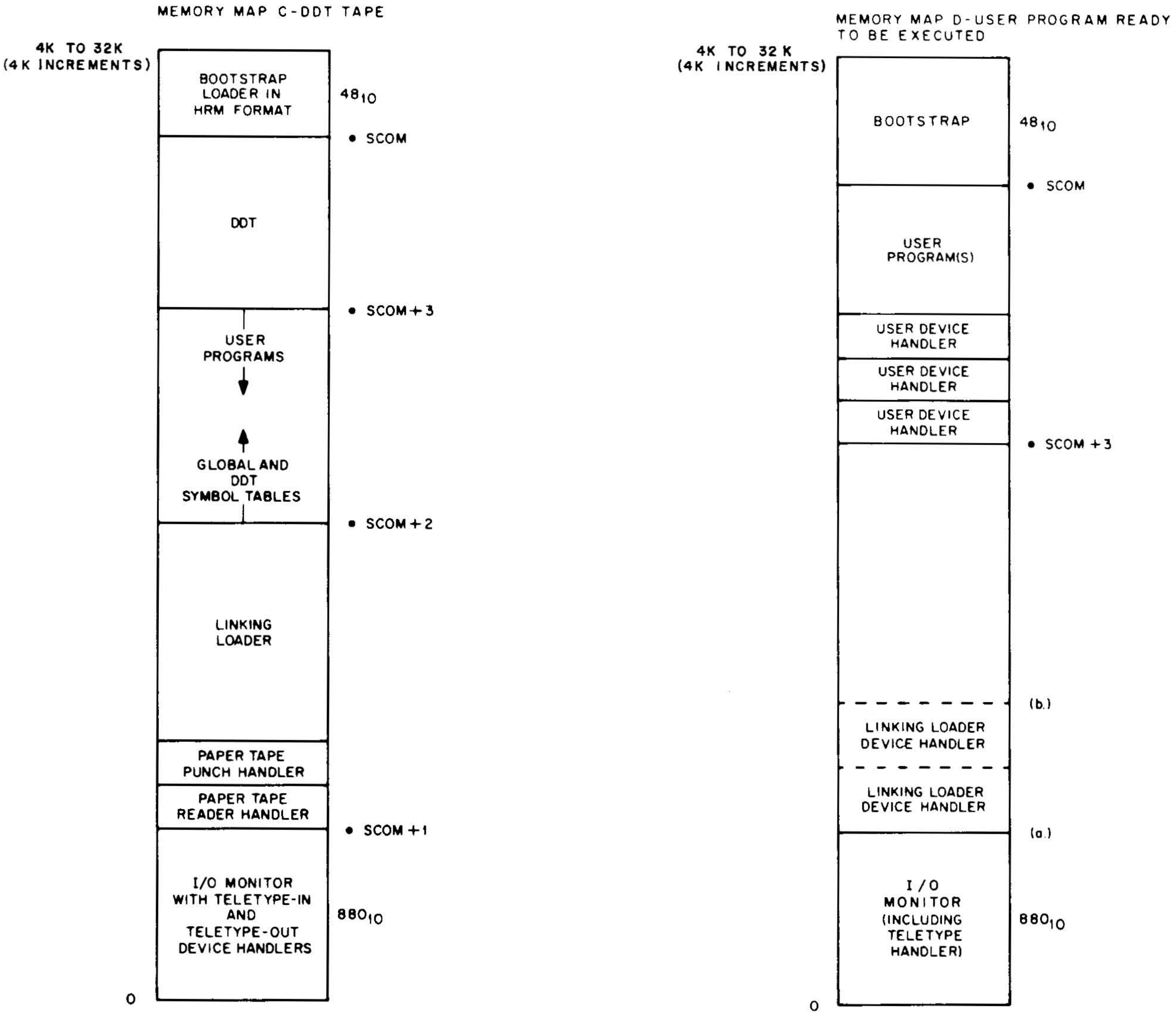
User programs, however, normally exist in relocatable form, as output from FORTRAN IV or MACRO-15, these tapes do not contain copies of the Monitor. To load these programs, the Linking Loader or DDT should be loaded first. The user should then initiate loading of his main program followed by all required subprograms. By loading subprograms in order of size (largest first, smallest last), the user has a better chance of satisfying core requirements for his program in systems with extended core memory. The Monitor (including the device handlers) contained on the Linking Loader or DDT tape may be used with user programs, and the Linking Loader or DDT can be used to load the necessary device handlers as well as the user's object programs.


MEMORY MAP A SYSTEM PROGRAMS
FORTRAN IV
MACRO-15
EDITOR
PIP-15
4K TO 32K
(4K INCREMENTS)
BOOTSTRAP LOADER IN HRM FORMAT
48_{10}
• SCOM
SYSTEM PROGRAM
• SCOM + 3
SYSTEM PROGRAM TABLE SPACE
• SCOM + 2
SYSTEM PROGRAM DEVICE HANDLER
SYSTEM PROGRAM DEVICE HANDLER
• SCOM + 1
I/O MONITOR WITH TELETYPE-IN AND TELETYPE-OUT DEVICE HANDLERS
880_{10}
0
MEMORY MAP B-LINKING LOADER, CHAIN
4K TO 32K
(4K INCREMENTS)
BOOTSTRAP LOADER IN HRM FORMAT
48_{10}
• SCOM AND • SCOM + 3
USER PROGRAMS FOCAL-15 AND 8 TRAN
GLOBAL SYMBOL TABLE
• SCOM + 2
LINKING LOADER OR CHAIN
PAPER TAPE READER HANDLER
• SCOM + 1
I/O MONITOR WITH TELETYPE-IN AND TELETYPE-OUT DEVICE HANDLERS
880_{10}
0
NOTE:
IN THE CASE OF CHAIN THE PAPER TAPE PUNCH HANDLER IS ALSO RESIDENT IN CORE.

Refer to Section 4.4 for sizes for device handlers.

Refer to Memory Map D for results of Linking Loader

Figure 3-1. BASIC I/O Monitor System Memory Maps



Refer to Memory Map E for results of Link Loading in DDT mode.

Paper Tape Punch Handler is only present in DDT versions with patch file capabilities.

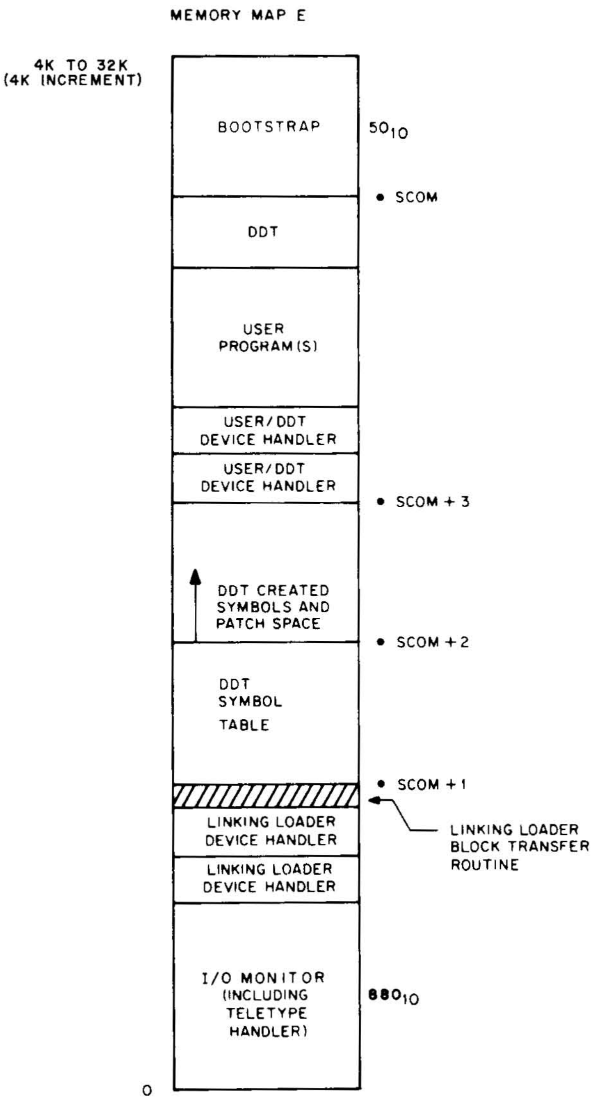
Refer to Section 4.4 for sizes of device handlers.

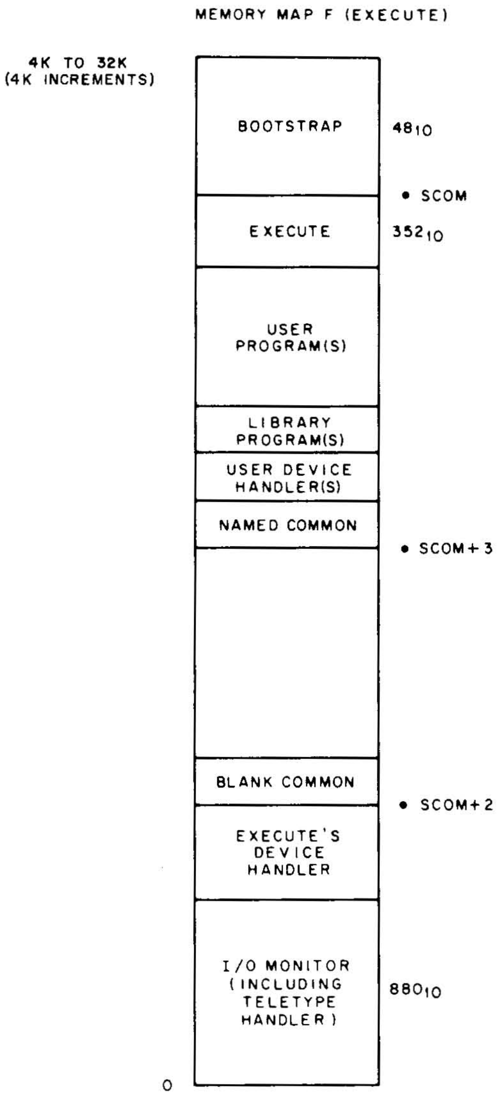
.SCOM+1 and .SCOM+2 both point to one of two places and non-BLOCK DATA COMMON (FORTRAN IV or MACRO-15) output may make use of core as low as they point.

a. If the user program did not have any device handlers in common with the Linking Loader.

b. If the user program did have at least one device handler in common with the Linking Loader.

Figure 3-1. BASIC I/O Monitor System Memory Maps (Cont)





Refer to Section 4.4 for sizes for device handlers.

Non BLOCK DATA COMMON (FORTRAN IV of MACRO-15 output) may make use of core as low as the DDT symbol table. However, trouble will occur if the user requests DDT to create symbols or make patches that cause overlaying of the COMMON area.

The Linking Loader device handlers would have been used to satisfy user device requests.

When the user types in the name of the XCT File to be run, .EXECUTE brings in the first chain from paper tape.

FORTRAN programs pass on data in blank common starting at .SCOM + 2. Macro programs pass on data between .SCOM + 2 and .SCOM + 3.

A call from the running chain to bring in another chain is effected by transferring control back to Execute.

Figure 3-1. BASIC I/O Monitor System Memory Maps (Cont)

3.3.2 Device Assignments

The device assignment table used by the Basic I/O Monitor is fixed in length and in the assignments it contains. It is composed of two sections; the upper section is for use by all system programs except PIP, the lower section is referenced by all user programs and PIP.

The upper portion of the .DAT contains 13 slots, referenced as -1 through -15_8. The lower section has 8 slots numbered 1 through 10_8. The standard assignments for the device assignment table for user programs and system programs other than PIP are shown in Figure 3-2. Figure 3-3 illustrates PIP assignments, which include the Card Reader (CR03B) and Line Printer as standard devices.

.DAT Slot		Device	Handler*	Use
.DATBG	-15	Paper Tape Punch	(PPA.)	Editor Output
		Paper Tape Reader	(PRA)	8TRAN Input
	-14	Paper Tape Reader	(PRA.)	Editor Input
		Paper Tape Punch	(PPA)	8TRAN Output
	-13	Paper Tape Punch	(PPB.)	MACRO, FORTRAN IV Output
	-12	TTY Printer	(TTA.)	MACRO, FORTRAN IV Listing
	-11	Paper Tape Reader	(PRB.)	MACRO, FORTRAN IV Input
	-10	Paper Tape Reader	(PRA.)	DDT Patch-file Input and Editor Secondary Input
		TTY Keyboard	(TTA.)	MACRO Secondary Input
	-7	0		Not Used
	-6	Paper Tape Punch	(PPA.)	Output (.DDT, Chain)
	-5	0		Not Used
	-4	Paper Tape Reader	(PRA.)	System Input (Linking Loader, DDT Chain and Execute)
	-3	TTY Printer	(TTA.)	Teleprinter Output
	-2	TTY Keyboard	(TTA.)	Keyboard Input
	-1	Paper Tape Reader	(PRA.)	System Device (Linking Loader, DDT and Chain)
DAT	.DAT			
	1	TTY Printer	(TTA.)	Teleprinter Output
	2	TTY Keyboard	(TTA.)	Keyboard Input
	3	Paper Tape Reader	(PRA.)	Input
	4	TTY Printer	(TTA.)	Listing
	5	Paper Tape Punch	(PPA.)	Output
	6	Paper Tape Reader	(PRA.)	Scratch
	7	Paper Tape Punch	(PPA.)	Scratch
	10	Paper Tape Reader	(PRA.)	Scratch
DATND=.				

Figure 3-2. Device Assignment Table (.DAT) for the Basic I/O Monitor

The negative .DAT slot assignments for use by system programs may be changed by DEC. For example, .DAT slot -10 might be associated with a card reader, while .DAT slot -12 could be assigned to a line printer. Provision for changing positive .DAT slot assignments for use by relocatable user programs is included in the PUNCH 15 utility program (see section 3.3.5). For example, a line printer handler (LPA) or a card reader handler (CDB) could be added.

*See Section 4.4 for a description of the handlers.

.DAT Slot	Device	Handler	Use
1	Teletype	(TTA.)	Input/Output
2	Teletype	(TTA.)	Input/Output
3	Paper Tape Reader	(PRA.)	Input
4	Line Printer	(LPA.)	Output
5	Paper Tape Punch	(PPA.)	Output
6	Card Reader	(CDB.)	Input
7	Paper Tape Punch	(PPA.)	Output
10	Paper Tape Reader	(PRA.)	Input

Figure 3-3. Device Assignment Table (.DAT) for PIP

3.3.3 Error Detection and Handling

Comprehensive error checking is provided by the Linking Loader and the Input/Output Programming System. Detailed lists of errors that may occur are given in Appendix C and D, respectively. The other system programs also provide comprehensive error checking. Refer to the appropriate PDP-15 manual (see paragraph 1.3 in Part I of this manual).

3.3.4 Control Character Commands in the Basic I/O Monitor Environment

All control character commands recognized by the Basic I/O Monitor are summarized in Table 3-1. These commands (except RUBOUT) are formed by holding down the CTRL key while striking a letter key. The character or characters echoed on the Teletype and the resulting action is given in the table for each command.

3.3.5 Modifying System Programs and Building Executable User Core Loads in the Basic I/O Monitor Environment

The capability of modifying or patching system programs in the Basic I/O Monitor environment is provided by the utility program PUNCH 15. PUNCH 15 allows for producing an executable core load on paper tape in .ABS format with the standard .ABS loader (HRM 17720 of the highest available core bank) on the front of the tape. This is particularly useful when the core load consists of a relocatable main program, subroutines and library routines, the repetitive loading of which tends to be time consuming. It is further possible to specify the number of .ABS tapes to be output (1-9) for convenience of tape handling.

The areas of memory output by PUNCH 15 are 0 up to .SCOM + 2 (the first free cell) and .SCOM + 3 (last free cell) up to the .ABS loader (17720 modulo 8K). PUNCH 15 is loaded (I/O RESET and READIN) at 17720 of the highest core bank available. It loads and relocates part II of itself in free core, i.e., from the cell in .SCOM + 2 and up. When loaded it types the following message on Teletype:

```
P, T or S?↵
>
```

Table 3-1. Control Character Commands

Command	Echo	Action
CTRL S	↑ S	Starts user program after Linking Loader has brought it into core via a LOAD command.
CTRL T	↑ T	CTRL T is applicable only when using DDT and transfers control to DDT which types DDT > to indicate its readiness for another DDT command. All previous DDT conditions remain intact (breakpoints, register modifications, etc.).
CTRL R	↑ R	Allows the user to continue when an IOPS4 (device not ready) error occurs. The user must first ready the device, and then type CTRL R.
CTRL P	↑ P	Forces control to last address specified in the .INIT command referencing Teletype. Used by system programs to reinitialize or restart.
CTRL D		Generates EOT code – used to terminate Teletype input.
CTRL U	@	Cancels current line on Teletype (input or output).
RUBOUT	\	Cancels last character input from Teletype (not applicable with DDT).

The user is expected to type CTRL P or CTRL T or CTRL S to define the starting location of the program to be punched followed by carriage return (1 tape) or a number (1-9) specifying the total number of tapes into which binary output is to be divided.

All tapes output by PUNCH 15 are loaded by pressing I/O RESET and the READIN key with ADDRESS switches set to 17720 of the highest core bank.

CTRL T should be used if DDT is part of the core load.

CTRL P should be used for all other system programs and user programs which have already initialized (.INIT) the teletype with a restart address at the time PUNCH 15 was executed.

CTRL S should be used only if the core load was output by PUNCH 15 after Linking Loader operation at the moment when the loader itself was expecting the CTRL S command.

All tapes output by PUNCH 15 are loaded by pressing I/O RESET and the READIN key with ADDRESS switches to 17720 of the highest core bank.

3.3.6 Modifying User .DAT Slots in the I/O Monitor Environment

Although it is not possible to reassign negative .DAT slots at load time in the Basic I/O Monitor environment for system programs (reassembly is required), PUNCH 15 provides this capability for the user or positive .DAT slots.

For example, Figure 3-2 lists the standard .DAT slot assignments. A relocatable user program wanting to use a card reader (CDB.) or line printer (LPA.) whose handlers are included in the paper tape library has no way of doing so unless the positive .DAT table can be modified. The modification procedure is as follows:

1. Load the Linking Loader (or DDT) tape into core (HRM 17720 modulo 8K).

2. Stop the computer and modify the appropriate .DAT slot cell* according to the Loader-I/O correspondence table below. For example, if .DAT slot 7 is to be assigned to the line printer using handler LPA, cell 144 should be changed to 000007.

3. Load PUNCH into core (HRM 17720 modulo 8K).

4. Type CTRL P in response to the query from PUNCH: P, T or S?

5. Load the resultant punched tape into core (HRM 17720 modulo 8K).

6. The Loader will restart ready for acceptance of typed program names to be loaded. If .DAT slot 7 is referenced by the user program (e.g., IODEV 7), LPA for the line printer will be loaded from the I/O Library.

Loader - I/O Correspondence

Handler	*.DAT Slot Value*
TTA.	1
PRA.	2
PRB.	3
PPA.	4
PPB.	5
PPC.	6
LPA.	7
CDB.	11
MTF	13*

*.DAT table (cell 0) begins at location 135

**With unit number in bits 0-2.

CHAPTER 4
I/O DEVICE HANDLERS

This chapter contains information that is essential for a good understanding and proper use of I/O device handlers for the Basic I/O Monitor system. Included is a general description of I/O hardware and API software level handlers, a complete section on writing special I/O device handlers, a summary of I/O handlers acceptable to system programs, and a summary of standard I/O handler features.

It is assumed that the reader is familiar with all related material in the PDP-15 Reference Manual (DEC-15-BRZA-D), especially Chapter 5, Organization of the Input/Output Processor. It is also assumed that the reader is familiar with the PDP-15 Monitor environment and other pertinent information contained in this manual.

4.1 DESCRIPTION OF I/O HARDWARE AND API SOFTWARE LEVEL HANDLERS

4.1.1 I/O Device Handlers

All communications between user programs and I/O device handlers are made via CAL instructions (see Chapter 2) followed by argument lists. The CAL Handler in the Monitor performs preliminary setups, checks on the CAL calling sequence, and transfers control via a JMP instruction to the entry point of the device handler. When the control transfer occurs, the AC contains the address of the CAL in bits 3 through 17 and bits 0, 1, and 2 indicate the status of the Link, extend mode and memory protect, respectively, at the time of the CAL. Note that the content of the AC at the time of the CAL is not preserved.

On machines that have an API, the execution of a CAL instruction automatically raises the priority to the highest software level (level 4). Control passes to the handler while it is still at level 4, allowing the handler to complete its re-entrant procedures before debreaking (DBK) from level 4. This permits the handler to receive re-entrant calls from software levels higher than the priority of the program that contained this call. If a device handler does not contain re-entrant procedures, system failure caused by inadvertent re-entries can be prevented by remaining at level 4 until control is returned to the user.

If the non-reentrant method is used, the debreak and restore (DBR) instruction should be executed just prior to the JMP* which returns control to the user, allowing debreak from level 4 and restoring the conditions of the Link, extend mode, and memory protect. Any IOTs issued at the CAL level (level 4 if API present, mainstream if no API) should be executed immediately before the

```
DBR
XCT  .+1
JMP*
```

exit sequence to ensure that the exit takes place before the interrupt from the issued IOT occurs. (The XCT is necessary to ensure that the 3 cycles requested by the API on a debreak operation occur in the instruction after the DBR.)

The CAL instruction must not be used at any hardware priority level (API or PIC), since interrupts to these levels are not closed out by the execution of a CAL and recovery is not possible from such sequences of events as

a. An I/O flag coming up during a CAL at level 7,

b. Control going to the I/O device handler at level 3,

c. The handler at level 3 CALing and thus destroying the content of location 00020 for the previous CAL.

The highest API software level (level 4) is also used for processing CALs and care must be taken when executing CALs at this level. For example, a routine that is CAL'd from level 4 must know that if a debreak (DBR or DBK) is issued, control will return to the calling program at a level lower than 4. The calling routine will also debreak; however, this second debreak will not be from level 4 but from the next highest active level.

4.1.1.1 **Setting Up the Skip Chain and API (Hardware) Channel Registers** – When the Monitor is loaded, the Program Interrupt Control (PIC) skip chain and the Automatic Priority Interrupt (API) channels are set up to handle the Teletype keyboard, teleprinter and clock interrupts, only. The skip chain contains the other skip IOT instructions, but indirect jumps to an error routine result if a skip occurs, as follows:

```
SKP         PRA         /Skip if Reader flag.
SKP
JMP*        INIT1       /INT1 contains error address.
SKP         LPT         /Skip if line printer flag.
            SKP
JMP*        INT2        /INT2 contains error address.
SKP         TT1         /Skip if Teletype flag.
SKP
JMP         TELINT      /To Teletype interrupt handler.
              .
              .
              .
```

All unused API channels also contain JMPs to the error address.

When a device handler is called for the first time via an .INIT user program command, it must call a Monitor routine (.SETUP) to set up its skip chain entry or entries and API channel, prior to performing any I/O functions. The calling sequence is as follows.

```
          CAL            N          /N = API channel register 40 through 77 (see section
                                    /4.1.3 for standard channel assignments), 0 if device
                                    /not connected to API.
          16                        /.SETUP function code.
          SKP            IOT        /Skip IOT for this device.
          DEVINT                    /Address of interrupt handler.
          (normal return)
```

DEVINT exists in the device handler in the following format.

```
DEVPIC    DAC       DEVAC           /SAVE AC.
          LAC*      (0
          DAC       DEVOUT          /SAVE PC, LINK, EX.MODE, MEM.PROT.
          LAC       DEVION          /FORCE ION AT DISMISSAL.
          JMP       DVSTON
DEVINT    JMP       DEVPIC          /PIC ENTRY.
          DAC       DEVAC           /API ENTRY, SAVE AC.
          LAC       DEVINT
          DAC       DEVOUT          /SAVE PC, LINK, EX.MODE, MEM.PROT.
          IORS                      /CHECK STATUS OF PIC
          SMA!CLA                   /FOR RESTORATION AT DISMISSAL.
          LAW       17740           /PIC OFF, BUILD IOF IOT.
          TAD       DEVION          /PIC ON, BUILD ION IOT.
DVSTON    DAC       DVSWCH
          DEVCF                     /CLEAR DEVICE DONE FLAG
DEVION    ION                       /ENABLE PIC SO THAT OTHER DEVICES
                                    /AREN'T SHUT OUT.
            .
            .
            .
          IOF                       /DISABLE PIC TO INSURE
          DEVIOT                    /DISMISSAL BEFORE INTERRUPT
            .                       /FROM THIS IOT OCCURS
            .
            .
/DISMISS ROUTINE
          LAC       (JMP DEVPIC     /RESTORE DEVINT IN
          DAC       DEVINT          /CASE API DISABLED.
          LAC       DEVAC           /RESTORE AC
DVSWCH    ION                       /ION OR IOF
          DBR                       /DEBREAK AND RESTORE CONDITIONS
          JMP*      DEVOUT          /OF LINK, EX.MODE AND MEM.PROT.
```

Since the auto-index registers and EAE registers are not used by the standard I/O device handlers, it is not necessary to save and restore them.

The Monitor routine (.SETUP) checks the skip chain for the instruction which matches SKP IOT; if there is a match it places the address, DEVINT, in the appropriate transfer vector (INTn) and places JMS* INTn in the corresponding API channel register. If a match cannot be found, IOPS outputs the following error message,

```
          .IOPS  05  XXXXXX
```

indicating that the skip IOT in the CAL calling sequence at location XXXXXX was not in the skip chain.

Refer to paragraph 4.2.4 for the method of incorporating new handlers and associated skip chain entries into the Monitor.

4.1.2 **API Software Level Handlers**

4.1.2.1 **Setting Up API Software Level Channel Registers** – When the Monitor is loaded, the API software-level channel registers (40 through 43) are initialized to

```
JMS*    .SCOM+12                    /LEVEL 4
JMS*    .SCOM+13                    /LEVEL 5
JMS*    .SCOM+14                    /LEVEL 6
JMS*    .SCOM+15                    /LEVEL 7
```

where the .SCOM registers are at absolute locations 00112 through 00115 and contain the address of an error routine.

Therefore, prior to requesting any interrupts at these software priority levels, the user must modify the contents of the .SCOM registers so that they point to the entry point of the user's software level handlers.

Example:

```
.SCOM = 100

     LAC            (LV5INT
     DAC*           (.SCOM+13
       .
       .
       .
```

LV5INT exists in the user's area in the following format:

```
LV5INT  0                               /PC, LINK AND MEM. PROT.
        DAC                     SAV5AC  /SAVE AC
        /SAVE AUTO INDEX REGISTERS
        /IF LEVEL 5 ROUTINES
        /USE THEM AND LOWER LEVEL
        /ROUTINES ALSO USE THEM
        /SAVE MQ AND STEP COUNTER
        /IF SYSTEM HAS EAE AND IT
        /IS USED AT DIFFERENT LEVELS.
              .
              .
              .
        /RESTORE SAVED REGISTERS.
        DBR                             /DEBREAK FROM LEVEL 5
        XCT                     .+1
        JMP*                    LV5INT  /AND RESTORE LINK AND MEM.PROT.
```

4.1.2.2 **Queueing** – High priority/high data rate/short access routines cannot perform complex calculations based on unusual conditions without holding off further data inputs. To perform the calculations, the high priority program segment must initiate a lower priority (interruptable) segment to perform the calculations. Since, in general, many data handling routines will be requesting calculations, there will exist a queue of calculation jobs waiting to be performed at the software level. Each data handling routine must add its job request to the appropriate queue (taking care to raise the API priority level as high as the highest level that manipulates the queue before adding the request) and issue an interrupt request (ISA) at the corresponding software priority level. The general flow chart, Figure 4-1 depicts the structure of a software level handler involved with queued requests.

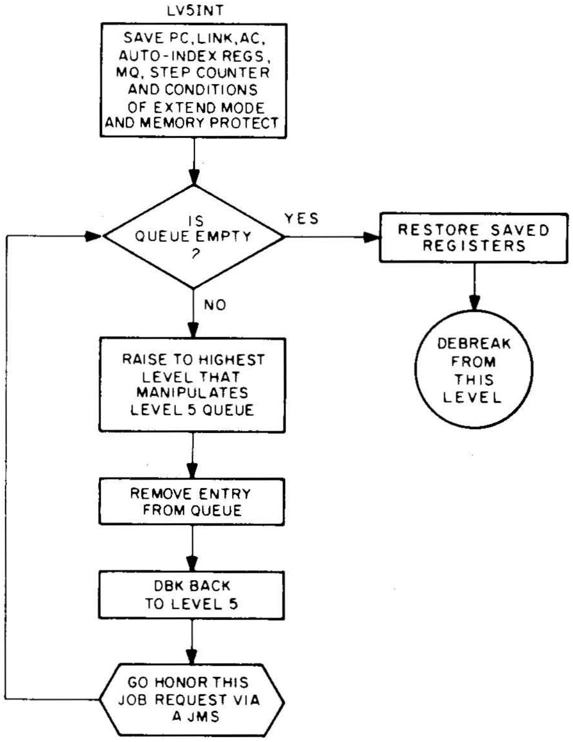

LV5INT
SAVE PC, LINK, AC, AUTO-INDEX REGS, MQ, STEP COUNTER AND CONDITIONS OF EXTEND MODE AND MEMORY PROTECT
IS QUEUE EMPTY ?
YES
NO
RESTORE SAVED REGISTERS
DEBREAK FROM THIS LEVEL
RAISE TO HIGHEST LEVEL THAT MANIPULATES LEVEL 5 QUEUE
REMOVE ENTRY FROM QUEUE
DBK BACK TO LEVEL 5
GO HONOR THIS JOB REQUEST VIA A JMS

Figure 4-1. Structure of API Software Level Handler

Care must be taken about which routines are called when a software level request is honored; that is, if a called routine is "open" (started but not completed) at a lower level, it must be reentrant or errors will result.

NOTE

The standard hardware I/O device handlers do not contain reentrant procedures and must not be reentered from higher software levels.

New resident handlers for Power Fail, Memory Parity, nonexistent memory violation, and Memory Protect violation have been incorporated into the system and effect an IOPS error message if the condition is detected (see Appendix E for IOPS errors). The user can, via a .SETUP, tie his own handler to these skip IOT or API channel registers.

4.1.3 **Standard API Channel/Priority Assignments**

Channel	Device	Option Number	Priority	Channel Register
0	Software Priority	–	4	40
1	Software Priority	–	5	41
2	Software Priority	–	6	42
3	Software Priority	—	7	43
4	DECtape	TC02	1	44
5	MAGtape	TC59	1	45
6	RESERVED	—	—	46
7	RESERVED	—	—	47
10	Paper Tape Reader	–	2	50
11	Clock overflow	–	3	51
12	Power Fail	KF15	0	52
13	Parity	MP15B	0	53
14	Display (LP flag)	VP15	2	54
15	Card Reader	CR03B	2	55
16	Line Printer	LP15A/LP15C	2	56
17	A/D	AF01/ADC1/9	0	57
20	DB99/DB98	DB09	3	60
21	RESERVED	–	–	61
22	Data Phone	DP09A	2	62
23	DECdisk	RF15	1	63
24	DISK Pack	RP15	1	64
25	Plotter	565	2	65

Channel	Device	Option Number	Priority	Channel Register
34	Multi-Station TTY Control	LT19A (Teleprinter)	2	74
35	Multi-Station TTY Control	LT19A (Keyboard)	2	75

NOTE: Channels 26-33, 36 and 37 and corresponding Channel Registers are reserved.

4.2 WRITING SPECIAL I/O DEVICE HANDLERS

This section contains information prepared specifically to aid those users who plan to write their own special I/O device handlers for the Basic I/O Monitor system.

Although special handlers cannot be incorporated directly into the Basic Monitor System, they can be designed to run with user programs in the Basic Monitor environment (see paragraph 4.2.4). If a user wishes to incorporate a special handler into a systems program (for example, a card punch handler for MACRO 15), he must purchase the source tape and assemly listings, modify them symbolically, and reassemble using at least a 16K machine.

It is assumed that the user is familiar with paragraph 4.1. To summarize, the handler is entered via a JMP from the Monitor as a result of a CAL instruction. The contents of the AC contain the address of the CAL in bits 3 through 17. Bit 0 contains the Link, bit 1 contains the extend mode status, and bit 2 contains the memory protect status. The previous contents of the AC and Link are lost.

To show the steps required in writing an I/O device handler, a complete handler (Example B) was developed with the aid of a skeleton handler (Example A). This handler is a non-reentrant type (discussed briefly at the beginning of this chapter) and uses the Debreak and Restore instruction (DBR) to leave the handler at software priority level 4 (if API), and restore the status of the Link, extend mode, and memory protect. Example A is referenced by part numbers to illustrate the development of Example B, a finished Analog to Digital Converter (ADC) I/O Handler. The ADC handler shown in Example B, was written for a hypothetical basic analog-to-digital converter with IOT instructions as shown in Table 4-1. This handler is used to read data from the ADC and store it in the user's line buffer. The handler shown in Example B is for instructional purposes only; it has not been thoroughly tested.

TABLE 4-1. Hypothetical A/D Converter IOT Instructions

Mnemonic Symbol	Octal Code	Operation Executed
ADSF	701301	Skip if converter flag is set. This flag is connected to the program interrupt.
ADSC	701304	Select and convert. The converter flag is cleared and a conversion of an incoming voltage is initiated. When the conversion is complete, the converter flag is set.
ADRB	701312	Read converter buffer. Places the content of the buffer in the AC, left adjusted. The remaining AC bits are cleared. The converter flag is cleared.

The reader, while looking at the skeleton of a specialized handler as shown in Example A, should make the following decisions about his own handler (The decisions made in this case are in reference to developing the ADC handler):

a. *Services that are required of the handler* (flags, receiving or sending of data, etc.). By looking at the ADC IOT's shown in Table 4-2, it can be seen that there are three IOT instructions to be implemented. These instructions are: Skip if Converter Flag Set; Select and Convert; and Read Converter Buffer.

The only service the ADC handler performs is that of receiving data and storing it in user specified areas. This handler will have a standard 256-word buffer.

b. *Data Modes used (for example, IOPS ASCII, etc.).* Assuming that there is only one data mode for this device, mode specification is unnecessary in Example B.

c. *Which I/O macros are needed* for the handler's specific use, that is, .INIT, .CLOSE, .READ, etc. These are fully described in Chapter 2 of this manual. For an ADC, the user would be concerned with three of the macros.

> .INIT would be used to set up the associated API channel register and the interrupt skip IOT sequence in the Program Interrupt (PIC) skip chain. This is done by a CAL (N) as shown in Part III of Example A, where (N) is the channel address. The standard device/API channel associations can be found in paragraph 4.1.3.
>
> .READ is used to transfer data from the ADC. When the .READ macro is issued, the ADC handler will initiate reading of the specified number of data words and then return control to the user. The analog input data received is in its raw form; it is up to the programmer to convert the data to a usable format.
>
> .WAIT detects the availability of the user's buffer area and ensures that the I/O transfer is completed. It would be used to ensure a complete transfer before processing the requested data.

d. *Implementation of the API or PIC* interrupt service routine. Example A shows an API or PIC interrupt service routine that handles interrupts, processes the data and initiates new data requests to fully satisfy the .READ macro request. Note that the routines in Example A will operate with or without API. Example B used the routines exactly as they are shown in Example A.

During the actual writing of Example B, consideration was given to the implementation of the I/O Macros in the new handler in one of the following ways:

(1) *Execute the function* in a manner appropriate to the given device as discussed in (c). .INIT, .READ, .WAIT were implemented into the ADC handler (Example B) under the subroutine names ADINIT, ADREAD, ADWAIT.

Wait for completion of previous I/O. (Example B shows the setting of the ADUND switch in the ADREAD subroutine to indicate I/O underway.)

(2) *Ignore the function* if meaningless to the device. See Example B (.FSTAT results in JMP ADIGN2) in the dispatch table DSPCH. For ignored macros, the return address must be incremented depending upon the argument string after the CAL. The number of arguments for each macro is shown in Chapter 2.

(3) *Issue an error message* in the case where it is not possible to perform the I/O function. (An example would be trying to execute an .ENTER on the paper tape reader.) In Example B the handler jumps to DVERR6 which returns to the Monitor with a standard error code in the AC.

After the handler has been written and assembled, the user must assemble the handler and splice it to the I/O Library Tape. This procedure is described in 4.2.4.

4.2.1 Discussion of Example A by Parts

Part 1	Stores CAL pointer and argument pointer; also picks up function code from argument string.
Part 2	By getting proper function code in Part 1 and adding a JMP DSPCH, the CAL function is dispatched to proper routine.
Part 3	This is the .SETUP CAL used to set up the API channel register and PIC skip chains. Section 4.1.3 of this manual shows the standard device/API associations.
Part 4	Shows the API and PIC handlers. It is suggested these be used as shown.
Part 5	This area reserved for processing interrupt and performing any additional I/O.
Part 6	Interrupt dismiss routine.
Part 7	Increments argument pointer in bypassing arguments of ignored macro CAL's.

4.2.2 Example A, Skeleton I/O Device Handler

Part 1

```
/SPECIALIZED I/O HANDLER
/CAL ENTRY ROUTINE
        .GLOBL  DEV.             /MUST BE OF FORM AAA.
.MED=3                           /.MED (MONITOR ERROR DIAGNOSTIC)
DEV.    DAC     DVCALP           /SAVE CAL POINTER
        DAC     DVARGP           /AND ARGUMENT POINTER
        ISZ     DVARGP           /POINTS TO FUNCTION CODE
        LAC*    DVARGP           /GET CODE
        AND     (77777           /REMOVE UNIT # IF APPLICABLE
        ISZ     DVARGP           /POINTS TO CAL + 2
        TAD     (JMP DSPCH)
        DAC     DSPCH            /DISPATCH WITH
```

Part 2

```
DSPCH   XX                       /MODIFIED JUMP
        JMP     DVINIT           /1 = .INIT
        JMP     DVFSAT           /2 = .FSTAT, .DELET, .RENAM
        JMP     DVSEEK           /3 = .SEEK
        JMP     DVENTR           /4 = .ENTER
        JMP     DVCLER           /5 = .CLEAR
        JMP     DVCLOS           /6 = .CLOSE
        JMP     DVMTAP           /7 = .MTAPE
        JMP     DVREAD           /1Ø = .READ
        JMP     DVWRTE           /11 = .WRITE
        JMP     DVWAIT           /12 = .WAIT
        JMP     DVTRAN           /13 = .TRAN
```

Part 2 (cont)

```
/ILLEGAL FUNCTIONS IN ABOVE TABLE CODED AS:
/       JMP     DVERR6

/FUNCTION CODE ERROR
DVERR6  LAW 6                   /ERROR CODE 6
        JMP*    (.MED+1)        /TO MONITOR

/DATA MODE ERROR
DVERR7  LAW 7                   /ERROR CODE 7
        JMP*    (.MED+1)        /TO MONITOR

/DEVICE NOT READY
DVERR4  LAC     (RETURN)        /RETURN (ADDRESS IN HANDLER
                                /TO RETURN TO WHEN NOT READY
                                /CONDITION HAS BEEN REMOVED)
        DAC*    (.MED)
        LAC (4)                 /ERROR CODE 4
        JMP*    (.MED+1)        /TO MONITOR



/I/O UNDERWAY LOOP
DVBUSY  DBR                     /BREAK FROM LEVEL 4
        JMP*    DVCALP          /LOOP ON CAL

/NORMAL RETURN FROM CAL
DVCK    DBR                     /BREAK FROM LEVEL 4
        JMP*    DVARGP          /RETURN AFTER CAL AND
                                /ARGUMENT STRING.
```

Part 3

```
/THE DVINIT ROUTINE MUST INCLUDE
/A. SETUP FOR
/EACH FLAG CONNECTED TO PIC (AT BUILD TIME)
/ONE OF THESE MAY ALSO BE THE API SETUP CALL.
/THE SETUP CALLING SEQUENCE IS:

DVINIT  CAL     N               /N = API CHANNEL REGISTER
                                /(40-77); Ø IF NOT
                                /CONNECTED TO API

        16                      /IOPS FUNCTION CODE
        SKP IOT                 /SKIP IOT TO TEST THE FLAG
        DBVINT                  /ADDRESS OF INTERRUPT
                                /HANDLER (API OR PIC)
```

```
/THIS SPACE CAN BE USED FOR I/O SUBROUTINES
```

Part 4

```
/INTERRUPT HANDLER FOR API OR PIC
DEVPIC  DAC     DEVAC           /SAVE AC
        LAC*    (Ø)             /SACE PC, LINK,
        DAC     DEVOUT          /MEM. PROT.
        LAC     DEVION          /FORCE ION AT DISMISSAL
        JMP     DVSTON
DEVINT  JMP     DEVPIC          /PIC ENTRY
        DAC     DEVAC           /API ENTRY, SAVE AC
        LAC     DEVINT          /SAVE PC, LINK,
        DAC     DEVOUT          /MEM. PROT.
        IORS                    /CHECK STATUS OF PIC
```

Part 4 (cont)

```
         SMA!CLA                      /FOR RESTORATION AT
         LAW      17740               /DISMISSAL
         TAD      DEVION
DVSTON   DAC      DVSWCH
         DEVCF                        /CLEAR FLAG
DEVION   ION                          /ENABLE PIC
```

Part 5

```
/THIS IS THE AREA DEVOTED TO PROCESSING INTERRUPT AND
/PERFORMING ANY ADDITIONAL I/O DESIRED.

         IOF                          /DISABLE PIC TO INSURE
         DEVIOT                       /DISMISSAL BEFORE
                                      /INTERRUPT FROM THIS
                                      /IOT OCCURS
```

Part 6

```
/INTERRUPT HANDLER DISMISS RTE
DVDISM   LAC      (JMP DEVPIC)        /RESTORE PIC ENTRY
         DAC      DEVINT
         LAC      DEVAC               /RESTORE AC
DVSWCH   ION                          /ION OR IOF
         DBR                          /DEBREAK AND RESTORE
         JMP*     DEVOUT              /LINK, MEM. PROT.
```

```
/IF THE HANDLER USES THE AUTO-INDEX
/OR
/EAE REGISTERS, THEIR CONTENTS
/SHOULD BE
/SAVED AND RESTORED.
/FUNCTIONS POSSIBLY IGNORED SHOULD
/CONTAIN PROPER INDEXING TO BYPASS
/ARGUMENT STRING.
```

Part 7

```
DVIGN2   ISZ      DVARGP              /BYPASS FILE POINTER
         JMP      DVCK
```

4.2.3 **Example B, Special I/O Handler for Hypothetical A/D Converter**

```
/ADC IOT'S
ADSF=701301
ADSC=701304
ADRB=701312
/ADSF=SKIP IF CONVERTER FLAG IS SET
/ADSC=SELECT AND CONVERT(ADC FLAG IS LEARED
/AND A CONVERSION IS INITIALITED)
/ADRB=READ CONVERTER BUFFER(PUTS CONTENTS IN AC)
/CAL ENTRY ROUTINE
         .GLOBL ADC.                  /ADC. IS GLOBAL NAME FOR HANDLER
.MED=3                                /MED(MONITOR ERROR DIAGNOSTIC)
ADC.     DAC ADCALP                   /SAVE CAL POINTER
         DAC ADARGP                   /AND ARGUMENT POINTER
         ISZ ADARGP                   /POINTS TO FUNCTION CODE
         LAC* ADARGP                  /GET CODE
         ISZ ADARGP                   /POINTS TO CAL + 2
         TAD (JMP DSPCH)
         DAC DSPCH                    /DISPATCH WITH
```

```
DSPCH   XX                          /MODIFIED JUMP
        JMP ADINIT                  /1 = .INIT
        JMP ADIGN2                  /2 = .FSTAT, .DELET, .RENAM
        JMP ADIGN2                  /3 = .SEEK
        JMP ADFRR6                  /4 = .ENTER
        JMP ADFRR6                  /5 = .CLEAR
        JMP ADOK                    /6 = .CLOSE
        JMP ADOK                    /7 = .MTAPE
        JMP ADREAD                  /10 = .READ
        JMP ADERR6                  /11 = .WRITE
        JMP ADWAIT                  /12 = .WAIT
        JMP ADERR6                  /13 = .TRAN
/ILLEGAL FUNCTIONS IN ABOVE TABLE CODED AS:
/       JMP ADERR6
/FUNCTION CODE ERROR
ADERR6  LAW 6                       /ERROR CODE 6
        JMP* (.MED+1)               /TO MONITOR
/DATA MODE ERROR
ADERR7  LAW 7                       /ERROR CODE 7
        JMP* (.MED+1)               /TO MONITOR


/THE ADINT ROUTINE MUST INCLUDE A
/.SETUP FOR
/EACH FLAG ASSOCIATED WITH THE
/DEVICE
/THE .SETUP CALLING SEQUENCE IS:
ADINIT  ISZ ADARGP                  /IDX TO RET STNDRD BUFF SIZE
        .DEC
        LAC (256                    /STANDARD BUFFER SIZE (DECIMAL)
        .OCT
        DAC* ADARGP                 /PUT BACK STANDARD BUFFER SIZE
        ISZ ADARGP
        CAL 57                      /57=API CHANNEL REGISTER
                                    /(40-77): Ø IF NOT
                                    /CONNECTED TO API
ADCKSM  16                          /.SETUP IOPS FUNCTION CODE
ADCBP   ADSF                        /ADC SKIP IOT TEST THE FLAG
ADLBHP  ADCINT                      /ADDRESS OF INTERRUPT
                                    /HANDLER (API OR PIC)
ADUND   LAC .+2                     /ERASES CALL FUNCTION IN CASE
ADWRC   DAC .-5                     /OF FURTHER .INIT'S THEY WILL BE
ADWPCT  JMP ADSTOP                  /IGNORED BY THIS JMP TO ADSTOP
                                    /WHERE THE I/O UNDERWAY SWITCH
                                    /IS CLEARED AND ALL I/O IS
                                    /TERMINATED
/THE PREVIOUS SIX TAGS IN THE CAL AREA ARE USED FOR TEMP
/STORAGE DURING THE ACTUAL .READ FUNCTION
/ADCKSM IS FOR STORING THE CHECKSUM
/ADDRP IS THE CURRENT BUFFER POINTER
/ADLRHP IS THE LINE BUFFER HEADER POINTER
/ADUND IS FOR DEVICE UNDERWAY SWITCH
/ADRWC IS USED AS A -WORD COUNT REGISTER
/ADRWCT IS USED TO STORE CURRENT WORD PAIR COUNT
/STOP ADC ROUTINE CLEARS I/O UNDER WAY SWITCH
ADSTOP DZM ADUND
/ADC WAIT LAC ADUND
        SNA
        JMP ADOK
```

```
/I/O UNDERWAY LOOP
ADBUSY  DBR
        JMP* ADCALP

ADREAD  LAC ADUND               /CHECK TO SEE IF I/O IS UNDERWAY
        SZA!CMA                 /IF NOT SET IT WITH -1
        JMP ADBUSY              /IT WAS SET, GO BACK TO CAL
        DAC ADUND               /SET IT
        LAC* ADARGP             /GET LINE BUFFER HEADER POINTER
        DAC ADDBP               /STORE IT
        DAC ADLBHP              /ALSO STORE IT FOR LATER HEADER
        ISZ ADARGP              /INCREMENT ARG. POINTER
        LAC* ADRAGP             /GET -L.B.W.C.(2'S COMP)
        DAC ADRWC               /STORE IT IN WORD COUNT REGISTER
        ISZ ADARGP              /INCREMENT FOR EXIT FROM .READ
        DZM ADWPCT              /ZERO WORD PAIR COUNT REG.
        DZM ADCKSM              /ZERO CHECKSUM REG.
        ISZ ADDBP               /GET PAST HEADER PAIR
        ISZ ADDBP               /NOW POINTING AT BEGINNING OF
                                /BUFFER
        ADSC                    /START UP DEVICE
/NORMAL RETURN FROM CAL
ADOK    DBR                     /BREAK FROM LEVEL 4
        JMP* ADARGP             /RETURN AFTER CAL


/INTERRUPT HANDLER FOR API OR PIC
ADCPIC  DAC ADCAC               /SAVE AC
        LAC* (0)                /SAVE PC, LINK,
        DAC ADCOUT              /MEM. PROT.
        LAC ADCION              /FORCE ION AT DISMISSAL
        JMP ADSTON
ADCINT  JMP ADCPIC              /PIC ENTRY.
        DAC ADCAC               /API ENTRY, SAVE AC
        LAC ADCINT              /SAVE PC, LINK,
        DAC ADCOUT              /MEM. PROT.
        LAC (JMP ADCPIC)        /RESTORE PIC ENTRY BECAUSE PIC
        DAC ADCINT              /A JMS CALL NOT A JUMP
        IORS                    /CHECK STATUS OF PIC
        SMA!CLA                 /FOR RESTORATION AT
        LAW 17740               /DISMISSAL
        TAD ADCION
ADSTON  DAC ADSWCH
        ADRB                    /READ CONVERTER BUFFER
ADCION  ION                     /ENABLE PIC FOR OTHER DEVICES
        DAC* ADDBP              /STORE DATA IN USER BUFFER
        ISZ ADDBP               /INC. BUFFER POINTER
        ISZ ADWPCT              /INC. WORD PAIR COUNTER
        TAD ADCKSM              /ADD CHECKSUM
        DAC ADCKSM              /STORE IT
        ISZ ADRWC               /IS I/O COMPLETE
        JMP ADCONT              /NO KEEP GOING
        LAC ADWPCT              /YES, COMPUTE WORD COUNT PAIR
        TAD (1                  /ADD ONE MAY BE ODD
        RTL                     /DIVIDE BY TWO, AT SAME TIME

        RTL                     /ADJUST IT, FOR HEADER
        RTL
        RTL
```

```
          AND (377ØØØ                /ALL SET
          DAC* ADLBHP                /STORE IN HEADER #1
          ISZ ADLBHP                 /INC. TO STORE CKSUM
          TAD ADCKSM                 /ADD WORD PAIR COUNT
          DAC* ADLBHP                /STORE IN HEADER #2
          DZM ADUND                  /CLEAR DEVICE UNDERWAY
          JMP ADDISM                 /EXIT
ADCONT    IOF                        /DISABLE PIC TO INSURE
          ADSC                       /DISMISSAL BEFORE

                                     /INTERRUPT FROM THIS
                                     /IOT OCCURS
/INTERRUPT HANDLER DISMISS RTE
ADDISM    LAC ADCAC                  /RESTORE AC
ADSWCH    ION                        /ION OR IOF
          DBR                        /DEBREAK AND RESTORE
          JMP* ADCOUT                /LINK, MEM. PROT.
ADCAIP    Ø                          /ADC CAL POINTER
ADARGP    Ø                          /ADC ARGUEMENT POINTER
ADCOUT    Ø                          /PC,L,MP
ADCAC     Ø                          /AC SAVED HERE

/IF THE HANDLER USES THE AUTO-INDEX
/OR
/EAE REGISTERS, THEIR CONTENTS
/SHOULD BE
/SAVED AND RESTORED.


/FUNCTIONS POSSIBLY IGNORED SHOULD
/CONTAIN PROPER INDEXING TO BYPASS
/ARGUMENT STRING.
ADIGN2    ISZ ADARGP                 /BYPASS FILE POINTER
          JMP ADOK
          .END
```

4.2.4 Incorporating User-Generated I/O Handlers

4.2.4.1 The following example shows how to incorporate a user generated I/O handler for a special or non-standard device (e.g., Type 350 incremental plotter) into the standard I/O Library. This means that the Linking Loader, DDT, or DDTNP must be modified to recognize the new handler. The changes to be made include .DAT Slot modification and skip chain modification. If the user wishes to use his own handler name, instead of one of the standard handler names (e.g., PLA., instead of LPA.), a radix 50 value in the Loader I/O Configuration (IOC) Table must be changed.

4.2.4.2 **Procedure:**

a. Load the Linking Loader as follows:

(1) Set the ADDRESS Switches to 17720 of the highest core bank.

(2) Place the Linking Loader tape in the reader.

(3) Press RESET then READIN.

b. Stop the computer at the point where the Loader types ↑S and modify the appropriate .DAT Slot (Slot 0 begins at location 135) according to the Loader IOC Table (Table 4-2). Suppose .DAT Slot 4 is chosen and the user wishes to substitute his plotter handler for handler LPA.. Cell 141 (.DAT Slot 4) should be changed from 000001 (Loader code for TTA.) to 000007 (Loader code for LPA.).

c. Modify the Monitor Skip Chain (Table 4-3) as follows:

(1) Determine the proper Skip IOT instruction code (octal) assigned to the device, either from the User's Guide DEC-15-H2DA-D (in the case of standard DEC I/O devices) or from engineering documentation supplied to the customer (for customer specified devices).

(2) Load the new IOT in the desired Skip Chain location. In case of the plotter mentioned above, the Skip IOT (from the User's Guide) is 702401 (PLSF). The skip associated with LPA. is at location 1533 as shown in Table 4-3. Location 1533 should be changed from 706501 (LSDF) to 702401 (PLSF).

If the user has named his plotter handler LPA., he may proceed to step E below.

d. If the user prefers a different name* (e.g., PLA.), the radix 50 value associated with the LPA handler in the IOC Table (Table 4-2) must be changed. Compute the new radix 50 value as follows:

(1) Find P in column 1 of Table 4-4 (062000).

(2) Find L in column 2 (000740).

(3) Find A in column 3 (000001).

(4) The resultant radix 50 value is 062741.

(5) The fourth character of the mnemonic PLA. is the period (.).

This is signified to the Loader by making the sign bit of the computer word (bit 0) a 1. The resulting radix 50 value is now 462741.

*The new name must consist of from one to three alphabetic characters followed by a period.

(6) Hence the contents of location 4505 (Loader) or 5264 (DDT) should be changed to 462741.

e. Load PUNCH-15 as follows:

(1) Set the ADDRESS Switches to 17720 of the highest core bank.

(2) Place the PUNCH-15 tape in the reader.

(3) Press RESET then READIN.

f. When the PUNCH-15 query "↑P, T OR S?" appears on the teletype, type CTRL P.

g. When PUNCH-15 replies with a >, type the number of output tapes desired (in this case 1 – up to 9 if desired.)

h. To load the resultant tape(s), proceed as in Step a. The computer stops after loading each tape except the last. Press CONTINUE when the next tape is in the reader. Program execution begins automatically when the end of the last tape is reached.

i. Assemble the new handler in relocatable binary form. Remove the end-of-file (EOF) block using PIP-15 with the "W" switch option. Attach this tape to the end of the first tape in the I/O Library.

Table 4-2. Linking Loader IOC Table

Handler	Code Slot	Location		Radix 50 Value
		Loader/DDTNP	DDT	
TTA.	1	4477	5256	500041
PRA.	2	4500	5257	463321
PRB.	3	4501	5260	463322
PPA.	4	4502	5261	463201
PPB.	5	4503	5262	463202
PPC.	6	4504	5263	463203
LPA.	7	4505	5264	446601
CDA.	10	4506	5265	411541
CDB.	11	4507	5266	411542
CDC.	12	4510	5267	411543
MTF.	13	4511	5270	452146
DRA.	14	4512	5271	415721
DRB.	15	4513	5272	415722
DRC.	16	4514	5273	415723
DRD.	17	4515	5274	415724

†The IOC Table Locations shown above are subject to change in later versions of DDT and the Linking Loader. For information regarding these changes, see the Reader Service Card at the end of this manual.

Table 4-3. Basic I/O Monitor Skip Chain*

Location†	Contents	Mnemonic	Device Meaning
1521	703201	SPFAL	POWER FAIL
1522	741000	SKP	
1523	62157$\overset{2}{7}$	JMP* INT6	
1524	707341	MTSF	MAGTAPE DONE
1525	741000	SKP	
1526	621604	JMP* INT13	
1527	700001	CLSF	CLOCK OVERFLOW
1530	741000	SKP	
1531	600476	JMP CLKPIC	
1532	706701	RCSF	CARD COLUMN READY
1533	741000	SKP	
1534	621575	JMP* INT4	
1535	706721	RCSD	CARD DONE
1536	741000	SKP	
1537	621576	JMP* INT5	
1540	706501	LSDF	LINE PRINTER DONE
1541	741000	SKP	
1542	621572	JMP* INT1	
1543	700101	RSF	PAPER TAPE READER DONE
1544	741000	SKP	
1545	621573	JMP* INT2	
1546	700201	PSF	PAPER TAPE PUNCH DONE
1547	741000	SKP	
1550	621574	JMP* INT3	
1551	700301	KSF	KEYBOARD READY
1552	741000	SKP	
1553	601110	JMP TIINT	
1554	700401	TSF	TELEPRINTER DONE
1555	741000	SKP	
1556	601344	JMP TOINT	
1557	701741	MPSNE	NON-EXISTENT MEMORY
1560	741000	SKP	
1561	621600	JMP* INT7	
1562	701701	MPSK	MEMORY PROTECT VIOLATION
1563	741000	SKP	
1564	621601	JMP* INT10	
1565	702701	SPE	MEMORY PARITY ERROR
1566	741000	SKP	
1567	621602	JMP* INT11	

*This skip chain memory map applies only when the Linking Loader, DDT, DDTNP, CHAIN, EXECUTE or relocatable user programs are in core.

†The skip chain locations shown above are subject to change in later versions of the associated software. For information regarding these changes, see the Reader Service Care at the end of this manual.

Table 4-4. Radix 50_8 Values

X - -		-X-		--X	
A	003100	A	000050	A	000001
B	006200	B	000120	B	000002
C	011300	C	000170	C	000003
D	014400	D	000240	D	000004
E	017500	E	000310	E	000005
F	022600	F	000360	F	000006
G	025700	G	000430	G	000007
H	031000	H	000500	H	000010
I	034100	I	000550	I	000011
J	037200	J	000620	J	000012
K	042300	K	000670	K	000013
L	045400	L	000740	L	000014
M	050500	M	001010	M	000015
N	053600	N	001060	N	000016
O	056700	O	001130	O	000017
P	062000	P	001200	P	000020
Q	065100	Q	001250	Q	000021
R	070200	R	001320	R	000022
S	073300	S	001370	S	000023
T	076400	T	001440	T	000024
U	101500	U	001510	U	000025
V	104600	V	001560	V	000026
W	107700	W	001630	W	000027
X	113000	X	001700	X	000030
Y	116100	Y	001750	Y	000031
Z	121200	Z	002020	Z	000032
%	124300	%	002070	%	000033
	127400		002140		000034
0	132500	0	002210	0	000035
1	135600	1	002260	1	000036
2	140700	2	002330	2	000037
3	144000	3	002400	3	000040
4	147100	4	002450	4	000041
5	152200	5	002520	5	000042
6	155300	6	002570	6	000043
7	160400	7	002640	7	000044
8	163500	8	002710	8	000045
9	166600	9	002760	9	000046
#	171700	#	003030	#	000047

4.3 I/O HANDLERS ACCEPTABLE TO SYSTEM PROGRAMS

This section lists the .DAT slot requirements of system programs, the uses made of the .DAT slots, and the I/O handlers that may be assigned to each. It is imperative that one and only one I/O handler for a device be in core at the same time; that is, PRA and PRB should not be brought in together since there is no communication between the two interrupt handlers.

NOTE

All system programs use .DAT slots -2 and -3 for teletype input and output, respectively. These assignments cannot be changed.

4.3.1 FORTRAN IV

.DAT Slot	Use	Handler
-11	Input	TTA PRA PRB* MTF (non-file oriented) CDB
-12	Listing	TTA* LPA PPA MTF (non-file oriented)
-13	Output	PPA PPB* PPC MTF (non-file oriented)

4.3.2 MACRO-15

MACRO-15 is identical to FORTRAN IV, with two exceptions: a If .ABS binary output is requested on .DAT slot -13, PPC. cannot be used. b When the P option (in the command string) is used, .DAT slot -10 is the secondary input device:

TTA*
PRA
PRB
CDB

4.3.3 FOCAL

.DAT Slot	Use	Handler
+3	Input	PRA* PRB CDB

*DEC Standard assignment.

.DAT Slot	Use	Handler
+5	Output	PPA* LPA

4.3.4 Editor

.DAT Slot	Use	Handler
-10	Secondary Input	TTA PRA* PRB CDB
-14	Input	TTA PRA* PRB CDB
-15	Output	LPA TTA PPA*

4.3.5 Linking Loader and DDT

.DAT Slot	Use	Handler
-1	System Library	PRA*
-4	Input	PRA*
-5	External User Library	NONE* PRA

4.3.6 PIP

.DAT Slot	Use	Handler
1	I/O	TTA*
2	I/O	TTA*
3	Input	PRA*
4	Output	LPA*
5	Output	PPA*
6	Input	CDB*
7	Output	PPA*
10	Input	PRA*

*DEC Standard assignment

4.3.7 Chain Builder

.DAT Slot	Use	Handler
-1	System Library	PRA*
-4	User Programs	PRA*
-5	External Library	NONE*
-6	Output	PPA* PPB PPC

4.3.8 Chain Execute

.DAT Slot	Use	Handler
-4	Chain Input	PRA*

4.3.9 8TRAN

.DAT Slot	Use	Handler
-14	Output	PPA*
-15	Input	PRA* CDB

4.4 SUMMARY OF STANDARD I/O HANDLER FEATURES

4.4.1 TTA·(Teletype)

a. *Functions* – All function descriptions (except READ and WRITE) refer to action taken when either the teleprinter or the keyboard is addressed.

Mnemonic **Code** **Action**

.INIT 1

(1) Return standard buffer size (34_{10}).

(2) Assign return addresses for certain control characters from contents of CAL ADDRESS +2. Bits 0 through 1 in CAL + 2 address are set to designate caller:

B0, 1 = 01	caller = Monitor
B0, 1 = 10	caller = DDT
B0, 1 = 00	caller = any other user

(3) Set I/O UNDERWAY indicator.

(4) Print carriage return and line feed (CR/LF).

*DEC Standard Assignment

Mnemonic	Code	Action
.DLETE	2	Ignored
.RENAM	2	Ignored
.FSTAT	2	Ignored
.SEEK	3	Ignored
.ENTER	4	Ignored
.CLEAR	5	Ignored
.CLOSE	6	(1) Set I/O underway indicator. (2) Print CR/LF. (3) Wait on .CLOSE for completion of I/O.
.MTAPE	7	Ignored
.READ	10	(1) Set I/O underway indicator. (2) Set up to accept characters from keyboard.
.WRITE	11	(1) Set I/O Underway indicator. (2) Print line.
.WAIT	12	Allow input at output to finish.
.TRAN	13	IOPS06-Illegal function.

b. *Legal Data Modes*

IOPS ASCII (Mode 2)
IMAGE ALPHANUMERIC (Mode 3)

c. *Vertical Carriage Control Characters*

Output – (1) Line feed (12_8)
IOPS ASCII: Ignore all leading line feeds; otherwise output
IMAGE: Output
(2) Others (vertical tab, form feed) output

Input – Inserted in buffer

d. *Horizontal Carriage Control Characters*

Tab (11_8) in or out

IOPS ASCII
(1) Model 33 - output sufficient number of spaces to place the next typed character in column 11, . . ., 71. Insert only 11_8 in buffer on input.
(2) Model 35 - input, insert 11_8 in buffer. Output, print tab.

Image Alphanumeric
Input, insert 11_8 in buffer. Output, print tab.

e. *Program Control Characters-IN*

Stop current I/O to Teletype.

Decode character and echo on Teleprinter.*
(1) CTRL C transfers control to the address specified as return in the .INIT performed by the Monitor.

*Character will be ignored (not echoed) in cases (1), (2), and (3), if respective. .INIT has not been performed.

(2) CTRL P transfers control to the address specified as return in the .INIT performed by the user (other than the Monitor or DDT).

(3) CTRL T transfers control to the address specified as return in the .INIT performed by DDT.

(4) CTRL S transfers control to the address specified in .SCOM + 6.

f. *Data Control Characters-IN*

Image Alphanumeric — All characters inserted in buffer as 7-bit characters.

IOPS ASCII

(1) Rubout. Delete previous character typed. Type out reverse slash (\).

(2) CTRL U. Delete entire line typed so far. Type out commercial at (@). If output is UNDERWAY, printing is terminated and a CR/LF is output.

g. *Data Control Characters-OUT* (both modes)

Ignore RUBOUT (177_8) and NULL (00).

In Image Alpha mode, a RUBOUT should be used to fill the last word pair when an odd number of characters is to be output.

h. *Errors (no program-initiated recovery)*

IOPS6 – Illegal Function
IOPS7 – Illegal Data Mode

i. *Program Stze*

469_{10} registers (this is included in resident MONITOR).

j. *Teletype I/O* – Can be requested only from mainstream in API systems, since the Teletype is not connected to the API.

4.4.2 **PP (Paper Tape Punch)**

a. *Functions*

Mnemonic	Code		Action
.INIT	1	(1)	Return standard buffer size (52_{10}).
		(2)	.SETUP - no API.
		(3)	Punch two fanfolds of leader.

Mnemonic	Code	Action
.DLETE	2	Ignored
.RENAM	2	Ignored
.FSTAT	2	Ignored
.SEEK	3	IOPS6 – Illegal function
.ENTER	4	Ignored
.CLEAR	5	Ignored
.CLOSE	6	(1) Allow previous output to terminate.
		(2) Punch EOF if IOPS Binary
		(3) Punch two fanfolds of trailer
		(4) Allow trailer punching to terminate
.MTAPE	7	Ignored
.READ	10	IOPS6 – Illegal function
.WRITE	11	(1) Allow previous output to terminate.
		(2) Output buffer
.WAIT	12	Allow previous output to terminate.
.TRAN	13	IOPS6 – Illegal function

b. *Legal Data Modes*

IOPS Binary	(mode 0)
IMAGE Binary	(mode 1)
IOPS ASCII	(mode 2)
IMAGE ALPHANUMERIC	(mode 3)
Dump	(mode 4)

c. *Vertical Control Characters (IOPS ASCII only)*

May appear only as first character of line, if elsewhere in line will be ignored; if no vertical control character at beginning of line, a line feed (012) will be used.

012	Line feed
013	Vertical tab, followed by four deletes (177)
014	Form feed, followed by 40_8 nulls (000)

d. *Horizontal Control Characters (IOPS ASCII only)*

011	Horizontal tab, followed by one delete (177)

e. *Recoverable Errors*

No tape in punch – Monitor error IOPS4

(1) Put tape in punch

(2) Type CTRL R

f. *Unrecoverable Errors*

IOPS6 – Illegal Function:

(1) .SEEK

(2) .READ

(3) .TRAN

IOPS7 – Illegal Data Mode

g. *Program Size*

PPA. (all data modes)	397 decimal registers
PPB. (all except IOPS ASCII)	270 decimal registers
PPC. (IOPS binary only)	210 decimal registers

NOTE

In API systems, the paper tape punch can be called only from mainstream, since the punch is not connected to the API.

4.4.3 PR (Paper Tape Reader)

a. *Functions*

Mnemonic	Code	Action
.INIT	1	(1) Return standard line buffer size (52_{10}) (2) .SETUP API channel register 50_8

Mnemonic	Code	Action
		(3) Clear I/O UNDERWAY indicator
.DLETE	2	Ignored
.RENAM	2	Ignored
.FSTAT	2	Ignored
.SEEK	3	Ignored
.ENTER	4	IOPS6 – Illegal function
.CLEAR	5	IOPS6 – Illegal function
.CLOSE	6	Allow previous input to finish and then clear I/O UNDERWAY indicator.
.MTAPE	7	Ignored
.READ	10	(1) Allow previous input to be completed.
		(2) Input line or block of data (see modes below).
.WRITE	11	IOPS6 – Illegal function
.WAIT	12	Allow previous input to be completed before allowing user program to continue.
.TRAN	13	IOPS6 – Illegal function

b. *Legal Data Modes*

IOPS ASCII
(mode 2)

(1) Constructs line buffer header, computing:

Word pair count

Data mode

Data validity bits

(2) Packs characters into the line buffer in 5/7 ASCII, checking parity (eighth bit, even) on each character.

(3) Allows vertical form control characters. (FF, LF, VT) only in character position 1 of the line buffer. Otherwise, ignored.

(4) Terminates reading on CR or line buffer overflow. In the latter case, tape is moved past the next CR to be encountered.

IOPS Binary
(Mode 0)

(1) Reads binary data in alphanumeric mode, checking parity (seventh hold, odd) on each frame.

(2) Accepts line buffer header at head of input data, modifying data validity bits if parity or checksum errors (or short line) have occurred.

(3) Terminates reading on overflow of word pair count in line buffer header or word count in .READ macro, whichever is smaller, moving tape to end of line or block if necessary.

Image
Alphanumeric
(Mode 3)

(1) Constructs line buffer header, computing:

Word pair count

Data mode

(2) Stores characters, without editing, or parity checking in the line buffer, one per register.

(3) Terminates reading as a function of .READ macro word count.

Image Binary
(Mode 1)

Same as Image Alphanumeric, except a binary read is issued to the PTR.

Dump
(Mode 4)

Same as Image Alphanumeric except, no header is constructed; loading begins at the core address specified in the .READ macro. A binary read is issued to the PTR.

NOTE

An end of tape condition causes the PTR interrupt service routine to terminate the input line, turning off the I/O UNDERWAY program indicator and marking the header (data mode bits) as an EOM (end of medium) for all modes except Dump.

c. *Unrecoverable Errors*

Illegal Function (IOPS6)

(1) .ENTER

(2) .CLEAR

(3) .WRITE

(4) .TRAN

Illegal Data Mode (IOPS7)

d. *Program Size*

PRA. (all data modes)	436 decimal registers
PRB. (IOPS ASCII only)	287 decimal registers

4.4.4 CDB. (CARD READER CR03B)

NOTE

When the CR03B is used as an EDITOR input, a blank card must be placed immediately after the End of File (EOF) card (card column 1 punched 12-11-0-1-2-3-4-5-6-7-8-9). If an EOF card is not used to end a deck, the blank card is not required.

a. *Functions*

Mnemonic	Code	Action
.INIT	1	(1) Return standard buffer size (36_{10}) (2) Call .SETUP to update skipchain with PIC servicer addresses for column ready and card done flags and to place API servicer address in location 55_8 (API channel 13).
.DLETE	2	Ignored
.RENAM	2	Ignored
.FSTAT	2	Ignored
.SEEK	3	Ignored
.ENTER	4	IOPS6 – Illegal function
.CLEAR	5	IOPS6 – Illegal function
.CLOSE	6	Allow previously requested input to terminate
.MTAPE	7	Ignored
.READ	10	(1) Allow previously requested input to terminate. (2) Ensure that device is ready. (3) Initiate input of next card.
.WRITE	11	IOPS6 – Illegal function

Mnemonic	Code	Action
.WAIT	12	Allow previously requested input to terminate.
.TRAN	13	IOPS6 – Illegal function

b. *Legal Data Modes*

IOPS ASCII (mode 2) Eighty card columns are read and interpreted as Hollerith (029 or 026) data, mapped into the corresponding 64-graphic subset of ASCII, and stored in the user's line buffers in 5/7 format (36_{10} locations required to store an 80 column card). Compression of internal blanks to tabs and truncation of trailing blanks is not performed; all 80 characters appearing on the card are delivered to the caller's line buffer. In addition, a carriage return (015_8) character is appended to the input line. A total of 81 ASCII characters are thus returned by the handler in IOPS ASCII mode.

All illegal punch configurations (i.e., those not appearing in the 029 or 026) cause an IOPS4 error. In response to the error, the user can:

(a) correct the card, reinsert it into the reader input hoppers, and type CTRL R to restart the read operation;

(b) attempt to restart read (CTRL R) without correcting the card;

(c) remove the card from the read hopper and restart read (CTRL R).

The single addition to the Hollerith set, one made necessary by the constraints of system programs, is the provision for the internal generation of the ALT MODE terminator. The appearance of a 12-1-8 punch (multiple-punched A/8) on the card is mapped into the standard PDP-15 ALT MODE character (175_8) in the user's line.

When card processing is complete, word 1 of the header is constructed and stored in the caller's line buffer area. Word 2 of the header, the checksum location, is never disturbed by the card reader handler in IOPS ASCII mode.

Refer to Appendix B for a listing of legal Hollerith codes and their corresponding ASCII graphics.

c. *Recoverable Errors*

IOPS4 – Reader Not Ready

(1) Hopper Empty

(2) Stacker Full

(3) Feed check, LIGHT CHECK (may be hardware failure)

(4) Read Check, DARK FAIL (may be hardware failure)

(5) Reader not ready:

(a) Stop button depressed.

(b) Start button *not* depressed.

(6) Validity check.

(7) Pick Fail card selected but not passed from hopper to read stations.

d. *Unrecoverable Errors*

IOPS6 – Illegal Function

(1) .ENTER

(2) .CLEAR

(3) .WRITE

(4) .TRAN

e. *Program Size*

CDB. occupies 410_{10} locations.

f. *Assembly Procedures*

CDB. exists in the I/O Library in a version which recognizes DEC 029 code (Appendix B). It also is supplied to the user as a source tape to allow an alternate assembly of the handler which will recognize DEC 026 code.

The following is the procedure for assembling CDB. in 026 mode:

(1) Prepare to assemble the CDB. source tape using MACRO-15 with the "P" option.*

(2) Enter the following parameter on the Teletype:

DEC026 = 1 (CTRL D)

(3) After assembly, remove the end-of-file (EOF) block from the new CDB. tape by using PIP with the "W" switch option.

(4) This tape becomes the first tape in the I/O Library.

4.4.5 LPA·(647 LINE PRINTER)

a. *Function*

Mnemonic	Code		Action
.INIT	1	(1)	Return standard line buffer size (52_{10})
		(2)	.SETUP — API channel register 56_8
		(3)	Clear line printer buffer
		(4)	Form feed

*Assembly without the "P" option produces the 029 version of the handler.

Mnemonic	Code		Action
.DLETE	2		Ignored
.RENAM	2		Ignored
.FSTAT	2		Ignored
.SEEK	3		IOPS6 – Illegal function
.ENTER	4		Ignored
.CLEAR	5		Ignored
.CLOSE	6	(1)	Allow previous output to terminate
		(2)	Form feed
		(3)	Allow form feed to terminate
.MTAPE	7		Ignored
.READ	10		IOPS6 – Illegal function
.WRITE	11	(1)	Allow previous output to terminate
		(2)	Output line
.WAIT	12		Allow previous output to terminate
.TRAN	13		IOPS6 – Illegal function

b. *Legal Data Modes*

IOPS ASCII
(mode 2)

c. *Vertical Control Characters (when first character of line)*

12	Print every line
21	Print every second line
22	Print every third line
13	Print every sixth line
23	Print every tenth line
24	Print every twentieth line

20	Overprint
14	Form feed

d. *Horizontal Control Characters (anywhere in line)*

11	Horizontal tab – converted to N spaces, where N is the number necessary to have the next character in column 11, 21, 31, 41, ...

e. *Recoverable Errors*

Device not ready	Monitor error message, .IOPS4. Make device ready, then type CTRL R to continue.

f. *Unrecoverable Errors*

Illegal function		Monitor error message, .IOPS6 XXXXXX, where XXXXXX is address of error CAL.
	(1)	.SEEK
	(2)	.READ
	(3)	.TRAN
Illegal data mode		Monitor error message, .IOPS7 XXXXXX, where XXXXXX is address of error CAL. Any mode other than IOPS ASCII.

g. *Program Size*

297 (decimal) registers.

4.4.6 MTF. (TU20, TU20A, TU30, TU30A Tape Transports)

a. *Introduction*

MTF. is a general magnetic tape I/O handler intended principally to meet the immediate needs of PDP-15 FORTRAN users. It will accept data-transfer requests in data modes used by the FORTRAN IV Object Time System and in addition, will honor the following FORTRAN statements:

REWIND u

ENDFILE u

BACKSPACE u

The magnetic tape facility is also available, via MTF., to MACRO-15 users. It should be noted, however, that legal data modes and functions are limited to those required in the FORTRAN environment and the special 9-channel dump mode.

b. *Functions*

Mnemonic	**Code**	**Action**
.INIT	1	(1) Return standard buffer size ($56_{10} = 70_8$). (2) Call .SETUP for API channel 45, if first time through. (3) Set transfer direction for drive referenced. Direction is specified by bit 8 of CAL instruction: Bit 8 = 1, this drive is output; Bit 8 = 0, this drive is input. (4) Set default conditions for this drive, viz: (a) This drive will transfer in odd parity. (b) This drive will transfer at 800 BPI. (c) This drive is 7-channel.*
.OPER	2	Ignored
.FSTAT	2	Ignored
.DLETE	2	Ignored
.RENAM	2	Ignored
.SEEK	3	(1) If the specified drive has been INITed and transfer direction is input, return to caller. (2) If the specified drive has been INITed and transfer direction is *not* input, IOPS6. (3) If the specified drive has *not* been INITed and transfer direction is input, perform .INIT for this drive (input).
.ENTER	4	(1) If the specified drive has been INITed and transfer direction is output return to caller.

*Unless the assembly parameter had been used to change the default assumption to 9-channel. Variable MT9CHN need only be defined for 9-channel paper-tape systems.

b. *Functions (cont)*

Mnemonic	Code		Action
		(2)	If the specified drive has been INITed and transfer direction is *not* output, IOPS6.
		(3)	If the specified drive has *not* been INITed and transfer direction is output, perform .INIT of this drive (output).
.CLEAR	5		Ignored
.CLOSE	6	(1)	Wait for current I/O to complete.
		(2)	Is this drive an input drive? Yes: return to caller. No: continue.
		(3)	Is this drive open for transfers? No: return to caller. Yes: continue.
		(4)	Write two EOF markers (7-channel = 17_8; 9-channel = 23_8), hang on CAL during transfers.
		(5)	Backspace one record, hang on CAL during backspace. This positions the read head between the two EOF markers written.
		(6)	Set switch indicating this drive is not open for transfers.
		(7)	Return to caller at CAL + 2
.MTAPE	7	(1)	Wait for current I/O to complete.
		(2)	Interpret subfunction (bits 5-8) of CAL instruction. Subfunctions: 0 – Rewind (a) Set indicator that I/O is *not* underway. (b) Issue rewind to this drive. 1 – Undefined IOPS6. 2 – Backspace Record

Mnemonic *Code* *Action*

(a) Set indicator that I/O *is* underway.

(b) Issue backspace-one-record to this drive. Note: the backspace is attempted at the density currently assigned to this drive (via .INIT or .MTAPE 10-17).

3 – Backspace File

Illegal: IOPS6

4 – Write EOF

5 – Space Forward Record

(a) Check I/O transfer direction for this drive.

i. Output: IOPS 6
ii. Input: continue.

(b) Set indicator that I/O *is* underway

(c) Issue space-forward-one-record to this drive.

6 – Space Forward File

Illegal: IOPS6

7 – Space to Logical EOT

Illegal: IOPS error, code 6.

10-17 – Describe Tape Configuration

Update format descriptor bits for this drive. Note that these bits will be reset by any subsequent .INIT to this drive.

I/O transfers (including space operations) will be performed in the parity, density, and channel-count specified in .MTAPE 10-17, thus:

10:7-channel, even,	200 BPI
11:7-channel, even,	556 BPI
12:7-channel, even,	800 BPI
14:7-channel, odd,	200 BPI
15:7-channel, odd,	556 BPI
16:7-channel, odd,	800 BPI
13:9-channel, even,	800 BPI
17:9-channel, odd,	800 BPI

.READ 10 (1) Wait for current I/O to complete.

Mnemonic **Code** **Action**

(2) Ensure that requested drive is available (IOPS4 if not).

(3) Parameters:

(a) Buffer address: from CAL sequence

(b) Word count (maximum): from CAL sequence

(c) Tape format: from previous .INIT or .MTAPE 10-17.

(4) Data Format: Data are transferred directly from magtape to the user's core buffer area. No editing of any kind is performed by MTF.

(5) Parity Errors: Hardware parity errors during an input transfer cause the proper bits to be set in word 0 of the line buffer header (bits 12-13 = 01 if an error occurred). No reread is attempted.

(6) End of File, End of Tape

(a) End of File: When an end-of-file marker (EOF) is encountered on the tape being read, an EOF pseudo-line is constructed and stored in the user's line buffer area.* This EOF line consists of a two-word header and one data word, thus:

header word 0:	002005
header word 1:	775773
data word 0:	000000
data word 1:	Unchanged

(b) Physical End of Tape: When an end-of-tape is encountered on the tape being read, an end-of-medium line is constructed and stored in the user's line buffer area.* The EOM line consists of a two-word header and one data word, thus:

header word 0:	002006
header word 1:	775772
data word 0:	000000
data word 1:	unchanged

An attempt is then made to backspace over the record being read. The assumption is made that the record was written through the end-of-tape.

.WRITE 11 (1) Wait for current I/O to complete.

(2) Ensure that requested drive is available (IOPS 4 if not).

*The action described takes place only if the I/O transfer had been performed in Mode 0 or Mode 2 (see Section C). For error activity in Mode 5 transfers, see Section C.3.

Mnemonic	Code	Action
		(3) Parameters: (a) Line buffer address: from CAL sequence. (b) Word count: from bits 1-8 of line buffer header.* (c) Tape format: from previous .INIT or .MTAPE 10-17. (4) Data Format: Data are transferred directly from the user's line buffer area to the device. No editing of any kind is performed by MTF. Each .WRITE results in exactly one physical record on tape. (5) Write errors: 5 attempts are made to rewrite the record. Before each attempt, a 3-inch length of tape is erased. If the fifth try is unsuccessful, an IOPS 4 return is made to the monitor. The user then has the option of trying to write 5 more times (CTRL R) or restarting with a fresh tape. (6) End of Tape: When EOT is detected during writing, an error return (IOPS15) is made to the monitor.
.WAIT	12	(1) Wait for current I/O to complete. (2) Return to CAL + 2
.WAITR	13	Determine whether current I/O is complete. (1) If I/O is incomplete, transfer to location specified in CAL + 2. (2) If I/O is complete, return to CAL + 3.
.TRAN	14	The .TRAN request (for forward-direction Transfers only) is honored and results in a binary transfer from or to the next physical record on tape. The device-address agreement in the CAL sequence (CAL + 2) is ignored by the .TRAN processor.

c. *Legal Data Modes*

(1) IOPS Binary (Mode 0) - 7 or 9 channel transports

(2) IOPS ASCII (Mode 2) - 7 or 9 channel transports

(3) 9-Channel Dump (mode 5) - 9 channel transports only.

(a) This mode is designed to take advantage of all 8 data bits in each frame of 9-channel tape and thus ensure maximum throughput for

*In Modes 0 and 2 only (see Section C). For Mode 5 transfers, the word count is taken from the CAL sequence.

each word transferred. Whereas normal (Modes 0 and 2) transfers require three tape frames for each PDP-15 register, transfer in Mode 5 require only two frames per word. Only 16 bits of each PDP-15 word, however, are read from or written on tape. Word format for Mode 5 is described below:

Parity Bits (P_0, P_1): Bits 0 and 1 of the PDP-15 word are used as parity bits for the 2 8-bit data bytes (D_0 and D_1), respectively, in the low-order portion of the register. During an output (write) transfer, these bits are ignored. The hardware will generate the proper parity for each data frame. During an input (read) transfer, these bits are set to the actual parity values for the two data frames as read from tape.

Data Bytes (D_0, D_1): Bytes 2-9 and 10-17 hold the values of two adjacent 8-bit frames. During reads and writes, D_0 is the first frame transferred. If a record containing an odd number of frames is read, the final frame is stored in D_0; D_1 is set to binary zeroes. An even number of frames are always written during an output operation.

(b) Error Treatment in Mode 5

Because no line-buffer header is present in data transferred in Mode 5, there is presently no facility for indicating I/O errors to the user program, particularly during input transfers. MTF. will do its best to transfer correct data but, if it fails, cannot so inform the calling program.

Read Errors

Parity Errors: 5 attempts are made to reread the error record. If the record cannot be read successfully, the data resulting from the fifth reread attempt are left in the user's buffer area.

Buffer Overflow: If the number of words in the record being read exceeds the count given in the CAL sequence, transfer is halted.

End of ~~File~~ Tape: No data are transferred. The read head is positioned preceding the EOT reflective spot.

Write Errors

As described in paragraph e. under .WRITE.

(c) Restrictions

Mode 5 is a legal I/O mode only if the referenced transport is 9-channel. An attempt to use Mode 5 in reading or writing a

Mnemonic	Code	Action
		7-channel drive results in an error return (IOPS7) to the Monitor. Requests for transfers in modes other than 0, 2, and 5 result in error returns (IOPS7) to the Monitor.

d. *Program Size*

MTF. currently occupies 1144_8 PDP-15 registers.