PRINTED IN U.S.A. DEC-08-XIAA-D

digital introduction to programming

```
/TWO'S COMPLEMENT SINGLE PRECISION MULTIPLY ROUTINE
/RETURN HIGH ORDER PRODUCT IN AC, LOW IN MP1

MULT,   0
        CLL                     /CLEAR LINK
        SPA                     /TEST FOR NEGATIVE MULTIPLIER
        CMA CML IAC             /IF NEGATIVE, COMPLEMENT
                                /MULTIPLIER; SET LINK
        DCA MP1                 /STORE MULTIPLIER
        DCA MP5
        TAD I MULT              /LOAD MULTIPLICAND
        SNA                     /TEST FOR ZERO MULTIPLICAND
        JMP MPSN+2              /JMP IF MULTIPLICAND = 0
        SPA                     /TEST FOR NEGATIVE MULTIPLICAND
        CMA CML IAC             /IF NEGATIVE, COMPLEMENT
                                /MULTIPLICAND AND LINK
        DCA MP2                 /STORE MULTIPLICAND
        TAD THIR
        DCA MP3
        TAD MP1                 /MULTIPLY LOOP
MP,     RAR                     /GET NEXT BIT OF MULTIPLIER
        DCA MP1
        SZL                     /TEST MULTIPLIER
        JMP MP2                 /LOAD MULTIPLICAND
        TAD MP5                 /LOAD MOST SIGNIFICANT
                                /PRODUCT
        SZL                     /TEST IF MULTIPLICAND SHOULD
                                /BE ADDED
        TAD MP2
        CLL RAR                 /SHIFT ACCUMULATED RESULT
        DCA MP5
        TAD MP1
        RAR
MPSN,   SZL                     /SHOULD PRODUCT BE POSITIVE
        JMP COMP                /NO-COMPLEMENT
        DCA MP1                 /YES
        TAD MP5
MP2,    ISZ MULT                /EXIT TO CALLING PROGRAM
        JMP I MULT
COMP,   CMA CLL IAC             /COMPLEMENT PRODUCT
        DCA MP1
        TAD MP5
        CMA
        SZL
        IAC
        JMP MP2                 /RETURN
THIR,   7764                    /ELEVEN IN DECIMAL
MP1,    0
MP5,    0
MP2,    0
MP3,    0
PAUSE
```

# introduction to programming

digital pdp-8 handbook series

# contents

# introduction to programming

DEC-08-XINPA-A-D

digital

# introduction to programming

prepared
by
small systems software
documentation group
digital equipment corporation

pdp-8 handbook series

First Edition, January 1969
Second Printing, July 1969
Second Edition, September 1970
Third Edition, May 1972
Fourth Edition, September 1973
Fifth Edition, April 1975

The description and availability of the software products described in this manual are subject to change without notice. The availability or performance of some features of the software products may depend on a specific configuration of equipment. Consequently, DEC makes no claim and shall not be liable for the accuracy of the software products. Distribution of software products shall be in accordance with the then standard policy for each such software product.



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

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

# Foreword

As few as five years ago, the suggestion that a computer or computer-based system could be readily available to users at *all* levels of technical knowledge and ability still evoked surprise and concern among many. To help convey our position that programming a minicomputer was not a restricted undertaking, we introduced in January of 1969 our first major handbook dealing specifically with the fundamentals of machine and assembly language programming on a minicomputer—*Introduction to Programming.* Since that time, the demand for this handbook by users in every field and occupation, experienced programmer and novice alike, has clearly proven the value of such a book.

In addition to *Introduction to Programming,* we include several other volumes in our PDP-8 handbook series. The *Small Computer Handbook* provides extensive technical information concerning hardware options, interfacing, system operation and installation planning; this handbook is invaluable to those who will develop and maintain a minicomputer installation. The *EduSystem Handbook* contains a complete self-instruction course in the use of the BASIC programming language, plus user guides to each of the existing EduSystems—systems designed specifically for classroom use. Finally, the forthcoming *OS/8 Handbook* will present comprehensive information dealing with DEC's complete computer system for the PDP-8—OS/8. OS/8 provides the programmer with an extended library of system programs, including a text editor, octal debugging program, assemblers, loaders, and FORTRAN IV.

Once again, I wish to thank all programmers, writers, teachers and students who have contributed to our handbooks. Through your support we can continue producing extensive low-cost programming information for PDP-8 computers.

Kenneth H. Olsen
President,
Digital Equipment Corporation

## ERROR REPORTING

If you find any errors in this handbook, or if you have any questions or comments concerning the clarity or completeness of this handbook, please direct your remarks to:

Digital Equipment Corporation
Software Communications, Parker Street
Maynard, Massachusetts 01754

# introduction

*Introduction to Programming* is DEC's introductory computer programming handbook. The first five chapters provide a thorough explanation of computer mathematics and an introduction to machine language and assembly language programming and to the basics of program loading and error correction (debugging).

The experienced programmer may choose to skip the first five chapters and begin reading the Advanced Programming Techniques described in Chapters 6, 7, and 8. These chapters describe input/output (I/O) programming techniques, DECtape programming techniques, and DEC's floating-point packages.

## BINARY ARITHMETIC

The binary number system is the basic concept behind digital computers. The numbers 0 and 1 can easily be substituted for the two physical states associated with computer hardware: a switch is either on (1) or off (0), an area in the computer's memory is either magnetized (1) or not magnetized (0). In computer programming terminology, the binary digits 0 and 1 are sometimes called *bits* (*bi*nary dig*its*). Binary numbers are the *machine language* of computers. All computations, no matter how complex, performed by digital computers are made in binary arithmetic; therefore, students of computer programming must be thoroughly familiar with the binary system before they can begin to write programs. Chapter 1 and part of Chapter 2 provide an excellent introduction to binary arithmetic.

## ASSEMBLY LANGUAGE PROGRAMMING

Since it is quite burdensome to write programs in strings of 12-bit binary numbers (called *words*) an assembly language was developed. Assembly language enables the programmer to substitute English-like mnemonics for the binary numbers. For example,

the mnemonic JMP (for jump) is interpreted by the computer as the 12-bit binary number 101 000 000 000. The three bits in each of the four groups are given the octal values 4, 2, and 1, reading from left to right, so that if all three bits of a group were set to 1 (111), the octal value would be 7 (4+2+1=7). Thus the octal value for JMP is 5000 (101 binary = 4+0+1 octal = 5).

| 101 | 000 | 000 | 000 |
|---|---|---|---|
| 5 | 0 | 0 | 0 |

Assembly language programming is discussed in Chapters 2, 3, and 5.

## PDP-8/E SYSTEM DESCRIPTION AND USE

The programs presented in this handbook are designed to run on a PDP-8/E computer. The PDP-8/E and the peripheral equipment which comprises the PDP-8/E system are described in detail in Chapter 4.

The program commands which cause the PDP-8/E to accept data from one of its peripherals or to send data to one of its peripherals are called input/output commands. Input/output commands are discussed in Chapter 6.

One of the most important peripheral devices is the DECtape unit. The DECtape unit serves as an auxiliary magnetic tape data storage facility. DECtape is easier to use than ordinary computer magnetic tape because it allows the user to store information at fixed positions which may be directly addressed. DECtape programming is presented in Chapter 7.

DEC's floating-point packages provide for easy performance of basic arithmetic operations such as addition, subtraction, multiplication, and division, while maintaining a high degree of precision. Floating-point numbers represent quantities in the form of a fractional number multiplied by the number base 10 (for decimal) with an exponent (e.g., $12 = .12 \times 10^2$). Floating-point representation is a great help to computer programmers because it allows them to use and save very large and very small numbers by saving only the significant digits and computing an exponent to account

for leading and trailing zeroes. The floating-point packages are fully explained in Chapter 8.

## COMMON PROGRAMMING TERMS

Such words as loop, jump, nesting, and array have special meanings to computer programmers. Familiarity with these terms is prerequisite to an understanding of the information presented in this handbook. The Index/Glossary at the end of the handbook defines many of the commonly used computer programming terms.

# contents

## CHAPTER 1 COMPUTER FUNDAMENTALS

## CHAPTER 2 PROGRAMMING FUNDAMENTALS



## CHAPTER 3 ELEMENTARY PROGRAMMING TECHNIQUES

## CHAPTER 5 LOADING, EDITING AND DEBUGGING

## CHAPTER 6 INPUT/OUTPUT PROGRAMMING

## CHAPTER 7 DECTAPE PROGRAMMING

## CHAPTER 8 FLOATING-POINT PACKAGES

# computer programming fundamentals

computer
fundamentals

programming
fundamentals

elementary
programming
techniques

# chapter 1

# computer fundamentals

## INTRODUCTION

During the past 20 years, the computer revolution has dramatically changed our world, and it promises to bring about even greater changes in the years ahead.

The general purpose, digital computers being built today are much faster, smaller and more reliable and can be produced at lower cost than the earlier computers. But even more significant breakthroughs have come in the many new ways we have learned to use computers.

The first big electronic computers were usually employed as super calculators to solve complex mathematical problems that had been impossible to attack before. In recent years, computer programmers have begun using computers for non-numerical operations, such as control systems, communications, and data handling and processing. In these operations, the computer system processes vast quantities of data at high speed.

### The Computer Challenge

It has been said that a computer can be programmed to do any problem that can be defined. The key word here is defined, which means that the solution of the problem can be broken down into a series of steps that can be written as a sequence of computer instructions. The definition of some problems, such as the translation of natural languages, has turned out to be very difficult. A few years ago it was thought that computer programs could be written to translate French into English, for example. As a matter of fact, it is quite easy to translate a list of French words into English words with similar meanings. However, it is very difficult to precisely translate sentences because of the many shades of meanings associated with individual words and word combinations. For this reason, it is not practical to

try to communicate with a computer using a conventional spoken language.

While natural languages are impractical for computer use, programming languages, such as FOCAL, ALGOL, and FORTRAN with their precisely defined structure and syntax, greatly simplify communication with a computer. Programming languages are problem oriented and contain familiar words and expressions; thus, by using a programming language, it is possible to learn to write programs after a relatively short training period. Since most computer manufacturers have adopted standard programming languages and implemented the use of these languages on their computers, a given program can be executed on a large number of computers. PDP-8 programmers use FORTRAN and ALGOL-8 for scientific and engineering problems and use FOCAL-8 and BASIC-8 for shorter numerical calculations. Computer languages have been developed for programmed control of machine tools, computer typesetting, music composition, data acquisition, and many other applications. It is likely that there will be many more new programming languages in the future. Each new language development will enable the user to more easily apply the power of the computer to his particular problem or task.

Who can be a programmer? In the early days of computer programming, most programmers were mathematicians. However, as this text illustrates, most programming requires only an elementary ability to handle arithmetic and logical operations. Perhaps the most basic requirement for programming is the ability to reason logically.

The rapid expansion of the computer field in the last decade has made the resources of the computer available to hundreds of thousands of people and has provided many new career opportunities.

**Computer Applications**

A computer, like any other machine, is used because it does certain tasks better and more efficiently than humans. Specifically, it can receive more information and process it faster than a human. Often, this speed means that weeks or months of pencil and paper work can be replaced by a method requiring only minutes of computer time. Therefore, computers are used when the time saved by using a computer offsets its cost. Further, because of its capacity to handle large volumes of data in a very short time, a computer may be the *only* means of resolving problems where time is of the essence. Because of the advantages of high speed and high capacity, computers are being used more and more in business, industry, and research. Most computer applications can be classified as either *business* uses, which usually

rely upon the computer's capacity to store and quickly retrieve large amounts of information, or *scientific* uses, which require accuracy and speed in performing mathematical calculations. Both of these are performed on general purpose computers. Some examples of computer applications are given below.

*Solving Design Problems.* The computer is a very useful calculating tool for the design engineer. The wing design of a supersonic aircraft, for example, depends upon many factors. The designer describes each of these factors in the form of mathematical equations in a programming language. The computer can then be used to solve these equations.

*Scientific and Laboratory Experiments.* In scientific and laboratory experiments, computers are used to evaluate and store information from numerous types of electronic sensing devices. Computers are particularly useful in such systems as telemetry where signals must be quickly recorded or they are lost. These applications require rapid and accurate processing for both fixed conditions and dynamic situations.

*Automatic Processes.* The computer is a useful tool for manufacturing and inspecting products automatically. A computer may be programmed to run and control milling machines, turret lathes, and many other machine tools with more rapid and accurate response than is humanly possible. It can be programmed to inspect a part as it is being made and adjust the machine tool as needed. If an incoming part is defective, the computer may be programmed to reject it and start the next part.

*Training by Simulation.* It is often expensive, dangerous and impractical to train a large group of men under actual conditions to fly a commercial airplane, control a satellite, or operate a space vehicle. A computer can simulate all of these conditions for a trainee, respond to his actions, and report the results of the training. The trainee can therefore receive many hours of on-the-job training without risk to himself, others, or the expensive equipment involved.

Applications, such as those given above often require the processing of both analog and digital information. Analog information consists of continuous physical quantities that can be easily generated and controlled, such as electrical voltages or shaft rotations. Digital information, however, consists of discrete numerical values, which represent the variables of a problem. Normally, analog values are converted to equivalent digital values for arithmetic calculations to solve problems. Some computers, such as the LINC-8, combine the analog digital characteristics in one computer system.

**Computer Capabilities and Limitations**

A computer is a machine and, as all machines, it must be directed and controlled in order to perform a useful task. Until a program is prepared and stored in the computer's core memory, the computer "knows" absolutely nothing, not even how to receive input. Thus, no matter how good a particular computer may be, it must be "told" what to do. The usefulness of a computer therefore can not be fully realized until the capabilities (and the limitations) of the computer are recognized.

*Repetitive operation*—A computer can perform similar operations thousands of times, without becoming bored, tired or careless.

*Speed*—A computer processes information at enormous speeds, which are directly related to the ingenuity of the designer and the programmer. Modern computers can solve problems millions of times faster than a skilled mathematician.

*Flexibility*—General purpose computers may be programmed to solve many types of problems.

*Accuracy*—Computers may be programmed to calculate answers with a desired level of accuracy as specified by the programmer.

*Intuition*—A computer has no intuition. It can only proceed as it is directed. A man may suddenly find the answers to a problem without working out the details, but a computer must proceed as ordered.

The remainder of this chapter is devoted to the general organization of the computer and the manner in which it handles data. Included are the number systems used in programming together with the arithmetic and logical operations of the computer. This information provides a necessary background for all who desire a basic appreciation of computers and their uses, and it is a prerequisite to machine-language programming, covered in chapters 2 through 5.

## NUMBER SYSTEM PRIMER

The concept of writing numbers, counting, and performing the basic operations of addition, subtraction, multiplication, and division has been directly developed by man. Every person is introduced to these concepts during his formal education. One of the most important factors in scientific development was the invention of the decimal numbering system. The system of counting in units of tens probably developed because man has ten fingers. The use of the number 10 as the base of our number system is not of prime importance; any standard unit would do as well. The main use of a number system in early times was measuring quantities and keeping records, not performing mathematical calculations. As the sciences developed, old numbering systems became more and more outdated. The lack of an adequate numerical system greatly hampered the scientific development of early civilizations.

Two basic concepts simplified the operations needed to manipulate numbers; the concept of position, and the numeral zero. The concept of position consists of assigning to a number a value which depends both on the symbol and on its position in the whole number. For example, the digit 5 has a different value in each of the three numbers 135, 152, and 504. In the first number, the digit 5 has its original value 5; in the second, it has the value of 50; and in the last number, it has the value of 500, or 5 times 10 times 10. Sometimes a position in a number does not have a value between 1 and 9. If this position were simply left out, there would be no difference in notation between 709 and 79. This is where the numeral zero fills the gap. In the number 709, there are 7 hundreds, 0 tens and 9 units. Thus, by using the concept of position and the numeral 0, arithmetic becomes quite easy.

A few basic definitions are needed before proceeding to see how these concepts apply to digital computers.

*Unit*—The standard utilized in counting separate items is the unit.

*Quantity*—The absolute or physical amount of units.

*Number*—A number is a symbol used to represent a quantity.

*Number System*—A number system is a means of representing quantities using a set of numbers. All modern number systems use the zero to indicate no units, and other symbols to indicate quantities. The *base* or *radix* of a number system is the number of symbols it contains, including zero. For example the decimal number system is base or radix 10, because it contains 10 different symbols (viz., 0,1,2,3,4,5,6,7,8, and 9).

**Binary Number System**

The fundamental requirement of a computer is the ability to physically represent numbers and to perform operations on the numbers thus represented. Although computers which are based on other number systems have been built, modern digital computers are all based on the binary (base 2) system. To represent ten different numbers (0,1,2, . . . , 9) the computer must possess ten different states with which to associate a digit value. However, most physical quantities have only two states: a light bulb is on or off; switches are on or off; holes in paper tape or cards are punched or not punched; current is positive or negative; material is magnetized or demagnetized; etc. Because it can be represented by only two such physical states, the binary number system is used in computers.

To understand the binary number system upon which the digital computer operates, an analysis of the concepts underlying the decimal number system is beneficial.

POSITION COEFFICIENT

In the decimal numbering system (base 10), the value of a numeral depends upon the numeral's position in a number, for example:

$$\begin{array}{rcl} 347 = & 3 \times 100 = & 300 \\ & 4 \times 10 = & 40 \\ & 7 \times 1 = & 7 \\ & & \overline{347} \end{array}$$

The value of each position in a number is known as its *position coefficient*. It is also called the digit position weighting value, weighting value, or *weight*, for short. A sample decimal weighting table follows:

$$\ldots 10^3 \qquad 10^2 \qquad 10^1 \qquad 10^0$$

and, as shown above,

$$347 = 3 \times 10^2 + 4 \times 10^1 + 7 \times 10^0.$$

Weighting tables appear to serve no useful purpose in our familar decimal numbering system, but their purpose becomes apparent when we consider the binary or base 2 numbering system. In binary we have only two digits, 0 and 1. In order to represent the numbers 1 to 10, we must utilize a count-and-carry principle familar to us from the decimal

system (so familiar we are not always aware that we use it). To count from 0 to 10 in decimal, we count as follows:

0
1
2
3
4
5
6
7
8
9
10 with a carry to the $10^1$ column

Continuing the counting, when we reach 0 in the units column again, we carry another 1 to the tens column. This process is continued until the tens column becomes 0 and a 1 is carried into the hundreds column, as shown below:

| | | |
|---|---|---|
| 0 | 10 | 90 |
| 1 | 11 | 91 |
| 2 | 12 | 92 |
| 3 | 13 | 93 |
| 4 | 14 | 94 |
| 5 | 15 | 95 |
| 6 | 16 | 96 |
| 7 | 17 | 97 |
| 8 | 18 | 98 |
| 9 | 19 | 99 |
| 10 one carry | 20 one carry | 100 two carries |

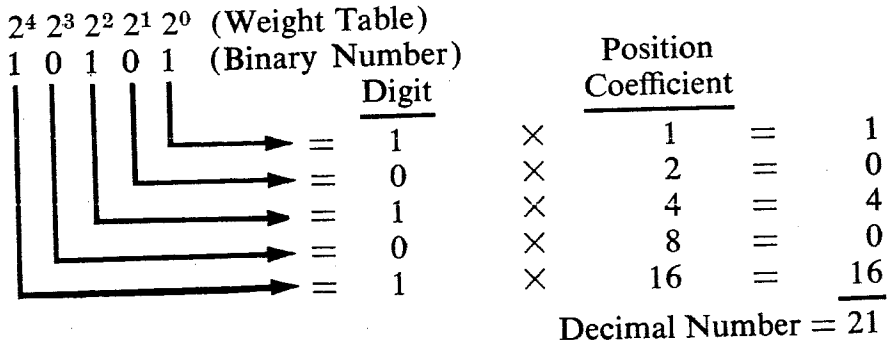
## COUNTING IN BINARY NUMBERS

In the binary number system, the carry principle is used with only two digit symbols, namely 0 and 1. Thus, the numbers used in the binary number system to count up to a decimal value of 10 are the following.

| Binary | Decimal | Binary | Decimal |
|---|---|---|---|
| 0 | (0) | 110 | (6) |
| 1 | (1) | 111 | (7) |
| 10 | (2) | 1000 | (8) |
| 11 | (3) | 1001 | (9) |
| 100 | (4) | 1010 | (10) |
| 101 | (5) | | |

When using more than one number system, it is customary to subscript numbers with the applicable base (e.g., $101_2=5_{10}$).

A weighting table is used to convert binary numbers to the more familiar decimal system.




It should be obvious that the binary weighting table can be extended, like the decimal table, as far as desired. In general, to find the value of a binary number, multiply each digit by its position coefficient and then add all of the products.

## ARRANGEMENTS OF VALUES

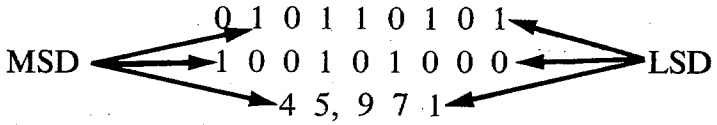
By convention, weighting values are always arranged in the same manner; the highest on the extreme left and the lowest on the extreme right. Therefore, the position coefficient begins at 1 and increases from right to left. This convention has two very practical advantages. The first advantage is that it allows the elimination of the weighting table, as such. It is not necessary to label each binary number with weighting values, as the digit on the extreme right is always multiplied by 1, the digit to its left is always multiplied by 2, the next by 4, etc. The second advantage is the elimination of some of the 0s. Whether a 0 is to the right or left, it will never add to the value of the binary number. Some 0s are required, however, as any 0s to the right of the highest valued 1 are utilized as spaces or place keepers, to keep the 1s in their correct positions. The 0s to the left, however, provide no information about the number and may be discarded, thus the number 0001010111 = 1010111.

The PDP-8 family computers operate upon 12-bit (binary digit) numbers. This means that the numbers from 0 to $111111111111_2$ ($4095_{10}$) can be directly represented.

## SIGNIFICANT DIGITS

The "leftmost" 1 in a binary number is called the *most significant digit*. This is abbreviated MSD. It is called the "most significant" in that it is multiplied by the highest position coefficient. The *least significant digit*, or LSD, is the extreme right digit. It may be a 1 or 0, and has the lowest weighting value, namely 1. The terms LSD and

MSD have the same meaning in the decimal system as in the binary system, as shown below.




## CONVERSION OF DECIMAL TO BINARY

There are two commonly used methods for converting decimal numbers to binary equivalents. The reader may choose whichever method he finds easier to use.

1. *Subtraction of Powers Method*—To convert any decimal number to its binary equivalent by the subtraction of powers method, proceed as follows.

> Subtract the highest possible power of two from the decimal number, and place a "1" in the appropriate weighting position of the partially completed binary number. Continue this procedure until the decimal number is reduced to 0. If, after the first subtraction, the next lower power of 2 cannot be subtracted, place a 0 in the appropriate weighting position. Example:

| | | |
|---|---|---|
| $42_{10}$ | = ? binary | |
| 42 | 10 | 2 |
| −32 | − 8 | −2 |
| 10 | 2 | 0 |

| $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | Power |
|---|---|---|---|---|---|---|
| 32 | 16 | 8 | 4 | 2 | 1 | Value |
| 1 | 0 | 1 | 0 | 1 | 0 | Binary |

Therefore, $42_{10} = 101010_2$.

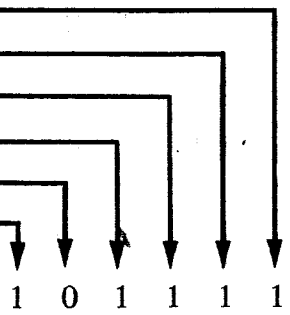
2. *Division Method*—To convert a decimal number to binary by the division method, proceed as follows.

> Divide the decimal number by 2. If there is a remainder, put a 1 in the LSD of the partially formed binary number; if there is no remainder, put a 0 in the LSD of the binary number. Divide the quotient from the first division by 2, and repeat the process. If there is a remainder,

record a 1; if there is no remainder, record a 0. Continue until the quotient has been reduced to 0. Example:

$47_{10}$ = ? Binary

| | | Quotient | Remainder |
|---|---|---|---|
| $2\overline{)47}$ | = | 23 | 1 |
| $2\overline{)23}$ | = | 11 | 1 |
| $2\overline{)11}$ | = | 5 | 1 |
| $2\overline{)5}$ | = | 2 | 1 |
| $2\overline{)2}$ | = | 1 | 0 |
| $2\overline{)1}$ | = | 0 | 1 |




Therefore, $47_{10} = 101111_2$.

## EXERCISES

a. Decimal-to-Binary Conversion — Convert the following decimal numbers to their binary equivalents.

1. $15_{10}$
2. $18_{10}$
3. $42_{10}$
4. $100_{10}$
5. $235_{10}$
6. $1_{10}$
7. $294_{10}$
8. $117_{10}$
9. $86_{10}$
10. $4090_{10}$
11. $4095_{10}$
12. $1502_{10}$
13. $377_{10}$
14. $501_{10}$
15. $828_{10}$
16. $907_{10}$
17. $4000_{10}$
18. $3456_{10}$
19. $2278_{10}$
20. $1967_{10}$

b. Binary to Decimal Conversion — Convert the following binary numbers to their decimal equivalents.

1. $110_2$
2. $101_2$
3. $1110110_2$
4. $1011110_2$
5. $0110110_2$
6. $11111_2$
7. $1010_2$
8. $110111_2$
9. $11011011101_2$
10. $111000111001_2$
11. $111010110100_2$
12. $111111110111_2$
13. $101011010101_2$
14. $111111_2$
15. $000101001_2$
16. $111111111111_2$

## Octal Number System

It is probably quite evident at this time that the binary number system, although quite nice for computers, is a little cumbersome for human usage. It is very easy for humans to make errors in reading and writing quantities of large binary numbers. The octal or base 8 numbering system helps to alleviate this problem. The base 8 or octal number system utilizes the digits 0 through 7 in forming numbers. The count-and-carry method mentioned earlier applies here also. Table 1-1 shows the octal numbers with their decimal and binary equivalents.

**Table 1-1 Decimal-Octal-Binary Equivalents**

| Decimal | Octal | Binary | Decimal | Octal | Binary |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 7 | 7 | 111 |
| 1 | 1 | 1 | 8 | 10 | 1000 |
| 2 | 2 | 10 | 9 | 11 | 1001 |
| 3 | 3 | 11 | 10 | 12 | 1010 |
| 4 | 4 | 100 | 11 | 13 | 1011 |
| 5 | 5 | 101 | 12 | 14 | 1100 |
| 6 | 6 | 110 | 13 | 15 | 1101 |

The octal number system eliminates many of the problems involved in handling the binary number system used by a computer. To make the 12-bit numbers of the PDP-8 computers easier to handle, they are often separated into four 3-bit groups. These 3-bit groups can be represented by one octal digit using the previous table of equivalents as seen below.

A binary number 11010111101

is separated into 3-bit groups by starting with the LSD end of the number and supplying leading zeros if necessary:

011 010 111 101

The binary groups are then replaced by their octal equivalents:

$011_2 = 3_8$
$010_2 = 2_8$
$111_2 = 7_8$
$101_2 = 5_8$

and the binary number is converted to its octal equivalent:

3 2 7 5.

Conversely, an octal number can be expanded to a binary number using the same table of equivalents.

$5307_8 = 101\ 011\ 000\ 111_2$

## OCTAL-TO-DECIMAL CONVERSION

Octal numbers may be converted to decimal by multiplying each digit by its weight or position coefficient and then adding the resulting products. The position coefficients in this case are powers of 8, which is the base of the octal number system. Example:

$2167_8$= ? decimal

$$
\begin{aligned}
2167_8 = \quad & 7 \times 8^0 = 7 \times 1 = 7 \\
& +6 \times 8^1 = 6 \times 8 = 48 \\
& +1 \times 8^2 = 1 \times 64 = 64 \\
& +2 \times 8^3 = 2 \times 512 = +1024 \\
& \qquad\qquad\qquad\qquad\quad 1143
\end{aligned}
$$

Therefore, $2167_8 = 1143_{10}$.

## DECIMAL-TO-OCTAL CONVERSION

There are two commonly used methods for converting decimal numbers to their octal equivalents. The reader may choose the method which he prefers.

SUBTRACTION OF POWERS METHOD. The following procedure is followed to convert a decimal number to its octal equivalent. Subtract from the decimal number the highest possible value of the form $a8^n$, where $a$ is a number between 1 and 7, and n is an integer. Record the value of $a$. Continue to subtract decreasing powers of 8 (recording the value of $a$ each time) until the decimal number is reduced to zero. Record a value of $a$=0 for all powers of 8 which could not be subtracted. Table 1-2 may be used to convert any number which can be represented by 12-bits ($4095_{10}$ or less). Appendix F contains a similar table for converting larger numbers. Example:

$2591_{10}$ = ? octal

$$
\begin{aligned}
2591 \\
-2560 &= 5 \times 8^3 = 5 \times 512 \qquad 5\ 0\ 3\ 7 \\
31 \\
-\ 0 &= 0 \times 8^2 = 0 \times 64 \\
31 \\
-\ 24 &= 3 \times 8^1 = 3 \times 8 \\
7 \\
-\ 7 &= 7 \times 8^0 = 7 \times 1 \\
0
\end{aligned}
$$

Therefore, $2591_{10} = 5037_8$.

**Table 1-2 Octal-Decimal Conversion**

| Octal Digit Position/ $8^n$ | Position Coefficients (Multipliers) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1st $(8^0)$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2nd $(8^1)$ | 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
| 3rd $(8^2)$ | 0 | 64 | 128 | 192 | 256 | 320 | 384 | 448 |
| 4th $(8^3)$ | 0 | 512 | 1,024 | 1,536 | 2,048 | 2,560 | 3,072 | 3,584 |

DIVISION METHOD. A second method for converting a decimal number to its octal equivalent is by successive division by 8. Divide the decimal number by 8 and record the remainder as the least significant digit of the octal equivalent. Continue dividing by 8, recording the remainders as the successively higher significant digits until the quotient is reduced to zero. Example:

$1376_{10} = ?$ octal

| | Quotient | Remainder |
|---|---|---|
| $8\overline{)1376}$ | 172 | 0 |
| $8\overline{)172}$ | 21 | 4 |
| $8\overline{)21}$ | 2 | 5 |
| $8\overline{)2}$ | 0 | 2 |

2 5 4 0

Therefore, $1376_{10} = 2540_8$.

EXERCISES

a. Convert the following binary numbers to their octal equivalents.

1. 1110
2. 0110
3. 111
4. 101111101
5. 110111110
6. 100000
7. 11000111
8. 011000
9. 10111111
10. 111111111111
11. 010110101011
12. 111110110100
13. 010100001011
14. 000010101101
15. 110100100100
16. 010011111010

b. Convert the following octal numbers to their binary equivalents.

1. 354
2. 736
3. 15
4. 10
5. 7
6. 5424
7. 307
8. 1101
9. 70
10. 64
11. 7777
12. 7765
13. 3214
14. 4532
15. 7033
16. 1243

c. Convert the following decimal numbers to their octal equivalents.

1. 796
2. 32
3. 4037
4. 580
5. 1000
6. 3
7. 1080
8. 1344
9. 1512
10. 3077
11. 4056
12. 4095

d. Convert the following octal numbers to their decimal equivalents.

1. 17
2. 37
3. 734
4. 1000
5. 1200
6. 742
7. 7773
8. 7777
9. 3257
10. 4577
11. 0012
12. 0256

**Fractions**

The binary and octal number systems represent fractional parts of numbers in a similar manner to the decimal system. Furthermore, fractions may be converted from one number system to another by the same techniques developed for converting whole numbers.

Before investigating the mechanics of fraction conversion, consider what a fraction is. A fraction is a number between 0 and 1, or a number less than a unit. Until now only whole numbers in the following three systems have been considered: decimal, binary, and octal. In each of these systems, the position of the symbol in the number denotes its power, and the symbol is the coefficient of that power. These are positive powers. For example, in the decimal system the number 598, 5 is the coefficient of $10^2$, 9 is the coefficient of $10^1$, and 8 is the coefficient of $10^0$. In binary and octal the same rule applies to using the powers of the base of the system.

When working with fractions, an important point to keep in mind is that fractions contain coefficients of negative powers, with the radix point being the dividing line between the non-negative and negative powers of the number system being used. Any number to the immediate right of the radix point has a power of negative (minus) 1. The first digit of the fractional number is the MSD. For example, in the decimal fraction .637; 6 is the coefficient of $10^{-1}$, 3 is the coefficient of $10^{-2}$, and 7 is the coefficient of $10^{-3}$. The coefficient of a negative power of the base is actually the numerator of a proper fraction whose denominator is the positive power of that base. For example, $.6_{10}$ ($6 \times 10^{-1}$) is equivalent to 6 divided by $10^1$ or 6/10, and also $.3_8$ ($3 \times 8^{-1}$) is equivalent to 3 divided by $8^1$ or 3/8. It should be apparent that this general rule applies to any base that may be considered. Table 1-3 contains proper fractions which have been changed to decimal, binary, and octal for comparison purposes.

## CONVERTING DECIMAL FRACTIONS TO BINARY AND OCTAL FRACTIONS

SUBTRACTION OF POWERS METHOD. One method of converting a decimal fraction to a different number system is the subtraction of powers method. In this method, subtractions of the highest possible negative power of a number in another system that is contained in the decimal fraction, are performed. In each subtraction, recording the power and its coefficient gives the equivalent number in the other system. When no subtraction is possible, a 0 is recorded. To convert a decimal fraction to a binary fraction, the powers of 2 are associated

**Table 1-3 Fraction Equivalents**

| Proper Fraction | Decimal Equivalent | Octal Equivalent | Binary Equivalent |
|---|---|---|---|
| 1/2 | .5 | .4 | .1 |
| 1/4 | .25 | .20 | .01 |
| 1/8 | .125 | .10 | .001 |
| 1/16 | .0625 | .04 | .0001 |
| 1/32 | .03125 | .02 | .00001 |
| 1/64 | .015625 | .01 | .000001 |
| 1/128 | .0078125 | .004 | .0000001 |
| 1/256 | .00390625 | .002 | .00000001 |
| 1/512 | .001953125 | .001 | .000000001 |
| 1/1024 | .0009765625 | .0004 | .0000000001 |

with coefficients of 0 or 1, since they are the only coefficients used in this system. In the octal system, the coefficients 0 through 7 are used. The following example and explanation will show the conversion of the decimal fraction .5625 to binary.

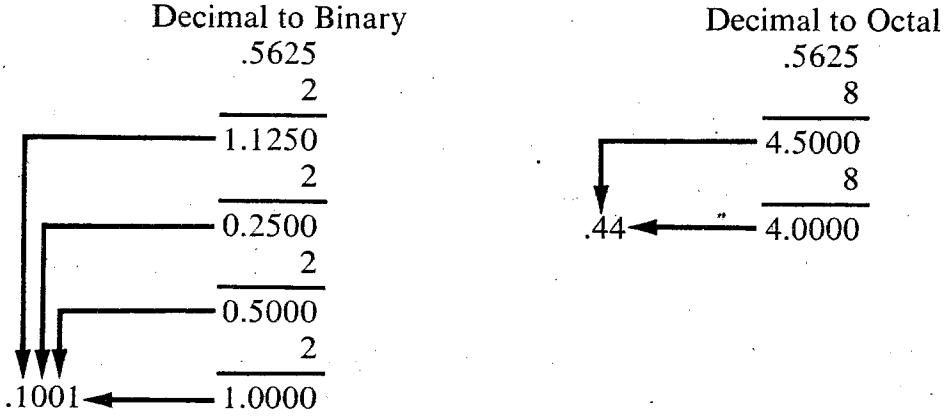
| | .5625 | .0625 | | |
|---|---|---|---|---|
| | $-.5000=2^{-1}$ | $-.0625=2^{-4}$ | | |
| | .0625 | .0000 | | |
| Negative Powers of 2 | $2^{-1}$ | $2^{-2}$ | $2^{-3}$ | $2^{-4}$ |
| Decimal Equivalents | .5000 | .2500 | .1250 | .0625 |
| Bit Values of Answer | 1 | 0 | 0 | 1 |

$= .1001$

The largest negative power of 2 contained in the decimal fraction .5625 is $2^{-1}$, which is equivalent to decimal .5000; subtract $.5000_{10}$ from $.5625_{10}$ and record a 1 in the $2^{-1}$ column. It is not possible to subtract $2^{-2}$ from the remainder, so record a 0 in the $2^{-2}$ column, $2^{-3}$ cannot be subtracted from the remainder, so record a 0 in the $2^{-3}$ column; $2^{-4}$ can be subtracted from the remainder, so record a 1 in the $2^{-4}$ column. Thus, the binary equivalent of a decimal .5625 is $.1001_2$.

Conversion to octal fractions follows the same procedure, but more than one subtraction of a given power of the base is possible. The number of times this subtraction is possible yields the coefficient of that particular power of the base. This method will not be demonstrated here, since it is very cumbersome, and easier methods are available.

MULTIPLICATION METHOD. This method of conversion is frequently used to change from a decimal fraction to another base. To convert, the decimal fraction is multiplied through by the base of the system being converted to. For example, convert decimal fraction .5625 to binary. Multiply the decimal fraction by 2. Since a whole number is obtained, record a 1 in the $2^{-1}$ column, discard the whole number portion of the number, and multiply the remainder by 2 again. No whole number is obtained, so record a 0 in the $2^{-2}$ column, and multiply the result by 2. No whole number is obtained, so record a 0 in the $2^{-3}$ column, and multiply by 2 again. A whole number is obtained, so record a 1 in the $2^{-4}$ column. The remainder, now reduced to 0, completes the conversion, and $.5625_{10}$ is $.1001_2$. The following examples show the conversion just described, and the same decimal fraction converted to octal.




## CONVERTING BINARY AND OCTAL TO DECIMAL FRACTIONS

EXPANSION METHOD. This method can be used in converting fractions from any base to a decimal fraction. Remember that the MSD is the first digit to the right of the radix point in a fractional number, and that it is multiplied by the base to the $-1$ power. The second digit is that digit multiplied by the base to the $-2$ power, etc. For example, to convert the binary fraction .10001 to decimal, proceed, as follows. The MSD is $1 \times (2^{-1})$ or 1/2, the second digit is $0 \times (2^{-2})$ or 0, the third digit is $0 \times (2^{-3})$ or 0, the fourth digit is $0 \times (2^{-4})$ or 0, and the fifth digit is $1 \times (2^{-5})$ or 1/32. The binary numbers are multiplied by the respective powers and added together to get the answer. Thus $1/2 + 1/32$ which is $16/32 + 1/32$ equals 17/32 or $.53125_{10}$.

The octal fraction .42 can be converted in the same manner, as follows. The MSD is $4 \times (8^{-1})$ or 4/8 and $2 \times (8^{-2})$ or 2/64. The fractions are now added together to get the result; $4/8 + 2/64$ or $16/32 + 1/32 = 17/32$ or $.53125_{10}$. If you look carefully at the binary fraction $.10001_2$ and divide it into groups of 3 to convert to octal, you can see that $.10001_2$ does equal $.42_8$. Zeros may be added to the right of a fraction without changing the value.

"SHORT CUT" METHOD. This is another method of converting fractions from another base to decimal. In this method, start at the LSD of the fraction and proceed to the MSD of the fraction, counting the powers of the base, the next higher power of the base will be utilized as a common denominator. The number is *assumed* to be a whole number for counting purposes. The number $.10001_2$ would be converted as follows:

| .1 | 0 | 0 | 0 | 1 |
|---|---|---|---|---|
| $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ |

The MSD is $2^4$ or 16, so the common denominator is the next higher power of 2, or 32. The numerator is converted as if it were a whole number. The result is then 17/32 which is $.53125_{10}$. The same method with the octal fraction .42 should yield the same result.

| .4 | 2 |
|---|---|
| $8^1$ | $8^0$ |

The MSD is $8^1$, or 8, so the common denominator is the next higher power of 8, or 64. Multiplying the digit values by the powers of the base and adding the products gives us the value of the numerator; thus, $4 \times (8^1) + 2 \times (8^0) = 34$, and the fraction 34/64 equals $.53125_{10}$.

**Arithmetic Operations with Binary and Octal Numbers**

Now that the reader understands the conversion techniques between the familiar decimal number system and the binary and octal number systems, arithmetic operations with binary and octal numbers will be described. The reader should remember that the binary numbers are used in the computer and that the octal numbers are used as a means of representing the binary numbers conveniently.

BINARY ADDITION

Addition of binary numbers follows the same rules as decimal or other bases. In adding decimal $1 + 8$ we have a sum of 9. This is the highest value digit. Adding one more requires the least significant digit

to become a 0 with a carry of 1 to the next place in the number. Similarly, adding binary 0 + 1 we reach the highest value a single digit can have in the binary system, and adding one more (1 + 1) requires a carry to the next higher power (1 + 1 = 10). Take the binary numbers 101 + 10 (5 + 2).

$$\begin{array}{rl} 101 & = 5_{10} \\ +010 & = 2_{10} \\ \hline 111 & = 7_{10} \end{array}$$

0 + 1 = 1, 1 + 0 = 1, and 0 + 1 = 1 with no carries required. The answer is 111, which is 7. Suppose we add 111 to 101.

$$\begin{array}{rl} 11 \leftarrow \text{carries} & \\ 111 & = 7_{10} \\ +101 & = 5_{10} \\ \hline 1100 & = 12_{10} \end{array}$$

Now 1 + 1 = 0 plus a carry of 1. In the second column, 1 plus the carry 1 = 0, plus another carry. The third column is 1 + 1 = 0 with a carry, plus the previous carry, or 1 + 1 + 1 = 11. Our answer 1100 is equal to $1 \times 2^3 + 1 \times 2^2$ or 8 + 4 = 12, which is the correct solution for 7 + 5.

## OCTAL ADDITION

Addition for octal numbers should be no problem if we keep in mind the following basic rules for addition.

1. If the sum of any column is equal to or greater than the base of the system being used, the base must be subtracted from the sum to obtain the final result of the column.
2. If the sum of any column is equal to or greater than the base, there will be a carry to the next column equal to the number of times the base was subtracted.
3. If the result of any column is less than the base, the base is not subtracted and no carry will be generated. Examples:

$$\begin{array}{rl} 5_8 & = 5_{10} \\ +\ 3_8 & = 3_{10} \\ \hline 8 & \\ -\ 8 & \\ \hline 10_8 & = 8_{10} \end{array} \qquad \begin{array}{rl} 3\ 5_8 & = 29_{10} \\ 6\ 3_8 & = 51_{10} \\ \hline 1\ 10\ 8 & \\ -8-8 & \\ \hline 1\ 2\ 0_8 & = 80_{10} \end{array}$$

**Negative Numbers and Subtraction**

Up to this point only positive numbers have been considered. Negative numbers and subtraction can be handled in the binary system in either of two ways: direct binary subtraction or by the two's complement method.

BINARY SUBTRACTION (DIRECT)

Binary numbers may be directly subtracted in a manner similar to decimal subtraction. The essential difference is that if a borrow is required, it is equal to the base of the system or 2.

$$\begin{array}{rcl} 110 & = & 6_{10} \\ -101 & = & 5_{10} \\ \hline 001 & = & 1_{10} \end{array}$$

To subtract 1 from 0 in the first column, a borrow of 1 was made from the second column which effectively added 2 to the first column. After the borrow, 2 — 1 = 1 in the first column; in the second column 0 — 0 = 0; and in the third column 1 — 1 = 0. The same numbers which were subtracted using the twos complement method are subtracted directly in the following example.

| | |
|---|---|
| 011 001 100 010 | B |
| 010 010 010 111 | A |
| 000 111 001 011 | B-A |

TWO'S COMPLEMENT ARITHMETIC

To see how negative numbers are handled in the computer, consider a mechanical register, such as a car mileage indicator, being rotated backwards. A 5-digit register approaching and passing through zero would read the following.

00005
00004
00003
00002
00001
00000
99999
99998
etc.

It should be clear that the number 99998 corresponds to −2. Further, if we add

```
      00005
      99998
    -------
1     00003
```

and ignore the carry to the left, we have effectively performed the operation of subtracting

$$5-2=3$$

The number 99998 in this example is described as the *ten's complement* of 2. Thus in the decimal number system, subtraction may be performed by adding the ten's complement of the number to be subtracted.

If a system of complements were to be used for representing negative numbers, the minus sign could be omitted in negative numbers. Thus all numbers could be represented with five digits; 2 represented as 00002, and −2 represented as 99998. Using such a system requires that a convention be established as to what is and is not a negative number. For example, if the mileage indicator is turned back to 48732, is it a negative 51268, or a positive 48732? With an ability to represent a total of 100,000 different numbers (0 to 99999), it would seem reasonable to use half for positive numbers and half for negative numbers. Thus, in this situation, 0 to 49999 would be regarded as positive, and 50000 to 99999 would be regarded as negative.

In this same manner, the two's complement of binary numbers are used to represent negative numbers, and to carry out binary subtraction, in the PDP-8 computer. In octal notation, numbers from 0000 to 3777 are regarded as positive and the numbers from 4000 to 7777 are regarded as negative.

The two's complement of a number is defined as that number which when added to the original number will result in a sum of zero. The binary number 110110110110 has a two's complement equal to 001001001010 as shown in the following addition.

```
      110 110 110 110
      001 001 001 010
    -----------------
1   000 000 000 000
```

The easiest method of finding a two's complement is to first obtain the one's complement, which is formed by setting each bit to the opposite value.

```
101 000 110 111     Number
010 111 001 000     One's complement of the number
```

The two's complement of the number is then obtained by adding 1 to the one's complement.

| | |
|---|---|
| 110 001 110 010 | Number |
| 001 110 001 101 | One's complement of the number |
| +1 | Add 1 |
| 001 110 001 110 | Two's complement of the number |

Subtraction in the PDP-8 is performed using the two's complement method. That is, to subtract A from B, A must be expressed as its two's complement and then the value of B is added to it. Example:

| | | |
|---|---|---|
| | 010 010 010 111 | A |
| | 101 101 101 001 | Two's complement of A |
| | 011 001 100 010 | B |
| (carry is ignored) | 1 000 111 001 011 | B – A |

OCTAL SUBTRACTION

Subtraction is performed in the octal number system in two ways which are directly related to the subtractions in the binary system. Subtraction may be performed directly or by the radix (base) complement method.

OCTAL SUBTRACTION (DIRECT). Octal subtraction can be performed directly as illustrated in the following examples.

| 3567–2533= ? | 2022–1234= ? |
|---|---|
| 3567 | 2022 |
| –2533 | –1234 |
| 1034 | 0566 |

Whenever a borrow is needed in octal subtraction, an 8 is borrowed as in the second example above. In the first column, an 8 is borrowed which is added to the 2 already in the first column and the 4 is subtracted from the resulting 10. In the second column, an 8 is borrowed and added to the 1 which is already in the column (after the previous borrow) and the 3 is subtracted from the resulting 9. In the third column the 2 is subtracted from a borrowed 1 (originally a borrowed 8), and in the last column 1—1=0.

EIGHT'S COMPLEMENT ARITHMETIC. Octal subtraction may be performed by adding the eight's complement of the subtrahend to the minuend. The eight's complement is obtained in the following manner.

| | |
|---|---|
| 3042 | Number |
| 4735 | Seven's complement of the number |
| +1 | Add 1 to seven's complement to obtain |
| 4736 | Eight's complement |

The seven's complement of the number is obtained by setting each digit of the complement to the value of 7 minus the digit of the number, as seen above. The eight's complement of the number is then obtained by adding 1 to the seven's complement. To prove that the complement is in fact a complement, the number is added to the complement and a result of zero and an overflow of 1 is obtained.

```
  3042
+4736
------
1 0000
```

The following example uses the eight's complement to subtract a number.

3567−2533= ?

| | | |
|---|---|---|
| | 2533 | Number |
| | 5244 | Seven's complement |
| | +1 | |
| | 5245 | Eight's complement |
| | 3567 | Minuend |
| | +5245 | Eight's complement of subtrahend |
| (carry is ignored) → | 1 1034 | Difference |

**Multiplication and Division in Binary and Octal Numbers**

Though multiplication in computers is usually achieved by means other than formal multiplication, a formal method will be demonstrated as a teaching vehicle.

BINARY MULTIPLICATION

In binary multiplication, the partial product is moved one position to the left as each successive multiplier is used. This is done in the same manner as in decimal multiplication. If the multiplier is a 0, the partial product can be a series of 0s as in example 2, or the next partial product can be moved two places to the left as in example 3, or three places as in example 4.

| Example 1. | | |
|---|---|---|
| | $462_{10}$ | Multiplicand |
| | $127_{10}$ | Multiplier |
| | 3234 | First partial product |
| | 924 | Second partial product |
| | 462 | Third partial product |
| | 58674 | Product |

Example 2.

$$\begin{array}{r} 1110110_2 \\ \times\ 1011_2 \\ \hline 1110110 \\ 1110110 \\ 0000000 \\ 1110110 \\ \hline 10100010010_2 \end{array}$$

Example 3.

$$\begin{array}{r} 1110110_2 \\ \times\ 1011_2 \\ \hline 1110110 \\ 1110110 \\ 1110110 \\ \hline 10100010010_2 \end{array}$$

Example 4.

$$\begin{array}{r} 11001110_2 \\ \times\ 11001_2 \\ \hline 11001110 \\ 11001110 \\ 11001110 \\ \hline 1010000011110_2 \end{array}$$

Because of the difficult binary additions resulting from multiplications such as the previous examples, octal multiplication of the octal equivalents of binary numbers is often substituted.

## OCTAL MULTIPLICATION

Multiplication of octal numbers is the same as multiplication of decimal numbers as long as the result is less than $10_8$. Obviously this could be a problem if it weren't for the fact that an octal multiplication table can be set up, similar to the decimal multiplication table, to make the job of multiplication of octal numbers quite simple. Table 1-4 is a partially completed octal multiplication table that will be quite useful once you have filled in the blank squares.

Using the completed octal multiplication table, the following problems may be solved.

$226_8 \times 12_8 = ?$

$$\begin{array}{r} 226_8 \\ \times 12_8 \\ \hline 454 \\ 226 \\ \hline 2734_8 \end{array}$$

$1247_8 \times 305_8 = ?$

$$\begin{array}{r} 1247_8 \\ \times 305_8 \\ \hline 6503 \\ 0000 \\ 3765 \\ \hline 405203_8 \end{array}$$

**Table 1-4 Octal Multiplication Table**

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2 | 0 | 2 | 4 | 6 | 10 | | | |
| 3 | 0 | 3 | 6 | 11 | 14 | | | |
| 4 | | | | | | | | |
| 5 | | | | | | | | |
| 6 | | | | | | | | |
| 7 | 0 | 7 | 16 | 25 | | | | |

## BINARY DIVISION

Once the reader has mastered binary subtraction and multiplication, binary division is easily learned. The following problem solutions illustrate binary division.

Divide $\frac{10010_2}{10_2}$

$$\begin{array}{r} 1001 \\ 10 \overline{)10010} \\ 10 \\ \hline 00 \\ 00 \\ \hline 01 \\ 00 \\ \hline 10 \\ 10 \\ \hline 0 \end{array}$$

$$\frac{10010_2}{10_2} = \frac{18_{10}}{2_{10}} = 1001_2 = 9_{10}$$

Divide $\frac{1110_2}{100_2} = \frac{14_{10}}{4_{10}} = 3.5_{10}$

```
      11.1
100 ) 1110.0
      100
      ---
       110
       100
       ---
        100
        100
        ---
          0
```

$11.1_2 = 3.5_{10}$

## OCTAL DIVISION

Octal division uses the same principles as decimal division. All multiplication and subtraction must however be done in octal. (Refer to the octal multiplication table.) The following problem solutions illustrate octal division.

$\frac{62_8}{2_8} = \frac{50_{10}}{2_{10}}$

```
     31
2 ) 62
     6
     --
     02
      2
     --
      0
```

$= 31_8 = 25_{10}$

$\frac{1714_8}{22_8}$

```
      66
22 ) 1714
     154
     ---
      154
      154
      ---
        0
```

## EXERCISES

a. Perform the following binary additions.

1. 10110 + 101
2. 100 + 10
3. 11011 + 0010
4. 10110111 + 1
5. 1101 + 101 + 11
6. 101 + 1 + 110
7. 1110 + 100 + 11
8. 1111 + 101 + 1000
9. 110111 + 100100 + 110001
10. 100111 + 111001 + 101101
11. 11011001 + 10010011 + 11100011
12. 11011011 + 10111011 + 00101011 + 01010111 + 01111101

b. Find the one's complement and the two's complement of the following numbers.

1. 011 100 110 010
2. 010 111 011 111
3. 011 110 000 000
4. 000 000 000 000
5. 000 000 000 001
6. 000 100 100 100
7. 000 000 000 111
8. 100 000 000 000
9. 100 000 010 010
10. 100 001 100 110
11. 111 111 111 110
12. 111 111 111 111

c. Subtract the following binary numbers directly.

1. 101000001 − 010111101
2. 1010111010 − 0101110101
3. 101011010111 − 011111111101
4. 101111100111 − 010101110010

d. Perform the following subtractions by the two's complement method. Check your work by direct subtraction. Show all work.

1. 011 011 011 011 — 001 111 010 110
2. 000 111 111 111 — 000 001 001 101
3. 011 111 111 101 — 010 101 100 011
4. 001 101 111 110 — 001 100 101 011
5. 011 111 111 111 — 010 101 101 101

e. Multiply the following binary numbers.

1. 11011 × 110
2. 1011101 × 101
3. 101011101011 × 10000

f. Divide the following binary numbers.

1. 100 / 10
2. 10000 / 100
3. 1100100 / 10100

g. Add the following octal numbers.

1. 42 + 53

2. 45 + 23

3. 34 + 76

6. 127 + 256 + 724

4. 77 + 11

5. 3357 + 562

7. 777 + 543 + 612

8. 437 + 426 + 772 + 747 + 575

h. Subtract the following octal numbers directly.

1. 42 − 23

2. 76 − 34

3. 77 − 11

4. 53 − 44

5. 7474 − 4777

6. 7000 − 6573

7. 2543 − 2174

8. 7500 − 6373

i. Perform the following octal subtractions by the eight's complement method. Check your work by subtracting directly. Show all work.

1. 0377 − 0233
2. 2345 − 1456
3. 1144 − 1046
4. 3000 − 0011
5. 2311 − 2277
6. 0044 − 0017
7. 3234 − 2777
8. 1111 − 0777

j. Multiply the following octal numbers.

1. 65 × 4

2. 14 × 13

3. 77 × 65

4. 716 × 472

5. 425 × 377

6. 571 × 246

k. Prove the answers to the problems in (j) by division, as follows:

Multiplicand × Multiplier = Product

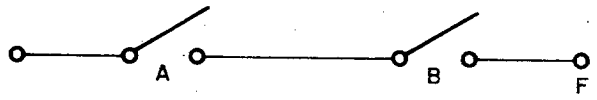
Multiplier ) Product (Multiplicand

## LOGIC OPERATION PRIMER

Computers use logic operations in addition to arithmetic operations to solve problems. The logic operations have a direct relationship with the algebraic system to represent logic statements known as Boolean algebra. In logic, there are two basic connectives that are used to express the relationship between two statements. These are the AND and the OR.

### The AND Operation

The following simple circuit with two switches illustrates the AND operation. If current is allowed to flow through a switch, the switch is said to have a value of 1. If the switch is open and current cannot flow, the switch has a value of 0. If the whole circuit is considered, it will have a value of 1 (i.e., current may flow through it) whenever *both* A *and* B are 1. This is the AND operation.




The AND operation is often stated A • B = F. The multiplication symbol (•) is used to represent the AND connective. The relationship between the variables and the resulting value of F is summarized in the following table.

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

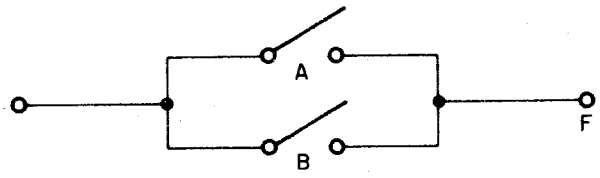
When the AND operation is applied to binary numbers, a binary 1 will appear in the result if a binary 1 appeared in the corresponding position of the two numbers.

The AND operation can be used to *mask* out a portion of a 12-bit number.

| To Be Masked Out | To Be Retained for Subsequent Operation | |
|---|---|---|
| 010 101 | 010 101 | (12-bit number) |
| 000 000 | 111 111 | (mask) |
| 000 000 | 010 101 | (result) |

**The OR Operation**

A second logic operation is the OR (sometimes called the inclusive OR). Statements which are combined using the OR connective are illustrated by the following circuit diagram.




Current in the above diagram may flow whenever *either* A *or* B (or both) is closed (F=1 if A=1, or B=1, or A=1 and B=1). This operation is expressed by the plus (+) sign; thus A+B=F. The following table shows the resulting value of F for changing values of A and B.

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

Thus, if A and B are the 12-bit numbers shown below, A+B is evaluated as follows.

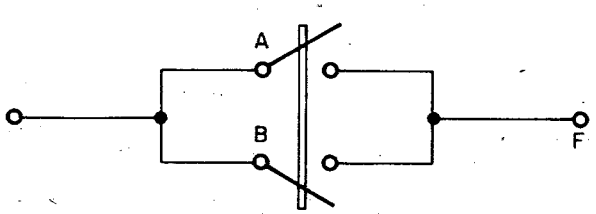
$$\begin{aligned} A &= 011\ 010\ 011\ 111 \\ B &= 100\ 110\ 010\ 011 \\ A+B &= 111\ 110\ 011\ 111 \end{aligned}$$

Remember that the "+" in the above example means "inclusive OR", not "add."

**The Exclusive OR Operation**

The third and last logic operation is the exclusive OR. The exclusive OR is similar to the inclusive OR with the exception that one set of conditions for A and B are *excluded*. This exclusion can be symbolized in the circuit diagram by connecting the two switches mechanically together. This connection makes it impossible for the switches to be closed

simultaneously, although they may be open simultaneously or individually.




Thus, the circuit is completed when A=1 and B=0, and when A=0 and B=1. The results of the exclusive OR operation are summarized in the table below.

| A | B | F |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

The exclusive OR of two 12-bit numbers is evaluated and labeled F in the following operation.

A = 011 010 011 111
B = 100 110 010 011
F = 111 100 001 100

## GENERAL ORGANIZATION OF THE PDP-8

Almost every general purpose digital computer has the basic units shown in Figure 1-1, on the following page.

If a machine is to be called a computer, it must have the capability of performing some types of arithmetic operations. The element of a digital computer that meets this requirement is called the arithmetic unit. In order for the arithmetic unit to be able to do its required task, it must be told what to do. Therefore, a control unit is necessary.

Since mathematical operations are performed by the arithmetic unit, it may be necessary to store a partial answer while the unit is computing another part of the problem. This stored partial answer can then be used to solve other parts of the problem. It is also helpful for the control unit and arithmetic unit to have information immediately available for their use, and for the use of other units within the computer. This requirement is met by the portion of the computer designated as the memory unit, or core storage unit.

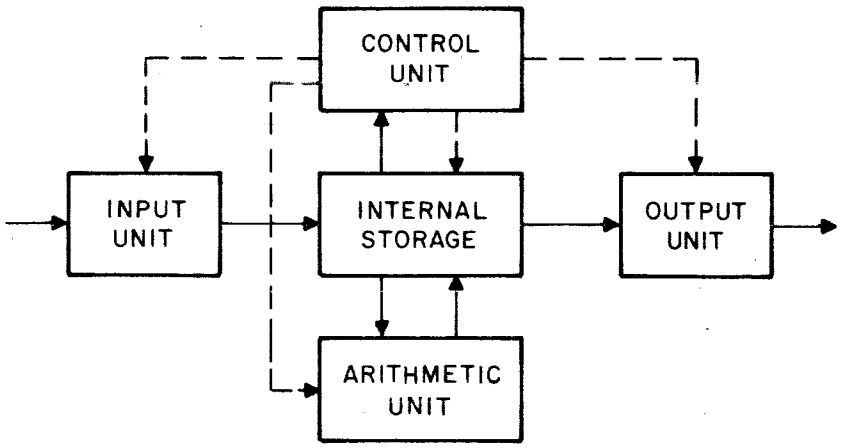



Figure 1-1 PDP-8 General Organization

The prime purpose of a digital computer is to serve humans in some manner. In order to do this there must be a method of transmitting our wants to the computer, and a means of receiving the results of the computer's calculations. The portions of the computer that carry out these functions are the input and output units.

**Arithmetic Unit**

The arithmetic unit of a digital computer performs the actual work of computation and calculation. It carries out its job by counting series of pulses or by the use of logic circuits. Modern computers use components such as transistors and integrated circuits. Switches and relays were used previously, and were acceptable as far as their ability to perform computations was concerned. Modern computers, however, because of the speed desired, make use of smaller electronic components whenever possible.

The arithmetic unit of the PDP-8 has, as its major component, a 12-bit *accumulator,* which is simply a register capable of storing a number of 12 binary digits. It is called the accumulator because it accumulates partial sums during the operation of the PDP-8. All arithmetic operations are performed in the accumulator of the PDP-8.

**Control Unit**

The control unit of a digital computer is an administrative or switching section. It receives information entering the machine and decides how and when to perform operations. It tells the arithmetic unit what to do and where to get the necessary information. It knows when the arithmetic unit has completed a calculation and it tells the arithmetic unit what to do with the results, and what to do next.

The control unit itself knows what to tell the arithmetic unit to do by interpreting a set of instructions. This set of instructions for the control unit is called a *program* and is stored in the computer memory.

**Memory Unit**

The memory unit, sometimes called the core storage unit, contains information for the control unit (instructions) and for the arithmetic unit (data). The terms core storage and memory may be used interchangeably. Some computer texts refer to external units as storage, such as magnetic tapes and disks, and to internal units as memory, such as magnetic cores. The requirements of the internal storage units may vary greatly from computer to computer.

The PDP-8 memory unit is composed of magnetic cores which are often compared to tiny doughnuts. These magnetic cores record binary information by the direction in which they are magnetized (clockwise or counterclockwise). The memory unit is arranged in such a way that it can store 4096 "words" of binary information. These words are each 12-bits in length. Each core storage location has an address, which is a unique number used by the control unit to specify that location. Storage of this type in which each location can be specified and reached as easily as any other is referred to as *random-access* storage. The other type of storage is sequential storage such as magnetic tape, in which case some locations (those at the beginning of the tape) are easier to reach than others (those at the end of the tape).

**Input Unit**

Input devices are used to supply the values needed by the computer and the instructions to tell the computer how to operate on the values. Input unit requirements vary greatly from machine to machine. A manually operated keyboard may be sufficient for a small computer. Other computers requiring faster input use punched cards for data inputs. Some systems utilize removable plugboards that can be pre-wired to perform certain instructions. Input may also be via punched paper tape or magnetic tape, two forms of input common in PDP-8 systems.

**Output Unit**

Output devices record the results of the computer operations. These results may be recorded in a permanent form (e.g., as a printout on the teleprinter) or they may be used to initiate a physical action (e.g., to adjust a pressure valve setting). Many of the media used for input, such as paper tape, punched cards, and magnetic tape, can also be used for output.

## COMPUTER DATA FORMATS

The PDP-8 uses 12-bit words to represent data. Some of the formats in which this data is represented are described in the following paragraphs.

### Alphabetic Characters

Computers are designed to operate upon the binary numbers which it conveniently represents with electronic components. There are occasions however when it is desirable to have the computer represent characters of the alphabet and punctuation marks. Binary codes are used to represent such characters. For example, the reader is familiar with punched cards, which use a system of punched holes to represent information. Each of these codes associates some character with a particular binary number. The computer can store the binary number (not the character) in its memory. When so directed, the computer will output the binary code to a device which will interpret the code and print the character. Some specific binary codes used to represent alphanumeric information (letters, numbers, and punctuation symbols) are presented in Appendix B.

### Number Representations

The PDP-8 operates upon 12-bit words (namely 0 to 111 111 111 $111_2$, or 0 to $7777_8$). By convention, one half of the numbers are considered positive (0 to 011 111 111 $111_2$, or 0 to $3777_8$), and one half (100 000 000 $000_2$ to 111 111 111 $111_2$ or $4000_8$ to $7777_8$) are considered negative. Therefore the PDP-8 can directly represent the portion of the number line shown in Figure 1-2.

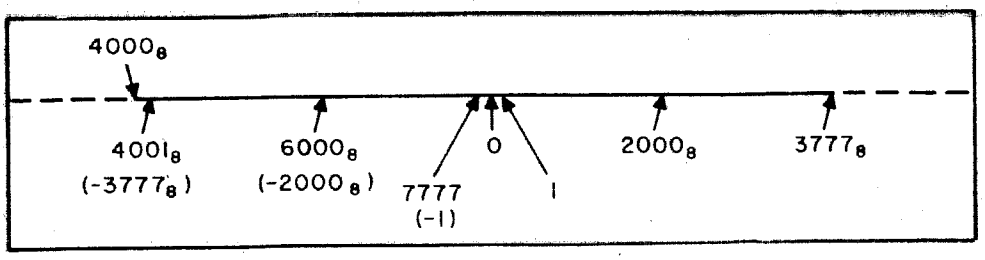



Figure 1-2 PDP-8 Octal Number Line

Notice that the first digit of the 12-bit binary numbers is in effect a "sign bit." That is, bit 0 (the first bit) specifies the sign of the number by the following rule. If bit 0 is a 0, the number is positive; if bit 0 is a 1, the number is negative. This is the means by which the computer

tests for positive and negative numbers. Thus, the zero is considered positive. In figure 1-2 it should be noted that the number 4000 is peculiar in that it has no positive counterpart. (Expressed in octal, the "two's complement" of 3776 is 4002; of 3777 is 4001; of 4000 is 4000.)

When the octal to decimal conversions are performed, the number line of Figure 1-2 is converted to the number line of Figure 1-3. Thus the PDP-8 can represent directly the numbers between $-2048_{10}$ and $+2047_{10}$. This would seem to be a serious restriction. Through two techniques however this limitation is overcome.

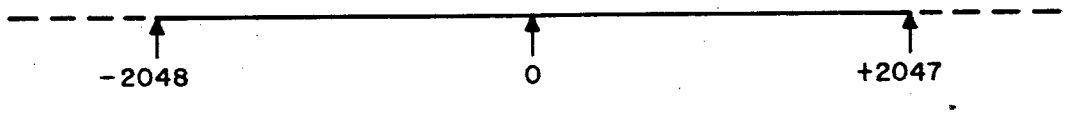



Figure 1-3 PDP-8 Decimal Number Line

## DOUBLE PRECISION NUMBERS

The PDP-8 memory is made up of 12-bit storage locations. Suppose however that a number larger than 12-bits were to be stored. By using two 12-bit storage locations, numbers between $-8{,}388{,}608_{10}$ and $8{,}388{,}607_{10}$ may be represented directly. This method of representation is appropriately called *double precision*. The method could be extended to triple precision and further if necessary.

It should be noted that to add double precision numbers, two additions are needed. Double precision arithmetic is described in Chapter 3.

## FLOATING POINT NUMBERS

Another method of representing numbers in the PDP-8 with more than one 12-bit word is floating point notation. In this notation, a number is divided into two parts, namely a mantissa (number part) and an exponent (to some base). In the decimal number system for example, the number 12 can be written in the following ways.

| MANTISSA | | EXPONENT |
|---|---|---|
| .12 | × | $10^2$ |
| 1.2 | × | $10^1$ |
| 12. | × | $10^0$ |
| 120. | × | $10^{-1}$ |
| 1200. | × | $10^{-2}$ |

PDP-8 floating point notation makes use of a representation similar to the above with the exception that the exponent and the mantissa are binary

numbers. The binary mantissa (number part) is stored in two locations and a third location stores the exponent. The exponent is selected such that the mantissa has no leading zeros, thereby retaining the maximum number of significant digits.

# chapter 2
# programming fundamentals

This chapter describes the three general types of computer instructions and the way in which they are used in computer programs. The first type of instruction is distinguished by the fact that it operates upon data that is stored in some memory location and must tell the computer where the data is located in core so that the computer can find it. This type of instruction is said to *reference* a location in core memory; therefore, these instructions are often called memory reference instructions (MRI).

When speaking of memory locations, it is very important that a clear distinction is made between the address of a location and the contents of that location. A memory reference instruction refers to a location by a 12-bit *address;* however, the instruction causes the computer to take some specified action with the *content* of the location. Thus, although the address of a specific location in memory remains the same, the content of the location is subject to change. In summary, a memory reference instruction uses a 12-bit address value to refer to a memory location, and it operates on the 12-bit binary number stored in the referenced memory location.

The second type of instructions are the operate microinstructions, which perform a variety of program operations without any need for reference to a memory location. Instructions of this type are used to perform the following operations: clear the accumulator, test for negative accumulator, halt program execution, etc. Many of these operate microinstructions can be combined (microprogrammed) to increase the operating efficiency of the computer.

The third general type of instructions are the input/output transfer (IOT) instructions. These instructions perform the transfer of information between a peripheral device and the computer memory. IOT instructions are discussed in Chapter 5.

## PROGRAM CODING

Binary numbers are the only language which the computer is able to understand. It stores numbers in binary and does all its arithmetic operations in binary. What is more important to the programmer, however, is that in order for the computer to understand an instruction it must be represented in binary. The computer can not understand instructions which use English language words. All instructions must be in the form of binary numbers (binary code).

### Binary Coding

The computer has a set of instructions in binary code which it "understands". In other words, the circuitry of the machine is wired to react to these binary numbers in a certain manner. These instructions have the same appearance as any other binary number; the computer can interpret the same binary configuration of 0's and 1's as data or as an instruction. The programmer tells the computer whether to interpret the binary configuration as an instruction or as data by the way in which the configuration is encountered in the program.

Suppose the computer has the following binary instruction set.

| | | |
|---|---|---|
| Instruction A | 001 000 010 010 | This binary number instructs the computer to add the contents of location 000 000 010 010 to the accumulator. |
| Instruction B | 001 000 010 111 | This binary number instructs the computer to add the contents of location 000 000 010 111 to the accumulator. |

If instruction B is contained in a core memory location with an address of 000 000 010 010 and the binary number 000 111 111 111 is stored in a location with an address of 000 000 010 111, the following program could be written:

| Location | Content |
|---|---|
| 000 000 010 010 | 001 000 010 111 |
| 000 000 010 111 | 000 111 111 111 |

If this program were to be executed, the number 000 111 111 111 would be added to the accumulator.

### Octal Coding

If binary configurations appear cumbersome and confusing, the reader will now understand why most programmers seldom use the binary number system in actual practice. Instead, they substitute the

octal number system which was discussed in Chapter 1. The reader should not proceed until he understands these two number systems and the conversions between them.

Henceforth, octal numbers will be used to represent the binary numbers which the computer uses. Although the programmer may use octal numbers to describe the binary numbers within the computer, it should be remembered that the octal representation itself does not exist within the computer.

When the conversion to octal is performed, Instruction B becomes $1027_8$ and the previous program becomes

| Location | Content |
|---|---|
| $0022_8$ | $1027_8$ |
| $0027_8$ | $0777_8$ |

To demonstrate that a computer cannot distinguish between a number and an instruction, consider the following program.

| Location | Content | |
|---|---|---|
| 0021 | 1022 | (Instruction A) |
| 0022 | 1027 | (Instruction B) |
| . | | |
| . | | |
| . | | |
| 0027 | 0777 | (The number $777_8$) |

Instruction A, which adds the contents of location 0022 to the accumulator, has been combined with the previous program. Upon execution of the program (assuming the initial accumulator value=0), the computer will execute instruction A and add $1027_8$ as a number to the accumulator obtaining a result of $1027_8$. The computer will then execute the next instruction, which is 1027, causing the computer to add the contents of 0027 to the accumulator. After the execution of the two instructions the number $2026_8$ is in the accumulator. Thus, the above program caused the number $1027_8$ to be used as an instruction and as a number by the computer.

**Mnemonic Coding**

Coding a program in octal numbers, although an improvement upon binary coding, is nevertheless very inconvenient. The programmer must learn a complete set of octal numbers which have no logical connection with the operations they represent. The coding is difficult for the programmer when he is writing the program, and this difficulty is compounded when he is trying to debug or correct a program. There is no easy way to remember the correspondence between an octal number and a computer operation.

To simplify the process of writing or reading a program, each instruction is often represented by a simple 3- or 4-letter mnemonic symbol. These mnemonic symbols are considerably easier to relate to a computer operation because the letters often suggest the definition of the instruction. The programmer is now able to write a program in a language of letters and numbers which suggests the meaning of each instruction.

The computer still does not understand any language except binary numbers. Now, however, a program can be written in a symbolic language and translated into the binary code of the computer because of the one-to-one correspondence between the binary instructions and the mnemonics. This translation could be done by hand, defeating the purpose of mnemonic instructions, or the computer could be used to do the translating for the programmer. Using a binary code to represent alphabetic characters as described in Chapter 1, the programmer is able to store alphabetic information in the computer memory. By instructing the computer to perform a translation, substituting binary numbers for the alphabetic characters, a program is generated in the binary code of the computer. This process of translation is called "assembling" a program. The program that performs the translation is called an assembler.

Although the assembler is described in detail in Chapter 6, it is well to make some observations about the assembler at this point.

1. The assembler itself must be written in binary code, not mnemonics.
2. It performs a one-to-one translation of mnemonic codes into binary numbers.
3. It allows programs to be written in a symbolic language which is easier for the programmer to understand and remember.

A specific mnemonic language for the PDP-8, called PAL (Program Assembly Language), is introduced later in this chapter. The next section describes the general PDP-8 characteristics and components. This information is necessary to an understanding of the PDP-8 instructions and their uses within a program.

## PDP-8 ORGANIZATION AND STRUCTURE

The PDP-8 is a high-speed, general purpose digital computer which operates on 12-bit binary numbers. It is a single-address parallel machine using two's complement arithmetic. It is composed of the five basic computer units which were discussed in Chapter 1. The com-

ponents of the five units and their interrelationships are shown in Figure 2-1. For simplicity, the input and output units have been combined.

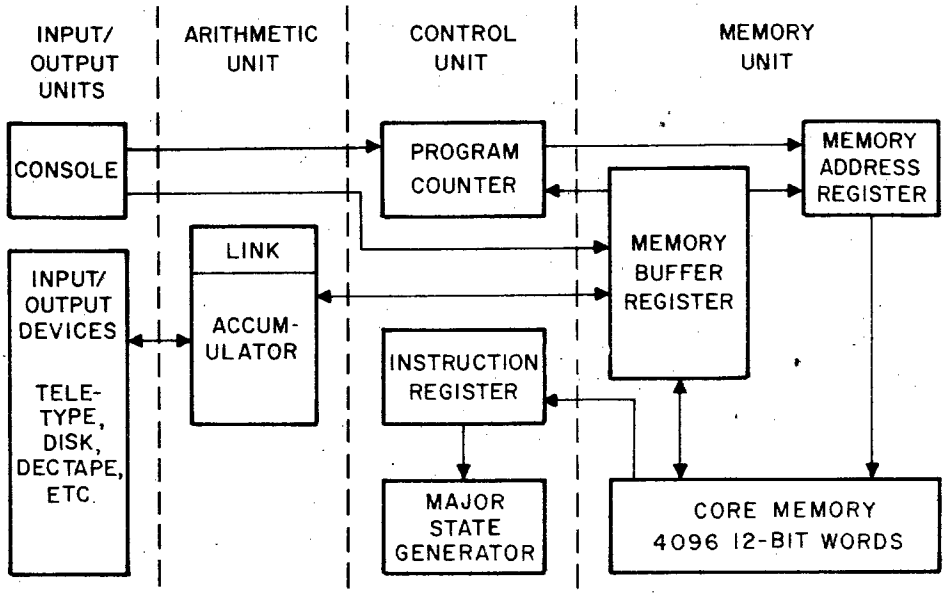



Figure 2-1 Block Diagram of the PDP-8

**Input and Output Units**

The input and output units are combined in Figure 2-1 because in many cases the same device acts as both an input and an output unit. The Teletype console, for example, can be used to input information which will be accepted by the computer, or it can accept processed information and print it as output. Thus, the two units of input and output are very often joined and referred to as input/output or simply I/O. Chapter 5 describes the methods of transmitting data as either input or output; but for the present, the reader can assume that the computer is able to accept information from devices such as those listed in the block diagram and to return output information to the devices. The PDP-8 console allows the programmer direct access to core memory and the program counter by setting a series of switches, as described in detail in Chapter 4.

**Arithmetic Unit**

The second unit contained in the PDP-8 block diagram is the arithmetic unit. This unit, as shown in the diagram, accepts data from input devices and transmits processed data to the output devices as well. Primarily, however, the unit performs calculations under the direction of the control unit. The Arithmetic Unit in the PDP-8 consists of an *accumulator* and a *link* bit.

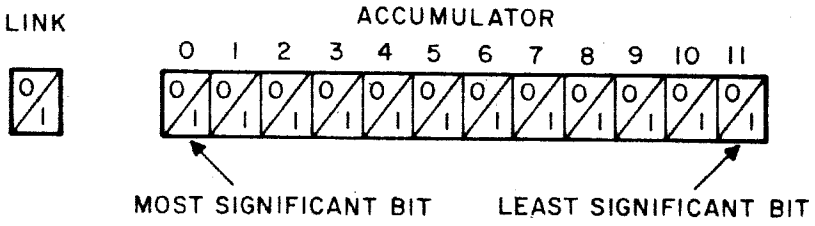
ACCUMULATOR (AC)

The prime component of the arithmetic unit is a 12-bit register called the accumulator. It is surrounded by the electronic circuits which perform the binary operations under the direction of the control unit. Its name comes from the fact that it accumulates partial sums during the execution of a program. Because the accumulator is only twelve bits in length, whenever a binary addition causes a carry out of the most significant bit, the carry is lost from the accumulator. This carry is recorded by the *link bit.*

LINK (L)

Attached logically to the accumulator is a 1-bit register, called the link, which is complemented by any carry out of the accumulator. In other words, if a carry results from an addition of the most significant bit in the accumulator, this carry results in a link value change from 0 to 1, or 1 to 0, depending upon the original state of the link.

Below is a diagram of the accumulator and link. The twelve bits of the accumulator are numbered 0 to 11, with bit 0 being the most significant bit. The bits of the AC and L can be either binary 0's or 1's as shown below.




**Control Unit**

The *instruction register, major state generator,* and *program counter* can be identified as part of the control unit. These registers keep track of what the computer is now doing and what it will do next, thus specifying the flow of the program from beginning to end.

PROGRAM COUNTER (PC)

The program counter is used by the PDP-8 control unit to record the locations in memory (addresses) of the instructions to be executed. The PC always contains the address of the next instruction to be executed. Ordinarily, instructions are stored in numerically consecutive locations and the program counter is set to the address of the next instruction to be executed merely by increasing itself by 1 with each successive instruction. When an instruction causing transfer of command to another portion of the stored program is encountered, the PC is set

to the appropriate address. The PC must be initially set by input to specify the starting address of a program, but further actions are controlled by program instructions.

INSTRUCTION REGISTER (IR)

The 3-bit instruction register is used by the control unit to specify the main characteristics of the instruction being executed. The three most significant bits of the current instruction are loaded into the IR each time an instruction is loaded into the memory buffer register from core memory. These three bits contain the operation code which specifies the main characteristics of an instruction. The other details are specified by the remaining nine bits (called the operand) of the instruction.

MAJOR STATE GENERATOR

The major state generator establishes the proper states in sequence for the instruction being executed. One or more of the following three major states are entered serially to execute each programmed instruction. During a *Fetch* state, an instruction is loaded from core memory, at the address specified by the program counter, into the memory buffer register. The *Defer* state is used in conjunction with indirect addressing to obtain the effective address, as discussed under "Indirect Addressing" later in this chapter. During the *Execute* state, the instruction in the memory buffer register is performed.

**Memory Unit**

The PDP-8 basic memory unit consists of 4,096 12-bit words of *magnetic core memory,* a 12-bit *memory address register,* and a 12-bit *memory buffer register.* The memory unit may be expanded in units of 4,096 words up to a maximum of 32,768 words.

CORE MEMORY

The core memory provides storage for the instructions to be performed and information to be processed. It is a form of random access storage, meaning that any specific location can be reached in memory as readily as any other. The basic PDP-8 memory contains 4,096 12-bit magnetic core *words.* These 4,096 words require that 12-bit addresses be used to specify the address for each location uniquely.

MEMORY BUFFER REGISTER (MB)

All transfers of instructions or information between core memory and the processor registers (AC, PC, and IR) are temporarily held in the memory buffer register. Thus, the MB holds all words that go into and out of memory, updates the program counter, sets the instruction register, sets the memory address register, and accepts information from or provides information to the accumulator.

## MEMORY ADDRESS REGISTER (MA)

The address specified by a memory reference instruction is held in the memory address register. It is also used to specify the address of the next instruction to be brought out of memory and performed. It can be used to directly address all of core memory. The MA can be set by the memory buffer register, or by input through the program counter register, or by the program counter itself.

## MEMORY REFERENCE INSTRUCTIONS

The standard set of instructions for the PDP-8 includes eight basic instructions. The first six of these instructions are introduced in the following paragraphs and are presented in both octal and mnemonic form with a description of the action of each instruction.

The memory reference instructions (MRI) require an operand to specify the address of the location to which the instruction refers. The manner in which locations are specified for the PDP-8 is discussed in detail under "Page Addressing" later in this chapter. In the following discussion, the first three bits (the first octal digit) of an MRI are used to specify the instruction to be performed. (The last nine bits, three octal digits, of the 12-bit word are used to specify the address of the referenced location—that is, the operand.)

The six memory reference instructions are listed below with their mnemonic and octal equivalents as well as their memory cycle times.

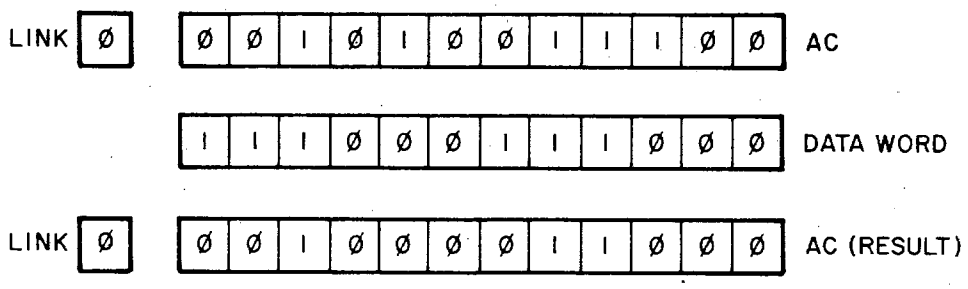
| Instruction | Mnemonic[2] | Octal Value | Memory Cycles[1] |
|---|---|---|---|
| Logical AND | AND | 0nnn | 2 |
| Two's Complement Add | TAD | 1nnn | 2 |
| Deposit and Clear the Accumulator | DCA | 3nnn | 2 |
| Jump | JMP | 5nnn | 1 |
| Increment and Skip if Zero | ISZ | 2nnn | 2 |
| Jump to Subroutine | JMS | 4nnn | 2 |

[1] Memory cycle time for the PDP-8 and -8/I is 1.5 microseconds; for the PDP-8/L, it is 1.6; for the PDP-8/S, it is 8 microseconds. (Indirect addressing requires an additional memory cycle.)

[2] The mnemonic code is meaningful to and translated by an assembler into binary code.

**AND ($0nnn_8$)**

The AND instruction causes a bit-by-bit Boolean AND operation between the contents of the accumulator and the data word specified by the instruction. The result is left in the accumulator as illustrated below.

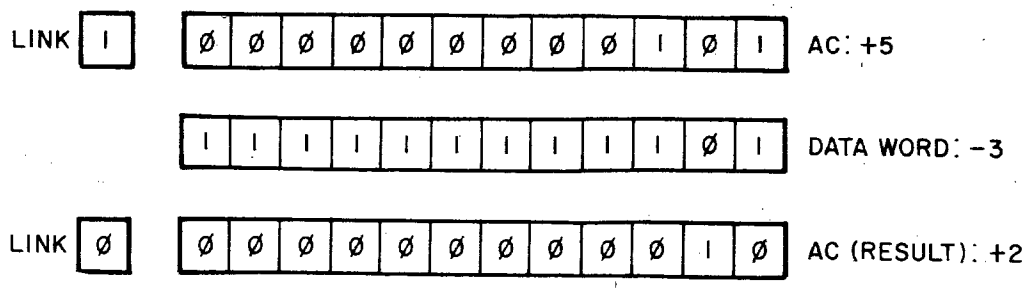



The following points should be noted with respect to the AND instruction:

1. A 1 appears in the AC only when a 1 is present in both the AC and the data word (The data word is often referred to as a mask);
2. The state of the link bit is not affected by the AND instruction; and
3. The data word in the referenced location is not altered.

**TAD ($1nnn_8$)**

The TAD instruction performs a binary addition between the specified data word and the contents of the accumulator, leaving the result of the addition in the accumulator. If a carry out of the most significant bit of the accumulator should occur, the state of the link bit is complemented. The add instruction is called a Two's Complement Add to remind the programmer that negative numbers must be expressed as the two's complement of the positive value. The following figure illustrates the operation of the TAD instruction.



The following points should be remembered when using the TAD instruction:

1. Negative numbers must be expressed as a two's complement of the positive value of the number;
2. A carry out of the accumulator will complement the link; and
3. The data word in the referenced location is not affected.

**DCA ($3nnn_8$)**

The DCA instruction stores the contents of the AC in the referenced location, destroying the original contents of the location. The AC is then set to all zeroes. The following example shows the contents of the accumulator, link, and location 225 before and after executing the instruction DCA 225.

DCA 225

| | AC | Link | Loc. 225 |
|---|---|---|---|
| *Before Execution* | 1234 | 1 | 7654 |
| *After Execution* | 0000 | 1 | 1234 |

The following facts should be kept in mind when using the DCA instruction:

1. The state of the link bit is not altered;
2. The AC is cleared; and
3. The original contents of the addressed location are replaced by the value of the AC.

**JMP ($5nnn_8$)**

The JMP instruction loads the effective address of the instruction into the program counter, thereby changing the program sequence since the PC specifies the next instruction to be performed. In the following example, execution of the instruction in location 250 (JMP 300) causes the program to jump over the instructions in locations 251 through 277 and immediately transfer control to the instruction in location 300.

| Location | Content | |
|---|---|---|
| 250 | JMP 300 | (This instruction transfers program control to location 300.) |
| . | . | |
| . | . | |
| . | . | |
| 300 | DCA 330 | |

NOTE: The JMP instruction does not affect the contents of the AC or link.

**ISZ ($2nnn_8$)**

The ISZ instruction adds a 1 to the referenced data word and then examines the result of the addition. If a zero result occurs, the instruction following the ISZ is skipped. If the result is not zero, the instruction

following the ISZ is performed. In either case, the result of the addition replaces the original data word in memory. The example in Figure 2-2 illustrates one method of adding the contents of a given location to the AC a specified number of times (multiplying) by using an ISZ instruction to increment a tally. The effect of this example is to multiply the contents of location 275 by 2. (To add the contents of a given location to the AC twice, using the ISZ loop, as shown in Figure 2-2, requires more instructions than merely repeating the TAD instruction. However, when adding the contents four or more times, use of the ISZ loop requires fewer instructions.) In the first pass of the example, execution of ISZ 250 increments the contents of location 250 from 7776 to 7777 and then transfers control to the following instruction (JMP 200). In the second pass, execution of ISZ 250 increments the contents of location 250 from 7777 to 0000 and transfers control to the instruction in location 203, skipping over location 202.

CODING FOR ISZ LOOP

| Location | Content |
|---|---|
| 200 | TAD 275 |
| 201 | ISZ 250 |
| 202 | JMP 200 |
| 203 | DCA 276 |
| . | . |
| . | . |
| . | . |
| 250 | 7776 |
| . | . |
| . | . |
| . | . |
| 275 | 0100 |
| 276 | 0000 |

SEQUENCE OF EXECUTION FOR ISZ LOOP

| | | Content After Instruction Execution | | | |
|---|---|---|---|---|---|
| Location | Content | AC | 250 | 275 | 276 |
| FIRST PASS | | | | | |
| 200 | TAD 275 | 0100 | 7776 | 0100 | 0000 |
| 201 | ISZ 250 | 0100 | 7777 | 0100 | 0000 |
| 202 | JMP 200 | 0100 | 7777 | 0100 | 0000 |
| SECOND PASS | | | | | |
| 200 | TAD 275 | 0200 | 7777 | 0100 | 0000 |
| 201 | ISZ 250 | 0200 | 0000 | 0100 | 0000 |
| 202 | JMP 200 | (Skipped during second pass) | | | |
| 203 | DCA 276 | 0000 | 0000 | 0100 | 0200 |

Figure 2-2. ISZ Instruction Incrementing a Tally

The following points should be kept in mind when using the ISZ instruction:

1. The contents of the AC and link are not disturbed;
2. The original word is replaced in main memory by the incremented value;
3. When using the ISZ for looping a specified number of times, the tally must be set to the negative of the desired number; and
4. The ISZ performs the incrementation first and then checks for a zero result.

**JMS ($4nnn_8$)**

A program written to perform a specific operation often includes sets of instructions which perform intermediate tasks. These intermediate tasks may be finding a square root, or typing a character on a keyboard. Such operations are often performed many times in the running of one program and may be coded as subroutines. To eliminate the need of writing the complete set of instructions each time the operation must be performed, the JMS (jump to subroutine) instruction is used. The JMS instruction stores a pointer address in the first location of the subroutine and transfers control to the second location of the subroutine. After the subroutine is executed, the pointer address identifies the next instruction to be executed. Thus, the programmer has at his disposal a simple means of exiting from the normal flow of his program to perform an intermediate task and a means of return to the correct location upon completion of the task. (This return is accomplished using indirect addressing, which is discussed later in this chapter.) The following example illustrates the action of the JMS instruction.

| Location | Content | |
|---|---|---|
| PROGRAM | | |
| 200 | JMS 350 | (This instruction stores 0201 in location 350 and transfers program control to location 351.) |
| 201 | DCA 270 | (This instruction stores the contents of the AC in location 270 upon return from the subroutine.) |
| . | . | |
| . | . | |
| . | . | |

SUBROUTINE

| | | |
|---|---|---|
| 350 | 0000 | (This location is assumed to have an initial value of 0000; after JMS 350 is executed, it is 0201.) |
| 351 | iii | (First instruction of subroutine) |
| . | . | |
| . | . | |
| . | . | |
| 375 | JMP I 350 | (Last instruction of subroutine) |

The following should be kept in mind when using the JMS:

1. The value of the PC (the address of the JMS instruction +1) is always stored in the first location of the subroutine, replacing the original contents;
2. Program control is always transferred to the location designated by the operand +1 (second location of the subroutine);
3. The normal return from a subroutine is made by using an indirect JMP to the first location of the subroutine (JMP I 350 in the above example); (Indirect addressing, as discussed later in this chapter, effectively transfers control to location 201.);
4. When the results of the subroutine processing are contained in the AC and are to be used in the main program, they must be stored upon return from the subroutine before further calculations are performed. (In the above example, the results of the subroutine processing are stored in location 270.)

## ADDRESSING

When the memory reference instructions were introduced, it was stated that nine bits are allocated to specify the operand (the address referenced by the instruction). The method used to reference a memory location using these nine bits will now be discussed.

### PDP-8 Memory Pages

As previously described, the format of an MRI is three bits (0, 1, and 2) for the operation code and the remaining nine bits the operand. However, a full twelve bits are needed to uniquely address the 4,096 (10,000 octal) locations that are contained in the PDP-8 memory unit. To make the best use of the available nine bits, the PDP-8 utilizes a logical division of memory into blocks (pages) of $200_8$ locations each, as shown in the following table.

| Page | Memory Locations | Page | Memory Locations |
|---|---|---|---|
| 0 | 0-177 | 20 | 4000-4177 |
| 1 | 200-377 | 21 | 4200-4377 |
| 2 | 400-577 | 22 | 4400-4577 |
| 3 | 600-777 | 23 | 4600-4777 |
| 4 | 1000-1177 | 24 | 5000-5177 |
| 5 | 1200-1377 | 25 | 5200-5377 |
| 6 | 1400-1577 | 26 | 5400-5577 |
| 7 | 1600-1777 | 27 | 5600-5777 |
| 10 | 2000-2177 | 30 | 6000-6177 |
| 11 | 2200-2377 | 31 | 6200-6377 |
| 12 | 2400-2577 | 32 | 6400-6577 |
| 13 | 2600-2777 | 33 | 6600-6777 |
| 14 | 3000-3177 | 34 | 7000-7177 |
| 15 | 3200-3377 | 35 | 7200-7377 |
| 16 | 3400-3577 | 36 | 7400-7577 |
| 17 | 3600-3777 | 37 | 7600-7777 |

Since there are $200_8$ locations on a page and seven bits can represent $200_8$ different numbers, seven bits (5 through 11 of the MRI) are used to specify the page address. Before discussing the use of the page addressing convention by an MRI, it should be emphasized that memory does not contain any physical page separations. The computer recognizes only absolute addresses and does not know what page it is on, or when it enters a different page. But, as will be seen, page addressing allows the programmer to reference all of the $4{,}096_{10}$ locations of memory using only the nine available bits of an MRI. The format of an MRI is shown in Figure 2-3.

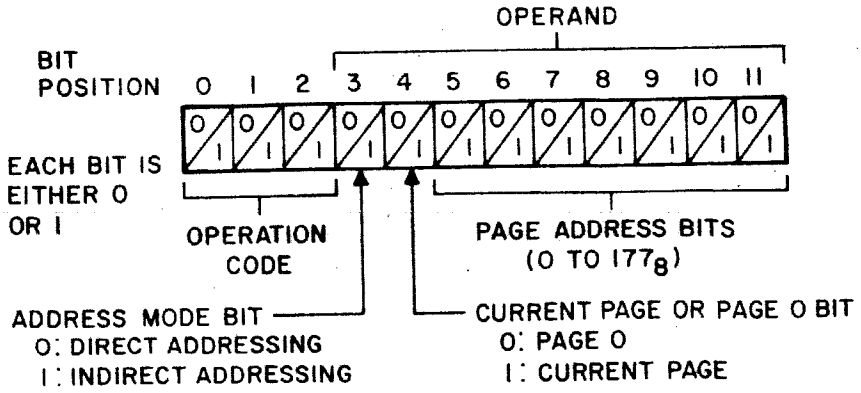



Figure 2-3. Format of a Memory Reference Instruction

As previously stated, bits 0 through 2 are the operation code for the MRI. Bits 5 through 11 identify a specific location on a given page, but they do not identify the page itself. The page is specified by bit 4, often called the *current page or page 0 bit*. If bit 4 is a 0, the page address is interpreted as a location on page 0. If bit 4 is a 1, the page address specified is interpreted to be on the current page (the page on which the MRI itself is stored). For example, if bits 5 through 11 represent $123_8$ and bit 4 is a 0, the location referenced is absolute address $123_8$. However, if bit 4 is a 1 and the current instruction is in a core memory location whose absolute address is between $4{,}600_8$ and $4{,}777_8$, the page address $123_8$ designates the absolute address $4{,}723_8$. Note that, as shown in the following example, this characteristic of page addressing results in the octal coding for two TAD instructions on different memory pages being identical when their operands reference the same relative location (page address) on their respective pages.

| Location | Content | | Explanation |
|---|---|---|---|
| | Mnemonic | Octal | |
| 200 | TAD 250 | 1250 | TAD 250 and TAD 450 both mean add the contents of location 50 on the current page (bit 4 = 1) to the accumulator. |
| | . | | |
| | . | | |
| 400 | TAD 450 | 1250 | |

Except when it is on page 0, a memory reference instruction can reference $400_8$ locations directly, namely those $200_8$ locations on the page containing the instruction itself and the $200_8$ locations on page 0, which can be addressed from any memory location.

NOTE: If an MRI is stored in one of the first $200_8$ memory locations (0 to $177_8$), current page *is* page 0; therefore, only locations 0 to $177_8$ are directly addressable.

**Indirect Addressing**

In the preceding section, the method of directly addressing $400_8$ memory locations by an MRI was described—namely those on page 0 and those on the current page. This section describes the method for addressing the other $7400_8$ memory locations. Bit 3 of an MRI, shown in Figure 2-3 but not discussed in the preceding section, designates the address mode. When bit 3 is a 0, the operand is a direct address. When bit 3 is a 1, the operand is an indirect address. An indirect address (pointer address) identifies the location that contains the desired address (effective address). To address a location that is not directly addressable, the absolute address of the desired location is stored in one of the $400_8$ directly addressable locations (pointer address); the pointer address is written as the operand of the MRI; and the letter I is written

between the mnemonic and the operand. (During assembly, the presence of the I results in bit 3 of the MRI being set to 1.) Upon execution, the MRI will operate on the contents of the location identified by the address contained in the pointer location.

The two examples in Figure 2-4 illustrate the difference between direct addressing and indirect addressing. The first example shows a TAD instruction that uses direct addressing to get data stored on page 0 in location 50; the second is a TAD instruction that uses indirect addressing, with a pointer on page 0 in location 50, to obtain data stored in location 1275. (When references are made to them from various pages, constants and pointer addresses can be stored on page 0 to avoid the necessity of storing them on each applicable page.) The octal value 1050, in the first example, represents direct addressing (bit 3 = 0); the octal value 1450, in the second example, represents indirect addressing (bit 3 = 1). Both examples assume that the accumulator has previously been cleared.

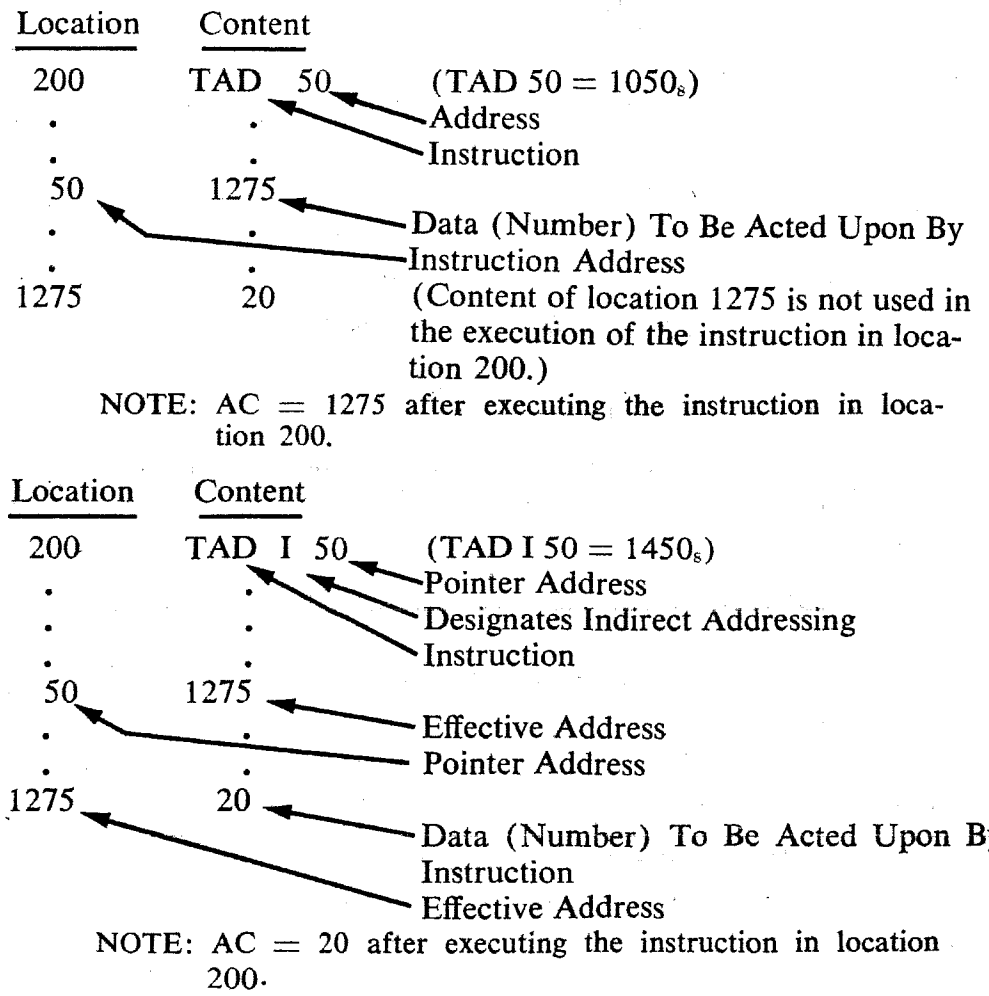
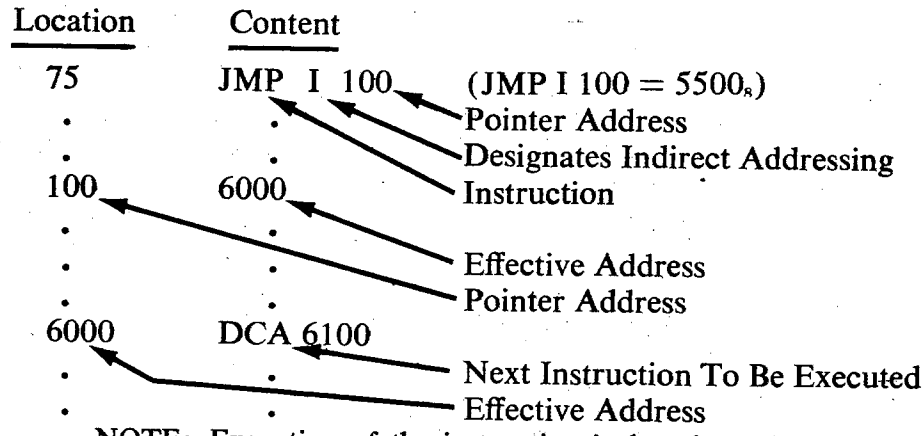



Figure 2-4. Comparison of Direct and Indirect Addressing

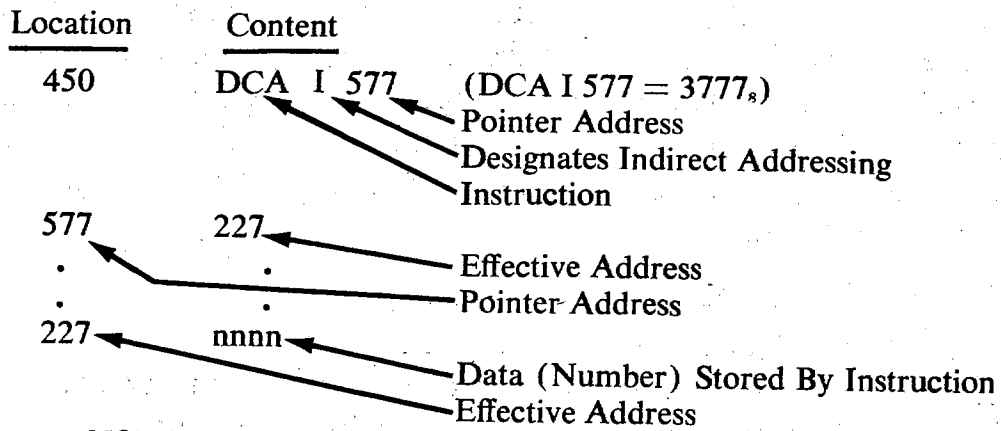
The following three examples illustrate some additional ways in which indirect addressing can be used. As shown in example 1, indirect addressing makes it possible to transfer program control off page 0 (to any desired memory location). (Similarly, indirect addressing makes it possible for other memory reference instructions to address any of the $4{,}096_{10}$ memory locations.) Example 2 shows a DCA instruction that uses indirect addressing with a pointer on the current page. The pointer in this case designates a location off the current page (location 227) in which the data is to be stored. (A pointer address is normally stored on the current page when all references to the designated location are from the current page.) Indirect addressing provides the means for returning to a main program from a subroutine, as shown in example 3. Indirect addressing is also effectively used in manipulating tables of data as described and illustrated in conjunction with autoindexing in Chapter 3.

EXAMPLE 1




NOTE: Execution of the instruction in location 75 causes program control to be transferred to location 6000, and the next instruction to be executed is the DCA 6100 instruction.

EXAMPLE 2




NOTE: Execution of the instruction in location 450 causes the contents of the accumulator to be stored in location 227.

EXAMPLE 3

| Location | Content | |
|---|---|---|
| 207 | JMS I 70 | (JMS I 70 = $4470_8$) |
| 210 | TAD 250 | (The next instruction to be executed upon return from the subroutine.) |
| . | . | |
| . | . | |
| 70 | 2000 | (Starting address of the subroutine stored here.) |
| . | . | |
| . | . | |
| 2000 | aaaa | (Return address stored here by JMS instruction.) |
| 2001 | iii | (First instruction of subroutine.) |
| . | . | |
| . | . | |
| 2077 | JMP I 2000 | (Last instruction of subroutine.) |

NOTES: 1. Execution of the instruction in location 207 causes the address 210 to be stored in location 2000 and the instruction in location 2001 to be executed next. Execution of the subroutine proceeds until the last instruction (JMP I 2000) causes control to be transferred back to the main program, continuing with the execution of the instruction stored in location 210.

2. A JMS instruction that uses indirect addressing is useful when the subroutine is too large to store on the current page.

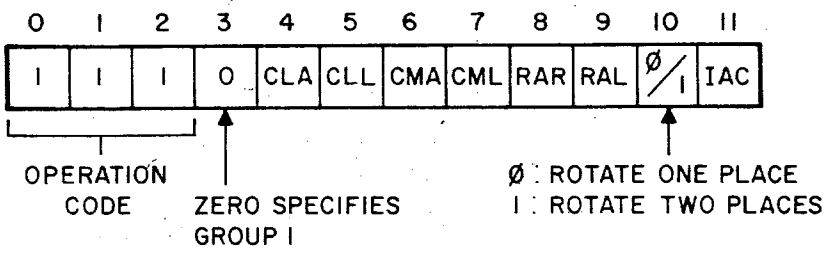
3. Storing the pointer address on page 0 enables instructions on various pages to have access to the subroutine.

## OPERATE MICROINSTRUCTIONS

The operate instructions (octal operation code = 7) allow the programmer to manipulate and/or test the data that is located in the accumulator and link bit. A large number of different instructions are possible with one operation code because the operand bits are not needed to specify an address as they are in an MRI and can be used to specify different instructions. The operate instructions are separated into two groups: Group 1, which contains manipulation instructions, and Group 2, which is primarily concerned with testing operations. Group 1 instructions are discussed first.

### Group 1 Microinstructions

The Group 1 microinstructions manipulate the contents of the accumulator and link. These instructions are microprogrammable; that is, they can be combined to perform specialized operations with other Group 1 instructions. Microprogramming is discussed later in this chapter.




The preceding diagram illustrates the manner in which a PDP-8 instruction word is interpreted when it is used to represent a Group 1 operate microinstruction. As previously mentioned, $7_8$ is the operation code for operate microinstructions; therefore, bits 0 through 2 are all 1's. Since a reference to core memory is not necessary for the operation of microinstructions, bits 3 through 11 are not used to reference an address. Bit 3 contains a 0 to signify that this is a Group 1 instruction, and the remaining bits are used to specify the operations to be performed by the instruction. The operation of each individual instruction specified by these bits is described below.

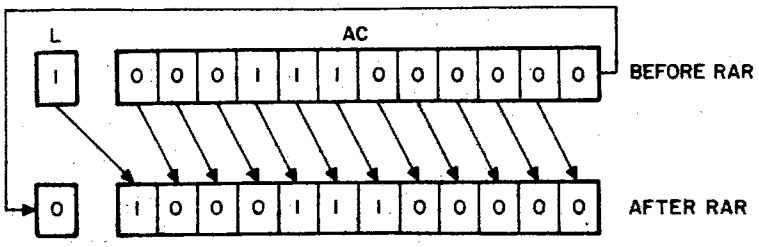
CLA *Clear the accumulator.* If bit 4 is a 1, the instruction sets the accumulator to all zeroes.

CLL *Clear the link.* If bit 5 is a 1, the link bit is set to 0.

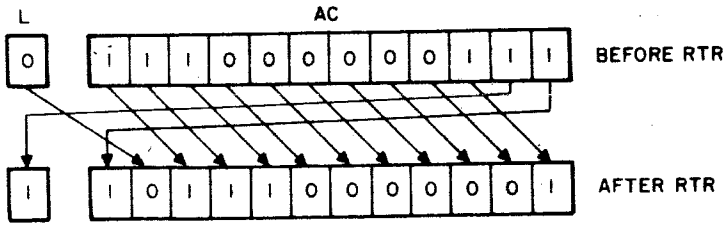
CMA *Complement the accumulator.* If bit 6 is a 1, the accumulator is set to the 1's complement of its original value; that is, all 1's become 0's, and all 0's become 1's.

CML *Complement the link.* If bit 7 is a 1, the state of the link bit is reversed.

RAR *Rotate the accumulator and link right.* If bit 8 is a 1 and bit 10 is a 0, the instruction treats the AC and L as a closed loop and shifts all bits in the loop one position to the right. This operation is illustrated by the following diagram.




RTR *Rotate the accumulator and link twice right.* If bit 8 is a 1 and bit 10 is also a 1, a shift of two places to the right is executed. Both the RAR and RTR instructions use what is commonly called a circular shift, meaning that any bit rotated off one end of the accumulator will reappear at the other end. This operation is illustrated below.




RAL *Rotate the accumulator and link left.* If bit 9 is a 1 and bit 10 is a 0, this instruction treats the AC and L as a closed loop and shifts all bits in the loop one position to the left, performing a circular shift to the left.

RTL *Rotate the accumulator and link twice left.* If bit 9 is a 1 and bit 10 is a 1 also, the instruction rotates each bit two positions to the left. (The RAL and RTL microinstructions shift the bits in the reverse direction of that directed by the RAR and RTR microinstructions.)

IAC *Increment the accumulator.* When bit 11 is a 1, the contents of the AC is increased by 1.

NOP *No operation.* If bits 0 through 2 contain operation code $7_8$, and the remaining bits contain zeros, no operation is performed and program control is transferred to the next instruction in sequence.

A summary of Group 1 instructions, including their octal forms, is given below.

| Mnemonic[1] | Octal[2] | Operation | Sequence[3] |
|---|---|---|---|
| NOP | 7000 | No operation | — |
| CLA | 7200 | Clear AC | 1 |
| CLL | 7100 | Clear link bit | 1 |
| CMA | 7040 | Complement AC | 2 |
| CML | 7020 | Complement link bit | 2 |
| RAR | 7010 | Rotate AC and L right one position | 4 |
| RAL | 7004 | Rotate AC and L left one position | 4 |
| RTR | 7012 | Rotate AC and L right two positions | 4 |
| RTL | 7006 | Rotate AC and L left two positions | 4 |
| IAC | 7001 | Increment AC | 3 |

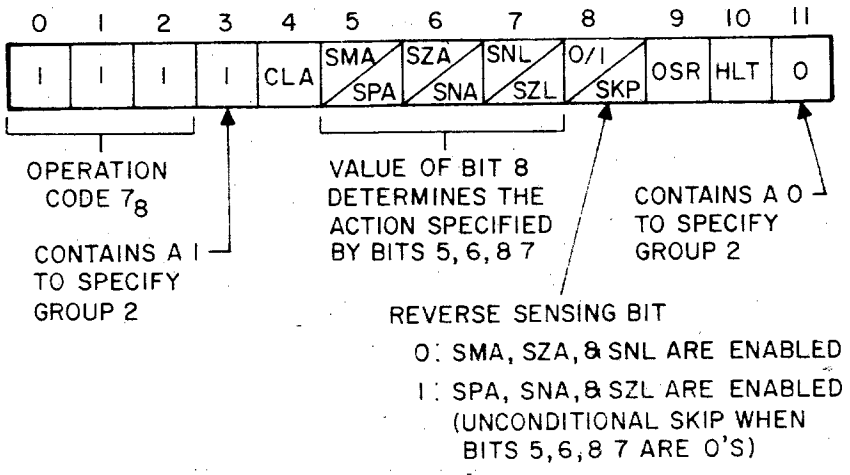
1 Mnemonic code is meaningful to and translated by an assembler into binary code.

2 Octal numbers conveniently represent binary instructions.

3 Sequence numbers indicate the order in which the operations are performed by the PDP-8/I and PDP-8/L (sequence 1 operations are performed first, sequence 2 operations are performed next, etc.).

## Group 2 Microinstructions

Group 2 operate microinstructions are often referred to as the "skip microinstructions" because they enable the programmer to perform tests on the accumulator and link and to skip the next instruction depending upon the results of the test. They are usually followed in a program by a JMP (or possibly a JMS) instruction. A skip instruction causes the computer to check for a specific condition, and, if it is present, to skip the next instruction. If the condition were not present, the next instruction would be executed.




The available instructions are selected by bit assignment as shown in the above diagram. The operation of each individual instruction specified by these bits is described below.

CLA *Clear the accumulator.* If bit 4 is a 1, the instruction sets the accumulator to all zeros.

SMA *Skip on minus accumulator.* If bit 5 is a 1 and bit 8 is a 0, the next instruction is skipped if the accumulator is less than zero.

SPA *Skip on positive accumulator.* If bit 5 is a 1 and bit 8 is a 1, the next instruction is skipped if the accumulator is greater than or equal to zero.

SZA *Skip on zero accumulator.* If bit 6 is a 1 and bit 8 is a 0, the next instruction is skipped if the accumulator is zero.

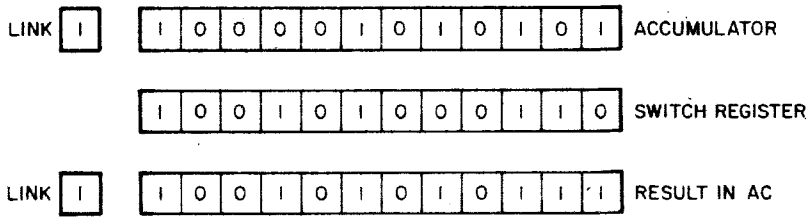
SNA *Skip on nonzero accumulator.* If bit 6 is a 1 and bit 8 is a 1 also, the next instruction is skipped if the accumulator is not zero.

SNL *Skip on nonzero link.* If bit 7 is a 1 and bit 8 is a 0, the next instruction is skipped when the link bit is a 1.

SZL *Skip on zero link.* If bit 7 is a 1 and bit 8 is a 1, the next instruction is skipped when the link bit is a 0.

SKP *Unconditional skip.* If bit 8 is a 1 and bit 5, 6 and 7 are all zeros, the next instruction is skipped. (Bit 8 is a reverse sensing bit when bits 5, 6 or 7 are used—see SMA, SPA, SZA, SNA, SNL, and SZL above.)

OSR *Inclusive OR of switch register with AC.* If bit 9 is a 1, an inclusive OR operation is performed between the content of the accumulator and the console switch register. The result is left in the accumulator and the original content of the accumulator is destroyed. In short, the inclusive OR operation consists of the comparison of the corresponding bit positions of the two numbers and the insertion of a 1 in the result if a 1 appears in the corresponding bit position in *either* number. See Chapter 1 for further discussion. The action of the instruction is illustrated below.




HLT *Halt.* If bit 10 is a 1, the computer will stop at the conclusion of the current machine cycle.

A summary of Group 2 instructions, including their octal representation, is given in the following table.

| Mnemonic | Octal | Operation | Sequence |
|---|---|---|---|
| CLA | 7600 | Clear the accumulator | 2 |
| SMA | 7500 | Skip on minus accumulator | 1 |
| SPA | 7510 | Skip on positive accumulator (or AC = 0) | 1 |
| SZA | 7440 | Skip on zero accumulator | 1 |
| SNA | 7450 | Skip on nonzero accumulator | 1 |
| SNL | 7420 | Skip on nonzero link | 1 |
| SZL | 7430 | Skip on zero link | 1 |
| SKP | 7410 | Skip unconditionally | 1 |
| OSR | 7404 | Inclusive OR, switch register with AC | 3 |
| HLT | 7402 | Halts the program | 3 |

## MICROPROGRAMMING

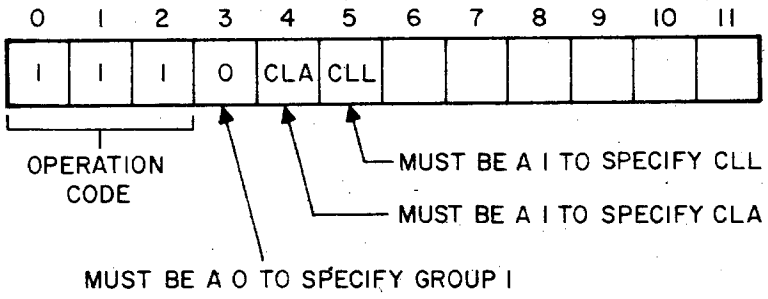
Because PDP-8 instructions of Group 1 and Group 2 are determined by bit assignment, these instructions may be combined, or microprogrammed, to form new instructions enabling the computer to do more operations in less time.

### Combining Microinstructions

The programmer should make certain that the program clears the accumulator and link before any arithmetic operations are performed. To perform this task, the program might include the following instructions (given in both octal and mnemonic form).

| | |
|---|---|
| CLA | 7200 (octal) |
| CLL | 7100 (octal) |

However, when the Group 1 instruction format is analyzed, the following is observed.




Since the CLA and the CLL instructions occupy separate bit positions, they may be expressed *in the same instruction,* thus combining the two operations into one instruction. This instruction would be written as follows.

| | |
|---|---|
| CLA CLL | 7300 (octal) |

In this manner, many operate microinstructions can be combined making the execution of the program much more efficient. The assembler for the PDP-8 will combine the instructions properly when they are written as above, that is, on the same coding line, and separated by a space.

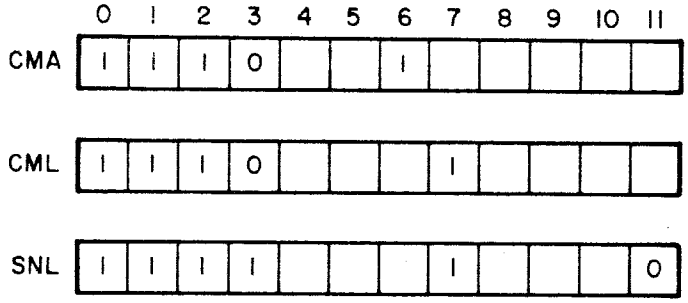
### Illegal Combinations

Microprogramming, although very efficient, can also be troublesome for the new programmer. There are many violations of coding which the assembler will not accept.

One rule to remember is: "If you can't code it, the computer can't do it." In other words, the programmer could write a string of mnemonic microinstructions, but unless these microinstructions can be coded correctly in octal representation, they cannot be performed. To illustrate this fact, suppose the programmer would like to complement the accumulator (CMA), complement the link (CML), and then skip on a nonzero link (SNL). He could write the following.

```
CMA CML SNL
```

These instructions require the following bit assignments.




The three microinstructions cannot be combined in one instruction because bit 3 is required to be a 0 and a 1 simultaneously. Therefore, no instructions may be used which combine Group 1 and Group 2 microinstructions because bit 3 usage is not compatible. The CMA and CML can, however, be combined because their bit assignments are compatible. The combination would be as follows.

```
CMA CML          7060 (octal)
```

To perform the original set of three operations, two instructions are needed.

```
CMA CML          7060 (octal)
SNL              7420 (octal)
```

Because Group 1 and Group 2 microinstructions cannot be combined, the commonly used microinstruction CLA is a member of both groups. Clearing the AC is often required in a program and it is very convenient to be able to microprogram the CLA with the members of both groups.

The problem of bit assignment also arises when some instructions within a group are combined. For example, in Group 1 the rotate instructions specify the number of places to be rotated by the state of bit 10. If bit 10 is a 0, rotate one place; if bit 10 is a 1, rotate two places. Thus, the instruction RAL can not be combined with RTL because bit 10 would be required to have two different values at once. If the pro-

grammer wishes to rotate right three places, he must use two separate instructions.

| | |
|---|---|
| RAR | 7010 (octal) |
| RTR | 7012 (octal) |

Although he can write the instruction "RAR RTR", it cannot be correctly converted to octal by the assembler because of the conflict in bit 10; therefore, it is illegal.

**Combining Skip Microinstructions**

Group 2 operate microinstructions use bit 8 to determine the instruction specified by bits 5, 6, and 7 as previously described. If bit 8 is a 0, the instructions SMA, SZA, and SNL are specified. If bit 8 is a 1, the instructions SPA, SNA, and SZL are specified. Thus, SMA cannot be combined with SZL because of the opposite values of bit 8. The skip condition for combined microinstructions is established by the skip conditions of the individual mcroinstructions in accordance with the rules for logic operations (see "Logic Primer" in Chapter 1).

*OR* GROUP—SMA *OR* SZA *OR* SNL

If bit 8 is a 0, the instruction skips on the logical OR of the conditions specified by the separate microinstructions. The next instruction is skipped if *any* of the stated conditions exist. For example, the combined microinstruction SMA SNL will skip under the following conditions:

1. The accumulator is negative, the link is zero.
2. The link is nonzero, the accumulator is not negative.
3. The accumulator is negative and the link is nonzero.

(It will not skip if all conditions fail.) This manner of combining the test conditions is described as the logical OR of the conditions.

*AND* GROUP—SPA *AND* SNA *AND* SZL

A value of bit 8 = 1 specifies the group of microinstructions SPA, SNA, and SZL which combine to form instructions which act according to the logical AND of the conditions. In other words, the next instruction is skipped only if *all* conditions are satisfied. For example, the instruction SPA SZL will cause a skip of the next instruction only if the accumulator is positive and the link is zero. (It will not skip if either of the conditions fail.)

NOTES:
1. The programmer is not able to specify the manner of combination. The SMA, SZA, SNL conditions are always combined by the logical OR, and the SPA, SNA, SZL conditions are always joined by a logical AND.
2. Since the SPA microinstruction will skip on either a positive or a zero accumulator, to skip on a strictly positive (positive, nonzero) accumulator the combined microinstruction SPA SNA is used.

**Order of Execution of Combined Microinstructions**

The combined microinstructions are performed by the computer in a very definite sequence. When written separately, the order of execution of the instructions is the order in which they are encountered in the program. In writing a combined instruction of Group 1 or Group 2 microinstructions, the order written has no bearing upon the order of execution. This should be clear, because the combined instruction is a 12-bit binary number with certain bits set to a value of 1. The order in which the bits are set to 1 has no bearing on the final execution of the whole binary word.

The definite sequence, however, varies between members of the PDP-8 computer family. The sequence given here applies to the PDP-8/I and PDP-8/L. The applicable information for other members of the PDP-8 family is given in Appendix E. The order of execution for PDP-8/I and PDP-8/L microinstructions is as follows.

GROUP 1

Event 1 CLA, CLL—Clear the accumulator and/or clear the link are the first actions performed. They are effectively performed simultaneously and yet independently.

Event 2 CMA, CML—Complement the accumulator and/or complement the link. These operations are also effectively performed simultaneously and independently.

Event 3 IAC—Increment the accumulator. This operation is performed third allowing a number in the AC to be complemented and then incremented by 1, thereby forming the two's complement, or negative, of the number.

Event 4 RAR, RAL, RTR, RTL—The rotate instructions are performed last in sequence. Because of the bit assignment previously discussed, only one of the four operations may be performed in each combined instruction.

GROUP 2

Event 1 *Either* SMA *or* SZA *or* SNL when bit 8 is a 0. *Both* SPA *and* SNA *and* SZL when bit 8 is a 1. Combined microinstructions specifying a skip are performed first. The microinstructions are *combined* to form one specific test, therefore, skip instructions are effectively performed simultaneously.

Because of bit 8, only members of one skip group may be combined in an instruction.

Event 2 CLA—Clear the accumulator. This instruction is performed second in sequence thus allowing different arithmetic operations to be performed after testing (see Event 1) without the necessity of clearing the accumulator with a separate instruction before some subsequent arithmetic operation.

Event 3 OSR—Inclusive OR between the switch register and the AC. This instruction is performed third in sequence, allowing the AC to be cleared first, and then loaded from the switch register.

Event 4 HLT—The HLT is performed last to allow any other operations to be concluded before the program stops.

This is the order in which all *combined* instructions are performed. In order to perform operations in a different order, the instructions must be written separately as shown in the following example. One might think that the following combined microinstruction would clear the accumulator, perform an inclusive OR between the SR and the AC, and then skip on a nonzero accumulator.

```
CLA OSR SNA
```

However, the instruction would not perform in that proper manner, because the SNA would be executed first. In order to perform the skip last, the instructions must be separated as follows.

```
CLA OSR
SNA
```

Microprogramming requires that the programmer carefully code mnemonics legally so that the instruction does in fact do what he desires it to do. The sequence in which the operations are performed and the legality of combinations is crucial to PDP-8 programming.

The following is a list of commonly used combined microinstructions, some of which have been assigned a separate mnemonic.

| | Instruction | Explanation |
|---|---|---|
| —— | CLA CLL | Clear the accumulator and link. |
| CIA | CMA IAC | Complement and increment the accumulator. (Sets the accumulator equal to its own negative.) |
| LAS | CLA OSR | Load accumulator from switches. (Loads the accumulator with the value of the switch register.) |
| STL | CLL CML | Set the link (to a 1). |
| —— | CLA IAC | Sets the accumulator to a 1. |
| —— | CLA CMA | Sets the accumulator to a −1. |

In summary, the basic rules for combining operate microinstructions are given below.

1. Group 1 and Group 2 microinstructions cannot be combined.
2. Rotate microinstructions (Group 1) cannot be combined with each other.
3. OR Group (SMA, SZA, or SNL) microinstructions cannot be combined with AND Group (SPA, SNA, or SZL) microinstructions.
4. OR Group microinstructions are combined as the logical OR of their respective skip conditions. AND Group microinstructions are combined as the logical AND of their respective skip conditions.
5. Order of execution for combined instructions (PDP-8/I and PDP-8/L only) is listed below.

| Group 1 | Group 2 |
|---|---|
| 1. CLA, CLL | 1. SMA/SZA/SNL or SPA/SNA/SZL |
| 2. CMA, CML | 2. CLA |
| 3. IAC | 3. OSR |
| 4. RAR, RAL, RTR, RTL | 4. HLT |

## EXERCISES

1. The following is a list of current addresses and locations to be addressed. Determine whether the second location should be directly or indirectly addressed from the first.

| Current Address | Location to be Addressed |
|---|---|
| a. 2456 | 2577 |
| b. 1500 | 1600 |
| c. 1230 | 0030 |
| d. 0050 | 0120 |
| e. 6555 | 6400 |
| f. 6555 | 6600 |
| g. 4343 | 4100 |
| h. 2742 | 2450 |
| i. 2507 | 5507 |
| j. 3200 | 3377 |

2. What type of instruction is each of the following (MRI, operate Group 1 or operate Group 2 microinstruction)?

a. 7430
b. 0024
c. 7240
d. 7000
e. 4706
f. 7700

3. Why are each of the following not legal instructions for the PDP-8?

a. 6509 b. 15007 c. 1581 d. 635 e. 7778

4. What is the effect of each of the following octal instructions?

| | Octal | Mnemonic | Operation |
|---|---|---|---|
| a. | 0000 | | |
| b. | 4010 | | |
| c. | 2300 | | |
| d. | 1777 | | |
| e. | 3500 | | |
| f. | 5400 | | |
| g. | 1030 | | |
| h. | 2577 | | |
| i. | 5273 | | |
| j. | 3150 | | |

5. Separate the following octal instructions into microinstruction mnemonics.

a. 7260
b. 7112
c. 7440
d. 7632
e. 7550
f. 7007
g. 7770

6. Write the octal representation for each location in the following program. What are the contents of the accumulator and locations 205, 206, and 207 after execution of the program?

| Location | Mnemonic | Octal |
|---|---|---|
| 0200 | CLA | |
| 0201 | TAD 0205 | |
| 0202 | TAD 0206 | |
| 0203 | DCA 0207 | |
| 0204 | HLT | |
| 0205 | 1537 | |
| 0206 | 2241 | |
| 0207 | 0000 | |

7. Write the octal form of the following microinstructions. Identify any illegal combinations.

   a. CLA CLL CMA CML
   b. CLL RTL HLT
   c. SPA CLA
   d. CLA IAC RTL
   e. CLA IAC RAL RTL
   f. SMA SZA CLA
   g. SMA SZL
   h. CLA OSR HLT
   i. CLA OSR IAC
   j. CLA SMA SZA

8. What instructions could be used to perform a skip only if the accumulator is zero *and* the link is nonzero?
9. Why is it not possible to write *one* combined microinstruction that will load the accumulator from the console switch register, and then test that number, skipping on a positive value?
10. Write the following programs.
    a. Program starts in location 0200 and adds 2 and 8. Give both mnemonic and octal representations.
    b. Program beginning in location 400 which interchanges the contents of locations 550 and 551. Give both mnemonic and octal representations.
11. Write programs to add three numbers A, B, and C in the specified locations below and put the result in the given address for the SUM. All programs start in location 200. Give octal and mnemonic coding.

| | A | B | C | SUM |
|---|---|---|---|---|
| a. | 0030 | 0031 | 0032 | 0033 |
| b. | 0300 | 0301 | 0302 | 0303 |
| c. | 3000 | 3001 | 3002 | 3003 |

# chapter 3

# elementary programming techniques

Mastery of the instruction set is the first step in learning to program the PDP-8 family computers. The next step is to learn to use the instruction set to obtain correct results and to obtain them efficiently. This is best done by studying the following programming techniques. Examples, which should further familiarize the reader with the instructions and their uses, are given to illustrate each technique.

The modern digital computer is capable of storing information, performing calculations, making decisions based on the results and arriving at a final solution to a given problem. The computer cannot, however, perform these tasks without direction. Each step which the computer is to perform must first be worked out by the programmer.

The programmer must write a program, which is a list of instructions for the computer to follow to arrive at a solution for a given problem. This list of instructions is based on a computational method, sometimes called an algorithm, to solve the problem. The list of instructions is placed in the computer memory to activate the applicable circuitry so that the computer can process the problem. This chapter describes the procedure to be followed when writing a program to be used on the PDP-8 family of computers.

## PROGRAMMING PHASES

In order to successfully solve a problem with a computer, the programmer proceeds through the five programming phases listed below:

1. Definition of the problem to be solved,
2. Determination of the most feasible solution method,
3. Design and analysis of the solution—flowcharting,
4. Coding the solution in the programming language, and
5. Program checkout.

*The definition of the problem* is not always obvious. A great amount of time and energy can be wasted if the problem is not adequately defined. When the problem is to sum four numbers, the defining phase is obvious. However, when the problem is to monitor and control a performance test for semiconductors, a precise definition of the problem is necessary. The question that must be answered in this phase is: "What precisely is the program to accomplish?"

*Determining the method* to be followed is the second important phase in solving a problem with a computer. There are perhaps an infinite number of methods to solve a problem, and the selection of one method over another is often influenced by the computer system to be used. Having decided upon a method based on the definition of the problem and the capabilities of the computer system, the programmer must develop the method into a workable solution.

The programmer must *design and analyze the solution* by identifying the necessary steps to solve the problems and arranging them in a logical order, thus implementing the method. Flowcharting is a graphical means of representing the logical steps of the solution. The flowcharting technique is effective in providing an overview of the logical flow of a solution, thereby enabling further analysis and evaluation of alternative approaches.

Having designed the problem solution, the programmer begins *coding the solution in the programming language*. This phase is commonly called programming but is actually coding and is only one part of the programming process. When the program has been coded and the program instructions have been stored in the computer memory, the problem can be solved. At this point, however, the programming process is rarely complete. There are very few programs written which initially function as expected. Whenever the program does not work properly, the programmer is forced to begin the fifth step of programming, that of checking out or "debugging" the program.

The *program checkout* phase requires the programmer to methodically retrace the flow of the instructions step-by-step to find any program errors that may exist. The programmer cannot tell a computer: "You know what I mean!", as he might say in daily life. The computer does not know what is meant until it is told, and once given a set of instructions, the computer follows them precisely. If needed instructions are left out or coding is done incorrectly, the results may be surprising. These flaws, or "bugs" as they are often called, must be found and corrected. There are many different approaches to finding bugs in a program; however, the chosen approach must be organized and painstakingly methodical if it is to be successful. Several techniques for debugging programs for the PDP-8 family of computers are described in Chapter 6.

## FLOWCHARTING

A simple problem to add three numbers together is solved in a few, easily determined steps. A programmer could sit at his desk and write out three or four instructions for the computer to solve the problem. However, he probably could have added the same three numbers with paper and pencil in much less time than it took him to write the program. Thus, the problems which the programmer is usually asked to solve are much more complex than the addition of three numbers, because the value of the computer is in the solution of problems which are inconvenient or time consuming by human standards.

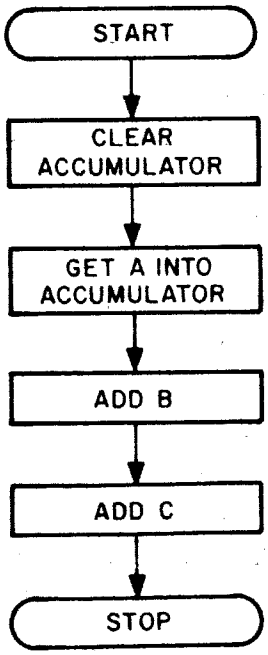
When a more complex problem is to be solved by a computer, the program involves many steps, and writing it often becomes long and confusing. A method for solving a problem which is written in words and mathematical equations is extremely hard to follow, and coding computer instructions from such a document would be equally difficult. A technique called flowcharting is used to simplify the writing of programs. A flowchart is a graphical representation of a given problem, indicating the logical sequence of operations that the computer is to perform. Having a diagram of the logical flow of a program is a tremendous advantage to the programmer when he is determining the method to be used for solving a problem, as well as when he writes the coded program instructions. In addition, the flowchart is often a valuable aid when the programmer checks the written program for errors.

The flowchart is basically a collection of boxes and lines. The boxes indicate what is to be done and the lines indicate the sequence of the boxes. The boxes are of various shapes which represent the action to be performed in the program. Appendix C is a guide to the flowchart symbols and procedures which are used in this text.

The following are examples of flowcharts for specific problems, illustrating methods of attacking problems with a computer program as well as illustrating flowcharting techniques. Example 1 adds three numbers together. Example 2 puts three numbers in increasing order.

**Example 1 — Straight-Line Programming**

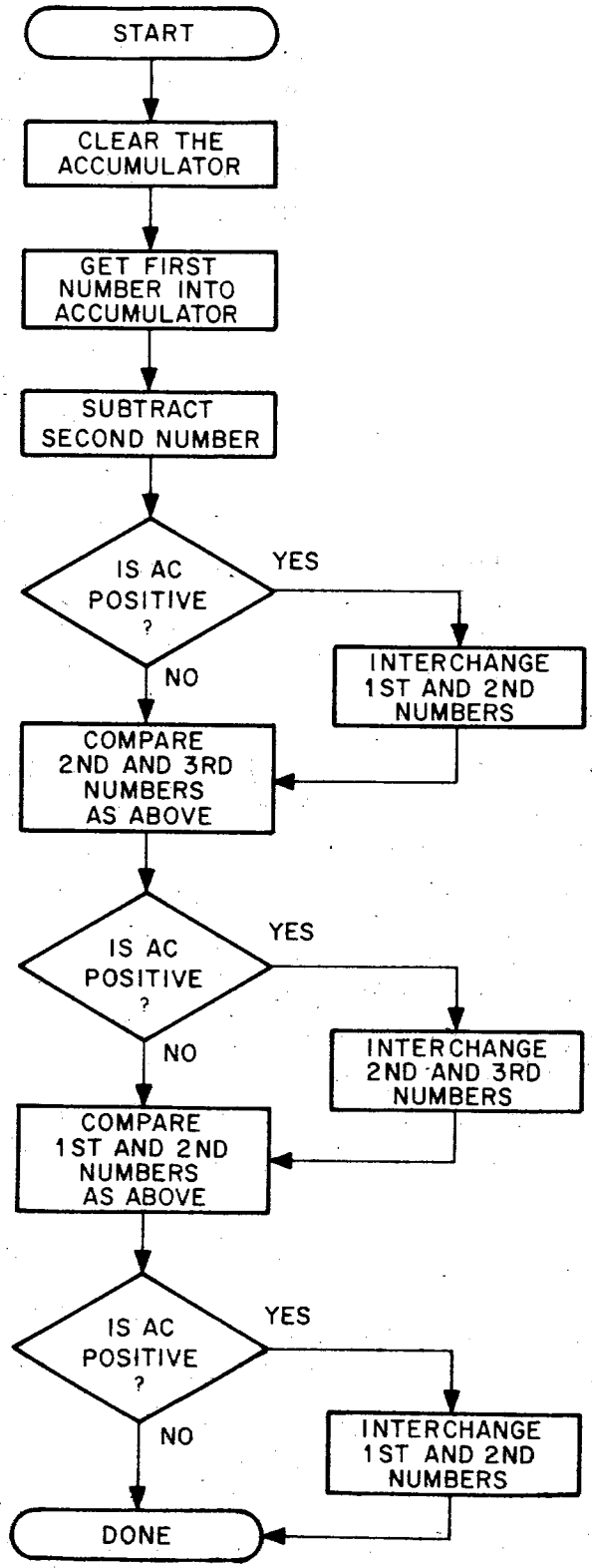
Example 1 is an illustration of straight-line programming. As the flowchart shows, there is a straight-line progression through the processing steps with no change in course. The value of X, which is equal to A+B+C is in the accumulator when the program stops.




Example 1 — Add Three Numbers

**Example 2 — Program Branching**

Example 2 is designed to arrange three numbers in increasing order. The program must branch to interchange numbers that are out of order. (Branching, a common feature of programming, is described in detail later in this chapter.) Note that the arithmetic operations of subtraction are done in the accumulator, which must be cleared initially.




Example 2 — Arrange Three Numbers in Increasing Order

## CODING A PROGRAM

The introduction of an assembler in Chapter 2 enabled the programmer to write a symbolic program using meaningful mnemonic codes rather than the octal representation of the instructions. The programmer could now write mnemonic programs such as the following example, which multiplies $18_{10}$ by $36_{10}$ using successive addition.

```
200/    CLA CLL     (Initialize)
201/    TAD 210     (Set up a Tally
202/    CIA         equal to —18(10) to
203/    DCA 212     count the additions of 36)
204/    TAD 211     (Add 36)
205/    ISZ 212     (Skip if Tally is 0)
206/    JMP 204     (Add another 36 if not done)
207/    HLT         (Stop after 18 times)
210/    0022        (Equal to 18(10))
211/    0044        (Equal to 36(10))
212/    0000        (Holds the tally)
```

Writing the above program was greatly simplified because mnemonic codes were used for the octal instructions. However, writing down the absolute address of each instruction is clearly an inconvenience. If the programmer later adds or deletes instructions, thus altering the location assignments of his program, he has to rewrite those instructions whose operands refer to the altered assignments. If the programmer wishes to move the program to a different section of memory, he must rewrite the program. Since such changes must be made often, especially in large programs, a better means of assigning locations is needed. The assembler provides this better means.

### Location Assignment

As in the previous program example, most programs are written in successive memory locations. If the programmer assigned an absolute location to the first instruction, the assembler could be told to assign the next instructions to the following locations in order. In programming the PDP-8, the initial location is denoted by a precedent asterisk (*). The assembler maintains a *current location counter* by which it assigns successive locations to instructions. The asterisk causes the current location counter to be set to the value following the asterisk. With this improvement incorporated, the previous example appears as shown in the following example.

```
*200
        CLA CLL
        TAD 210
        CIA
        DCA 212
        TAD 211
        ISZ 212
        JMP 204
        HLT
        0022
        0044
        0000
```

NOTE: In this example, CLA CLL is stored in location 200 and the successive instructions are stored in 201, 202, etc.

**Symbolic Addresses**

The programmer does not at the outset know which locations he will use to store constants or the tally. Therefore he must leave blanks after each MRI and come back to fill these in after he has assigned locations to these numbers. In the previous program, he must count the number of locations after the assigned initial address in order to assign the correct values to the MRI operands. Actually this is not necessary, because he may assign symbolic names (a symbol followed by a comma is a *symbolic address*) to the locations to which he must refer, and the assembler will assign address values for him. The assembler maintains a symbol table in which it records the octal values of all symbolic addresses. With symbolic address name tags, the program is as shown below.

```
*200
START,      CLA CLL
            TAD A
            CIA
            DCA TALLY
MULT,       TAD B
            ISZ TALLY
            JMP MULT
            HLT
A,          0022
B,          0044
TALLY,      0000
$
```

NOTES: 1. The dollar sign is the terminal character for the assembler.
2. The comma after a symbol (e.g., START,) indicates to the assembler that the symbol is a symbolic address.

**Symbolic Programming Conventions**

Any sequence of letters (A, B, C . . . , Z) and digits (0, 1 , . . . , 9) beginning with a letter and terminated by a delimiting character (see Table 3-1) is a *symbol.* For example, the mnemonic codes for the PDP-8 instructions are symbols for which the assembler retains octal equivalents in a permanent symbol table.

*User-defined symbols* (stored in the external symbol table) may be of any length; however, only the first six characters are considered, and any additional characters are ignored. (Symbols which are identical in their first six characters are considered identical.)

Any sequence of digits followed by a delimiting character forms a *number.* The assembler will accept numbers which are octal or decimal. The radix is initially set to octal and remains octal unless otherwise specified. The pseudo-instruction DECIMAL may be inserted in the coding to instruct the assembler to interpret all numbers as decimal until the next occurrence of the pseudo-instruction OCTAL in the coding. These pseudo-instructions affect all numbers included in the symbolic program including those preceeded by an * to denote change of origin.

Each symbol or number written in a PDP-8 program must represent a 12-bit binary value in order to be interpreted by the assembler.

The *special characters* in Table 3-1 are used to specify operations to be performed by the assembler upon symbols or numbers in PDP-8 symbolic programs.

The comma after a symbol in a line of coding (e.g., MULT, TAD B) indicates to the assembler that the value of MULT is the address of the location in which the instruction is stored. When an instruction that references MULT (now a *symbolic address*) is encountered, the assembler supplies the correct address value for MULT. (Care must be taken that a symbolic address is never used twice in the same program and that all locations referenced by an MRI are identified somewhere in the program.)

The space and tab are used to delimit a symbol or number. In a combined microinstruction such as CLA CLL, the space delimits the first mnemonic from the second, and the assembler combines the two mnemonic into one instruction. The space and tab similarly delimit the mnemonic from the symbolic address.

TAD A (SPACE) or TAD A (CTRL/TAB)

**Table 3-1 Special Characters for the PDP-8 Symbolic Language**

| Character | | Use |
|---|---|---|
| Keyboard | Name | |
| SPACE | space (nonprinting) | combine symbols or numbers (delimiting) |
| CTRL/TAB | tab (nonprinting) | combine symbols or numbers or format the symbolic tape (delimiting) |
| RETURN | carriage return (nonprinting) | terminate line (delimiting) |
| + | plus | combine symbols or numbers |
| − | minus | combine symbols or numbers |
| , | comma | assign symbolic address |
| = | equals | define parameters |
| * | asterisk | set current location counter |
| ; | semicolon | terminate coding line (delimiting) |
| $ | dollar sign | terminate pass (delimiting) |
| . | point | has value equal to current location counter |
| / | slash | indicates start of a comment |

The carriage return is used to terminate a line of coding. The assembler will also recognize a semicolon as a line terminating character.

```
TAD A
TAD B        is the same as      TAD A; TAD B
```

One of these two characters (i.e., semicolon or carriage return) must be used to separate each line of coding.

The assembler will recognize the arithmetic symbols + and − in conjunction with numbers or symbols, thereby enabling "address arithmetic". For example, the instruction JMP START+1 will cause the computer to execute the instruction in the next location after START. The numbers specified in such instructions are subject to the pseudo-instructions DECIMAL and OCTAL, therefore the number is interpreted as an octal number unless the pseudo-instruction DECIMAL is in effect.

The decimal point, or period, is a character which is interpreted by the assembler as the value of the current location counter. This special symbol can be used as the operand of an instruction; for example, the instruction JMP .−1 causes the computer to execute the preceding instruction.

The equal sign is used to define symbols. This character is used to replace an undefined symbol with the value of a known quantity. For example, the programmer could define a "new instruction" NEGATE

by writing that NEGATE = CIA. The programmer could then write the following instructions to subtract B from A.

```
START,    TAD B
          NEGATE
          TAD A
          HLT
NEGATE = CIA
```

The above coding would be assembled as if the instruction CIA had been included in the actual coding.

The slash is used to insert comments and headings as described later in this chapter.

The dollar sign as previously noted, is a terminal character for the assembler itself. When this character is encountered, the assembler stops accepting input and terminates the assembly pass, as described in Chapter 6.

These characters and conventions will be used throughout the remainder of this text to code programs in PAL III, the symbolic language of the PDP-8 family of computers. Thus, all examples given may be directly punched on paper tape as described in Chapter 4 and assembled by the procedure described in Chapter 6.

## PROGRAMMING ARITHMETIC OPERATIONS

The instructions for the PDP-8 may be used to perform the basic arithmetic operations *within the limits of the machine to represent the necessary numbers*. That is, numbers may be added unless the sum exceeds $4095_{10}$ or $7777_8$. When a sum exceeds the size of the accumulator, *overflow* occurs and incorrect answers result. This condition can usually be detected by checking the value of the link bit.

The following instructions will add numbers and check for overflow, halting the program if the link is 1.

```
ADD,      CLA CLL
          TAD A
          TAD B
          SZL
          HLT
          DCA SUM
            .
            .
            .
```

Since the link is initially cleared in the above example, a link value equal to 1 is an indication that the sum of the contents of locations A and B is too large to be represented by the 12-bit accumulator alone. The computer will halt if the overflow is detected with the actual sum in the combined 13 bits of the accumulator and link.

**Arithmetic Overflow**

Since the PDP-8 regards the numbers 0 through $3777_8$ as positive numbers and the numbers $4000_8$ through $7777_8$ as negative numbers, the addition of two positive numbers could result in either a positive or a negative number depending upon the size of the numbers added. *Arithmetic overflow* is said to occur whenever two positive numbers add to form a negative number, as shown in the following example.

| | |
|---|---|
| $2433_8$ | (a positive number) |
| $+2211_8$ | (a positive number) |
| $4644_8$ | (considered a negative number by the PDP-8) |

Likewise, two negative numbers could be added to yield a positive number as in the following example.

| | | |
|---|---|---|
| | $5275_8$ | $(-2503_8)$ |
| | $+5761_8$ | $(-2017_8)$ |
| Disregarded→1 | 3256 | (considered a positive number by the PDP-8) |

Because of situations like those illustrated in the two preceding examples, the programmer must consider the size of the numbers used in programmed arithmetic operations. If the programmer suspects that overflow may occur in the result of an arithmetic operation, he should follow such an operation by a set of instructions to correct the error or at least to indicate that such an overflow occurred.

The conditions outlined below may be used to test for arithmetic overflow.

| Signs of Numbers Added | Overflow and Link Value |
|---|---|
| Positive + Negative | No overflow possible; link value ignored. |
| Positive + Positive | May result in negative sum; no change in link value. |
| Negative + Negative | May result in positive sum; link is always complemented regardless of the sign of the result. |

The program coding on the next page uses the following facts, assuming an initially cleared link, to quickly determine the sign of the sum of two unknown quantities, A and B.

| Sign of A | Sign of B | Result of Adding only Bit 0 of A to all of B | |
|---|---|---|---|
| | | Link Value | Bit 0 of AC |
| Positive | Negative | 0 | 1 |
| Negative | Positive | 0 | 1 |
| Positive | Positive | 0 | 0 |
| Negative | Negative | 1 | 0 |

```
/CODING TO ADD TWO NUMBERS
/TESTING FOR ARITHMETIC OVERFLOW.
START,      CLA CLL
            TAD A
            AND MASK        /MASK OUT ALL BUT BIT 0.
            TAD B           /ADD B TO BIT 0 OF A.
            SZL             /LINK = 1 IMPLIES BOTH
            JMP BTHNEG      /ARE NEGATIVE.
            RAL             /ROTATE BIT 0 INTO LINK.
            SZL CLA         /BIT 0 = 1 IMPLIES
            JMP OPPSGN      /OPPOSITE SIGNS.
            JMP BTHPOS      /BIT 0 = 0, BOTH POSITIVE.
OPPSGN,     TAD A           /IF A AND B ARE OF OPPOSITE
            TAD B           /SIGNS, THE ADDITION
            DCA SUM         /CANNOT RESULT IN
            HLT             /OVERFLOW.
BTHNEG,     CLA CLL
            TAD A           /IF TWO NEGATIVE NUMBERS
            TAD B           /ADD TO FORM A
            SMA             /POSITIVE NUMBER,
            JMP NEGERR      /JMP TO ERROR ROUTINE.
            DCA SUM         /OTHERWISE, STORE SUM.
            HLT
BTHPOS,     TAD A           /IF TWO POSITIVE
            TAD B           /NUMBERS ADD TO FORM
            SPA             /A NEGATIVE NUMBER, JMP
            JMP POSERR      /TO ERROR ROUTINE.
            DCA SUM         /OTHERWISE, STORE SUM.
            HLT
SUM,        0
MASK,       4000
A,          nnnn            /ANY NUMBERS A AND B
B,          nnnn
POSERR,     .               /ROUTINE TO SIGNAL
            .               /ARITHMETIC OVERFLOW
            .               /OF POSITIVE NUMBERS.
NEGERR,     .               /ROUTINE TO SIGNAL
            .               /ARITHMETIC OVERFLOW
            .               /OF NEGATIVE NUMBERS.
```

**Subtraction**

Subtraction in the PDP-8 family of computers is accomplished by negating the subtrahend (replacing it by its two's complement) and then adding it to the minuend, ignoring the overflow if any. The following example shows the contents of the accumulator for each step of the subtraction process.

| Subtraction Program | Resulting Contents | | |
|---|---|---|---|
| | Link | Accumulator | |
| CLA CLL | 0 | 000 000 000 000 | (0000) |
| TAD B | 0 | 000 000 011 111 | (0037) |
| CMA | 0 | 111 111 100 000 | (7740) |
| IAC | 0 | 111 111 100 001 | (7741) |
| TAD A | 1 | 000 000 000 111 | (0007) |
| . | | | |
| . | | | |
| . | | | |

A, 0046 (000 000 100 110)
B, 0037 (000 000 011 111)

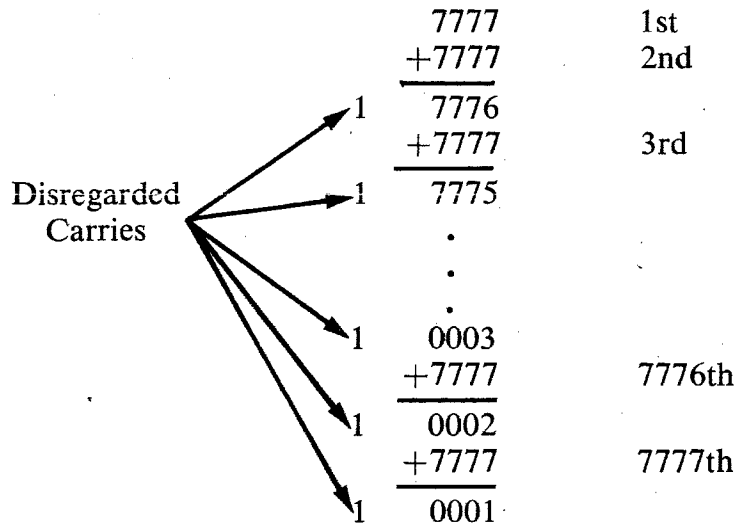
Note that the number to be subtracted (subtrahend) is brought into the accumulator, complemented (1's complement) and incremented by 1 (to form the 2's complement). (The 2's complement could be obtained directly through the one microinstruction CIA.) The number from which A is to be subtracted (minuend) is then added to the accumulator and the difference is obtained.

If A were already in the accumulator from a previous calculation, an alternate procedure could be followed. The number A could be negated first, then B added to it to get B–A. Negating this result yields the same answer because – (B–A) is equal to A–B.

**Multiplication and Division**

A previous example illustrated the method of performing multiplication with the basic PDP-8 instructions, namely by repeated addition. Obviously, multiplication by this method is also subject to the limitation of overflow. The largest positive number which can be directly represented is $2047_{10}$ or $3777_8$.

Multiplication by repeated addition will properly handle positive and negative numbers within the limits of positive or negative arithmetic overflow. For example $7777_8$ is the PDP-8 representation for $-1$. If it is multiplied by itself the answer should be $+1$. In other words, adding $7777_8$ to itself $7777_8$ times should leave (after carries from the most significant bit) the accumulator equal to 1.




Thus, successive addition will work properly as a method of multiplying negative as well as positive numbers in the PDP-8 family of computers.

Similarly, division could be performed by repeated subtraction. This method of division could be used to obtain a quotient and remainder, because only whole numbers are directly represented in the PDP-8. There are, however, much more efficient means of multiplying and dividing numbers in the PDP-8. One means is through the extended arithmetic element (EAE) option, which is described in Chapter 4. Multiplication and division can also be performed through use of the floating point packages, mathematical routines, and interpretive languages of the system software for the PDP-8. These "software" approaches to multiplication and division are described in Chapter 6 of this book.

**Double Precision Arithmetic**

Two memory location (24 bits) are used to express double precision numbers. Using these 24 bits allows the representation of numbers in the range $-8\times10^6$ to $8\times10^6$. The following program adds two double precision numbers, obtaining a double precision result.

Note that if the addition of AL and BL produces a carry, it will appear in the link. The accumulator is cleared by the DCA CL instruction, and the RAL instruction moves the value of the link into the least significant bit position. The values of AH and BH are then added to the carry (if any) and the higher part of the answer is deposited in CH.

This technique may be extended to any order of multiple precision.

```
*200
DUBADD,     CLA CLL
            TAD AL
            TAD BL
            DCA CL
            RAL
            TAD AH
            TAD BH
            DCA CH
            HLT
AH,         1345
AL,         2167
BH,         0312
BL,         0110
CH,         0
CL,         0
$
```

A similar procedure is followed to subtract two double precision numbers. The following program illustrates the technique.

```
*200
DUBSUB,     CLA CLL
            TAD BL
            CIA
            TAD AL
            DCA CL
            RAL
            DCA KEEP
            TAD BH
            CMA
            TAD AH
            TAD KEEP
            DCA CH
            CLL
            HLT
AH,         1345
AL,         2167
BH,         0312
BL,         0110
CH,         0
CL,         0
KEEP,       0
$
```

The location KEEP is used to save the contents of the link while the value of BH was complemented in the accumulator. To form a double precision two's complement number, a double precision one's comple-

ment is formed and the 1 is added to it *once*. Thus, the value of BL is complemented using the CIA instruction, while the value of BH is complemented with the CMA instruction. The CLL instruction is used to clear the link and disregard the carry resulting from using two's complement numbers to perform subtraction.

**Powers of Two**

In the decimal number system, moving the decimal point right (or left) multiplies (or divides) a number by powers of ten. In a similar way, rotating a binary number multiplies (or divides) by powers of two. However, because of the logical connection between the accumulator and the link bit, care must be taken that unwanted digits do not reappear in the accumulator after the passage through the link. Multiplication by powers of two is performed by rotating the accumulator left; division is performed by rotating the accumulator right. Multiplication and division by this method are subject to the limitation of 12-bit numbers (unless double precision is used). That is, significant bits rotated out of the accumulator by multiplication or division are lost and incorrect results are therefore obtained. For example, the following program multiplies a number by 8 ($2^3$).

```
*200
MULT8,      CLA CLL
            TAD NUMBER
            CLL RAL
            CLL RAL
            CLL RAL
            DCA NUMBER
            HLT
NUMBER,     0231
$
```

The program will replace the number $0231_8$ by $2310_8$. Notice that multiplying any number with four significant octal digits (such as $1234_8$) using this program will yield incorrect results.

## WRITING SUBROUTINES

Included in the memory reference instructions, given in Chapter 2, was the instruction JMS (jump to subroutine). This instruction is a modified JMP command which makes return to the point of departure from the main program possible. The JMS instruction automatically stores the location of the next instruction after the JMS in the location to which the program is instructed to jump, thereby enabling a return.

The programmer need only terminate the subroutine with an indirect JMP to the first location of the subroutine in order to return to the next instruction following the JMS instruction. The following simple program illustrates the use of a subroutine to double a number contained in the accumulator.

```
                    (Main Program)
START,   CLA CLL
         TAD N          (Get the number in the AC)
         JMS DOUBLE     (Jump to subroutine to double N)
         DCA TWON       (First instruction after the subrou-
                         tine)
                .
                .
                .
N,       nnnn           (Any number, N)
TWON,    nnnn           (2N will be stored here)


                    (Subroutine)
DOUBLE,  0000
         DCA STORE      (Save value of N)
         TAD STORE      (Get N back in the AC)
         CLL RAL        (Rotate left, multiplying by 2)
         SNL            (Did overflow occur?)
         JMP I DOUBLE
         CLA CLL        (If overflow occurs, display the
         TAD STORE       number to be doubled in the AC
         HLT             and then stop the computer.)
STORE,   0000
$
```

Notice that the first instruction of the subroutine is located in the second location of the subroutine. Any instruction stored in location DOUBLE would be lost when the return address is stored. Also note that the subroutine as it is written must be located on page 0 or current page, because it is directly addressed. (A subroutine is often located on another page and addressed indirectly as the next example demonstrates.)

The following program multiplies a number in the accumulator by a number stored in the location immediately following the JMS instruction.

```
  (Main Program)
*200
START,          CLA CLL
                TAD A
                DCA .+3
                TAD B
                JMS I 30
                0000
                DCA PRDUCT
                .
                .
                .
PRDUCT,         0000
A,              0051
B,              0027
*30
                MULT
  (Subroutine)
*6000
MULT,           0000
                CIA
                DCA MTALLY
                TAD I MULT
                ISZ MTALLY
                JMP .-2
                ISZ MULT
                JMP I MULT
MTALLY,         0000
```

The preceding example illustrates the following important points.

1. The JMS I 30 instruction could be used anywhere in core memory to jump to this subroutine because the pointer word (stored in location 30) is located on page 0 and all pages of memory can reference page 0.

2. The period was used to denote the current location in the instructions DCA .+3 and JMP .—2.

3. Since the result of the subroutine is left in the AC when jumping back to the main program, the next instruction should store the result for future use.

4. The first instruction of the subroutine is in location MULT +1 since the next address in the main program is stored in MULT by the JMS instruction.

5. The first two instructions of the subroutine set the tally with the negative of the number in the AC.

6. The second number to be multiplied is brought into the subroutine by the TAD I MULT instruction since it is stored in the location specified by the address that the JMS instruction automatically stores in the first location of the subroutine. This is a common technique for transferring information into a subroutine.

7. The ISZ MTALLY instruction is used in the subroutine to count the number of additions. The ISZ MULT instruction is used to increment the contents of MULT by one, thereby making the return jump (JMP I MULT) proceed to the next instruction after the location which held the number to be multiplied.

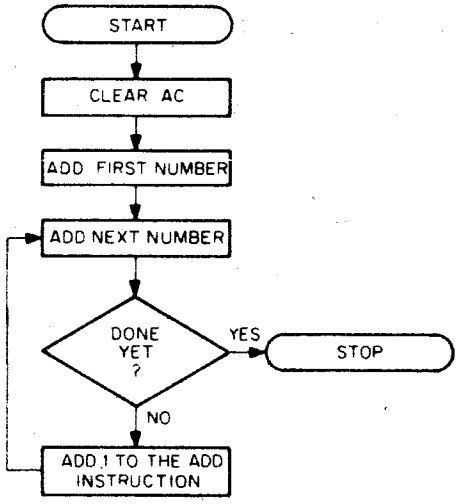
8. An interesting modification of the previous program is achieved by defining a "new operation" MLTPLY by including in the coding the statement MLTPLY=JMS I 30. The assembler would make a replacement such that any time the programmer writes MLTPLY, the computer would perform a jump to the subroutine and return to the program with the product in the AC.

## ADDRESS MODIFICATION

A very powerful tool often used by the programmer is address modification, meaning the inclusion of instructions in a program to modify the operand portion of a memory reference instruction. It is a particularly useful technique when working with large blocks of stored data as illustrated by the two programs that follow.

The first program sums $100_8$ numbers in locations $300_8$ to $377_8$. The program begins in location $200_8$. The block of $100_8$ numbers is summed using only one TAD instruction merely by repeatedly incrementing and performing the instruction.

The second example program moves data between memory pages as well as performing an operation upon the data. The program computes the square of the $200_8$ numbers in locations $4000_8$ to $4177_8$. The program starts in location $200_8$. All numbers to be squared must not exceed $45_{10}$ or the square is too large to be represented in the normal format.




Program

```
*200
START,    CLA CLL
          TAD K100
          CIA
          DCA TALLY
ADD,      TAD 300
          ISZ ADD
          ISZ TALLY
          JMP ADD
          DCA SUM
          HLT
K100,     0100
TALLY,    0000
SUM,      0000
$
```

The second example illustrates the method of using indirect addressing in an address modification situation. It should be noted that in the first example the actual instruction was incremented to perform the modification. In the second example, the modification was done by incrementing the contents of a location which was used for indirect addressing. The second example could be simplified further through use of autoindexing, a feature that will be discussed later.

```
*200
START,      CLA CLL
            TAD K200
            CIA
            DCA TALLY
            TAD K4000
            DCA NUM
            TAD K4200
            DCA RESULT
AGAIN,      TAD I NUM
            JMS SQUARE
            DCA I RESULT
            ISZ RESULT
            ISZ NUM
            ISZ TALLY
            JMP AGAIN
            HLT
K200,       0200
TALLY,      0000
K4000,      4000
NUM,        0000
K4200,      4200
RESULT,     0000
*300
SQUARE,     0000
            DCA STORE
            TAD STORE
            CIA
            DCA COUNT
            TAD STORE
            ISZ COUNT
            JMP .-2
            JMP I SQUARE
STORE,      0000
COUNT,      0000
$
```

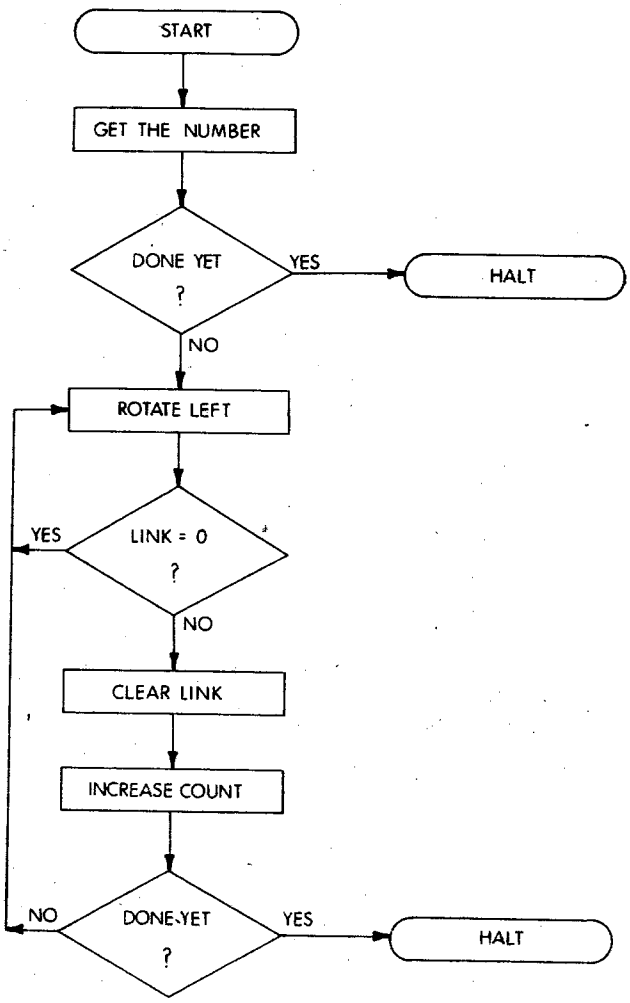
The reader should note that the first eight instructions of the second example are concerned with intializing the program. This intializing enables the stored program to be restarted several times and still operate on the correct locations. If the program had merely incremented locations K4000 and K4200 and utilized those locations for indirect addressing the program would only operate on the correct locations of the first running. On successive runnings the program would be operating on successively higher locations in memory. With the program written as shown however the pointer words are automatically reset. This procedure is often referred to as "housekeeping."

## INSERTING COMMENTS AND HEADINGS

Because programs very seldom are written, used, and then forgotten, the programmer should strive to document his procedure and coding as much as is reasonably possible. There are many instances where changes or corrections must be made by people unfamiliar with a program, or more commonly the original programmer is asked to modify a program months after his original effort. In both cases, the success of the attempt to change the program depends largely upon the documentation provided by the original programmer. A complete and accurate flowchart is the first form of documentation. It is extremely important to document modifications made in the program by incorporating these changes in the flowchart as well.

Many times it is desired to include headings and dates to identify a program within the actual coding of the symbolic program. It is often helpful to add comments to simplify the reading of a symbolic program and to indicate the purpose of any less than obvious instruction. PDP-8 programming allows comments and headings to be inserted simply by preceding any comments with a slash (/).

The following example illustrates the method used to insert comments and headings in a PDP-8 program. It also illustrates the use of a rotate instruction. The program takes a binary word stored in memory and counts the number of non-zero bits. Although the program may have no useful application, it does serve to familiarize the reader with the structure of the accumulator and link bit and the action of a rotate instruction. The flowchart and comments will aid the reader to understand the program.




```
/COUNT THE BINARY ONES PROGRAM
/20 SEPTEMBER 1968
*200
START,      CLA CLL
            DCA  COUNT      /SET COUNT TO 0.
            TAD  I WORD     /GET THE WORD.
            SNA
            HLT             /STOP IF THE WORD IS 0.
ROTATE,     RAL             /ROTATE ONE BIT INTO LINK.
            SNL             /WAS THE BIT = 0?
            JMP  -2         /YES: ROTATE AGAIN.
            CLL             /NO: CLEAR LINK.
            ISZ COUNT       /COUNT THE NUMBER OF 1'S.
            SNA
            HLT             /STOP IF THE WORD IS NOW 0.
            JMP ROTATE
COUNT,      0
WORD,       3000            /ANY 12-BIT NUMBER.
$
```

The following points should be observed in the preceding example.

1. The word was checked to see that it was non-zero to begin with. If this check were not made, a zero word would be rotated endlessly by the remaining instructions in the program.
2. Because a rotate right instruction (RAR) would transfer the bits into the link just as the RAL instruction does, either could be used in the above program. Both instructions use a circular shift of the accumulator and link bits.
3. Because the link bit is rotated into the accumulator by the rotate instructions, the link must be cleared each time a 1 is rotated into it.

## LOOPING A PROGRAM

As many of the examples given have already shown, the use of a program loop, in which a set of instructions is performed repeatedly, is common programming practice. Looping a program is one of the most powerful tools at the programmer's disposal. It enables him to perform similar operations many times using the same instructions, thus saving memory locations because he need not store the same instructions many times. Looping also makes a program more flexible because it is relatively easy to change the number of loops required for differring conditions by resetting a counter. It is good to remember that looping is little more than a jump to an earlier part of the program; however, the jump is usually conditioned upon changing program conditions.

There are basically two methods of creating a program loop. The first method is using an ISZ ($2nnn_8$) instruction to count the number of passes made through the loop. The ISZ is usually followed by a JMP instruction to the beginning of the loop. This technique is very efficient when the required number of passes through the loop can be readily determined.

The second technique is to use the Group 2 Operate Microinstructions to test conditions other than the number of passes which have been made. Using this second technique, the program is required to loop until a specific condition is present in the accumulator or link bit, rather than until a predetermined number of passes are made.

To illustrate the use of an ISZ instruction in a program loop situation, consider the following program which simply sets the contents of all addresses from 2000 to 2777 to zero.

```
*200
CLEAR,    CLA
          TAD CONST
          DCA COUNT       /SET COUNT TO -1000.
          TAD TTABLE
          DCA STABLE      /SET STABLE TO 2000.
          DCA I STABLE    /CLEAR ONE LOCATION.
          ISZ STABLE      /SELECT NEXT LOCATION.
          ISZ COUNT       /IS OPERATION COMPLETE?
          JMP .-3         /NO: REPEAT.
          HLT             /YES: HALT.
CONST,    7000            /2'S COMP OF 1000.
COUNT,    0
TTABLE,   TABLE
STABLE,   0               /POINTER TO TABLE.
*2000
TABLE,    0
$
```

Several points should be carefully noted.

1. The first five instructions initialize the loop, but are not in it. The location COUNT is set to −1000 at the beginning, and 1 is added to it during each passage of the loop. After the 1000th (octal) passage, COUNT goes to zero, and the program skips the JMP instruction, and executes the HLT instruction. On each previous occasion, it executed the JMP instruction.

2. In the list of constants following the HLT instruction, TTABLE contains TABLE, which is in turn defined below as *having the value 2000,* and *containing 0.* Therefore, STABLE contains 2000 initially. In order to understand this point it must be remembered that an asterisk character causes the first location after the asterisk to be set to the value after the asterisk. Therefore, in the previous example CLEAR equals 200 and TABLE equals 2000.

3. ISZ STABLE adds 1 to the contents of location STABLE, forming 2001 on the first pass, 2002 on the second pass, and so on. Since it never reaches zero, it will never skip. This is a very common use. It is said to be *indexing* the addresses from 2000 to 2777. (When using an ISZ instruction in this way, the programmer must be certain that it does not reach 0. Follow the ISZ instruction with a NOP if necessary.)

4. For every ISZ instruction used in a program, there must be two initializing instructions before the loop, and there must be a constant and a counting location in a table of constants. This procedure allows the program to be rerun with the counting locations reset to the correct values.

The following program utilizes a Group 2 skip instruction to create a loop. The program will search all of core memory to find the first occurrence of the octal number 1234.

```
*0
NUMBER,     1234
*200
BEGIN,      CLA CLL
            TAD NUMBER
            CIA
            DCA COMPARE     /STORES MINUS NUMBER.
            DCA ENTRY       /SETS ENTRY TO 0.
REPEAT,     ISZ ENTRY       /INCREASES ENTRY.
            CLA
            TAD I ENTRY     /COMPARISON IS
            TAD COMPARE     /DONE HERE.
            SZA CLA
            JMP REPEAT
            TAD ENTRY
            HLT             /ENTRY IS IN AC.
COMPARE,    0
ENTRY,      0
$
```

The previous example is not very useful perhaps but it is interesting to note that the program will search itself as well as all other core memory locations.

Also notice the following points with regard to the example.

1. The ISZ ENTRY instruction is used to index the locations to be tested. The next instruction (CLA) is unnecessary, thus if ENTRY becomes zero during the course of the program, the program will not be affected. It is very important to protect against an ISZ instruction going to zero and skipping a necessary part of a program, if the ISZ is being used to simply index.

2. The number to be searched for was stored in location 0, and the search starts in location 1. Therefore, the program will find at least one occurrence of the number and will halt after one complete pass through memory if not before.

3. The program could be modified to bound the area of the search. By setting the contents of ENTRY equal to one less than the desired start location and putting the number being searched for in the location following the last location to be searched, the program will search only the designated area of memory.

4. The program could be restarted at location REPEAT in order to find a second occurrence of 1234 after the program had halted with the first occurrence.

## AUTOINDEXING

The PDP-8 family computers have eight special registers in page 0, namely locations 0010 through 0017. Whenever these locations are addressed indirectly by a memory reference instruction, the content of the register is incremented before it is used as the operand of the instruction. These locations can therefore be used in place of an ISZ instruction in an indexing application. Because of this unique action these eight locations are called autoindex registers. It is important to realize that autoindex registers act as any other location when addressed directly. *The autoindexing feature is performed only when the location is addressed indirectly.*

The following example is a modification of the first program example in the preceding section with an autoindex register used in place of the ISZ instruction. (The purpose of the program is to clear memory locations 2000 through 2777.)

Carefully notice the difference between the two examples, especially that TABLE now has to be set to TABLE–1 since this is incremented by the autoindexing register *before* being used for the first time. This

point must be remembered when using an autoindex register. The register increments before the operation takes place, therefore it must always be set to one less than the first value of the addresses to be indexed.

```
*10
INDEX,          0
*200
CLEAR,          CLA
                TAD CONST
                DCA COUNT
                TAD TTABLE
                DCA INDEX
                DCA I INDEX
                ISZ COUNT
                JMP -2
                HLT
CONST,          7000
COUNT,          0
TTABLE,         TABLE-1
*2000
TABLE,          0
$
```

The memory search example of the preceding section could also be simplified using an autoindex register as shown below.

```
*0
NUMBER,         1234
*10
ENTRY,          0
*200
BEGIN,          CLA CLL
                TAD NUMBER
                CIA
                DCA COMPARE
                DCA ENTRY
REPEAT,         TAD I ENTRY
                TAD COMPARE
                SZA CLA
                JMP REPEAT
                TAD ENTRY
                HLT
COMPARE,        0
$
```

Notice that in this case ENTRY originally equals 0 because its content is incremented before being used to obtain data for the comparison.

## PROGRAM DELAYS

Because the development of a computer was primarily sparked by a desire for speed in performing calculations, it seems inconsistent and self-defeating to slow the computer down with program delays. However, there are many occasions when a computer must be told to slow down or to wait for further information. This is because most peripheral equipment, and certainly the human operator, is very much slower than the computer program. A temporary delay may be introduced into the execution of a program when needed by causing the computer to enter one or more futile loops which it must traverse a fixed number of times before jumping out. It is often necessary to have a computer perform a temporary delay while a peripheral device is processing data to be submitted to the computer. The delays can be accurately timed so as not to waste any more computer time than necessary.

The following is a simple delay routine using the ISZ instruction for an inner loop and an outer loop. The reader should remember when analyzing the example that the PDP-8 represents only positive numbers up to $3777_8$ or $2047_{10}$. Therefore, the computer counts up to $2047_{10}$ and then continues to count starting at the next octal number $4000_8$, which the computer interprets as $-2048_{10}$. Successive increments of this number will finally bring the count to zero. Thus, a location could be used to count from 1 up to 0 by using an ISZ instruction.

```
            (main program)
                  .
                  .
                  .
            TAD CONST       /START OF DELAY ROUTINE
            DCA COUNT
            ISZ COUNT1      /INNER
            JMP .-1         /LOOP
            ISZ COUNT
            JMP .-3
CONST,      6030            /SETS DELAY
COUNT,      0
COUNT1,     0
```

The inner loop consists of an ISZ instruction with an execution time on the PDP-8/I of 3.0 microseconds (a microsecond is $10^{-6}$ seconds) and a JMP instruction with an execution time of 1.5 microseconds. Therefore, the inner loop takes 4.5 microseconds for one pass, and each time it is entered the program will traverse it $4096_{10}$ times before leaving. This means that a delay of 18.432 milliseconds (a millisecond is

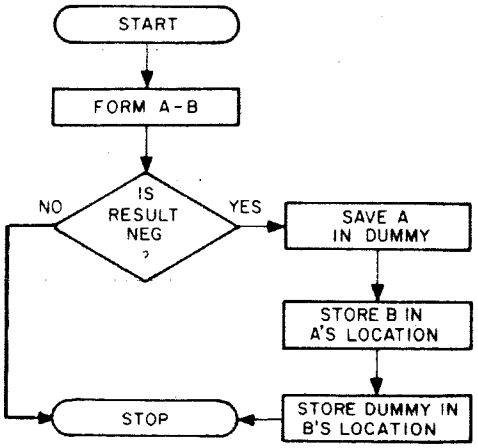
$10^{-3}$ seconds) has occurred. If, as in the example above, the value of CONST is $6030_8$, this loop will be entered $1000_{10}$ times giving a total delay of 18.432 seconds. For any given purpose, a desired delay from milliseconds to seconds can be obtained precisely by varying the values of CONST and the initial value of COUNT1. Similar reasoning can be used to design delays for other members of the PDP-8 family.

A second type of delay, which waits for a device response, is discussed in Chapter 5. This type is not a timed delay but causes the computer to wait until it receives a response from an external device.

## PROGRAM BRANCHING

Very few meaningful programs are written which do not take advantage of the computer's ability to determine the future course the program should follow based upon intermediate results. The procedure of testing a condition and providing alternative paths for the program to travel for each of the different results possible is called branching a program. The Group 2 microinstructions presented in Chapter 2 are most often used for this purpose. The ISZ instruction also provides a branch in a program. These instructions are often referred to as conditional skip instructions. The ISZ instruction operates upon the contents of a memory location, while the Group 2 microinstructions test the contents of the AC and L.

A typical example of a conditional skip would be a program to compare A and B and to reverse their order if B is larger than A.



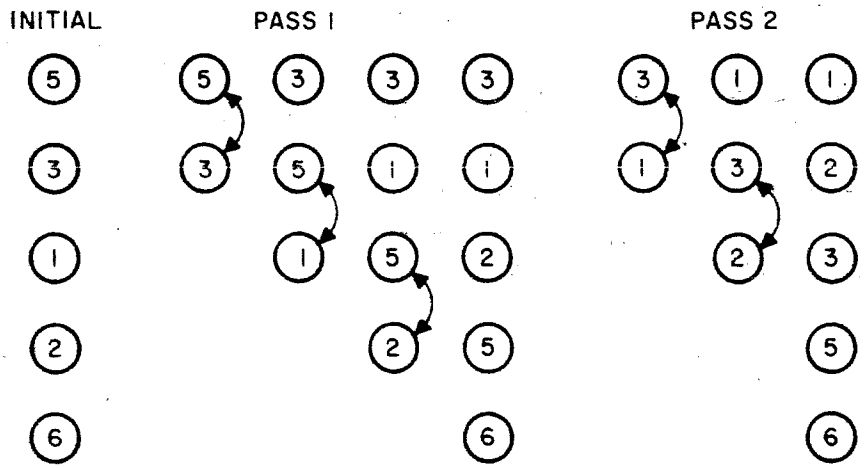
```
*200
TEST,   CLA CLL
        TAD B           /SUBTRACT B
        CIA             /FROM A
        TAD A           /HERE.
        SMA CLA
        HLT             /STOP HERE IF A IS GREATER
                         OR EQUAL

        TAD A           /THE REMAINDER OF
        DCA DUMMY       /THE PROGRAM
        TAD B           /DOES THE SWITCH.
        DCA A
        TAD DUMMY
        DCA B
        HLT
A,      1234            /SUBSTITUTE ANY POSITIVE
B,      2460            /VALUES FOR A AND B.
DUMMY,  0
$
```

If A is less than B, their difference will be negative and the HALT will be skipped. The program will proceed to reverse the order of A and B. If A is greater than or equal to B, the program will halt.

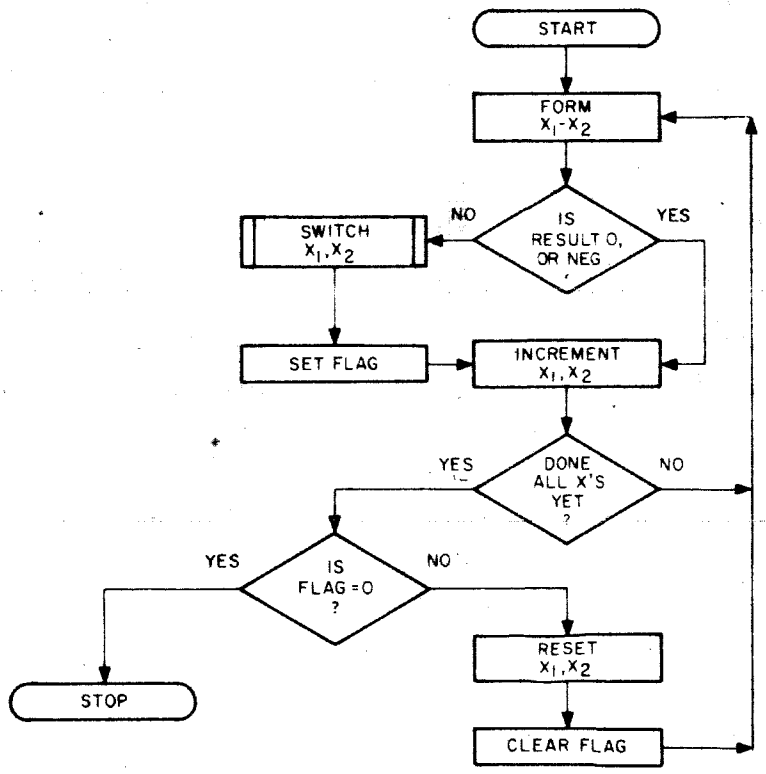
The concept illustrated by the above example can be included in a larger program that will take a set of elements and arrange them in increasing order. The following important concepts should be learned from the example.

1. The program contains two loops to perform the sort. The inner loop starts at TEST and is traversed $20_8$ times to switch adjacent elements of the set. The outer loop begins at START and is re-entered until the elements are in the correct order.
2. A "software flag" was created to signal the program that a switch has been performed on the last pass. The flag is checked upon every exit from the inner loop. If the flag is non-zero (equal to $-1$), a reverse was performed on the last pass and the next pass is started. If the flag is zero, the set is now in order and the program halts.
3. The flag is set to zero on each pass through the outer loop by depositing AC=0 in it. It can only be set to a non-zero value by a pass through the REVERSE subroutine.
4. The TALLY had to be set to $-$(AMOUNT) $+1$ or in this case to $-20_8$ because if the set contains n elements there are n$-1$ comparisons between an element and the immediately succeeding element, thus, in this case, TALLY$=-20_8$.

5. The following sort of five elements illustrates the technique used in the program.




In performing the above sort, the program makes three passes. On the third pass through the table of data, the flag is not raised; therefore, the program stops.



```
*200
START,  CLA CLL
        TAD AMOUNT      /THESE INSTRUCTIONS SET
        CIA             /UP A TALLY EQUAL TO
        IAC             /AMOUNT -1 TO COUNT THE
        DCA TALLY       /PASSES THRU TEST LOOP.
        DCA FLAG        /CLEARS FLAG BEFORE EACH PASS
        TAD BEGIN       /THESE INSTRUCTIONS
        DCA X1          /SET THE POINTERS
        TAD BEGIN       /X1 AND X2 TO THE
        IAC             /PROPER VALUES
        DCA X2          /INITIALLY.
TEST,   TAD I X2        /SUBTRACTION FOR THE
        CIA             /TEST IS
        TAD I X1        /DONE HERE.
        SPA SNA CLA
        SKP
        JMS REVERSE     /DO SWITCH IF AC IS POSITIVE.
        ISZ X1          /SET UP THE X'S FOR
        ISZ X2          /THE NEXT PASS.
        ISZ TALLY       /HAVE ALL X'S BEEN TESTED?
        JMP TEST        /NO: KEEP TESTING.
        TAD FLAG        /YES: WERE ANY SWITCHES
        SZA             /DONE ON THE LAST PASS?
        JMP START       /YES: GO THRU PROGRAM AGAIN.
        HLT             /NO: STOP, TABLE IS IN ORDER.
AMOUNT, 21
TALLY,  0000
BEGIN,  2000
X1,     0000
X2,     0000
FLAG,   0000
HOLD,   0000
REVERSE,0000            /SUBROUTINE TO SWITCH X'S
        TAD I X1
        DCA HOLD
        TAD I X2
        DCA I X1
        TAD HOLD
        DCA I X2
        CLA CLL CMA     /SETS AC EQUAL TO -1.
        DCA FLAG        /SET FLAG=-1 ON A SWITCH.
        JMP I REVERSE
$
```

6. This program can perform a sorting for any specified block of data merely by specifying the octal number of entries to be sorted in the location AMOUNT and by specifying the beginning address of the block in BEGIN. The data to be sorted must be placed in consecutive memory locations.

## Exercises

1. Write a subroutine SUB to subtract the number in the AC from the number in the location after the JMS instruction that calls the subroutine. Return to the main program with the difference in the AC. Use a flowchart and comments to document the procedure.

2. Write two programs to put 0 into memory location 2000, 1 into 2001, 2 into 2002, etc., up to $777_8$ into 2777 using (a) an ISZ instruction for indexing and (b) autoindexing. Use flowcharts and comments to document the procedure.

3. The following program was previously given to multiply two numbers together.

```
*200
START,      CLA CLL
            TAD A
            CIA
            DCA TALLY
MULT,       TAD B
            ISZ TALLY
            JMP MULT
            HLT
A,          } substitute any numbers for A and B
B,          }
TALLY,      0000
$
```

   a. What is the largest product that the PDP-8 can compute using this program?

   Using the following value for A and B, verify that the program will obtain the correct answers. Remember that any carry from the most significant bit is lost from the accumulator.

| | A | B | A × B |
|---|---|---|---|
| b. | $7756(-18_{10})$ | $0027(23_{10})$ | |
| c. | 0000 | 0005 | |
| d. | $7700(-64_{10})$ | 0000 | |

4. Write a program TRIADD which will add two triple precision numbers A+B=C. There are three parts to each number, namely AH (A high) AM (A medium), and AL (A low); BH, BM, and BL; CH, CM, and CL. Use a flowchart and comments to document the procedure.

5. Write a program to perform a multiplication between two single-precision numbers to yield a double-precision product. Use comments and a flowchart to document the procedure.

6. Write a program to multiply any number n by a power of 2 (the exponent is stored in location EXP), the product being expressed in double precision. Use comments and a flowchart to document the procedure.

7. Write a program to find how many of the numbers stored in a table from address 3000 to address 3777 are negative. Use a flowchart and comments to document the procedure.

8. Write a program that will run for exactly 20 seconds on the PDP-8 or PDP-8/I before it halts. Use a flowchart and comments to document the procedure.

9. Modify the program written for exercise 8 such that if bit 11 of the console switch register is a 1, the program runs for 20 seconds, and if it is a 0, the program runs for 40 seconds.

   Hint: The OSR instruction must be used to check the switch register.

10. The program on the next page rotates a bit left or right depending on the value of bit 0 and faster or slower depending on the value of the remaining bits. Analyze the program and comment each instruction to indicate its use in the program.

```
*200
ROTATE,     CLA CLL CML
            HLT
BEGIN,      DCA SAVEAC
            RAL
            DCA SAVEL
            TAD MASK
            OSR
            DCA COUNT
            OSR
            RAL
            SZL CLA
            JMS LEFT
            JMS RIGHT
            CLL
GO,         TAD SAVEL
            RAR
            TAD SAVEAC
INSTR,      RAR
            ISZ COUNTR
            JMP .-1
            ISZ COUNT
            JMP .-3
            JMP BEGIN
SAVEAC,     0
SAVEL,      0
MASK,       7000
COUNTR,     0
COUNT,      0
LEFT,       0
            ISZ LEFT
            TAD KRAL
            DCA INSTR
            JMP I LEFT
RIGHT,      0
            TAD KRAR
            DCA INSTR
            JMP I RIGHT
KRAR,       7010
KRAL,       7004
$
```

# on-line operations

## system description and operation

## loading, editing, and debugging

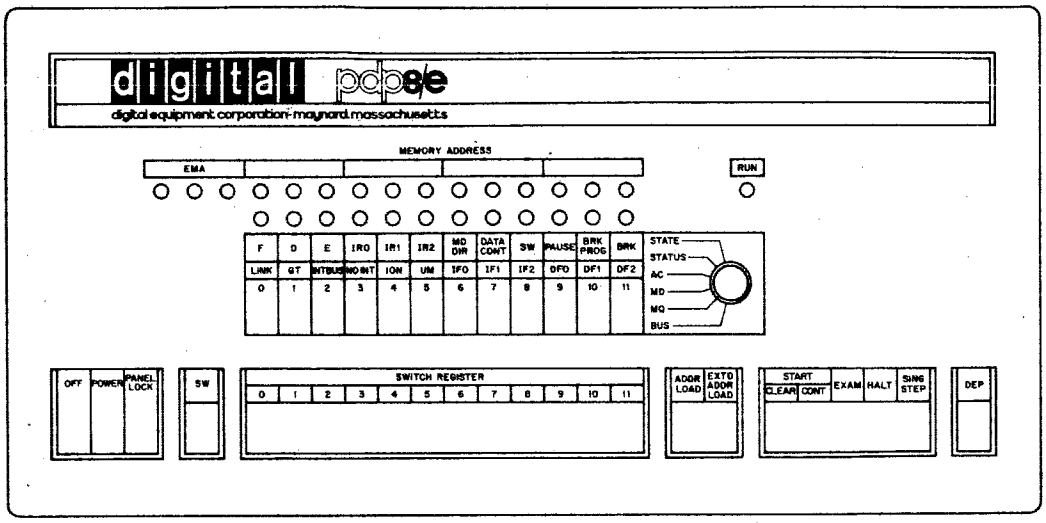
# chapter 4

# system description and operation

A PDP-8/E system is composed of the computer (programmer's console), a keyboard/printer terminal and possibly other peripheral equipment. While normal operation of a computer system is by programmed control, manual operation is necessary for many tasks. This chapter describes the manual control and operation of the PDP-8/E and provides an introduction to the more common peripheral devices which may be included in a PDP-8/E system. Chapter 6 describes the programmed control of peripheral devices and the means for transferring information between peripheral equipment and the central processor.

## PROGRAMMER'S CONSOLE OPERATION

The programmer's console allows manual control of the computer and provides the most elementary means of storing a program in memory. It consists of switches and indicator lamps which enable the programmer to examine or alter the contents of memory locations and determine the current status of a running program. The PDP-8/E programmer's console is shown in Figure 4-1. For reference purposes, the switches and indicators are identified in Table 4-1.



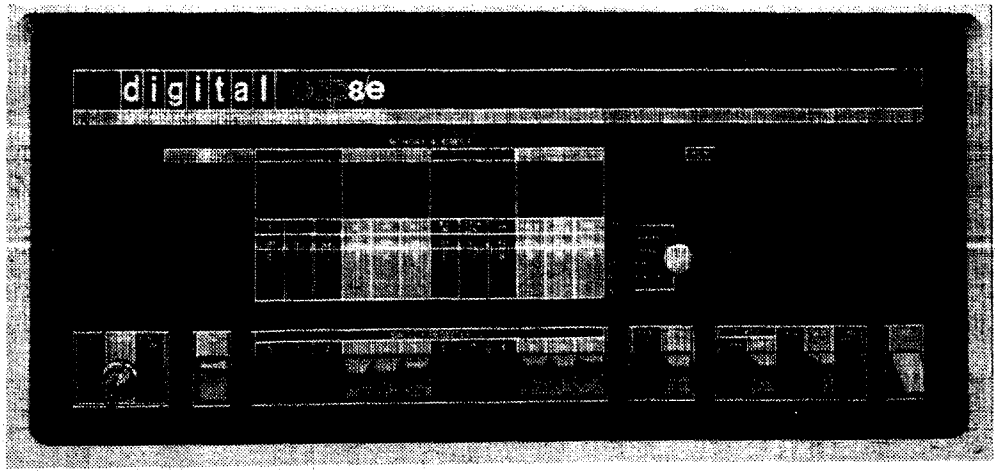




Figure 4-1 PDP-8/E Programmer's Console

**Table 4-1 Programmer's Console Control and Indicator Functions**

| Control or Indicator | Function |
|---|---|
| OFF/POWER/PANEL LOCK | In the counter-clockwise, or OFF position, this key operated switch disconnects all primary power to the computer. In the POWER, or vertical position, it applies power to the computer and all manual controls. In the PANEL LOCK, or clockwise position, it applies power to the computer, the switch register and the RUN light only. In this position, a running program is protected from inadvertent switch operation. |
| SW | When this switch is up, the Omnibus SW line is high (logical 1). When it is down, the SW line is low. This switch is used by special peripheral routines. |
| SWITCH REGISTER | The SWITCH REGISTER (SR) may be loaded with a 12-bit binary number by setting the twelve switches either up, for a 1, or down, for a 0. |
| ADDR LOAD | Pressing the ADDRess LOAD switch loads the contents of the SR into the central processor MA register and forces the processor to enter a fetch state. This causes the contents of the core memory location designated by the SR to be loaded into the MB register. |
| EXTD ADDR LOAD | Pressing the EXTendeD ADDRess LOAD switch loads the contents of SR bits 6-8 into the instruction field register and the contents of SR bits 9-11 into the data field register. |
| CLEAR | Pressing the CLEAR switch loads a binary 0 into bits 0-11 of the accumulator, the link, all I/O device flag registers, and the interrupt |

request flag register. This is equivalent to executing a CAF (Clear All Flags) instruction.

CONT — Pressing the CONTinue switch sets the run flip-flop and issues a memory start to begin program execution at the address specified by the current contents of the central processor MA register.

EXAM — Pressing the EXAMine switch loads the contents of core memory at the address specified by the MA register into the MB register and then increments the MA register and the PC. Repeated operation of this switch permits the contents of sequential core memory locations to be examined.

HALT — Pressing HALT clears the run flip-flop and causes the computer to stop at the beginning of the next fetch state. Operating the computer with HALT depressed causes one machine cycle to be executed whenever the CONTinue switch is pressed.

SING STEP — Pressing SINGle STEP clears the run flip-flop and causes the computer to execute one instruction and halt at the beginning of the next fetch state.

DEP — Lifting the DEPosit switch loads the contents of the SR into the MB register and into core memory at the address specified by the current contents of the central processor MA register, then increments the PC and the MA registers. This facilitates manual storage of information in sequential core memory locations.

EMA — The 3-bit Extended Memory Address register displays the memory field designation of the memory field currently being accessed.

MEMORY ADDRESS — The MEMORY ADDRESS register displays the contents of the central processor MA register. It combines with the EMA register to provide the 15-bit address of the next core location to be accessed.

RUN — The RUN indicator is lit whenever all machine timing circuits are activated and capable of executing instructions.

Indicator Selector Switch — This 6-position rotary knob designates which of six possible registers (or combinations of registers) is to be loaded into the adjacent 12-bit display.

Setting this knob to:

BUS — Displays the logical state of the data gating lines which connect the major registers.

MQ — Displays the contents of the multiplier quotient register.

MD — Displays the contents of the MB register. This indicates the last information read from or written into core memory.

AC — Displays the contents of the accumulator.

STATUS — Each display light is turned on to indicate the designated condition:

| Indicator Light/Bit Position | Turned On to Indicate: |
|---|---|
| 0 | The link contains a binary 1. |
| 1 | The Greater Than Flag (GTF) is raised. |
| 2 | The interrupt request line is asserted. |
| 3 | A processor condition which prevents program interrupts has been initiated by software. |
| 4 | The interrupt enable flip-flop is on. |
| 5 | The user mode line is asserted. |
| 6-8 | Displays the contents of the instruction field register. |
| 9-11 | Displays the contents of the data field register. |

STATE — With the Indicator Selector knob in the STATE position, each display light is turned on to indicate the following condition:

| Indicator Light/Bit Position | Turned on to indicate: |
|---|---|
| 0 | Currently in fetch state. |
| 1 | Currently in defer state. |
| 2 | Currently in execute state. |
| 3-5 | Displays the contents of the instruction register. |
| 6 | The MD DIR line is asserted. |
| 7 | The BREAK DATA CONT line is asserted. |
| 8 | The SW line is asserted. |
| 9 | The PAUSE I/O line is asserted. |
| 10 | The BREAK IN PROG line is asserted. |
| 11 | The BREAK CYCLE line is asserted. |

Note: The function of the various transmission lines cited and their associated control logic is documented in the *Small Computer Handbook*.

## MANUAL PROGRAM LOADING

After writing a program, the programmer must store the instructions in memory before they can be executed. One means of accomplishing this is to load the octal value of each instruction directly into core memory from the programmer's console. The following procedure will initialize the programmer's console for manual program loading:

1. Place the OFF/POWER/PANEL LOCK switch in the POWER position.
2. Load 0000 into the switch register (all switches in the down position).
3. Place all other switches in the up position, except for the spring-loaded DEPosit switch.
4. Turn off all peripheral devices.

Once the programmer's console has been initialized, a program may be loaded manually by following the instructions in Figure 4-2. Figures 4-3 and 4-4 give further instructions for checking and

running the stored program. These procedures will also permit a single core memory location to be examined or altered. The content of the desired location is considered to be a one-word "program," in this case, and the left-hand (clockwise) loops in the flow charts are never taken.

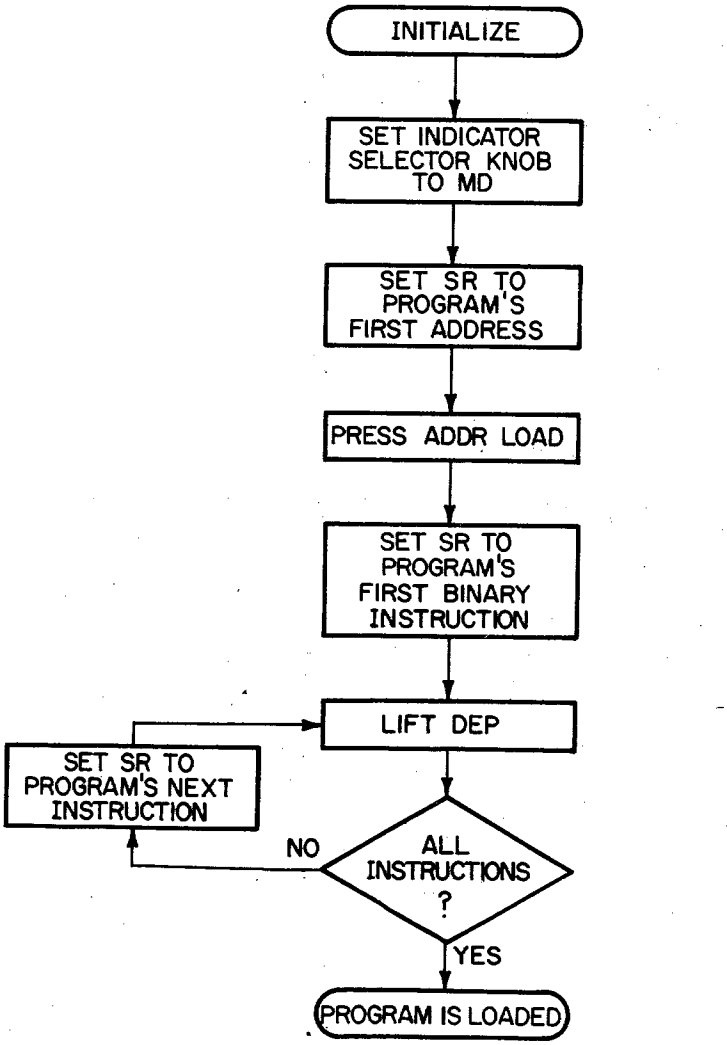



Figure 4-2 Manually Loading a Program

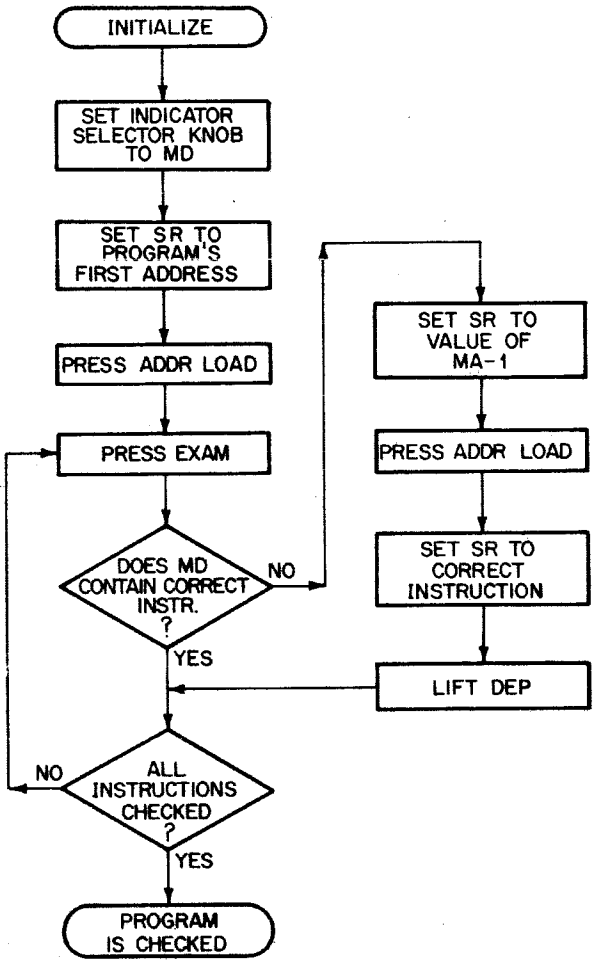



Figure 4-3 Checking a Stored Program

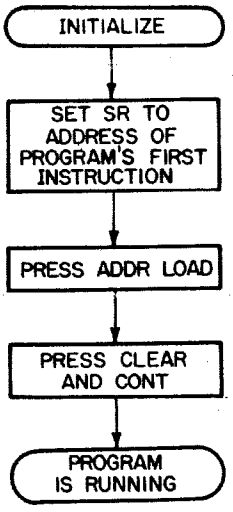



Figure 4-4 Running a Stored Program

## KEYBOARD/PRINTER CONSOLE DEVICES

The techniques described in the previous section permit direct interaction between the operator and the computer. This type of interaction is very convenient for many special applications such as program modification and debugging, data modification and controlled program execution. In general, however, a keyboard/printer peripheral device is also necessary so that routine input/output operations may proceed at the fastest possible rate of speed. The VT05 Display Terminal (Figure 4-5) and the LA-30 DECwriter Data Terminal (Figure 4-6) are console devices which permit fast, convenient interaction between the operator and the computer. They are the recommended medium for most routine input/output operations. These devices are described in the *Small Computer Handbook*. The LT-33 Teletype may also be used as a console device. Teletype operation is described in the following sections.

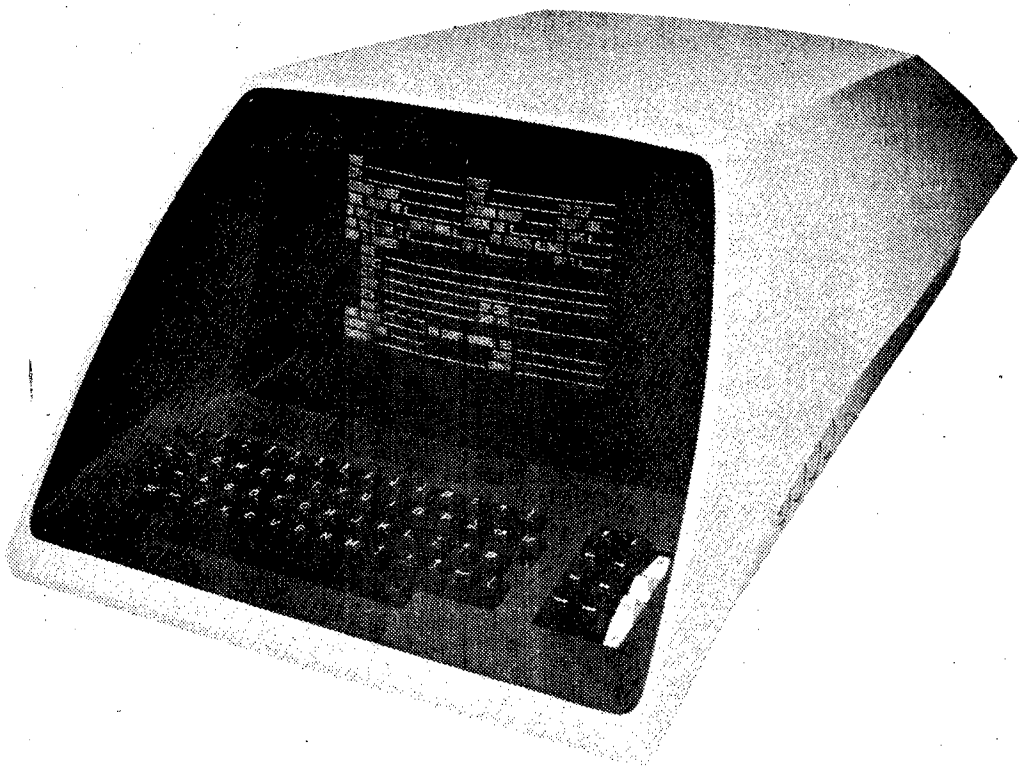


Figure 4-5 VT05 Display Terminal

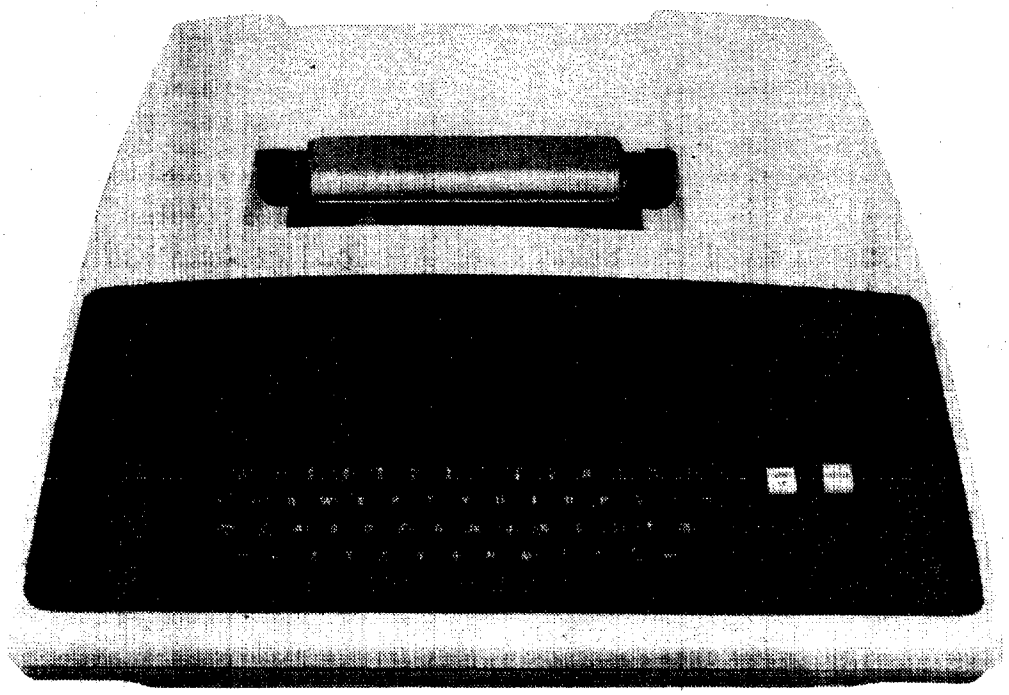


Figure 4-6 LA-30 DECwriter Data Terminal

## TELETYPE OPERATION

The LT-33 Teletype consists of a printer, keyboard, paper tape reader, and paper tape punch. The Teletype unit can operate under program control or manual control. Programmed operation of the Teletype unit is described in detail in Chapter 6. Operation of the Teletype unit as an independent device for generating paper tapes is described later in this section. Major components and their functions are listed below.

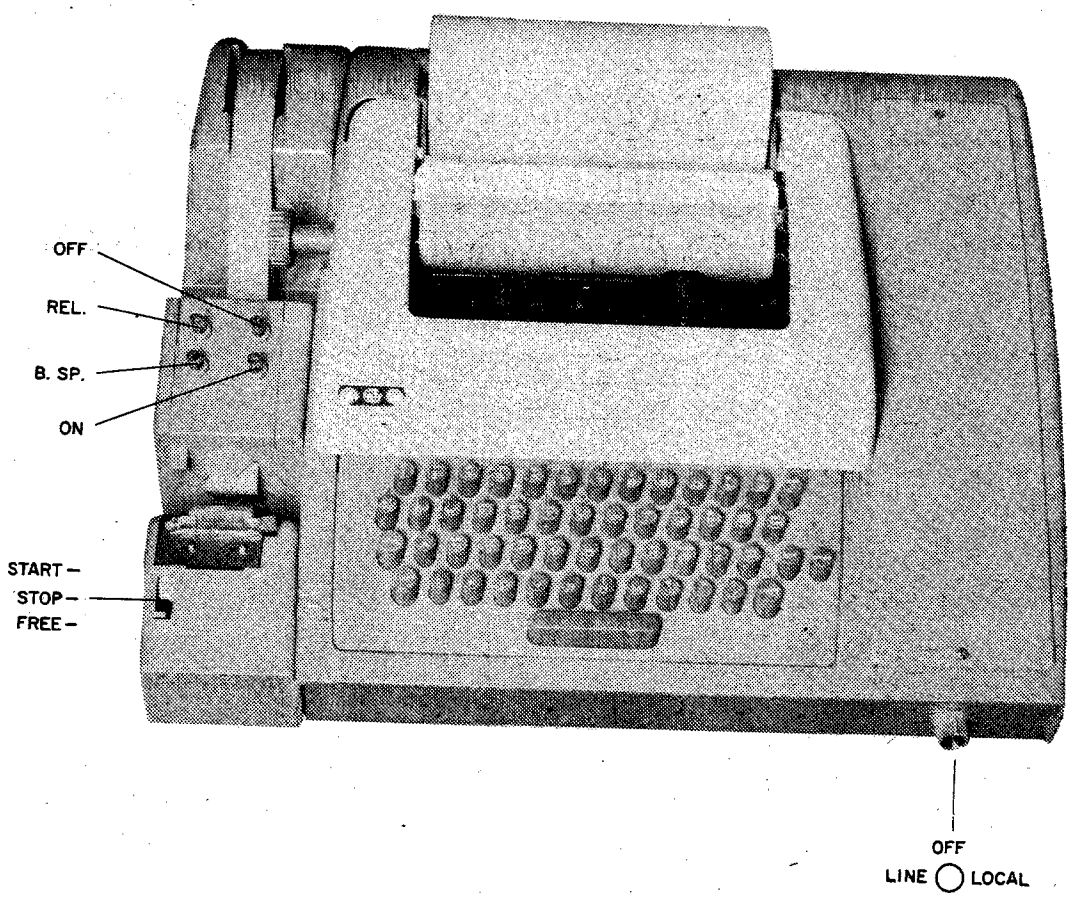



Figure 4-7 LT-33 Teletype Console

**Teletype Control Knob**

The control knob of the LT-33 Teletype console (see Figure 4-7) has the following three positions.

| | |
|---|---|
| LINE | The Teletype console is energized and connected to the computer as an input/output device under computer control. |
| OFF | The Teletype console is de-energized. |
| LOCAL | The Teletype console is energized for off-line operation under control of the Teletype keyboard and switches exclusively. |

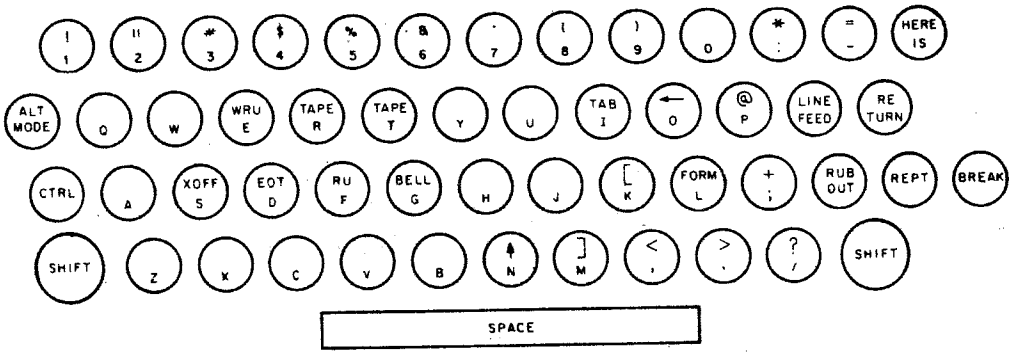



Figure 4-8 Teletype Keyboard

**Teletype Keyboard**

The Teletype keyboard shown in Figure 4-8 is similar to a typewriter keyboard, except that some nonprinting characters are included as upper case elements. When typing upper case characters or symbols such as $, % or #, which appear on the upper portion of numeric keys and certain alphabetic keys, the SHIFT key is held depressed while the desired character or symbol key is operated. When typing the nonprinting operational functions which appear on the upper portion of some alphabetic keys, the control (CTRL) key is held depressed while the desired key is operated. Table 4-2 lists several commonly used keys that have special functions in the symbolic language of PDP-8 series computers.

**Table 4-2 Special Keyboard Functions**

| Key | Function | Use |
|---|---|---|
| SPACE | Space | Used to combine and delimit symbols or numbers in a symbolic program. |
| RETURN | Carriage Return | Used to terminate a line of input. |
| HERE IS | Blank Tape | Used to generate leader/trailer tape. Effective in LOCAL control mode only. |

**Table 4-2 (cont'd) Special Keyboard Functions**

| Key | Function | Use |
|---|---|---|
| RUBOUT | Rubout | Used for deleting erroneous characters. Punches all eight channels. |
| CTRL/SHIFT/REPT/P | Code 200 | Used for leader/trailer on BIN format tapes. Keys must be released in reverse order: P, REPT, SHIFT, CTRL. |
| LINE FEED | Line Feed | Follows carriage return to advance printer one line. |
| SHIFT | | Used to type the characters and symbols which appear on the upper portion of certain keys. |

**Teletype Printer**

The Teletype printer operates at a maximum rate of ten characters per second. When the Teletype unit is on line (LINE control mode), all printed copy is generated under program control by the computer. When the Teletype unit is off line (LOCAL control mode), all printed copy is generated automatically whenever a key is typed.

**Teletype Paper Tape Reader**

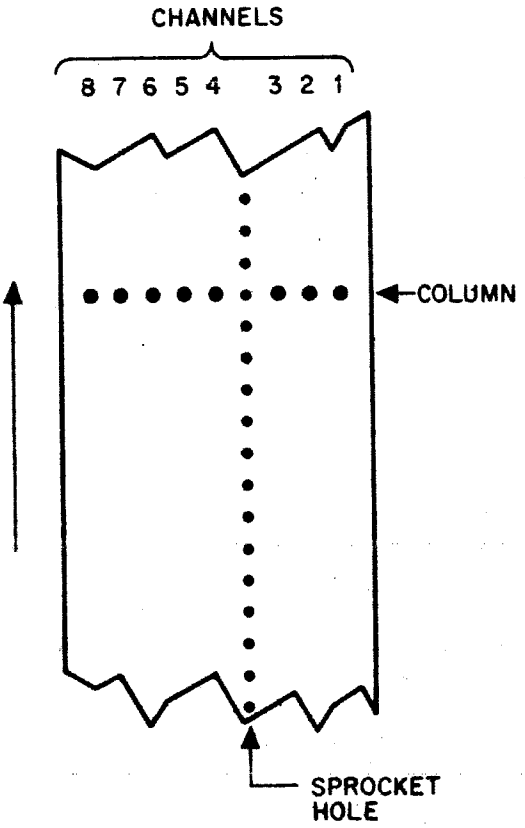
The Teletype paper tape reader (also called the low-speed reader) is used to read data punched on paper tape into core memory. The data is read from an eight-channel, perforated paper tape at a maximum rate of ten characters per second. Operation is controlled by a three-position switch, shown in Figure 4-7. The control positions are described below.

| | |
|---|---|
| START | Activates the reader; reader sprocket wheel is engaged and operative. |
| STOP | Deactivates the reader; reader sprocket wheel is engaged but not operative. |
| FREE | Deactivates the reader; reader sprocket wheel is disengaged. |

**Teletype Paper Tape Punch**

The paper tape punch is used to perforate eight-channel, rolled, oiled paper tape at a maximum rate of ten characters per second. The punch controls are shown in Figure 4-7 and described below.

| | |
|---|---|
| REL. | Disengages the tape to allow tape removal or loading. |
| B.SP. | Backspaces the tape one space for each firm depression of the B.SP. button. |
| ON | Activates the paper tape punch. |
| OFF | Deactivates the paper tape punch. |




Data is recorded (punched) on paper tape by groups of holes arranged in a definite format along the length of the tape. The tape is divided into *channels*, which run the length of the tape, and into *columns*, which extend across the width of the tape, as shown in the adjacent diagram. The paper tape readers and punches used with PDP-8 series computers accept eight-channel paper tape.

**Generating A Symbolic Tape**

Symbolic tapes to be used as input to the Assembler are most conveniently prepared on-line, using the Symbolic Editor. This program, described in Chapter 5, has several formatting and error-

correcting features which greatly facilitate the process of writing and editing symbolic tapes. If it is not possible to generate a symbolic tape on-line, the following procedure may be employed to generate such a tape on the Teletype. This procedure is best employed on an isolated Teletype unit, when the computer is unavailable.

1. Set the Teletype Control Knob to LOCAL and turn the paper tape punch ON. In LOCAL mode the Teletype unit is independent of the computer and functions much like an electric typewriter.
2. Press the HERE IS key to produce several inches of leader tape.
3. Type out the symbolic program, following the conventions described in Chapter 3. To correct an error, press B.SP. until the error is under the print/punch station, then press RUBOUT until the error and all subsequent characters have been deleted. The erroneous character and all subsequent characters may now be retyped.
4. Press the HERE IS key to produce several inches of trailer following the symbolic program, then remove the tape by tearing it against the plastic cover of the punch.

The following procedure is employed to obtain a listing of an ASCII-coded symbolic tape:

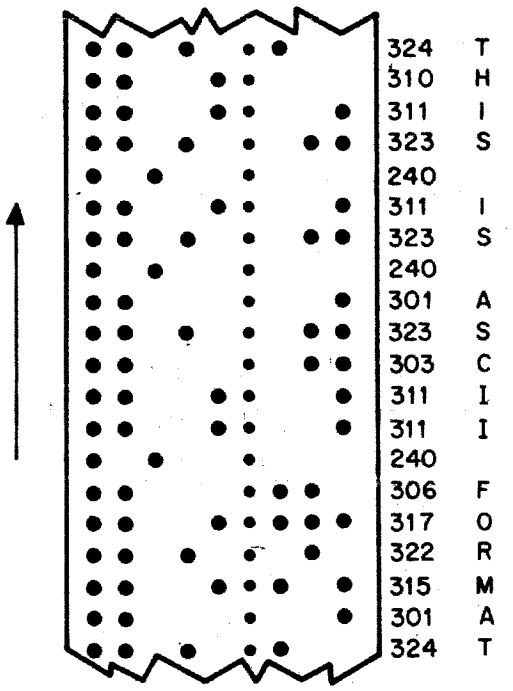
1. Set the paper tape reader switch to STOP or FREE.
2. Release the plastic cover of the reader unit and place the tape over the read station with the small sprocket holes over the sprocket wheel. Close the cover.
3. Set the Teletype Control Knob to LOCAL.
4. Push the paper tape reader switch to START and release. A printed copy of the tape will be produced on the Teletype. If the paper tape punch is ON, a duplicate of the tape will also be generated.

**Paper Tape Formats**

Manual use of the toggle switches on the programmer's console is a tedious and inefficient means of loading a program. This procedure is necessary in some instances, however, because PDP-8

series computers must be programmed before any form of input to the memory unit is possible. For example, before any paper tape can be used to input information into the computer, the memory unit must contain a stored program which will interpret the paper tape code and store the interpreted data in core memory. This loader program must often be entered into core with the console switches.

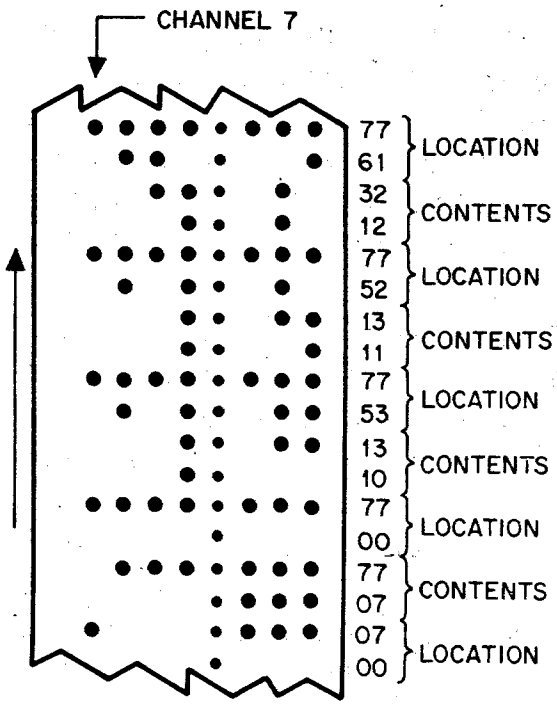
Before a loader program can be written to accept information, the format in which the data is represented on paper tape must be established. There are three basic paper tape formats commonly used by PDP-8 series computers. The following paragraphs describe and illustrate these formats.




ASCII FORMAT

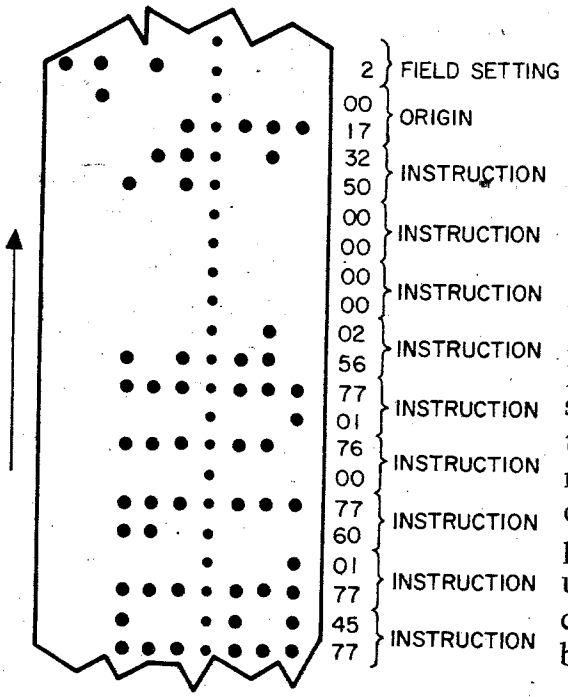
The USA Standard Code for Information Interchange (ASCII) format uses all eight channels[1] of the paper tape to represent a single character (letter, number, or symbol) as shown in the diagram at left. The complete code is given in Appendix B1.

[1] Channel 8 is normally designated for parity check. The Teletype units used with PDP-8 series computers do not generate parity, and Channel 8 is always punched.




## RIM (READ IN MODE) FORMAT

RIM format tape uses pairs of adjacent columns to represent 12-bit binary words directly. Channels 1 through 6 are used to represent either addresses or information to be stored. A channel 7 punch indicates that the current column and the following column are to be interpreted as an address specifying the location at which the information contained in the following two columns is to be stored. The tape leader and trailer for RIM format tape must be punched in channel 8 only (octal 200).




## BIN (BINARY) FORMAT

Binary format is similar to RIM format except that only the first address in a series of consecutive addresses is specified. A channel 7 punch indicates that the current column and the following column are to be interpreted as an address. Successive pairs of columns are stored in sequential locations following this address until another channel 7 punch is encountered. A channel 7 and a channel 8 punch designate the current column as a memory field specification. Leader/trailer tape must be punched in channel 8 only.

### Paper Tape Loader Programs

Each of the three paper tape formats has its own primary application. The ASCII format is used for symbolic programs which provide input to the Assembler. As described in the previous chapters, the Assembler translates ASCII-coded mnemonic instructions and symbolic addresses into binary instructions and absolute addresses. Once this translation has been performed, a binary format tape is generated.

Binary format tapes are the usual means of loading assembled programs into PDP-8 core memory. Binary tapes are loaded under program control, using the BIN Loader, which is an 83-instruction program that must be placed in core before any binary format tape may be loaded.

RIM format is easier to load than BIN format because RIM format supplies a core address for every instruction. If the BIN Loader is not in core, the 17-instruction RIM Loader may be toggled into core from the programmer's console and used to load a RIM format tape of the BIN Loader. RIM Loader instructions and corresponding core memory locations are listed in Appendix E , which also includes directions for storing and using the RIM and BIN Loaders. Chapter 5 contains further discussion of the use of paper tape loaders.

## PERIPHERAL EQUIPMENT AND OPTIONS

PDP-8/E computers are used in many different environments and are interfaced with many different peripheral devices. The Teletype unit is the most common peripheral device, but other equipment and options often incorporated in a system with the PDP-8/E include high-speed paper tape reader and punch units, DECtape, DECdisk, extended memory, and the extended arithmetic element (EAE). These options give the basic PDP-8/E new capabilities of which the programmer should be aware. The purpose and features of each of these options are described in the following sections.

### High-Speed Paper Tape Reader and Punch Unit

Loading a long paper tape program into core memory with the low-speed reader of the LT-33 Teletype unit is very time consuming. Punching a long program on paper tape from an assembly program is also very slow. If handling of lengthy paper tapes is commonly required, much computer time is wasted while low-

speed input/output devices read or punch data. The high-speed paper tape reader and punch unit performs paper tape input and output at a considerably faster rate than the low-speed reader and punch. It is of great value in any system that relies on paper tape as a primary medium of data and program storage.

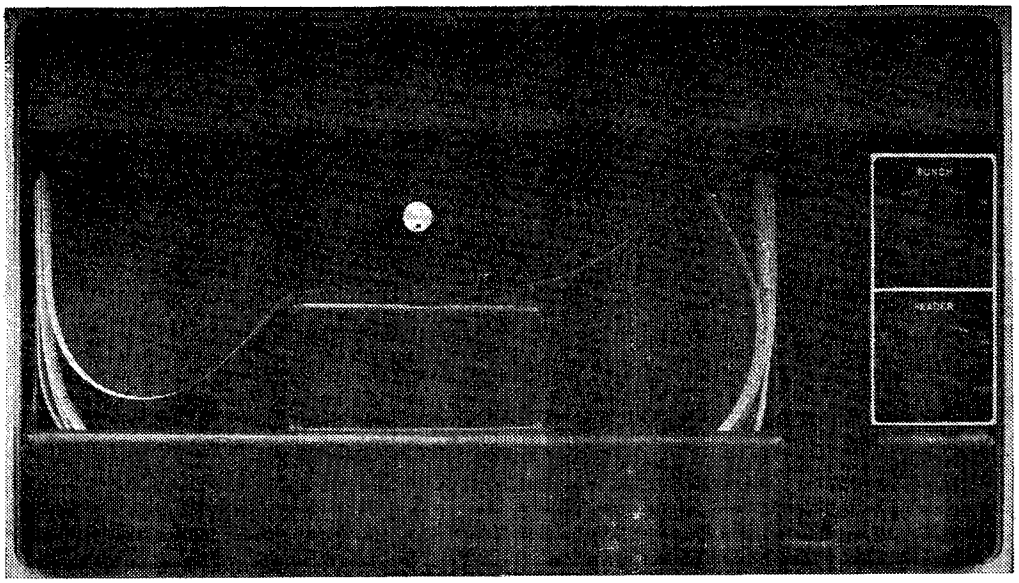


Figure 4-9 High-Speed Paper Tape Unit

The high-speed paper tape reader and punch unit is available in two versions: the rack mounted PC8-EA illustrated in Figure 4-9, and the table top PC8-EB. Both units consist of a PR8-E high-speed paper tape reader and a PC8-E high-speed paper tape punch, mounted on a single chassis. The reader and punch are also available separately.

The high-speed reader accepts input data from eight-channel, fan-folded, non-oiled paper tape at a maximum rate of 300 characters per second, or thirty times the LT-33 maximum input rate. The high-speed punch records output data at a maximum rate of 50 characters per second. All reader and punch operations are executed on-line. They may be controlled directly by the computer or from the keyboard through the computer.

The reader and punch are each supplied with an ON/OFF rocker switch which applies power to the respective units in the

ON position and disconnects power in the OFF position. Each device is also provided with a FEED switch which advances the tape without reading, in the case of the reader, or advances tape with only the feed holes punched, in the case of the punch unit. The reader is supplied with a control knob which may be turned counter-clockwise to raise the tape retaining lever and free the tape, or clockwise to lower this lever and engage the sprocket wheel.

The following procedure is employed to position tapes in the high-speed reader:

1. Turn the control knob to raise the tape retaining lever.
2. Place a fan-folded tape in the right-hand bin.
3. Place several folds of leader in the left-hand bin and position the tape so that the sprocket wheel engages the feed holes.
4. Turn the control knob to lower the tape retaining lever.
5. Press the FEED switch briefly to ensure that the tape is properly positioned.
6. Tape is advanced and read by programmed computer instructions.

**Extended Memory**

PDP-8 series computers have a basic core memory composed of 4096 twelve-bit words. Core memory may be expanded by the addition of up to seven 4096-word memory modules, providing a maximum storage capacity of up to 32,768 words. Each module is called a *memory field*. Memory fields are numbered from 0 to 7, with memory field 0 designating the original 4096 words of core. Core locations within each memory field are numbered from 0 to $7777_8$ ($4095_{10}$).

One twelve-bit data word is capable of addressing a maximum of $2^{12}$, or 4096, unique locations. However, PDP-8 series computers use two special 3-bit registers which permit $2^{15}$, or 32,768, locations to be addressed.

During program execution the content of the *instruction field register* determines the memory field from which the operand of a directly addressed instruction should be taken. Any directly addressed TAD, AND, ISZ or DCA will obtain its operand from the designated memory field, and any indirectly addressed TAD,

AND, ISZ or DCA will obtain its pointer from the designated memory field.

After an indirectly addressed TAD, AND, ISZ or DCA instruction obtains its pointer from the memory field designated by the instruction field register, it then obtains its operand in a similar manner from the memory field designated by the *data field register.*

The instruction field and data field registers are originally set by loading the desired binary designations into switch register bit positions 6-8 (instruction field register) and 9-11 (data field register), then pressing the EXT ADDR LOAD switch. These registers may also be loaded under program control.

Aside from its more obvious applications, extended memory is commonly used to store important system software. For example, the BIN Loader might be permanently stored in memory field 1. If the instruction field register is loaded with 001 and the data field register is loaded with 000, in this case, the BIN Loader will execute in memory field 1 and store a binary program (which is simply data to the Loader) in memory field 0.

**DECtape System**

DECtape provides an optional auxiliary magnetic tape storage and updating facility for PDP-8 series computers. A DECtape system consists of at least one DECtape control unit and one to eight TU56 DECtape transport units. In contrast to conventional magnetic tape devices which store information in sequential, variable-length positions, DECtape permits allocation of fixed, *addressable* positions for information storage. DECtape is bidirectional, although data is usually read in the same direction that it was written, and it records information on pairs of identical, nonadjacent channels, to minimize any chance of data loss.

A standard PDP-8 DECtape contains $2702_8$ blocks of $201_8$ 12-bit data words each. This provides a total usable storage of $563{,}302_8$ (or $190{,}146_{10}$) 12-bit words per standard tape. Nonstandard tapes may be prepared for special applications.

The TU56 DECtape Transport Unit shown in Figure 4-10 reads and writes the 10-channel magnetic tape. Tape movement may be controlled by program instructions or by manual operation of switches located on the front panel of the transport. Data is transferred only under program control. The manual transport controls are identified in Table 4-3.

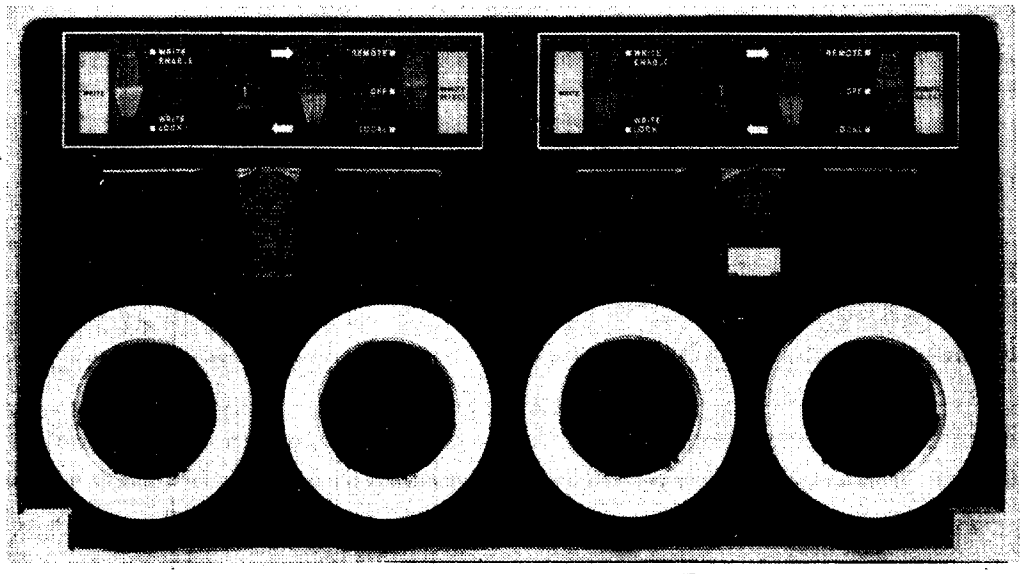


Figure 4-10 TU56 DECtape Transport Unit

**Table 4-3 TU56 DECtape Transport Controls**

| Transport Control Position | Function |
|---|---|
| REMOTE | This switch position energizes the DECtape transport and places it under program control. |
| OFF | This switch position disables the DECtape transport. |
| LOCAL | This switch position energizes the DECtape transport and places it under operator control from external transport switches. |
| WRITE ENABLED | This switch position enables the DECtape for search, read, and write activities. |
| WRITE LOCK | This switch position limits the DECtape transport to search and read activities only. (This prevents accidental destruction of permanent data.) |
| Unit Selector | The value specified by this eight-position rotary dial identifies the transport to the control unit. |
| → | With the transport in LOCAL mode, depressing this switch causes tape to feed onto the right hand spool. |
| ← | With the transport in LOCAL mode, depressing this switch causes tape to feed onto the left-hand spool. |

The following operations will initialize a DECtape transport unit for use under program control. It is assumed that a preformatted DECtape (see Chapter 7) is employed.

1. Set the REMOTE/OFF/LOCAL switch to OFF.
2. Place a DECtape on the left spindle with the DECtape label out.
3. Wind four turns of tape onto the right spindle.
4. Set the REMOTE/OFF/LOCAL switch to LOCAL.
5. Wind a few turns of tape onto the right spindle with the → switch to make sure the tape is properly mounted.
6. Dial the correct unit number on the unit selector dial.
7. Set the REMOTE/OFF/LOCAL switch to REMOTE. Select either WRITE ENABLE or WRITE LOCK.

**DECdisk Systems**

The DF32D DECdisk is a low-cost, random-access, bulk storage device and control mechanism with a storage capacity of 32,768 twelve-bit words. Data is transferred by means of the data break facility at a maximum rate of 34 microseconds per word. The DF32D system can accommodate up to three DS32D expander disks, providing an economical storage capacity of up to 131,072 words.

The RS08 DECdisk and RF08 controller combination provides storage for 262,144 words per disk. Up to four RS08 disks may be driven by one controller, allowing a maximum system storage capacity of over 1 million words. Data is transferred at a maximum rate of 16.7 microseconds per word, and average access time is less than 20 milliseconds.

The RK8 Disk Cartridge System consists of one RK08-P Disk Interface Control which will drive up to four RK01 disk files. System capacity is 831,488 words per disk, or up to 3.3 million words with the maximum system configuration. The RK8 Disk System will transfer a full 4096 words in about 80 milliseconds.

**Extended Arithmetic Element**

The KE8-E Extended Arithmetic Element (EAE) is an option for the PDP-8/E which provides circuitry to perform arithmetic operations that cannot be performed directly using the basic PDP-8/E instruction set. The EAE microinstructions permit multipli-

cation and division of unsigned integers to be performed directly. Other microinstructions perform arithmetic or logical shifts and normalization. EAE microinstructions also permit double precision numbers to be added, complemented, incremented and stored directly.

The 12-bit Multiplier Quotient (MQ) Register is used in conjunction with the accumulator to perform multiplication, division and double-precision operations. The content of this register is displayed on the PDP-8/E programmer's console when the indicator selector knob is set to MQ.

A one-bit register called the Greater Than Flag (GTF) is used as an extension of the MQ Register during shift operations. The GTF is also used to compare signed numbers while a subtraction is performed. The state of the GTF is displayed by console indicator bit position 1 when the indicator selector knob is set to STATUS.

The EAE option is essentially an increase in instruction capability. The additional instructions, which are microprogrammable, are included in Appendix D. These instructions can effect a significant reduction in core requirements and program execution time by eliminating iterative coding. In addition, the double precision instructions may be used to good advantage for input buffering operations and whenever full 12-bit addressing is desired.

## EXERCISES

1. Toggle into memory and run the programs written in Chapter 2 for exercises 6 and 10.
2. Toggle the RIM Loader (Appendix E ) into memory using the console switches. Verify the contents of memory with the EXAM switch.
3. Write a program to set the contents of locations 2000 through 2007 to the value of the switch register and then halt. Toggle it in and verify that it works.
4. Write a program to accept two numbers from the switch register and add them displaying their contents in the accumulator. (Hint: precede each OSR instruction with a HLT. After setting the switch register, activate the CONT key.) Translate the program into octal and toggle it into memory. Verify that it works properly.

# chapter 5
# loading, editing, and debugging

## INTRODUCTION

This chapter introduces the reader to on-line operations with Digital Computers. It describes how to load programs into core both manually and by the use of available loaders, how to create and edit programs, and how to debug and correct programs after assembly or compilation. Most PDP-8/E systems require the use of at least some of the programs in this chapter. The loaders—RIM, Binary, Self-Starting Binary, and Hardware Bootstrap, the Symbolic Editor, and the debugging programs—ODT and DDT, are discussed thoroughly. The information on the Editor, ODT, and DDT applies to these programs as they are provided on a Paper Tape System. (Under certain other systems, such as OS/8, modifications may have been made to these programs either to meet core restrictions or to provide the user with an even more powerful version. Any changes or modifications are described in the appropriate manual.)

## LOADERS

A loader is a short program or routine which, when in core, enables the computer to accept and store other programs. DEC offers the user the following basic loaders:

1. Read-In-Mode (RIM) Loader—used to load into core any programs punched on paper tape in RIM format, in particular, the Binary Loader.
2. Binary (BIN) Loader—used to load into core memory any programs punched on paper tape in Binary format.
3. Self-Starting Binary (SS BIN) Loader—used to automatically load or load and start a program in Binary format. (SS BIN may be merged onto the beginning of a tape to provide quick and easy loading.)
4. MI8-E Hardware Bootstrap Option—a hardware option purchased by the user containing a pre-loaded bootstrap for a particular configuration.

**RIM Loader**

When a computer in the PDP-8 series is first received by the customer, its core memory is completely demagnetized. With the exception of the Hardware Bootstrap option (discussed later) which may have been purchased with the system, the computer "knows" nothing, not even how to accept input. However, it is possible to manually load data directly into core using the programmer console switches.

The RIM Loader is the first, and often only, program loaded in this manner, and allows the computer to receive and store in core data punched on paper tape in RIM coded format. (This format, and two others—Binary and ASCII—are described in detail in Chapter 4.) The RIM Loader is used in particular to load the Binary and Self-Starting Binary Loaders.

There are two RIM Loader programs—one for input from the low-speed (Teletype) paper tape reader, and the other for input from the high-speed reader. Table 5-1 lists the octal instructions for these programs. The loading and verifying procedures are detailed in the flowcharts in Figures 5-1 and 5-2. (These flowcharts are also contained in Appendix E.) After loading RIM, it is a good programming practice to verify that all instructions have been entered properly.

When loaded, the RIM loader occupies absolute locations 7756 through 7776.

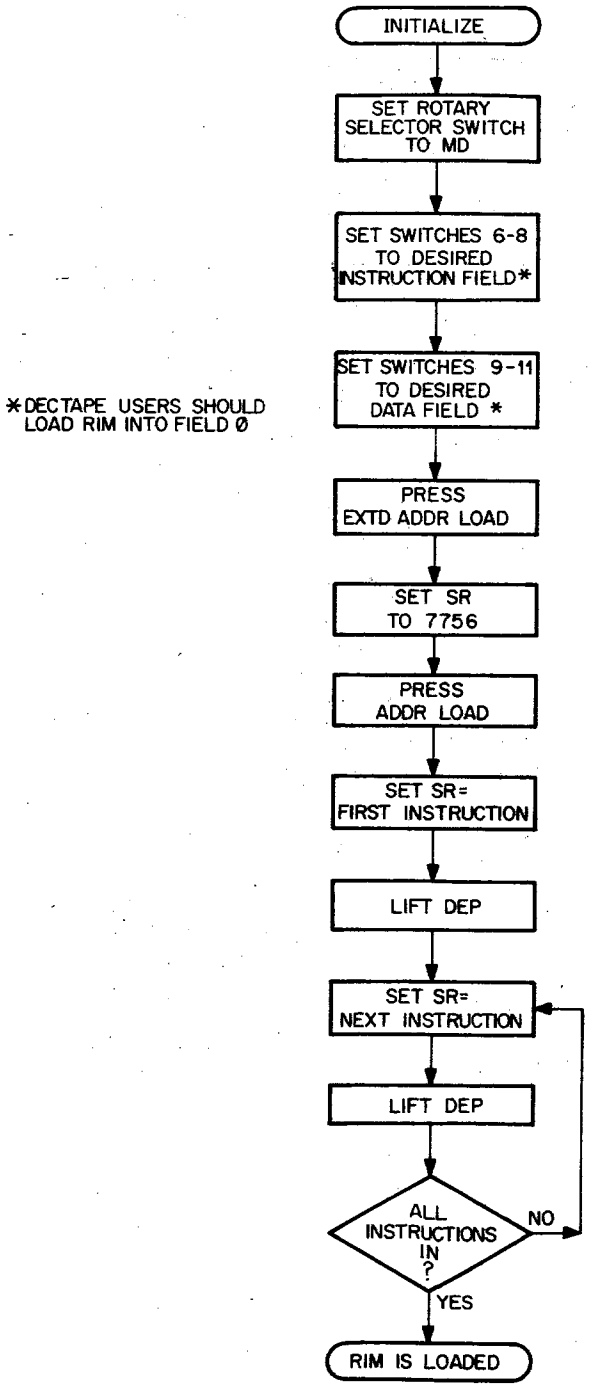



Figure 5-1 Loading the RIM Loader

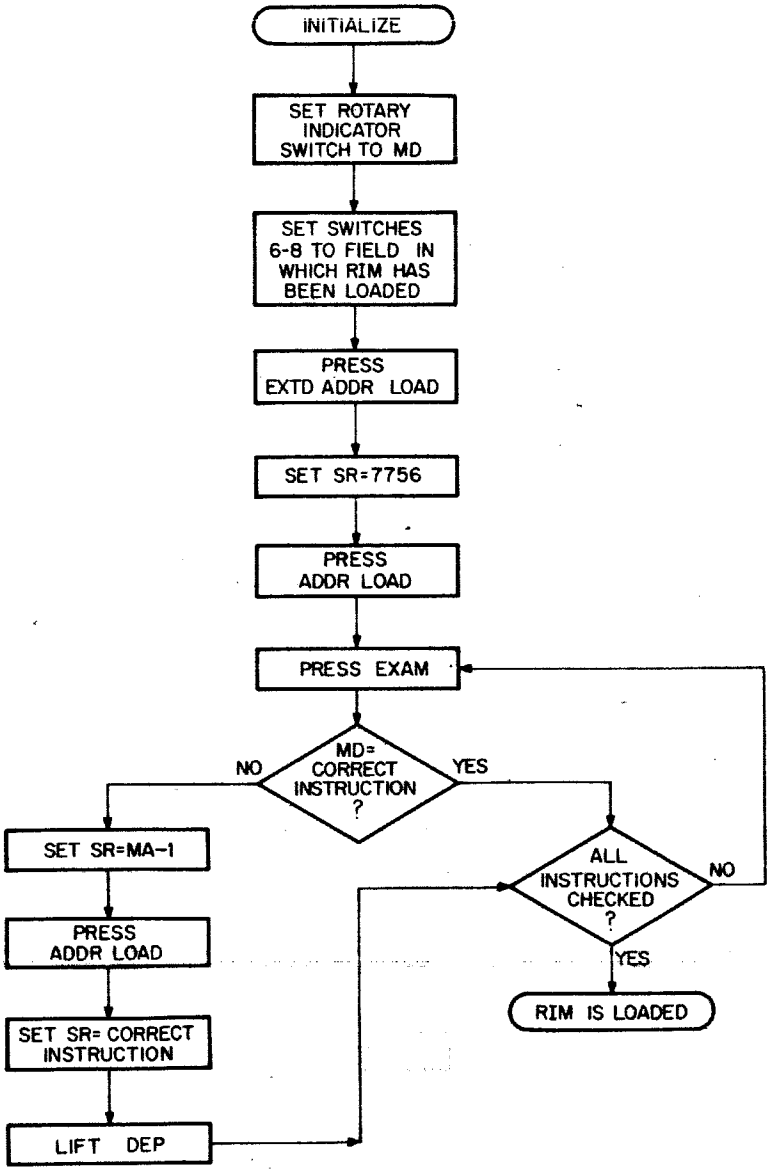



Figure 5-2 Checking the RIM Loader

**Table 5-1 RIM Loader Programs**

| Location | Instruction | |
|---|---|---|
| | Low-Speed Reader | High-Speed Reader |
| 7756 | 6032 | 6014 |
| 7757 | 6031 | 6011 |
| 7760 | 5357 | 5357 |
| 7761 | 6036 | 6016 |
| 7762 | 7106 | 7106 |
| 7763 | 7006 | 7006 |
| 7764 | 7510 | 7510 |
| 7765 | 5357 | 5374 |
| 7766 | 7006 | 7006 |
| 7767 | 6031 | 6011 |
| 7770 | 5367 | 5367 |
| 7771 | 6034 | 6016 |
| 7772 | 7420 | 7420 |
| 7773 | 3776 | 3776 |
| 7774 | 3376 | 3376 |
| 7775 | 5356 | 5357 |
| 7776 | 0000 | 0000 |

**Binary Loader**

The Binary Loader is a short utility program which, when in core, instructs the computer to read-binary-coded data punched on paper tape and store it in core memory. BIN is used primarily to load DEC-supplied binary programs and binary tapes produced by PDP-8/E assembly programs such as PAL III and MACRO-8.

BIN is furnished to the programmer on punched paper tape in RIM coded format; therefore, RIM must be in core before BIN can be loaded. When loading BIN, the input device (low-speed or high-speed reader) must be the same as that selected when loading RIM, and RIM and BIN must be loaded into the same field.

Once stored in core, BIN resides on the last page of core, occupying absolute locations 7625 through 7752 and 7777 of the field in which it was loaded. BIN was placed on the last page of core so that it would always be available for use, as all DEC's software (with the exception of the Disk Monitor and OS/8) is careful not to use this page. The programmer must be aware that if he writes a program which uses the last page of core, BIN will be destroyed when the program is run, and both RIM and BIN must be reloaded before another program can be loaded into the computer. Figure 5-3 details the method of loading BIN.

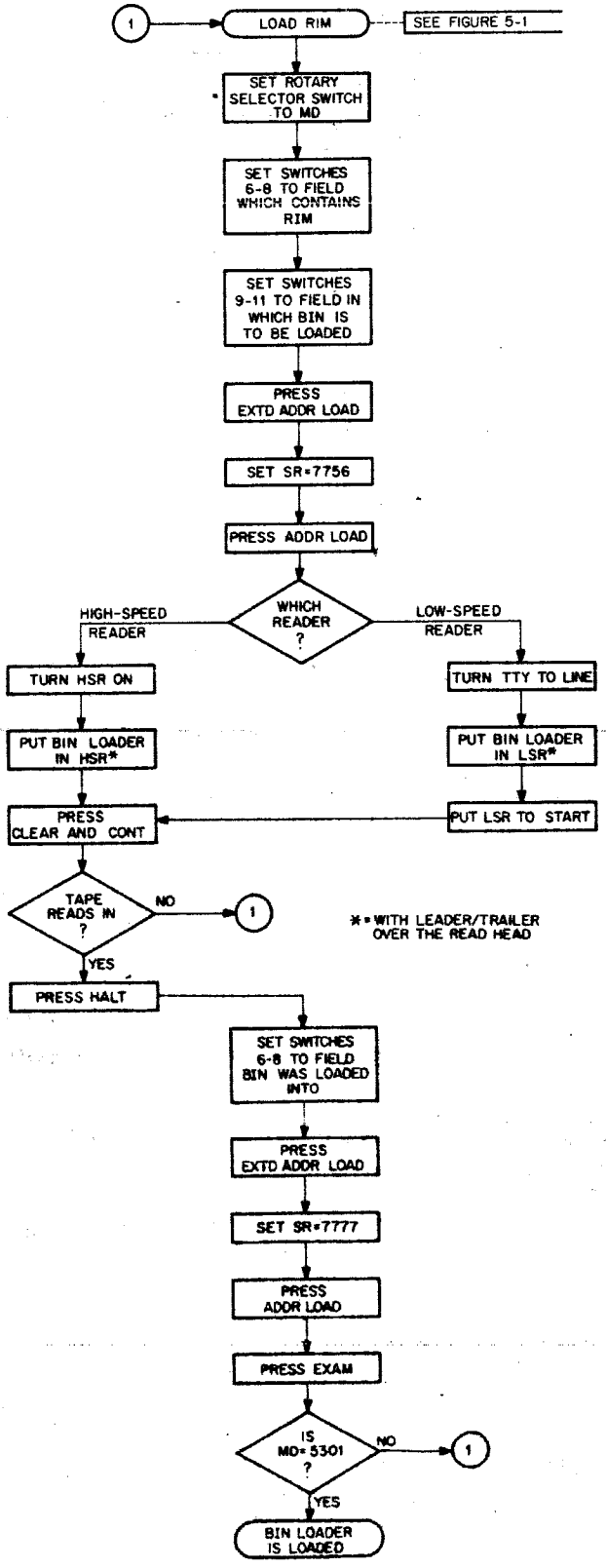



Figure 5-3 Loading the Binary Loader

The programmer is now able to load binary tapes using the method described in Figure 5-4.

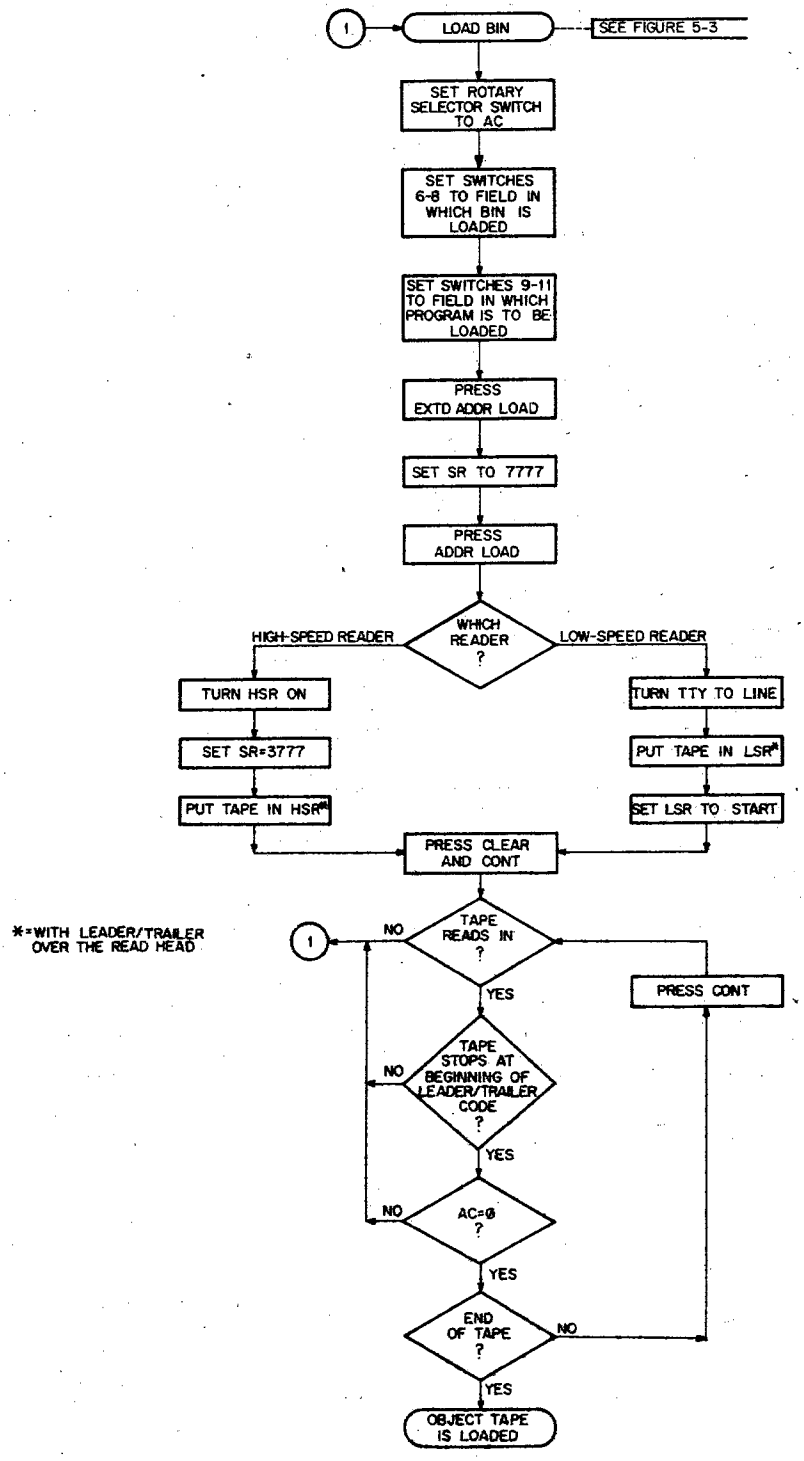



Figure 5-4 Loading A Binary Tape Using BIN

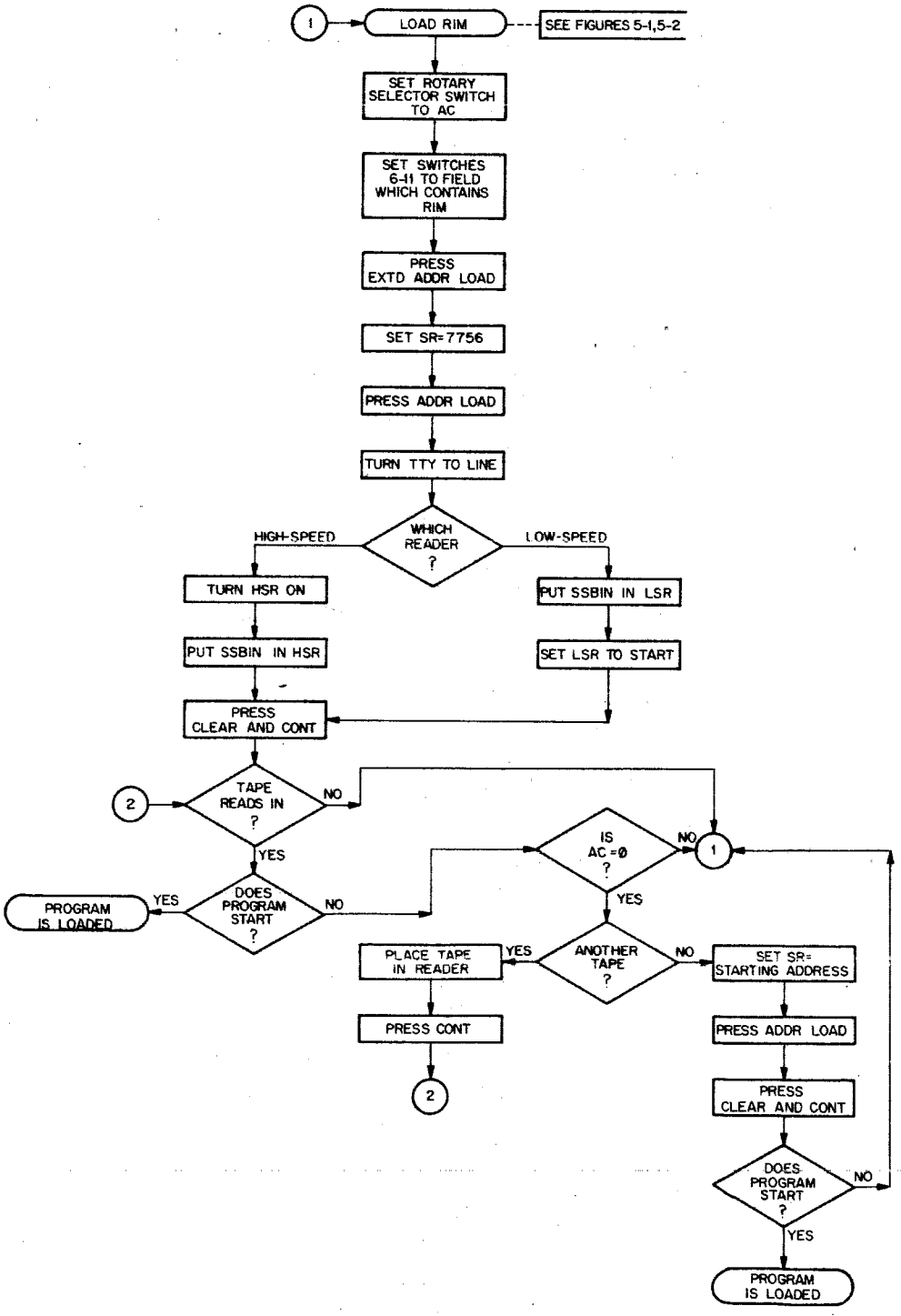



Figure 5-5 Using SS BIN with the RIM Loader

**Self-Starting Binary Loader**

The Self-Starting Binary Loader reads binary format paper tapes from either the high-speed or low-speed reader and, providing a starting address has been specified, automatically starts the program at the completion of loading.

SS BIN itself is a RIM format program and is loaded with the RIM Loader or the hardware bootstrap, generally as the first part of a two-part tape. (The second part of this tape is the object program or data to be loaded, and is physically separated from SS BIN by leader/trailer code.)

Many DEC-supplied programs are now being distributed as self-loading binaries that include a starting address for automatic starting and loading. However, SS BIN may be used independently, in which case the binary object tape must be manually loaded; if a starting address has been specified, it will be automatically started.

The user may generate his own self-loading binaries; the procedures involved in this process and other detailed information concerning SS BIN are contained in the library write-up DEC-8E-XBINA-A-D, which is available from the Software Distribution Center.

SS BIN occupies locations 7600 through 7755 and location 7777 of the memory field into which it has been loaded. Figure 5-5 describes instructions for loading the SS BIN and object tape(s).

**MI8-E Hardware Bootstrap Loader**

The MI8-E Hardware Bootstrap Loader is a hardware option available on the PDP-8/E which provides a specific bootstrap loader stored in read-only-memory as a RIM format program. Using this option, the user can automatically bootstrap into core any DEC-supported system contained in any *one* of the configurations listed in Table 5-2. (The Bootstrap Loader option and the configuration are decided by the customer when he orders his system.)

As can be seen from this table, the hardware bootstrap saves the user the necessity of manually toggling the RIM loader into memory (as described in Figure 5-1) and in most cases, provides additional bootstrapping capabilities as well.

To operate the Bootstrap Loader, the user need only follow the procedure outlined in Figure 5-6.

**Table 5-2 Hardware Bootstrap Loaders**

| Option Designation | RIM Program |
|---|---|
| MI8-E | Unencoded—The user may specify any RIM program of his choosing providing it occupies less than 32 words of core. |
| MI8-EA<br>M18-EB | Paper Tape—Provides the bootstrapping of the low/high-speed RIM loader. |
| MI8-EC | DECtape—Provides the bootstrapping of a DEC-tape-based system. |
| MI8-ED | RK8—Provides the bootstrapping of a disk-based system. |
| MI8-EE | Typeset—Allows the bootstrapping of the PDP-8/E Typesetting system. |
| MI8-EF | EduSystem (low)—Provides the bootstrapping of a system in the EduSystem series using the low-speed reader. |
| MI8-EG | EduSystem (high)—Provides the bootstrapping of a system in the EduSystem series using the high-speed reader. |
| MI8-EH | TD8-E DECtape—Allows the bootstrapping of a TD8-E DECtape-based system. |

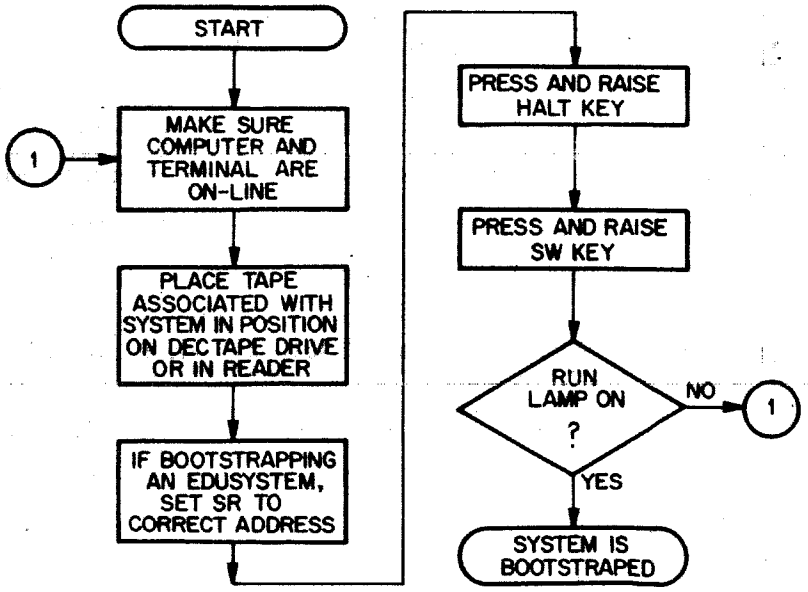



Figure 5-6 Using the Hardware Bootstrap

## SYMBOLIC EDITOR

### Introduction

The Symbolic Editor (as used on a Paper Tape System) is used to create and modify symbolic (source) program tapes from the Teletype keyboard. Thus the tedious task of preparing source program tapes off-line is eliminated.

The Symbolic Editor (hereafter, simply Editor), uses the Teletype keyboard as a very sophisticated typewriter. That is, as the source program is typed, it is entered into core where it can be checked, corrected, and modified. Then, when the user is ready to generate the source program tape, Editor will respond to the appropriate command by producing a tape suitable for assembling or compiling into an object (binary) tape which will, in turn, run on the computer.

Editor is a very helpful tool; still, it must be told precisely what to do. The user directs its operations by typing certain commands in the form of a single letter or a letter with arguments. All commands are executed by typing the RETURN key directly after the command.

Editor occupies about 1000 locations of core and leaves all but the last page of core for the source program—allowing (in a 4K core) approximately 60 lines of heavily commented text or about 340 lines of text without comments (about $4200_{10}$ characters). The source program is stored in the text buffer area of core. When the text buffer is full, Editor rings the Teletype bell. The buffer may then be enlarged, as explained later under Editing a Symbolic Tape, or the contents of the buffer may be punched (dumped) onto paper tape. If punched on paper tape, the Editor may be restarted (also explained later) and editing continues with a completely refreshed (clear) text buffer. The rest of the source program can then be placed into core and punched out so that the entire source program is on one long tape, ready to be assembled or compiled.

Each line of text includes the terminating carriage return/line feed combination which is produced by the RETURN key. All lines in the buffer are implicitly numbered in decimal, starting with 1. This implicit enumeration is continually updated by Editor to account for line insertions, moves, and deletions—a few of the

features available using Editor. For editing and listing purposes, each line is referred to by its current implicit decimal line number.

Text may be entered into core using the Teletype or high-speed paper tape reader (Editor does not distinguish paper tape from keyboard input; it perceives only symbolic input). Loading and operating instructions are contained toward the end of the chapter.

**Modes of Operation**

To distinguish between editing commands and the actual text to be entered in the buffer, Editor operates either in *command mode* or *text mode*. In command mode all input typed on the keyboard is interpreted as commands to Editor to perform some operation or allow some operation to be performed on the text stored in the buffer. In text mode, all typed input is interpreted as text to replace, be inserted into, or be appended to the contents of the text buffer.

TRANSITION BETWEEN MODES

Immediately after being loaded into core memory and started, Editor is in command mode; that is Editor is waiting for a command. The desired command code is then typed and terminated by the RETURN key, which instructs the Editor to carry out the command.

With Editor in text mode (through use of the Insert or Append commands) corrections or insertions may be typed to the text. To terminate text mode, a form feed (CTRL/FORM combination) or CTRL/G may be entered, instructing the Editor to return to command mode; Editor answers by ringing the Teletype bell to indicate the transition back to command mode.

**Command Structure**

A command directs Editor to perform a desired operation. Each command consists of a single letter, preceded by zero, one, two or three arguments. The command letter tells Editor *what* to do; the arguments usually specify *which* numbered line or lines of text are affected. Commands to Editor must take one of the following forms, where E represents any command letter.

**NOTE**

Except where specified, each line typed by the user must be entered to the computer by typing the RETURN key.

**Table 5-3 Command Structure**

| Type of Command | Command Format | Meaning |
|---|---|---|
| No Argument | E | Perform operation E. |
| One Argument | nE | Perform operation E on the referenced line. |
| Two Arguments | m,nE | Perform operation E on lines m through n, inclusive (m < n). |
| Three Arguments | m,n$jE | Used by MOVE command only (move lines m through n to before line j). |

The arguments m and n, which refer to numbered lines in memory, must be *positive,* and n must be greater than m. Two arguments must be separated by a comma, but no comma is allowed between the arguments and the command.

**Special Characters and Functions**

A number of Teletype keys have special operating functions. These keys and their associated functions are listed below:

RETURN KEY

In both command and text modes, typing the RETURN key signals Editor to process the information just typed. In command mode, it allows Editor to execute the command just typed. A command will not be executed until it is terminated by the RETURN key (with the exception of =, explained later). In text mode, RETURN causes the line of text which it follows to be entered into the text buffer. A typed line is not actually part of the buffer until terminated by the RETURN key.

ERASE (CTRL/U)

The erase character (CTRL/U combination) is used for error recoveries in both command and text modes. It is generated by holding down the CTRL key while typing a U and is not echoed on the Teletype. When used in text mode, CTRL/U cancels everything to the left of itself back to the beginning of the line; Editor performs a carriage return/line feed (CR/LF). The user then continues typing on the next line. When used in command mode, CTRL/U cancels the entire command; Editor prints a ? and per-

forms a CR/LF. The erase character cannot cancel past a CR/LF in either command or text mode. For example:

Command Mode:

```
A?
```

Text Mode:

```
THIS
HERE IS A TEXT MODE EXAMPLE
```

The Editor automatically performs a carriage return/line feed after CTRL/U is struck. In the first example above, CTRL/U was typed immediately after "A", in the second example it was typed immediately after "THIS".

RUBOUT KEY

Rubout is used in error recovery in both command and text modes with one exception. When executing a READ command (explained below) from the paper tape reader, rubouts are ignored completely and do not go into the buffer. It is necessary for the READ command to disable the rubout function since all tab characters on paper tape are, for timing purposes, followed by rubouts which would destroy the tabs. Rubouts are not stored in the text buffer but are inserted by Editor following all tab characters on the output tape.

At any other time in text mode typing the RUBOUT key echoes a backslash (\) and deletes the last typed character. Repeated rubouts delete from right to left up to, but not including, the CR/LF which separates the current line from the previous one. For example:

```
THE QUUICK\\\\ICK BROWN FOX
```

will be entered in the buffer as:

```
THE QUICK BROWN FOX
```

When used in command mode, RUBOUT is equivalent to the CTRL/U and cancels the entire command; Editor then prints a ?, performs a CR/LF, and waits for the user to type another command.

FORM FEED (CTRL/FORM)

Form feed signals Editor to return to command mode. A form feed character is generated by pressing the CTRL key while typing the FORM (/L) key. This combination is typed while in text mode to indicate that the desired text has been entered and that Editor should now return to command mode. Editor rings the Teletype bell in response to a CTRL/FORM to indicate that it is back in command mode. If Editor is already in command mode when CTRL/FORM is typed, no bell will sound. CTRL/BELL (/G) is equivalent to CTRL/FORM (except in the case of a SEARCH command as explained later).

DOT (.)

Editor keeps track of the implicit decimal number of the line on which it is currently operating. At any given time the dot, which is produced by typing the period key, stands for this number and may be used as an argument to a command. For example:

```
.L
```

means list the current line, and

```
.-1,.+1L
```

means list the line preceding the current line, the current line, and the line following it, then update the current line counter to the decimal number of the last line printed. The current line counter, represented by the dot, is generally updated as follows:

1. After a READ or APPEND command, dot is equal to the number of the last line in the buffer.
2. After an INSERT or CHANGE command, dot is equal to the number of the last line entered.
3. After a LIST or SEARCH command, dot is equal to the number of the last line listed.
4. After a DELETE command, dot is equal to the number of the line immediately after the deletion.
5. After a KILL command, dot is equal to 0.
6. After a GET command, dot is equal to the number of the line printed by the GET.

7. After a MOVE command, dot is not updated and remains whatever it was before the command.

SLASH (/)

The symbol slash (/) has a value equal to the decimal number of the last line in the buffer. It may also be used as an argument to a command. For example:

```
10,/L
```

means list from line 10 to the end of the buffer.

LINE FEED KEY

Commands and lines of text are terminated by the RETURN key which generates a carriage return *and* a line feed combination. Line feed characters are completely ignored when input is on paper tape. During output, Editor automatically punches a line feed following each carriage return.

Typing the LINE FEED while in command mode is equivalent to typing:

```
.+1L
```

and will cause Editor to print the line following the current one and to increment the value of the current line counter (dot) by one.

ALT MODE KEY

Typing the ALT MODE key while in command mode will also cause the line following the current line to be printed and the current line counter (dot) to be incremented by one. If the current line is also the last line in the buffer, typing either ALT MODE or LINE FEED will cause a ? to be typed by Editor indicating that there is no next line. (Some Teletypes have an escape key (ESC) in place of the ALT MODE; the function is identical for both ESCape and ALT MODE.)

RIGHT ANGLE BRACKET (>)

Typing the right angle bracket (>) while in command mode is equivalent to typing:

```
.+1L
```

and will cause Editor to echo > and then print the line following the current line. The value of the current line counter is increased by one so that it refers to the last line printed.

LEFT ANGLE BRACKET (<)

Typing the left angle bracket (<) while in command mode is equivalent to typing:

```
.-1L
```

and will cause Editor to echo < and then print the line preceding the current line. The value of the current line counter is decreased by one so that it refers to the last line printed.

EQUAL SIGN (=)

The equal sign is used in conjunction with the line indicators dot (.) or slash (/). When typed in command mode it causes Editor to print the decimal value of the argument preceding it. In this way the number of the current line may be found (.=*XXX*), or the total number of lines in the buffer (/=*XXX*) or the number of some particular line (/−8=*XXX*) may be determined without counting from the beginning.

COLON (:)

Colon is a lower case character with exactly the same function as the equal sign (=).

BLANK TAPE AND LEADER/TRAILER

Both blank tape and leader/trailer (code 200) are completely ignored on an input tape as are line feed characters and rubouts. Line feeds and rubouts are automatically replaced wherever necessary on output, whereas blank tape and leader/trailer are not.

TABULATION (CTRL/TAB)

Editor is written in such a way as to simulate tab stops at eight space intervals across the Teletype paper. When the CTRL key and I key are held down simultaneously, Editor produces a tabulation. A tabulation consists of from one to eight spaces, depending on the number needed to bring the carriage to the next tab stop. Thus, the user may use Editor to produce neat columns on the hard copy (printed page).

This tab function is used in connection with two switch register options (for input and output) )to allow the user to produce and control tabulations in the text buffer during input and output operations (see Switch Register Options below). On input (under a READ command) Editor can replace a group of two or more spaces with a tabulation if the user chooses to set bit 0. On output it will produce either a tab character followed by a rubout (for timing purposes) or enough spaces to reach a tab stop, depending on the settings of bit 1. Editor cannot output tab characters unless tabulations have been entered in the buffer either from the keyboard or through setting bit 0 on input.

**NOTE**

Location 0002 contains the negative (2's complement) of the number of spaces used to simulate tab stops. To change the tabulation simply change the constant in location 0002 after loading the Editor.

**Switch Register Options**

Editor uses five switch register bits in conjunction with the actual input and output commands to control the reading and punching of paper tape. It is sometimes desirable to be able to interrupt a command before it finishes. For example, if you mistakenly gave a LIST command in place of a PUNCH command and you do not wish to wait for the Teletype to list a large section of text; bit 2 on the console switch register allows you to interrupt any output command and return immediately to command mode. Table 5-4 lists the switch register bit options:

**Table 5-4 Switch Register Options**

| Bit | Position | Action and Explanation |
|---|---|---|
| 0 | 0 | Read the input tape exactly as it is. |
| | 1 | Read the input tape taking note of spaces. Each time two or more successive spaces are found, substitute in the buffer a tabulation for that whole group of spaces. This option affects only the READ command. |

**Table 5-4 Switch Register Options (Cont.)**

| Bit | Position | Action and Explanation |
|---|---|---|
| 1 | 0 | On punching (or listing) text from the buffer, tabulations are to be interpreted as an appropriate number of spaces. |
| | 1 | Tabulations are interpreted as a tab character followed by a rubout (211;377). |
| 2 | 0 | Normal operation. All output commands completed as specified. |
| 2 | 1 | Suppress list, punch, or search operation. If at any time during execution of an output command this switch is set to 1, output will cease, Editor will return immediately to command mode. (If this occurs while a line is being searched, any modifications to the line made during that search will be disregarded.) The current line counter (.) will be equal to the number of the line being printed or punched at that time. Until the switch is set to 0, any further output command will be ignored. |
| 10 | 0 | Low-speed output. All punching will be done via the Teletype punch. |
| | 1 | High-speed output. All punching will be done via the high-speed punch. |
| 11 | 0 | Low-speed input. The READ command expects the source tape to be in the Teletype reader. DO NOT use the APPEND command to read tapes. |
| | 1 | High-speed input. The source tape will be read from the high-speed reader. |

## Command Repertoire

Commands to the Editor are grouped under three general headings: Input, Output, and Editing. Explanation of the three types of commands is given in the following sections. Each command description will state if the Editor returns to command or text mode after completing the operation specified by the command.

The Editor will print an error message consisting of a question mark whenever the user has requested nonexistent information or

used an inconsistent or incorrect format in typing a command. For example, if a command requires two arguments, and only one (or none) is provided, the Editor will print ?, perform a carriage return/line feed, and ignore the command as typed. Similarly, if a nonexistent command character is typed, the error message ? will be printed, followed by a carriage return/line feed; the command will be ignored. (However, if an argument is provided for a command that does not require one, the argument will be ignored and the normal function of the command performed.) For example:

| Message | Explanation |
|---|---|
| L<br>? | The buffer is empty. The user is asking for nonexistent information. |
| 7,5L<br>? | The arguments are in the wrong order. The Editor cannot list backwards. |
| 17$10M<br>? | This command requires two arguments before the $, only one was provided. |
| H<br>? | Nonexistent command letter. |

## INPUT COMMANDS

Input commands allow text to be entered into the text buffer either from the paper tape reader or the Teletype keyboard. The Editor is in text mode until a form feed (CTRL/FORM) is encountered.

**Table 5-5 Input Commands**

| Command | Action and Explanation |
|---|---|
| R | READ a page of text from paper tape reader. Depending on the position of switch register bit 11, reading will be done from either the high-speed or the Teletype reader. The Editor will read information from the input tape until a form feed character (CTRL/FORM key combination) is detected or until the Editor senses a text buffer full condition. All incoming text except the form feed is appended to the contents of the text buffer. Information already in the buffer remains there.<br>In the case of input via the high-speed reader, the end of the tape will be interpreted as a form feed if an actual form feed character does not appear on the tape; |

**Table 5-5 Input Commands (Cont.)**

| Command | Action and Explanation |
|---|---|
| | the Editor returns to command mode. In the case of input via the Teletype reader, a form feed must be entered via the keyboard to return the Editor to command mode if an actual form feed character does not appear on the tape. If this is not done, the READ command is still in effect and all subsequent commands will be interpreted erroneously as text and appended to what was just read from tape.<br>Any rubout encountered during a READ command will be ignored. (See RUBOUT.) |
| A | APPEND the incoming text from the Teletype keyboard to the information already in the buffer (the buffer may be empty initially thus allowing the user to create a new file). The Editor will enter the text mode upon receiving this command and the user may then type in any number of lines of text. The new text will be appended to the information already in the buffer, if any, until the form feed (CTRL/FORM key combination) is typed.<br>As mentioned, by giving the APPEND command with an empty buffer, a symbolic program may effectively be generated on-line by entering the program via the keyboard.<br>Any rubout encountered during execution of an APPEND command will actually delete the last typed character. Repeated rubouts will delete from right to left up to but not beyond the beginning of the current line.<br>The APPEND command *must not* be used to read paper tapes from the Teletype reader since every rubout on the tape will delete a character. |
| nI | INSERT before line n, the text entered from the Teletype keyboard. The Editor enters text mode to accept input, and the first line typed becomes the new line n. Both the line count and the numbers of all lines following the insertion are increased by the number of lines inserted; the value of the current line counter (.) is equal to the number of the last line inserted. To reenter command mode, the CTRL/FORM (form feed) combination must be typed. This terminates text mode; if not typed, all subsequent commands will be interpreted erroneously as text and entered in the program immediately after the intended insertion. |
| I | INSERT, without an argument, will insert text before line 1. |

**NOTE**

In these commands, the Editor ignores ASCII codes 340 through 376. These codes include the codes for the lower case alphabet (ASCII 341-372). The Editor returns to command mode only after the detection of a form feed or when Editor senses a buffer full condition (see section on Editing a Symbolic Tape).

## OUTPUT COMMANDS

Output commands are subdivided into *list* and *punch* commands. List commands will cause the printout on the Teletype of all or any part of the contents of the text buffer to permit examination of the text. Punch commands provide for the output of leader trailer, form feeds, corrected text, or for the duplication of pages of an input tape. List or punch commands do not affect the contents of the buffer.

### *List Commands*

The following commands cause part or all of the contents of the text buffer to be listed on the Teletype.

**Table 5-6 List Commands**

| Command | Action and Explanation |
|---|---|
| L | LIST the entire page. This causes the Editor to list the entire contents of the text buffer. |
| nL | LIST line n. This line will be printed, followed by a carriage return and a line feed. |
| m,nL | LIST lines m through n, inclusive (m must be less than n). Lines m through n will be printed on the Teletype. |

The Editor remains in command mode after a list command and the value of the current line counter is updated to be equal to the number of the last line printed.

### *Punch Commands*

The following commands control the punching onto paper tape of leader/trailer, text, and form feeds.

Note that the PUNCH and NEXT commands halt the computer before executing the command. This is intended to let the user be sure the control switches are set correctly before any tape is punched. Pressing the CONTinue key on the console will cause the Editor to proceed with the command. The Editor remains in command mode after execution of any command which punches tape.

The Editor is designed to minimize the possibility of illegal or meaningless characters being punched into a source tape, therefore the illegal (*nonexistant*) codes 340-376 and 140-177, and most illegal control characters will not be punched. In this way a tape containing illegal characters may be corrected by simply reading it into the Editor and punching it out.

Depending on the position of switch register bit 10 (see Table 5-4) punching will be done by either the Teletype punch or the high-speed punch.

**Table 5-7 Punch Commands**

| Command | Action and Explanation |
|---|---|
| P | PUNCH the entire contents of the text buffer. |
| nP | PUNCH line n only. |
| m,nP | PUNCH lines m through n, inclusive (where m must be less than n). |

*The above commands do not output a form feed character following the text.*

| | |
|---|---|
| F | FORM FEED. This command causes the punching of four blanks, a form feed character, and approximately two inches of blank tape. If using low-speed punch, *turn punch off* before typing command then turn on immediatelv after typing carriage return. |
| T | TRAILER. This command causes about four inches of blank tape to be punched. If using low-speed punch, *turn punch off* before typing command then turn on immediately after typing carriage return. |

*The F and T commands do not halt the computer before punching tape. If using the low-speed punch, the user must therefore turn on the punch immediatelv after typing the carriage return. The codes for a carriage return and line feed may be punched onto the tape in some instances.*

**Table 5-7 Punch Commands (Cont.)**

| Command | Action and Explanation |
|---|---|
| N | NEXT. This is a utility command which combines the functions of four commands. It punches the contents of the buffer, punches some blank tape, a form feed, more blank tape, kills the buffer, and reads in the next page of text from the reader specified by switch register bit 11 (i.e., it executes P, F, K, R). |
| nN | Executes the above sequence n times. The Editor halts only before the first punching. If n is greater than the number of pages of input tape the command will proceed in the specified sequence until it reads the end of the input tape, then it will return to command mode. (If using Teletype reader, when tape runs out type CTRL/FORM to return to command mode.) |

**NOTE**

Output operations may be interrupted by setting bit 2 of the switch register (see section on Switch Register Options).

## EDITING COMMANDS

The following commands permit deletion, alteration, or expansion of text in the buffer.

**Table 5-8 Editing Commands**

| Command | Action and Explanation |
|---|---|
| nC | CHANGE line n. Line n is deleted, and the Editor enters text mode to accept input. The user may now type in as many lines of text as he desires in place of the deleted line. Rubouts are recognized during any CHANGE operation. If more than one line is inserted, all subsequent lines will be automatically renumbered and the line count will be updated appropriately. |
| m,nC | CHANGE lines m through n, inclusive (m must be numerically less than n). Lines m through n are deleted and the Editor enters text mode allowing the user to type in any number of lines in their place. All subsequent lines will be automatically renumbered to account for the change and the line count will be updated. |

**Table 5-8 Editing Commands (Cont.)**

| Command | Action and Explanation |
|---|---|
| | After any CHANGE operation, return to command mode is accomplished by typing a form feed (CTRL/FORM) to terminate input. After a CHANGE, the value of the current line counter (.) is equal to the number of the last line of the change. |
| nD | DELETE line n. Line n is removed from the text buffer. The numbers of all succeeding lines are reduced by one, as is the line count. |
| m,nD | DELETE lines m through n, inclusive. The line following n becomes the new line m and the rest of the lines are renumbered accordingly. The value of the current line counter (.) is equal to the number of the line after the deleted line or lines. The Editor remains in command mode after all DELETE operations. |
| G | GET and list the next line which begins with a tag. The Editor begins with the line following the current line and tests for a line which does not begin with a tab, slash, or space. This will most often be a line beginning with a tag. It might also be a line containing an origin. For example:<br>`TAD` this is the current line<br>`DCA`<br>`/THIS IS A COMMENT`<br>`HERE, Ø` this line would be printed<br>`TAD` by the command G<br>`ISZ`<br>`*5ØØØ`<br>this line would be printed next if another G were typed. |
| nG | GET and list the next line which begins with a tag starting the search with line n. The Editor begins with line n and tests it and each succeeding line as described above.<br>Both G and nG update the current line counter after finding the specified line. However, if either version of the GET command reaches the end of the buffer before finding a line beginning with other than a tab, slash, or space, the current line counter retains the value it had before the GET was issued, and a ? is typed to indicate that no tagged line was found. |

**Table 5-8 Editing Commands (Cont.)**

| Command | Action and Explanation |
|---|---|
| | The Editor remains in command mode after a GET command. |
| K | KILL the entire page in the buffer. The values of the special characters / and . are set to zero. The Editor remains in command mode. |
| m,n$jM | MOVE lines m through n inclusive to before line j (m must be numerically less than n and j may not be in the range between m and n). Lines m through n are moved from their current position and inserted before line j. The lines are renumbered *after* the move is completed although the value of the current line counter (.) is unchanged. Moving lines does not use any additional buffer space.<br>A line or group of lines may be moved to the end of the buffer by specifying j as /+1. For example: 1,10$/+1M. Since the MOVE command requires three arguments, it must have three arguments to move even one line. This is done by specifying the same line number twice. Example:<br>`5,5$23M`<br>This will move line 5 to before line 23. The Editor remains in command mode after a MOVE command. |
| nS | SEARCH line n for the character specified after the carriage return which enters the command. Allow modification of line when this character is found.<br>The SEARCH command is one of the most useful functions in the Editor. It is also structured somewhat differently from the other Editor commands. After terminating the command nS with a carriage return the user has told the Editor to SEARCH line n, but he hasn't specified what to search for. The Editor is, therefore, waiting for the user to type a character. The character he types is taken as the object of the search but is not echoed. The Editor instead immediately begins printing the specified line. After typing the character for which it is searching, the Editor stops. All of the editing features are then available to the user. He may proceed using any of the following options:<br>1. Delete the entire printed portion of the line by typing CTRL/U (erase), (a carriage return/ line feed is generated). |

**Table 5-8 Editing Commands (Cont.)**

| Bit | Position | Action and Explanation |
|---|---|---|

2. Delete the entire unprinted portion and terminate the line and the search by typing the RETURN key.
3. Delete from right to left one of the printed characters for each \ (rubout) typed.
4. Insert characters after the last one printed simply by typing them.
5. Insert a carriage return/line feed, thus dividing the line into two, by typing the LINE FEED key.
6. Continue searching to the next occurrence of the search character by typing CTRL/FORM. When printing stops all options are again available.
7. Change the search character and continue searching by typing CTRL/BELL followed by the new search character.

Each time the Editor prints the character for which it is searching, printing stops and all or any combinations of the above operation may be carried out.

m,nS — SEARCH lines m through n inclusive in the same way as described above. The search character is input after the carriage return and all of the options are available. The only difference is in option number 2; typing the RETURN key deletes the entire unprinted portion of the line and terminates that line, but the search continues on the next line.

By typing CTRL/BELL to change search characters, all editing of a single line may be done in one pass. Clearly, typing CTRL/BELL twice will cause the search to proceed to termination, since the search character will now be BELL which is not stored in the buffer.

S — By typing S with no arguments the entire buffer may be searched for occurrences of a single character. It must be remembered, however, that as with every CHANGE command, every SEARCH command uses additional buffer space for storage of the

**Table 5-8 Editing Commands (Cont.)**

| Command | Action and Explanation |
|---|---|
| | new line. This is obviously necessary, since the program can have no prior knowledge of whether the size of the line will be less than, greater than, or equal to that of the old line, and it must therefore assume that it will be larger. As the entire buffer is searched, a new image of the text is created in core that is guaranteed to occupy the same or less space than previously, since all deleted spaces have been removed. The Editor recognizes this and immediately moves the text image back to the top of the buffer space. Thus, the only prerequisite to condensing the text image is that there be enough core space left to contain another image of the edited text. The options available to the user are the same as described for m,nS. |

**Operating Procedures**

After the Editor has been loaded, it may be used to read into the text buffer a page of the symbolic program to be corrected. Corrections and additions may be either entered from the Teletype keyboard or inserted from paper tape via the reader. The corrected lines, groups of lines, or the entire page of text may then be listed or punched.

The following pages describe the sequence of operations necessary to load, edit, and punch out a corrected symbolic program tape; an example of Editor usage is given at the end of this section.

LOADING AND OPERATING THE SYMBOLIC EDITOR

The Symbolic Editor is loaded into core using the Binary Loader (or SS BIN). The loading procedure is illustrated in the flowchart in Figure 5-7. After loading, the user should select the desired switch register settings as detailed in Table 5-4. Loading of any symbolic tapes to be edited and all subsequent operations are performed through the use of the Editor by giving appropriate commands from the Teletype keyboard.

The Editor resides in core in locations 200-1624.

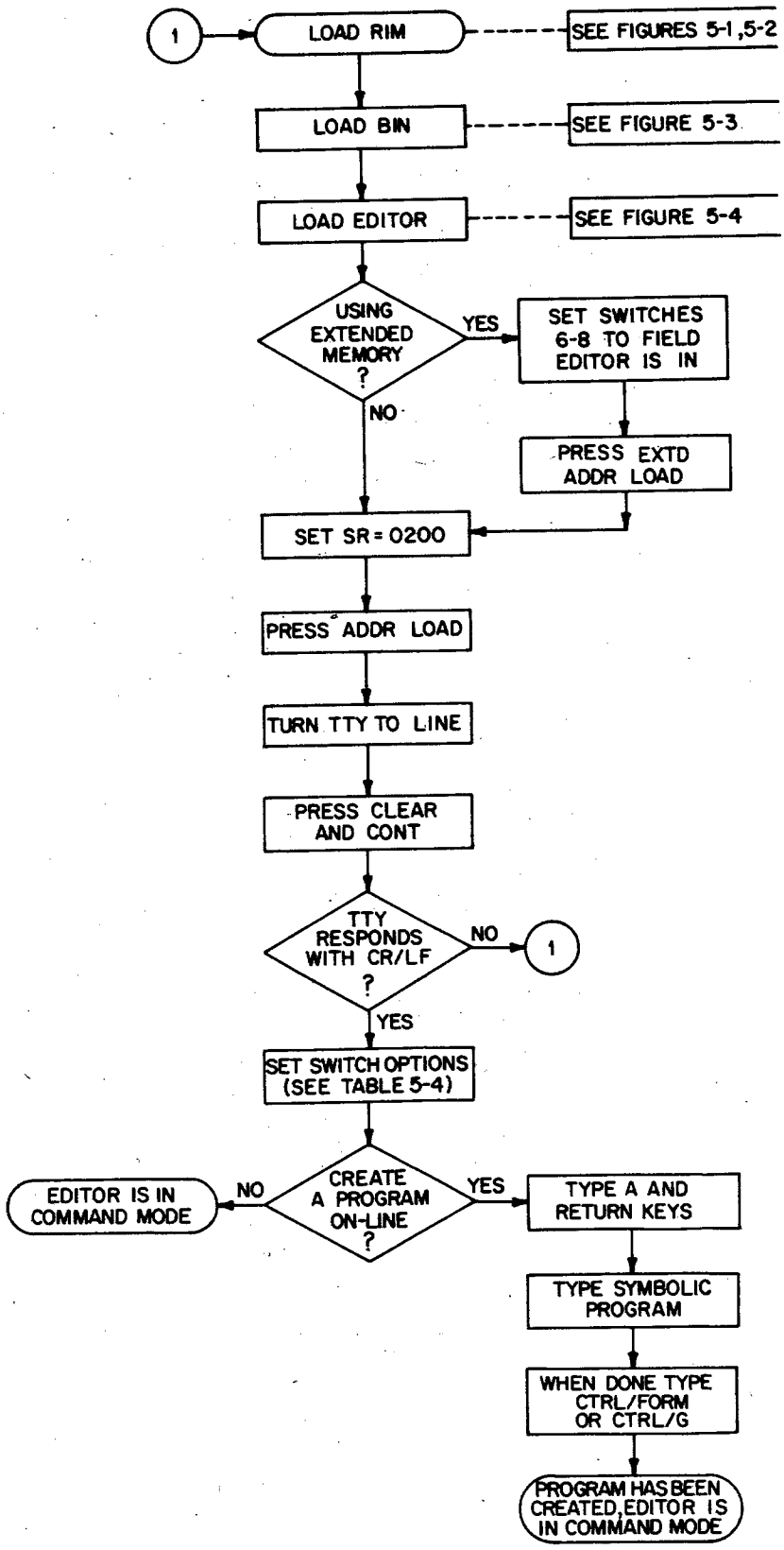



Figure 5-7 Loading the Editor and Generating a Symbolic Program On-Line

## GENERATING A SYMBOLIC PROGRAM OFF-LINE

The flowchart in Figure 5-8 shows how to generate a symbolic program off-line using the LT-33 low-speed punch. This procedure is generally much slower than using the symbolic Editor, but in some cases (such as creating extremely short programs) may prove advantageous. Leader/trailer made up of 200 code (rather than the blank tape produced by the HERE IS key) may be generated off-line by pressing (in order) the SHIFT, CTRL, REPT and @ keys and holding all down simultaneously.

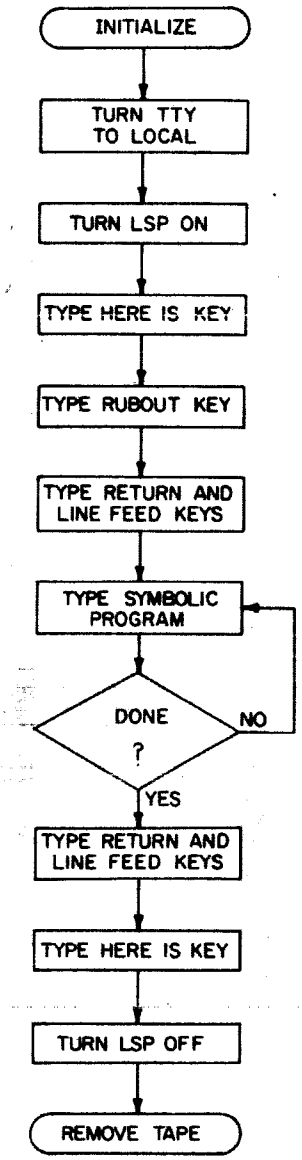



Figure 5-8 Generating a Symbolic Program Off-Line

## LOADING A SYMBOLIC TAPE USING THE EDITOR

The flowchart in Figure 5-9 shows the user the method followed in loading a symbolic tape using either the low-speed or high-speed reader. The Editor will continue reading a tape until a form feed code is encountered (see Input Commands). Upon recognizing the form feed character, the Editor enters command mode and rings the Teletype bell to indicate that it is ready to accept a command.

**CAUTION**

When using the Teletype reader, if the form feed code is encountered before the symbolic tape has completely read in (as indicated by the ringing of the bell), turn off the paper tape reader. Otherwise, characters on tape will be interpreted as *commands* to the Editor. The section of tape read in up to the form feed code should then be edited first before proceeding with the remainder of the tape.

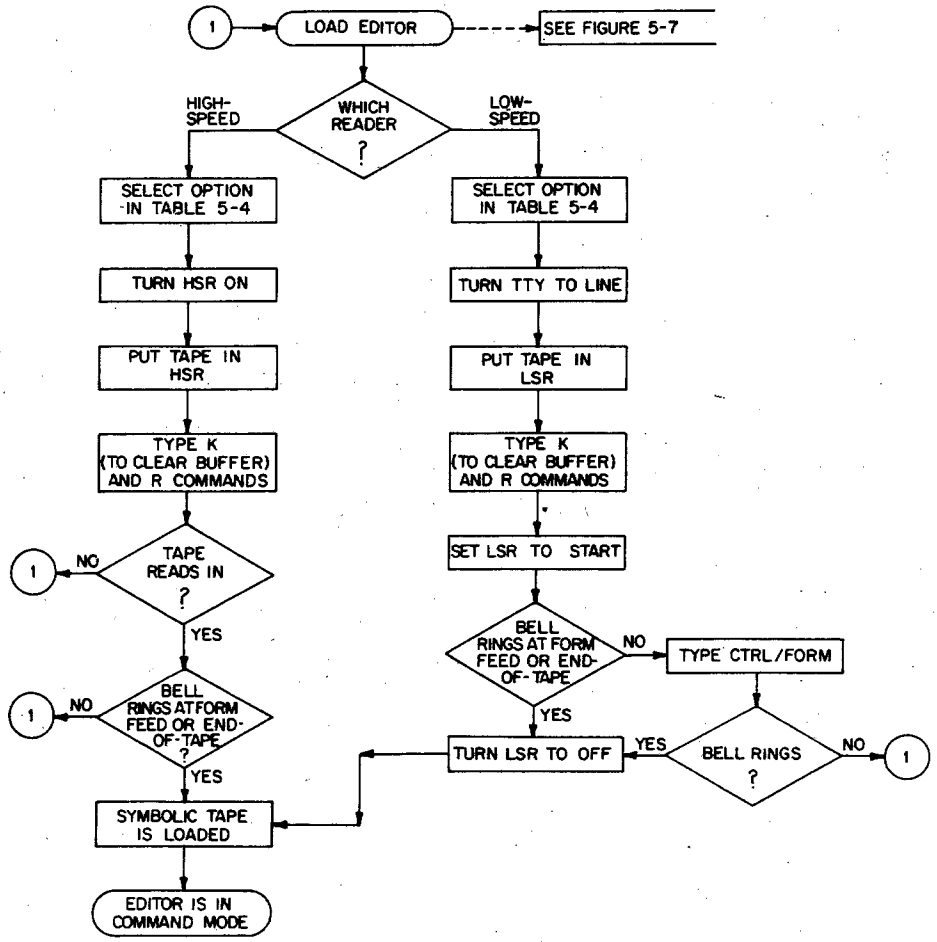



Figure 5-9 Loading a Symbolic Tape Using Editor

## RESTART PROCEDURE

If the user stops the computer for any reason, the Editor may be restarted. The user has the option of either clearing the text buffer or restarting so that the text in the buffer is not disturbed.

1. To clear the buffer, place 0176 in the switch register; press ADDR LOAD, CLEAR and CONT.
   To restart without clearing the buffer, set 0177 in the switch register; press ADDR LOAD, CLEAR and CONT.
2. Set 0200 in the switch register.
3. Press ADDR, LOAD, CLEAR and CONT.

The Editor is restarted and in command mode.

## EDITING A SYMBOLIC TAPE

The actual editing procedure depends, of course, on a particular user's requirements. The general procedure is illustrated in the example presented shortly. For input, editing, and output commands to the Editor, refer to the detailed explanation of the command structure, command repertoire, and special characters and functions under Operating Features, or see the corresponding summaries of commands and special characters at the end of the chapter. Observe also the following operating notes and precautions.

1. Terminate each command to the Editor by typing the RETURN key. This directs the Editor to execute the command.
2. After a command to insert, change, or append text to a symbolic program has been executed, the Editor *remains in the text mode* until the operator types the CTRL/FORM key combination on the teletypewriter. This combination generates the form feed code, which tells the Editor to return to command mode.
3. The Editor senses a buffer full condition (buffer capacity is approximately 60 lines of generously commented text or 340 lines uncommented) when, after completing input of a text line, it finds that characters have been packed in the last 128 locations in the text buffer. When this condition occurs, Editor rings the Teletype bell five times and exits to command mode. The user then has a choice of deleting text and continuing editing as normal or attempting to input more than

200 additional characters. After each line, the buffer full alarm will precede a return to command mode. When no more characters can be packed, Editor will again ring five times and exit from the input routine. Any further attempts to input text will be answered in the same manner until deletions have been executed to make room for text input. Although characters may be received through the input device, they probably will not be appended as text.

If Editor runs out of buffer space while searching a line, the unsearched portion of the line may be lost or the text line counter may be incorrectly set during the buffer full exit so that Editor thinks there is one more line of text in the buffer than actually exists. Occurrence of the latter will cause an error return after or during any output operation involving the last line, (for example, an N operation will be terminated as soon as the text buffer is punched). After the error return the line counter will contain the correct value.

Users should note that all such problems may be avoided by logically segmenting a program on paper tape into "pages" of 50 to 60 lines. This is done by punching groups of 50 lines followed by a form feed character (see Output Commands).

4. The Editor may be stopped at any time by pressing the HALT key; to continue press the CONTinue key.

## PUNCHING THE CORRECTED SYMBOLIC TAPE

The procedure for punching out the corrected symbolic tape depends to some extent on the user's requirements. The general sequence is given below and in the flowchart in Figure 5-10.

1. As desired, enter output commands to punch blank tape for leader/trailer purposes (T), form feeds (F), the appropriate lines of text (m,nP) or the entire text buffer (P).
2. Following the Punch command, the computer will halt, giving the user the opportunity to check the switch register (see Table 5-4) and to turn on the appropriate punch if he has not done so already. Punching is intiated by pressing the CONTinue key on the programmer's console.

**NOTE**

If the low-speed punch is used, it should be turned off during the typing of commands, as otherwise these codes will be punched on the symbolic tape.

3. Punching the symbolic program does not delete it from memory. The page remains in the text buffer in core until the KILL command (K) is given to erase it. If it is desired to read another tape into the buffer, the user must first delete the entire page of text (K). Remember that the recommended page length, as delimited by the form feed, is approximately 60 lines of heavily commented text. However, the Editor will accept more text if necessary.

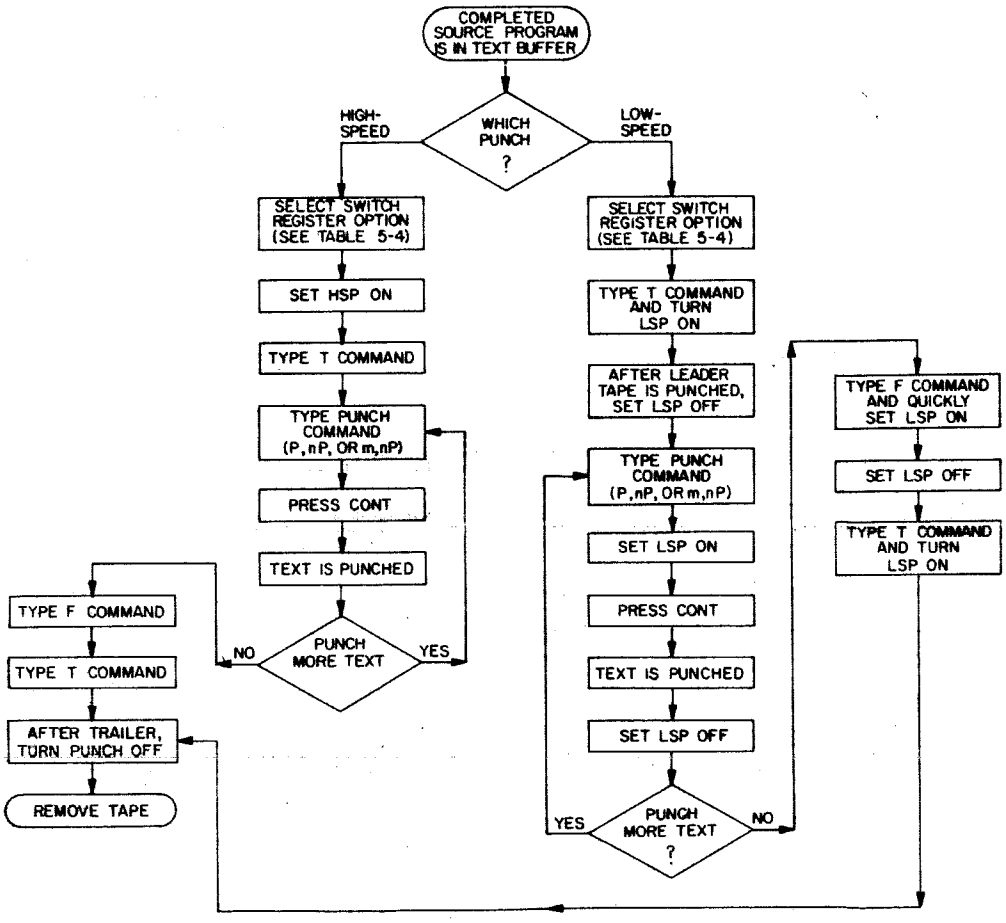



Figure 5-10 Generating a Symbolic Tape Using Editor

**Error Messages**

The proper rules for giving commands must be observed during editing, as is explained under Operating Features. If commands are given in an incorrect format or if arguments are either missing, erroneous, or extraneous, the Editor will respond by printing a question mark. Notice that some commands can legitimately take from zero to two arguments, and one takes three. In general, if an argument is either missing or extraneous, the Editor prints ? and ignores the command. Similarly, if a *negative* argument is encountered or an illegitimate command string is typed, the Editor again responds with the error message ?.

**Example of Use**

The following detailed example of the editing of a page of text is intended to familiarize the user with the basic operations of the Symbolic Editor. Where details of the loading sequence and operating procedures are not shown, it is assumed that the user has followed the correct procedures previously explained.

This example concerns a program for adding up numbers stored in locations $200_8$ through $207_8$ of the computer, with the answer to be stored in location $410_8$. The program is to start in location 600. The program listing is shown below.

```
/ADD UP NUMBERS
*600
BEGN,   HLT
/TO START THE PROGRAM, HIT "CONTINUE" ON THE CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
        CLA             /CLEAR THE ACCUMULATOR
        TAD M10         /LOAD AC WITH THE NUMBER -10
        DCA COUNTR      /PUT INTO COUNTER
        TAD TWOHUN      /LOAD AC WITH FIRST ADDRESS
        DCA POINTR      /PUT INTO POINTER
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
BEGN,   TAD I POINTR    /ADD NEXT NUMBER
        ISZ POINTR      /INDEX POINTER
        ISZ COUNTR      /INDEX COUNTER, IS IT ZERO?
        JMP BEGN        /NO;CONTINUE ADDING
        DCA I ANSWER    /YES;STORE ANSWER
        HLT             /HALT
        JMP BEGN+1
```

```
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS
M10,    -10                     /NEGATIVE TALLY NUMBER
TWOHUN, 200                     FIRST ADDRESS IN BUFFER
ANSWER  410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR VARIABLES
COUNTR, 0
POINTR, 0
$
```

Assume that this program has been assembled using the PAL III Symbolic Assembler. On pass 1, however, the assembler printed the following:

```
DT   BEGN     AT   0606


UA   ADDRES   AT   0616

UA   ANSWER   AT   0612
BEGN     0600

UA   BUFFER   AT   0616
COUNTR   0620

UA   FIRST    AT   0616

UA   IN       AT   0616
M10      0615
POINTR   0621
TWOHUN   0616
```

The message DT BEGN at 0606 signifies that the programmer has mistakenly used identical tags to specify two different addresses. An inspection of the program listing above shows that the tag BEGN has, indeed, been duplicated. It appears in line 3 of the listing as BEGN, HLT, then in line 14, starting with BEGN, TAD I POINTR. (Since the line numbers are implicit only, they are not shown in the example; they may be obtained by counting from the top down.)To correct the situation, the Symbolic Editor was read in by the Binary Loader, as explained earlier under Loading Sequence. The symbolic tape to be corrected was then loaded by the Editor using the READ (R) command, and a series of editing commands were given.

```
R

14L
BEGN,    TAD I POINTR          /ADD NEXT NUMBER

14C
ADDR,    TAD I POINTR          /ADD NEXT NUMBER

17L
         JMP BEGN              /NO;CONTINUE ADDING

17C
         JMP ADDR              /NO;CONTINUE

13,18L
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR,    TAD I POINTR          /ADD NEXT NUMBER
         ISZ POINTR            /INDEX POINTER
         ISZ COUNTR            /INDEX COUNTER, IS IT ZERO?
         JMP ADDR              /NO;CONTINUE
         DCA I ANSWER          /YES;STORE ANSWER

25L
ANSWER   410

.S
ANSWER, 410

24L
TWOHUN, 200                    FIRST ADDRESS IN BUFFER

24C
TWOHUN, 200                    /FIRST ADDRESS IN BUFFER

.-1,.+2L
M10,     -10                   /NEGATIVE TALLY NUMBER
TWOHUN, 200                    /FIRST ADDRESS IN BUFFER
ANSWER, 410
/
```

Having made the desired corrections, the programmer finally asks the Editor to list the entire text by giving the LIST (L) command, but still withholds the PUNCH command (P), pending final corrections. A new program listing is printed out and the entire text is preserved in the buffer. The programmer now punches the entire text onto paper tape by giving the PUNCH (P) command and depressing CONTinue on the console. The text will be printed as follows on the Teleprinter (by the LIST command).

```
/ADD UP NUMBERS
*600
BEGN,    HLT
/TO START THE PROGRAM, HIT "CONTINUE" ON THE CONSOLE
/
/THE NEXT FIVE INSTRUCTIONS INITIALIZE THE ROUTINE
         CLA                 /CLEAR THE ACCUMULATOR
         TAD M10             /LOAD AC WITH THE NUMBER -10
         DCA COUNTR          /PUT INTO COUNTER
         TAD TWOHUN          /LOAD AC WITH FIRST ADDRESS
         DCA POINTR          /PUT INTO POINTER
/
/THE NEXT SEVEN INSTRUCTIONS ARE THE PROGRAM ITSELF
ADDR,    TAD I POINTR        /ADD NEXT NUMBER
         ISZ POINTR          /INDEX POINTER
         ISZ COUNTR          /INDEX COUNTER, IS IT ZERO?
         JMP ADDR            /NO;CONTINUE
         DCA I ANSWMR        /YES;STORE ANSWER
         HLT                 /HALT
         JMP BEGN+1
/
/THE NEXT THREE REGISTERS CONTAIN THE CONSTANTS
M10,     -10                 /NEGATIVE TALLY NUMBER
TWOHUN,  200                 /FIRST ADDRESS IN BUFFER
ANSWER,  410
/
/THE NEXT TWO REGISTERS ARE RESERVED FOR VARIABLES
COUNTR,  0
POINTR,  0
$
```

Once the program is corrected, it can be used as input to the PAL III assembler. The pass 1 result of assembling this program is:

```
ADDR     0606
ANSWER   0617
BEGN     0600
COUNTR   0620
M10      0615
POINTR   0621
TWOHUN   0616
```

## Summary of Symbolic Editor Operations

SPECIAL KEYS

| Key | Function |
|---|---|
| RETURN key | *Text mode*—Enter the line in the text buffer.<br>*Command mode*—Execute the command. |

| | |
|---|---|
| CTRL/U keys | *Text mode*—Cancel the entire line of text, continue typing on next line.<br>*Command mode*—Cancel command. Editor issues a ? and carriage return/line feed. |
| RUBOUT key | *Text mode*—Delete from right to left one character for each rubout typed. Does not delete past the beginning of the line. Is not in effect during a READ command.<br>*Command mode*—Same as CTRL/U. |
| Form Feed (CTRL/FORM) | *Text mode*—End of input, return to command mode. |
| CTRL/G | *Text mode*—End of input, return to command mode. |
| Dot (.) | *Command mode*—Current line counter used as argument alone or in combinations with + or − and a number (as in .,.+5L). |
| Slash (/) | *Command mode*—Value equal to number of last line in buffer. Used as argument (as in /−5,/L). |
| LINE FEED key | *Text mode*—Used in SEARCH command to insert a CR/LF combination into the line being searched.<br>*Command mode*—List the next line (equivalent to .+1L). |
| ALT MODE key or ESCape key | *Command mode*—List the next line (equivalent to .+1L). |
| Right Angle Bracket (>) | *Command mode*—Same as ALT MODE key. |
| Left Angle Bracket (<) | *Command mode*—List the previous line (equivalent to .−1L). |
| Equal Sign (=) | *Command mode*—Used in conjunction with . and / to obtain their value (.=27). |
| Colon (:) | *Command mode*—Same as equal sign. |
| Tabulation (CTRL/TAB) | *Text mode*—Provides a tabulation which, on output, is interpreted as spaces or a tab character/rubout combination depending on a switch option. |

## SWITCH OPTIONS

| Switch | Position | Meaning |
|---|---|---|
| 0 | 0 | Read input tape as is |
| | 1 | Convert spaces to tabulations on input |
| 1 | 0 | Output tabulations as spaces |
| | 1 | Output tab character rubout combination for each tabulation |
| 2 | 0 | Normal operation |
| | 0 | Suppress output |
| 10 | 0 | Low-speed output |
| | 1 | High-speed output |
| 11 | 0 | Low-speed input |
| | 1 | High-speed input |

## COMMAND SUMMARY

| Command | Format(s) | Meaning |
|---|---|---|
| READ | R | Read incoming text from reader and append to buffer until a form feed is encountered. |
| APPEND | A | Append incoming text from keyboard to any already in buffer until a form feed is encountered. |
| LIST | L | List the entire buffer. |
| | nL | List line n. |
| | m,nL | List lines m through n inclusive. |
| PUNCH | P | Halt. Upon striking CONTinue key on console, punch the entire buffer. |
| | nP | Halt. Upon CONTinue, punch line n. |
| | m,nP | Halt. Upon CONTinue, punch lines m through n inclusive. |
| FORM FEED | F | Punch trailer, punch a form feed (214), punch trailer. |
| TRAILER | T | Punch four inches of trailer. |
| NEXT | N | Punch the entire buffer and a form feed, Kill the buffer and Read the next page. |
| | nN | Repeat the above sequences n times. |
| KILL | K | Kill the buffer. |

| Command | Format(s) | Meaning |
|---|---|---|
| DELETE | nD | Delete line n of the text. |
| | m,nD | Delete lines m through n inclusive. |
| INSERT | I | Insert before line 1 all the text from the keyboard until a form feed is entered. |
| | nI | Insert before line n until a form feed is entered. |
| CHANGE | nC | Delete line n, replace it with any number of lines from the keyboard until a form feed is entered. |
| | m,nC | Delete lines m through n, replace from keyboard as above until form feed is entered. |
| MOVE | m,n$kM | Move lines m through n inclusive to before line k. |
| GET | G | Get and list the next line beginning with a tag. |
| | nG | Get and list the next line which begins with a tag (starting the search with line n). |
| SEARCH | S | Search the entire buffer for the character specified (but not echoed) after the carriage return. Allow modification when found. |
| | nS | Search line n, as above, allow modification. |
| | m,nS | Search lines m through n inclusive, allow modification. |

## DEBUGGING PROGRAMS

Dynamic Debugging Technique (DDT) and Octal Debugging Technique (ODT) are the two debugging programs for the PDP-8/E. Dynamic debugging programs are service programs which allow the programmer to run his binary program on the computer and use the Teletype keyboard to control program execution, examine registers, change their contents, and make alterations to a program.

A symbolic program can be assembled correctly and still contain logical errors, i.e., errors which cause the program to do something other than what is intended. The assembler checks for certain syntax errors, but not logical errors. (Syntax errors include undefined tags, misspelled tags and incorrect formats, e.g., omission of a required operand.) Logical errors are detected only when the program is running on the computer.

### Debugging Without DDT or ODT

If the programmer feels sure that his program is correct and ready for use, he can simply load the program and let it run until it stops (if it stops). And if the program doesn't produce the correct results, the programmer without a DDT program can use the console switches to examine specific locations one-by-one to try to find the error(s) through interpreting the console lights. There are two hazards to this approach. First, by the time the program stops the error may have caused all pertinent information, including itself, to be altered or eliminated. Second, the program may not stop at all; it might continue to run in an infinite loop, and such loops are not always easy to detect.

Added to these problems are the difficulties of interpreting binary console displays and translating them into symbolic expressions related to the user's program listing. Further, adding corrections to a program in the form of patches (altered and added instructions or routines) requires seemingly endless manipulation of the console switches. In all this, the chance of programmer error at the console is large and is likely to obscure any real gain made from debugging.

The programmer can use the program assembly listing to mentally execute his program. This method is frequently used with very short programs, very short only—human memory cannot retain every step and instruction in even a fairly short program; it cannot match a computer memory.

What is needed to conveniently and accurately debug a user program is a service program which will assume the tasks the programmer would have to perform if he used the console switches. DDT and ODT are such debugging programs.

## DEBUGGING WITH DDT

The Dynamic Debugging Technique for the PDP-8/E facilitates program debugging by allowing the user to examine core memory locations (registers) and change and correct their contents, place and remove strategic halts and automatically restore and execute the instructions replaced by the halts. Communication is carried out via the Teletype keyboard using defined commands and the symbolic language of the source program or octal representation, with DDT performing all translation to and from the binary representation.

Tracking down a subtle error in a complex section of coding is a laborious and frustrating job if done by hand, but with the breakpoint facility (explained later) of DDT, the user can interrupt the operation of his program at any point and examine the state of the program and computer. In this way, sources of trouble can be isolated and corrected.

By the time the programmer is ready to start debugging a new program at the computer, he should have at the console:

1. The binary tape of the program to be debugged.
2. The symbol definition tape which was part of the assembly output from pass 1.
3. A list of the symbols and their definitions.
4. A complete octal/symbolic program listing.
5. A binary tape of the DDT program, which is loaded into core memory using the BIN Loader.

## LOADING DDT

The BIN Loader is used to load DDT and the object program (program to be debugged) into core. Refer to Figure 5-3 for the procedure used in loading BIN, and Figure 5-4 for details concerning using BIN to load binary format programs.

DDT requires locations 0004 and 5237–7577 ($2341_8$ locations). The permanent symbol table requires locations 5000–5237, and the external symbol table is allotted locations 3030–5000 (250 symbol capacity). The starting address is 5400.

The flowchart in Figure 5-11 presents the basic loading procedure for both DDT and the binary tape of the program to be debugged.

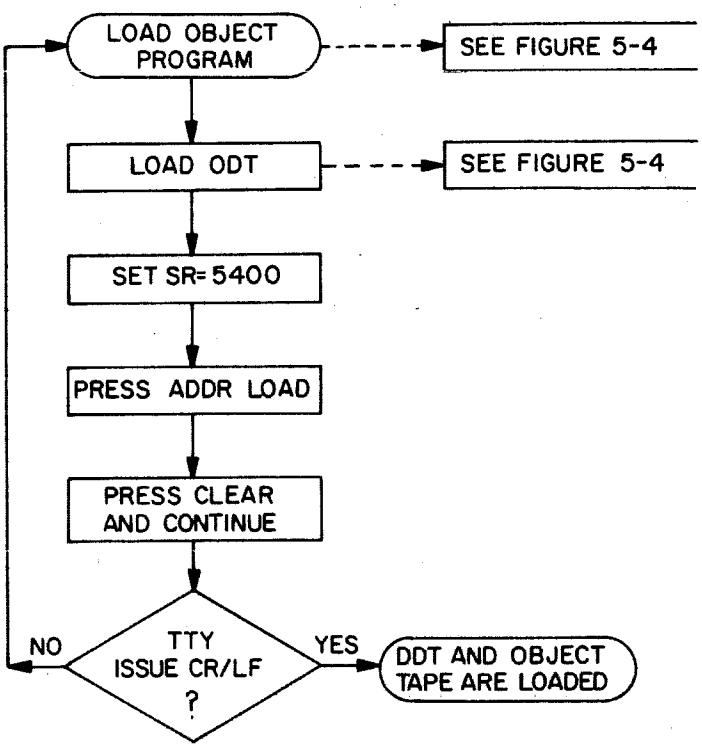



Figure 5-11 Loading DDT and Object Program

The following are now in core memory:

1. The DDT program, which occupies upper memory between registers 5240 and 7577, inclusive.
2. The user's program(s) which must not overlap the area occupied by DDT or its permanent symbol table.
3. A table of symbol definitions, extending downward from location 5237 to 5004. This table includes the definitions for all of the PDP-8 memory reference instructions, operate instructions, ten basic IOT instructions, and the combined operations CIA and LAS.

**Symbol Table Tapes**

Part of the punched output of a MACRO-8 or PAL assembly may be a tape containing the symbol definitions of the assembled program The definitions from a symbol tape are entered into

the DDT external symbol table by the procedure outlined in Figure 5-12. *Only* the LT-33 reader may be used.

Reading will continue until the end of the tape is reached or until a total of 250 symbol definitions have been read. If this maximum limit is reached, no further symbols may be added to the table until some have been deleted, even if the limit is reached in the middle of a tape. However, the user may proceed with debugging by typing EOT (CTRL/D), then turning the reader off and pressing CONTinue. The remaining symbols left unread will not be in the table.

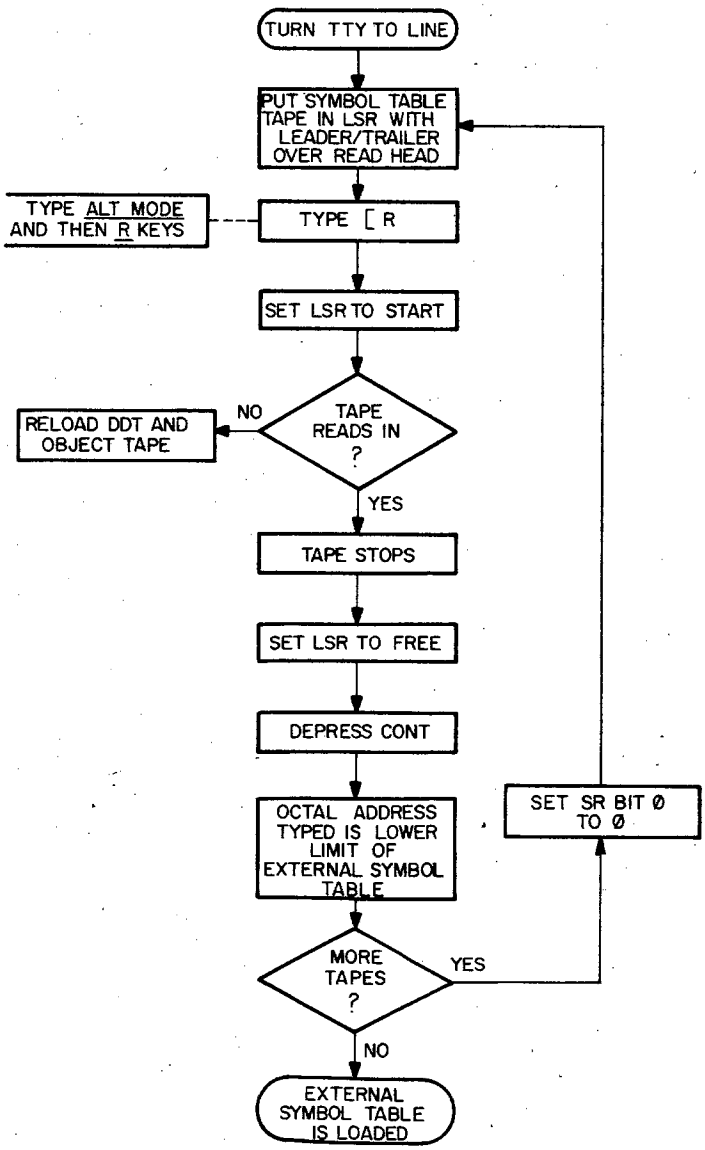



Figure 5-12 Loading External Symbol Table Tapes (LSR Only)

## DEFINING NEW SYMBOLS

Often, during the course of a debugging run, the user will want to add new symbols to the external table. This is especially so when he adds a sequence of instructions to his program as a patch elsewhere in memory. The patch is usually identified by a symbol which is the address tag of the first instruction in the patch. In order to use the symbol in subsequent debugging operations, he must add its definition to the external table as explained in the following flowchart:

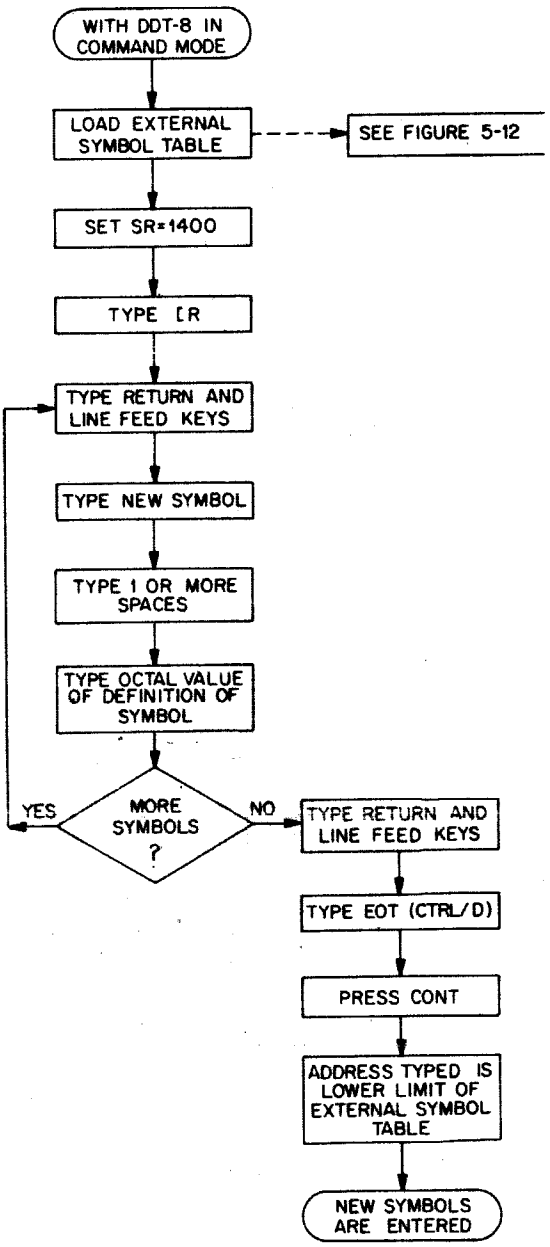



Figure 5-13 Appending New Symbols to External Symbol Table

For example: To define the symbols PATCH1 and PATCH2, the operations will appear as follows (Assume that the current limit of the table is 4775):

```
                          (CR/LF)
PATCH1 610                (CR/LF)
PATCH2 620                (CR/LF)
                            (EOT)
                  (Press CONTinue)
4665              (new limit of the table)
```

If the user makes an error while typing a definition, he *cannot* use ← to eliminate the information. The erroneous definition must be entered.

A symbol already in the table may not be redefined. Only new symbols can be added.

**NOTE**

Extra carriage return/line feed pairs may *not* be inserted between definitions; they will cause errors in subsequent table lookups when DDT is operating.

A new or updated symbol tape may be created off-line using the method described in Figure 5-8 in the section concerning the Symbolic Editor.

To completely expunge the external symbol table (for instance, when starting a new debugging run with DDT already in memory), the following command is used:

```
↑X
```

On receipt of this command, DDT removes all definitions in the external table, types a carriage return and line feed, and prints the new lower limit of the (now empty) external symbol table. DDT is then ready to accept another command. The permanent table is unaffected.

**Storage Requirements**

The operating portion of DDT occupies storage in upper memory from location 5240 to location 7577, inclusive. The permanent symbol table extends downward in memory from location 5237 to location 5004, inclusive. This table contains the definitions of the mnemonics for all the basic memory reference instructions, the operate class instructions of both Group 1 and Group 2, the combined instruction CIA and LAS, the symbol I for indirect addressing, and the basic IOT instructions: KCC, KRS, KRB, KSF, TSF, TCF, TPC, TLS, ION, and IOF. There is a list of all the symbols and definitions in the permanent table at the end of this section on DDT.

Space is reserved for the user's symbol table immediately below the permanent table. A maximum of 250 such external symbols is allowed; hence if the user's table is filled, the lower limit of space occupied by DDT is 3030. However, space not used for external symbols is available to the user. Each new symbol defined on-line uses locations in the external table.

During operation, DDT uses location 4 on page 0 for the breakpoint link; thus this register is not available to the user.

**Definitions**

A *symbol* is a string of up to six letters and numerals, the first of which must be a letter. The following are legal symbols: FIMAGE, K2, X464PQ, PMLA. The following are not acceptable:

| | |
|---|---|
| 4WD | Does not begin with a letter |
| F2.8 | Contains an illegal character |
| AN PRC | A space cannot be imbedded in a symbol |
| GANDALF | More than six characters |

A *number* is a string of up to four octal digits (integers). Hence, a number may have a maximum value of $7777_8$. The digits 8 and 9, however, may be used only as characters in a symbol.

An *expression* is a symbol, an integer, or a sequence of symbols and integers separated by any of the following *operators:*

+ An operator designating addition (arithmetic plus).

– An operator designating subtraction (arithmetic minus).

space An operator which indicates that the remainder of the expression is to be treated as the address part of an instruction

All other characters, except those for DDT control commands, are illegal.

If two or more spaces appear in succession, all but the first are ignored. Thus, TAD  TEM and TAD TEM are identical expressions.

DDT will respond to an extra carriage return (CR) with a carriage return/line feed combination (CR/LF); the extra CR's are otherwise ignored.

The following errors will cause DDT to print a question mark (?) and ignore all the information typed between the point of the error and the previous tab or CR.

1. Undefined symbol; illegal symbol.
2. Illegal character.
3. Undefined control command.
4. Cross-page addressing.

**Mode Control**

Any expression containing a symbol is *symbolic;* an expression containing only integers is *octal.* The programmer is free to use whichever mode of DDT is most convenient for the information he is typing. On output, DDT will print exclusively in one mode or the other, as determined by one of the commands described below.

**NOTE**

When DDT is first set into operation, the output mode is symbolic. "[" corresponds to typing the ALT MODE (or ESC) key.

| Command | Meaning |
|---|---|
| [O | This command causes DDT to print any subsequent item of information as an octal integer. Typed input may be symbolic or octal. If LOC = 2642 then the following commands will both print the same answer: |

```
LOC/1263
2642/1263
```

[S This command causes DDT to print any subsequent item of information as a symbolic expression. Typed input may be symbolic or octal. If LOC = 2642 and the contents of LOC = TAD DATA+4, then:

```
LOC/TAD DATA+0004
2642/TAD DATA+0004
```

If the user wishes to find the octal value of a symbolic expression typed by himself or by DDT without changing the output mode, he may use the following command:

= Typed immediately after a symbolic expression, this will cause DDT to print the value of the expression as an octal integer. For example:

```
LOC=2642
LOC/TAD DATA+0004      =1263
```

In the second example above, the prevailing output mode is symbolic and remains so after the use of the equal sign.

OUTPUT

When operating in symbolic mode, DDT will always attempt to make a symbolic expression out of the contents of an opened register, regardless of whether the contents are intended to be such or not. For example, if register DATA contains the number 6115, opening the register will result in the following line:

```
DATA/IOT+0115
```

The user can use the equal sign to ascertain the octal value:

```
DATA/IOT+0115      =6115
```

**Program Examination and Modification**

The commands and operations in Table 5-9 allow the user to examine and change the contents of any register in the PDP-8/E core memory.

**NOTE**

Be careful not to open and modify any register within the DDT symbol table or program itself. DDT does not protect itself against such intrusions, which will inevitably cause errors in operation.

**Table 5-9 DDT Commands**

| Command | Meaning |
|---|---|
| | This is the register examination character. Typed immediately after an expression, it causes DDT to print the contents of the register whose address is specified by that expression. For example, if the user types:<br><br>`LOC/`<br><br>DDT will type out the contents of LOC, thus:<br><br>`LOC/TAD DATA+0004`<br><br>The user may now change the contents of the register if he wishes:<br><br>`LOC/TAD DATA+0004  JMP LOC+10` |
| carriage return (CR) | This causes DDT to close the opened register after making the specified changes (if any) in its contents. For example:<br><br>`LOC/TAD DATA+0004  JMP LOC+10`<br><br>Typing additional CR's will have no effect on the operation of DDT. |
| line feed (LF) | If, after examining and/or modifying the contents of a location, the user wishes to open the next sequence location, he types a line feed instead of a CR. The open register is closed, and DDT then opens the next location, printing the |

**Table 5-9 DDT Commands (Cont.)**

| Command | Meaning |
|---|---|
| | address, a back slash to indicate that the register was not opened by the user, the contents of the new register, and another five spaces. For example, assume that after examining and changing the contents of LOC, the user wishes to examine the contents of LOC+1:<br><br>`LOC/TAD DATA+0004  JMP LOC+10` (LF)<br>`LOC+1/DCA DATA`<br><br>The register LOC+1 is now open.<br>Line feed may be used at any time, even if the last location examined has been closed or if other operations have intervened. For example, if the following sequence of operations occurs:<br><br>`LOC/TAD DATA+0004  JMP .+10`<br>`^O`<br>`.+5^B` (LF)<br><br>DDT will still open register LOC+1. The breakpoint address (explained shortly) has no effect on the counter within DDT which keeps track of the last opened register. |
| ↑ (up arrow) | If instead of changing the contents of a register, the user wishes to examine the register addressed by those contents, he types ↑, as follows:<br><br>`LOC/TAD DATA+0004`<br>`DATA+4/OPR+337`<br><br>The register DATA+4 is now open.<br>Note that this operation is intended for use with unmodified locations. If the user types it after typing some modifying information, the location *addressed* will be the one which is changed. For example, if the following sequence occurs:<br><br>`LOC/TAD DATA+0004  JMP LOC+10↑`<br><br>the information will be placed in DATA+4, so that the next line, printed by DDT, will look like this:<br><br>`DATA+4\JMP LOC+0010` |

**Table 5-9 DDT Commands (Cont.)**

| Command | Meaning |
|---|---|
| | The register LOC will not be changed. An indirect address modifier will not be interrupted by the ↑ operation. If, for example, the register LOC contained TAD I DATA+4, and the user typed ↑ as in the previous example, DDT would still open the register DATA+4. |
| (dot) | The *dot* is used as a symbol whose value is the address of the last previous register opened, and can be used in several ways:<br>1. To check the results of a modification.<br>`LOC/TAD DATA+0004  JMP LOC+10`<br>`•/JMP LOC+0010`<br>2. To refer to the currently open register.<br>`LOC/TAD DATA+0004  JMP .+10`<br>3. To execute any command starting at an address relative to the last opened register.<br>`LOC/TAD DATA+0004  JMP .+10`<br>`•-5[G` |
| ← (back arrow) | An error may be deleted by typing a back arrow. All information between the ← and the previous tab or CR is ignored; DDT responds by printing a tab. For example: |

```
LOC/TAD DATA+0004      JMP LC←      JMP .+10
```

## CROSS-PAGE ADDRESSES

When the user types an instruction to be placed in an open register, the address of that instruction must be in the same page as the address which contains the open register. If such a cross-page address is attempted, DDT will signal an error by typing ? and will ignore the information. For example: if LOC = 2642 and XPAG = 2770, the following sequence would result in an error indication:

```
LOC/TAD DATA+0004      DCA XPAG+20
?
```

The expression XPAG+20 is equal to 3010, which is outside the page containing LOC. The location LOC will be closed without modification.

Conversely, an expression containing symbols defined outside the page is acceptable if its value is in the current page. For example if LOC = 2642 and XPAG = 3010, the following sequence is acceptable, since XPAG − 20 has a value which brings it within the current page:

```
LOC/TAD DATA+0004       DCA XPAG-20
```

## USING COMBINED OPERATE OR IOT INSTRUCTIONS

Except for CIA and LAS, combined Operate and IOT instructions are not defined in the DDT permanent symbol table. To enter such instructions into an open register, the combination must contain no more than two mnemonics, the second of which must be CLA. Any other combination will be treated as an error, and the information will be ignored. For example, the following attempt is an error:

```
XPAG/CLA        CLA CMA
?
```

This attempt is correct:

```
XPAG/CLA        CMA CLA
```

If the desired combination does not include CLA, the user may do one of two things. He can define the combined operation as a new symbol whose value is the combined operation code. For example, the operation CLL RAR can be defined as a symbol, say, CLAR, whose value is 7110.

Alternatively, the user may enter the combined operation as an expression containing the symbol OPR. For example, the operation CLL RAR can be entered as OPR+110. The user may similarly use the symbol IOT in entering new I/O combinations.

## SPECIAL LOCATIONS

There are five registers within DDT which hold information of interest to the user. These registers may be opened and their contents changed.

To open any of the following special locations, type the ALT MODE (or ESCape) key followed by the name of the location. DDT will print a backslash (\) followed by the contents of the special location. The contents may be altered at this point in the same way as any ordinary location.

| Location | Meaning |
|---|---|
| A | When a breakpoint is encountered, the contents of the accumulator, C(AC), at that point are placed in this register. |
| Y | When a breakpoint is encountered, the C(L), where L means the link bit, at that point are placed in this register. |
| L | This register contains the address of the lower limit of a word search. Initially, C(L) = 0001. |
| U | This register contains the address of the upper limit of a word search. Initially, C(U) = 5000. |
| M | This register contains the mask used in a word search. Initially, C(M) = 7777. |

The use of these registers is explained in the following pages.

**Program Execution and Control**

The commands described in Table 5-10 allow the user to control the execution of his program.

**Table 5-10 DDT Execution Commands**

| Command | Meaning |
|---|---|
| k[G | This command causes DDT to begin the execution of the user's program, starting with the instruction in the register whose address is specified by the expression k. If a breakpoint (see below) has been requested, it is inserted just before control is passed to the user's program. For example, if the user types: |

```
BGINCG
```

**Table 5-10 (Cont.) DDT Execution Commands**

| Command | Meaning |
|---|---|
| | DDT will transfer control to location BGIN. Likewise:<br><br>`FILI-5[G`<br><br>will cause the user's program to start in the fifth register preceding the one labeled FILI.<br>Using [G without an argument is an error. DDT will ignore the command, and type ? to indicate the mistake. |
| k[B | This causes DDT to insert a *breakpoint* at the location specified by the expression k. The breakpoint is not placed immediately, however. When this command is typed, DDT stores the value of the address indicated by k. Then, when the user next types either a [G or a [C (see below) command, the breakpoint is placed just before control passes to the user's program. At that time the sequence of operations performed by DDT is as follows:<br><br>1. The contents of location k are saved in a special register.<br>2. In place of the instruction in location k, DDT substitutes the instruction, JMP I 4. Location 4 contains the address of a special breakpoint handling subroutine within DDT.<br>3. After the breakpoint has been placed, DDT passes control to the user's program.<br><br>When, during execution, the user's program encounters the location containing the breakpoint, control is immediately passed (via location 4) to the breakpoint subroutine in DDT. The C(AC) and C(L) at the point of the interruption are saved in the special registers A and Y, respectively. DDT then prints out the address of the register containing the breakpoint, followed by a right parenthesis and the contents of A as an octal number. Control has now returned to DDT, and the user is free to examine and modify his program.<br>Only one breakpoint may be in effect at one time. As soon as the user requests a new breakpoint using the B command, any previous existing breakpoint is removed. To eliminate the breakpoint entirely, the command is typed without an argument, thus:<br><br>`[B` |

**Table 5-10 (Cont.) DDT Execution Commands**

| Command | Meaning |
|---|---|
| | When the breakpoint is removed, the original contents of the break location are restored.<br>After the breakpoint has occurred and the user has examined his program and made the changes he wishes, he can cause his program to continue *from the point of the break* by means of the following command: |
| [C | This *continue* command causes DDT first to execute the instruction which was originally in the break location, and then pass control to the next location in the user's program. The breakpoint remains in effect. |

The following example illustrates the use of the three commands just described. Explanations are to the right of the commands.

| | |
|---|---|
| `FILI+7[B` | Breakpoint inserted at location FILI+7. |
| `BGIN[G` | Program execution is initiated at BGIN; program runs until breakpoint location is encountered. |
| `FILI+0007)7721` | DDT prints the address of the break location and the contents of the AC at the time of the break; note that location FILI+7 is *not* opened. |
| `...` | The user performs such examination and modification as he desires. |
| `[C` | The user's program continues, beginning with the execution of the instruction originally in FILI+7; the breakpoint remains in effect. |

Often the user would like to place a breakpoint at a location within a loop in his program. Since loops can run to thousands of repetitions, some means must be available to prevent a break from occurring every time the location is encountered. This is done using the [C command; after the breakpoint is encountered the first time, the user can specify how many times the loop must be executed before another break is to occur, as follows: After the

first breakpoint occurrence, the user wishes to wait for $250_8$ repetitions before the next break.

| | |
|---|---|
| `FILI+7[B` | The breakpoint is inserted. |
| `BGIN[G` | User program execution begins. |
| `FILI+0007)7721` | The first breakpoint occurs. |
| `250[C` | The program continues; the next breakpoint will not occur until the location FILI+7 has been encountered 250 times. |
| `FILI+0007)2534` | The next break occurs after 250 times through the loop. |

RESTRICTIONS ON USE OF BREAKPOINTS

The user must not place a breakpoint at any of the following places in his program:

1. Within any section of the program which operates with the program interrupt enabled.
2. At any location that contains an instruction which is modified during the course of the program. For example, the program contains a sequence which includes the following instructions:

```
        ISZ B

        ...
B,      TAD A
```

   A breakpoint may not be inserted at location B.

   When the user's program comes to a halt, control may be returned to DDT by setting the switch register to 5400 and pressing ADDR LOAD, CLEAR and CONTinue.
3. In a location containing a subroutine jump (JMS) which is followed by one or more arguments for that subroutine.

A breakpoint may be inserted at the point of a subroutine call if the JMS instruction is not followed by any subroutine arguments, but the breakpoint may not be removed until control has returned from the subroutine to the calling program.

WORD SEARCHES

The searching operations are used to determine if a given quantity is present in any of the locations of a particular seetion of memory. The search is initiated by the following command:

| Command | Meaning |
|---|---|
| k[W | DDT will perform a *word* search and print the address and contents of every register in the desired section of memory whose contents are equal to the value of the expression k. If the expression k is omitted, a search for the quantity 0000 masked by C(M) is assumed. |

The conditions for any search are set by the following criteria:

1. The contents of every location searched are masked by the contents of the special register M, using the Boolean AND operation. The resulting logical product is then compared with the value of k. If the two quantities are identical, the address and contents of the examined location are printed at the Teletype.

2. The search is conducted over that section of memory whose lower limit is given by C(L), and whose upper limit is given by C(U), except for the special case described in the next paragraph.

3. If C(M) = 7777 and the expression k contains any symbol in its address part (for instance, ISZ FILI+5; FILI is the symbol), the search will be conducted only on the page for which that symbol is defined, regardless of the search limits specified by C(L) and C(U). For any other case, including that where the address tag of k is defined for page 0, the search is conducted according to the limits set.

A search never alters the contents of any location examined.

Addresses and location contents are printed as symbolic expressions or octal integers, according to the mode at the time of the search.

For example: Search locations 2000 to 4000 for all occurrences of an ISZ instruction.

```
[L\0001       2000
[U\5000       4000
[M\7777       7000
ISZ[W
```

The addresses rather than tags are printed when symbols are not defined.

```
2002\ISZ 2135
2053\ISZ 2135
2111\ISZ 0017
```

The search will continue until all registers containing an ISZ are found. Note that the setting of the mask limits the investigation to the first three bits of each register, so that only instruction codes are considered.

Third example: Obtain a dump of any section of memory. The search is conducted between the limits set, and the addresses and contents of all registers in the searched section are printed.

```
[L\0001       2600
[U\5065       3000
[M\7777       0
[W
2600\0000
2601\CLA
2602\TAD 2610
```

The search will continue to the specified limit, printing the contents of every register. Note the following points: The mask is set to 0 to insure that results of every comparison are the same, i.e., 0. The search is conducted for all registers containing 0, so that the results of each comparison are equal to the desired quantity, 0. Always remember that the contents of the registers themselves are *not* altered.

## PUNCHING BINARY TAPES

After making the desired corrections and changes, the user may punch out a new binary tape of his program. This allows the debugged program to be used immediately, without waiting for the programmer to incorporate the corrections in the source program and then reassemble the program. The punching procedure given in Figure 5-14 may be used for either the Teletype console punch or the optional high-speed punch.

The punching procedure uses the following commands:

| Command | Meaning |
|---|---|
| [T | This command is used to obtain a segment of leader-trailer. |
| a;b[P | This command causes DDT to punch a block of binary tape with the information contained in the section of core memory designated by the expressions *a* (lower limit) and *b* (upper limit), inclusive. *a* and *b* may be any kind of acceptable terms (refer to Definitions at the beginning of this section). |
| [E | This command is used at the end of punching operations and causes DDT to punch a checksum block, followed by a length of trailer tape. |

**NOTE**

The user should not try to punch the section of memory between 5000 and 7600 which contains DDT.

If the user wishes to restart DDT before he has punched a complete tape (i.e., between data blocks) he must set the console switches to 5401 to preserve the checksum. Subsequent restarts must also be set to 5401 until the checksum block has been punched.

At any other time DDT may be restarted at location 5400.

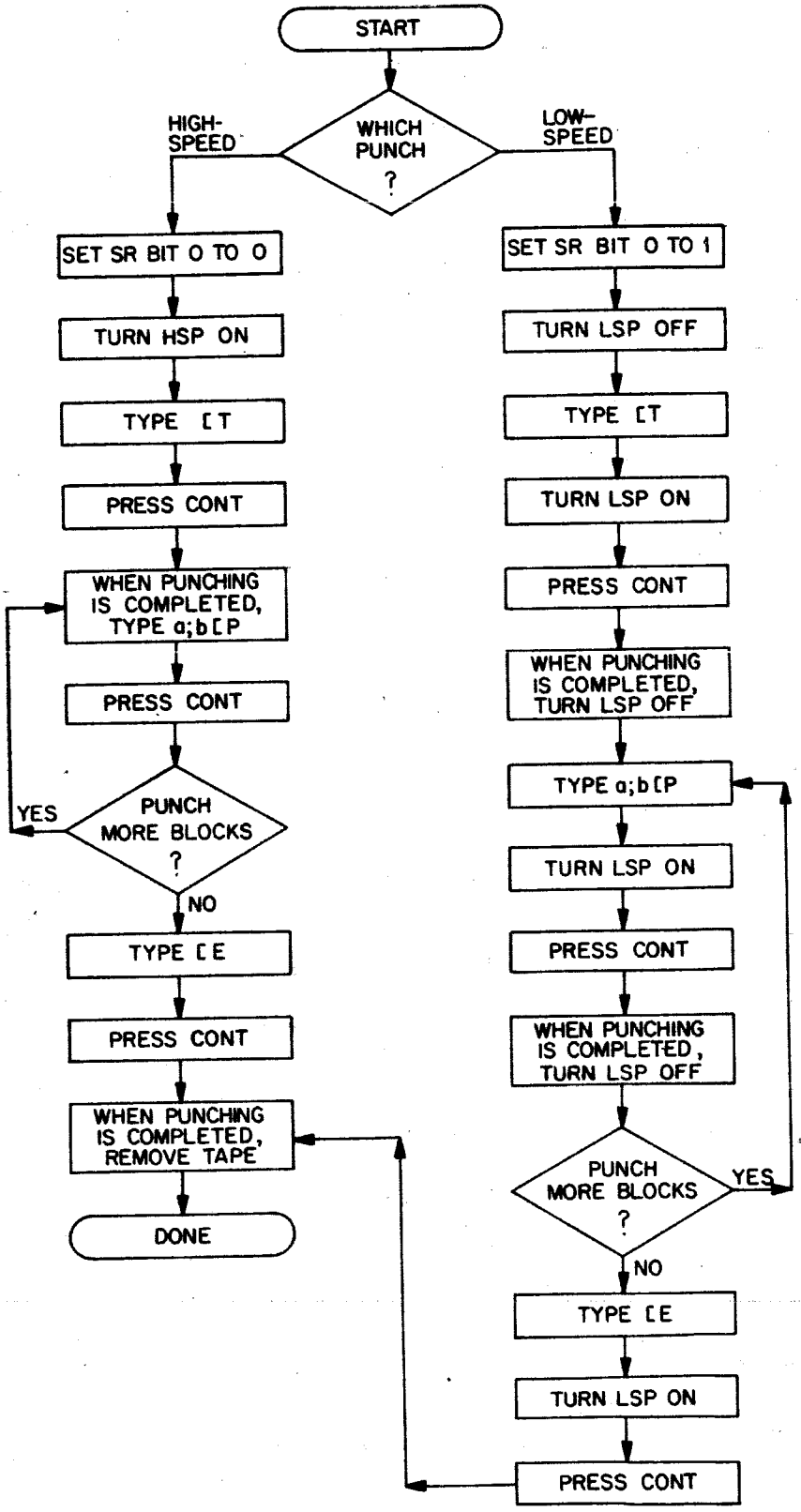



Figure 5-14 Punching Binary Tapes (DDT)

**Example Program Debugged**

The following is the third pass assembly listing of a program which adds five numbers and stores their result in the variable SUM. The programmer writes and assembles such a program (or one more complex), then loads the binary program into core along with DDT and the symbol table which was the output of assembly pass 1.

```
                *200
0200  7200      COMPSM, CLA             /INITIALIZE LOOP
0201  1213              TAD N
0202  3215              DCA INDEX
0203  1214              TAD ADDR
0204  3225              DCA POINTER
0205  3216              DCA SUM           /SET SUM TO ZERO
0206  1625      LOOP,   TAD I POINTR      /ADD THE INTERGERS
0207  2215              ISZ INDEX         /IS LOOP FINISHED
0210  5206              JMP LOOP          /NO
0211  3216              DCA SUM
0212  7402              HLT               /YES
0213  7773      N,      -5                /MINUS NR OF INTEGERS
0214  0216      ADDR,   LIST-1            /ADDRESS OF LIST
0215  0000      INDEX,  0                 /COUNTER FOR LOOP
0216  0000      SUM,    0                 /RESULT GOES HERE
0217  0001      LIST,   1                 /LIST OF NUMBERS
0220  0002              2
0221  0003              3
0222  0004              4
0223  0005              5
0224  0017      17
0225  0000      POINTR, 0                 /AUTO INDEX POINTER
                                          /TO LIST
```

In order to have DDT read in the symbol table, the programmer typed the ALT MODE and R keys (which echo as [R). DDT echoes the symbol table as shown below, following which it prints the address of the lowest memory location which is occupied by a symbol definition. The programmer is now ready to begin debugging the program.

```
[R
ADDR    0214
COMPSM  0200
INDEX   0215
LIST    0217
LOOP    0206
N       0213
POINTR  0225
SUM     0216

4750
```

The user notices at this point that POINTR was stored in location 225 instead of location 17, the * was left out where word 224 presently is. To correct this, the interaction with the computer looks as follows:

```
LOOP/TAD I POINR          TAD I 17
./TAD I 0017

.-2/DCA POINTR    DCA 17
LOOP+1[B
COMPSM[G
LOOP+0001)0001
[C
LOOP+0001)0003
[C
LOOP+0001)0006
[C
LOOP+0001)0012
N-1[B
[C
[LOOP+0004)0000

SUM/AND 0017      =0017
```

The user typed:

```
LOOP/
```

to which DDT returned the symbolic contents of the location labeled LOOP. The user changed POINTR to 17 and closed the location with the RETURN key. To check that the change had been made, the user typed:

```
./
```

where the dot indicates the current location, the slash (as above) allows the user to investigate the contents of the location. Since the contents were recorded, the user typed:

```
.-2
```

which refers to location 204 on the octal/symbolic listing. POINTR is changed to 17 in this instruction as well.

The user then inserts a breakpoint at location 207 on the listing (addressed by indicating LOOP+1). To begin execution of the program the user types:

```
COMPSM[G
```

COMPSM is the starting address of the program. [G is the command to DDT which says to transfer control to that location.

DDT prints:

```
LOOP+0001)0001
```

When the breakpoint occurs, DDT saves the contents of the AC. It then prints the address of the breakpoint, a right parenthesis, and the saved contents of the AC. The programmer sees that this is the correct value after the first time through the loop, so he issues the command [C which causes the program to cycle through the loop until it encounters the breakpoint again. Each time the user verifies the contents of the AC and has the program continue executing. Rather than check every cycle through the loop (which in this case is short, but might be very long), the user moves the breakpoint to the HLT instruction. The contents of the AC at that point was zero, which is not important.

To check the value of SUM upon program completion, the user types:

```
SUM/
```

and the computer returns what it attempts to express as a symbolic instruction. When the user types the equal sign (=), DDT returns the value of the symbolic instruction in octal.

This debugging session was very simple; the error was obvious. This is seldom the case, and with long or complex programs several debugging runs may be required. Being able to debug a program using symbolic expressions shortens the time required to arrive at a correct, workable program.

**Command Summary**

| Command | Action |
|---|---|
| space | Separation character. |
| + | Arithmetic plus. |
| — | Arithmetic minus. |
| / | Location examination character. When it follows the address location, it causes the register to be opened and its contents printed. |

| | |
|---|---|
| carriage return | Make modifications, if any, and close location. |
| line feed | Make modifications, if any, close location, and open next sequential location. |
| ↑ | When it immediately follows a location printout, it causes the location addressed therein to be opened. |
| = | Type last quantity as an octal integer. |
| . (dot) | Current location. |
| ← | Delete the line currently being typed. |
| [S | Sets DDT to print in symbolic mode. |
| [O | Sets DDT to print in octal mode. |
| N[W | *Word* search for all occurrences of the expression N masked with C(M). |
| k[B | Insert a *breakpoint* at the location specified by k. If no address is specified, remove any breakpoint. |
| n[C | *Continue* from a breakpoint n times automatically. I n is absent, it is assumed to be 1. |
| k[G | *Go* to the location specified by k. |

| Command | Action |
|---|---|
| [R | *Read* symbol table into external symbol table or define symbols on line. |
| [T | Punch leader-trailer code. |
| a;b[P | *Punch* binary tape from memory bounded by the addresses a and b. |
| [E | Punch *end* of tape: checksum and trailer. |

The following commands will open certain locations in DDT whose contents are available to the user.

| Command | Word Opened |
|---|---|
| [A | *Accumulator* storage (at breakpoints). |
| [Y | *Link* storage (at breakpoints). |
| [M | *Mask* used in search. |
| [L | *Lower* limit of search. |
| [U | *Upper* limit of search. |

**Internal Symbol Table**

| | | | | | |
|---|---|---|---|---|---|
| AND | = | 0 | CMA | = | 7040 |
| TAD | = | 1000 | CML | = | 7020 |
| ISZ | = | 2000 | RAR | = | 7010 |
| DCA | = | 3000 | RAL | = | 7004 |
| JMS | = | 4000 | RTR | = | 7012 |
| JMP | = | 5000 | RTL | = | 7006 |
| IOT | = | 6000 | IAC | = | 7001 |
| OPR | = | 7000 | SMA | = | 7500 |
| CLA | = | 7200 | SZA | = | 7440 |
| KCC | = | 6032 | SPA | = | 7510 |
| KRS | = | 6034 | SNA | = | 7450 |
| KRB | = | 6036 | SNL | = | 7420 |
| TSF | = | 6041 | SZL | = | 7430 |
| TCF | = | 6042 | SKP | = | 7410 |
| TPC | = | 6044 | OSR | = | 7404 |
| TLS | = | 6046 | HLT | = | 7402 |
| ION | = | 6001 | CIA | = | 7041 |
| IOF | = | 6002 | LAS | = | 7604 |
| KSF | = | 6031 | I | = | 400 |
| CLL | = | 7100 | | | |

## DEBUGGING WITH ODT

The Octal Debugging Technique is a debugging program which facilitates communication with and alteration of the object program. Communication is directed from the Teletype keyboard using octal numbers. ODT has the same capabilities as DDT except that the programmer must reference his program using its octal representation instead of mnemonic symbols, and ODT commands are formulated differently.

ODT occupies $600_8$ consecutive locations and one location on page zero, and can be loaded into either lower (starting address 1000) or upper (starting address 7000) core memory, depending on where the user's program resides. That is, if the user program resides in the first few pages of memory, then ODT should be loaded in the upper pages of memory, and vice versa. As with DDT, the user program cannot occupy (overlay) any location used by ODT, including the breakpoint location (location 0004 on page zero). The programmer will probably discover ODT to be more useful and convenient than DDT once he has adjusted to the octal notation.

### Features

ODT features include location examination and modification; binary punching (to the Teletype or high-speed punch) of user designated blocks of memory; octal core dumps to the Teletype using the word search mechanism, as in DDT; and instruction breakpoints to return control to ODT (breakpoints). ODT makes no use of the program interrupt facility and will not operate outside of the core memory bank in which it is reading.

The breakpoint is one of ODT's most useful features. When debugging a program, it is often desirable to allow the program to run normally up to a predetermined point, at which the programmer may examine and possibly modify the contents of the accumulator (AC), the link (L), or various instructions or storage locations within his program, depending on the results he finds. To accomplish this, ODT acts as a monitor to the user program.

The user decides how far he wishes the program to run and ODT inserts an instruction in the user's program which, when encountered, causes control to transfer back to ODT. ODT immediately preserves in designated storage locations the contents of the AC and L at the break. It then prints out the location at which

the break occurred, as well as the contents of the AC at that point. ODT will then allow examination and modification of any location of the user's program (or those locations containing the AC and L). The user may also move the breakpoint, and request that ODT continue running his program. This will cause ODT to restore the AC and L, execute the trapped instruction and continue in the user's program until the breakpoint is again encountered or the program is terminated normally.

**Using ODT**

When the programmer is ready to start debugging a new program at the computer, he should have at the console:

1. The binary tape of the new program.
2. A complete octal/symbolic program listing.
3. A binary tape of the ODT program (either high or low version).

To begin the debugging run, first be sure that the BIN Loader is in core memory, then load the ODT binary tape followed by the binary tape of the user program (see Figures 5-3 and 5-4 for a description of loading procedures with the BIN Loader).

**Operation and Storage**

STORAGE REQUIREMENTS

ODT can be run in a standard 4K PDP-8 series computer and requires 600 (octal) consecutive core locations and one location (0004) on page zero. As distributed by the Software Distribution Center, it resides in memory between 7000 and 7577 (1000 and 1577 for the low version). ODT is page-relocatable.

The source tape can be re-origined to the start of any memory page except page zero and assembled to reside in the three pages following that location, assuming they are all in the same memory bank.

ODT uses location 4 on page zero as an intercom location between itself and the user's program when executing a breakpoint. If the user wishes to change the location of the intercom word, he may do so by changing the value of ZPAT in the source and reassembling. The intercom location must remain on page zero.

## LOADING AND CALLING PROCEDURES

The user should note that ODT cannot be called as a subroutine.

ODT is normally distributed as a binary tape with the source listing available on request from the DEC Software Distribution Center and is loaded with the BIN Loader as described in Figure 5-15.

Load the binary tape of the program to be debugged in the same manner as ODT was loaded. Be sure the two do not overlap.

## STARTING PROCEDURE FOR ODT

The starting address of ODT is the address of the symbol START. For standard library versions the high version starts at 7000 and the low at 1000.

Set the starting address in the switch register. Press ADDRess LOAD, CLEAR and CONTinue. ODT will issue a carriage return and line feed to indicate that it is now running and awaiting commands from the keyboard.

To *restart* ODT without clearing the checksum, set the address of START + 1 (7001 high version or 1001 low version) into the switch register and press ADDRess LOAD, CLEAR and CONTinue on the computer console.

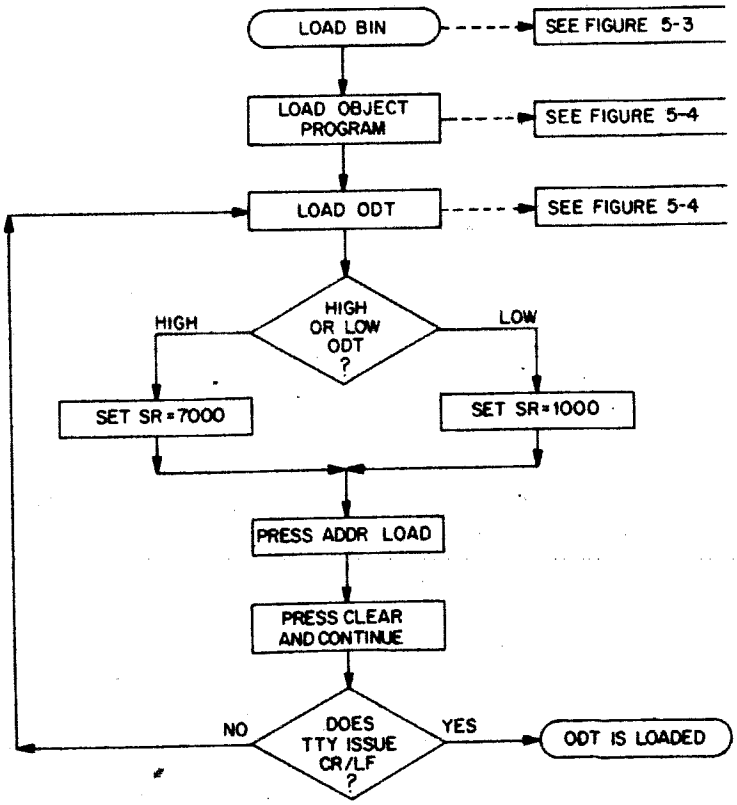



Figure 5-15 Loading ODT and the Object Tape

## SPECIAL CHARACTERS

### *Slash(/)—Open Preceding Location*

The location examination character (/) causes the location addressed by the octal number preceding the slash to be opened and its contents printed in octal. The open location can then be modified by typing the desired octal number and closing the location. Any octal number from 1 to 4 digits in length is legal input. Typing a fifth digit is an error and will cause the entire modification to be ignored and a question mark to be printed by ODT. Typing / with no preceding argument causes the latest named location to be opened (again). Typing 0/ is interpreted as / with no argument. For example:

```
400/6046
400/6046 2468?
400/6046 12345?
/6046
```

### *Return—Close Location*

If the use has typed a valid octal number after the content of a location is printed by ODT, typing the RETURN key causes the binary value of that number to replace the original contents of the opened location and the location to be closed. If nothing has been typed by the user, the location is closed but the content of the location is not changed. For example:

```
400/6046                location 400 is unchanged.
400/6046 2345           location 400 is changed to contain 2345.
/2345 6046              replace 6046 in location 400.
```

Typing another command will also close an opened register. For example:

```
400/6046 401/6031 2346
400/6046 401/2346       location 400 is closed and unchanged and
                        401 is opened and changed to 2346.
```

### *Line Feed—Close Location, Open Next Location*

The LINE FEED key has the same effect as the RETURN key, but, in addition, the next sequential location is opened and its contents printed. For example:

```
400/6046
0401 /6031 1234
0402 /5201
```

location 400 is closed unchanged and 401 is opened. User types change, 401 is closed containing 1234 and 402 is opened.

*↑(Shift/N)—Close Location, Take Contents as Memory Reference and Open Same*

The up arrow will close an open location just as will the RETURN key. Further, it will interpret the contents of the location as a memory reference instruction, open the location referenced and print its contents. For example:

```
404/3270 ↑
0470 /0212 0000
```

3270 symbolically is "DCA, this page, relative location 70," so ODT opens location 470.

*←(Shift/0) Close Location, Open Indirectly*

The back arrow will also close the currently open location and then interpret its contents as the address of the location whose contents it is to print and open for modification. For example:

```
365/5760 ↑
0360 /0426 ←
0426 /5201
```

## ILLEGAL CHARACTERS

Any character that is neither a valid control character nor an octal digit, or is the fifth octal digit in a series, causes the current line to be ignored and a question mark printed. For example:

```
4:?
4U?
```

} ODT opens no location.

```
406/4671 67K?
/4671
```

ODT ignores modification and closes location 406.

## CONTROL COMMANDS

*nnnnG—Transfer Control to User at Location nnnn*

Clear the AC then go to the location specified before the G. All indicators and registers will be initialized and the breakpoint, if any, will be inserted. Typing G alone will cause a jump to location 0.

*nnnnB—Set Breakpoint at User Location nnnn*

Instructs ODT to establish a breakpoint at the location specified before the B. If B is typed alone, ODT removes any previously established breakpoint and restores the original contents of the break location. A breakpoint may be changed to another location whenever ODT is in control, by simply typing nnnnB where nnnn is the new location. Only one breakpoint may be in effect at one time; therefore, requesting a new breakpoint removes any previously existing one.

A restriction in this regard is that a breakpoint may not be set on any of the floating-point instructions which appear as arguments of a JMS. For example:

```
TAD
DCA      } Breakpoint legal here.
JMS
FADD     Breakpoint illegal here.
```

The breakpoint (B) command does not make the actual exchange of ODT instruction for user instruction, it only sets up the mechanism for doing so. The actual exchange does not occur until a "go to" or a "proceed from breakpoint" command is executed.

When, during execution, the user's program encounters the location containing the breakpoint, control passes immediately to ODT (via location 0004). The C(AC) and C(L) at the point of the interruption are saved in special locations accessible to ODT. The user instruction that the breakpoint was replacing is restored, before the address of the trap and the content of the AC are printed. The restored instruction has not been executed at this time. It will not be executed until the "proceed from breakpoint" command is given. Any user location, including those containing the stored AC and Link, can now be modified in the usual manner. The breakpoint can also be moved or removed at this time.

An example of breakpoint usage follows the section "Continue and Iterate Loop . . ."

*A—Open C(AC)*

When the breakpoint is encountered the C(AC) and C(L) are saved for later restoration. Typing A after having encountered a breakpoint, opens for modification the location in which the AC was saved and prints its contents. This location may now be mod-

ified in the normal manner (see Slash) and the modification will be restored to the AC when the "proceed from breakpoint" command is given.

*A (line feed)—Open C(L)*

After opening the AC storage location, typing the LINE FEED key closes the AC storage location, then opens the Link storage location for modification and prints its contents. The Link location may now be modified as usual (see Slash) and that modification will be restored to the Link when the "proceed from the breakpoint" command is given.

*C—Proceed (Continue) From a Breakpoint*

Typing C, after having encountered a breakpoint, causes ODT to insert the latest specified breakpoint (if any), restore the contents of the AC and Link, execute the instruction trapped by the previous breakpoint, and transfer control back to the user program at the appropriate location. The user program then runs until the breakpoint is again encountered.

**NOTE**

If a breakpoint set by ODT is not encountered while ODT is running the object (user's) program, the instruction which causes the break to occur will not be removed from the user's program.

*nnnnC—Continue and Iterate Loop nnnn Times Before Break*

The programmer may wish to establish the breakpoint at some location within a loop of his program. Since loops often run to many iterations, some means must be available to prevent a break from occurring each time the break location is encountered. This is the function of nnnnC (where nnnn is an octal number). After having encountered the breakpoint for the first time, this command specifies how many additional times the loop is to be iterated before another break is to occur. The break operations have been described previously in the section on the B command.

Given the following program, which increases the value of the AC by increments of 1, the use of the Breakpoint command may be illustrated.

```
                *200
0200   7300             CLA CLL
0201   1206   A,        TAD ONE
0202   2207   B,        ISZ CNT
0203   5202             JMP B
0204   5201             JMP A
0205   7402             HLT
0206   0001   ONE,      1
0207   0000   CNT,      0

A        0201
B        0202
CNT      0207
ONE      0206



0201B
200G
0201  (0000
C
0201  (0001
C
0201  (0002
4C
0201  (0007
```

ODT has been loaded and started. A breakpoint is inserted at location 0201 and execution stops here showing the AC initially set to 0000. The use of the Proceed command (C) executes the program until the breakpoint is again encountered (after one complete loop) and shows the AC to contain a value of 0001. Again execution continues, incrementing the AC to 0002. At this point, the command 4C is used, allowing execution of the loop to continue 4 more times (following the initial encounter) before stopping at the breakpoint. The contents of the AC have now been incremented to 0007.

*M—Open Search Mask*

Typing M causes ODT to open for modification the location containing the current value of the search mask and print its contents. Initially the mask is set to 7777. It may be changed by opening the mask location and typing the desired value after the value printed by ODT, then closing the location.

*M Line Feed—Open lower search limit*

The word immediately following the mask storage location contains the location at which the search is to begin. Typing the LINE FEED key to close the mask location causes the lower search limit to be opened for modification and its contents printed. Initially the

lower search limit is set to 0001. It may be changed by typing the desired lower limit after that printed by ODT, then closing the location.

*M Line Feed—Open upper search limit*

The next sequential word contains the location with which the search is to terminate. Typing the LINE FEED key to close the lower search limit causes the upper search limit to be opened for modification and its contents printed. Initially, the upper search limit is the beginning of ODT itself, 7000 (1000 for low version). It may also be changed by typing the desired upper search limit after the one printed by ODT, then closing the location with the RETURN key.

*nnnnW—Word Search*

The command nnnnW (where nnnn is an octal number) will cause ODT to conduct a search of a defined section of core, using the mask and the lower and upper limits which the user has specified, as indicated above. Word searching with ODT is similar to word searching with DDT. The searching operations are used to determine if a given quantity is present in any of the locations of a particular section of memory.

The search is conducted as follows: ODT masks the expression nnnn which the user types preceding the W and saves the result as the quantity for which it is searching. (All masking is done by performing a Boolean AND between the contents of the mask word, C(M), and the word containing the instruction to be masked.) ODT then masks each location within the user's specified limits and compares the result to the quantity for which it is searching. If the two quantities are identical, the address and the actual unmasked contents of the matching location are printed and the search continues until the upper limit is reached.

A search never alters the contents of any location. For example: search locations 3000 to 4000 for all ISZ instructions, regardless of what location they refer to (i.e. search for all locations beginning with an octal 2).

| | |
|---|---|
| `M7777 7000` | Change the mask to 7000, open lower search limit. |
| `7453/0001 3000` | Change the lower limit to 3000, open upper limit. |

```
7454/7000 4000

2000W

2000 /2467
3057 /2501
3124 /2032
4000 /2152
```

Change the upper limit to 4000, close location.

Initiate the search for ISZ instructions.

These are 4 ISZ instructions in this section of core.

## PUNCH COMMANDS

### *T—Punch Leader Tape*

ODT is capable of producing leader (code 200) tape on-line. This is done by typing T and *then* turning ON the punch. When enough leader has been punched, turn off the punch and hit the HALT key on the computer console. It is *imperative* that the punch be turned OFF before typing again on the Teletype keyboard, since anything typed will also be punched if the punch is left on. To issue any further commands, reload the starting address (1000 or 7000), then press the ADDR LOAD, CLEAR and CONTinue keys on the computer console.

### *nnnn;mmmmP—Punch Binary*

To punch a binary core image of a particular section of core, the above command is used where nnnn is the initial (octal) address and mmmm is the final (octal) address of the section of core to be punched. The computer will halt (with 7402 displayed) to allow the user to turn ON the punch. Pressing the CONTinue key on the console initiates the actual punching of the block. The punching terminates without having punched a checksum, to allow subsequent blocks to be punched and to allow an all inclusive checksum to be punched at the end by a separate command. This procedure is optional, however, and the user may punch individually checksummed blocks.

Binary tapes may be punched using either the low-speed or high-speed punches. If using the low-speed punch, it is imperative that the punch be turned OFF before typing commands, since the keyboard and punch are linked. Using the high-speed punch requires switch manipulation. The flowchart in Figure 5-16 details the procedures involved in punching binary tapes.

### *E—Punch Checksum and Trailer*

Given the E command, ODT will halt to allow the punch to be turned on. Pressing the CONTinue key on the console will cause it

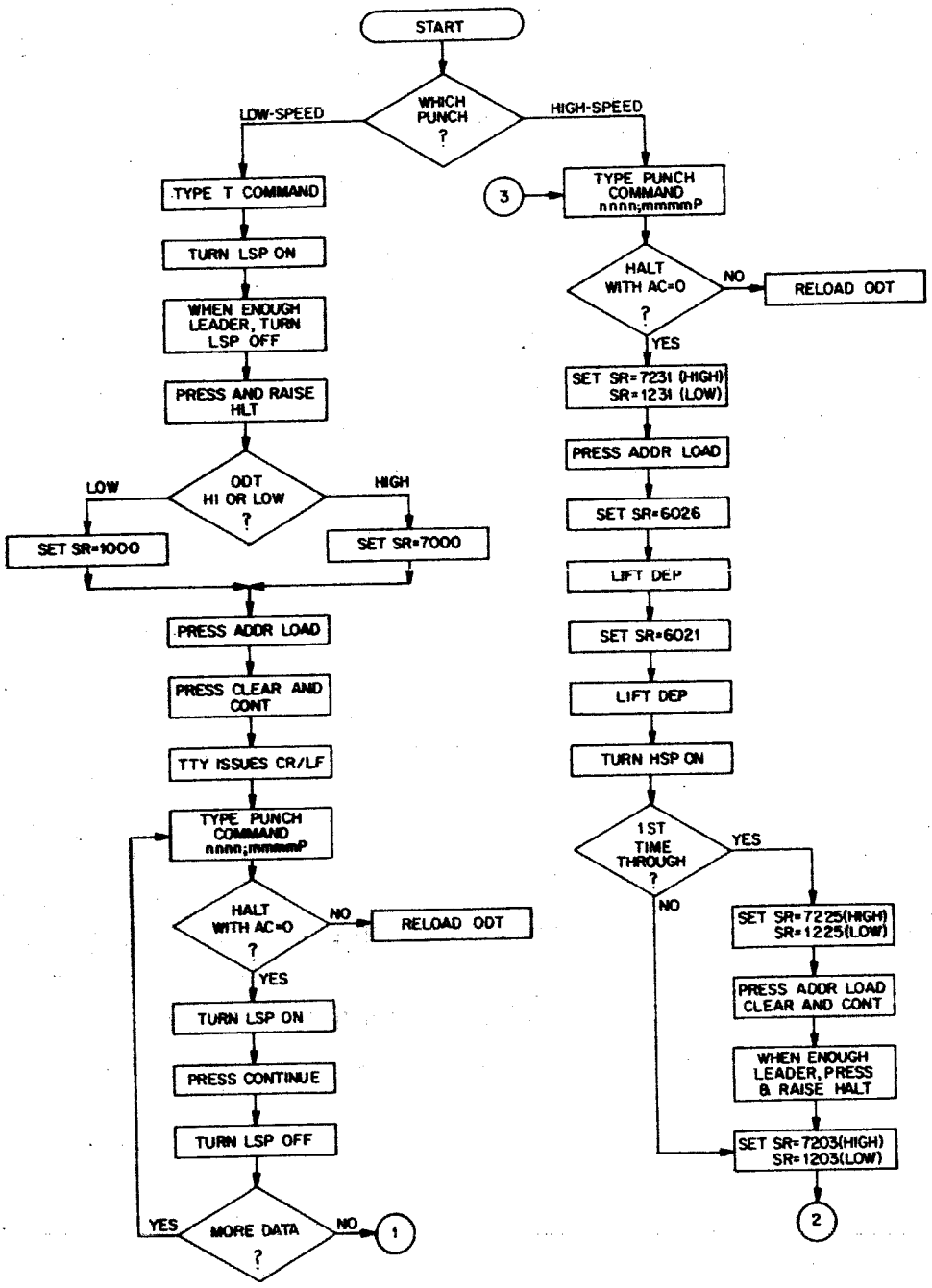



Figure 5-16 Punching Binary Tapes (ODT)

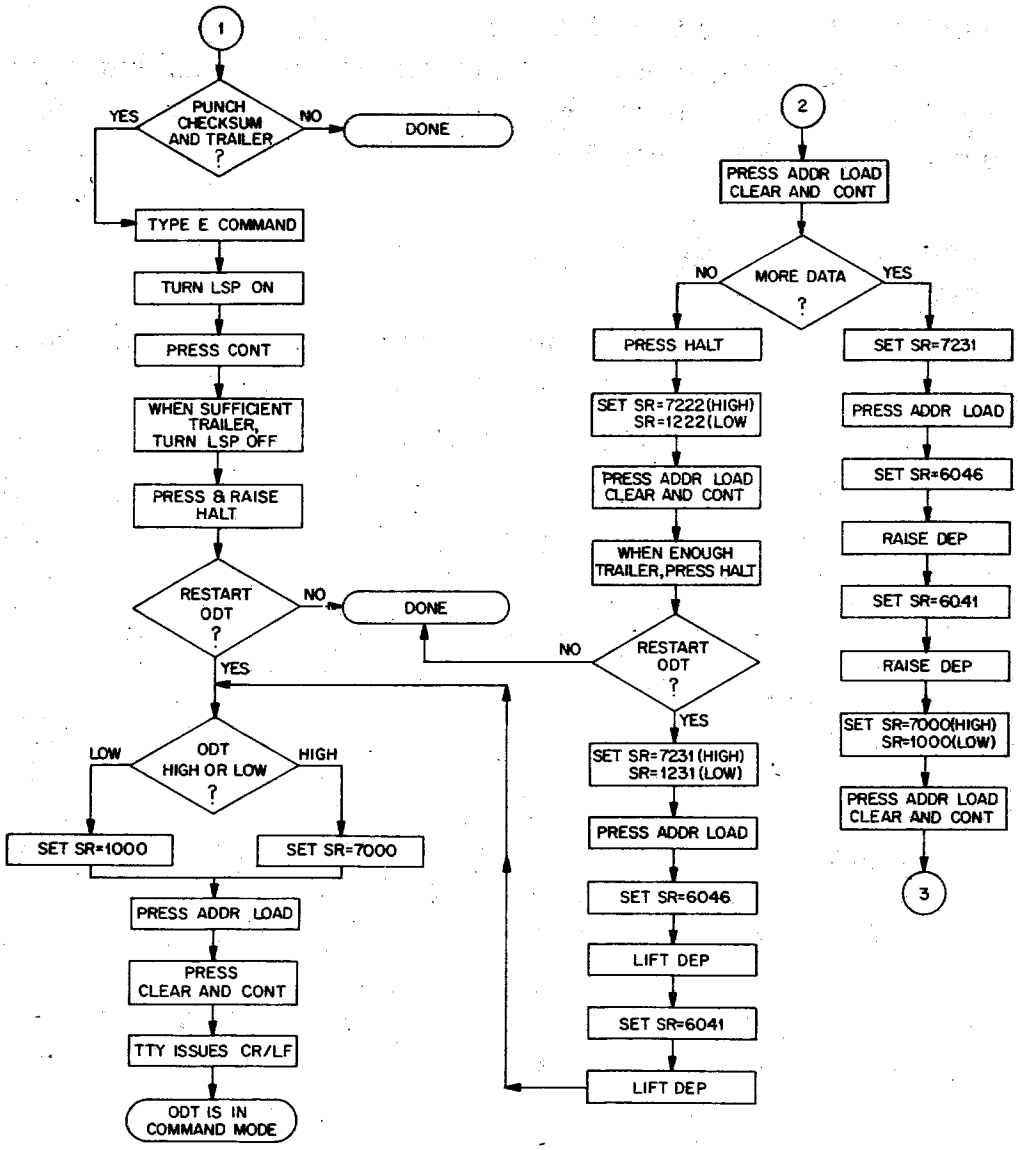



Figure 5-16 Punching Binary Tapes (ODT) (cont)

to punch the accumulated checksum for the preceding block(s) of binary output followed by trailer (code 200) tape. When a sufficient length of trailer has been output, turn OFF the punch and press the HALT key on the console. To continue using ODT, reload the starting address (1000 or 7000), then press the ADD LOAD, CLEAR and CONTinue keys on the console.

The binary tape produced by ODT can now be loaded into core and run. However, the changes should be made to the symbolic source tapes as soon as possible.

**Additional Techniques**

TTY I/O-FLAG

Sometimes the program being debugged may require that the Teletype (TTY) flag be up before it can continue output, i.e., the program output routine will be coded as follows:

```
TSF
JMP .-1
TLS
```

Since ODT normally leaves the TTY flag in an off state, the above coding will cause the program to loop at the JMP.-1. To avoid this, ODT may be modified to leave the TTY flag in the on state when transferring control through either a "go to" or a "continue" command. This modification is accomplished by changing location XCONT-3 (normally at 7341) to a NOP (7000). To make the actual change, load ODT as usual. Open location XCONT-3 and modify it as follows:

```
7341/6042 7000          (RETURN key) for high version
1341/6042 7000          (RETURN key) for low version
```

CURRENT LOCATION

The address of the current location or last location examined is remembered by ODT and remains the same, even after the commands G, C, B, T, E, and P. This location may be opened for inspection merely by typing the slash (/) character.

## PROGRAMS WRITTEN IN ODT COMMANDS

ODT will also correctly read tapes prepared off-line (e.g., a tape punched with 1021/1157↑7775 will cause location 1021 to be opened and changed to 1157; then the memory reference address 157 will be opened and changed to 7775. This procedure will work with breakpoints, continues, punch commands, etc. Thus, debugging programs may be read into ODT to execute the program, list locations of interest, modify locations, etc.

## INTERRUPT PROGRAM DEBUGGING

ODT executes an IOF when a breakpoint is encountered. (It does not do this when more iterations remain in an nnnnC command.) This is done so that an interrupt will not occur when ODT prints the breakpoint information. ODT thus protects itself against spurious interrupts and may be used safely in debugging programs that turn on the interrupt mode.

However, the user must remember that ODT does not know whether the interrupt was on when the breakpoint was encountered, and hence it does not turn on the interrupt when transferring control back to the program after receiving a "go" or a "continue" command.

## OCTAL DUMP

By setting the search mask to zero and typing W, all locations between the search limits will be printed on the Teletype. An Octal Memory Dump program (DEC-8I-RZPA-D) is available from the Software Distribution Center upon request.

## INDIRECT REFERENCES

When an indirect memory reference instruction is encountered, the actual address may be opened by typing ↑ and ← (SHIFT/N and SHIFT/O, respectively).

### Errors

The only legal inputs are control characters and octal digits. Any other character will cause the character or line to be ignored and a question mark to be printed by ODT. Typing G alone is an error. It must be preceded by an address to which control will be transferred. This will elicit no question mark also if not preceded by an address, but will cause control to be transferred to location 0.

Typing any punch command with the punch ON is an error and will cause ASCII characters to be punched on the binary tape. This means the tape cannot be loaded and run properly.

**Programming Notes Summary**

ODT will not operate outside of the memory field in which it is located.

ODT must begin at the start of a memory page (other than page zero) and must be completely contained in one memory field.

ODT will not turn on the program interrupt, since it does not know if the user's program is using the interrupt. It does, however, turn off the interrupt when a breakpoint is encountered, to prevent spurious interrupts.

The user's program must not use or reference any core locations occupied or used by ODT, and vice versa.

Register ZPAT is used as an intercom location by ODT when executing a breakpoint. In library distributed versions ZPAT = 0004. This location must be left free by the user since it is filled with an address within ODT which is used to transfer control between the user program and ODT.

Breakpoints are fully invisible to "open location" commands; however, breakpoints may not be placed in locations which the user program will modify in the course of execution or the breakpoint will be destroyed.

If a trap set by ODT is not encountered by the user's program, the breakpoint instruction will not be removed.

ODT can be used to debug programs using floating-point instructions, since the intercom location is 0004, and since breakpoints may be set on a JMS with arguments following.

This version of ODT will operate on a Teletype with an ALT MODE key or an ESCape key.

To restart ODT without clearing the checksum, see the section on Starting Procedures for ODT.

The high-speed punch may be used by patching three locations *after* typing the punch command. See the section on Binary Tapes from the High-Speed Punch.

## Command Summary

| | |
|---|---|
| nnnn/ | Open location designated by the octal number nnnn. |
| / | Reopen latest opened location. |
| RETURN key | Close previously opened location. |
| LINE FEED key | Close location and open the next sequential one for modification. |
| ↑ (SHIFT/N) | Close location, take contents of that location as a memory reference and open it. |
| ← (SHIFT/O) | Close location, open indirectly. |
| Illegal character | Current line typed by user is ignored, ODT types ? (CR/LF). |
| nnnnG | Transfer program control to location nnnn. |
| nnnnB | Establish a breakpoint at location nnnn. |
| B | Remove the breakpoint. |
| A | Open for modification the location in which the contents of AC were stored when the breakpoint was encountered. |
| C | Proceed from a breakpoint. |
| nnnnC | Continue from a breakpoint and iterate past the breakpoint nnnn times before interrupting the user's program at the breakpoint location. |
| M | Open the search mask. |
| LINE FEED key | Open lower search limit. |
| LINE FEED key | Open upper search limit. |
| nnnnW | Search the portion of core as defined by the upper and lower limits for the octal value nnnn. |
| T | Punch leader. |
| nnnn;mmmmP | Punch a binary core image defined by the limits nnnn and mmmm. |
| E | Punch checksum and trailer. |

# advanced programming techniques

## input/output programming

## dectape programming

## floating-point packages

# chapter 6
# input/output programming

## INTRODUCTION

Programming a computer to do calculations is of little use unless there is some means of obtaining the result of the calculations from the machine. In most applications, it is also necessary to supply the computer with data before calculations may be performed. A programmer must be able to translate information efficiently between the computer and the peripheral devices that supply input or serve as a means of output.

There are three methods for the transfer of information between input/output (I/O) devices and PDP-8 series computers. The first two methods provide for computer control over the transfer. One such method is *programmed transfer*, in which instructions to accept or transmit information are included at some point in the program. Programmed transfers are program initiated and executed under program control.

Information may also be transferred via *program interrupt*, a standard feature of PDP-8 series computers that allows I/O devices to signal the computer when they are ready to transfer information. The computer interrupts its normal flow, jumps to a special routine which processes the information, and then returns to the point at which the main program was interrupted. Program interrupt transfers are device initiated and executed under program control.

Both programmed transfers and program interrupt transfers use the accumulator as the *buffer*, or storage area, for all data transfers. Since data may be transferred only between the accumulator and the device, only one 12-bit word at a time may be transferred by programmed transfer or by program interrupt.

The third method of data transfer is the *data break*. A data

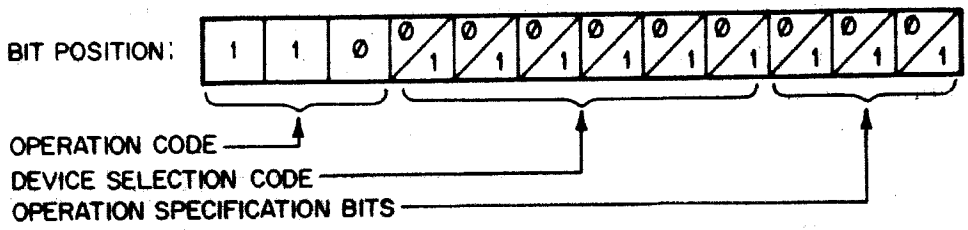
break is essentially device controlled and allows for direct exchange of large quantities of information between the I/O device and core memory. It differs from the previous two types of transfer in that there are no program instructions to handle the actual transfer, and the accumulator is not used as a buffer. Data break transfers are device initiated and device controlled.

## PROGRAMMED DATA TRANSFERS

Programmed transfers of information are accomplished by program instructions. The instructions used are similar to the operate microinstructions in that there is no need to specify an address in memory. All programmed transfers occur directly between the accumulator and the I/O device. Since many different devices could be connected to one computer and each device might transfer information at some time, the instruction must identify the proper device for each transfer. It must also specify the exact nature of the function to be performed.

### IOT Instruction Format

The instructions used to perform programmed data transfers are called input/output transfer (IOT) instructions. An IOT instruction is a 12-bit word that has the following format:




An IOT instruction is divided into three parts: operation code, device selection code, and operation specification bits. The first three bits of the instruction contain the *operation code.* These bits are always set to $6_8$ (110) to specify an IOT instruction.

The next six bits of the instruction contain the *device selection code.* This code is transmitted to all peripheral equipment whenever the IOT instruction is executed. A device selector within each peripheral monitors the device codes. When the selector recognizes

a code as that device's assigned code, the device accepts the last three bits of the IOT instruction.

The last three IOT instruction bits are the *operation specification bits*, which may be set to specify up to eight functions or combinations of functions. If a device is capable of performing more functions than can be coded into the three operation specification bits, then more than one device code must be assigned to that device.

**Checking Ready Status**

The computer operates much faster than most peripheral devices. For this reason, I/O routines must check the status of a device before executing an IOT instruction to ensure that the device is not still performing a previous operation. Device status is signalled through a system of one-bit registers called *flags*. Every I/O device has a device flag which is set to 1 as soon as the device finishes a current operation and is ready to begin a new operation. If the flag is cleared (set to 0), the device is still performing the operation specified by the last IOT instruction received. Ready status is usually checked by means of a skip-on-flag IOT instruction, such that the computer does not skip out of a waiting loop until the I/O device is ready to begin a new operation.

**Instruction Uses**

In general, for each I/O device there are at least three instructions:

1. An instruction to transfer information and/or operate the device.
2. An instruction to test the ready status of the device and skip on the ready (or not-ready) status of the device.
3. An instruction to clear or set the device flag.

These instructions may be microprogrammed. In particular, the instructions to clear the flag and to operate the device are often combined.

Specific instructions for various devices are presented in the following sections. The Teletype unit is described in depth to illustrate the fundamentals of programming data transfers. The general techniques developed for the Teletype unit may be extended to other I/O devices.

### ASCII Code

The ASCII code (American Standard Code for Information Interchange) is described in Appendix B. Many of the programs presented in this chapter use ASCII code to transmit information to the PDP-8/E. Note that the ASCII code for octal digits 0 through 7 is the sum of the digit plus $260_8$.

## PROGRAMMING THE TELETYPE UNIT

One of the most common I/O devices is the Teletype unit, which consists of a keyboard, printer, paper tape reader, and paper tape punch. The Teletype unit can use either the keyboard or the paper tape reader to provide input information to the computer and either the printer or the paper tape punch to accept output information from the computer. The Teletype is therefore assigned two device codes. Functioning as an input device, the keyboard/reader is assigned the device code $03_8$, and functioning as an output device, the printer/punch is assigned the device code $04_8$.

### Keyboard/Reader Instructions

Figure 6-1 shows the format for the keyboard/reader IOT instruction. Table 6-1 lists the mnemonic instructions used to set bits 9, 10 and 11.

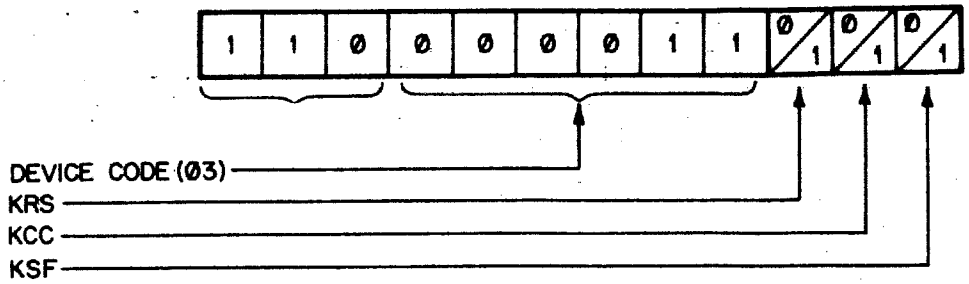



Figure 6-1 Keyboard/Reader Instruction Format

Figure 6-2 shows a program using the keyboard/reader IOT instructions to read one ASCII character from the keyboard or paper tape reader. This program does not print the character on the teleprinter. It merely stores the ASCII code for the character in core memory location STORE.

**Table 6-1 Keyboard/Reader Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| KCF | 6030 | Clear keyboard/reader flag without operating the device. |
| KSF | 6031 | Skip the next instruction if the keyboard/reader flag is a 1. |
| KCC | 6032 | Clear the accumulator and the keyboard/reader flag. |
| KRS | 6034 | Read a character from the keyboard/reader buffer. The keyboard/reader flag is set when the operation is completed. |
| KIE | 6035 | Enable the keyboard/reader to cause program interrupts if accumulator bit 11 is a 1. Disable the keyboard/reader from causing interrupts if accumulator bit 11 is a zero.[1] |
| KRB | 6036 | Clear the accumulator and the keyboard/reader flag, and read a character from the keyboard/reader buffer. This instruction is a microprogrammed combination of KCC and KRS. |

```
*200Ø
INPUT,  KCC              /CLEAR KEYBOARD FLAG
        JMS LISN         /ENTER SUBROUTINE
        DCA STORE        /STORE ASCII CHARACTER
        HLT              /HALT UPON COMPLETION
LISN,   Ø                /LISN SUBROUTINE
        KSF              /KEYBOARD FLAG RAISED YET?
        JMP .-1          /NO: CHECK AGAIN
        KRB              /YES: READ THE CHARACTER
        JMP I LISN       /RETURN TO MAINLINE
STORE,  Ø                /CHARACTER STORAGE
$
```

Figure 6-2 Coding to Accept One ASCII Character

[1] Use of this instruction will be discussed later in this chapter.

The program begins with a KCC instruction. In general, any program should begin by clearing the flags of all I/O devices to be used later in the program. If the above program is started at location 0200, it will proceed to the KSF, JMP.−1 loop, and continue looping indefinitely until a key on the Teletype unit is pressed or a paper tape is loaded into the reader. As soon as the ASCII code for a character has been assembled in the keyboard/reader buffer register, the keyboard/reader flag is set and the program skips out of the waiting loop. The content of the buffer is then transferred into the accumulator, and the keyboard/reader buffer and flag are cleared.

**Printer/Punch Instructions**

Figure 6-3 shows the format for the printer/punch IOT instructions. Table 6-2 lists the mnemonic instructions used to set bits 9, 10 and 11.

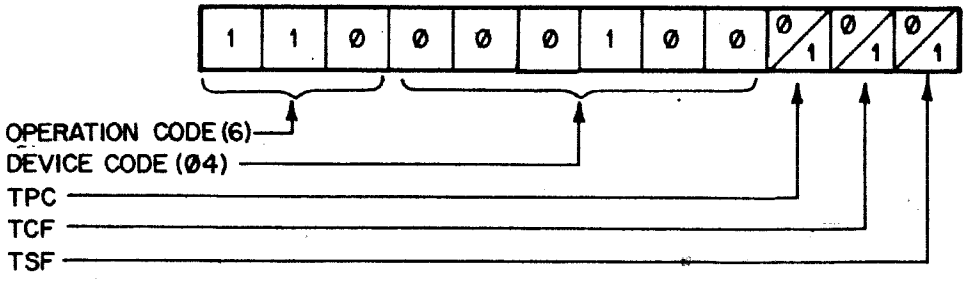



Figure 6-3 Printer/Punch Instruction Format

The program presented in Figure 6-4 prints out one ASCII character which is stored in core memory location HOLD. It begins by clearing the accumulator and executing a TLS instruction. This has no effect on the printer/punch, since 000 is the ASCII code for a blank, but it serves to clear the printer/punch buffer and then raise the device flag. If this instruction had not been included, the flag would never be raised and the program would remain in the TSF, JMP.−1 loop indefinitely. Instead, the flag is set as soon as execution of the first TLS instruction is complete,

**Table 6-2 Printer/Punch Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| TFL | 6040 | Set the printer/punch flag. |
| TSF | 6041 | Skip the next instruction if the printer/punch flag is a 1. |
| TCF | 6042 | Clear the printer/punch flag. |
| TPC | 6044 | Load the contents of accumulator bit positions 5-11 into the printer/punch buffer and operate the printer/punch. The printer/punch flag is set when the operation is completed. |
| TSK | 6045 | Skip the next sequential instruction if either the printer/punch interrupt request flag or the keyboard/reader interrupt request flag is set.[2] |
| TLS | 6046 | Clear the printer/punch flag, load the contents of accumulator bit positions 5-11 into the printer/punch buffer and operate the printer/punch. This instruction is a microprogrammed combination of TCF and TPC. |

```
*200Ø
OUTPUT, CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /RAISE PRINTER FLAG
        TAD HOLD         /GET THE CHARACTER
        JMS TYPE         /ENTER SUBROUTINE
        HLT              /HALT UPON COMPLETION
TYPE,   Ø                /TYPE SUBROUTINE
        TSF              /PRINTER FLAG RAISED YET?
        JMP .-1          /NO: CHECK AGAIN
        TLS              /YES: PRINT THE CHARACTER
        CLA CLL          /CLEAR ACCUMULATOR AND LINK
        JMP I TYPE       /RETURN TO MAINLINE
HOLD,   243              /STORED ASCII CHARACTER
$
```

Figure 6-4 Coding to Print One ASCII Character

[2] Use of this instruction will be discussed later in this chapter.

permitting the program to escape the skip loop and execute the second TLS instruction. Finally, the program again clears the accumulator. It is advisable to clear the accumulator at the end of any subroutine, unless meaningful data is contained in it.

**Format Routines**

Input and output routines are often written in the form of subroutines similar to the TYPE subroutine in the previous example. Figure 6-5 presents a carriage return/line feed subroutine that calls the TYPE subroutine to execute a carriage return and line feed on the teleprinter. Similar subroutines could be written to tab space the carriage a given number of spaces or to ring the bell of the Teletype by using the respective codes for these nonprinting characters. If such subroutines are commonly used in a program, they should be placed on page 0 (or a pointer to the subroutine should be placed on page 0) to facilitate reaching the routine from all memory locations.

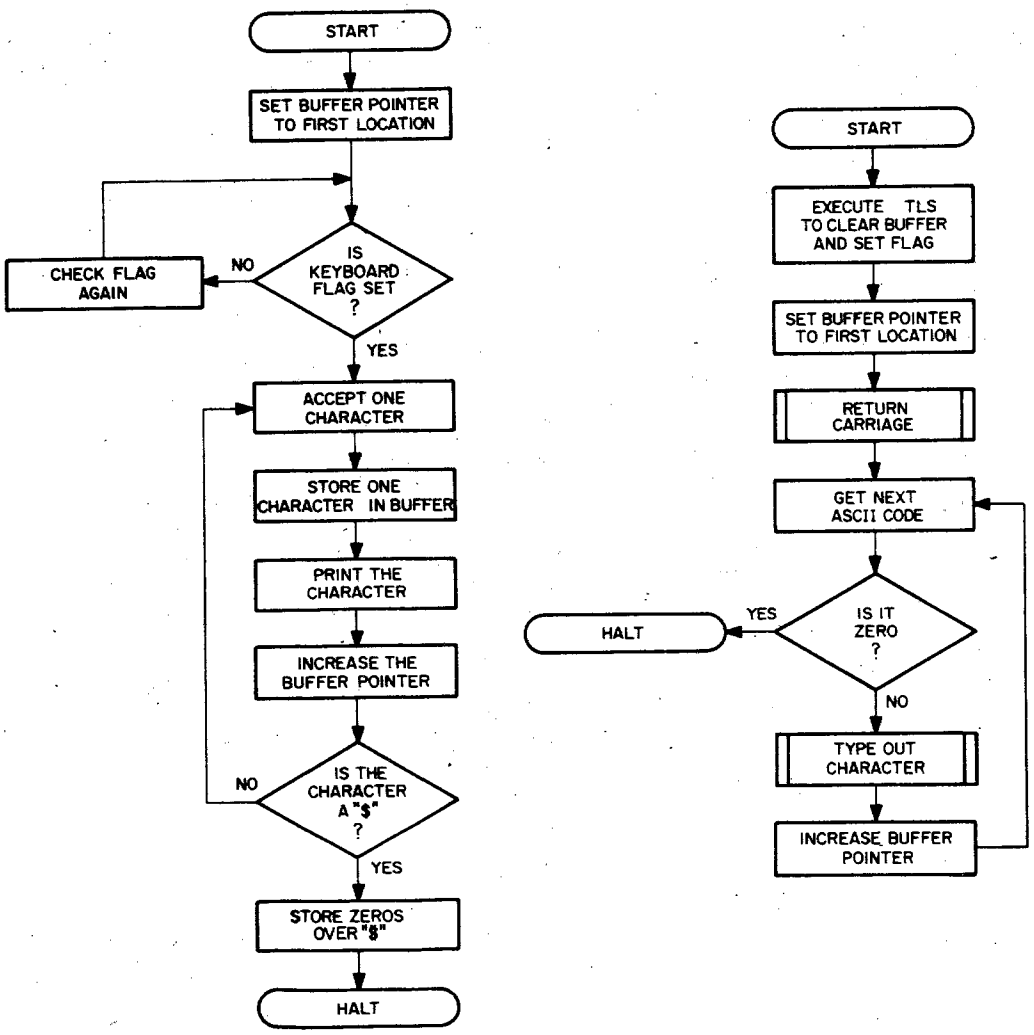
```
*300
CRLF,   0               /CRLF SUBROUTINE
        CLA CLL         /CLEAR ACCUMULATOR AND LINK
        TLS             /RAISE PRINTER FLAG
        TAD K215        /GET ASCII CARRIAGE RETURN
        JMS TYPE        /PRINT IT
        TAD K212        /GET ASCII LINE FEED
        JMS TYPE        /PRINT IT
        JMP I CRLF      /RETURN TO MAINLINE
K215,   215             /ASCII CARRIAGE RETURN
K212,   212             /ASCII LINE FEED
TYPE,   0               /TYPE SUBROUTINE
        TSF             /PRINTER FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        TLS             /YES: TYPE THE CHARACTER
        CLA CLL         /CLEAR ACCUMULATOR AND LINK
        JMP I TYPE      /RETURN
$
```

Figure 6-5 Carriage Return/Line Feed Subroutine

**Text Routines**

The examples in Figures 6-2 and 6-4 may be expanded to accept and print more than one character. Figures 6-6 and 6-7 illustrate one such expansion. These two programs are compatible in that the characters accepted by the first program may be typed

out by running the second program. The program in Figure 6-6 continues to accept character input until a dollar sign ($) is typed at the keyboard. It then stores 0000 in the next core location and halts. The program in Figure 6-7 types the characters whose ASCII codes were stored by the first program, and halts when a location with contents equal to zero is reached. Both programs use locations beginning at 2000 as a storage buffer for the ASCII characters. The following flowcharts illustrate the techniques used in the program coding.



```
*200
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TAD BUFF         /SET UP BUFFER SPACE
        DCA BUFFPT       /AND INITIALIZE POINTER
LISN,   KSF              /KEYBOARD STRUCK YET?
        JMP .-1          /NO: CHECK AGAIN
        KRB              /YES: GET THE CHARACTER
        TLS              /ACKNOWLEDGE IT ON PRINTER
        DCA I BUFFPT     /STORE THE CHARACTER
        TAD I BUFFPT     /CHECK FOR TERMINAL
        TAD MDOLAR       /DOLLAR SIGN ($)
        SNA              /IS CHARACTER A "$"?
        JMP DONE         /YES: STORE ZERO OVER $
        ISZ BUFFPT       /NO: INCREMENT POINTER
        JMP LISN         /GET ANOTHER CHARACTER
DONE,   CLA CLL          /CLEAR ACCUMULATOR
        DCA I BUFFPT     /STORE ZERO IN LAST
        HLT              /BUFFER LOCATION AND HALT
BUFF,   2000             /BUFFER BEGINS AT
BUFFPT, 0                /CORE LOCATION 2000
MDOLAR, 7534             /TWO'S COMPLEMENT OF 0244
$
```

Figure 6-6 Program to Accept ASCII Characters

```
*300
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /RAISE PRINTER FLAG
        TAD BUFF         /SET UP BUFFER SPACE AND
        DCA BUFFPT       /INITIALIZE BUFFER POINTER
        JMS CRLF         /RETURN CARRIAGE
CHRTYP, TAD I BUFFPT     /GET A CHARACTER
        SNA              /IS IT ALL ZEROS?
        HLT              /YES: HALT
        JMS TYPE         /NO: TYPE THE CHARACTER
        ISZ BUFFPT       /INCREMENT BUFFER POINTER
        JMP CHRTYP       /GET ANOTHER CHARACTER
CRLF,   0                /CRLF SUBROUTINE
        TAD K215         /GET ASCII CARRIAGE RETURN
        JMS TYPE         /PRINT IT
        TAD K212         /GET ASCII LINE FEED
        JMS TYPE         /PRINT IT
        JMP I CRLF       /AND RETURN
TYPE,   0                /TYPE SUBROUTINE
        TSF              /PRINTER READY YET?
        JMP .-1          /NO: CHECK AGAIN
        TLS              /YES: TYPE CHARACTER
        CLA CLL          /CLEAR ACCUMULATOR
        JMP I TYPE       /AND RETURN
BUFF,   2000             /BUFFER BEGINS AT
BUFFPT, 0                /CORE LOCATION 2000
K215,   215              /ASCII CARRIAGE RETURN
K212,   212              /ASCII LINE FEED
$
```

Figure 6-7 Program to Print Out ASCII Characters

The program to print ASCII characters may be specialized to print a specific message, as in the program example of Figure 6-8. This routine uses autoindex registers in place of the ISZ instruction. It types, "HELLO!".

```
*300
HELLO,  0               /MESSAGE SUBROUTINE
        CLA CLL         /CLEAR ACCUMULATOR AND LINK
        TLS             /RAISE PRINTER FLAG
        TAD CHARAC      /SET UP AUTOINDEX REGISTER
        DCA IR1         /FOR GETTING CHARACTERS
        TAD M6          /SET UP COUNTER FOR
        DCA COUNT       /TYPING CHARACTERS
NEXT,   TAD I IR1       /GET A CHARACTER
        JMS TYPE        /TYPE IT
        ISZ COUNT       /DONE YET?
        JMP NEXT        /NO: TYPE ANOTHER
        JMP I HELLO     /YES: RETURN TO MAINLINE
TYPE,   0               /TYPE SUBROUTINE
        TSF             /PRINTER FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        TLS             /YES: PRINT A CHARACTER
        CLA             /CLEAR ACCUMULATOR
        JMP I TYPE      /AND RETURN
CHARAC, .               /INITIAL VALUE OF IR1
        310             /H
        305             /E
        314             /L
        314             /L
        317             /O
        241             /!
M6,     -6              /CHARACTER COUNT
COUNT,  0               /CHARACTER COUNTER
IR1=10                  /AUTOINDEX REGISTER
$
```

Figure 6-8 Subroutine to Print the Message, "HELLO!"

**Numeric Translation Routines**

The ASCII code for a number must be converted to octal representation before the computer may use the number in calculations. For example, 6 is represented by the ASCII code 266. When the Teletype key for 6 is typed, the code 266 is transmitted to the computer upon execution of the next KRB instruction. Two methods may be used to remove the 260 from an ASCII-coded number and obtain the octal number itself.

One method is to clear the first eight bits of the ASCII-coded

number by using the AND instruction and an appropriate mask. If $17_8$ is ANDed with the coded number, as shown below, the octal value of the number is recovered.

| Instruction | Operation | Comment |
|---|---|---|
| | 000 010 110 110 | ASCII Code 266 in accumulator. |
| AND MASK | 000 000 001 111 | MASK: 17. |
| | 000 000 000 110 | Contents of accumulator after AND instruction is executed. |

The second method of stripping an ASCII-coded number is to subtract $260_8$ from the character code. This is accomplished by TADing the two's complement of $260_8$ to the coded number, as shown below.

| Instruction | Operation | Comment |
|---|---|---|
| TAD M260 | 000 010 110 110 | ASCII Code 266 in accumulator. |
| | 111 101 010 000 | M260: $7520_8$ (2's comp of 260) |
| | 000 000 000 110 | Contents of accumulator after TAD instruction is executed. |

Figure 6-9 shows two programs which accept and store an octal digit, using the LISN subroutine presented in Figure 6-2. This process may be reversed to print out a digit which is stored in memory by adding $260_8$ to the digit, as illustrated in the program of Figure 6-10. This program calls the TYPE subroutine of Figure 6-4 to print out the binary number 7.

```
/USING THE AND
/INSTRUCTION
*200
NUMIN,   KCC
         JMS LISN
         AND MASK
         DCA HOLD
         HLT
HOLD,    0
MASK,    17
$
```

```
/USING THE TAD
/INSTRUCTION
*200
NUMIN,   KCC
         JMS LISN
         TAD M260
         DCA HOLD
         HLT
HOLD,    0
M260,    7520
$
```

Figure 6-9 Two Methods of Converting ASCII Code to Binary

```
*200
NUMOUT, CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /RAISE PRINTER FLAG
        TAD NUMBER       /GET NUMBER
        TAD K260         /CONVERT TO ASCII
        JMS TYPE         /PRINT THE NUMBER
        HLT              /AND HALT
NUMBER, 7                /NUMBER TO BE PRINTED
K260,   260
$
```

Figure 6-10 Program to Type One Stored Digit

All of the routines presented so far have been designed to handle single-digit octal numbers. However, PDP-8/E core memory locations may contain octal numbers with up to four digits. The program shown in Figure 6-1 prints out a four-digit octal number which is stored in core memory. The program shown in Figure 6-13 accepts four octal digits from the Teletype keyboard, converts them to an octal number, stores that number, and then loops back to accept another four digits. Figure 6-11 is a flowchart illustrating the procedure employed in these routines.

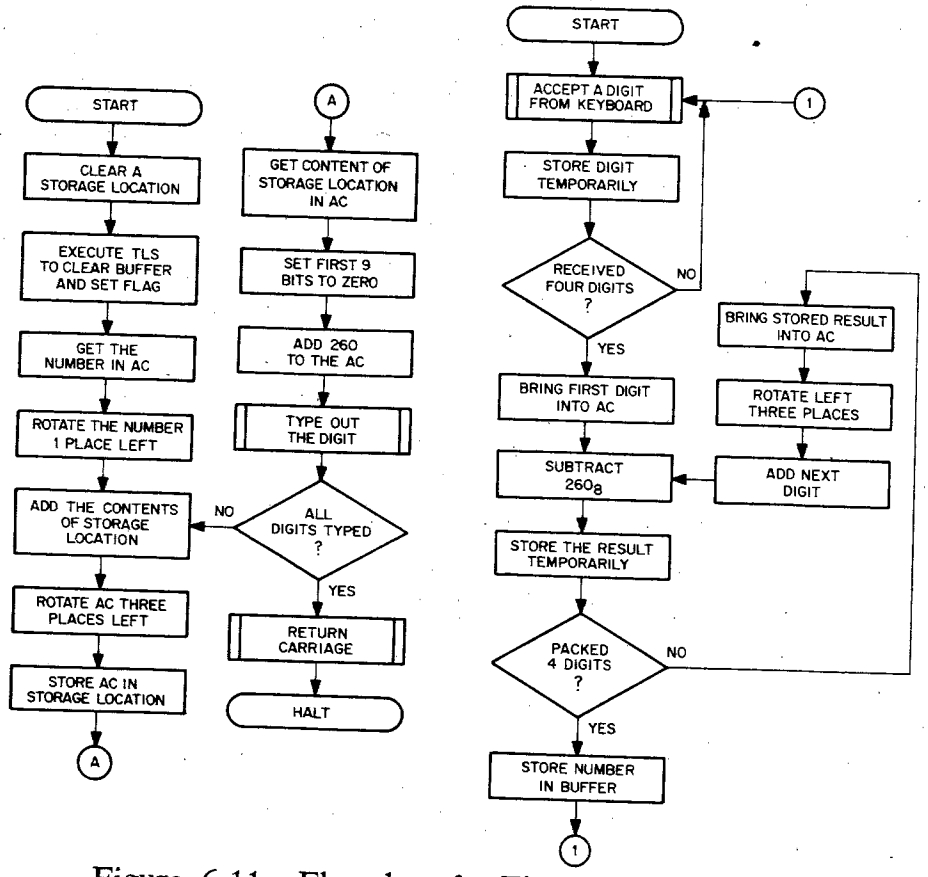



Figure 6-11 Flowchart for Figures 6-12 and 6-13

```
*200
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        DCA STORE        /CLEAR STORAGE LOCATION
        TLS              /RAISE PRINTER FLAG
        JMS CRLF         /RETURN CARRIAGE
        TAD M4           /SET LOCATION TO COUNT
        DCA DIGCTR       /NUMBER OF DIGITS
        TAD NUMBER       /GET NUMBER TO BE TYPED
        RAL              /ROTATE ONE PLACE LEFT
UNPACK, TAD STORE        /ADD STORED LOCATION
        RAL              /ROTATE THREE
        RTL              /PLACES LEFT
        DCA STORE        /STORE ROTATED NUMBER
        TAD STORE        /MASK OUT
        AND MASK7        /FIRST 9 BITS
        TAD K260         /ADD IN 260
        JMS TYPE         /AND TYPE A DIGIT
        ISZ DIGCTR       /TYPED FOUR DIGITS YET?
        JMP UNPACK       /NO: GO TYPE ANOTHER
        JMS CRLF         /YES: RETURN CARRIAGE
        HLT              /AND HALT
TYPE,   0                /TYPE SUBROUTINE
        TSF              /PRINTER FLAG RAISED YET?
        JMP .-1          /NO: CHECK AGAIN
        TLS              /YES: PRINT A CHARACTER
        CLA              /CLEAR ACCUMULATOR
        JMP I TYPE       /AND RETURN
CRLF,   0                /CRLF SUBROUTINE
        TAD K215         /GET ASCII CARRIAGE RETURN
        JMS TYPE         /PRINT IT
        TAD K212         /GET ASCII LINE FEED
        JMS TYPE         /PRINT IT
        JMP I CRLF       /AND RETURN
NUMBER, 1234             /NUMBER TO BE PRINTED
MASK7,  7                /AND MASK
M4,     -4               /DIGIT COUNT
DIGCTR, 0                /DIGIT COUNTER
STORE,  0                /STORAGE LOCATION
K212,   212              /ASCII LINE FEED
K215,   215              /ASCII CARRIAGE RETURN
K260,   260
$
```

Figure 6-12 Program to Type a Four-Digit Number

```
*200
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /RAISE PRINTER FLAG
        TAD K1777        /SET INDEX REGISTER TO
        DCA IR1          /STORE PACKED NUMBERS
        JMS CRLF         /RETURN CARRIAGE
NXTNUM, TAD M4           /SET COUNTER FOR
        DCA COUNTR       /4 DIGITS
        TAD K350         /SET UP TEMPORARY STORAGE
        DCA TEMP         /FOR ASCII INPUT
NXTDIG, JMS LISN         /GET A CHARACTER
        DCA I TEMP       /STORE IT TEMPORARILY
        ISZ TEMP         /INCREMENT STPOINTER
        ISZ COUNTR       /RECEIVED 4 DIGITS YET?
        JMP NXTDIG       /NO: GET ANOTHER
        JMS CRLF         /YES: RETURN CARRIAGE
        JMS PACK         /PACK THE 4 DIGITS
        DCA I IR1        /STORE PACKED NUMBER
        JMP NXTNUM       /GET A NEW NUMBER
PACK,   0                /PACK SUBROUTINE
        DCA STORE        /CLEAR STORAGE LOCATION
        TAD M4           /SET COUNTER FOR
        DCA COUNTR       /4 DIGITS
        TAD K350         /SET POINTER TO
        DCA TEMP         /ASCII INPUT CHARACTERS
PAKDIG, TAD STORE        /LOAD PARTIAL NUMBER
        CLL RAL          /ROTATE LEFT
        RTL              /THREE TIMES
        TAD I TEMP       /ADD NEXT STORED DIGIT
        TAD M260         /STRIP OFF THE 260
        DCA STORE        /STORE PARTIAL NUMBER
        ISZ TEMP         /INCREMENT POINTER
        ISZ COUNTR       /PACKED 4 DIGITS YET?
        JMP PAKDIG       /NO: PACK NEXT DIGIT
        TAD STORE        /YES: TAKE PACKED NUMBER
        JMP I PACK       /BACK TO MAINLINE
CRLF,   0                /CRLF SUBROUTINE
        TAD K215         /GET ASCII CARRIAGE RETURN
        JMS TYPE         /PRINT IT
        TAD K212         /GET ASCII LINE FEED
        JMS TYPE         /PRINT IT
        JMP I CRLF       /AND RETURN
```

Figure 6-13 Program to Pack and Store Four-Digit Numbers

```
LISN,   0               /LISN SUBROUTINE
        KSF             /KEYBOARD FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        KRB             /YES: READ A CHARACTER
        TLS             /ECHO ON PRINTER
        JMP I LISN      /AND RETURN
TYPE,   0               /TYPE SUBROUTINE
        TSF             /PRINTER FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        TLS             /YES: PRINT A CHARACTER
        CLA             /CLEAR ACCUMULATOR
        JMP I TYPE      /AND RETURN
K1777,  1777            /LAST LOC BEFORE BUFFER
M4,     7774            /DIGIT COUNT
COUNTR, 0               /DIGIT COUNTER
K350,   350             /BEGIN TEMP STORAGE
TEMP,   0               /TEMP STORAGE POINTER
STORE,  0               /PACK ROUTINE STORAGE
M260,   7520            /ASCII CONVERSION CONSTANT
K215,   215             /ASCII CARRIAGE RETURN
K212,   212             /ASCII LINE FEED
IR1=10                  /AUTO-INDEX REGISTER
$
```

Figure 6-13 (cont.) Program to Pack and Store Four-Digit Numbers

The text and numeric translation routines described earlier are combined in the following sample program, which accepts four-digit octal numbers delimited by carriage returns and then sorts the numbers into ascending numerical order and prints them on the teleprinter.

Any number of elements may be supplied. The end of input is signalled by typing a dollar sign ($). This program includes routines which ignore any non-octal digits of input and echo a question mark. Only positive octal digits are accepted. The program is presented in four illustrations.

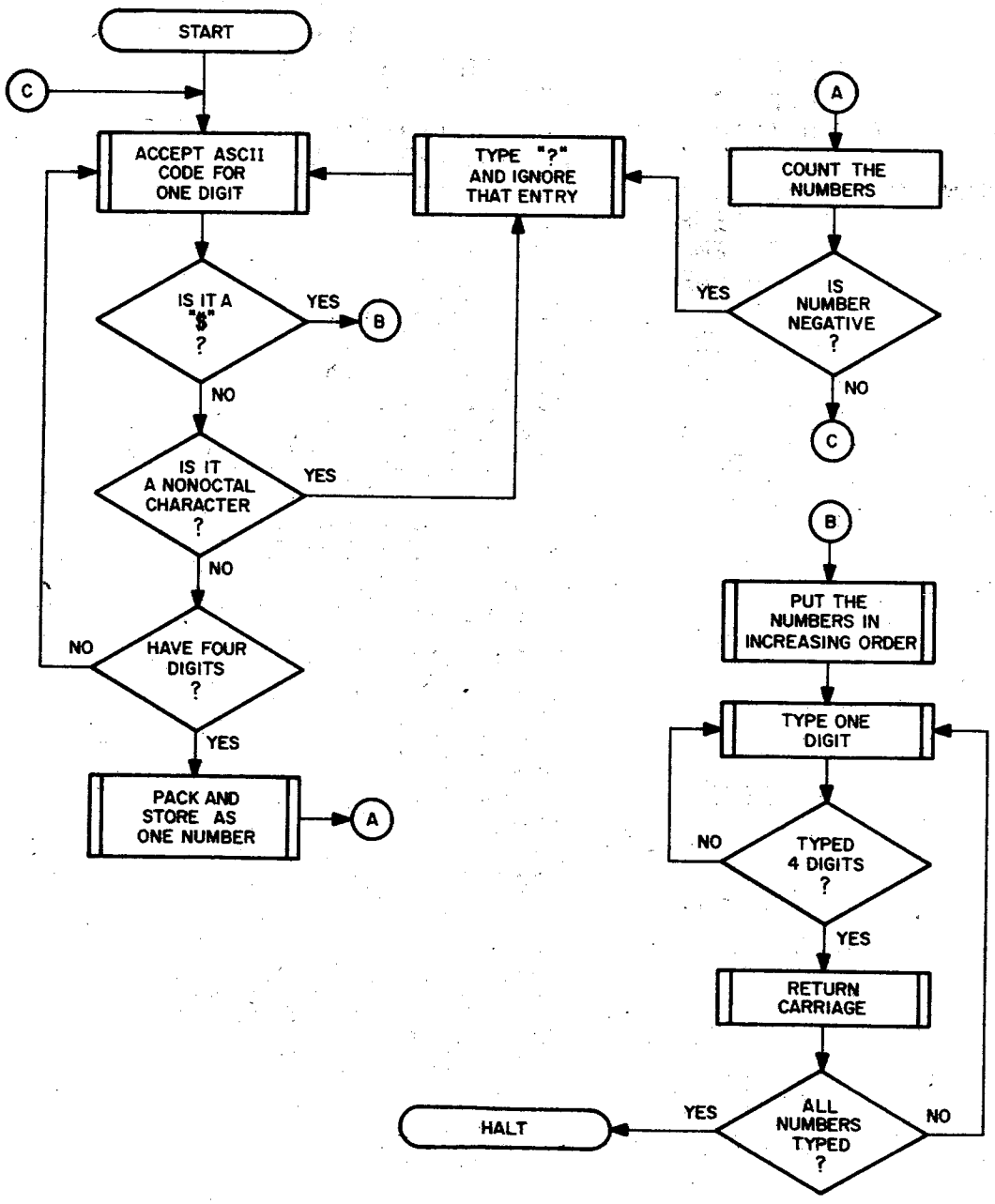



Figure 6-14 Flowchart for Sample Program

```
*200
START,  CLA CLL          /CLEAR ACCUMULATOR AND LINK
        TLS              /RAISE PRINTER FLAG
        TAD BUFF         /INITIALIZE STORAGE
        DCA BUFFPT       /BUFFER POINTER
        DCA AMOUNT       /AND ELEMENT COUNTER
ACCEPT, JMS CRLF         /RETURN CARRIAGE
        TAD M4           /INITIALIZE COUNTER
        DCA DIGCTR       /FOR 4 DIGIT INPUT
        TAD TEMP1        /SET A POINTER TO
        DCA TEMP         /TEMPORARY INPUT STORAGE
NEWDIG, JMS LISN         /GET A CHARACTER
        DCA I TEMP       /STORE IT
CHECK,  TAD I TEMP       /CHECK FOR A
        TAD MDOLAR       /TERMINAL $
        SNA CLA          /IS CHARACTER A $?
        JMP ORDER        /YES: ORDER INPUT
        TAD I TEMP       /NO: CHECK FOR
        TAD M260         /OCTAL INPUT
        SPA              /IS INPUT < 260?
        JMP ERROR        /YES: ENTER ERROR ROUTINE
        TAD M10          /NO: SUBTRACT 10
        SMA CLA          /IS INPUT > 260?
        JMP ERROR        /YES: ENTER ERROR ROUTINE
        ISZ TEMP         /NO: INCREMENT POINTER
        ISZ DIGCTR       /RECEIVED 4 DIGITS YET?
        JMP NEWDIG       /NO: GET ANOTHER
PACK,   TAD TEMP1        /YES: SET POINTER TO
        DCA TEMP         /STORAGE LOCATION
        DCA HOLD         /CLEAR LOCATION HOLD
        TAD M4           /SET COUNTER FOR
        DCA DIGCTR       /4 DIGITS
DIGPAK, TAD HOLD         /LOAD HOLD INTO AC
        CLL RAL          /ROTATE INTO CLEARED LINK
        RTL              /ROTATE TWICE MORE
        TAD I TEMP       /ADD ONE ASCII CHARACTER
        TAD M260         /SUBTRACT OUT THE 260
        DCA HOLD         /STORE IN LOCATION HOLD
        ISZ TEMP         /INCREMENT STORAGE POINTER
        ISZ DIGCTR       /PACKED 4 DIGITS YET?
        JMP DIGPAK       /NO: PACK ANOTHER
        TAD HOLD         /YES: STORE
        DCA I BUFFPT     /PACKED NUMBER
        TAD I BUFFPT     /CHECK FOR
        TAD K4000        /NEGATIVE INPUT
        SMA CLA          /IS ENTRY NEGATIVE?
        JMP ERROR        /YES: TAKE ERROR BRANCH
        ISZ AMOUNT       /NO: COUNT THE ENTRIES
        ISZ BUFFPT       /INITIALIZE FOR A NEW ENTRY
        JMP ACCEPT       /GET A NEW ENTRY
```

Figure 6-15 Initialization and Input Coding for Sample Program

```
ORDER,  TAD AMOUNT      /SET UP A TALLY
        CIA             /TO COUNT THE
        IAC             /NUMBER OF
        DCA TALLY       /COMPARISONS
        DCA FLAG        /CLEAR THE FLAG
        TAD BUFF        /SET X1 POINTER TO
        DCA X1          /FIRST BUFFER LOCATION
        TAD BUFF        /SET X2 POINTER TO
        IAC             /SECOND BUFFER
        DCA X2          /LOCATION
TEST,   TAD I X2        /GET X2 POINTER ENTRY
        CIA             /FORM TWO'S COMPLEMENT
        TAD I X1        /ADD X1 POINTER ENTRY
        SMA SZA CLA     /IS X1 ENTRY LARGER?
        JMS REVERSE     /YES: SWITCH X1 AND X2
        ISZ X1          /NO: INCREMENT X1 POINTER
        ISZ X2          /INCREMENT X2 POINTER
        ISZ TALLY       /COMPARED ALL ENTRIES?
        JMP TEST        /NO: LOOP BACK
        TAD FLAG        /YES: WAS SWITCH MADE
        SZA CLA         /ON LAST PASS?
        JMP ORDER       /YES: MAKE ANOTHER PASS
        JMP PRINT       /NO: TYPE THE ORDERED DATA
REVERSE,Ø               /SWITCH X1 AND X2
        TAD I X1        /GET X1 ENTRY
        DCA HOLD        /STORE TEMPORARILY
        TAD I X2        /GET X2 ENTRY
        DCA I X1        /STORE AT X1 LOCATION
        TAD HOLD        /GET X1 ENTRY
        DCA I X2        /STORE AT X2 LOCATION
        CLA CLL CMA     /SET FLAG WHENEVER
        DCA FLAG        /A SWITCH IS MADE
        JMP I REVERSE   /RETURN
```

Figure 6-16 Order Routine Coding for Sample Program

```
PRINT,  JMS CRLF        /RETURN THE CARRIAGE
        TAD BUFF        /INITIALIZE THE
        DCA BUFFPT      /BUFFER POINTER
        TAD AMOUNT      /INITIALIZE A COUNTER
        CIA             /TO COUNT
        DCA PRNTCT      /OUTPUT ELEMENTS
ANOTHR, JMS CRLF        /RETURN THE CARRIAGE
        TAD M4          /GET DIGIT COUNT
        DCA DIGCTR      /INITIALIZE DIGIT COUNTER
        DCA HOLD        /CLEAR STORAGE LOCATION
        TAD I BUFFPT    /GET A CHARACTER
        CLL RAL         /ROTATE INTO CLEARED LINK
MORE,   TAD HOLD        /ADD STORED LOCATION
        RAL             /ROTATE LEFT
        RTL             /THREE TIMES
        DCA HOLD        /STORE ROTATED NUMBER
        TAD HOLD        /GET STORED LOCATION
        AND MASK7       /MASK OUT FIRST 9 BITS
        TAD K260        /CONVERT TO ASCII
        JMS TYPE        /TYPE ONE DIGIT
        ISZ DIGCTR      /TYPED 4 DIGITS YET?
        JMP MORE        /NO: TYPE ANOTHER
        ISZ BUFFPT      /YES: INCREMENT POINTER
        ISZ PRNTCT      /TYPED ALL ENTRIES?
        JMP ANOTHR      /NO: TYPE ANOTHER
        JMS CRLF        /YES: RETURN CARRIAGE AND
        JMP START       /ACCEPT MORE INPUT
ERROR,  CLA             /ERROR ROUTINE
        TAD QUEST       /GET ASCII FOR "?"
        JMS TYPE        /PRINT QUESTION MARK
        JMS ACCEPT      /DISREGARD ILLEGAL ENTRY
END=.
```

Figure 6-17 Output Coding for Sample Program

```
*100
TYPE,   0               /TYPE SUBROUTINE
        TSF             /PRINTER FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        TLS             /YES: PRINT A CHARACTER
        CLA             /CLEAR ACCUMULATOR
        JMP I TYPE      /AND RETURN
CRLF,   0               /CARRIAGE RETURN/LINE FEED
        TAD K215        /GET ASCII CARRIAGE RETURN
        JMS TYPE        /PRINT IT
        TAD K212        /GET ASCII LINE FEED
        JMS TYPE        /PRINT IT
        JMP I CRLF      /AND RETURN
LISN,   0               /LISN SUBROUTINE
        KSF             /KEYBOARD FLAG RAISED YET?
        JMP .-1         /NO: CHECK AGAIN
        KRB             /YES: READ A CHARACTER
        TLS             /ECHO ON PRINTER
        JMP I LISN      /AND RETURN
BUFF,   END             /FIRST BUFFER LOCATION
BUFFPT, 0               /BUFFER POINTER
M4,     7774            /DIGIT COUNT
DIGCTR, 0               /DIGIT COUNTER
TEMP1,  .+2             /TEMP STORAGE POINTER
TEMP,   0;0;0;0;0       /TEMP DIGIT STORAGE
MDOLAR, 7534            /TWO'S COMP OF 244
M10,    -10             /OCTAL -10
K4000,  4000            /OCTAL 4000
HOLD,   0               /PACK ROUTINE STORAGE
M260,   -260            /ASCII CONVERSION FACTOR
AMOUNT, 0               /DATA ENTRY COUNTER
FLAG,   0               /SIGNALS DATA SWITCH
TALLY,  0               /COUNTS DATA COMPARASONS
X1,     0               /ORDER ROUTINE
X2,     0               /POINTERS
PRNTCT, 0               /COUNT OUTPUT ELEMENTS
MASK7,  7               /ASCII CONVERSION MASK
K260,   260             /ASCII DIGIT OFFSET
K212,   212             /ASCII LINE FEED
K215,   215             /ASCII CARRIAGE RETURN
QUEST,  277             /ASCII QUESTION MARK
$
```

Figure 6-18 Subroutines and Constants for Sample Program

## PROGRAM INTERRUPT FACILITY

The running time of programs using input and output routines is primarily made up of the time spent waiting for an I/O device to accept or transmit information. Specifically, this time is spent in loops such as:

```
TSF      /SKIP ON FLAG
JMP .-1
```

Waiting loops waste a large amount of computer time. In those cases where the computer can be doing something else while waiting, these loops may be removed and useful routines included to use the waiting time. This sharing of a computer between two tasks is often accomplished through the program interrupt facility, which is standard on all PDP-8 series computers. The program interrupt facility allows certain external conditions to interrupt the computer program. It is used to speed the processing of I/O devices or to allow certain alarms to halt program execution and initiate another routine.

Every device which is able to request a program interrupt contains a special one-bit register called the *interrupt request flag*. This register normally contains a 0, but it is set to 1 whenever the device requires servicing.

When the interrupt facility is enabled and any device flags an interrupt request, the computer automatically disables its interrupt system and executes a hardware JMS 0. This causes the contents of the program counter to be stored in core memory location 0000 and the instruction in location 0001 to be executed. Location 0001 usually contains an effective JMP SERVE, where SERVE is the entry address of an interrupt service routine. The interrupt service routine calls I/O device service routines to correct the condition which caused the interrupt, then re-enables the interrupt system and executes a JMP I 0 instruction to resume program execution.

Table 6-3 lists the eight IOT instructions used to program the PDP-8/E for program interrupt operation.

If an interrupt occurs while another interrupt is being serviced, the return address stored in location 0000 will be lost. This is prevented by leaving the interrupt system disabled while the interrupt

**Table 6-3 Program Interrupt IOT Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| SKON | 6000 | Skip the next instruction if the interrupt system is on and turn the interrupt system off. |
| ION | 6001 | Execute the next instruction, and then turn the interrupt system on. |
| IOF | 6002 | Turn the interrupt system off. |
| SRQ | 6003 | Skip the next instruction if one or more devices are requesting an interrupt. |
| GTF | 6004 | Get flags. The link is loaded into accumulator bit position 0. Accumulator bit 2 is set to a 1 if any device is requesting an interrupt. Accumulator bit 4 is set to a 1 if the interrupt system is enabled.[3] |
| RTF | 6005 | Restore flags. This instruction is the converse of the GTF instruction. Execution of an RTF instruction is deferred until after the next JMP or JMS instruction is executed. If accumulator bit 4 contains a 1, this instruction enables the interrupt system.[3] |
| SGT | 6006 | Skip the next instruction if the Greater Than Flag is set. This instruction is implemented only if the KE8-E Extended Arthmetic Element is installed. |
| CAF | 6007 | Clear all flags. This instruction is the logical equivalent of operating the CLEAR switch on the programmer's console. It should not be used while any I/O device is active. |

[3] The GTF and RTF instructions perform other operations on the remaining accumulator bit positions if the KE8-E Extended Arithmetic Element and/or KM8-E Extended Memory Control are installed. See the *Small Computer Handbook* for further information.

service routine is servicing an I/O device. The interrupt system may then be re-enabled by an ION instruction immediately before the JMP I 0 which terminates the interrupt service routine, or by an RTF instruction anywhere in the service routine. Execution of the ION instruction will be deferred until the following instruction has been executed, and execution of the RTF instruction will be deferred until the next JMP or JMS instruction has been executed. In this manner, an interrupt service routine protects the contents of location 0000 by executing with the interrupt system disabled and using deferred ION or RTF instructions to re-enable the interrupt system as soon as mainline execution has resumed.

Use of the interrupt system allows a mainline routine, referred to as the *background program*, to execute without wasting a large amount of time in waiting loops while I/O devices are assembling and transmitting information. The interrupt service routine, called a *foreground program,* is entered automatically whenever an I/O device requires servicing under program control.

**Programming an Interrupt**

The program presented in Figure 6-19 consists of a background routine which rotates one bit through the accumulator endlessly, and a foreground program, initiated by the interrupt service routine, which accepts and stores ASCII characters from the Teletype. Upon receipt of the ASCII code for a period, the foreground program prints out the characters which have been stored.

The coding begins with an initialization routine which allocates buffer space to store the incoming characters and sets the mode for input. The program signals input mode by a value of MODE =0 and output mode by a value of MODE=1. Once the initialization routine has enabled the interrupt facility, the background program is started.

The background program is a routine to rotate one bit through the accumulator and link. The first instruction clears all bits except bit 11. The program then counts through the two ISZ loops, after which it rotates the bit one place left and then returns to the count loops. With the console indicator select switch set to AC, the accumulator and link displays will exhibit a rapidly rotating light while waiting for an interrupt to initiate the foreground program.

An interrupt request will cause the computer to perform the following operations automatically:

1. The interrupt system is disabled.
2. The content of the program counter is stored at core memory location 0000.
3. The JMP I 2 instruction in location 0001 is executed. Location 0002 contains the entry address of the interrupt service routine.

The *interrupt service routine* then performs the following operations:

1. The contents of the accumulator and link are stored.
2. The source of the interrupt is determined.
3. A JMP to either the keyboard input routine or the printer output routine is executed.

The *keyboard input routine* is entered from the interrupt service routine whenever the keyboard flag is set. The flag is cleared to prevent further interruptions when the interrupt system is re-enabled. If the mode is set for output, program control returns to the background program. Otherwise, the routine accepts a character (KRB), acknowledges receipt by printing the character on the printer (TLS), and stores it in the buffer. No KSF, JMP .−1 loop is necessary. The routine then checks for the ASCII code for a period, returning to the background program if the character was not a period. Upon receipt of a period, the routine resets the buffer and sets the mode for output. Program control then returns to the background program. Since a TLS instruction was executed previously, another interrupt will be requested as soon as the printer becomes available, and the stored ASCII codes will be typed out by the printer output routine.

The *printer output routine* is entered from the interrupt service routine whenever the printer flag is set. This routine clears the device flag and checks for output mode, then prints one character from the buffer. No TSF, JMP .−1 loop is necessary. If the character is not a period, control returns to the background program while the printer finishes typing the character. If the character is a period, the routine resets the buffer, sets the mode for input, and then returns to the background program.

The keyboard and printer service routines return control to the background program via the *exit routine*. This routine restores the content of the accumulator, which was previously saved by the interrupt service routine, then re-enables the interrupt system by means of an RTF instruction, which also restores the link. The RTF instruction does not take effect until after the JMP I 0 instruction, which terminates the exit routine, has been executed. This allows background program execution to resume before another interrupt can occur. The constants used by the various routines conclude the listing.

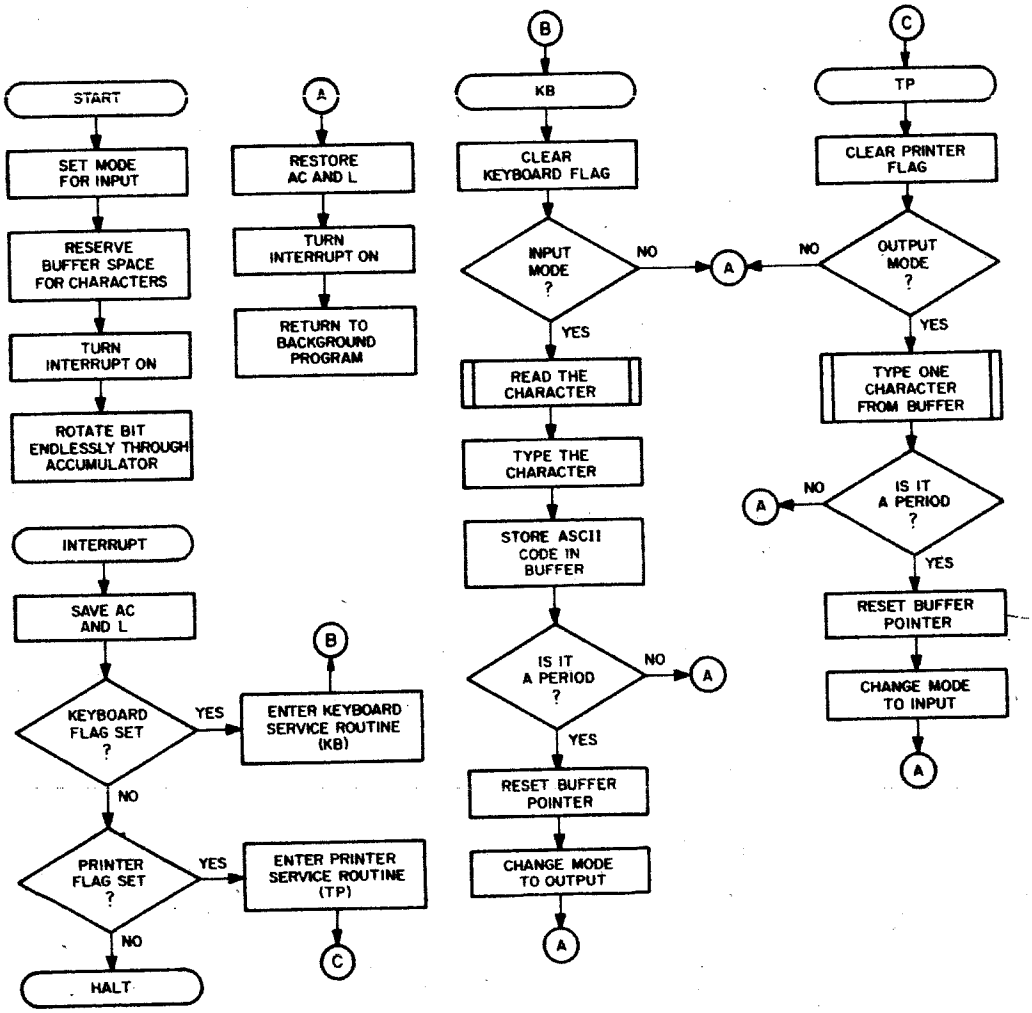



Figure 6-19A Interrupt Facility Program Flowchart

```
*0
                        /FIRST INSTRUCTION
                        /AFTER AN INTERRUPT:
        0               /STORE RETURN ADDRESS
        JMP I 2         /JUMP TO SERVICE ROUTINE
        SERVE           /POINTER TO SERVICE ROUTINE
*200                    /INITIALIZATION ROUTINE:
START,  CLA CLL         /CLEAR ACCUMULATOR AND LINK
        DCA MODE        /SET MODE FOR INPUT
        TAD K1777       /INITIALIZE DATA
        DCA BUFFER      /BUFFER POINTER
        ION             /TURN INTERRUPT ON
                        /BACKGROUND PROGRAM:
ROTATE, CLA CLL IAC     /SET ACCUMULATOR BIT 11
        ISZ COUNT       /COUNT
        JMP .-1         /TWICE
        ISZ COUNT       /THROUGH
        JMP .-1         /DELAY LOOP
        RAL             /ROTATE BIT LEFT
        JMP ROTATE+1    /RETURN TO DELAY LOOP
                        /SERVICE ROUTINE:
SERVE,  DCA AC          /SAVE ACCUMULATOR
        GTF             /SAVE INTERRUPT
        DCA FLAGS       /FLAGS AND LINK
        KSF             /KEYBOARD FLAG RAISED?
        SKP             /NO: CHECK PRINTER
        JMP KB          /YES: SERVICE KEYBOARD
        TSF             /PRINTER INTERRUPT?
        SKP             /NO: SKIP FOR EXIT
        JMP TP          /YES: SERVICE PRINTER
        CAF             /CLEAR ALL FLAGS
        JMP EXIT        /AND RETURN
                        /KEYBOARD INPUT ROUTINE:
KB,     KCC             /CLEAR KEYBOARD FLAG
        TAD MODE        /CURRENTLY IN
        SZA CLA         /INPUT MODE?
        JMP EXIT        /NO: RETURN TO BACKGROUND
        ISZ BUFFER      /YES: INCREMENT POINTER
        KRB             /READ THE CHARACTER
        TLS             /ECHO ON PRINTER
        DCA I BUFFER    /STORE THE CHARACTER
        TAD I BUFFER    /WAS THE
        TAD MPER        /CHARACTER A
        SZA CLA         /PERIOD?
        JMP EXIT        /NO: RETURN TO BACKGROUND
        TAD K1777       /YES: RESET BUFFER
        DCA BUFFER      /POINTER TO TYPE
        CLA CMA         /THE CHARACTERS
        DCA MODE        /SET MODE FOR OUTPUT AND
        JMP EXIT        /RETURN TO BACKGROUND
```

Figure 6-19B Interrupt Facility Program Coding

```
                        /PRINTER OUTPUT ROUTINE:
TP,     TCF             /CLEAR PRINTER FLAG
        TAD MODE        /CURRENTLY IN
        SNA CLA         /OUTPUT MODE?
        JMP EXIT        /NO: RETURN TO BACKGROUND
        ISZ BUFFER      /YES: INCREMENT POINTER
        TAD I BUFFER    /GET CHARACTER FROM BUFFER
        TLS             /PRINT IT
        TAD MPER        /WAS THE CHARACTER
        SZA CLA         /A PERIOD?
        JMP EXIT        /NO: RETURN TO BACKGROUND
        DCA MODE        /YES: SET MODE FOR INPUT
        TAD K1777       /RESET BUFFER
        DCA BUFFER      /POINTER AND
        JMP EXIT        /RETURN TO BACKGROUND
                        /EXIT ROUTINE:
EXIT,   TAD FLAGS       /GET FLAGS AND LINK
        RTF             /RESTORE FLAGS AND LINK
        CLA             /CLEAR ACCUMULATOR
        TAD AC          /RESTORE ACCUMULATOR
        JMP I Ø         /RETURN TO BACKGROUND
COUNT,  Ø               /DELAY LOOP COUNTER
MODE,   Ø               /MODE SWITCH
K1777,  1777            /LAST LOC BEFORE BUFFER
BUFFER, Ø               /DATA BUFFER POINTER
AC,     Ø               /SAVE ACCUMULATOR
FLAGS,  Ø               /SAVE FLAGS AND LINK
MPER,   -256.           /NEGATIVE OF ASCII CODE
$
```

Figure 6-19B (cont.) Interrupt Facility Program Coding

**Multiple Device Interrupt Programming**

Many programming applications use the interrupt system to service several devices. For example, a PDP-8/E may use the interrupt facility to control the operation of DECtape and DECdisk systems through a Teletype console. Systems of this type require a service routine that determines the source of an interrupt request (i.e., which device flag is set). The following instruction sequence uses dummy skip instructions to determine which device requested an interrupt.

```
DASF        /SKIP ON DEVICE A FLAG
SKP
JMP SERVA   /DEVICE A REQUESTED THE INTERRUPT
DBSF        /SKIP ON DEVICE B FLAG
SKP
JMP SERVB   /DEVICE B REQUESTED THE INTERRUPT
      .
      .
      .
DNSF        /SKIP ON DEVICE N FLAG
SKP
JMP SERVN   /DEVICE N REQUESTED THE INTERRUPT
```

The dummy instructions (DASF, DBSF, etc.) are skip-on-flag instructions for each of the devices in the interrupt system. Because of the predominance of SKP instructions, the instruction sequence which determines the source of an interrupt request is often called a *skip chain.*

A skip chain may be enlarged to test for almost any number of device flags, provided that high-speed devices which retain information for a relatively short period of time are tested near the top of the skip chain, so that the chain may be traversed and the high-speed devices serviced before the information is lost. High-speed devices should never be required to wait for service while a long skip chain is traversed. If several high-speed devices are tested at the top of the chain, it may still be necessary to check the timing constraints to ensure that the last high-speed device to be tested will be serviced before any loss of information can occur.

If two interrupts occur simultaneously, the high-speed device will be serviced first because it will be tested first in the skip chain. The low-speed device may be serviced later in either of two ways:

1. By terminating the high-speed device service routine in the usual manner, in which case the low-speed device will request another interrupt as soon as the interrupt system has been re-enabled and background program execution resumed.
2. By terminating the high-speed device service routine with a jump back into the skip chain, without re-enabling the interrupt system. In this case, the skip chain must terminate

with an ION, JMP I 0 instruction sequence to return control to the background program if no further interrupt requests are pending.

**A Software Priority Interrupt System**

The techniques described in the previous section may be used to guarantee that a high-speed device will be serviced before a slower device when two interrupts occur simultaneously. However, information will still be lost if the high-speed device requires servicing and requests an interrupt while the low-speed device is being serviced, because the interrupt system is disabled during this time.

An interrupt service routine may be written in such a way that a priority of device interrupts is established through software. This may become necessary in a system that includes high-speed devices which retain information for a short time and require immediate attention. For example, a PDP-8/E system employing Teletype and DECtape I/O may use a software priority interrupt system which permits the Teletype service routines to be interrupted whenever DECtape requests servicing. This is accomplished by modifying the keyboard and printer service routines to permit the following sequence of operations:

1. Begin the keyboard and printer service routines by storing the contents of location 0000, the accumulator, the link and any other important registers in temporary storage locations.
2. Execute an ION instruction.
3. Clear the low-speed device flag. The ION instruction will not take effect until this instruction has been executed.
4. Service the device. At this point, the service routine may be interrupted without danger of the background program return address being lost.
5. Terminate the service routine by restoring the contents of the accumulator, the link and any other registers which were stored in step 1, then executing a JMP I TEMP, where TEMP is the location in which the return address from location 0000 was stored.

A low-speed device service routine written in this manner operates with the interrupt system enabled most of the time. The

routine may be interrupted by an interrupt request from a higher priority device, and the second interrupt will not cause the contents of location 0000 (or any other important registers) to be lost.

In the example cited above, the DECtape and DECdisk service routines would not re-enable the interrupt system until they are ready to return control to the background program. However, the Teletype routines re-enable the interrupt system almost immediately. If DECtape or DECdisk interrupts one of the Teletype service routines, the Teletype service routine is treated like a background program; that is, execution of the Teletype service routine is suspended until the high-speed device has been serviced. Control then reverts back to the Teletype service routine, and finally back to the mainline program.

**Multiple Interrupt Demonstration Program**

Figure 6-20 presents a demonstration program designed to use the program interrupt facility. This program rotates a bit through the accumulator, with the speed and direction of rotation determined by switch register settings. Simultaneously, the program accepts 4-digit, positive octal numbers from the Teletype, automatically terminating each 4-digit number with a carriage return and line feed. Upon receipt of a typed dollar sign ($), the program sorts the data into ascending numerical order and prints the ordered numbers on the Teletype.

This example illustrates the power of interrupt programming because the computer appears to be performing two tasks at the same time. The programmer knows that this is impossible, and that the two tasks are actually sharing computer time; however, the appearance indicates simultaneous action.

When an interrupt request occurs, the current value of the PC is stored in location 0000 of page 0, while locations 0001 and 0002 contain an indirect jump to the interrupt service routine. The program constants which are stored on page 0 beginning at location 0050 include four software switches that record conditions within the program. The contents of location MODE indicates whether the program is currently performing input or output functions. Location SW1 indicates whether the next digit to be received

```
*0
        0                   /PAGE ZERO
        JMP I 2             /INTERRUPT
        SERVE               /HANDLER
*50                         /PAGE ZERO CONSTANTS:
MODE,   0                   /INPUT=0    OUTPUT=-1
SW1,    0                   /NEW # = 0    OLD # = 1
SW2,    0                   /CR=0    LF=-1    DATA=1
SW3,    0                   /MODE BYPASS SWITCH
AC,     0                   /SAVE ACCUMULATOR
FLAGS,  0                   /SAVE INTERRUPT FLAGS
PRINTR, TP                  /POINTER TO TP ROUTINE
KEYBRD, KB                  /POINTER TO KB ROUTINE
ORDPTR, ORDER               /POINTER TO ORDER ROUTINE
EXITPT, EXIT                /POINTER TO EXIT ROUTINE
BUFF,   END                 /FIRST BUFFER LOCATION
BUFFPT, 0                   /BUFFER POINTER
M4,     7774                /-4 OCTAL
DIGCTR, 0                   /DIGIT COUNTER
TEMP1,  .+2                 /TEMP STORAGE POINTER
TEMP,   0                   /TEMP DIGIT STORAGE
        0
        0
        0
        0
MDOLAR, 7534                /-244 OCTAL
M10,    7770                /-10 OCTAL
HOLD,   0                   /PACK ROUTINE STORAGE
HOLDL,  0                   /SAVE LINK
M260,   7520                /-260 OCTAL
AMOUNT, 0                   /DATA COUNTER
FLAG,   0                   /ORDER ROUTINE FLAG
TALLY,  0                   /ORDER ROUTINE COUNTER
X1,     0                   /ORDER ROUTINE POINTER
X2,     0                   /ORDER ROUTINE POINTER
PRNTCT, 0                   /OUTPUT ROUTINE COUNTER
K7,     7                   /ASCII CONSTANTS:
K260,   260                 /ASCII CONVERSION FACTOR
K212,   212                 /ASCII LINE FEED
K215,   215                 /ASCII CARRIAGE RETURN
QUEST,  277                 /ASCII QUESTION MARK
K4000,  4000                /4000 OCTAL
```

Figure 6-20A Multiple Interrupt Demonstration Program

```
CR,       TAD K215          /GET ASCII CARRIAGE RETURN
          TLS               /PRINT IT
          CLA CMA           /LOAD -1 INTO ACCUMULATOR
          DCA SW2           /SET SW2 FOR LINE FEED
          JMP I EXITPT      /ENTER EXIT ROUTINE
LF,       TAD K212          /GET ASCII LINE FEED
          TLS               /PRINT IT
          CLA               /CLEAR ACCUMULATOR
          TAD SW3           /GET MODE BYPASS SWITCH
          SNA CLA           /SET FOR BYPASS MODE?
          JMP SW2SET        /NO: SET SW2 AND RETURN
          DCA SW3           /YES: TURN OFF MODE BYPASS
          DCA SW2           /SET SW2 FOR CARRIAGE RETURN
          JMP I EXITPT      /ENTER EXIT ROUTINE
SW2SET,   CLA CLL IAC       /LOAD 1 INTO ACCUMULATOR
          DCA SW2           /SET SW2 FOR DATA
          JMP I EXITPT      /ENTER EXIT ROUTINE
```

Figure 6-20B Multiple Interrupt Demonstration Program

is the first digit of a new number, to be packed into a new storage location, or the next digit of a continuing number, to be packed and stored with previous digits. Location SW2 indicates whether a carriage return or line feed should be printed to delimit input elements, and location SW3 regulates the printing of carriage returns, line feeds and question marks while the program is in input mode. A question mark is printed whenever a non-octal digit or non-positive octal number is received.

Other constants include location AC, in which the content of the accumulator is saved during an interrupt, and location FLAGS, in which the interrupt flags and the link are stored. Location BUFF contains the address of the first data buffer location, which is the core location following the last program instruction. Thus, the data buffer is all of memory following the end of the program.

The 5-word block beginning at location TEMP is used to store

```
*200                        /INTERRUPT SYSTEM OFF
START,  IOF                 /DURING INITIALIZATION
        CLA CLL             /CLEAR ACCUMULATOR AND LINK
        DCA MODE            /SET INPUT MODE
        DCA SW1             /SET SW1 FOR NEW ENTRY
        DCA SW2             /SET SW2 FOR CARRIAGE RETURN
        DCA SW3             /SET MODE BYPASS
        TAD BUFF            /GET FIRST BUFFER LOCATION
        DCA BUFFPT          /INITIALIZE BUFFER POINTER
        DCA AMOUNT          /AND ELEMENT COUNTER
        ION                 /ENABLE INTERRUPTS
ROTATE, CLA CLL CML         /SET THE LINK
BEGIN,  DCA SAVEAC          /CLEAR ACCUMULATOR STORAGE
        RAL                 /ROTATE LINK BIT LEFT
        DCA SAVEL           /SAVE LINK BIT
        TAD K7000           /GET OR MASK
        OSR                 /READ SWITCH REGISTER
        DCA COUNT           /INITIALIZE DELAY COUNTER
        OSR                 /READ SR BIT ZERO
        RAL                 /ROTATE INTO LINK
        SZL CLA             /WAS BIT ZERO SET?
        JMS LEFT            /YES: ROTATE LEFT
        JMS RIGHT           /NO: ROTATE RIGHT
        CLL                 /CLEAR THE LINK
GO,     TAD SAVEL           /GET THE STORED LINK
        RAR                 /AND RESTORE IT
        TAD SAVEAC          /GET STORED ACCUMULATOR
INSTR,  HLT                 /OVERWRITTEN BY RAL OR RAR
        ISZ COUNTR          /INCREMENT COUNTER
        JMP INSTR+1         /UNTIL IT OVERFLOWS
        ISZ COUNT           /INCREMENT COUNT
        JMP INSTR+1         /UNTIL IT OVERFLOWS
        JMP BEGIN           /LOOP BACK AND REPEAT
SAVEAC, 0                   /STORAGE FOR ACCUMULATOR
SAVEL,  0                   /STORAGE FOR LINK
K7000,  7000                /SETS BITS 1-3
COUNTR, 0                   /ROTATE ROUTINE
COUNT,  0                   /COUNTERS
LEFT,   0                   /INSERT RAL INSTRUCTION
        ISZ LEFT            /INCREMENT RETURN ADDRESS
        TAD KRAL            /GET RAL INSTRUCTION
        DCA INSTR           /WRITE OVER HLT
        JMP I LEFT          /AND RETURN
RIGHT,  0                   /INSERT RAR INSTRUCTION
        TAD KRAR            /GET RAR INSTRUCTION
        DCA INSTR           /WRITE OVER HLT
        JMP I RIGHT         /AND RETURN
KRAR,   RAR                 /RAR INSTRUCTION
KRAL,   RAL                 /RAL INSTRUCTION
```

Figure 6-20C Multiple Interrupt Demonstration Program

```
SERVE,  DCA AC          /SAVE ACCUMULATOR
        GTF             /GET FLAGS
        DCA FLAGS       /SAVE FLAGS AND LINK
        TSF             /PRINTER FLAG RAISED?
        SKP             /NO: CHECK KEYBOARD
        JMP I PRINTR    /YES: SERVICE PRINTER
        KSF             /KEYBOARD FLAG RAISED?
        SKP             /NO: CLEAR ALL FLAGS
        JMP I KEYBRD    /YES: SERVICE KEYBOARD
        CAF             /AND RESUME
        JMP I EXITPT    /BACKGROUND EXECUTION
ORDER,  CLA CLL         /ORDER ROUTINE
        TAD AMOUNT      /GET NUMBER OF ENTRIES
        CIA             /SET TALLY TO
        IAC             /COUNT ITERATIONS
        DCA TALLY       /THRU ORDER LOOP
        DCA FLAG        /CLEAR FLAG ON EACH PASS
        TAD BUFF        /GET FIRST BUFFER ADDRESS
        DCA X1          /INITIALIZE X1 POINTER
        TAD BUFF        /FORM SECOND
        IAC             /BUFFER ADDRESS
        DCA X2          /INITIALIZE X2 POINTER
TEST,-  TAD I X2        /GET X2 POINTER ENTRY
        CIA             /FORM TWO'S COMPLEMENT
        TAD I X1        /ADD X1 POINTER ENTRY
        SPA SNA CLA     /IS ACCUMULATOR POSITIVE?
        JMP INCPTR      /NO: INCREMENT POINTERS
REVERS, TAD I X1        /YES: SWITCH THE ENTRIES
        DCA HOLD        /HOLD X1 ENTRY
        TAD I X2        /GET X2 ENTRY
        DCA I X1        /STORE AT X1 LOCATION
        TAD HOLD        /GET X1 ENTRY
        DCA I X2        /STORE AT X2 LOCATION
        CLA CLL CMA     /LOAD -1 INTO FLAG TO
        DCA FLAG        /SHOW SWITCH WAS MADE
INCPTR, ISZ X1          /INCREMENT X1 POINTER
        ISZ X2          /INCREMENT X2 POINTER
        ISZ TALLY       /COMPARED ALL ENTRIES?
        JMP TEST        /NO: LOOP BACK
        TAD FLAG        /YES: WAS SWITCH MADE
        SZA CLA         /ON LAST PASS?
        JMP ORDER       /YES: CONTINUE ORDERING
        CLA CMA         /NO: FORM -1 AND
        DCA MODE        /SET MODE FOR OUTPUT
        TAD BUFF        /GET FIRST BUFFER ADDRESS
        DCA BUFFPT      /INITIALIZE BUFFER POINTER
        DCA SW1         /SET SW1 FOR NEW NUMBER
        TAD AMOUNT      /GET NUMBER OF ENTRIES
        CIA             /FORM TWO'S COMPLEMENT
        DCA PRNTCT      /INITIALIZE COUNTER
        TLS             /RAISE PRINTER FLAG
        JMP I EXITPT    /RESUME BACKGROUND
```

Figure 6-20D Multiple Interrupt Demonstration Program

successive digits of a 4-digit number. Location TEMP serves as a pointer to the following four locations, and the content of location TEMP1 is used to reset location TEMP for the first digit of a new number. Page 0 also contains three subroutines which print carriage returns and line feeds, when indicated by the value of SW2, and then reset SW2 to continue data I/O.

The program coding begins with an initialiation routine at location 0200. This routine initializes the software switches and resets the data buffer pointer and counter, then enables the interrupt facility. At this point the background program is entered.

The ROTATE routine sets accumulator bit 11, then checks

```
*400                        /KEYBOARD SERVICE ROUTINE
KB,     KCC                 /CLEAR KEYBOARD FLAG
        TAD MODE            /CURRENTLY IN
        SZA CLA             /INPUT MODE?
        JMP EXIT            /NO: RESUME BACKGROUND
        TAD SW1             /YES: EXPECTING CONTINUED
        SZA CLA             /NUMBER OR NEW NUMBER?
        JMP CNTDIG          /CONTINUED NUMBER
        TAD M4              /NEW ENTRY
        DCA DIGCTR          /INITIALIZE DIGIT COUNTER
        TAD TEMP1           /AND TEMPORARY
        DCA TEMP            /DIGIT STORAGE
CNTDIG, KRS                 /READ A CHARACTER
        DCA I TEMP          /STORE IT
CHECK,  TAD I TEMP          /GET THE CHARACTER
        TLS                 /ECHO ON PRINTER
        TAD MDOLAR          /IS CHARACTER A
        SNA CLA             /DOLLAR SIGN ($)?
        JMP I ORDPTR        /YES: BEGIN ORDER ROUTINE
        TAD I TEMP          /NO: GET THE CHARACTER
        TAD M260            /SUBTRACT 260
        SPA                 /IS CHARACTER < 260?
        JMP ERROR           /YES: TAKE ERROR BRANCH
        TAD M10             /NO: ADD 10
        SPA                 /IS CHARACTER > 267?
        JMP LEGAL           /NO: MUST BE LEGAL
ERROR,  CLA IAC             /YES: FORM 1 AND
        DCA SW3             /SET SW3 FOR BYPASS
        DCA SW1             /RESET SW1
        TLS                 /RAISE PRINTER FLAG
        JMP EXIT            /RESUME BACKGROUND
```

Figure 6-20E Multiple Interrupt Demonstration Program

switch register bit 0 and rotates the accumulator right, if bit 0 is off, or left, is bit 0 is on. The appropriate instruction (RAL or RAR) is written into location INSTR, which is arbitrarily assigned an initial content of 7402 (HLT). After determining the direction of rotation, the ROTATE routine loads the value of switch register bits 1-11 into a counter and uses this value to control the speed of rotation. Increasing the value specified by switch register bits 1-11 decreases the speed of rotation.

```
LEGAL,  CLA CMA          /FORM -1 IN ACCUMULATOR
        DCA SW1          /SET SW1
        ISZ TEMP         /INCREMENT STORAGE POINTER
        ISZ DIGCTR       /INCREMENT DIGIT COUNTER
        JMP EXIT         /RESUME BACKGROUND
PACK,   TAD TEMP1        /INITIALIZE TEMP
        DCA TEMP         /STORAGE POINTER
        DCA HOLD         /CLEAR STORAGE LOCATION
        TAD M4           /INITIALIZE THE
        DCA DIGCTR       /DIGIT COUNTER
DIGPAK, TAD HOLD         /ADD STORED LOCATION
        RAL CLL          /ROTATE LEFT
        RTL              /THREE TIMES
        TAD I TEMP       /ADD IN A DIGIT
        TAD M260         /SUBTRACT THE 260
        DCA HOLD         /STORE PARTIAL NUMBER
        ISZ TEMP         /INCREMENT DIGIT POINTER
        ISZ DIGCTR       /PACKED 4 DIGITS YET?
        JMP DIGPAK       /NO: PACK ANOTHER
        TAD HOLD         /YES: GET PACKED NUMBER
        DCA I BUFFPT     /STORE IN BUFFER
        TAD I BUFFPT     /GET STORED NUMBER
        TAD K4000        /ADD 4000 OCTAL
        SPA CLA          /IS NUMBER NEGATIVE?
        JMP NOTNEG       /NO: KEEP ZERO ACCUMULATOR
        IAC              /YES: FORM 1 IN ACCUMULATOR
        JMP DISALO       /TAKE ERROR BRANCH
NOTNEG, ISZ BUFFPT       /INCREMENT POINTER
        ISZ AMOUNT       /AND COUNTER
        CLA CMA          /FORM -1 IN ACCUMULATOR
DISALO, DCA SW3          /SET MODE BYPASS
        DCA SW1          /RESET SW1
        TLS              /RAISE PRINTER FLAG
        JMP EXIT         /RESUME BACKGROUND
```

Figure 6-20F Multiple Interrupt Demonstration Program

The interrupt service routine, which begins in location SERVE, is simply a skip chain that directs control to the appropriate device service routine. If an interrupt occurs when none of the devices tested in the skip chain requested an interrupt, the routine clears all flags and resumes background execution.

The ORDER routine is initiated whenever receipt of a dollar sign indicates that input is complete. This routine uses techniques introduced in Chapter 3 to sort the input data into ascending order. The device service routines and the interrupt system exit routine conclude the program listing.

```
IP,     TCF             /CLEAR PRINTER FLAG
        TAD SW3         /SW3 SET FOR
        SNA CLA         /MODE BYPASS?
        JMP MODCHK      /NO: CHECK CURRENT MODE
        TAD SW3         /IS SW3 SET FOR
        SPA CLA         /A QUESTION MARK?
        JMP RETLF       /NO: RETURN CARRIAGE
        TAD QUEST       /YES: GET QUESTION MARK
        TLS             /PRINT IT
        CLA CMA         /SET SW3 FOR
        DCA SW3         /CR AND LF, THEN
        JMP EXIT        /RESUME BACKGROUND
MODCHK, TAD MODE        /CURRENTLY IN
        SNA CLA         /INPUT OR OUTPUT MODE?
        JMP EXIT        /INPUT: IGNORE REQUEST
RETLF,  TAD SW2         /OUTPUT: BEGIN OUTPUT
        SNA CLA         /PRINT A
        JMP CR          /CARRIAGE RETURN ON
        TAD SW2         /FIRST PASS, AND
        SPA CLA         /LINE FEED
        JMP LF          /ON SECOND PASS
DATA,   TAD SW1         /PRINT DATA ON THIRD PASS
        SZA CLA         /NEW NUMBER?
        JMP DIGTYP      /NO: TYPE ANOTHER DIGIT
        TAD M4          /YES: GET DIGIT COUNT
        DCA DIGCTR      /RESET DIGIT COUNTER
        DCA HOLD        /CLEAR TEMPORARY
        DCA HOLDL       /STORAGE LOCATIONS
        TAD I BUFFPT    /GET NEXT NUMBER TO PRINT
```

Figure 6-20G Multiple Interrupt Demonstration Program

```
DIGTYP, TAD HOLDL       /GET LINK AND
        CLL RAL         /RESTORE IT
        TAD HOLD        /ROTATE
        RAL             /STORED VALUE
        RTL             /LEFT
        DCA HOLD        /THREE TIMES
        RAR             /SAVE BIT THAT WAS
        DCA HOLDL       /ROTATED INTO LINK
        TAD HOLD        /GET ROTATED NUMBER
        AND K7          /MASK OFF FIRST 9 BITS
        TAD K260        /CONVERT TO ASCII
        TLS             /AND PRINT A DIGIT
        CLA CMA         /FORM -1 IN ACCUMULATOR
        DCA SW1         /SET SW1 TO CONTINUE NUMBER
        ISZ DIGCTR      /PRINTED 4 DIGITS YET?
        JMP EXIT        /NO: RESUME BACKGROUND
        CLA             /YES: SET SW1 TO
        DCA SW1         /SIGNAL A NEW NUMBER
        DCA SW2         /SET SW2 FOR CARRIAGE RETURN
        ISZ BUFFPT      /INCREMENT BUFFER POINTER
        ISZ PRNTCT      /AND ELEMENT COUNTER
        JMP EXIT        /RESUME BACKGROUND
RESTRT, CLA CLL         /CLEAR ACCUMULATOR AND LINK
        DCA MODE        /SET MODE FOR INPUT
        DCA SW2         /SET SW2 FOR CARRIAGE RETURN
        CMA             /SET SW3 FOR
        DCA SW3         /INPUT MODE BYPASS
        TAD BUFF        /GET FIRST BUFFER LOCATION
        DCA BUFFPT      /RESET BUFFER POINTER
        DCA AMOUNT      /AND ELEMENT COUNTER
        JMP EXIT        /RESUME BACKGROUND
EXIT,   CLA CLL         /CLEAR ACCUMULATOR AND LINK
        TAD FLAGS       /GET INTERRUPT FLAGS
        RTF             /RESTORE FLAGS
        CLA             /CLEAR ACCUMULATOR
        TAD AC          /RESTORE ACCUMULATOR
        JMP I 0         /RESUME BACKGROUND
END=.
$
```

Figure 6-20H Multiple Interrupt Demonstration Program

## DATA BREAK

Programmed transfers of data, including program interrupt transfers, pass through the accumulator. This requires that the content of the accumulator be saved before the transfer is performed, and later restored. This type of transfer is often too slow for use with extremely fast peripheral devices. Devices which operate at very high speed, or which require very rapid response from the computer, use the data break facility. The data break permits an external device to insert or extract core memory words directly, bypassing all program control. Because a computer program has no cognizance of transfers made in this manner, it is necessary to check for the presence of transferred data prior to using it. The data break is particularly well-suited to I/O devices that transfer large amounts of data in block form, such as random access disk files, high-speed magnetic tape systems, or high-speed drum memories.

### Accessing Data

Before a peripheral storage device may accept data from the computer, it must find the off-line location at which the data is to be stored. In the same manner, a peripheral device cannot send input data into the computer until it has found the location at which the data is presently stored. The process of finding a stored data word, or the location at which a data word is to be stored, is called *accessing data.* The time required for this process is the device *access time.* It is important to realize that even very fast peripheral storage devices have an access time of many machine cycles, so that many instructions could be executed in the time required for the peripheral device to access one data word.

### Single-Cycle Data Break

Data breaks are of two types: single-cycle and 3-cycle. In a single-cycle data break, the I/O device contains two registers which specify the core memory location with which the next data word is to be transferred (*current address,* or CA, register) and the negative of the number of words that remain to be transferred (*word count,* or WC, register). IOT instructions initiate the transfer by setting the WC and CA registers. Other IOT instructions indicate the direction of transfer and cause the device to perform

any preliminary operations which may be necessary to access the desired data. At this point, program execution continues and the device assumes full control of the data transfer.

Figure 6-21 provides a diagram of the single-cycle data break. After performing the necessary preliminary operations, the I/O device accesses the first data word and signals a data break request. This causes the central processor to suspend operation by disabling its major state generator and instruction register, while loading its memory address register with the contents of the device CA register. The device then generates a signal which either

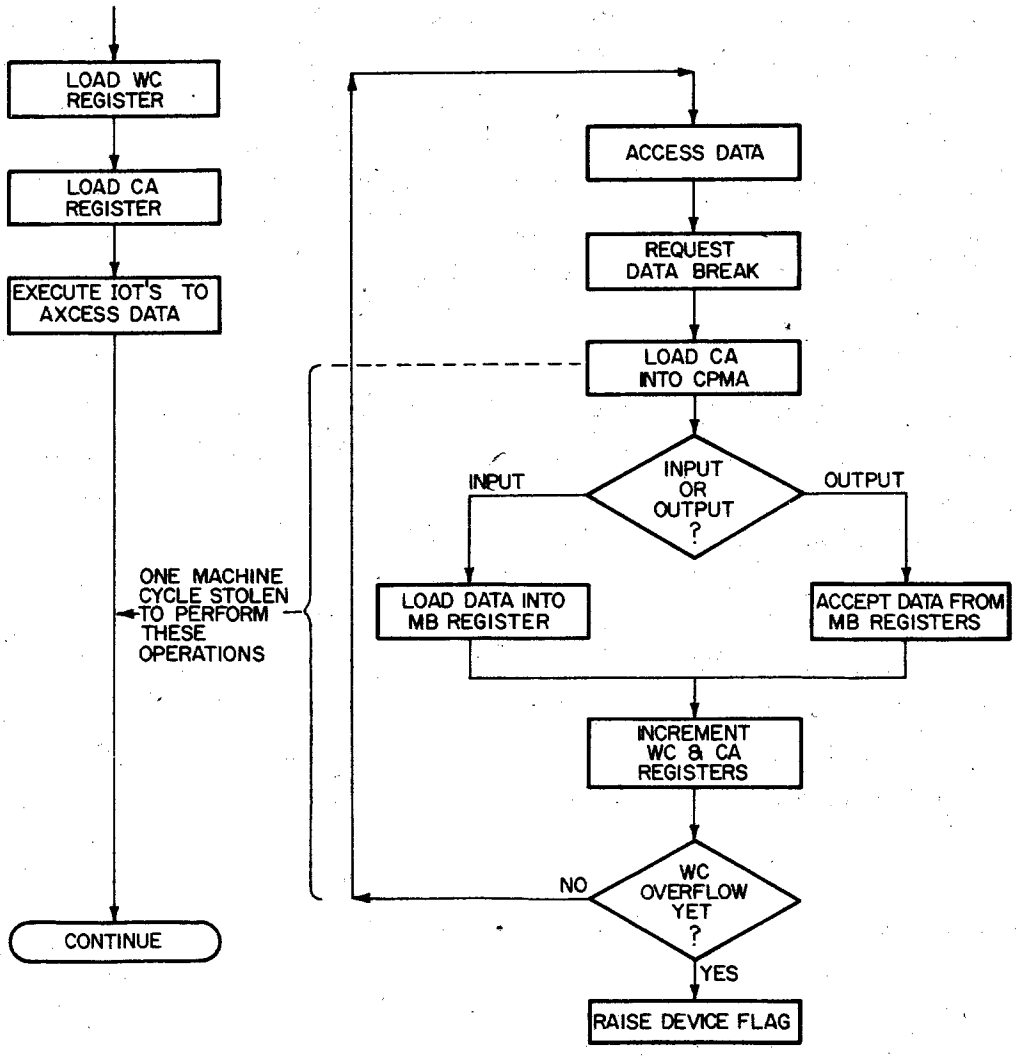



Figure 6-21 Single-Cycle Data Break

loads the memory buffer register with data or accepts data from this register, depending upon the direction of data transfer. When this operation is complete, the device increments its WC and CA registers and returns control to the central processor, which continues mainline execution.

Meanwhile, the I/O device begins to access the next data word to be transferred. This process continues until the device WC register overflows (contains 0000), indicating that no further data transfers are required.

**Cycle Stealing**

The process by which one cycle is "stolen" from the central processor in order to transfer data directly to or from core memory is called a *cycle-steal* operation. Each single-cycle data break causes the WC and CA registers to be incremented and transfers one word of data to or from core memory. The I/O device requests a data break whenever it is ready to transfer a word, and performs all transfers during cycles stolen from the central processor. Program execution continues between data breaks, while the I/O device is accessing the next data word.

**3-Cycle Data Break**

Devices which operate by means of a 3-cycle data break do not contain a WC register or a CA register. Instead, the addresses of two core memory locations are hard-wired into the device, and these core locations are used in place of the WC and CA registers. When a 3-cycle data break is requested, the first cycle is used to fetch the content of the core location designated as the WC register into the device, increment it, test for WC overflow, and return it to core memory. The CA register is incremented in the same manner during the second cycle, and loaded into the memory address register. The actual data transfer occurs during the third cycle.

In contrast to devices which use the single-cycle data break, 3-cycle devices increment the CA register *before* performing the actual data transfer. For this reason, the CA register of a 3-cycle device must always be loaded with one less than the address of the first data buffer location. The WC register of a 3-cycle device is loaded with the two's complement of the number of words to be transferred, just as with a single-cycle device.

From a programming viewpoint, the only other major difference between single-cycle and 3-cycle data breaks is that a single-cycle device contains its own WC and CA registers, which are loaded by means of IOT instructions. A 3-cycle device uses two core memory locations as its WC and CA registers, so that the registers must be loaded by means of memory reference instructions.

## EXERCISES

1. Write a subroutine ALARM which rings the teleprinter bell five times.

2. Write a format subroutine for the teleprinter to tab space the teleprinter carriage. The subroutine is entered with the number of spaces to be tabbed in the accumulator.

3. Write a program that will type a heading at the top of the paper and then type the numbers 1 through 10 down the left hand side of the page with a period after each number.

4. Write a program which will accept a 2-digit octal number from the Teletype keyboard and type "SQUARED=" and the value for the number squared, followed by "OCTAL" and a carriage return and line feed.

5. Extend the program written in Exercise 4 by adding routines to disallow the input of an 8 or 9 and type out an appropriate message.

6. Combine the program of Exercise 5 with a bit-rotating program to use the program interrupt facility.

# chapter 7
# dectape programming

## INTRODUCTION

The DECtape system is a standard option for PDP-8 series computers that serves as an auxiliary magnetic tape data storage facility. DECtape provides a distinct advantage over conventional magnetic tape because it stores information at fixed positions which may be directly addressed, much like conventional magnetic disk or drum devices. Yet unlike magnetic disk or drum devices, DECtape is bidirectional and adaptable to a wide variety of user data formats. In this manner, DECtape combines the versatility of magnetic disk and drum storage with the convenience and economy of magnetic tape.

## DATA BLOCKS

A reel of DECtape is organized into a series of addressable *blocks*. Each block consists of *control words*, used to identify the block, and *data words*, used for data storage. The data words contain twelve bits each; however, every control word contains eighteen bits. Blocks are usually numbered in sequence, from 0 to N-1, where N is the (octal) number of blocks on the tape. The maximum number of addressable blocks per tape is $2^{12}$, or 4096.

Every block contains ten control words. A block may contain any number of data words, with the restriction that the number of data words contained in each block must be an even multiple of three. If the number of blocks on a tape is known, the number of data words per block may be found by the following formula:

$$N_B = \frac{212080}{N_W+15} + 2$$

where $N_W$ = Decimal number of words per block.

$N_B$ = Decimal number of blocks per tape.

Disregard any fractional remainder.

## DATA CHANNELS

DECtape is divided longitudinally into five pairs of identical channels which run the entire length of the tape. Duplicating each channel helps to minimize any loss of information. The *timing channels* are used to reference fixed locations on the tape. The *mark-track channels* establish the format of information contained in the *data channels*. The three pairs of data channels are arranged in such a way that no two identical channels are adjacent. The arrangement of channels on the tape is shown in Figure 7-1.

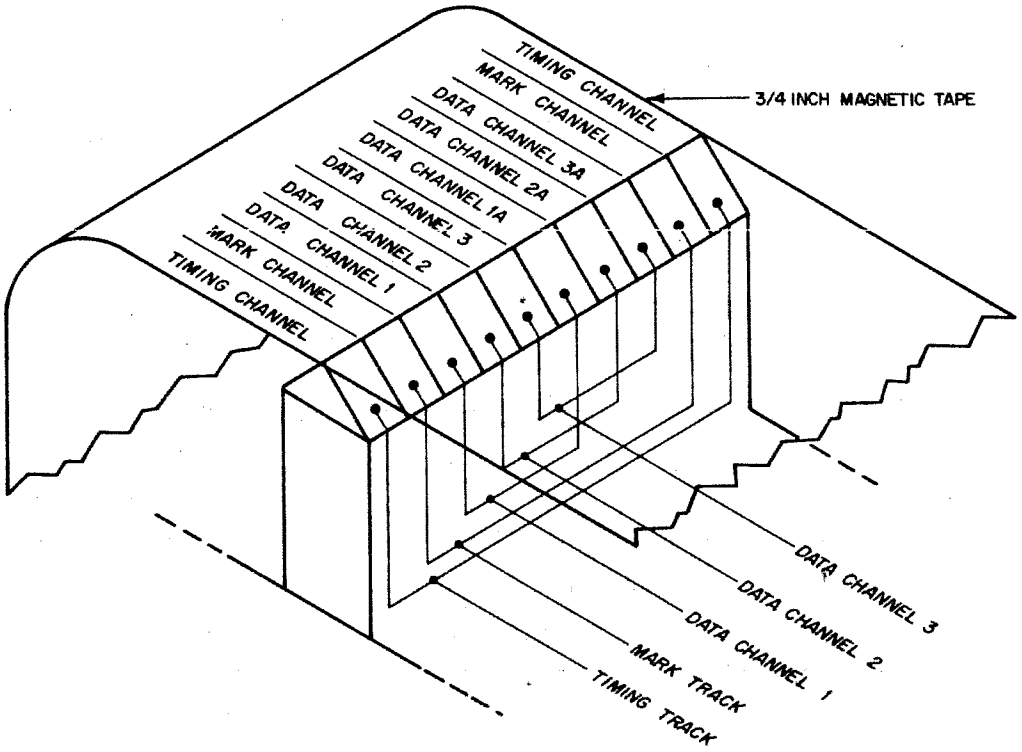



Figure 7-1 DECtape Data Channels

## STANDARD DECTAPES

Figure 7-2 indicates that a standard, 260 foot reel of DECtape contains $2702_8$ (or $1474_{10}$) *standard blocks* and two *end zones*. The figure also shows an enlarged diagram of one standard block, which contains $201_8$ (or $129_{10}$) twelve-bit data words and 10 eighteen-bit control words. Note that only the top half of the tape is shown. The five redundant channels have been omitted for clarity.

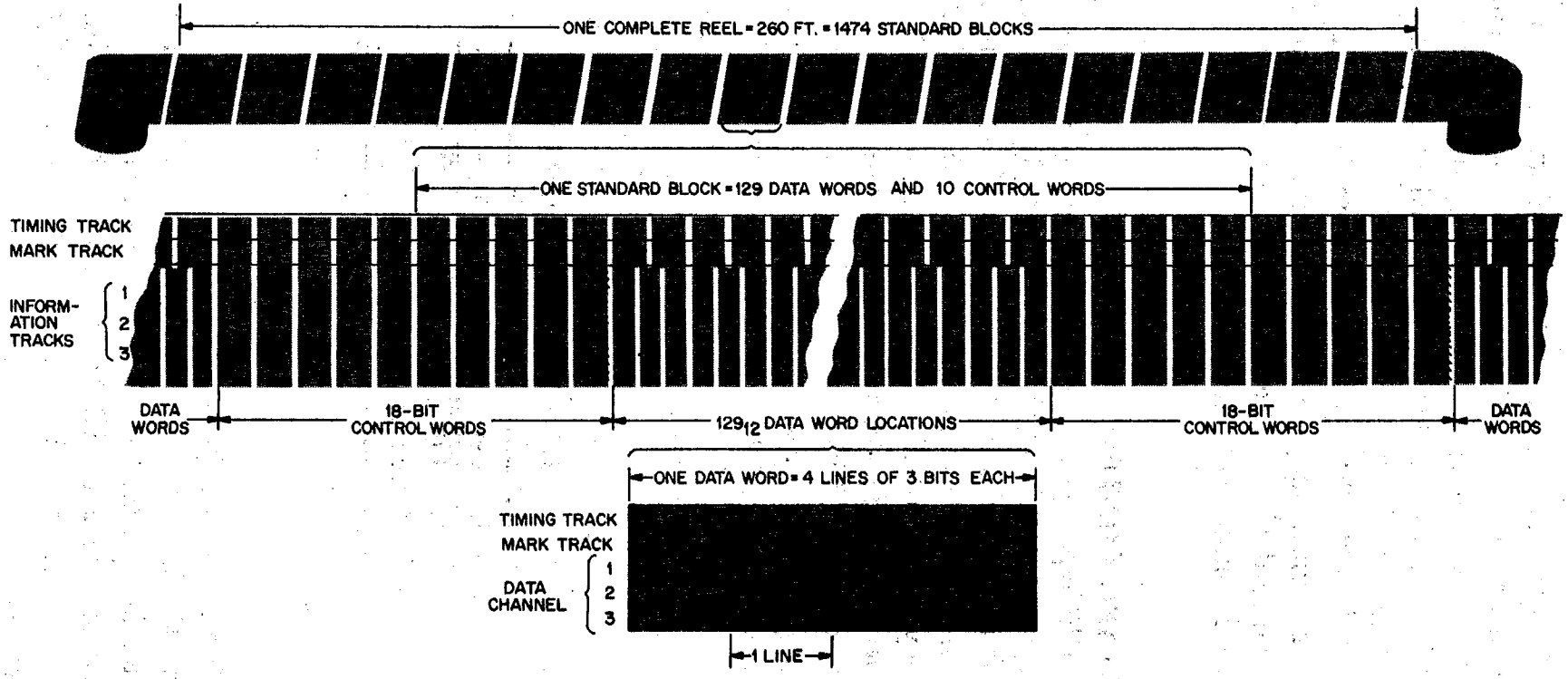



Figure 7-2 Standard DECtape Format

A further enlargement shows the structure of one data word. Once again, the redundant channels have been omitted. If one 12-bit binary number is written into this data word in the *forward* direction, the twelve bits will be stored sequentially in the numbered locations shown.

Nonstandard DECtapes having more or less than $129_{10}$ data words per block may be prepared to the user's specifications. As with standard blocks, the number of data words per block must be an even multiple of three.

## DECTAPE CONTROL UNIT

A DECtape system consists of up to eight single DECtape Transport Units (TU55) or four dual DECtape Transport Units (TU56), all operated by one TC08 DECtape Control Unit. Data is transferred between the computer and the DECtape Control Unit by means of the 3-cycle data break facility, described in Chapter 6. Transfers occur at a rate of one 12-bit word every 133 microseconds (±30%). The DECtape Control Unit uses core memory location 7754 in field zero as its word count register and core memory location 7755 in field zero as its current address register.

**TC08 DECtape Control Unit**

| Register | Core Address |
|---|---|
| Word Count (WC) | 7754 |
| Current Address (CA) | 7755 |

## DECTAPE STATUS REGISTERS

The DECtape Control Unit contains two 12-bit registers, designated Status Register A and Staus Register B, which may be loaded and read by IOT instructions. These registers consist of various switches and indicators, as shown in Figure 7-3 and Figure 7-4.

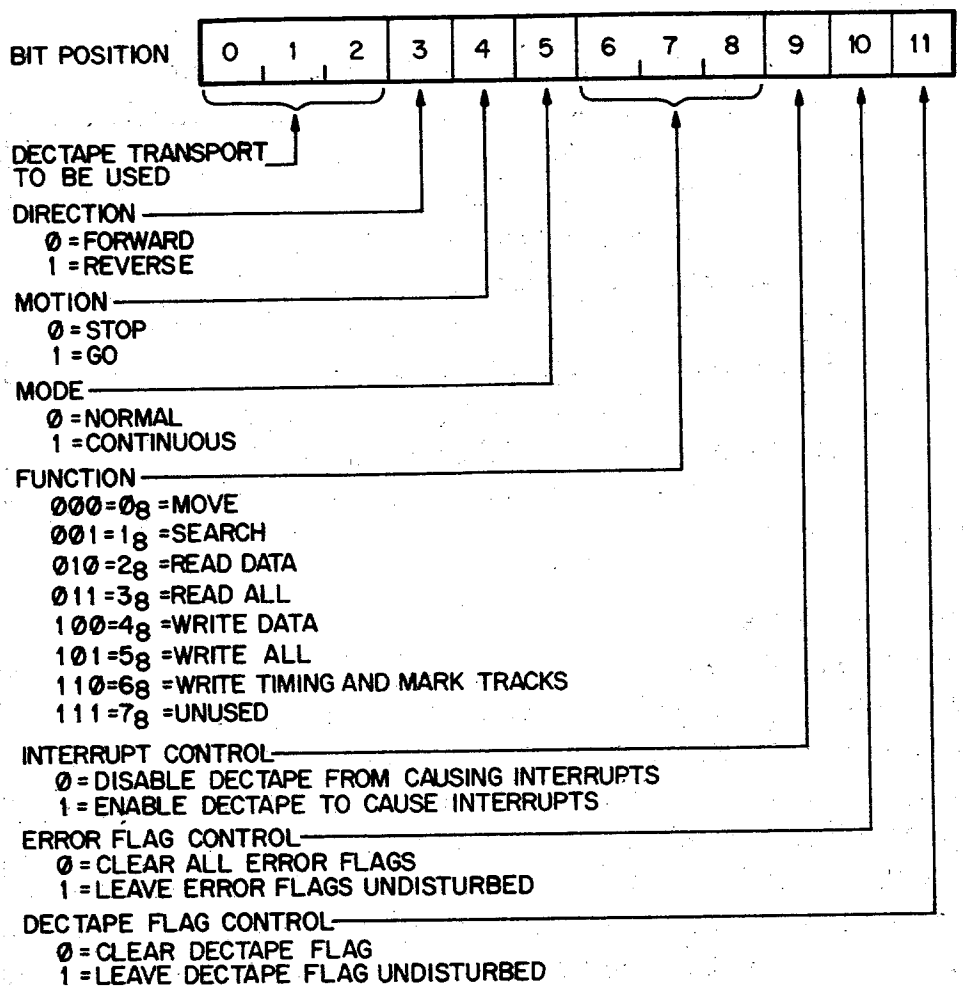



Figure 7-3 Status Register A

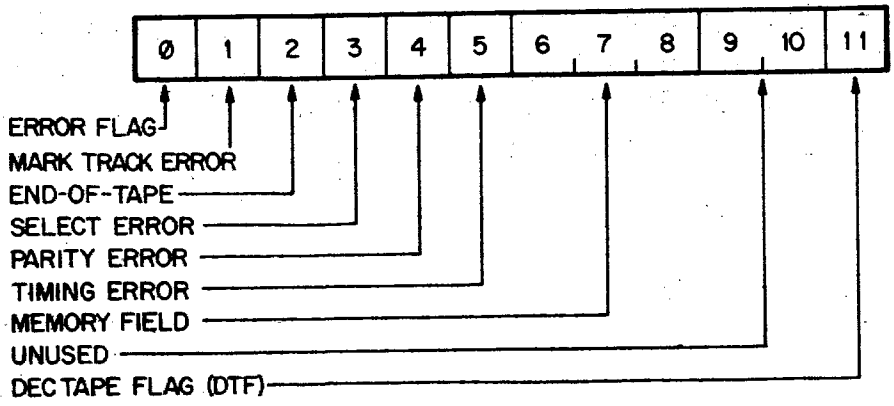



Figure 7-4 Status Register B

**Status Register B**

Bits 0-5 of Status Register B provide six indicators which flag error conditions that may occur. These indicators normally contain zeros, indicating the absence of an error condition. The possible errors, their abbreviations and causes are listed in Table 7-1. Note that parity and mark-track errors are hardware malfunctions, while timing and select errors are generally caused by the software.

Bits 6-8 of Status Register B specify the memory field from or to which data should be transferred. Bits 9 and 10 are not used. Bit 11 is called the DECtape Flag (DTF). In general, a value of 0 in this bit at any given time indicates that a DECtape operation is in progress. A value of 1 indicates that the current operation has been completed.

**Status Register A**

Status Register A designates which DECtape transport is to be used (bits 0-2), in which direction it should operate (bit 3) and whether operation should be commenced or discontinued (bit 4).

Bit 5 of Status Register A specifies the transfer mode, which may be either normal or continuous. In *normal mode*, data transfer continues until the end of a block is reached or a predetermined number of words has been transferred, whichever occurs first. In *continuous mode* data transfer continues, across block boundaries if necessary, until WC overflow occurs, indicating that the required number of words has been transferred.

Bits 6-8 of Status Register A designate the function to be performed. The seven possible functions are summarized in Table 7-2. The first four functions, MOVE, SEARCH, READ DATA and WRITE DATA, are used exclusively for most applications.

The remaining three bits of Status Register A indicate whether or not the interrupt system is to be used (bit 9), whether error flags in Status Register B are to be cleared (bit 10) and whether the DECtape Flag is to be cleared (bit 11).

**Table 7-1 TC08 Error Codes**

| Bit Position | Designation | A value of 1 in this bit indicates: |
|---|---|---|
| 0 | Error Flag | Detection of one or more of the following errors. |
| 1 | Mark-Track Error | Erroneous information was read from the mark track. |
| 2 | End-of-Tape Error | The end zone on either end of the tape is over the recording heads. |
| 3 | Select Error | This error is flagged 5 microseconds after loading Status Register A to indicate one or more of the following conditions:<br>a. The unit select code is not assigned to any transport or is assigned to more than one transport.<br>b. A write function was specified with the WRITE ENABLE/WRITE LOCK switch in the WRITE LOCK position.<br>c. An unused function code was specified (111 in bits 6-8 of Status Register A).<br>d. A write timing and mark track function was specified with the WRTM/NORMAL switch in the NORMAL position. |
| 4 | Parity Error | Checksum was incorrect during a read data function, or else WC overflow did not occur at the end of the first block read in continuous mode. This flag is only set simultaneously with the DTF. |
| 5 | Timing Error | A program fault caused one of the following conditions:<br>a. A data break could not occur within 66 microseconds ±30% of the data break request.<br>b. The DTF was not cleared before the control unit attempted to set it.<br>c. A read data or write data function was specified after the current block had been entered, preventing complete data transfer. |

**Table 7-2 DECtape Functions**

| Function | Description | | Possible Errors |
|---|---|---|---|
| MOVE (000) | Initiates motion of the specified tape drive in the specified direction. Errors are inhibited, except for end of tape errors. Used only to rewind tape. DTF, WC and CA are never changed. | | Select<br>End-of-Tape |
| SEARCH (001) | As the tape is moving in either direction, the sensing of a block mark causes a data transfer of the block number. The CA is not incremented. | | Select<br>End-of-Tape<br>Timing<br>Mark-Track |
| | Normal mode: DTF is set after each block number is transferred. | Continuous mode: WC is incremented after each block number is transferred. DTF is set when WC overflow occurs. | |
| READ DATA (002) | Transfers data into memory. Only data words are transferred. | | Select<br>End-of-Tape<br>Timing<br>Parity<br>Mark-Track |
| | Normal mode: DTF is set at the end of each block or when WC overflow occurs. | Continuous mode: DTF is set when WC overflow occurs. | |
| WRITE DATA (004) | Transfers data onto tape. When WC overflow occurs in the middle of a block, the block is filled out with zeros. Modes of operation are the same as for the READ DATA function. | | Select<br>End-of-Tape<br>Timing<br>Mark-Track |

**Table 7-2 (Cont'd)**

| Function | Description | | Possible Errors |
|---|---|---|---|
| READ ALL (003) | Transfers data into memory. The entire content of the information tracks is transferred, including control words.[1] | | Select End-of-Tape Timing Mark-Track |
| | Normal mode: DTF is set after each data word is transferred. | Continuous mode: DTF is set when WC overflow occurs. | |
| WRITE ALL (005) | Transfers data onto tape regardless of the data format. Only mark-track and end of tape errors are checked. Used to write block numbers on tape. Modes of operation are the same as for the READ ALL function.[1] | | Select End-of-Tape Timing Mark-Track |
| WRITE TIMING AND MARK TRACKS (006) | Used by DECtape formatting routines to write timing and mark-tracks, and to establish or change the length of blocks.[1] | | Select Timing |

## DECTAPE IOT INSTRUCTIONS

The six basic IOT instructions used to program the PDP-8 for DECtape operation are listed in Table 7-3. An instruction to load Status Register A is, in effect, an instruction to begin the operation specified by the bit configuration loaded. Once begun, the operation continues to execute until the DTF flags completion, and the tape remains in motion until a 1 is loaded into bit 3 of Status Register A.

[1] These functions are normally used only by DECtape formatting routines and diagnostics.

The memory field bits (positions 6-8) of Status Register B are loaded by means of a DTLB instruction. The remaining bits in Status Register B are loaded by specifying proper values for bits 10 and 11 of Status Register A. Note that Status Register A is loaded by exclusive ORing the contents of accumulator bits 0-9 into the register. Accumulator bits 10 and 11 are sampled directly, without the exclusive OR operation, and used to set the Status Register B flags.

**Table 7-3 DECtape IOT Instructions**

| Mnemonic | Octal Code | Operation |
|---|---|---|
| DTRA (Read Status Register A) | 6761 | The contents of Status Register A bits 0-9 are ORed with the contents of accumulator bits 0-9, and the result is placed in accumulator bits 0-9. |
| DTCA (Clear Status Register A) | 6762 | Status Register A is cleared. Status Register B is undisturbed. |
| DTXA (Load Status Register A) | 6764 | Bits 0-9 in the accumulator are EXCLUSIVE ORed with bits 0-9 of Status Register A and the result is loaded into bits 0-9 of Status Register A. Bits 10 and 11 in the accumulator are sampled, and the error flags are cleared if bit 10 is a 0. The DECtape flag is cleared if bit 11 is a 0. The accumulator is cleared. |
| DTSF (Skip on Flags) | 6771 | If either the error flag or the DECtape flag is set, the program counter is incremented to skip the next sequential instruction. |
| DTRB (Read Status Register B) | 6772 | The content of Status Register B is ORed with the content of the accumulator and the result is placed in the accumulator. |
| DTLB (Load Status Register B) | 6774 | Bits 6-8 of the accumulator are loaded into bit positions 6-8 of Status Register B. The accumulator is cleared. Error flags and DECtape flag are undisturbed. |

## PROGRAMMED DECTAPE OPERATION

Prior to using the TC08 DECtape system for data storage, a reel of DECtape must be prerecorded to insert the timing track, mark-track and block numbers. This function is performed in two passes by a DECtape formatting program utilizing the WRTM control operation. Successful execution requires that the manual switch on the DECtape Control Unit be placed in the WRTM position during the first pass. After one prerecording, a reel of DECtape may be used indefinitely.

Tape I/O operations are usually performed by a DECtape service routine, which uses the six DECtape IOTs to clear, read and load the status registers. Since the actual data transfer occurs by means of the 3-cycle data break, service routines often begin by using memory reference instructions to set the word count and current address registers. By loading an appropriate bit configuration into Status Register A, the SEARCH function is then initiated to locate the block number selected for transfer.

While searching, the DECtape control reads only block numbers. In normal mode the DTF is raised at every block number and the block number is transferred into the address specified by the CA register, which is not incremented during SEARCH. The contents of this location may be monitored until the desired block number is transferred, at which time a READ DATA operation must be instituted promptly (within 400 microseconds $\pm 30\%$) by reloading Status Register A.

During the READ DATA operation, the DECtape service routine monitors Status Register B constantly by means of a DTSF (skip on flags) loop to check for error conditions and determine when the DTF flags completion of data transfer. Meanwhile, data transfer continues without program control until the data block is exhausted or WC overflow occurs. Either of these events will raise the DTF (in normal mode), at which time the service routine may again load Status Register A, forcing a 0 into bit 4 to terminate the operation.

With the interrupt facility enabled, the DECtape service routine described above may be designed to execute as a foreground program, and mainline operations in the background program may be performed while the DECtape system is accessing data.

The operations summarized above may be broken down as follows:

1. Use an MRI to load the WC register with the negative (two's complement) of the number of words to be transferred.
2. Use an MRI to load the CA register with one less than the core address with which the first transfer is to be made. Successive transfers will be made with successive core locations.
3. Use a DTLB instruction to load Status Register B bits 6-8 with the desired memory field designation.
4. Begin to search for the desired block number by using a DTXA instruction to load Status Register A with an appropriate bit configuration. This should also clear all flags.
5. Remain in a skip-on-flag loop and test each block number as it is transferred until the desired block number has been transferred.
6. Reload Status Register A to clear all flags and begin READ DATA function.
7. Remain in skip-on-flag loop until the DTF flags completion.
8. Load a 1 into bit position 4 of Status Register A to halt the tape.

The procedure summarized above is neither complete nor efficient, although it does illustrate the sequence of operations generally employed. Step 5 is inefficient because it may be necessary to search the entire length of a tape before the desired block number is transferred. Steps 5 and 7 are incomplete because they do not test for error conditions, and any error condition will cause the program to skip out of the waiting loop before the current operation is finished. Finally, this procedure makes no provision for reversing the direction of tape motion, which may become necessary if either end zone is reached or if the desired block is initially accessed in the wrong direction. Techniques for dealing with all of these conditions will be developed in the following sections.

## USE OF THE DECTAPE FLAG

The DECtape flag may be tested to determine the status (in progress, or finished) of the current operation. In normal mode, the flag is set when WC overflow occurs. If WC overflow has *not* occurred and the flag is set:

1. During SEARCH function: A block mark has just been read and transferred to the address contained in the CA register. The CA register was not incremented, and the tape is still moving.

2. During READ DATA function: A full block of data has just been transferred to core memory via data break. The control unit is still in READ DATA mode and the tape is moving. The CA register contains the address of the next sequential core location following the last one read into. The next data block will be read into core beginning at this location.

3. During WRITE DATA function: A block of data has just been written on tape from core memory via data break. The control unit is still in WRITE DATA mode, and the tape is moving. The CA register contains the address of the next sequential core location following the last one transferred. The contents of core beginning at this location will be written into the next block on the tape.

In *continuous mode*, the DECtape flag is set only when WC overflow occurs, regardless of the function being performed.[2] This usually indicates that the current operation is complete and that execution should now be terminated under program control. Note that the tape does not stop moving automatically when the DTF flags completion of the current operation; however, any error condition except a parity error will cause the tape to stop.

---

[2] If transfer has continued across one or more block boundaries, the Parity Error Flag is also set as soon as WC overflow occurs.

## SELECTING DIRECTION

Data is normally read in the direction in which it was written. If data is read in the wrong direction, the data buffer will be filled in reverse order, with what was originally the first data word being placed in the last buffer position, and so on. Furthermore, since the DECtape system breaks every word into four 3-bit increments which it stores sequentially, each data word will be returned with its bit positions changed as shown below:

| ORIGINAL BIT POSITIONS: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RETURNED BIT POSITIONS: | 9 | 10 | 11 | 6 | 7 | 8 | 3 | 4 | 5 | 0 | 1 | 2 |

Finally, reading a data word in the wrong direction causes the word to be returned with every bit complemented. Thus, an octal 1234 written in the reverse direction would be read as 3456 in the forward direction.

## REVERSING DIRECTION

The DECtape transport requires at least one full standard block to reverse direction and cycle back up to reading speed. Most programmers allow two full blocks to assure successful turn around. If block number 105 is to be read when block 200 is positioned under the recording heads, for example, it is necessary to search in the reverse direction for block 102. The tape may be turned around just inside block 102, and it will cycle up to reading speed before reaching block 105. In general, to read block N forward, search in reverse for block N-3.

The standard DECtape format provides sufficient space between the end data blocks and the end of the tape to perform a turn around in the end zone and successfully access either block 0 in the forward direction or block 2701 in reverse. Some bootstrap loaders take advantage of this by performing a MOVE reverse until the end zone is reached, and then a READ forward to load a routine stored in block 0.

## ACCESSING DATA BLOCKS

The most efficient method for accessing data blocks is summarized below:

1. SEARCH forward in normal mode until the first block number is transferred into core at the location specified by the CA register.
2. When the DTF flags completion of this transfer, test the block number to determine how many blocks must be bypassed and in which direction. Allow two extra blocks if a turn around is required.
3. Load the WC register with the number of blocks which must be bypassed, then SEARCH in continuous mode until the DTF flags WC overflow. Turn around, if necessary.
4. The next block encountered will be the desired block.

## ALLOCATING STORAGE AREAS

Any process which attempts to access single DECtape blocks sequentially will be inefficient, because it may be necessary to turn around twice to access each block. This problem develops whenever large amounts of code or slow I/O operations must be executed between accessing adjacent blocks of DECtape. Under these circumstances, the block being accessed may overshoot the recording heads before the intermediate operation is complete. Stopping the tape to allow full execution of the intermediate process is not a solution, because the tape requires two blocks to cycle up to reading speed, so that it will still overshoot the block being accessed.

An efficient way of solving this problem is to break up data files and programs into large segments spanning many blocks, and then alternate a recorded segment with a blank segment on the tape. While a blank segment is passing under the recording heads, it is possible to perform an intermediate process and complete it in time to access the first block of the next recorded segment. Blank segments may be recorded in a similar manner in the reverse direction. The DECtape Copy program uses this type of operation to access 14 block segments separated by 14 block intervals. It is very efficient at reading and copying DECtapes.

## PROGRAMMING FOR ERROR CONDITIONS

It is generally desirable for every DECtape service routine to provide an extensive method for handling error conditions. This might include:

1. Requesting operator intervention after a select error.
2. Counting changes in tape direction to guard against tape rocking loops, which can occur when the desired block is nonexistent or when the tape mechanism is malfunctioning.
3. Attempting to read a block a second time following a parity error.
4. Informing the user when a nonrecoverable error condition exists.

## PROGRAMMING FOR INTERRUPTS

The DECtape control unit contains a 1-bit register called the DECtape Control Flag (DTCF, not to be confused with the DECtape flag) which always contains the inclusive OR of the Error Flag and the DECtape flag. It is this register which is actually sampled by the DTSF skip-on-flag instruction.

If the interrupt facility is enabled (ION instruction) a 1 in bit position 9 of Status Register A causes an interrupt to occur whenever an error condition or an operation-complete condition (or both) is flagged by the DTCF. This makes it unnecessary for the DECtape service routine to monitor Status Register B during every operation by means of skip-on-flag waiting loops. The time spent in waiting loops may be used to execute mainline instructions, provided that these instructions do not attempt to use data before the DECtape service routine has had time to transfer it.

Because the DECtape system is comparatively fast, an interrupt service routine will normally test the DTCF near the top of the skip chain. This creates a problem when the DECtape system is not in use. An interrupt from a lower priority device will eventually occur, in which case any bit configuration in Status Register A will be interpreted as a select error when the DTCF is tested, and the program may loop indefinitely. Such problems may be prevented by replacing the DTSF skip-on-flag test with a NOP whenever DECtape is not in use.

## IDTAPE SUBROUTINE[3]

The IDTAPE subroutine illustrates one method of programming for DECtape operation. IDTAPE operates with the interrupt facility disabled. It performs data transfers in continuous mode, so that multiple blocks of data may be transferred in a single operation. Searching may commence in either direction, to minimize access time. One full block is allowed for turn around when searching in the reverse direction. IDTAPE reads and writes in the forward direction only.

IDTAPE must be stored in field zero, but it will read or write data into any specified memory field. It will not automatically cross field boundaries. Errors flagged while searching cause IDTAPE to remain in the SEARCH loop indefinitely. If the specified block number is nonexistent, it hangs up in a tape rocking loop. Errors flagged during data transfer cause the branch IDSERR to be taken with tape motion halted and Status Register B error flags set.

The IDTAPE calling sequence is:

```
0000  JMS (IDTAPE)  Effective JMS to IDTAPE, i.e., indirect
                     JMS if IDTAPE is not on same page
                     as calling sequence.
0001  WORD 1,        Bits 0–2, unit number.
                     Bit 3, start search (0=forward 1=re-
                     verse).
                     Bits 6–8, memory field for transfer.
                     Bit 10, error return (0=JMP WORD 5
                     1=JMP I WORD 5).
                     Bit 11, function (0=READ 1=WRITE).
0002  WORD 2,        Block number for start of transfer.
0003  WORD 3,        Two's complement of the number of
                     words to transfer.
0004  WORD 4,        Memory address minus 1 of first transfer.
0005  WORD 5,        Error return or address for error return
                     (to correspond to Bit 10 of WORD 1).
0006  RETURN,        Transfer completed; return with AC
                     cleared.
```

On exit, the DECtape drive will halt.

---

[3] This subroutine is not currently available from the DEC Software Distribution Center, and is included here for the user's convenience.

```
*200
/AN ORIGIN OF 0200 IS USED FOR TESTING
/HOWEVER THIS SUBROUTINE MAY OCCUPY THE FIRST
/113 (OCTAL) LOCATIONS OF ANY PAGE IN FIELD ZERO
/LOCATIONS 7754 AND 7755 ARE ALSO USED
ID7400, 7400            /AND MASK MUST BE
                        /FIRST WORD IN PAGE
IDTAPE, 0               /ENTER SUBROUTINE
        CLA             /CLEAR ACCUMULATOR
        TAD I IDTAPE    /GET WORD 1
        DCA IDCODE      /STORE IT
        ISZ IDTAPE      /ADVANCE POINTER TO WORD 2
        TAD IDCODE      /GET WORD !
ID0200, AND ID7400      /MASK OFF BITS 4-11
        TAD ID0010      /PUT INTO SEARCH MODE
        DTCA DTXA       /CLEAR AND LOAD STATUS A
        DTLB            /LOAD STATUS B FOR FIELD 0
        TAD IDWC        /GET ADDR OF WC REGISTER
        DCA I IDCA      /INITIALIZE CA FOR SEARCH
                        /THEN FALL THRU ERROR
                        /ROUTINE TO START TAPE
                        /NORMALLY ENTERED WITH
                        /STATUS B IN ACCUMULATOR
IDSERR, RTL             /MOVE END ZONE FLAG
        RAL             /INTO LINK
        CLA CML         /COMPLEMENT LINK
        TAD ID0200      /MASK STOP/GO BIT ON AND
                        /ENTER DECTAPE SEARCH LOOP
IDCONT, SNL             /TAPE IN END ZONE?
        TAD ID0400      /YES: REVERSE DIRECTION
        DTXA            /NO: BEGIN SEARCH
        DTSF DTRB       /MONITOR FLAGS UNTIL
        JMP .-1         /A FLAG IS RAISED
        SPA             /ERROR FLAG RAISED?
        JMP IDSERR      /YES: TAKE ERROR BRANCH
        DTRA            /NO: MUST BE DTF, SO GET
        RTL             /DIRECTION BIT AND
        RTL             /ROTATE IT INTO THE LINK
        SZL CLA         /MOVING FORWARD?
        TAD ID0002      /NO: GET "BLOCK TO FIND" -2
        TAD I IDWC      /YES: GET LAST BLOCK SEEN
        CMA             /FORM ONE'S COMPLEMENT
        TAD I IDTAPE    /ADD IN BLOCK TO FIND
        CMA             /TWO'S COMP MIGHT SET LINK
        SZA CLA         /BLOCK NUMBERS MATCH?
        JMP IDCONT      /NO: REENTER SEARCH LOOP
        SZL             /YES: MOVING FORWARD?
        JMP IDCONT+1    /NO: TURN AROUND
        ISZ IDTAPE      /ADVANCE POINTER TO WORD 3
        TAD I IDTAPE    /GET WORD 3
        DCA I IDWC      /LOAD WC REGISTER
        ISZ IDTAPE      /ADVANCE POINTER
        TAD I IDTAPE    /GET WORD 4
        DCA I IDCA      /LOAD CA REGISTER
```

```
        TAD IDCODE      /GET WORD 1
        DTLB            /LOAD STATUS B FIELD BITS
        IAC             /SET BIT 11 FOR
        AND IDCODE      /READ OR WRITE THEN
        RTL CLL         /ROTATE INTO POSITION AND
        RTL             /BUILD XOR MASK TO
        TAD ID0130      /LOAD STATUS REGISTER A
        DTXA            /BEGIN READ/WRITE
        DTSF DTRB       /MONITOR STATUS B UNTIL
        JMP .-1         /A FLAG IS RAISED
        ISZ IDTAPE      /ADVANCE TO WORD 5
        SMA             /IS ERROR FLAG SET?
        ISZ IDTAPE      /NO: ADVANCE TO WORD 6
        SPA CLA         /YES: KEEP ERROR RETURN
        TAD IDCODE      /GET INDIRECT RETURN BIT
        RTR             /ROTATE INTO LINK
        SNL CLA         /INDIRECT RETURN?
        JMP .+3         /NO: MAKE NORMAL RETURN
        TAD I IDTAPE    /YES: CHANGE INDIRECT RETURN
        DCA IDTAPE      /TO DIRECT RETURN
        DTRA            /GET STATUS REGISTER A
        AND ID0200      /MASK STOP/GO BIT OFF
        TAD ID0002      /PRESERVE ERROR FLAGS
        DTXA            /LOAD STATUS A TO STOP TAPE
        JMP I IDTAPE    /RETURN
IDWC,   7754            /POINTER TO WC REGISTER
IDCA,   7755            /POINTER TO CA REGISTER
ID0010, 10              /SETS BIT 8 FOR SEARCH
ID0400, 400             /SETS BIT 4 TO REVERSE TAPE
ID0130, 130             /BUILDS XOR MASK
ID0002, 2               /HANDY CONSTANT
IDCODE, 0               /STORAGE FOR WORD 1
$
```

## DECTAPE SYSTEM SOFTWARE

DECtape Software, described in the following sections, provides the programmer with four major operational materials:

1. Input/output subroutines which may be included in larger programs.
2. Routines for copying and formatting DECtapes. These programs will duplicate non-standard tapes, write timing and mark-track channels, insert block format information and perform similar chores for the programmer.
3. A library system for storing and retrieving programs on DECtape.
4. OS/8, which provides the most efficient means of using both DECtape and disk storage.

## DECTAPE SUBROUTINES

DECtape subroutines allow the programmer to read, write and search DECtapes using prewritten and tested subroutines which execute with the interrupt facility enabled, so that mainline instructions may be performed during tape operations. The DEC library supplies these subroutines on one ASCII symbolic tape which has no origin, and ends with the pseudo-op PAUSE. It must be assembled with a user program, and the resulting binary tape loaded with the Binary Loader. The following restrictions should be observed:

1. The routines are designed to be used with standard tapes (129 words per block). The last data word in each block is not accessed, so that each block is effectively one core page long.

2. The routines will read or write in the forward direction only.

3. Data may be transferred from or into any memory field, but these routines will not transfer data across field boundaries.

4. The routines require $200_8$ consecutive storage locations. They may be located on any page of core in field 0, except page 0, and they must all reside on the same page, filling that page completely.

5. An ION instruction will be executed by the routines upon entry; however, the user is responsible for storing the contents of the accumulator and link when an interrupt occurs. The user must define a register called MCOM on page 0. This register is initialized by the tape routines and is used for communication with the interrupt system. If the DTF is set on the occurrence of an interrupt, the accumulator must be cleared and the instruction JMP I MCOM must be executed to return control to the tape routines.

**DWAIT Subroutine**

The DWAIT subroutine tests the motion bit (bit 4) of Status Register A. If the motion bit is a 1, the routine cycles through this test process until the motion bit is cleared, indicating that the tape has stopped and the previous operation is now complete. It then returns control to the location following the JMS which called it. DWAIT is called automatically, whenever necessary, by all of the following DECtape subroutines.

| DWAIT Calling Sequence | | Explanation |
|---|---|---|
| GOSUB, | JMS I DWAITI | Where DWAITI contains the address of subroutine DWAIT. |
| RETURN, | | After tape motion has stopped, DWAIT returns control to the location following the JMS which called it. |

**SEARCH Subroutine**

The SEARCH subroutine should only be used to position the tape. It is called automatically by the READ and WRITE subroutines. In the event that the tape is moving, the SEARCH subroutine will transfer control to the DWAIT subroutine, and continue to execute when the tape stops moving. If the user wishes to continue mainline execution immediately upon completion of the search, the completion return address in the following calling sequence should be the address of the interrupt exit routine.

| SEARCH Calling Sequence | | Explanation |
|---|---|---|
| GOSUB, | CLA CLL<br>TAD BLOCK | Where BLOCK contains the number of the block being sought. Any other method of getting the block number into the AC is acceptable. |

| SEARCH Calling Sequence | Explanation |
|---|---|
| JMS I SERCHI | Where SERCHI contains the address of routine SEARCH. SERCHI may be on page 0 or the same page as the calling sequence. |
| CMPADD | Where CMPADD is the completion return address. When the correct block has been found, control transfers to the address contained on this line with the tape stopped and the interrupt system off. |
| U000 | Bits 0-2 contain the DECtape transport unit number. The rest of the word contains zeros. For example, using unit 3 this would be 3000. |
| IRETRN | Where IRETRN is an intermediate absolute return. When searching starts, control is transferred to this line with the interrupt on to allow multiprocessing while the tape is in motion. If multiprocessing is not desired this line should read "JMP .—1" which causes the program to idle in a closed loop consisting of one instruction until the requested block has been found and an interrupt occurs. |

**READ and WRITE Subroutines**

The READ and WRITE subroutines will transfer any number of memory pages to or from standard blocks of DECtape, however only whole pages will be transferred onto whole data blocks, and vice versa. The user's calling sequence specifies the starting location, the memory field, the starting block number and the tape unit number. As with the SEARCH routine, the program waits until any previous tape operation has been completed. These routines use the SEARCH subroutine to find a specified block number, however they do not stop the tape until after execution is complete.

| READ/WRITE Calling Sequence | Explanation |
|---|---|
| JMS I READI<br>or<br>JMS I WRITEI | Where READI contains the address of R128 and WRITEI contains the address of W128 |
| CORADD | Where CORADD is the address of the first core location on the page to be used for transfer. |
| U0F0 | Where bits 0-2 contain the DECtape transport unit number and bits 6-8 contain the memory field. The remainder of the word contains zeros. (E.g., for unit 3 and memory field 1 this would be 3010.) |
| -NUM | Where -NUM is the negative (2's complement) of the number of successive blocks to be read or written. |
| BLCK | Where BLCK is the number of the first block on tape to be read or written. |
| IRETRN | Where IRETRN is an intermediate absolute return. When the routines start searching for the block specified, control is transferred to this line to allow multiprocessing while the tape is in motion. If multiprocessing is not desired, this line may contain a JMS to the DWAIT routine. |

**DECtape Copy Program**

The DECtape Copy program (DTC8) provides a simple, efficient method for copying from one DECtape to another on PDP-8 series computers. Features include the capability of handling nonstandard block lengths (up to $1550_{10}$ 12-bit words per block) and the ability to reread and verify the copied data. The routine is internal and monitor independent.

DTC8 resides in memory between locations 0000 and 1547, with the remainder of core divided into two buffers. The starting address is 0200, and the routine may be restarted at this address at any time. DTC8 is normally distributed as a binary tape and loaded with the Binary Loader. Alternately, it may be stored on DECtape or disk and loaded with the monitor system. Two DECtapes must be mounted, with WRITE LOCK enabled on the input tape.

Teletype input is required at the conclusion of each printed message generated by DTC8. The operator may type:

1. CTRL/C, which causes a branch to location 7600.
2. A string of octal digits (0 to 7), which will be interpreted as an octal number. Only the rightmost four digits are preserved, so that 1234567 is interpreted as 4567.
3. RETURN, which terminates the input string. If no digits were typed, the input value is zero.
4. Any other characters, which are ignored and not echoed.

The operator is expected to respond to the following messages printed by the program:

DECTAPE COPY FROM UNIT

Type the input unit number (0-7). Only the last digit typed is accepted. Terminate with a carriage return.

TO UNIT

Type the output unit number, which must be different from the input unit number. Terminate with a carriage return.

FINAL BLOCK TO COPY (OCTAL)

Type the octal block number of the last block to be copied. If no digits are typed, the program will copy to the end of the tape. If a number is typed, it must be greater than or equal to the number of the first block to be copied. Terminate with a carriage return.

PDP-8 WORDS PER BLOCK nnn

The program types the octal number of 12-bit words in each block. No operator response is required.

VERIFY OUTPUT? (0=YES, 1=NO):

If a 1 is typed, the program does not reread and verify the output copy. Terminate with a carriage return.

DONE

Program indicated completion. Type RETURN key to restart the program or CRTL/C to branch to location 7600.

Any illegal Teletype input causes the program to restart itself. CTRL/C may be typed any time after the start of the program to cause a branch to location 7600.

A parity error while reading from the input unit causes the message:

PARITY ERROR ON BLOCK nnn

Copying continues, but when execution is finished the operator may attempt to copy the designated block separately.

A validation error is flagged when the information written is not the same as the information read in. Three consecutive validation errors cause the message:

WRITE ERRORS ON UNIT #n

This message can occur only if validation was requested. Typing the RETURN key causes the program to attempt to reread and validate the erroneous data.

A select error on either tape drive causes the message:

SELECT ERROR ON UNIT #n

After a select error, the operator should correct the error condition and then type the RETURN key to continue program execution.

Mounting an output tape with more than $1550_{10}$ 12-bit words per block causes the message:

BLOCK LENGTH ERROR

After a block length error, the program automatically restarts.

**DECtape Formatting Program**

A software package is available to perform formatting and maintenance operations on DECtapes. It provides for recording of timing and mark-track channels and permits block formats to be recorded for any block length. Patterns may be written in these blocks, and then read out and verified. Specified areas of tape may also be "rocked" for specified periods of time. In this manner, a reel of tape may be thoroughly checked before it is used for data storage. For detailed information, refer to the TC01-TU55 DECtape Formatter (DEC-08-EUFB) available from the DEC Software Distribution Center.

## DECTAPE LIBRARY SYSTEM

The DECtape Library System permits the user to build a complete file of his active programs and continuously update it. It is capable of calling programs by name from the keyboard and allowing for expansion of programs stored on tape. It conforms to existing system conventions, in that all of core except for the last memory page is available to the programmer, and it permits the Binary Loader to reside in core at all times. The Library System fully restores the initial state of the computer when it exits.

One use for the Library System may be illustrated as follows. A program that will be run repeatedly is written in PDP-8/E FORTRAN. At the keyboard, the operator may call the FORTRAN compiler from the library tape and compile his program, obtaining an object program on paper tape. He may then call the FORTRAN operating system from the library to load and run his object program. Finally, the library program UPDATE may be called. The operator defines a new program file, consisting of his object program and the FORTRAN operating system, then adds it to the library tape. The program is now available for easy access on the library tape.

The minimum library system, called a *skeleton library*, occupies the first $40_8$ blocks of a standard DECtape. It contains the following routines:

INDEX — Causes the names of all routines on the library tape to be printed on the Teletype.

UPDATE — Allows the user to add routines to the library tape.

GETSYS — Used to generate a new library tape.

DELETE — Causes a specified routine to be deleted from the library tape.

ESCAPE — Causes the library system to exit.

A prerecorded skeleton library tape may be obtained from the DEC Software Distribution Center. This tape should be duplicated with DTC8 (or else GETSYS should be used to generate a second skeleton library tape) and the original should then be stored in a safe place.

**The Directory**

The directory is part of the library system. It contains the names of files on the library tape and all information that is required by the library system to load, delete, or add named files. The directory contains $348_{10}$ usable locations, with each entry requiring a minimum of seven locations. The structure of a single directory entry is as follows:

First 3 Words: File name in trimmed ASCII with characters packed 2 per word. For example, INDEX would appear as:

| | |
|---|---|
| 1116 | /IN |
| 0405 | /DE |
| 3000 | /X |

Fourth Word: Starting DECtape block number for the file.

Fifth Word: Starting core address of the file.

Sixth Word: First in a list of core specifications for the file. The format of these specifications is shown in Figure 7-5.

Last Word: 0000 to terminate list of core specifications. If there is only one specification, the directory entry is seven words long. This is the minimum length for an entry.

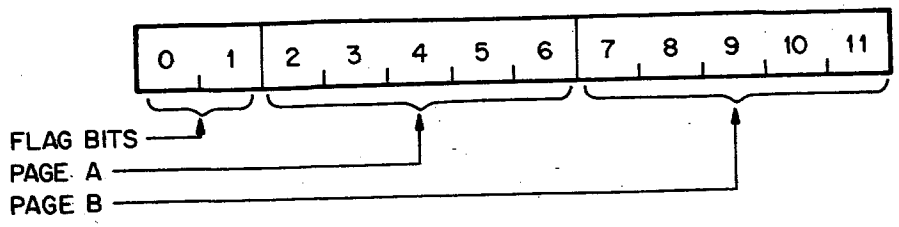



Figure 7-5 Format of Core Specifications

Bit 0: If set to 0, the two 5-bit page-number specifications (page A and page B) designate discrete single pages which belong to the file.

If set to 1, page A signifies the first page in a series of contiguous pages and page B signifies the last page in the series.

For example: 0247 or 00 00101 00111 means that page 5 and page 7 belong to this file. Observe that the user may not save the last page of core (7600-7777) with one of his files.

Bit 1: This is always set to the same value as bit 0.

Bits 2-6: Five bit page number specification (Page A above).

Bits 7-11: Five bit page number specification (Page B above).

There are spaces for almost $50_{10}$ names in the directory. UPDATE will determine whether or not the directory is full and if so, it will print a message to this effect. There are $2640_8$ usable blocks on the library tape, which is more than adequate in view of this limitation on the directory size.

**Using the Library System**

All skeleton library routines are called from the Teletype by typing the name of the routine and then a carriage return. Some of the routines generate messages which request operator input. Typing a RUBOUT causes any input on the current line to be ignored and the line to be retyped.

Typing INDEX and RETURN causes a list of all routines stored on the library tape to be generated on the Teletype.

The UPDATE routine assumes that a file to be added to the library tape was in core before the library system was loaded. Typing UPDATE and RETURN generates the message:

NAME OF PROGRAM:

The operator must type a name consisting of one to six characters, and then a carriage return. All characters are legal except:

@, ↑, TAB, FORM FEED, and LINE FEED. UPDATE will now type:

SA (OCTAL):

The operator must type the octal starting address of the file being loaded. If the file does not have a proper starting address (as, for example, the floating point package), the starting address of the system loader (7600) or the HLT in the BIN Loader (7700) may be specified. UPDATE will now type:

PAGE LOCATIONS:

The operator must specify the page locations in core memory at which the file is presently stored. This information may be provided to UPDATE in either of two forms: <XXXX> which means the single page on which the octal address XXXX falls, or <XXXX, XXXX> which means the page on which the first address falls through and including the page on which the second address falls. UPDATE accepts information of this type until a semicolon (;) is received. Spaces, tabs, carriage returns, and line feeds are ignored between location elements. For example, if a program occupies locations 1-2354, 4600-7577, and 2400-2577, UPDATE might be told: <0,2200> <2400> <4600,7577>. The numbers must be in ascending order, and any numbers lying within the same page are considered equivalent. UPDATE will now add the new file to the library, add an entry to the directory, and transfer control to the file-loading program, leaving the directory in core.

Typing DELETE and RETURN generates the following message:

NAME OF FILE TO BE DELETED

The operator must type a name (up to six characters) followed by a carriage return. DELETE searches the directory for the designated name and generates an error message if it is not found or if it is the name of a system program. If the name is found, DELETE removes it from the directory, removes the named program from the tape, and repacks the tape. DELETE then transfers control to the file-loading program, leaving the directory in core.

Typing GETSYS and RETURN generates the following message:

SKELETON TAPE WILL BE CREATED ON UNIT #

The operator must type a single digit (1-7) and then a carriage return. GETSYS expects a preformatted, standard DECtape to be mounted on the specified tape unit, with the WRITE ENABLE /WRITE LOCK switch in the WRITE ENABLE position. When the new library system has been created, GETSYS transfers control to the file-loading program, leaving the directory in core.

Typing ESCAPE and RETURN causes the library system to restore all of core and exit. The condition of the computer should be identical to its condition before the library system was loaded.

In the example shown below, the operator used the Binary Loader to load the FORTRAN compiler. He then loaded the library system, and the following interaction took place:

```
INDEX                                          Typed by user.

ESCAPE
UPDATE
DELETE                                         Presently stored pro-
GETSYS                                         grams listed by system.
PALIII

UPDATE                                         Typed by user.

PROGRAM NAME     :FRTRAN
SA(OCTAL)        :200                          User supplies data re-
PAGE LOCATIONS   :<0,7400>;                    quested by system.

INDEX                                          Second INDEX.

ESCAPE
UPDATE
DELETE                                         Stored programs listed
GETSYS                                         by system.
PALIII
FRTRAN

DELETE                                         Typed by user.

NAME OF FILE TO BE DELETED:PALL                User typed a mistake
NAME OF FILE TO BE DELETED:PALIII              and used RUBOUT
                                               key.
INDEX

ESCAPE
UPDATE                                         Third INDEX.
DELETE
GETSYS
FRTRAN
ESCAPE
```

The amount of time required to load a file from tape into core memory depends upon the file location on the tape. If the file is near the beginning, loading time will be about 8 seconds. UPDATE executes in 30 to 45 seconds. DELETE time varies too much to make an estimate possible; it may take as long as several minutes. GETSYS requires approximately 30 seconds.

The system has one DECtape error halt at location 7670. No recovery is possible at this point, and any attempt to restart may result in destruction of the library tape data.

**TC01 Bootstrap Loader**

The library system bootstrap loader resides in the last page of core with the RIM and BIN loaders; however, with the bootstrap in core, only the RIM loader is operable. Since this bootstrap consists of only a few instructions, it may be loaded from the switch register, following the procedure for toggling in the RIM loader,

**TC01 Bootstrap Loader**

| Location | Instruction |
|---|---|
| 7600 | 6224 |
| 7601 | 6774 |
| 7602 | 1221 |
| 7603 | 4213 |
| 7604 | 1222 |
| 7605 | 3355 |
| 7606 | 1223 |
| 7607 | 4213 |
| 7610 | 0000 |
| 7611 | 0000 |
| 7612 | 0000 |
| 7613 | 0000 |
| 7614 | 6766 |
| 7615 | 3354 |
| 7616 | 6771 |
| 7617 | 5216 |
| 7620 | 5613 |
| 7621 | 0600 |
| 7622 | 7577 |
| 7623 | 0220 |

explained in Appendix E. Check carefully to ensure that the bootstrap is loaded correctly. If it is not, the system tape may be destroyed by an attempt to load the system. For checking procedures, See Figure 4-3.

To load the library system using the bootstrap loader, ensure that the library system tape is mounted on a DECtape transport. The tape may be wound to any point along its length, but at least four turns of tape should be on either reel.

1. Set the transport unit selector dial to 0 (or 8).
2. Set the LOCAL/OFF/REMOTE rocker switch to REMOTE.
3. Set the WRITE LOCK/WRITE ENABLE rocker switch to WRITE ENABLE.
4. Set the starting address of the bootstrap (7600) into the console switch register and press ADDR LOAD and CLEAR then CONT. The library system will load from the DECtape into core.

When the system is loaded, the Teletype keyboard will be enabled, awaiting operator commands.

**TD8-E DECTAPE SUBROUTINE**

A TD8-E DECtape system consists of up to four TU56 dual DECtape transport units with one TD8-E DECtape control unit for each dual transport unit. This system is similar to the TC08 DECtape system in that it uses the same hardware transport unit and tape data format as the TC08. However, the TD8-E system relies on programmed transfer for I/O operations, rather than the 3-cycle data break, and requires that status monitoring operations such as error sensing and checksum generation be performed by software under program control.

A TD8-E DECtape system is best employed by utilizing OS/8 or the TD8-E DECtape Subroutine, which is a general data handling routine with a standardized calling sequence that facilitates tape I/O operations and provides full compatibility with OS/8 device handlers. This subroutine requires two adjacent pages of core memory, in any memory field, for each TU56 dual transport to be serviced. Thus, operation of a maximum configuration system requires that four copies of the subroutine (with four different sets of assembly parameters) reside in core.

**Assembly Parameters**

Values for five parameters must be supplied when the TD8-E subroutine is assembled. These parameters and the permissible values which may be supplied are listed below.

| | |
|---|---|
| DRIVE = 10 | DECtape units 0 and 1 |
| DRIVE = 20 | DECtape units 2 and 3 |
| DRIVE = 30 | DECtape units 4 and 5 |
| DRIVE = 40 | DECtape units 6 and 7 |
| ORIGIN = nnnn | Specify an absolute origin, which will also be the entry point for the lowest-numbered DECtape transport unit. The entry point for the other unit will be ORIGIN + 4. |
| AFIELD = n | Designate a field ($0 \leqslant$ AFIELD $\leqslant 7$) into which the subroutine will be loaded. |
| MFIELD =n0 | Specify 10 times the value designated for AFIELD. |
| WDSBLK = | Indicate the octal number of data words per block. |

Assume, for example, that an assembly of the subroutine to handle DECtape units 0 and 1 is to be loaded into memory field 0, pages 24 and 25, and standard DECtape format is to be used. Input to the assembler would consist of:

```
DRIVE  = 10
ORIGIN = 5000
AFIELD = 0
MFIELD = 0
WDSBLK = 201
```

This would be followed by a source copy of the subroutine. If DECtape units 2 and 3 are to be used simultaneously, a second assembly of the subroutine may be loaded into, say, memory field 1, pages 35 and 36. Assuming that standard format is also employed on these tape drives, the assembly parameters will be:

```
DRIVE  = 20
ORIGIN = 7200
AFIELD = 1
MFIELD = 10
WDSBLK = 201
```

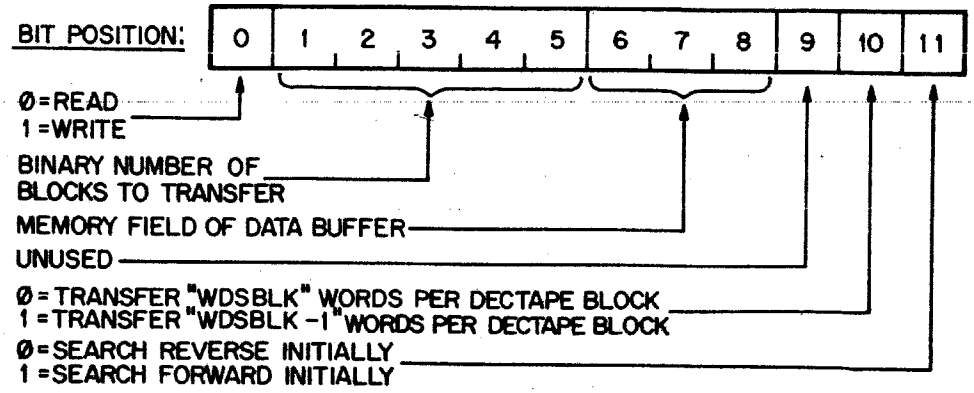
In this manner, as many assemblies of the TD8-E Subroutine as are necessary may be loaded into any available core locations, provided that each assembly occupies two contiguous pages. The subroutine may be relocated temporarily during program execution, if necessary, but it may not be called at any location except the one for which it was assembled.

**Calling Sequence**

A call to the TD8-E Subroutine should have the following general format:

```
GOSUB, CDF DATA      /DESIGNATE DATA FIELD=
                     /CURRENT FIELD
       CIF MFIELD    /FIELD IN WHICH SUBROUTINE
                     /WAS LOADED
       JMS ENTRY     /EITHER "ORIGIN" OR
                     /"ORIGIN + 4"
       ARG1
       ARG2
       ARG3
       JMP ERROR     /ERROR RETURN
       JMP CONT      /NORMAL RETURN
```

The first argument which must be supplied has the following format:



The second argument (ARG2) contains the core memory address of the data buffer. Data will be transferred to or from sequential core locations, beginning at this address.
The third argument (ARG3) specifies the DECtape block number at which transfer should commence. Data will be transferred to or from sequential blocks until the number of blocks designated by bits 1-5 of the first argument (ARG1) has been transferred.

If no errors are encountered, the subroutine takes the normal return to the fifth location following the JMS which called it with all tape motion stopped, the accumulator cleared and the instruction and data fields equal to the value of DATA specified in the subroutine calling sequence. Any error condition sensed during a data transfer operation causes the subroutine to take the error return to the fourth location following the JMS which called it. An error return with the accumulator cleared indicates a select error. Parity, timing and checksum errors return a value of 4000 in the accumulator, as does any attempt to access a non-existent block number.

# chapter 8

# floating-point packages

## INTRODUCTION

Floating-point packages provide an easy means of performing basic arithmetic operations, such as addition, subtraction, multiplication and division, using floating-point numbers. They also provide extended function capabilities for the computation of natural logarithms, exponential functions, basic trigonometric functions and the like. The floating-point package maintains a high degree of precision, and greatly facilitates I/O operations in floating-point notation. This is particularly useful for computations involving numerous arithmetic operations on variables whose magnitudes may vary widely. The floating-point package, or FPP, stores very large or very small numbers by saving only the significant digits and assigning an exponent to account for leading and trailing zeros.

There are three floating-point packages designed for use with the PDP-8/E. The 23-bit Extended Arithmetic Element Floating-Point Package (EAE FPP) may be employed on any PDP-8/E equipped with a KE8-E Extended Arithmetic Element and an LT33 Teletype. The EAE FPP is described in detail in this chapter.

The 23-bit Floating-Point Package (non-EAE FPP) may be used on any PDP-8 series computer with an LT33 Teletype. This FPP is functionally equivalent to the EAE FPP in many respects; in particular, the two are fully program compatible. Important differences between the EAE FPP and the non-EAE FPP are noted throughout this chapter.

The 27-bit Floating-Point Package (27-bit FPP) is also designed for use on any PDP-8 series computer equipped with an LT33 Teletype. The 27-bit FPP provides extended precision for computations that require accuracy in excess of 6 or 7 significant digits. The 27-bit FPP is program compatible with both the EAE FPP and the non-EAE FPP, and similar to these packages in most respects. Differences are noted throughout the chapter.

Each floating-point package is supplied as a binary paper tape and is loaded into memory via the Binary (BIN) Loader (see Appendix E). Since a floating-point package is actually a collection of subroutines, the user must also load a program of his own which calls the floating-point package (or one of the individual subroutines) and tells it what operations to perform. The binary tape of the user-program is normally loaded after that of the package; this is particularly important if the user is not calling part of the package (for example, the extended functions) and his program overlays that part.

## ASSEMBLY INSTRUCTIONS

Each Floating-Point Package is also available on three source paper tapes which the user may assemble with PAL III (or the equivalent) if he intends to alter the package. The source tapes should be assembled in ascending numerical order: tape 1 first, followed by tape 2 and tape 3. Tapes 1 and 2 end with the pseudo-op PAUSE, while tape 3 ends with a dollar sign.

If the FPP is assembled with PAL8, the user must define several floating-point and PDP-8/E instructions which are used by the interpreter but not contained in the PAL8 symbol table. The following paper tape should be prepared and used as the first tape of the assembly, before tape 1 of the FPP:

EAE FPP
(DEC-8E-NEAEA-A-PA1, PA2, and PA3)

```
FIXMRI FADD  = 1000
FIXMRI FSUB  = 2000
FIXMRI FMPY  = 3000
FIXMRI FDIV  = 4000
FIXMRI FGET  = 5000
FIXMRI FPUT  = 6000
FEXT  = 0000
FNOR  = 7000
SWP   = 7521
CAM   = 7621
MQA   = 7501
MQL   = 7421
SGT   = 6006
PAUSE
```

non-EAE FPP
(DEC-08-NFPPA-A-PA1, PA2, PA3)
27-bit FPP
(DEC-08-NFPEA-A-PA1, PA2, PA3)

```
FIXMRI FADD  = 1000
FIXMRI FSUB  = 2000
FIXMRI FMPY  = 3000
FIXMRI FDIV  = 4000
FIXMRI FGET  = 5000
FIXMRI FPUT  = 6000
FEXT  = 0000
FNOR  = 7000
PAUSE
```

Following assembly of the FPP, a user program may be assembled. The user program should begin with the following pseudo-instruction sequence, or a tape containing the following sequence terminated with a PAUSE pseudo-op should be assembled before the user program:

Assembling with PAL III

```
FIXMRI  FJMP  = 0000
FIXMRI  FJMS  = 7000
FISZ    = 0000
FEXT    = 0000
FSQU    = 0001
FSQR    = 0002
FSIN    = 0003
FCOS    = 0004
FATN    = 0005
FEXP    = 0006
FLOG    = 0007
FNEG    = 0010
FIN     = 0011
FOUT    = 0012
FFIX    = 0013
FLOT    = 0014
FNOR    = 7000
FCDF    = 7001
FSW0    = 7002
FSW1    = 7003
FHLT    = 7004
FSMA    = 7110
FSZA    = 7050
FSPA    = 7100
FSNA    = 7040
FNOP    = 7010
FSKP    = 7020
```

Assembling with PAL8

```
FIXMRI FJMP  = 0000
FIXMRI FADD  = 1000
FIXMRI FSUB  = 2000
FIXMRI FMPY  = 3000
FIXMRI FDIV  = 4000
FIXMRI FGET  = 5000
FIXMRI FPUT  = 6000
FIXMRI FJMS  = 7000
FISZ    = 0000
FEXT    = 0000
FSQU    = 0001
FSQR    = 0002
FSIN    = 0003
FCOS    = 0004
FATN    = 0005
FEXP    = 0006
FLOG    = 0007
FNEG    = 0010
FIN     = 0011
FOUT    = 0012
FFIX    = 0013
FLOT    = 0014
FNOR    = 7000
FCDF    = 7001
FSW0    = 7002
FSW1    = 7003
FHLT    = 7004
FSMA    = 7110
FSZA    = 7050
FSPA    = 7100
FSNA    = 7040
FNOP    = 7010
FSKP    = 7020
```

## FLOATING POINT NOTATION

A floating-point number may be written as a *mantissa,* which consists of the floating-point number with its decimal point shifted a given number of places in either direction, and an exponent,

which indicates the number of places that the decimal point was shifted and the direction of the shift. A negative exponent corresponds to a shift to the right, while a positive exponent corresponds to a shift to the left. For example, the decimal number 12.625 may be expressed in the following ways:

| | Mantissa | Exponent |
|---|---|---|
| $12.625 = 12.625 \times 10^{0}$ | 12.625 | 0 |
| $= 1.2625 \times 10^{1}$ | 1.2625 | 1 |
| $= 0.12625 \times 10^{2}$ | 0.12625 | 2 |
| $= 1265.0 \times 10^{-3}$ | 12625.0 | −3 |

A floating-point number which has been converted to a mantissa and an exponent may be recovered by mutiplying the mantissa by the Eth power of the radix (base) in use, where E is the exponent.

**Normalization**

A floating-point number may be represented in an infinite variety of ways, since the decimal point may be shifted any number of places in either direction. If the decimal point is shifted until it appears immediately to the left of the most significant digit, the number is said to be *normalized*. The mantissa of a normalized floating-point number may be stored as an integer, since the decimal point is understood to appear to the left of the most significant digit. In the discussion which follows, all floating-point numbers are assumed to be expressed in normalized floating-point notation, as indicated below.

| Decimal Number | Normalized Floating-Point Notation | |
|---|---|---|
| | Mantissa | Exponent |
| 12.625 | .12625 | 2 |
| 0.0012 | .12 | −2 |
| −1530.0 | −.153 | 4 |
| 0.0 | .0 | 0 |
| −89.9 | −.899 | 2 |
| −0.0899 | −.899 | −1 |

When a floating-point number is expressed in normalized notation, the mantissa usually falls in the range:

$$R^{-1} \leqslant | \text{ mantissa } | < 1$$

where R is the radix of the number system used. Thus, the absolute value of the mantissa of a decimal floating-point number will be greater than or equal to 1/10 but less than 1 if the decimal point is positioned to the left of the most significant digit, in accordance with the convention presented above. The only exception to this rule is the floating-point number zero, which is defined to have a mantissa and an exponent both equal to zero.

In computer applications, all numbers are manipulated and stored in binary notation. The preceding discussion applies equally well to decimal or binary numbers, in that a binary number may be converted to normalized floating-point notation by shifting the decimal point to the left of the most significant digit and assigning an exponent equal to the (directed) number of places that the decimal point was shifted. Since the left-most significant digit will always be a binary 1, the mantissa conforms to the convention:

$$1/2 \leqslant | \text{ mantissa } | < 1$$

except for the special case of zero.

**Number Representation**

PDP-8 floating-point numbers are stored in three consecutive 12-bit core memory locations as follows.

EAE AND NON-EAE FPP

The first location, which has the lowest core address, contains the exponent. The second word contains the twelve most significant bits of the mantissa, called the *high-order* mantissa. The third word, which has the highest core address, contains the last twelve (least significant) bits of the mantissa, or the *low-order* mantissa. As with one-word integers stored in the conventional manner, if the exponent or mantissa is negative, its two's complement is used, thus the first bit of both the exponent and the high-order mantissa may be considered as sign bits. The low-order mantissa is unsigned, however, and should be considered an extension of the high-order mantissa.

The core storage location of a floating-point number may be tagged with a symbolic address, as shown in Figure 8-1. In this

example, the floating-point number 12.625 is stored in core location "FPNUM." The tag FPNUM is associated with the core location at which the exponent is stored, while the mantissa occupies locations FPNUM+1 and FPNUM+2.

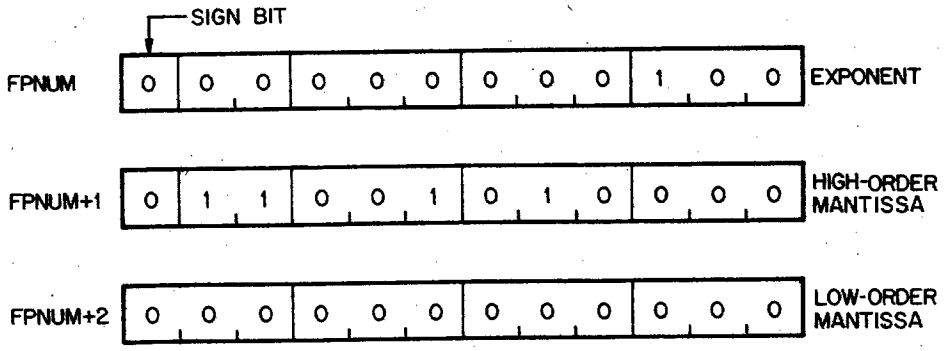



Figure 8-1 Storage Allocation for a 23-bit Floating-Point Number

27-BIT FPP

The first word (lowest core address) contains the sign of the mantissa, the exponent in bias $200_8$ notation, and the higher-order 3 mantissa bits. The second two words are the middle order and low order mantissa, respectively, in sign-magnitude notation.

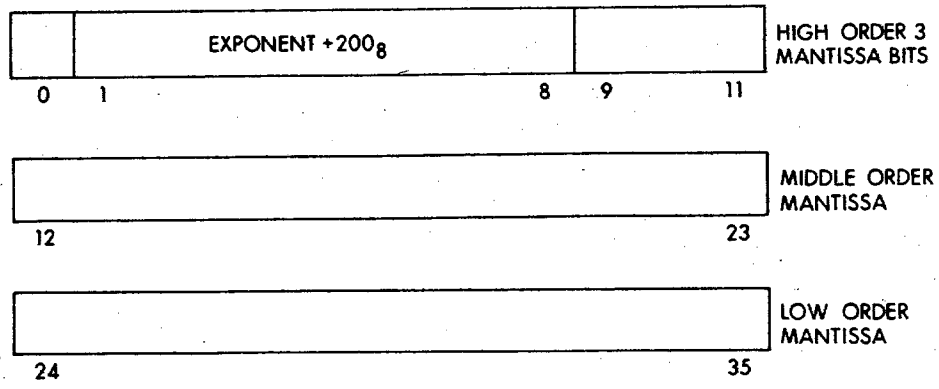



Figure 8-2 Storage Allocation for a 27-bit Floating-Point Number

The sign of the number is bit 0 of the first word. The value of the exponent is obtained by subtracting $200_8$ from bits 1 through 8 of the first word.

## USING THE FLOATING-POINT PACKAGE

The Floating-Point Package contains subroutines which perform floating-point operations using three core memory locations—44, 45 and 46, which are collectively designated as the floating accumulator, or FAC. Some of the subroutines require a floating-point

argument. The Floating Add Subroutine, for example, must be supplied with the address of a floating-point number which is to be added to the content of the FAC. Other floating-point subroutines do not require an argument because they operate on the FAC directly. The Floating Square Root Subroutine, for example, operates directly on the FAC by replacing the current value of the FAC with the square root of this value. It does not require an argument.

Most of the floating-point subroutines may be called by a user program at any time, in precisely the same manner that user subroutines are called, subject to the restriction that the subroutine call, the FPP itself and the argument (if required) must all reside in the same memory field. In general, all FPP subroutines should be entered with the hardware accumulator cleared.

The Floating Subtract Subroutine, for example, may be called by an indirect JMS to its entry address. The location following the subroutine call must contain the address of the floating-point argument to be subtracted from the FAC. Once this operation has been performed, the subroutine will return control to the core location following the address of the argument (i.e. the second location following the JMS). The Floating Square Subroutine may also be called by an indirect JMS to its entry address. Since this subroutine does not require an argument, it will square the contents of the FAC and return control to the location immediately following the JMS which called it.

When used in this manner, the FPP is being employed in *single-instruction mode*. In this mode of operation, every call to an FPP subroutine causes one floating-point operation to be performed on the FAC. Table 8-1 lists the floating-point subroutines which may be called in single-instruction mode, specifying which subroutines require an argument and which don't, along with their absolute entry addresses and a description of the operations they perform.

**NOTE**

Subroutine addresses are constant for all three floating-point packages. Thus, the user may easily upgrade from a NON-EAE to an EAE system if greater speed is required and from either 23-bit FPP to the 27-bit FPP if greater accuracy is required.

Figure 8-3 is a program which calls the 23-bit floating-point subroutines in single-instruction mode. This program calls the Floating Input Subroutine to accept a floating-point number from the Teletype, then calls the Floating Multiply Subroutine to multiply the input by 10.0. Finally, the program stores the result of these operations and halts.

```
*200
        KCC             /INITIALIZE KEYBOARD
        TLS             /AND TELEPRINTER
        JMS I FINP      /CALL INPUT SUBROUTINE
        JMS I FMPYP     /CALL MULTIPLY SUBROUTINE
        TEN             /ADDRESS OF OPERAND
        JMS I FPUTP     /CALL PUT SUBROUTINE
        STORE           /STORAGE ADDRESS
        HLT             /HALT UPON COMPLETION
FINP,   6200            /POINTER TO INPUT ROUTINE
FMPYP,  6600            /POINTER TO MULTIPLY ROUTINE
FPUTP,  7322            /POINTER TO PUT ROUTINE
TEN,    4               /EXPONENT
        2400            /HIGH-ORDER MANTISSA
        0000            /LOW-ORDER MANTISSA
STORE,  0               /3-WORD FLOATING-POINT
        0               /STORAGE LOCATION
        0
$
```

Figure 8-3 Using the FPP in Single-Instruction Mode

The same program could be converted for use under the 27-bit FPP by changing the constant TEN (necessary because of the different floating-point format) to look like:

```
TEN,    2045        /EXPONENT
        0000        /HIGH-ORDER MANTISSA
        0000        /LOW-ORDER MANTISSA
```

The user should note that in single-instruction mode, *the hardware AC must be zero* when calling a floating-point routine. The contents of the link are not important.

**Table 8-1 Floating-Point Subroutines**

| Subroutine and Entry Address | Function |
|---|---|
| FLOATING ADD 7000 | Add the argument to the content of the FAC and store this result in the FAC. |
| FLOATING SUBTRACT 7117 | Subtract the argument from the content of the FAC and store this result in the FAC. |
| FLOATING MULTIPLY 6600 | Multiply the argument by the content of the FAC and store this result in the FAC. |
| FLOATING DIVIDE 6722 | Divide the content of the FAC by the argument and store this result in the FAC. An error results from any attempt to divide by zero. |
| FLOATING GET 7306 | Load the argument into the FAC. |
| FLOATING PUT 7322 | Replace the argument with the content of the FAC. |
| FLOATING INVERSE SUBTRACT 6400 | Subtract the content of the FAC from the argument and store this result in the FAC. |
| FLOATING INVERSE DIVIDE 6412 | Divide the content of the FAC into the argument and store this result in the FAC. |
| FLOATING NORMALIZE 7265 | Normalize the content of the FAC and store this result in the FAC. |
| FLOATING SQUARE 7564 | Multiply the content of the FAC by itself and store this result in the FAC. |
| FLOATING SQUARE ROOT 6451 | Compute the positive square root of the absolute value of the content of the FAC and store this result in the FAC. |
| FLOATING SINE 5000 | Compute the sine of the content of the FAC (in radians) and store this result in the FAC. |
| FLOATING COSINE 5053 | Compute the cosine of the content of the FAC and store this result in the FAC. |
| FLOATING ARCTANGENT 5200 | Compute the primary arctangent of the content of the FAC and store this result in the FAC. |

**Table 8-1 (Cont.) Floating-Point Subroutines**

| Subroutine and Entry Address | Function |
|---|---|
| FLOATING EXPONENTIAL 5135 | Compute the exponential function of the content of the FAC and store this result in the FAC. |
| FLOATING LOGARITHM 5263 | Compute the natural logarithm of the content of the FAC and store this result in the FAC. An error results if the content of the FAC is negative or zero. |
| FLOATING NEGATE 7135 | Negate the content of the FAC and store this result in the FAC. |
| FLOATING INPUT 6200 | Accept a floating-point decimal number from the Teletype keyboard, convert it to a floating-point binary number, normalize and truncate if necessary, and store this result in the FAC. |
| FLOATING OUTPUT 5600 | Convert the content of the FAC to a decimal floating-point number and print this result on the Teletype printer. |
| FIX 5500 | Compute the largest integer that is not larger than the content of the FAC and store this result in the exponent word of the FAC (loc. 44). An error occurs if this result falls outside the range $-2047 \leqslant X \leqslant 2047$. |
| FLOAT 5533 | Compute the floating-point number that is equal to the content of the exponent word of the FAC (loc. 44) and store this result in the FAC. |

The FPP also contains an *interpreter,* which is a subroutine that decodes and executes floating-point pseudo-instructions. The interpreter may be called in the same manner as any other subroutine, however the FPP loads a pointer to the interpreter into core location 0007 of the memory field in which it resides, so that the interpreter is most conveniently called by a JMS I 7 instruction.

To call the FPP from a memory field other than the one in which it resides, use an effective JMS to location 7400 of the field in which the package has been loaded. The following is an

example of a call to the floating-point package where multiple data fields are involved.

```
        N=MEMORY FIELD CONTAINING CALL
        M=MEMORY FIELD CONTAINING FPP
        CDF   N0      /MAY BE OMITTED IF CURRENT DF=N
        CIF   M0      /MAY BE OMITTED IF CURRENT IF=M
        JMS I P7400
         .
         .
        (PSEUDO INSTRUCTIONS)
         .
         .
P7400,  7400
```

The floating data field is originally set to the hardware data field upon entry to the interpreter. This may be changed via the FCDF instruction which has a format similar to the normal PDP-8 CDF instruction. The floating data field is interpreted like the normal data field in that it applies only to the operand of an indirectly addressed memory reference instruction (op codes 1-6).

The interpreter essentially accepts all of the pseudo-instructions following the JMS which called it as arguments. Beginning with the first pseudo-instruction following the interpreter call, the interpreter decodes each pseudo-instruction as an effective JMS to the appropriate floating-point subroutine and passes the subroutine an argument, if required. When used in this manner, the FPP is being employed in *interpretive mode*. In the interpretive mode the package can be called from any memory field, and the user can access data in any memory field.

The pseudo-instructions which are interpreted as calls to those subroutines that require an argument are closely analogous to standard memory reference instructions. The first three bits of the pseudo-instruction specify which floating-point operation is to be performed (i.e. which subroutine to call) while the last nine bits specify the effective address of the argument according to the same conventions used for effective address generation by standard memory reference instructions. The pseudo-instructions which are interpreted as calls to those subroutines that do not require an argument are analogous to the operate microinstructions in that these pseudo-instructions do not reference a core memory location.

The basic arithmetic operations (FADD, FSUB, FMPY, FDIV) require that both FAC and floating operand be normalized. All four yield a normalized result.

**NOTE**

Pseudo-Instructions are the same for each of the three FPP's to facilitate conversion of user programs from one package to another. No alteration of the user program is necessary to convert from the EAE FPP to the NON-EAE FPP and vice versa. To convert a user program from either 23-bit package to the 27-bit FPP, only the floating-point constants need be changed.

**Table 8-2 Floating-Point Pseudo-Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| FEXT | 0000 | Exit the interpreter and execute the next sequential instruction as a normal machine instruction. |
| FSQU | 0001 | Call the Floating Square Subroutine. |
| FSQR | 0002 | Call the Floating Square Root Subroutine. |
| FSIN | 0003 | Call the Floating Sine Subroutine. |
| FCOS | 0004 | Call the Floating Cosine Subroutine. |
| FATN | 0005 | Call the Floating Arctangent Subroutine. |
| FEXP | 0006 | Call the Floating Exponential Subroutine. |
| FLOG | 0007 | Call the Floating Logarithm Subroutine. |
| FNEG | 0010 | Call the Floating Negate Subroutine. |
| FIN | 0011 | Call the Floating Input Subroutine. |
| FOUT | 0012 | Call the Floating Output Subroutine. |
| FFIX | 0013 | Call the Fix Subroutine. |
| FLOT | 0014 | Call the Float Subroutine. |

**Table 8-2 (Cont.) Floating-Point Pseudo-Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| FNOP | 0015 | No operation. Available to the user. |
| FNOP | 0016 | No operation. Available to the user. |
| FNOP | 0017 | No operation. Available to the user. |
| FISZ | 0020 to 0177 | Floating Increment and Skip if Zero: Increment the content of the core memory location designated by bits 5-11 of the FISZ pseudo-instruction and skip the next sequential pseudo-instruction if the content of this locaton becomes zero. An FISZ pseudo-instruction may only reference page zero locations between 0020 and 0177 inclusive. |
| FJMP | 0200 to 0777 | Floating Jump: Performs the same function as a standard JMP memory reference instruction except that it is not possible to FJMP to a location on page zero directly. |
| FADD | 1nnn | Call the Floating Add Subroutine. Use nnn to calculate the effective address of the operand. |
| FSUB | 2nnn | Call the Floating Subtract Subroutine. Use nnn to calculate the effective address of the operand. |
| FMPY | 3nnn | Call the Floating Multiply Subroutine. Use nnn to calculate the effective address of the operand. |
| FDIV | 4nnn | Call the Floating Divide Subroutine. Use nnn to calculate the effective address of the operand. |
| FGET | 5nnn | Call the Floating Get Subroutine. Use nnn to calculate the effective address of the operand. |
| FPUT | 6nnn | Call the Floating Put Subroutine. Use nnn to calculate the effective address of the operand. |
| FNOR | 7000 | Call the Floating Normalize Subroutine. |
| FNOP | 7010 | No operation. |
| FSKP | 7020 | Floating Skip: Skip the next sequential pseudo-instruction. |

**Table 8-2 (Cont.) Floating-Point Pseudo-Instructions**

| Mnemonic | Octal | Operation |
|---|---|---|
| FSNA | 7040 | Floating Skip on Non-Zero Accumulator: Skip the next sequential pseudo-instruction if the content of the FAC is not zero. |
| FSZA | 7050 | Floating Skip on Zero Accumulator: Skip the next sequential pseudo-instruction if the content of the FAC is zero. |
| FSPA | 7100 | Floating Skip on Positive Accumulator: Skip the next sequential pseudo-instruction if the content of the FAC is positive. |
| FSMA | 7110 | Floating Skip on Minus Accumulator: Skip the next sequential pseudo-instruction if the content of the FAC is less than zero. |
| FCDF | 70n1 | Change Floating Data Field: Obtain the operand for all subsequent indirectly addressed, floating-point, memory reference pseudo-instructions from memory field n. |
| | 71n1 | Unused. |
| FSW0 | 70n2 or 71n2 | Restore the normal order of all FDIV and FSUB operations until the next FSW1 pseudo-instruction is executed. |
| FSW1 | 70n3 or 71n3 | Reverse the order of all FDIV and FSUB operations until either the next FSW0 pseudo-instruction is executed, the interpreter is re-entered by the next effective JMS I 7 instruction, or the Floating Input Subroutine is entered. |
| FHLT | 70n4 or 71n4 | Floating Halt: Halt and display the content of the floating PC in the hardware accumulator. |
| | 70n5 to 70n7 | Unused. |
| | 71n5 to 71n7 | Unused. |
| FJMS | 7200 to 7777 | Floating Jump to Subroutine: Performs the same function as a standard JMS memory reference instruction, except that it is not possible to FJMS to a location on page zero directly. |

Table 8-2 lists all of the floating-point pseudo-instructions, their mnemonics and the operations they perform. The memory reference pseudo-instructions (octal codes 1nnn to 6nnn inclusive) are interpreted as calls to floating-point subroutines which require arguments. All of these functions are also available to the user in single-instruction mode. The FJMP and FJMS pseudo-instructions perform operations which are not possible in single-instruction mode, but they are essentially equivalent to the standard PDP-8 JMP and JMS instructions. Note, however, that no directly addressed FJMP or FJMS pseudo-instruction may reference a location on page 0. This restriction allows octal codes 0000 to 0177 and 7000 to 7177 to be interpreted as extended pseudo-instructions.

The pseudo-instructions corresponding to octal codes 0001 to 0017 generate calls to subroutines which are available to the user in single-instruction mode. 0000 is the FEXT (leave interpreter) operation, and 0020 to 0177 are FISZ operations (see Table 8-2). Aside from the FNOR operation, the pseudo-instructions corresponding to octal codes 7000 to 7177 are *not* available in single-instruction mode. The octal codes for some of these pseudo-instructions may have either a 1 or a 0 in bit position 5; this is because the interpreter does not decode bit 5 of the designated pseudo-instructions. Several possible octal codes do not have assigned pseudo-instructions (e.g. 71n1). These codes are unused, and should not be supplied as input to the interpreter.

```
*200
        KCC             /INITIALIZE KEYBOARD
        TLS             /AND TELEPRINTER
        JMS I 7         /ENTER INTERPRETER
        FIN             /GET NUMBER FROM TELETYPE
        FMPY TEN        /MULTIPLY BY 10.0
        FPUT STORE      /STORE RESULT
        FEXT            /LEAVE INTERPRETER
        HLT             /AND HALT
TEN,    4               /EXPONENT
        2400            /HIGH-ORDER MANTISSA
        0000            /LOW-ORDER MANTISSA
STORE,  0               /3-WORD FLOATING-POINT
        0               /STORAGE LOCATION
        0
$
```

Figure 8-4 Using the 23-bit FPP in Interpretive Mode

Using the 27-bit FPP, the constant TEN would appear as follows:

```
TEN,    2045        /EXPONENT
        0000        /HIGH-ORDER MANTISSA
        0000        /LOW-ORDER MANTISSA
```

The program example of Figure 8-4 performs the same operations as the example of Figure 8-3, however this program has been coded to execute in interpretive mode. Note that this program requires less core storage than the equivalent single-instruction mode version, however the execution time required in interpretive mode will be considerably longer.

**Floating Input and Output**

The FIN pseudo-instruction calls the Floating Input Subroutine to accept one decimal floating-point number from the Teletype keyboard, convert this input to a binary floating-point number, normalize and truncate the number if necessary, and load the number into the FAC.[1] The input routine is normally called in interpretive mode using the FIN command. The input routine may also be called by an effective JMS to the start of the routine (see Table 8-1 for exact memory location). Input is terminated when the routine recognizes any typed character which could not be part of the input. For example, the conversion of "12.0." would be terminated upon receipt of the second ".". The characters + and − will not be recognized as input terminators and should not be used as such. The following numbers, all terminated by carriage returns, are examples of legal input. They are all equivalent.

```
 726.7
.7267E3
.7267E+03
+7267E-1
```

The Floating Input Subroutine echoes each character as it is typed, including the terminator. Upon completion of any floating input operation, core memory location 0052 of the memory field

[1] Programs using floating input should begin with a KCC instruction to initialize the Teletype keyboard and a TLS instruction to initialize the printer.

in which the FPP resides contains zero if the input was invalid, and a non-zero value if the input was valid. Core location 0053 in this field contains the ASCII code for the terminating character last received. Core location 0054 of this field may be set by the user. A value of zero loaded into this location causes the input routine to echo only a carriage return whenever a carriage return is typed as an input terminator. Any non-zero value loaded into location 0054 causes the input routine to echo a carriage return *and* a line feed whenever a carriage return is received. Location 0054 is originally loaded with $7777_8$.

Using the 23-bit FPP's, if the example illustrated in Figure 8-5 is started at location 0200 and the user types "0X1.0Y" at the Teletype, the program will halt at location 0210 with storage locations A and B containing:

```
A,      0000
        0000
        0000
B,      0001
        2000
        0000
```

while location 0053 will contain $0331_8$, the ASCII code for the second terminator.

Using the 27-bit FPP, storage locations A and B contain:

```
A,      0           /ZERO
        0
        0
B,      2014        /ONE
        0
        0
```

and register 0053 will contain 0331 — the second terminator.

**NOTE**

Since the input routine calls the floating-point interpreter, after input, FSWITCH is set to 0 (FSW0) even if it was set to 1 when input was called (see the section, Floating Switch).

The input routine recognizes RUBOUT as a special character which is not echoed. If a RUBOUT is typed during input, all characters received since the last input terminator are ignored. For example, typing:

276(RUBOUT) 1T

to the input routine has the same effect as typing:

1T

Input will be terminated with the binary floating-point equivalent if decimal 1 in the FAC. The current input element must be retyped from the beginning.

```
*200
        KCC             /INITIALIZE KEYBOARD
        TLS             /AND TELEPRINTER
        JMS I 7         /ENTER INTERPRETER
        FIN             /INPUT A NUMBER
        FPUT A          /STORE IN LOCATION A
        FIN             /ACCEPT ANOTHER NUMBER
        FPUT B          /STORE IN LOCATION B
        FEXT            /EXIT INTERPRETER
        HLT             /AND HALT
A,      0
        0
        0
B,      0
        0
        0
$
```

Figure 8-5 Floating Input Routine

The FOUT pseudo-instruction calls the Floating Output Subroutine to print the contents of the FAC on the Teletype.[2] The FPP maintains four core memory locations on page 0 of the memory field in which it resides. These locations may be set by the user to determine the format for all floating-point output.

Loading any non-zero value into location 0056 causes all output to be printed in FORTRAN Fa.b format, where a is the content

[2] Programs using floating output should begin with a TLS instruction to initialize the Teletype printer.

of core location 0057 and b is the content of location 0060. As with FORTRAN, field overflow causes the field to be filled with asterisks, while field underflow causes the output to be right justified.

If location 0056 contains zero, the FPP loads $0016_8$ into location 0057 and $0006_8$ ($0007_8$ for the 27-bit FPP) into location 0060, then prints all output in FORTRAN E14.6 format (E14.7 for 27-bit FPP).

In either case, each element of output will be followed by a carriage return and line feed if the content of location 0055 is not zero. Loading a zero into location 0055 suppresses the terminating carriage return and line feed.

Upon loading, the FPP initializes these four core locations for E14.6 format (E14.7 in 27-bit FPP), with each element terminated by a carriage return and line feed.

**NOTE**

The output routine destroys the contents of the FAC. If the number to be typed is needed for further calculation, it should be saved prior to calling the output routine.

OVERFLOW AND UNDERFLOW

Under the 27-bit FPP only, overflow and underflow on input are treated like exponent overflow and underflow (see section on Error traps). Typing a number like:

```
100000E+61
```

will result in exponent overflow, and the error trap will be taken if the user has set it. The capacity of the input routine is approximately $10^{-38} < X < 10^{38}$. If more than 8 significant digits are input, the result will be truncated.

**Use of FISZ and Auto-Indexing**

Core memory locations 0010 through 0017 may be used for auto-indexing in interpretive mode. If one of these locations is referenced indirectly in interpretive mode, the contents of the location will be incremented by three before it is used for effective address generation.

The Floating ISZ (FISZ) operation is only available in interpretive mode. A FISZ pseudo-instruction must be directly addressed, and may only reference a page 0 location greater than 0017. A hardware ISZ is performed on the referenced page 0 location, and the next pseudo-instruction is skipped if the content of the location becomes zero. If the content of the referenced location does not become zero, the next sequential pseudo-instruction is executed.

The program example of Figure 8-6 uses auto-indexing in interpretive mode to pick up 20 (octal) floating-point numbers from a buffer in core and calculate the sine of each. The sines of the numbers are stored in a separate buffer area and are also printed on the Teletype. After each iteration, an FISZ is performed on a counter and the program loops back until the counter becomes zero, at which time the program exits the interpreter and halts. (Assume that the floating-point package and the user's program are in the same data field.)

```
BUF1=400                    /INPUT BUFFER BEGINS AT 0400
BUF2=600                    /OUTPUT BUFFER BEGINS AT 0600
*16
R16,    0                   /INPUT BUFFER POINTER
R17,    0                   /OUTPUT BUFFER POINTER
R20,    0                   /DATA ELEMENT COUNTER
*200
        KCC
        TLS                 /INITIALIZE TELEPRINTER
        JMS I 7             /ENTER INTERPRETER
        FGET INDXR          /INITIALIZE THE THREE
        FPUT R16            /AUTO-INDEX REGISTERS
LOOP,   FGET I R16          /GET NUMBER FROM INPUT BUFFER
        FSIN                /TAKE ITS SINE
        FPUT I R17          /PUT RESULT IN OUTPUT BUFFER
        FOUT                /PRINT RESULT
        FISZ R20            /DONE ALL NUMBERS?
        FJMP LOOP           /NO: LOOP BACK
        FEXT                /YES: EXIT INTERPRETER
        HLT                 /AND HALT
INDXR,  BUF1-3              /INITIAL VALUE OF R16
        BUF2-3              /INITIAL VALUE OF R17
        -20                 /INITIAL VALUE OF R20
$
```

Figure 8-6 Use of FISZ and Auto-Indexing

**User Subroutines**

Users who require special floating-point functions may code the functions as assembly language subroutines and call them through the interpreter in the same manner as the extended functions are called. Up to three such subroutines may be inserted in place of the three FNOPs having octal codes 0015, 0016 and 0017. This is accomplished by assigning a mnemonic to the user function, equating the assigned mnemonic to the octal code of the FNOP to be deleted, and inserting the entry address of the user subroutines into core location 7246 (for octal code 0015), 7247 (for octal code 0016) or 7250 (for octal code 0017) of the memory field in which the FPP and user subroutine reside.

User subroutines called through the interpreter can themselves call the interpreter and use all interpreter functions except calling another user function or extended function through op code 0. The extended functions could be called, however, using single-instruction mode. All user subroutines must be in the same memory field as the floating-point package. They may enter the interpreter and change the floating data field. They can even change the hardware data field, but when one user subroutine returns to the interpreter, the floating data field, hardware data field, and floating instruction field are all restored to the value they had prior to calling the user subroutine.

The program example of Figure 8-7 contains a user subroutine called through the interpreter. This subroutine has been assigned the mnemonic FUSR and octal code 0015. If the FPP is loaded into the same memory field, the program will accept floating-point numbers from the Teletype, add all such input elements greater than 0.5 to a running sum, and print the sine of the cumulative total after each input element is received.

```
          FUSR=0015          /ASSIGN MNEMONIC AND CODE
          FIELD 0
*200
START,    KCC                /INITIALIZE KEYBOARD
          TLS                /AND TELEPRINTER
          TAD KUSR           /INSERT USER SUBROUTINE ENTRY
          DCA I INTABL       /ADDRESS IN INTERPRETER TABLE
          JMS I 7            /ENTER INTERPRETER
          FIN                /ACCEPT NUMBER FROM KEYBOARD
          FUSR               /CALL USER SUBROUTINE
          FOUT               /PRINT RESULT (SINE OF SUM)
          FJMP .-3           /ACCEPT NEW INPUT


KUSR,     USUB               /ENTRY ADDRESS FOR USER
                             /SUBROUTINE TO BE ENTERED
                             /IN INTERPRETER TABLE
INTABL,   7246               /POINTER TO INTERPRETER
                             /TABLE ENTRY CORRESPONDING
                             /TO OCTAL CODE 0015
USUB,     0                  /ENTER USER SUBROUTINE
          JMS I 7            /ENTER THE INTERPRETER
          FPUT TEM           /STORE INPUT
          FCDF 10            /CHANGE TO FLOATING
                             /DATA FIELD 1
          FGET I COMPR       /LOAD 0.5 INTO FAC
          FSUB TEM           /SUBTRACT INPUT
          FSMA               /IS INPUT GREATER THAN 0.5?
          FJMP .+4           /NO: DON'T ADD TO SUM
          FGET SUM           /YES: GET CUMULATIVE SUM
          FADD TEM           /ADD IN LATEST INPUT
          FPUT SUM           /STORE NEW SUM
          FGET SUM           /LOAD SUM INTO FAC AND
          FEXT               /LEAVE INTERPRETER
          JMS I PSIN         /TAKE SINE OF SUM
          JMP I USUB         /RETURN TO MAINLINE
PSIN,     5000               /POINTER TO SINE ROUTINE
COMPR,    PT5                /POINTER TO FIELD 1 CONSTANT
SUM,      0                  /FLOATING-POINT STORAGE
          0
          0
TEM,      0                  /FLOATING-POINT TEMPORARY
          0
          0
          FIELD 1
*200
PT5,      0                  /FIELD 1 CONSTANT = 0.5
          2000
          0
$
```

Figure 8-7 Coding a User Subroutine (23-bit FPP)

If using the 27-bit FPP, the field 1 constant is stored as:

```
        FIELD 1
*200
PT5,    2004            /CONSTANT = .5
        0
        0
```

**Floating Skips**

The interpreter maintains a core memory location designated as the floating program counter (floating PC) which is originally set to the address of the location following the JMS I 7 which called the interpreter, and thereafter updated by the interpreter whenever a floating-point pseudo-instruction is executed. The floating PC may be incremented by the floating skip pseudo-instructions, which are functionally equivalent to normal PDP-8 skip microinstructions except that floating skips reflect the status of the FAC rather than the hardware accumulator. Floating skip pseudo-instructions may be microprogrammed in the same manner as group 2 operate microinstructions. All floating skips assume that the content of the FAC is normalized. There is no provision for clearing the FAC when skipping on condition.

The example of Figure 8-8 uses floating skip pseudo-instructions to accept floating-point numbers from the Teletype until it receives a number with an absolute value greater than 10.0.

```
*200
        KCC             /INITIALIZE KEYBOARD
        TLS             /AND TELEPRINTER
        JMS I 7         /ENTER INTERPRETER
INPUT,  FIN             /ACCEPT INPUT FROM KEYBOARD
        FSNA            /IS THE INPUT NUMBER 0.0?
        FJMP .-2        /YES: GET NEW INPUT
        FPUT TEM        /NO: STORE TEMPORARILY
        FSPA            /IS INPUT POSITIVE?
        FNEG            /NO: TAKE ABSOLUTE VALUE
        FSUB TEN        /YES: SUBTRACT 10.0
        FSPA FSNA       /IS INPUT GREATER THAN 10.0?
        FJMP INPUT      /NO: GET NEW INPUT
        FGET TEM        /YES: USE THIS INPUT
        FEXT            /LEAVE INTERPRETER
        HLT             /HALT WITH INPUT IN FAC
TEN,    4               /CONSTANT = 10.0
        2400
        0000
TEM,    0;0;0
$
```

Figure 8-8 Use of Floating Skips (23-bit FPP)

Using the 27-bit FPP, the constant ten is stored as follows:

```
TEN,    2045          /CONSTANT = 10.0
        0000
        0000
```

**Floating Data Field**

The floating CDF (FCDF) pseudo-instruction is used to change the data field from which the operand of an indirectly addressed, floating-point, memory reference pseudo-instruction is obtained. The program example shown in Figure 8-9 illustrates the use of the FCDF pseudo-instruction.

```
        FIELD 0             /LOAD FPP INTO FIELD 1 AND
*200                        /USER PROGRAM INTO FIELD 0
        KCC
        TLS                 /INITIALIZE PRINTER
        CDF 00              /SET DATA FIELD = 0
        CIF 10              /INSTRUCTION FIELD = 1
        JMS I K7400         /ENTER INTERPRETER FROM
        FGET I PF8PT1       /FIELD 0 THEN
        FCDF 10             /SET DATA FIELD = 1
        FPUT I K56          /SET OUTPUT REGISTER
        FGET TEN            /GET NUMBER FROM FIELD 1
        FOUT                /AND PRINT IT
        FEXT                /LEAVE INTERPRETER
        HLT                 /AND HALT
K7400,  7400                /POINTER TO INTERPRETER
TEN,    4;2400;0            /CONSTANT = 10.0
K56,    56                  /POINTER TO OUTPUT REGISTER
PF8PT1, F8PT1               /POINTER TO FORMAT SPEC
*400
F8PT1,  1                   /CONSTANTS TO
        10                  /SET OUTPUT TO
        1                   /F8.1 FORMAT
$
```

Figure 8-9 Use of the FCDF Pseudo-Instruction (23-bit FPP)

Using the 27-bit FPP, the constant ten is stored as:

```
TEN,    2045          /CONSTANT = 10.0
        0000
        0000
```

If the FPP is loaded into memory field 1, this program will enter the interpreter from memory field 0, load the floating output regis-

ter with a format specification stored in field 0, and then print a floating-point number which is also stored in field 0.

**Floating Switch**

The floating switch instructions (FSW0 and FSW1) regulate the operation of the FDIV and FSUB pseudo-instructions. Using floating switch 0, FSUB and FDIV are interpreted normally. Following any occurrence of an FSW1 pseudo-instruction in interpretive mode, the order of all FDIV and FSUB operations is reversed. That is, every FDIV operation will divide the content of the FAC into the floating-point argument supplied (rather than vice versa) and every FSUB operation will subtract the content of the FAC from the floating-point operand. The result of these operations will be stored in the FAC in either case. After each occurrence of an FSW1 pseudo-instruction, the order of FDIV and FSUB operations will continue to be reversed until either:

1. The next sequential FSW0 pseudo-instruction is executed.
2. The interpreter is re-entered by the next effective JMS I 7 instruction.
3. A floating input operation is performed in either mode.

The floating switch is normally set to 0.

Certain mathematical calculations may be facilitated by using the FSW1 pseudo-instruction to reverse the order of FDIV and FSUB operations. For example, Figures 8-10 and 8-11 contain two program segments, both of which calculate the value of the expression:

$$\frac{\pi}{2} - X\left(B_0 + \cfrac{A_1}{X^2 + B_1 + \cfrac{A_2}{X^2 + B_2 + \cfrac{A_3}{X^2 + B_3}}}\right)$$

Both program segments execute in interpretive mode, however the first program segment uses the FSW1 pseudo-instruction to reverse the order of certain FDIV and FSUB operations, while the second program segment does not.

```
*200
        JMS I 7         /ENTER INTERPRETER
        FGET X          /GET X
        FSQU            /SQUARE IT
        FPUT XSQR       /STORE TEMPORARILY
        FADD B3         /FORM X↑2+B3
        FSW1            /USE INVERSE DIV AND SUB
        FDIV A3         /FORM A3/(X↑2+B3)
        FADD B2         /FORM B2+A3/(X↑2+B3)
        FADD XSQR       /(X↑2+B2+A3/(X↑2+B3))
        FDIV A2         /A2/(X↑2+B2+A3/(X↑2+B3))
        FADD B1
        FADD XSQR       /X↑2+B1+A2/(X↑2+B2+A3/(X↑2+B3))
        FDIV A1
        FADD B0
        FMPY X
        FSUB PIOV2      /PI/2-X(B0+A1/(X↑2+B1+A2/(...)))
        FPUT ANS
        FEXT            /EXIT INTERPRETER
        HLT
```

Figure 8-10 Use of the FSW1 Pseudo-Instruction

### Floating Halt

The floating halt pseudo-instruction is used mainly for debugging floating-point code. When a floating halt is detected in interpretive mode, the interpreter will halt with the address of the next floating-point instruction displayed in the hardware accumulator. The user can continue execution by pressing the CONT switch on the programmer's console. Normally, the floating halt pseudo-instructions would be removed or replaced by FNOPs once the program has been debugged.

## SINGLE INSTRUCTION MODE VERSUS INTERPRETIVE MODE

The relative advantages and disadvantages of each of the two modes of FPP operation are summarized in Table 8-3. For most applications, interpretive mode is more convenient than single-instruction mode and more economical in terms of core storage requirements. However, the user who is especially speed-conscious may employ single-instruction mode to eliminate interpreter overhead and reduce execution time.

```
*200
        JMS I 7         /ENTER INTERPRETER
        FGET X          /GET X
        FSQU            /SQUARE IT
        FPUT XSQR       /STORE X↑2
        FADD B3         /FORM X↑2+B3
        FPUT TEM        /STORE TEMPORARILY
        FGET A3         /GET A3
        FDIV TEM        /DIVIDE BY X↑2+B3
        FADD B2
        FADD XSQR       /X↑2+B2+A3/(X↑2+B3)
        FPUT TEM        /STORE TEMPORARILY
        FGET A2
        FDIV TEM
        FADD B1
        FADD XSQR       /X↑2+B1+A2/(X↑2+B2+A3/(X↑2+B3))
        FPUT TEM        /STORE TEMPORARILY
        FGET A1
        FDIV TEM
        FADD B0
        FMPY X          /X(B0+A1/(X↑2+B1+A2/(...)))
        FNEG            /NEGATE AND ADD RATHER
        FADD PIOV2      /THAN INVERSE SUBTRACT
        FPUT ANS
        FEXT
        HLT

TEM,    0               /EXTRA TEMPORARY LOCATION
        0               /NEEDED IF FLOATING SWITCH
        0               /NOT USED
```

Figure 8-11 Program Segment of Figure 8-10 Without FSW1 Pseudo-Instruction

As an example of the trade-offs to be considered when choosing between interpretive mode and single-instruction mode, the interpretive mode program segment of Figures 8-10 and 8-11 is recoded in single-instruction mode and presented in Figure 8-12 (pointers to FPP routines are given in Table 8-1). The single instruction mode version requires 18 (decimal) additional storage locations, but it is approximately 800 microseconds faster than the interpretive mode version.

**Table 8-3 Single-Instruction Mode Versus Interpretive Mode**

| | Single-Instruction | Interpretive Mode |
|---|---|---|
| Instruction Set: | FISZ, FJMP, FJMS and floating skips not available. | Full instruction set. |
| Execution Time: | Generally shorter. (50 microseconds or 20-25% shorter for the EAE FPP.) | Generally longer. |
| Core Requirements: | Generally up to 33% higher (requires 2 or 3 words per operation). | Generally lower (requires 1 or 2 words per operation). |
| Memory Field Limitations: | Instructions must reside in the field which contains the FPP. | All of extended memory is available for program and data storage. |
| Other Advantages: | User programs may overlay the unavailable floating-point functions and the interpreter. | |

## ERROR TRAPS

The following error conditions are detected by the FPP:

1. Attempt to FIX a number whose absolute value is greater than 2047.
2. Attempt to divide by zero.
3. Illegal function argument (log(X) where X is not a positive number).
4. Exponent Overflow (27-bit FPP only).
5. Exponent Underflow (27-bit FPP only).

Exponent overflow and underflow errors are detected only by the 27-bit FPP. These conditions occur whenever a calculation results in a number whose exponent has an absolute value greater than $128_{10}$. The result of such a calculation is meaningless. In the 23-bit FPP's, underflow or overflow occurs when a calculation results in a number whose exponent has an absolute value greater than $615_{10}$.

```
*200
        JMS I FGETP     /GET X
        X
        JMS I FSQP      /SQUARE IT
        JMS I FPUTP     /STORE X↑2
        XSQR
        JMS I FADDP     /FORM X↑2+B3
        B3
        JMS I FDIV1P    /INVERSE DIVIDE TO GET
        A3              /A3/(X↑2+B3)
        JMS I FADDP     /FORM B2+A3/(X↑2+B3)
        B2
        JMS I FADDP     /X↑2+B2+A3/(X↑2+B3)
        XSQR
        JMS I FDIV1P    /A2/(X↑2+B2+A3/(X↑2+B3))
        A2
        JMS I FADDP     /ADD B1
        B1
        JMS I FADDP     /ADD X↑2
        XSQR
        JMS I FDIV1P    /A1/(X↑2+B1+A2/(...))
        A1
        JMS I FADDP     /ADD B0
        B0
        JMS I FMPYP     /X(B0+A1/(X↑2+B1+A2/(...)))
        X
        JMS I FSUB1P-   /CALL INVERSE SUBTRACT TO GET
        PIOV2           /PI/2-X(B0+A1/(...))
        JMS I FPUTP
        ANS
        HLT             /POINTERS TO FPP ROUTINES:
FGETP,  7306            /FLOATING GET
FSQP,   7564            /FLOATING SQUARE
FPUTP,  7322            /FLOATING PUT
FADDP,  7000            /FLOATING ADD
FDIV1P, 6412            /FLOATING INVERSE DIVIDE
FMPYP,  6600            /FLOATING MULTIPLY
FSUB1P, 6400            /FLOATING INVERSE SUBTRACT
```

Figure 8-12 Program Segment of Figure 8-10 in Single Instruction Mode

Error 3 leaves the FAC unchanged, while errors 1 and 2 set the FAC to zero. Error 4 in the 27-bit FPP sets the FAC to a very large number. Error 5 in the 27-bit FPP sets the FAC to 0. The contents of the hardware accumulator and multiplier quotient register are unpredictable when any of these errors occurs. The user may set the interpreter to jump to a specific location in his program upon detection of one of these errors. To do this using the

23-bit FPP, core locations 7574, 7575 and 7576 should be loaded with the address (in the same memory field) to which control should be transferred upon detection of error 1, 2, or 3, respectively. Using the 27-bit FPP, core locations 7574, 7575, 7576, 7573, and 7577 should be loaded with the address for transfer of control upon detection of error 1, 2, 3, 4, or 5 respectively. Normally, these locations would be set to point to user error message routines, and the user program would abort upon detection of one of these errors.

## EXTENDED FUNCTION ALGORITHMS

The algorithms to approximate the extended functions use either Chebyshev optimized Taylor series expansions or continued fraction polynomials to calculate the value of a function over some small range. These algorithms provide fairly uniform accuracy over the specified range and, as opposed to converging series, they consist of a definite number of computations. Various extended function algorithms are described individually in the following paragraphs.

### SIN(X)

The argument (the FAC in radian measure) is first reduced to the interval:

$$-\frac{\pi}{2} < X \leqslant \frac{\pi}{2}$$

using the proper identities, and the quadrant of the original argument is determined. SIN(X) is then calculated as a function of the modified argument Y = G(F) as follows:

| Quadrant | Y= | Using the Identity: |
|---|---|---|
| 0 | F | SIN(Y) = SIN(Y) |
| 1 | 1−F | SIN(Y) = SIN(π−Y) |
| 2 | −F | SIN(Y) = SIN(−(Y−π) ) |
| 3 | F−1 | SIN(Y) = SIN(−(2π−Y) ) |

where F is the fractional portion of $(\pi/2)X$.

Using either of the 23-bit FPP s, SIN(Y) is then calculated

over the range $(-\pi/2, \pi/2)$ using the Chebychev optimized Taylor series expansion:

$$\text{SIN}\frac{\pi}{2}Y = A_1Y+A_3Y^3+A_5Y^5+A_7Y^7$$

where:

$$\begin{aligned} A_1 &= 1.570949 \\ A_3 &= -.64592098 \\ A_5 &= .07948766 \\ A_7 &= -.004362476 \end{aligned}$$

Using the 27-bit FPP, SIN(Y) is calculated:

$$\text{SIN}\frac{\pi}{2}Y = A_1Y+A_3Y^3+A_5Y^5+A_7Y^7+A_9Y^9$$

where:

$$\begin{aligned} A_1 &= 1.57079633 \\ A_3 &= -.645963711 \\ A_5 &= .079689679 \\ A_7 &= -.00467376557 \\ A_9 &= .00015148419 \end{aligned}$$

COS(X)

The function COS(X) is calculated using the SIN function on the basis of the following identity:

$$\text{COS}(X)=\text{SIN}((\pi/2)+X)$$

EXP(X)

The algorithm for the calculation of EXP(X) uses a continued fraction polynomial to calculate $e^x$ as:

$$e^x = 2^{\log_2 e^x} = 2^{X\log_2 e}$$

If: $n$ = Integer part of $X\log_2 e$
$r$ = Fractional part of $X\log_2 e$

then: $$e^x = 2^n 2^r$$

where: $$|r|<1$$

Under the 23-bit FPP, the algorithm calculates $2^r$ as follows:

$$2^r = \left(1 + \frac{2y}{A_0 - y + \frac{A_1}{B_1 + y^2}}\right)^2$$

where: $y = \frac{r\ 1n2}{2}$

$\log_2 e = 1.442695$

$\frac{1n2}{2} = .34657359$

$A_0 = 12.015017$

$A_1 = -601.80427$

$B_1 = 60.090191$

Under the 27-bit FPP, $2^r$ is calculated using the values:

$$y = \frac{r1n2}{2}$$

$\log_2 e = 1.442695$

$\frac{1n2}{2} = .3465735903$

$A_0 = 12.01501675$

$A_1 = -601.804267$

$B_1 = 60.0901907$

ARCTANGENT (X)

The algorithm for the calculation of $TAN^{-1}(X)$ also uses a continued fraction polynomial to calculate the $TAN^{-1}(X)$ for $0<X<1$.

The argument is reduced to the range $0<X<1$ using the identities:

If: $X<0$, $TAN^{-1}(-X) = -TAN^{-1}(X)$

$0<X\leqslant 1$, $TAN^{-1}(X) = TAN^{-1}(X)$

$1<X$, $TAN^{-1}(X) = (\pi/2) - TAN^{-1}(1/X)$

Once X has been reduced, $TAN^{-1}(X)$ is calculated as:

$$TAN^{-1}(X) = X \left( B_0 + \cfrac{A_1}{X^2 + B_1 + \cfrac{A_2}{X^2 + B_2 + \cfrac{A_3}{X^2 + B_3}}} \right)$$

where (under the 23-bit FPP): $B_0 = .17465544$
$A_1 = 3.7092563$
$B_1 = 6.762139$
$A_2 = -7.10676$
$B_2 = 3.3163354$
$A_3 = -.26476862$
$B_3 = 1.44863154$

while under the 27-bit FPP: $B_0 = .174655439$
$A_1 = 3.70925626$
$B_1 = 6.7621392$
$A_2 = -7.10676005$
$B_2 = 3.31633543$
$A_3 = -.26476862$
$B_3 = 1.44863154$

## LOG(X)

Log(X) is calculated by using a Chebyshev optimized Taylor series to approximate the $\log_2$ of the mantissa of X.

**If** F = Mantissa of X
I = Exponent of X

**then** $\log_e(X) = [I + \log_2(F)] \log_e(2)$

The $\log_2(F)$ is approximated as follows:

$$\log_2(F) = C_1Z + C_3Z^3 + C_5Z^5 - 1/2$$

Under the 23-bit FPP:

$Z = (F - \sqrt{.5})/(F + \sqrt{.5})$
$C_1 = 2.8853913$
$C_3 = .9614706$
$C_5 = .59897865$
$\sqrt{.5} = .7071068$
$\log_e 2 = .6931472$

while under the 27-bit FPP:

$$
\begin{aligned}
Z &= (F - \sqrt{.5})/(F + \sqrt{.5}) \\
C_1 &= 2.88539129 \\
C_3 &= .961470632 \\
C_5 &= .59897865 \\
\sqrt{.5} &= .707106781 \\
\log_e 2 &= .69314718
\end{aligned}
$$

The ranges of the extended functions are as follows:

| | |
|---|---|
| SIN, COS | $-2047<X<2046$ |
| EXP | $-1415<X<1415$ |
| ATAN | all X |
| LOG | all $X>0$ |

The SQUARE, SQUARE ROOT, FIX, and FLOAT routines are explained in Table 8-1.

**Execution Time for EAE Floating-Point Operations**

Instruction times are for single-instruction mode calls. Add 50 μsec. for interpretive mode, or 71 μsec. for interpretive mode with indirect addressing.

| Operation | Typical Time |
|---|---|
| FADD | 160 μsec. |
| FSUB | 180 μsec. |
| FMPY | 200 μsec. |
| FDIV | 160 μsec. or 190 μsec. |
| FGET | 55 μsec. |
| FPUT | 35 μsec. |
| FSUB (with FSW1) | 215 μsec. |
| FDIV (with FSW1) | 210 μsec. or 240 μsec. |

**Execution Time for EAE Extended Functions**

| Function | Range | Average Execution Time |
|---|---|---|
| SIN | $-10^{615} \leqslant X < 0$ | 2.30 msec. |
| | $0 \leqslant X < 10^{615}$ | 2.25 msec. |
| COS | $-10^{615} < X < 0$ | 2.45 msec. |
| | $0 \leqslant X < 10^{615}$ | 2.40 msec. |
| ATAN | $-10^{615} < X < -1$ | 2.78 msec. |
| | $-1 \leqslant X < 0$ | 2.33 msec. |
| | $0 \leqslant X \leqslant 1$ | 2.27 msec. |
| | $1 < X < 10^{615}$ | 2.73 msec. |
| EXP | $-1415 < X < 1415$ | 2.33 msec. |
| LOG | $0 < X < 10^{615}$ | 2.43 msec. |
| SQROOT | $-10^{615} < X < 10^{615}$ | 1.43 msec. |
| SQUARE | $-10^{300} < X < 10^{300}$ | 200 $\mu$sec. |

**Execution Time for Non-EAE Floating-Point Operations**

Instruction times are for single-instruction mode calls on the PDP-8/E. All 55 $\mu$sec. for interpretive mode, or 72.8 $\mu$sec. for interpretive mode with indirect addressing.

| Instruction | Typical Time |
|---|---|
| FADD (Operands have same order of magnitude.) | 180 $\mu$sec. plus normalization time. |
| FADD (Operands differ by 3 orders of magnitude.) | 270 $\mu$sec. plus normalization time. |
| FADD (Operands differ by 6 orders of magnitude.) | 360 $\mu$sec. plus normalization time. |
| FADD (Operands differ by 12 orders of magnitude.) | 530 $\mu$sec. plus normalization time. |
| FSUB | FADD time plus 25 $\mu$sec. |
| FMPY (Positive operands.) | 990 $\mu$sec. average. |
| FMPY (At least one negative operand.) | 1025 $\mu$sec. average. |

| | |
|---|---|
| FDIV (Both operands positive.) | 1077 μsec. or 1113 μsec. |
| FDIV (At least one negative operand.) | 1118 μsec. or 1153 μsec. |
| FNOR | 21+41.6N μsec. where N shifts are required. |
| FGET | 57.6 μsec. |
| FPUT | 39.6 μsec. |
| FSUB with FSW1 | Same as FSUB. |
| FDIV with FSW1 | Add 54.0 μsec. to FDIV time. |
| FSQU | Same as FMPY. |
| FSQR | 1965 μsec. average. |

**Accuracy of Extended Functions**

| Function | Approximate Range | Accuracy (no. of significant digits) 23-Bit | 27-Bit |
|---|---|---|---|
| SIN | $-\pi/2<X<\pi/2$ | 6 digits | 7 digits |
| | $-50<X<50$ | | 6½ digits |
| | $-200<X<200$ | 5 digits | 6 digits |
| COS | $-\pi/2<X<\pi/2$ | 6 digits | 6 digits |
| | $-30<X<30$ | | 6½ digits |
| | $-100<X<100$ | 5 digits | |
| | $-125<X<125$ | | 6 digits |
| EXP | $-89<X<88$ | | 6½ digits |
| | $-50<X<100$ | 5 digits | |
| | $-25<X<35$ | | 7 digits |
| | $-1<X<1$ | 6 digits | 7 digits |
| LOG | $0<X<1000$ | 6 digits | 7 digits |
| $TAN^{-1}$ | $-1<X<1$ | 6 digits | 7 digits |
| | all X | 6 digits | 7 digits |

Conditions determining function accuracy:

1. All functions are accurate to six digits (7 digits under 27-bit FPP) over their primary range. Primary ranges. are as follows:

| | |
|---|---|
| SIN, COS | $-\pi/2<X<=\pi/2$ |
| EXP | $-1<X<=1$ |
| LOG | $0<X<=1$ |
| $TAN^{-1}$ | $-1<X<=1$ |

2. The SIN and COS functions are accurate to six digits (7 digits under 27-bit FPP) for all X, but the answer loses significance as X diverges from zero.

3. Accuracy decreases as X becomes very large, or very close to critical points.

## CORE STORAGE MAPS

User programs may overlay portions of the FPP which are not used, such as I/O routines and extended functions. The following storage maps list core allocations for the Extended Arithmetic Element FPP, the non-EAE package, and the 27-bit FPP. Page 0 storage is the same for all packages.

EXTENDED ARITHMETIC ELEMENT
FLOATING-POINT PACKAGE

| Core location | Contents |
|---|---|
| 7400-7577 | Interpretive dispatch routines. |
| 7200-7377 | Argument fetch, FPUT, FGET, FNOR and extended functions calling sequence. |
| 7000-7177 | FADD, FSUB, FNEG and part of FDIV. |
| 6600-6777 | FMPY and FDIV. |
| 6400-6577 | Inverse Floating Subtract and Divide, FSQR and FHLT. |
| 5600-6377 | FIN and FOUT. |
| 5400-5577 | FIX, FLOT and constants for extended functions. |
| 5200-5377 | FATN and FLOG. |
| 5000-5177 | FSIN, FCOS, FEXP and utility routines. |

NON-EAE FLOATING-POINT PACKAGE

| Core location | Contents |
|---|---|
| 7400-7577 | Interpretive dispatch routines, FSQU, FJMP and FJMS. |
| 7200-7377 | Argument fetch, FPUT, FGET, FNOR, extended functions calling sequence and part of FDIV. |
| 7000-7177 | FADD, FSUB, FNEG, part of FDIV. |
| 6600-6777 | FMPY and part of FDIV. |
| 6400-6577 | Inverse Floating Subtract and Divide, and FSQR. |
| 5600-6377 | FIN and FOUT. |
| 5400-5577 | FIX, FLOT and constants for extended functions. |
| 5200-5377 | FATN and FLOG. |
| 5000-5177 | FSIN, FCOS, FEXP and utility routines. |
| 4600-4777 | Part of FMPY and FDIV, and interpreter routines. |

27-BIT FLOATING-POINT PACKAGE

| Core location | Contents |
|---|---|
| 7400-7577 | Interpretive Dispatch Routines; Floating Square, JMP, JMS. |
| 7200-7377 | FPUT, FGET; Normalize; Extended Functions Call Sequence; Floating Halt. |
| 7000-7177 | Floating Add, Subtract, Negate. |
| 6600-6777 | Floating Multiply, Divide. |
| 6400-6577 | Inverse Floating Subtract and Divide; Square Root. |
| 5600-6377 | Floating Input and Output Routines. |
| 5400-5577 | FIX; FLOAT; Constants for Extended Functions. |
| 5200-5377 | Arctangent; Log. |
| 5000-5177 | SIN; COS; Exponential; Utility Routines. |
| 4600-4777 | Parts of Floating Multiply, Divide; Interpreter Routines; Argument Fetch. |

PAGE ZERO STORAGE MAP (all FPPs)

| Core location | Contents |
|---|---|
| 0007 | Pointed to interpreter. |
| 0040-0042 | Floating-point temporary storage. |
| 0043 | Interpreter constant storage. |
| 0044-0046 | Floating accumulator (FAC). |
| 0047-0051 | Floating-point operand storage. |
| 0052 | Valid input switch. |
| 0053 | Last input terminator register. |
| 0054 | Line feed after carriage return switch. |
| 0055 | CR/LF after output switch. |
| 0056 | E/F output format switch. |
| 0057 | F format output field width specification. |
| 0060 | F format decimal digit specification. |
| 0061 | Internal pointer for interpreter. |
| 0062 | Reserved for future expansion. |

## SUMMARY OF FLOATING-POINT INSTRUCTIONS

### Memory Reference Instructions

| Instr | Code |
|---|---|
| FJMP | 0000 |
| FADD | 1000 |
| FSUB | 2000 |
| FMPY | 3000 |
| FDIV | 4000 |
| FGET | 5000 |
| FPUT | 6000 |
| FJMS | 7000 |

**Expanded Instructions**

| Instr | Code | |
|---|---|---|
| FISZ= | 0 | |
| FEXT= | 0 | |
| FSQU= | 1 | (Square) |
| FSQR= | 2 | (Square Root) |
| FSIN= | 3 | |
| FCOS= | 4 | |
| FATN= | 5 | |
| FEXP= | 6 | |
| FLOG= | 7 | |
| FNEG= | 10 | |
| FIN= | 11 | |
| FOUT= | 12 | |
| FFIX= | 13 | |
| FLOT= | 14 | |
| FNOP= | 15 | (Available to User) |
| FNOP= | 16 | (Available To User) |
| FNOP= | 17 | (Available To User) |
| FNOR= | 7000 | |
| FCDF= | 7001 | (Bits 6-8 New Fltg. Data Field) |
| FSW0= | 7002 | |
| FSW1= | 7003 | |
| FHLT= | 7004 | |
| FNOP= | 7005 | |
| FNOP= | 7006 | |
| FNOP= | 7007 | |
| FSMA= | 7110 | |
| FSZA= | 7050 | |
| FSPA= | 7100 | |
| FSNA= | 7040 | |
| FNOP= | 7010 | |
| FSKP= | 7020 | |

# appendices

# appendix a

# answers to selected exercises

## Chapter 1

### Answers to selected exercises on page 1-10

a. 2. 10010
4. 1100100
6. 1
8. 1110101
10. 111 111 111 010

b. 2. 5
4. 94
6. 31
8. 55
12. 10 111 011 110
14. 111 110 101
16. 1 110 001 011
18. 110 110 000 000
20. 11 110 101 111
10. 3641
12. 4087
14. 63
16. 4095

### Answers to selected exercises on page 1-14

a. 2. 6
4. 575
6. 40
8. 30

b. 2. 111 011 110
4. 1 000
6. 101 100 010 100
8. 1 001 000 001

c. 2. 40
4. 1104
6. 3

d. 2. 31
4. 512
6. 482
10. 7777
12. 7664
14. 255
16. 2372
10. 110 100
12. 111 111 110 101
14. 100 101 011 010
16. 1 010 100 011
8. 2500
10. 6005
12. 7777
8. 4095
10. 2431
12. 174

### Answers to selected exercises on page 1-26

a. 2. 110
4. 10 111 000
6. 1 100

b. *One's Complement*
2. 101 000 100 000
4. 111 111 111 111
6. 111 011 011 011
8. 011 111 111 111
10. 011 110 011 001
12. 000 000 000 000

c. 2. 101 000 101

d. 2. 110 110 010
8. 11 100
10. 10 001 101
12. 1 010 010 101

*Two's Complement*
101 000 100 001
000 000 000 000
111 011 011 100
100 000 000 000
011 110 011 010
000 000 000 001

4. 11 001 110 101
4. 1 010 011

e. 2. 111 010 001

f. 2. 100

g. 2. 70
4. 110
6. 1331
8. 3623

h. 2. 42
4. 7
6. 205
8. 1105

i. 2. 667
4. 2767
6. 25
8. 112

j. 2. 204
4. 433,254
6. 172,166

## Chapter 2

**Answers to selected exercises**

1. The locations listed in parts b, f, g, h, and i must be addressed indirectly. All others may be addressed directly.

2. a. Group 2 (SZL)
   b. MRI (AND)
   c. Group 1 (CLA CMA)
   d. Group 1 (NOP)
   e. MRI (JMS)
   f. Group 2 (SMA CLA)

3. Parts a, c, and e contain digits which can not be represented with binary numbers. Part b has too many digits to be represented by 12-bits. Part d is a legal instruction if a leading zero is assumed.

4. a. AND 0 — The logical AND of the AC with the contents of location 0 replaces the accumulator.
   c. ISZ Y — Increment the contents of location 100 on the current page and skip the next instruction if the contents become 0 after the incrementation.
   e. DCA I Y — Deposit and clear the accumulator indirectly into the location whose address is contained in location 100.
   g. TAD 30 — Two's complement add the contents of location 30 to the accumulator.
   i. JMP Y — Transfer program control to location 73 on the current page of memory.

5. a. CLA CMA CML
   c. SZA
   e. SPA SNA
   g. CLA SPA SNA SZL

6. Program:

```
7200
1205
1206
3207
7402
1537
2241
0000
```

After Execution

| Location | Content (octal) |
|---|---|
| AC | 0000 |
| 205 | 1537 |
| 206 | 2241 |
| 207 | 4000 |

7. a. 7360

   c. 7710

   e. illegal — One instruction may not be used to rotate once and rotate twice at the same time. On the PDP-8 and PDP-8/S it is also illegal to combine an increment and a rotate microinstruction, thus part d is legal on the PDP-8/I and PDP-8/L but it is illegal on the PDP-8 and PDP-8/S.

   g. illegal. One instruction may not include members of both skip groups.

   i. illegal — One instruction may not combine microinstructions from Group 1 and Group 2.

8. SZA
   SKP
   SNL
   Instruction to be skipped

9. Any testing of the accumulator is done before the OSR instruction is executed.

10. a.

| Location | Content | Octal |
|---|---|---|
| 200 | CLA | 7200 |
| 201 | TAD 210 | 1210 |
| 202 | TAD 211 | 1211 |
| 203 | DCA 212 | 3212 |
| 204 | HLT | 7402 |
| 210 | 0002 | 0002 |
| 211 | 0010 | 0010 |
| 212 | 0000 | 0000 |

b.

| Location | Content | Octal |
|---|---|---|
| 400 | CLA | 7200 |
| 401 | TAD 550 | 1350 |
| 402 | DCA 552 | 3352 |
| 403 | TAD 551 | 1351 |
| 404 | DCA 550 | 3350 |
| 405 | TAD 552 | 1352 |
| 406 | DCA 551 | 3351 |
| 407 | HLT | 7402 |

## Chapter 3

**Answers to selected exercises**

1.
```
/SUBROUTINE TO SUBTRACT TWO NUMBERS
*200
START,    CLA CLL
          TAD K1200
          JMS SUB
          1500
          HLT
*300
SUB,      0
          CIA
          TAD I SUB
          ISZ SUB
          JMP I SUB
K1200,    1200
$
```

2a.
```
/LOAD LOCATIONS 2000 TO 2777
*200
START,    CLA CLL
          TAD K2000
          DCA LOCPTR
          DCA COUNT
DEPOSIT,  TAD COUNT
          DCA I LOCPTR
          ISZ COUNT
          ISZ LOCPTR
          TAD LOCPTR
          TAD M3000
          SZA CLA
          JMP DEPOSIT
          HLT
COUNT,    0
K2000,    2000
LOCPTR,   0
M3000,    -3000
$
```

4.
```
/TRIPLE PRECISION ADD
*200
TRIADD,   CLA CLL
          TAD AL
          TAD BL
          DCA ANSL
          RAL
          TAD AM
          TAD BM
          DCA ANSM
```

```
                RAL
                TAD AH
                TAD BH
                DCA ANSH
                HLT
AH,             1211
AM,             0314
AL,             7125
BH,             0114
BM,             4157
BL,             0176
ANSH,           0
ANSM,           0
ANSL,           0
$
```

5.

```
/DOUBLE PRECISION RESULT
*200
START,          CLA CLL
                TAD A
                CIA
                DCA MINUSA
                TAD B
                SZL
                ISZ CH
                NOP
                CLL
                ISZ MINUSA
                JMP .-6
                DCA CL
                HLT
MINUSA,         0
A,              0011
B,              1234
CL,             0
CH,             0
$
```

6.

```
/DOUBLE PRECISION MULTIPLE OF 2
*200
START,          CLA CLL
                DCA NH
                TAD EXP
                CIA
                DCA MINUSE
ROTATE,         TAD N
                RAL
                DCA N
                TAD NH
                RAL
                DCA NH
```

```
                CLL
                ISZ MINUSE
                JMP ROTATE
                HLT
N,              1234
NH,             0
EXP,            3
MINUSE,         0
$
```

7.

```
/HOW MANY NEGATIVES?
*200
START,      CLA CLL
            DCA NEGS
            TAD K2777
            DCA 10
            TAD M1000
            DCA COUNT
TEST,       TAD I 10
            SPA CLA
            ISZ NEGS
            ISZ COUNT
            JMP TEST
            TAD NEGS
            HLT
NEGS,       0
K2777,      2777
M1000,      -1000
COUNT,      0
$
```

8.

```
/20 SECOND DELAY
*200
START,      TAD CONST
            DCA COUNT
            TAD CONST1
            DCA COUNT1
            ISZ COUNT1
            JMP .-1
            ISZ COUNT
            JMP .-3
            HLT
CONST,      6030        /-1000 DECIMAL
COUNT,      0
CONST1,     64          /52 DECIMAL
COUNT1,     0
$
```

9.
```
/20 0R 40 SECOND DELAY
*200
START,    CLA CLL
          TAD M2
          HLT
          OSR
          DCA TWICE
DELAY,    TAD CONST
          DCA COUNT
          TAD CONST1
          DCA COUNT1
          ISZ COUNT1
          JMP .-1
          ISZ COUNT
          JMP .-3
          ISZ TWICE
          JMP DELAY
          HLT
CONST,    5703
COUNT,    0
CONST1,   44
COUNT1,   0
M2,       -2
TWICE,    0
$
```

## Chapter 4

**Answers to selected exercises**

3.
```
          /SET LOCATIONS TO SWITCH REGISTER
          /VALUE
LOC. CONT. *200
0200 7300            CLA CLL
0201 1214            TAD K2000
0202 3215            DCA POINT
0203 1213            TAD M10
0204 3216            DCA COUNT
0205 7404            OSR
0206 3615            DCA I POINT
0207 2215            ISZ POINT
0210 2216            ISZ COUNT
0211 5205            JMP .-4
0212 7402            HLT
0213 7770 M10,       7770
0214 2000 K2000,     2000
0215 0000 POINT,     0
0216 0000 COUNT,     0
          $
```

4. /ADD TWO NUMBERS AND DISPLAY SUM

```
LOC.  CONT.  *200
0200  7300            CLA CLL
0201  7402            HLT
0202  7404            OSR
0203  3211            DCA A
0204  7402            HLT
0205  7404            OSR
0206  1211            TAD A
0207  7402            HLT
0210  5200            JMP .-10
0211  0000  A,         0
            $
```

## Chapter 6

**Answers to selected exercises**

1. /SUBROUTINE ALARM AND CALLING FOR IT

```
*200
START,      CLA CLL
            TLS
            JMS ALARM
            HLT
ALARM,      0
            TAD M5
            DCA RING5
            TAD KBELL
            JMS TYPE
            ISZ RING5
            JMP .-3
            JMP I ALARM
TYPE,       0
            TSF
            JMP .-1
            TLS
            CLA CLL
            JMP I TYPE
M5,         -5
RING5,      0
KBELL,      207     /ASCII FOR THE BELL
$
```

2. /TAB SPACE THE TELEPRINTER

```
*200
START,      CLA CLL
            TLS
            HLT
            OSR             /ACCEPT NUMBER OF
            JMS TAB         /SPACES FROM SR
            JMP .-3         /READY TO TAB MORE
TAB,        0
            CIA
            DCA NUMTAB
            TAD KSPACE
            JMS TYPE
            ISZ NUMTAB
            JMP .-3
            JMP I TAB
TYPE,       0
            TSF
            JMP .-1
            TLS
            CLA CLL
            JMP I TYPE
NUMTAB,     0
KSPACE,     240
$
```

3. /TEST ANSWER SHEET

```
*200
START,      CLA CLL
            TLS
HEADING,    TAD HEAD1
            DCA POINTR
            TAD AMOUNT
            DCA COUNT
            JMS CRLF
            TAD I POINTR
            JMS TYPE
            ISZ POINTR
            ISZ COUNT
            JMP .-4
            JMS CRLF
NUMBRS,     TAD K260
            DCA INTS
            ISZ INTS
            TAD INTS
            JMS NUMTYP
            TAD INTS
            TAD M271
            SZA CLA
            JMP .-6
```

```
TEN,        TAD K260
            IAC
            JMS TYPE
            TAD K260
            JMS NUMTYP
            HLT
HEAD1,      HEAD
POINTR,     0
HEAD,       310     /H
            311     /I
            323     /S
            324     /T
            317     /O
            322     /R
            331     /Y
            240     /SPACE
            324     /T
            305     /E
            323     /S
            324     /T
AMOUNT,     -14     /# OF HEADING CHARACTERS
COUNT,      0
K260,       260
INTS,       0
M271,       -271    /NEGATIVE OF ASCII FOR A 9
K215,       215     /ASCII FOR CR
K212,       212     /ASCII FOR LF
K256,       256     /ASCII FOR PERIOD
TYPE,       0
            TSF
            JMP .-1
            TLS
            CLA CLL
            JMP I TYPE
CRLF,       0
            TAD K215
            JMS TYPE
            TAD K212
            JMS TYPE
            JMP I CRLF
NUMTYP,     0
            JMS TYPE
            TAD K256
            JMS TYPE
            JMS CRLF
            JMP I NUMTYP

$
```

4.
```
/TWO DIGIT OCTAL SQUARE CONVERSATIONAL
/ PROGRAM
*200
START,  CLA CLL
        TLS
        JMS CRLF
        JMS LISN        /GET FIRST DIGIT
        TAD M260
        RAL CLL
        RTL
        DCA NUMBER
        JMS LISN        /GET SECOND DIGIT
        TAD M260
        TAD NUMBER      /NUMBER IS NOW IN AC
        DCA NUMBER
MULT,   TAD NUMBER
        CIA
        DCA TALLY
        TAD NUMBER
        ISZ TALLY
        JMP .-2
        DCA NUMSQR
TYPSQU, TAD MESAG1
        DCA POINTR
        TAD M10
        DCA ENDCHK
        JMS MESAGE
TYPANS, TAD M4
        DCA DIGCTR
        DCA STORE
        TAD NUMSQR
        CLL RAL
UNPACK, TAD STORE
        RAL
        RTL
        DCA STORE
        TAD STORE
        AND K7
        TAD K260
        JMS TYPE
        ISZ DIGCTR
        JMP UNPACK
TYPOCT, TAD MESAG2
        DCA POINTR
        TAD M7
        DCA ENDCHK
        JMS MESAGE
        JMS CRLF
        JMP START+2
```

```
TYPE,     0
          TSF
          JMP .-1
          TLS
          CLA
          JMP I TYPE
CRLF,     0
          TAD K215
          JMS TYPE
          TAD K212
          JMS TYPE
          JMP I CRLF
LISN,     0
          KSF
          JMP .-1
          KRB
          TLS
          JMP I LISN
MESAGE,   0
          TAD I POINTR
          JMS TYPE
          ISZ POINTR
          ISZ ENDCHK
          JMP .-4
          JMP I MESAGE
NUMBER,   0
M260,     -260
TALLY,    0
NUMSQR,   0
MESAG1,   START1
POINTR,   0
M10,      -10
ENDCHK,   0
STORE,    0
M4,       -4
DIGCTR,   0
K7,       7
M7,       -7
K260,     260
K212,     212
K215,     215
MESAG2,   START2
START1,   323     /S
          321     /Q
          325     /U
          301     /A
          322     /R
          305     /E
          304     /D
          275     /=
```

```
START2,   240     /SPACE
          317     /O
          303     /C
          324     /T
          301     /A
          314     /L
          256     /PERIOD
$
```

# appendix b
# character codes

**ASCII[1] Character Set**

| Character | 8-Bit Octal | 6-Bit Octal | Character | 8-Bit Octal | 6-Bit Octal |
|---|---|---|---|---|---|
| A | 301 | 01 | ! | 241 | 41 |
| B | 302 | 02 | " | 242 | 42 |
| C | 303 | 03 | # | 243 | 43 |
| D | 304 | 04 | $ | 244 | 44 |
| E | 305 | 05 | % | 245 | 45 |
| F | 306 | 06 | & | 246 | 46 |
| G | 307 | 07 | ' | 247 | 47 |
| H | 310 | 10 | ( | 250 | 50 |
| I | 311 | 11 | ) | 251 | 51 |
| J | 312 | 12 | * | 252 | 52 |
| K | 313 | 13 | + | 253 | 53 |
| L | 314 | 14 | , | 254 | 54 |
| M | 315 | 15 | - | 255 | 55 |
| N | 316 | 16 | . | 256 | 56 |
| O | 317 | 17 | / | 257 | 57 |
| P | 320 | 20 | : | 272 | 72 |
| Q | 321 | 21 | ; | 273 | 73 |
| R | 322 | 22 | < | 274 | 74 |
| S | 323 | 23 | = | 275 | 75 |
| T | 324 | 24 | > | 276 | 76 |
| U | 325 | 25 | ? | 277 | 77 |
| V | 326 | 26 | @ | 300 | |
| W | 327 | 27 | [ | 333 | 33 |
| X | 330 | 30 | \ | 334 | 34 |
| Y | 331 | 31 | ] | 335 | 35 |
| Z | 332 | 32 | ↑(∧)[2] | 336 | 36 |
| 0 | 260 | 60 | ←(—)[2] | 337 | 37 |
| 1 | 261 | 61 | Leader/Trailer | 200 | |
| 2 | 262 | 62 | LINE FEED | 212 | |
| 3 | 263 | 63 | Carriage RETURN | 215 | |
| 4 | 264 | 64 | SPACE | 240 | 40 |
| 5 | 265 | 65 | RUBOUT | 377 | |
| 6 | 266 | 66 | Blank | 000 | |
| 7 | 267 | 67 | BELL | 207 | |
| 8 | 270 | 70 | TAB | 211 | |
| 9 | 271 | 71 | FORM | 214 | |

[1] An abbreviation for American Standard Code for Information Interchange.
[2] The character in parentheses is printed on some console terminals.

## CHARACTER CODES

| 8-bit ASCII Code | 6-bit Code | DEC 029 Card Code | DEC 026 Card Code | Character Representation | Remarks |
|---|---|---|---|---|---|
| 240 | 40 | blank | blank | | space (non-printing) |
| 241 | 41 | 11-8-2 | 12-8-7 | ! | exclamation point |
| 242 | 42 | 8-7 | 0-8-5 | " | quotation marks |
| 243 | 43 | 8-3 | 0-8-6 | # | number sign(10) |
| 244 | 44 | 11-8-3 | 11-8-3 | $ | dollar sign |
| 245 | 45 | 0-8-4 | 0-8-7 | % | percent |
| 246 | 46 | 12 | 11-8-7 | & | ampersand |
| 247 | 47 | 8-5 | 8-6 | ' | apostrophe or acute accent |
| 250 | 50 | 12-8-5 | 0-8-4 | ( | opening parenthesis |
| 251 | 51 | 11-8-5 | 12-8-4(1) | ) | closing parenthesis |
| 252 | 52 | 11-8-4 | 11-8-4 | * | asterisk |
| 253 | 53 | 12-8-6 | 12 | + | plus |
| 254 | 54 | 0-8-3 | 0-8-3 | , | comma |
| 255 | 55 | 11 | 11 | - | minus sign or hyphen |
| 256 | 56 | 12-8-3 | 12-8-3 | . | period or decimal point |
| 257 | 57 | 0-1 | 0-1 | / | slash |
| 260 | 60 | 0 | 0 | 0 | |
| 261 | 61 | 1 | 1 | 1 | |
| 262 | 62 | 2 | 2 | 2 | |
| 263 | 63 | 3 | 3 | 3 | |
| 264 | 64 | 4 | 4 | 4 | |
| 265 | 65 | 5 | 5 | 5 | |
| 266 | 66 | 6 | 6 | 6 | |
| 267 | 67 | 7 | 7 | 7 | |
| 270 | 70 | 8 | 8 | 8 | |
| 271 | 71 | 9 | 9 | 9 | |
| 272 | 72 | 8-2 | 11-8-2 | : | colon |
| 273 | 73 | 11-8-6 | 0-8-2 | ; | semicolon |
| 274 | 74 | 12-8-4 | 12-8-6 | < | less than |
| 275 | 75 | 8-6 | 8-3 | = | equals |
| 276 | 76 | 0-8-6 | 11-8-6 | > | greater than |
| 277 | 77 | 0-8-7 | 12-8-2 | ? | question mark |
| 300 | 00 | 8-4 | 8-4 | @ | at sign |
| 301 | 01 | 12-1 | 12-1 | A | |
| 302 | 02 | 12-2 | 12-2 | B | |
| 303 | 03 | 12-3 | 12-3 | C | |
| 304 | 04 | 12-4 | 12-4 | D | |
| 305 | 05 | 12-5 | 12-5 | E | |
| 306 | 06 | 12-6 | 12-6 | F | |
| 307 | 07 | 12-7 | 12-7 | G | |

## CHARACTER CODES

| 8-bit ASCII Code | 6-bit Code | DEC 029 Card Code | DEC 026 Card Code | Character Representation | Remarks |
|---|---|---|---|---|---|
| 310 | 10 | 12-8 | 12-8 | H | |
| 311 | 11 | 12-9 | 12-9 | I | |
| 312 | 12 | 11-1 | 11-1 | J | |
| 313 | 13 | 11-2 | 11-2 | K | |
| 314 | 14 | 11-3 | 11-3 | L | |
| 315 | 15 | 11-4 | 11-4 | M | |
| 316 | 16 | 11-5 | 11-5 | N | |
| 317 | 17 | 11-6 | 11-6 | O | |
| 320 | 20 | 11-7 | 11-7 | P | |
| 321 | 21 | 11-8 | 11-8 | Q | |
| 322 | 22 | 11-9 | 11-9 | R | |
| 323 | 23 | 0-2 | 0-2 | S | |
| 324 | 24 | 0-3 | 0-3 | T | |
| 325 | 25 | 0-4 | 0-4 | U | |
| 326 | 26 | 0-5 | 0-5 | V | |
| 327 | 27 | 0-6 | 0-6 | W | |
| 330 | 30 | 0-7 | 0-7 | X | |
| 331 | 31 | 0-8 | 0-8 | Y | |
| 332 | 32 | 0-9 | 0-9 | Z | |
| 333 | 33 | 12-8-2(5) | 11-8-5 | [ | opening bracket, SHIFT/K |
| 334 | 34 | 11-8-7(6) | 8-7 | \ | backslash, SHIFT/L(8) |
| 335 | 35 | 0-8-2 | 12-8-5 | ] | closing bracket, SHIFT/M |
| 336 | 36 | 12-8-7(7) | 8-5 | ^ | circumflex(2) |
| 337 | 37 | 0-8-5(3) | 8-2(3) | _ | underline(4,9) |

Footnotes:

(1) On some DEC 026 Keyboards this character is graphically represented as □.

(2) On most DEC Teletypes circumflex is replaced by up-arrow (↑).

(3) A card containing this code in column 1 with all remaining columns blank is an end-of-file card.

(4) On most DEC Teletypes underline is replaced by backarrow (←).

(5) On some 029 keyboards this character is graphically represented as a cent sign (¢).

(6) On some 029 keyboards this character is graphically represented as logical NOT (¬).

(7) On some 029 keyboards this character is graphically represented as vertical bar (|).

(8) On some LP8 line printers, the character diamond (◊) is printed instead of backslash.

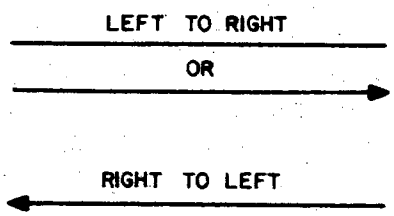
(9) On some LP8 line printers, the character heart (♡) is printed instead of underline.

(10) The number sign on some terminals is replaced by pound sign (£).

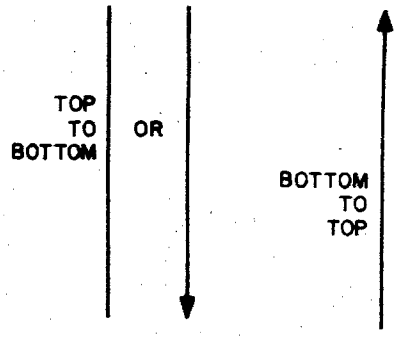
# appendix C
# Flowchart guide

The following is a partial list of flowchart symbols which can be used to diagram the logical flow of a program. The symbols may be made sufficiently large to include the pertinent information.

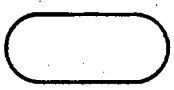
## REPRESENTATION OF FLOW




The direction of flow in a program is represented by lines drawn between symbols. These lines indicate the order in which the operations are to be performed. Normal direction of flow is from left to right and top to bottom. When the flow direction is not from left to right or top to bottom, arrowheads are placed on the reverse direction flowlines. Arrowheads may also be used on normal flow lines for increased clarity.




## TERMINAL

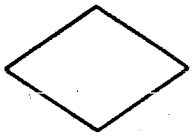


The oval symbol represents a terminal point in a program. It can be used to indicate a start, stop, or interrupt of program flow. The appropriate word is included within the symbol.

## PROCESSING

The rectangular symbol represents a processing function. The process which the symbol is used to represent could be an instruction or a group of instructions to carry out a given task. A brief description of the task to be performed is included within the symbol.

DECISION

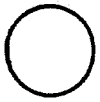


A diamond is used to indicate a point in a program where a choice must be made to determine the flow of the program from that point. A test condition is included within the symbol and the possible results of the test are used to label the respective flows from the symbol.

PREDEFINED PROCESS

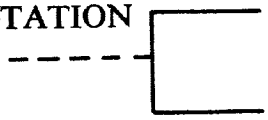
This symbol is used to represent an operation or group of operations not detailed in the flowchart. It is usually detailed in another flowchart. A subroutine is often represented in this manner.

CONNECTOR



The circular symbol represents an entry from or an exit to another part of the program flowchart. A number or a letter is enclosed to label the corresponding exits and entries. This symbol does not represent a program operation.

ANNOTATION

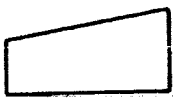


An addition of descriptive comments or explanatory notes for clarification is included within this symbol.

INPUT/OUTPUT

This symbol is used in a flowchart to represent the input or output of information. This symbol may be used for all input/output functions, or symbols for specific types of input or output (such as those which follow) may be used.

MANUAL INPUT

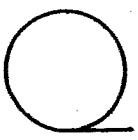


This symbol may be used to represent the manual input of information by means of online keyboards, switch settings, etc.

PUNCHED TAPE

The input or output of information in which the medium is punched tape may be represented by this symbol.

MAGNETIC TAPE



This symbol is used in a flowchart to represent magnetic tape input or output.

# appendix d

**Table D-1 PDP-8/E Memory Reference Instructions**
**(Refer to Chapter 3)**

| Mnemonic Symbol | Octal Code | Indicators | Execution Times: Direct Address | Execution Times: Indirect Address | Execution Times: Auto-Indexed | Operation |
|---|---|---|---|---|---|---|
| AND Y | 0 | IR = 0,F,E | 2.6 | 3.8 | 4.0 | Logical AND between Y and AC |
| TAD Y | 1 | IR = 1,F,E | 2.6 | 3.8 | 4.0 | Two's complement Add Y to AC |
| ISZ Y | 2 | IR = 2,F,E | 2.6 | 3.8 | 4.0 | Increment Y and skip if zero |
| DCA Y | 3 | IR = 3,F,E | 2.6 | 3.8 | 4.0 | Deposit at Y and clear AC |
| JMS Y | 4 | IR = 4,F,E | 2.6 | 3.8 | 4.0 | Jump to subroutine at Y |
| JMP Y | 5 | IR = 5,F | 1.2 | 2.4 | 2.6 | Jump to Y |

**Table D-2 Loading Constants Into The Accumulator**

| Mnemonic | Decimal Constant | Octal Code | Instructions Combined |
|---|---|---|---|
| NL0000 = | 0 | 7300 | CLA CLL |
| NL0001 = | 1 | 7301 | CLA CLL IAC |
| NL0002 = | 2 | 7305 | CLA CLL IAC RAL |
| | | (or) | |
| NL0002 = | 2 | 7326 | CLA CLL CML RTL |
| NL0003 = | 3 | 7325 | CLA CLL CML IAC RAL |
| NL0004 = | 4 | 7307 | CLA CLL IAC RTL |
| NL0006 = | 6 | 7327 | CLA CLL CML IAC RTL |
| NL0100 = | 64 | 7203 | CLA IAC BSW |
| NL2000 = | 1024 | 7332 | CLA CLL CML RTR |
| NL3777 = | 2047 | 7350 | CLA CLL CMA RAR |
| NL4000 = | −0 | 7330 | CLA CLL CML RAR |
| NL5777 = | −1025 | 7352 | CLA CLL CMA RTR |
| NL6000 = | −1024 | 7333 | CLA CLL CML IAC RTL |
| NL7775 = | −3 | 7346 | CLA CLL CMA RTL |
| NL7776 = | −2 | 7344 | CLA CLL CMA RAL |
| NL7777 = | −1 | 7340 | CLA CLL CMA |

**Table D-3 Group 1 Operate Microinstructions**

| Mnemonic Symbol | Octal Code | Sequence | Operation |
|---|---|---|---|
| NOP | 7000 | — | No operation. Causes a 1.2 μs program delay. |
| IAC | 7001 | 3 | Increment AC. The content of the AC is incremented by one in two's complement arithmetic. |
| RAL | 7004 | 4 | Rotate AC and L left. The content of the AC and the L are rotated left one place. |
| RTL | 7006 | 4 | Rotate two places to the left. Equivalent to two successive RAL operations. |
| RAR | 7010 | 4 | Rotate AC and L right. The content of the AC and L are rotated right one place. |
| RTR | 7012 | 4 | Rotate two places to the right. Equivalent to two successive RAR operations. |
| BSW | 7002 | 4 | Byte swap. |
| CML | 7020 | 2 | Complement L. |
| CMA | 7040 | 2 | Complement AC. The content of the AC is set to the one's complement of its current content. |
| CIA | 7041 | 2, 3 | Complement and increment accumulator. Used to form two's complement. |
| CLL | 7100 | 1 | Clear L. |
| CLL RAL | 7104 | 1, 4 | Shift positive number one left. |
| CLL RTL | 7106 | 1, 4 | Clear link, rotate two left. |
| CLL RAR | 7110 | 1, 4 | Shift positive number one right. |
| CLL RTR | 7112 | 1, 4 | Clear link, rotate two right. |
| STL | 7120 | 1, 2 | Set link. The L is set to contain a binary 1. |
| CLA | 7200 | 1 | Clear AC. To be used alone or in OPR 1 combinations. |
| CLA IAC | 7201 | 1, 3 | Set AC = 1. |
| GLK | 7204 | 1, 4 | Get link. Transfer L into AC11. |
| CLA CLL | 7300 | 1 | Clear AC and L. |
| STA | 7240 | 2 | Set AC =-1. Each bit of the AC is set to contain a 1. |

**Table D-4 Group 2 Operate Microinstructions**

| Mnemonic Symbol | Octal Code | Sequence | Operation |
|---|---|---|---|
| HLT | 7402 | 3 | Halt. Stops the program after completion of the cycle in process. If this instruction is combined with others in the OPR 2 group the other operations are completed before the end of the cycle. |
| OSR | 7404 | 3 | OR with switch register. The OR function is performed between the content of the SR and the content of the AC, with the result left in the AC. |
| SKP | 7410 | 1 | Skip, unconditional. The next instruction is skipped. |
| SNL | 7420 | 1 | Skip if L $\neq$ 0. |
| SZL | 7430 | 1 | Skip if L = 0. |
| SZA | 7440 | 1 | Skip if AC = 0. |
| SNA | 7450 | 1 | Skip if AC $\neq$ 0. |
| SZA SNL | 7460 | 1 | Skip if AC = 0, or L $\neq$ 1, or both. |
| SNA SZL | 7470 | 1 | Skip if AC $\neq$ 0 and L = 0. |
| SMA | 7500 | 1 | Skip on minus AC. If the content of the AC is a negative number, the next instruction is skipped. |
| SPA | 7510 | 1 | Skip on positive AC. If the content of the AC is a positive number, including zero, the next instruction is skipped. |
| SMA SNL | 7520 | 1 | Skip if AC $<$ 0, or L = 1, or both. |
| SPA SZL | 7530 | 1 | Skip if AC $\geqslant$ 0 and if L = 0. |
| SMA SZA | 7540 | 1 | Skip if AC $\leqslant$ 0. |
| SPA SNA | 7550 | 1 | Skip if AC $>$ 0. |
| CLA | 7600 | 2 | Clear AC. To be used alone or in OPR 2 combinations. |
| LAS | 7604 | 1, 3 | Load AC with SR. |
| SZA CLA | 7640 | 1, 2 | Skip if AC = 0, then clear AC. |
| SNA CLA | 7650 | 1, 2 | Skip if AC $\neq$ 0, then clear AC. |
| SMA CLA | 7700 | 1, 2 | Skip if AC $<$ 0, then clear AC. |
| SPA CLA | 7710 | 1, 2 | Skip if AC $\geqslant$ 0, then clear AC. |

**Table D-5 Group 3 Operate Microinstructions**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| NOP | 7401 | No Operation |
| MQL | 7421 | Load Multiplier Quotient |
| MQA | 7501 | Multiplier Quotient OR into Accumulator |
| SWP | 7521 | Swap Accumulator and Multiplier Quotient |
| CLA | 7601 | Clear Accumulator |
| CAM | 7621 | Clear Accumulator and Multiplier Quotient (CLA MQL) |
| ACL | 7701 | Clear Accumulator, Load Multiplier Quotient into Accumulator (CLA MQA) |
| CLA SWP | 7721 | Load Multiplier Quotient into Accumulator, Clear Multiplier Quotient |

**Table D-6 Programmed Data Transfer Instructions**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| ION | 6001 | Interrupt Turn On |
| IOF | 6002 | Interrupt Turn Off |
| SKON | 6000 | Skip if Interrupt On, IOF |
| SRQ | 6003 | Skip if Interrupt Request |
| GTF | 6004 | Get Flags |
| RTF | 6005 | Restore Flag, ION |
| SGT | 6006 | Skip if "Greater Than" Flag is Set |
| CAF | 6007 | Clear All Flags |

**Table D-7 KM8-E Memory Extension**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| GTF | 6004 | Get Flags |
| RTF | 6005 | Restore Flags, ION |
| CDF | 62N1 | Change to Data Field N (N=0 to 7) |
| CIF | 62N2 | Change to Instruction Field N (N=0 to 7) |
| CDI | 62N3 | Change Data Field, Change Instruction Field (CDF CIF) |
| RDF | 6214 | Read Data Field |
| RIF | 6224 | Read Instruction Field |
| RIB | 6234 | Read Interrupt Buffer |
| RMF | 6244 | Restore Memory Field |

**Table D-8 KE8-E Extended Arithmetic Element**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| | | MODE CHANGING INSTRUCTIONS |
| SWAB | 7431 | Switch from Mode A to B |
| SWBA | 7447 | Switch from Mode B to A |
| SKB | 7471 | Skip if Mode B |
| | | STANDARD INSTRUCTIONS |
| CAM | 7621 | 0→ AC, 0→ MQ |
| MQA | 7501 | MQ "OR"ed with AC→ AC |
| ACL | 7701 | MQ→AC (MQA CLA) |
| MQL | 7421 | AC→ MQ, 0→ AC |
| SWP | 7521 | AC→ MQ, MQ→ AC |
| | | MODE A INSTRUCTIONS |
| SCA | 7441 | Step Counter "OR" with AC |
| SCA CLA | 7641 | Step Counter to AC |
| SCL | 7403 | Step Counter Load from Memory |
| MUY | 7405 | Multiply |
| DVI | 7407 | Divide |
| NMI | 7411 | Normalize |
| SHL | 7413 | Shift Left |
| ASR | 7415 | Arithmetic Shift Right |
| LSR | 7417 | Logical Shift Right |
| | | MODE B INSTRUCTIONS |
| ACS | 7403 | AC to Step Count |
| MUY | 7405 | Multiply |
| DVI | 7407 | Divide |
| NMI | 7411 | Normalize |
| SHL | 7413 | Shift Left |
| ASR | 7415 | Arithmetic Shift Right |
| LSR | 7417 | Logical Shift Right |
| | | DOUBLE PRECISION INSTRUCTIONS |
| DAD | 7443 | Double Precision Add |
| DST | 7445 | Double Precision Store |
| DPIC | 7573 | Double Precision Increment |
| DCM | 7575 | Double Precision Complement |
| DPSZ | 7451 | Double Precision Skip if Zero |

**Table D-9 Teletype Keyboard/Reader**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| KCF | 6030 | Clear Keyboard Flag |
| KSF | 6031 | Skip on Keyboard Flag |
| KCC | 6032 | Clear Keyboard Flag, and AC, Advance Reader |
| KRS | 6034 | Read Keyboard Buffer Static |
| KIE | 6035 | Set/Clear Interrupt Enable |
| KRB | 6036 | Read Keyboard Buffer, Clear Flag |

**Table D-10 Teletype Teleprinter/Punch**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| TFL | 6040 | Set Teleprinter Flag |
| TSF | 6041 | Skip on Teleprinter Flag |
| TCF | 6042 | Clear Teleprinter Flag |
| TPC | 6044 | Load Teleprinter and Print |
| TSK | 6045 | Skip on Printer or Keyboard Flag |
| TLS | 6046 | Load Teleprinter Sequence |

**Table D-11 PR8-E Paper Tape Readers**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| RPE | 6010 | Set Reader/Punch Interrupt Enable |
| RSF | 6011 | Skip on Reader Flag |
| RRB | 6012 | Read Reader Buffer |
| RFC | 6014 | Reader Fetch Character |
| RCC | 6016 | Read Buffer and Fetch New Character (RRB, RFC) |
| PCE | 6020 | Clear Reader/Punch Interrupt Enable |

**Table D-12 PP8-E Paper Tape Punch**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| RPE | 6010 | Set Reader/Punch Interrupt Enable |
| PCE | 6020 | Clear Reader/Punch Interrupt Enable |
| PSF | 6021 | Skip on Punch Flag |
| RCF | 6022 | Clear Punch Flag |
| PPC | 6024 | Load Punch Buffer and Punch Character |
| PLS | 6026 | Load Punch Buffer Sequence |

**Table D-13 PC8-E Reader/Punch**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| RPE | 6010 | Set Reader/Punch Interrupt Enable |
| RSF | 6011 | Skip on Reader Flag |
| RRB | 6012 | Read Reader Buffer |
| RFC | 6014 | Reader Fetch Character |
| RFC, RRB | 6016 | Read Buffer and Fetch New Character |
| PCE | 6020 | Clear Reader/Punch Interrupt Enable |
| PSF | 6021 | Skip on Punch Flag |
| PCF | 6022 | Clear Punch Flag |
| PPC | 6024 | Load Punch Buffer and Punch Character |
| PLS | 6026 | Load Punch Buffer Sequence |

**Table D-14 TC08-P DECtape Control**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| DTRA | 6761 | Read Status Register A | 2.6 |
| DTCA | 6762 | Clear Status Register A | 2.6 |
| DTXA | 6764 | Load Status Register A | 2.6 |
| DTLA | 6766 | Clear and Load Status Register A | 3.6 |
| DTSF | 6771 | Skip on Flag | 2.6 |
| DTRB | 6772 | Read Status Register B | 2.6 |
| DTXB | 6774 | Load Status Register B | 2.6 |

Address Locations: 7754 = Word Count
7755 = Current Address

**Table D-15 TC58 DECmagtape System**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| MTSF | 6701 | Skip on Error Flag or Magnetic Tape Flag |
| MTCR | 6711 | Skip on Tape Control Ready |
| MTTR | 6721 | Skip on Tape Transport Ready |
| MTAF | 6712 | Clear Registers, Error Flag and Magnetic Tape Flag |
| MTRC | 6724 | Inclusive OR Contents of Command Register |
| MTCM | 6714 | Inclusive OR Contents of AC |
| MTLC | 6716 | Load Command Register |
| none | 6704 | Inclusive OR Contents of Status Register |
| MTRS | 6706 | Read Status Register |
| MTGO | 6722 | Mag Tape "GO" |
| none | 6702 | Clear AC |

**Table D-16 RK08-P Control and RK01 Disk Drive and Control**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| DLDA | 6731 | Load Disk Address (Maintenance Only) | 2.6 |
| DLDC | 6732 | Load Command Register | 2.6 |
| DLDR | 6733 | Load Disk Address and Read | 2.6 |
| DRDA | 6734 | Read Disk Address | 2.6 |
| DLDW | 6735 | Load Disk Address and Write | 2.6 |
| DRDC | 6736 | Read Disk Command Register | 3.6 |
| DCHP | 6737 | Load Disk Address and Check Parity | 4.6 |
| DRDS | 6741 | Read Disk Status Register | 2.6 |
| DCLS | 6742 | Clear Status Register | 2.6 |
| DMNT | 6743 | Load Maintenance Register | 3.6 |
| DSKD | 6745 | Skip on Disk Done | 3.6 |
| DSKE | 6747 | Skip on Disk Error | 4.6 |
| DCLA | 6751 | Clear All | 2.6 |
| DRWC | 6752 | Read Word Count Register | 3.6 |
| DLWC | 6753 | Load Word Count Register | 3.6 |
| DLCA | 6755 | Load Current Address Register | 3.6 |
| DRCA | 6757 | Read Current Address Register | 4.6 |

**Table D-17 DF32-D Disk File and Control**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| DCMA | 6601 | Clear Disk Address Register | 2.6 |
| DMAR | 6603 | Load Disk Address Register and Read | 3.6 |
| DMAW | 6605 | Load Disk Address Register and Write | 3.6 |
| DCEA | 6611 | Clear Disk Extended Address | 2.6 |
| DSAC | 6612 | Skip on Address Confirmed Flag | 2.6 |
| DEAL | 6615 | Load Disk Extended Address | 3.6 |
| DEAC | 6616 | Read Disk Extended Address | 3.6 |
| DFSE | 6621 | Skip on Zero Error Flag | 2.6 |
| DFSC | 6622 | Skip on Data Completion Flag | 2.6 |
| DMAC | 6626 | Read Disk Memory Address Register | 3.6 |

Address Locations:

7750 = Word Count
7751 = Memory Address

**Table D-18 RF08 Disk File**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| DCIM | 6611 | Clear Disk Interrupt Enable and Core Memory Address Extension Register |
| DIML | 6615 | Load Interrupt Enable and Memory Address Extension Register |
| DIMA | 6616 | Load Interrupt and Extended Memory Address |
| DFSE | 6621 | Skip on Disc Error |
| DISK | 6623 | Skip Error or Completion Flag |
| DCXA | 6641 | Clear High Order Address Register |
| DXAL | 6643 | Clear and Load High Order Address Register |
| DXAC | 6645 | Clear AC & Load DAR into AC |
| DMMT | 6646 | Initiate Maintenance Register |

**Table D-19 TM8-E/F Control**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| LWCR | 6701 | Load Word Count Register |
| CWCR | 6702 | Clear Word Count Register |
| LCAR | 6703 | Load Current Address Register |
| CCAR | 6704 | Clear Current Address Register |
| LCMR | 6705 | Load Command Register |
| LFGR | 6706 | Load Function Register |
| LDBR | 6707 | Load Data Buffer Register |
| RWCR | 6711 | Read Word Count Register |
| CLT | 6712 | Clear Transport |
| RCAR | 6713 | Read Current Address Register |
| RMSR | 6714 | Read Main Status Register |
| RCMR | 6715 | Read Command Register |
| RFSR | 6716 | Read Function Register & Status |
| RDBR | 6717 | Read Data Buffer |
| SKEF | 6721 | Skip if Error Flag |
| SKCB | 6722 | Skip if Not Busing |
| SKJD | 6723 | Skip if Job Done |
| SKTR | 6724 | Skip if Tape Ready |
| CLF | 6725 | Clear Controller and Master |

**Table D-20 LE-8 Line Printer**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| PSKF | 6661 | Skip on Character Flag |
| PCLF | 6662 | Clear the Character Flag |
| PSKE | 6663 | Skip on Error |
| PSTB | 6664 | Load Printer Buffer, Print on Full Buffer or Control Character |
| PSIE | 6665 | Set Program Interrupt Flag |
| PCLF, PSTB | 6666 | Clear Line Printer Flag, Load Character, and Print |
| PCIE | 6667 | Clear Program Interrupt Flag |

**Table D-21 CR8-E Card Reader and Control or CM8-E Optical Mark Card Reader and Control**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| RCSF | 6631 | Skip on Data Ready |
| RCRA | 6632 | Read Alphanumeric |
| RCRB | 6634 | Read Binary |
| RCNO | 6635 | Read Conditions Out to Card Reader |
| RCRC | 6636 | Read Compressed |
| RCNI | 6637 | Read Condition In From Card Reader |
| RCSD | 6671 | Skip on Card Done Flag |
| RCSE | 6672 | Select Card Reader and Skip if Ready |
| RCRD | 6674 | Clear Card Done Flag |
| RCSI | 6675 | Skip If Interrupt Being Generated |
| RCTF | 6677 | Clear Transition Flags |

**Table D-22 XY8-E Incremental Plotter Control**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| PLCE | 6500 | Clear Interrupt Enable |
| PLSF | 6501 | Skip on Plotter Flag |
| PLCF | 6502 | Clear Plotter Flag |
| PLPU | 6503 | Pen Up |
| PLLR | 6504 | Load Direction Register, Set Flag |
| PLPD | 6505 | Pen Down |
| PLCF, PLLR | 6506 | Clear Flag, Load Direction Register, Set Flag |
| PLSE | 6507 | Set Interrupt Enable |

**Table D-23 VC8-E CRT Display Control**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| DILC | 6050 | Clears Enables, Flags and Delays |
| DICD | 6051 | Clears Done Flag |
| DISD | 6052 | Skip on Done Flag |
| DILX | 6053 | Load X Register |
| DILY | 6054 | Load Y Register |
| DIXY | 6055 | Clear Done Flag; Intensify; Set Done Flag |
| DILE | 6056 | Transfers AC to Enable Register |
| DIRE | 6057 | Transfers Display Enable/Status Register to AC |

**Table D-24 VW01 Writing Tablet**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| WTSC | 6054 | Set Tablet Controls |
| WTRX | 6052 | Read X |
| WTRS | 6072 | Read Status |
| WTSE | 6074 | Select Tablet |
| WTMN | 6064 | Clear Set XY |

**Table D-25 DC02-F 8-Channel Multiple Teletype Control**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| MTPF | 6113 | Read Transmitter Flag |
| MINT | 6115 | Set Interrupt Flip-Flop |
| MTON | 6117 | Select Specified Station |
| MTKF | 6123 | Read Receiver Flag Status |
| MINS | 6125 | Skip on Interrupt Request |
| MTRS | 6127 | Read Station Status |
| MKSF | 6111 | Skip on Key Board Flag |
| MKCC | 6112 | Clear Receive Flag |
| MKRS | 6114 | Receive Operation |
| NONE | 6116 | Combined MKRS & MICCC |
| MTSF | 6121 | Skip on Transmitter Flag |
| MTCF | 6122 | Clear Transmitter Flag |
| MTPC | 6124 | Transmit Operation |
| NONE | 6126 | Combined MTCF & MTPC |

**Table D-26 BB08-P General Purpose Interface Unit**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| GTSF | 6361 | Skip on Transmit Flag | 2.6 |
| GCTF | 6362 | Clear Transmit Flag | 2.6 |
| | 6564 | (User-Assigned) | 2.6 |
| GRSF | 6371 | Skip on Receive Flag | 2.6 |
| GCRF | 6372 | Clear Receive Flag | 2.6 |
| GRDB | 6374 | Read Device Buffer | 2.6 |

**Table D-27 Universal Digit Controller (UDC)**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| UDSS | 6351 | Skip on Scan Not Busy | 2.6 |
| UDSC | 6353 | Start Interrupt Scan | 3.6 |
| UDRA | 6356 | Read Address and Generic Type | 3.6 |
| UDLS | 6357 | Load Previous Status | 4.6 |
| UDSF | 6361 | Skip on UDC Flag and Clear Flag | 2.6 |
| UDLA | 6363 | Load Address | 3.6 |
| UDEI | 6364 | Enable UDC Interrupt Flag | 2.6 |
| UDDI | 6365 | Disable UDC Interrupt Flag | 3.6 |
| UDRD | 6366 | Clear AC and Read Data | 3.6 |
| UDLD | 6367 | Load Data and Clear AC | 4.6 |

**Table D-28 DR8-EA 12-Channel Buffered Digital I/O**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| DBDI | 65x0 | Disable Interrupt |
| DBEI | 65x1 | Enable Interrupt |
| DBSK | 65x2 | Skip on Done Flag |
| DBCI | 65x3 | Clear Selective Input Register |
| DBRI | 65x4 | Transfer Input to AC |
| DBCO | 65x5 | Clear Selective Output Register |
| DBSO | 65x6 | Set Selective Output Register |
| DBRO | 65x7 | Transfer Output to AC |

**Table D-29 MP8E-Memory Parity**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| DPI | 6100 | Disable Memory Parity Error Interrupt |
| SMP | 6101 | Skip on No Memory Parity Error |
| EPI | 6103 | Enable Memory Parity Error Interrupt |
| CMP | 6104 | Clear Memory Parity Error Flag |
| SMP, CMP | 6105 | Skip on No Memory Parity Error, Clear Memory Parity Error Flag |
| CEP | 6106 | Check for Even Parity |
| SPO | 6107 | Skip on Memory Parity Option |

**Table D-30 Synchronous Modem Interface**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| SGTT | 6405 | Transmit Go |
| SGRR | 6404 | Receive Go |
| SSCD | 6400 | Skip if Character Detected |
| SCSD | 6406 | Clear Sync Detect |
| SSRO | 6402 | Skip if Receive Word Count Overflow |
| SCSI | 6401 | Clear Synchronous Interface |
| SRTA | 6407 | Read Transfer Address Register |
| SLCC | 6412 | Load Control |
| SSRG | 6410 | Skip if Ring Flag |
| SSCA | 6411 | Skip if Carrier/AGC Flag |
| SRS2 | 6414 | Read Status 2 |
| SRS1 | 6415 | Read Status 1 |
| SLFL | 6413 | Load Field |
| SSBE | 6416 | Skip on Bus Error |
| SRCD | 6417 | Read Character Detected (if AC0=0)<br>Maintenance Instruction (if AC0=1) |
| SSTO | 6403 | Skip if Transmit Word Count Overflows |

Break Address Locations:

| | |
|---|---|
| 7720 | |
| 7721 | Test Characters |
| 7722 | |
| 7723 | |
| 7724 | Receive Word Count |
| 7725 | Receive Current Address |
| 7726 | Not Used |
| 7727 | Transmit Word Count |
| 7730 | Transmit Current Address |

For additional interfaces:

| Device Codes | Break Locations |
|---|---|
| 42, 43 | 7700-7710 |
| 44, 45 | 7660-7670 |
| 46, 47 | 7640-7650 |

**Table D-31 Multicycle Data Break Locations**

| Assigned Locations | Date Break Device | Channel |
|---|---|---|
| 7640-7650 | DP8-EA/EB | 4 |
| 7660-7670 | DP8-EA/EB | 3 |
| 7700-7710 | DP8-EA/EB | 2 |
| 7720-7730 | DP8-EA/EB | 1 |
| 7750,7751 | DF32-D | |
| 7752,7753 | (Reserved for Industry Standard Magnetic Tape) | |
| 7754,7755 | TC08-P | |

**Table D-32 KM8-E Time-Share**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| CINT | 6204 | Clear User Interrupt |
| SINT | 6254 | Skip on User Interrupt |
| CUF | 6264 | Clear User Flag |
| SUF | 6274 | Set User Flag |

**Table D-33 DK8-EP Programmable Real Time Clock**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| CLZE | 6130 | Clear Clock Enable Register per AC |
| CLSK | 6131 | Skip on Clock Interrupt |
| CLOE | 6132 | Set Clock Enable Register per AC |
| CLAB | 6133 | AC to Clock Buffer |
| CLEN | 6134 | Load Clock Enable Register |
| CLSA | 6135 | Clock Status to AC |
| CLBA | 6136 | Clock Buffer to AC |
| CLCA | 6137 | Clock Counter to AC |

**Table D-34 DK8-EA Line Frequency Clock**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| CLEI | 6131 | Enable Interrupt |
| CLDI | 6132 | Disable Interrupt |
| CLSK | 6133 | Skip on Clock Flag and Clear Flag |

**Table D-35 DK8-EC Crystal Clock**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| CLEI | 6131 | Enable Interrupt |
| CLDI | 6132 | Disable Interrupt |
| CLSK | 6133 | Skip on Clock Flag and Clear Flag |

**Table D-36 KP8-E Power Fail Detect**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| SPL | 6102 | Skip on Power Low |

**Table D-37 DP8-EP Redundancy Check Option**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| RCTV | 6110 | Test VRC and Skip |
| RCRL | 6111 | Read BCC Low |
| RCRH | 6112 | Read BCC High |
| RCCV | 6113 | Compute VRC |
| RCGB | 6114 | Generate BCC |
| RCLC | 6115 | Load Control |
| RCCB | 6116 | Clear BCC Accumulation |

**Table D-38 DR8-E Interprocessor Buffer**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| DBRF | 65x1 | Skip if the receive set to a 1 |
| DBRD | 65x2 | Read incoming data into the AC, clear receive flag |
| DBTF | 65x3 | Skip if the transmit flag is set to a 1 |
| DBTD | 65x4 | Load the AC into the transmit buffer, transmit and set the transmit flag |
| DBEI | 65x5 | Enable the Interrupt Request line |
| DBDI | 65x6 | Disable the Interrupt Request Line |
| DBCD | 65x7 | Clear done flag |

**Table D-39 AD01A 12-Bit Analog-to-Digital Converter**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| ADSF | 6531 | Skip on A/D Done Flag | 2.6 |
| ADRB | 6532 | Read A/D Buffer | 2.6 |
| ADCV | 6534 | Convert Analog Input | 2.6 |
| ADSC | 6535 | Select Multiplexer Channel and Gain | 3.6 |
| ADRC | 6536 | Read A/D Buffer, Clear Flag, and Start Conversion | 3.6 |
| ADSR | 6537 | Select Channel and Gain and Read A/D Buffer | 4.6 |

**Table D-40 AA50 Digital-to-Analog Converter**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| DAC 1 | 6551 | Select DAC 1 | 2.6 |
| DAC 2 | 6552 | Select DAC 2 | 2.6 |
| DAC 3 | 6553 | Select DAC 3 | 3.6 |
| DAC 4 | 6554 | Select DAC 4 | 2.6 |
| DAC 5 | 6555 | Select DAC 5 | 3.6 |
| DAC 6 | 6556 | Select DAC 6 | 3.6 |

**Table D-41 AD8-EA A/D Converter**

| Mnemonic Symbol | Octal Code | Operation |
|---|---|---|
| ADCL | 6530 | Clear Flags |
| ADLM | 6531 | Load Multiplexer |
| ADST | 6532 | Start Conversion |
| ADRB | 6533 | Read A/D Buffer |
| ADSK | 6534 | Skip on A/D Done Flag |
| ADSE | 6535 | Skip on Timing Error |
| ADLE | 6536 | Load Enable Register |
| ADRS | 6537 | Read Status Register |

**Table D-42 AA05A/AA07 Digital-to-Analog Converter and Control**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| DACL | 6551 | Clear DAC Address | 2.6 |
| DALD | 6552 | Load DAC Address | 2.6 |
| DALI | 6562 | Load DAC Input Register | 2.6 |
| DAUP | 6564 | Update All Channels | 2.6 |

**Table D-43 AFC-8 Low-Level Analog Multiplexer**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| ADSG | 6542 | Set Multiplexer Gain | 2.6 |
| ADSA | 6544 | Set Multiplexer Address | 2.6 |
| ADSF | 6531 | Skip on A/D Flag | 2.6 |
| ADRB | 6534 | Read A/D Converter Buffer | 2.6 |
| ADSG | 6542 | Set Multiplexer Gain | 2.6 |
| ADSA | 6544 | Set Multiplexer Address | 2.6 |

**Table D-44 AF04A Guarded Scanning Integrated Digital Voltmeter (IDVM)**

| Mnemonic Symbol | Octal Code | Operation | Time ($\mu$s) |
|---|---|---|---|
| VCNV | 6541 | Select Channel and Convert | 2.6 |
| VSEL | 6542 | Select Range and Gate | 2.6 |
| VINX | 6544 | Index Channel and Convert | 2.6 |
| VSDR | 6561 | Skip on Data Ready | 2.6 |
| VRD | 6562 | Read Data and Clear Flag | 2.6 |
| VBA | 6564 | Byte Advance | 2.6 |
| VSCC | 6571 | Sample Current Channel | 2.6 |

# appendix e

## READ-IN MODE (RIM) LOADER

The RIM Loader is used to load programs punched on RIM format paper tape into core memory. It is stored in core memory locations 7756-7776 ($21_8$ locations), and started at location 7756. There are two versions of the RIM Loader, permitting either the high- or the low-speed reader to be used as an input device. The locations and corresponding instructions for both versions are listed below. Figure E-1 provides instructions for loading the RIM Loader.

**Table E-1 RIM Loader Programs**

| Location | INSTRUCTION | |
|---|---|---|
| | Low-Speed Reader | High-Speed Reader |
| 7756 | 6032 | 6014 |
| 7757 | 6031 | 6011 |
| 7760 | 5357 | 5357 |
| 7761 | 6036 | 6016 |
| 7762 | 7106 | 7106 |
| 7763 | 7006 | 7006 |
| 7764 | 7510 | 7510 |
| 7765 | 5357 | 5374 |
| 7766 | 7006 | 7006 |
| 7767 | 6031 | 6011 |
| 7770 | 5367 | 5367 |
| 7771 | 6034 | 6016 |
| 7772 | 7420 | 7420 |
| 7773 | 3776 | 3776 |
| 7774 | 3376 | 3376 |
| 7775 | 5356 | 5357 |

Note: Location 7776 is used for temporary storage.

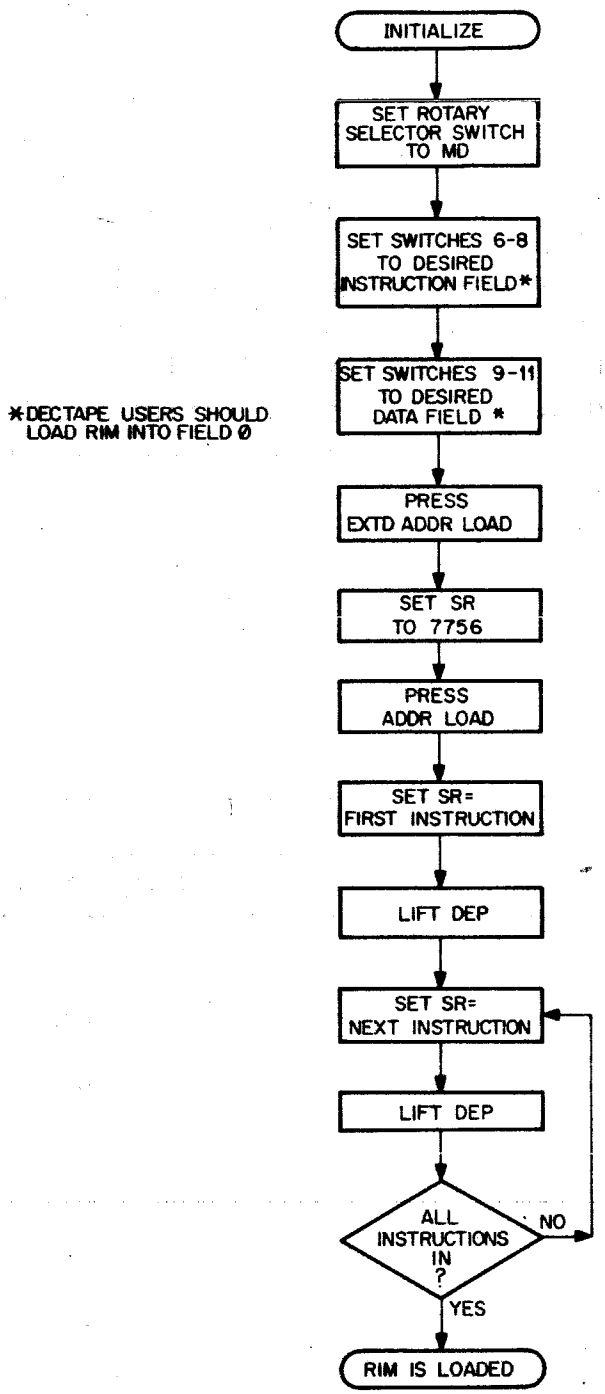



Figure E-1 Loading the RIM Loader

## BINARY (BIN) LOADER

The BIN Loader is used to load programs punched on BIN format paper tape into core memory. It is stored in core memory locations 7625-7752 and 7777 ($127_8$ locations), and started at location 7777. The RIM Loader is usually used to load a RIM format tape of the BIN Loader.

When the BIN Loader is used to load a binary tape, caution must be exercised to ensure that the tape is started with binary leader code (code 200) under the read station. If the tape is started before this code, the contents of core memory may be lost.

Figures E-2 and E-3 provide instructions for loading the BIN Loader by means of the RIM Loader and using the BIN Loader to store a binary program.

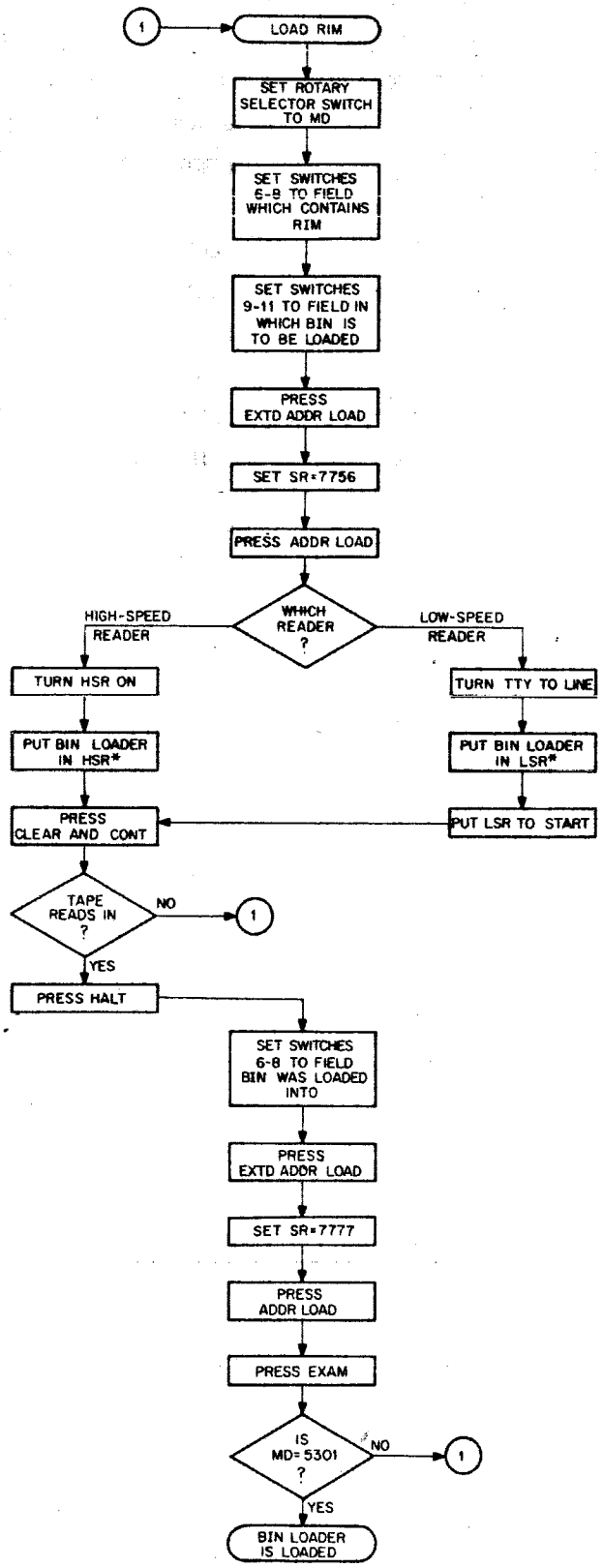



Figure E-2 Loading the BIN Loader

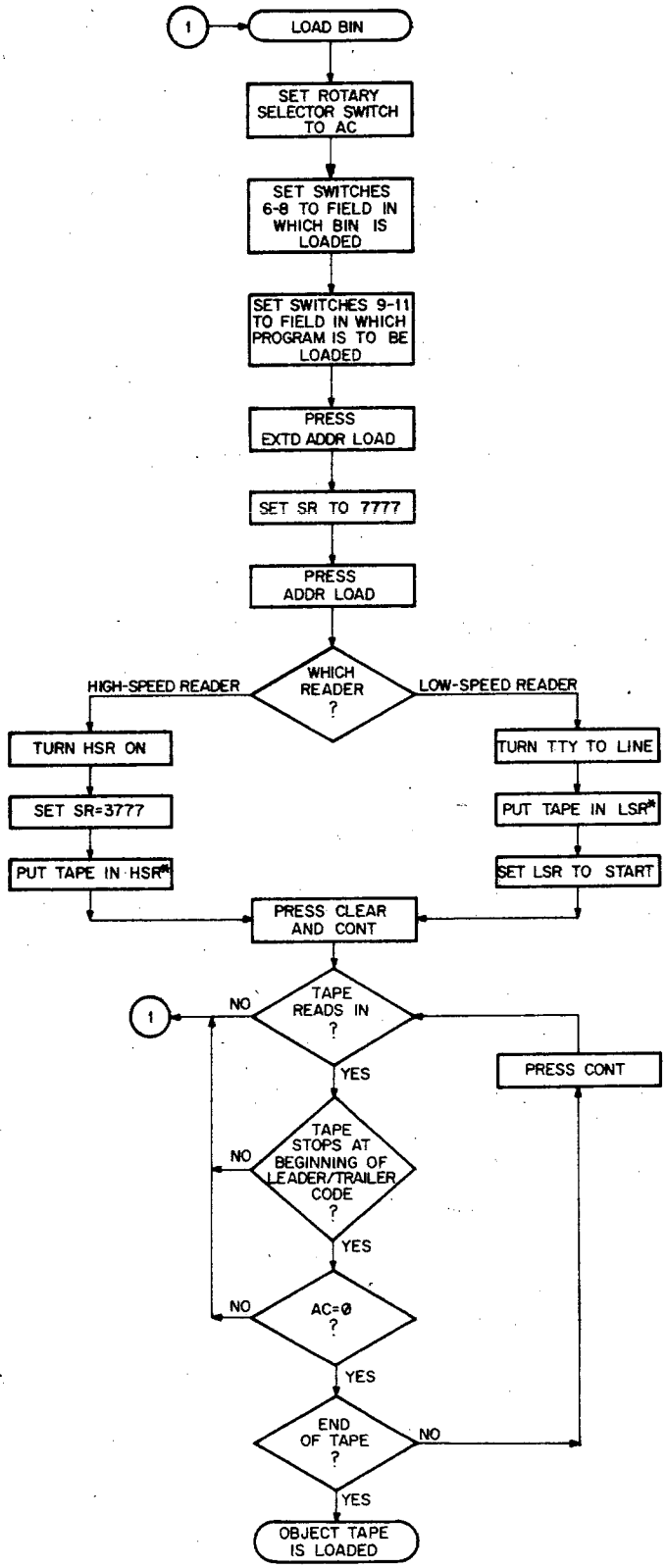



Figure E-3 Using the BIN Loader to Load a Binary Tape

# appendix F

# miscellaneous tables

## Powers of Two

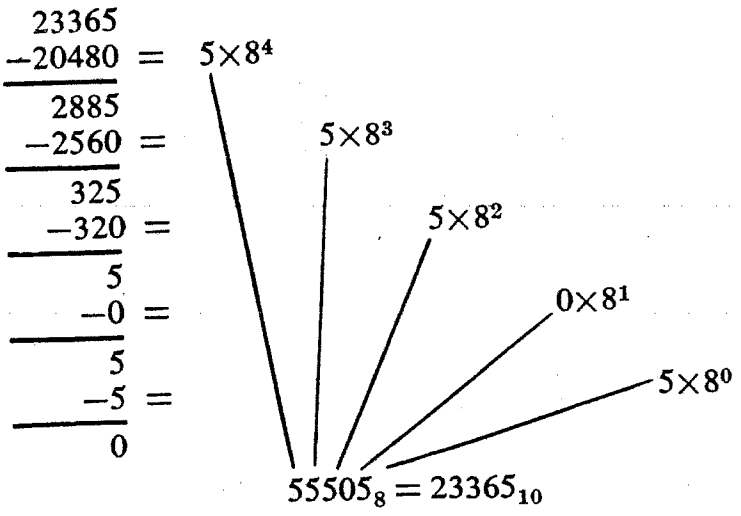
| $2^n$ | n | $2^{-n}$ |
|---|---|---|
| 1 | 0 | 1.0 |
| 2 | 1 | 0.5 |
| 4 | 2 | 0.25 |
| 8 | 3 | 0.125 |
| 16 | 4 | 0.062 5 |
| 32 | 5 | 0.031 25 |
| 64 | 6 | 0.015 625 |
| 128 | 7 | 0.007 812 5 |
| 256 | 8 | 0.003 906 25 |
| 512 | 9 | 0.001 953 125 |
| 1 024 | 10 | 0.000 976 562 5 |
| 2 048 | 11 | 0.000 488 281 25 |
| 4 096 | 12 | 0.000 244 140 625 |
| 8 192 | 13 | 0.000 122 070 312 5 |
| 16 384 | 14 | 0.000 061 035 156 25 |
| 32 768 | 15 | 0.000 030 517 578 125 |
| 65 536 | 16 | 0.000 015 258 789 062 5 |
| 131 072 | 17 | 0.000 007 629 394 531 25 |
| 262 144 | 18 | 0.000 003 814 697 265 625 |
| 524 288 | 19 | 0.000 001 907 348 632 812 5 |
| 1 048 576 | 20 | 0.000 000 953 674 316 406 25 |
| 2 097 152 | 21 | 0.000 000 476 837 158 203 125 |
| 4 194 304 | 22 | 0.000 000 238 418 579 101 562 5 |
| 8 388 608 | 23 | 0.000 000 119 209 289 550 781 25 |
| 16 777 216 | 24 | 0.000 000 059 604 644 775 390 625 |
| 33 554 432 | 25 | 0.000 000 029 802 322 387 695 312 5 |
| 67 108 864 | 26 | 0.000 000 014 901 161 193 847 656 25 |
| 134 217 728 | 27 | 0.000 000 007 450 580 596 923 828 125 |
| 268 435 456 | 28 | 0.000 000 003 725 290 298 461 914 062 5 |
| 536 870 912 | 29 | 0.000 000 001 862 645 149 230 957 031 25 |

**Octal-Decimal Conversion**

The following table gives the multiples of the powers of 8. To convert a number from octal to decimal using the table, add the decimal number opposite the digit value for each digit position. To convert $40277_8$ to decimal, the following numbers are obtained from the table and added.

| Position | Digit | Table entry |
|---|---|---|
| 5 | 4 | 16384 |
| 4 | 0 | 0 |
| 3 | 2 | 128 |
| 2 | 7 | 56 |
| 1 | 7 | 7 |
| | | $16575_{10} = 40277_8$ |

This process is reversed to convert a number from decimal to octal. Subtract out the largest table entry which allows a positive remainder, then take the column number (position coefficient) of the table entry as the Nth digit of the result, where N is the row number (digit position) of the table entry. Continue this process, operating on the remainder from each step in the next step, until all digits of the result have been found. For example, to convert $23365_{10}$ to an equivalent octal number:



**Octal-Decimal Conversion Table**

| Octal Digit Position/ $8^n$ | Position Coefficients (Multipliers) | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 1st ($8^0$) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 2nd ($8^1$) | 0 | 8 | 16 | 24 | 32 | 40 | 48 | 56 |
| 3rd ($8^2$) | 0 | 64 | 128 | 192 | 256 | 320 | 384 | 448 |
| 4th ($8^3$) | 0 | 512 | 1,024 | 1,536 | 2,048 | 2,560 | 3,072 | 3,584 |
| 5th ($8^4$) | 0 | 4,096 | 8,192 | 12,288 | 16,384 | 20,480 | 24,576 | 28,672 |
| 6th ($8^5$) | 0 | 32,768 | 65,536 | 98,304 | 131,072 | 163,840 | 196,608 | 229,376 |

## Octal-Decimal Fraction Conversion Table

| Octal | Decimal | Octal | Decimal | Octal | Decimal | Octal | Decimal |
|---|---|---|---|---|---|---|---|
| .000 | .000000 | .100 | .125000 | .200 | .250000 | .300 | .375000 |
| .001 | .001953 | .101 | .126953 | .201 | .251953 | .301 | .376953 |
| .002 | .003906 | .102 | .128906 | .202 | .253906 | .302 | .378906 |
| .003 | .005859 | .103 | .130859 | .203 | .255859 | .303 | .380859 |
| .004 | .007812 | .104 | .132812 | .204 | .257812 | .304 | .382812 |
| .005 | .009765 | .105 | .134765 | .205 | .259765 | .305 | .384765 |
| .006 | .011718 | .106 | .136718 | .206 | .261718 | .306 | .386718 |
| .007 | .013671 | .107 | .138671 | .207 | .263671 | .307 | .388671 |
| .010 | .015625 | .110 | .140625 | .210 | .265625 | .310 | .390625 |
| .011 | .017578 | .111 | .142578 | .211 | .267578 | .311 | .392578 |
| .012 | .019531 | .112 | .144531 | .212 | .269531 | .312 | .394531 |
| .013 | .021484 | .113 | .146484 | .213 | .271484 | .313 | .396484 |
| .014 | .023437 | .114 | .148437 | .214 | .273437 | .314 | .398437 |
| .015 | .025390 | .115 | .150390 | .215 | .275390 | .315 | .400390 |
| .016 | .027343 | .116 | .152343 | .216 | .277343 | .316 | .402343 |
| .017 | .029296 | .117 | .154296 | .217 | .279296 | .317 | .404296 |
| .020 | .031250 | .120 | .156250 | .220 | .281250 | .320 | .406250 |
| .021 | .033203 | .121 | .158203 | .221 | .283203 | .321 | .408203 |
| .022 | .035156 | .122 | .160156 | .222 | .285156 | .322 | .410156 |
| .023 | .037109 | .123 | .162109 | .223 | .287109 | .323 | .412109 |
| .024 | .039062 | .124 | .164062 | .224 | .289062 | .324 | .414062 |
| .025 | .041015 | .125 | .166015 | .225 | .291015 | .325 | .416015 |
| .026 | .042968 | .126 | .167968 | .226 | .292968 | .326 | .417968 |
| .027 | .044921 | .127 | .169921 | .227 | .294921 | .327 | .419921 |
| .030 | .046875 | .130 | .171875 | .230 | .296875 | .330 | .421875 |
| .031 | .048828 | .131 | .173828 | .231 | .298828 | .331 | .423828 |
| .032 | .050781 | .132 | .175781 | .232 | .300781 | .332 | .425781 |
| .033 | .052734 | .133 | .177734 | .233 | .302734 | .333 | .427734 |
| .034 | .054687 | .134 | .179687 | .234 | .304687 | .334 | .429687 |
| .035 | .056640 | .135 | .181640 | .235 | .306640 | .335 | .431640 |
| .036 | .058593 | .136 | .183593 | .236 | .308593 | .336 | .433593 |
| .037 | .060546 | .137 | .185546 | .237 | .310546 | .337 | .435546 |
| .040 | .062500 | .140 | .187500 | .240 | .312500 | .340 | .437500 |
| .041 | .064453 | .141 | .189453 | .241 | .314453 | .341 | .439453 |
| .042 | .066406 | .142 | .191406 | .242 | .316406 | .342 | .441406 |
| .043 | .068359 | .143 | .193359 | .243 | .318359 | .343 | .443359 |
| .044 | .070312 | .144 | .195312 | .244 | .320312 | .344 | .445312 |
| .045 | .072265 | .145 | .197265 | .245 | .322265 | .345 | .447265 |
| .046 | .074218 | .146 | .199218 | .246 | .324218 | .346 | .449218 |
| .047 | .076171 | .147 | .201171 | .247 | .326171 | .347 | .451171 |
| .050 | .078125 | .150 | .203125 | .250 | .328125 | .350 | .453125 |
| .051 | .080078 | .151 | .205078 | .251 | .330078 | .351 | .455078 |
| .052 | .082031 | .152 | .207031 | .252 | .332031 | .352 | .457031 |
| .053 | .083984 | .153 | .208984 | .253 | .333984 | .353 | .458984 |
| .054 | .085937 | .154 | .210937 | .254 | .335937 | .354 | .460937 |
| .055 | .087890 | .155 | .212890 | .255 | .337890 | .355 | .462890 |
| .056 | .089843 | .156 | .214843 | .256 | .339843 | .356 | .464843 |
| .057 | .091796 | .157 | .216796 | .257 | .341796 | .357 | .466796 |
| .060 | .093750 | .160 | .218750 | .260 | .343750 | .360 | .468750 |
| .061 | .095703 | .161 | .220703 | .261 | .345703 | .361 | .470703 |
| .062 | .097656 | .162 | .222656 | .262 | .347656 | .362 | .472656 |
| .063 | .099609 | .163 | .224609 | .263 | .349609 | .363 | .474609 |
| .064 | .101562 | .164 | .226562 | .264 | .351562 | .364 | .476562 |
| .065 | .103515 | .165 | .228515 | .265 | .353515 | .365 | .478515 |
| .066 | .105468 | .166 | .230469 | .266 | .355468 | .366 | .480468 |
| .067 | .107421 | .167 | .232421 | .267 | .357421 | .367 | .482421 |
| .070 | .109375 | .170 | .234375 | .270 | .359375 | .370 | .484375 |
| .071 | .111328 | .171 | .236328 | .271 | .361328 | .371 | .486328 |
| .072 | .113281 | .172 | .238281 | .272 | .363281 | .372 | .488281 |
| .073 | .115234 | .173 | .240234 | .273 | .365234 | .373 | .490234 |
| .074 | .117187 | .174 | .242187 | .274 | .367187 | .374 | .492187 |
| .075 | .119140 | .175 | .244140 | .275 | .369140 | .375 | .494140 |
| .076 | .121093 | .176 | .246093 | .276 | .371093 | .376 | .496093 |
| .077 | .123046 | .177 | .248046 | .277 | .373046 | .377 | .498046 |

# Scales of Notation

## $2^x$ in Decimal

| x | $2^x$ | x | $2^x$ | x | $2^x$ |
|---|---|---|---|---|---|
| 0.001 | 1.00069 33874 62581 | 0.01 | 1.00695 55500 56719 | 0.1 | 1.07177 34625 36293 |
| 0.002 | 1.00138 72557 11335 | 0.02 | 1.01395 94797 90029 | 0.2 | 1.14869 83549 97035 |
| 0.003 | 1.00208 16050 79633 | 0.03 | 1.02101 21257 07193 | 0.3 | 1.23114 44133 44916 |
| 0.004 | 1.00277 64359 01078 | 0.04 | 1.02811 38266 56067 | 0.4 | 1.31950 79107 72894 |
| 0.005 | 1.00347 17485 09503 | 0.05 | 1.03526 49238 41377 | 0.5 | 1.41421 35623 73095 |
| 0.006 | 1.00416 75432 38973 | 0.06 | 1.04246 57608 41121 | 0.6 | 1.51571 65665 10398 |
| 0.007 | 1.00486 38204 23785 | 0.07 | 1.04971 66836 23067 | 0.7 | 1.62450 47927 12471 |
| 0.008 | 1.00556 05803 98468 | 0.08 | 1.05701 80405 61380 | 0.8 | 1.74110 11265 92248 |
| 0.009 | 1.00625 78234 97782 | 0.09 | 1.06437 01824 53360 | 0.9 | 1.86606 59830 73615 |

## $10^{\pm n}$ in Octal

| $10^n$ | n | $10^{-n}$ | $10^n$ | n | $10^{-n}$ |
|---|---|---|---|---|---|
| 1 | 0 | 1.000 000 000 000 000 000 00 | 112 402 762 000 | 10 | 0.000 000 000 006 676 337 66 |
| 12 | 1 | 0.063 146 314 631 463 146 31 | 1 351 035 564 000 | 11 | 0.000 000 000 000 537 657 77 |
| 144 | 2 | 0.005 075 341 217 270 243 66 | 16 432 451 210 000 | 12 | 0.000 000 000 000 043 136 32 |
| 1 750 | 3 | 0.000 406 111 564 570 651 77 | 221 411 634 520 000 | 13 | 0.000 000 000 000 003 411 35 |
| 23 420 | 4 | 0.000 032 155 613 530 704 15 | 2 657 142 036 440 000 | 14 | 0.000 000 000 000 000 264 11 |
| 303 240 | 5 | 0.000 002 476 132 610 706 64 | 34 327 724 461 500 000 | 15 | 0.000 000 000 000 000 022 01 |
| 3 641 100 | 6 | 0.000 000 206 157 364 055 37 | 434 157 115 760 200 000 | 16 | 0.000 000 000 000 000 001 63 |
| 46 113 200 | 7 | 0.000 000 015 327 745 152 75 | 5 432 127 413 542 400 000 | 17 | 0.000 000 000 000 000 000 14 |
| 575 360 400 | 8 | 0.000 000 001 257 143 561 06 | 67 405 553 164 731 000 000 | 18 | 0.000 000 000 000 000 000 01 |
| 7 346 545 000 | 9 | 0.000 000 000 104 560 276 41 | | | |

## $n \log_{10} 2$, $n \log_2 10$ in Decimal

| n | $n \log_{10} 2$ | $n \log_2 10$ | n | $n \log_{10} 2$ | $n \log_2 10$ |
|---|---|---|---|---|---|
| 1 | 0.30102 99957 | 3.32192 80949 | 6 | 1.80617 99740 | 19.93156 85693 |
| 2 | 0.60205 99913 | 6.64385 61898 | 7 | 2.10720 99696 | 23.25349 66642 |
| 3 | 0.90308 99870 | 9.96578 42847 | 8 | 2.40823 99653 | 26.57542 47591 |
| 4 | 1.20411 99827 | 13.28771 23795 | 9 | 2.70926 99610 | 29.89735 28540 |
| 5 | 1.50514 99783 | 16.60964 04744 | 10 | 3.01029 99566 | 33.21928 09489 |

## Addition and Multiplication Tables

Addition | Multiplication

Binary Scale

$$0+0=0 \qquad 0+1=1+0=1 \qquad 1+1=10$$

$$0\times 0=0 \qquad 0\times 1=1\times 0=0 \qquad 1\times 1=1$$

Octal Scale

| 0 | 01 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|---|
| 1 | 02 | 03 | 04 | 05 | 06 | 07 | 10 |
| 2 | 03 | 04 | 05 | 06 | 07 | 10 | 11 |
| 3 | 04 | 05 | 06 | 07 | 10 | 11 | 12 |
| 4 | 05 | 06 | 07 | 10 | 11 | 12 | 13 |
| 5 | 06 | 07 | 10 | 11 | 12 | 13 | 14 |
| 6 | 07 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |

| 1 | 02 | 03 | 04 | 05 | 06 | 07 |
|---|---|---|---|---|---|---|
| 2 | 04 | 06 | 10 | 12 | 14 | 16 |
| 3 | 06 | 11 | 14 | 17 | 22 | 25 |
| 4 | 10 | 14 | 20 | 24 | 30 | 34 |
| 5 | 12 | 17 | 24 | 31 | 36 | 43 |
| 6 | 14 | 22 | 30 | 36 | 44 | 52 |
| 7 | 16 | 25 | 34 | 43 | 52 | 61 |

## Mathematical Constants in Octal Scale

| | | |
|---|---|---|
| $\pi = 3.11037\ 552421_8$ | $e = 2.55760\ 521305_8$ | $\gamma = 0.44742\ 147707_8$ |
| $\pi^{-1} = 0.24276\ 301556_8$ | $e^{-1} = 0.27426\ 530661_8$ | $\ln \gamma = -0.43127\ 233602_8$ |
| $\sqrt{\pi} = 1.61337\ 611067_8$ | $\sqrt{e} = 1.51411\ 230704_8$ | $\log_2 \gamma = -0.62573\ 030645_8$ |
| $\ln \pi = 1.11206\ 404435_8$ | $\log_{10} e = 0.33626\ 754251_8$ | $\sqrt{2} = 1.32404\ 746320_8$ |
| $\log_2 \pi = 1.51544\ 163223_8$ | $\log_2 e = 1.34252\ 166245_8$ | $\ln 2 = 0.54271\ 027760_8$ |
| $\sqrt{10} = 3.12305\ 407267_8$ | $\log_2 10 = 3.24464\ 741136_8$ | $\ln 10 = 2.23273\ 067355_8$ |

# appendix g
# digital equipment computer users society

**OBJECTIVES**

Digital Equipment Computer Users Society (DECUS) was established in March of 1961 to advance the effective use of Digital Equipment Corporation's computers and peripheral equipment. It is a voluntary, non-profit users group supported by DIGITAL, whose objectives are to:

- advance the art of computation through mutual education and interchange of ideas and information,
- establish standards and provide channels to facilitate the free exchange of computer programs among members, and
- provide feedback to the manufacturer on equipment and programming needs.

The Society sponsors technical symposia; twice a year (Spring and Fall) in the U.S., once a year in Europe, Canada, and Australia. It maintains a Program Library, publishes a library catalog, proceedings of symposia, and a periodic newsletter: DECUSCOPE.

A DECUS-Europe organization was formed in 1970 to assist in the servicing of European members.

## DECUS PROGRAM LIBRARY

The DECUS Program Library is one of the major activities of the users group. It is maintained and operated separately from the DIGITAL library and contains programs contributed by users. Programs are available for all DIGITAL computer lines. The Library contains many types of programs, such as executive routines, editors, debuggers, special functions, games, maintenance and various other classes of programs. Library Catalogs are issued which list all programs available from DECUS.

There are submission standards which programs must meet before they are accepted into the Library. Review procedures determine whether the program remains in the Library, is changed, or is removed.

Forms and information for submitting programs to the Library may be obtained from a DECUS office.

Programs are available to all members on a request basis. Requests for programs should be made on DECUS Library Request forms and directed to the DECUS Program Library. In most cases, a nominal service charge will be associated with a program. Information on program charges is published in the Library Catalog. European members may forward requests through the European DECUS office in Geneva.

As of December 1972, the Library contained approximately 1,600 programs.

## DECUSCOPE

DECUSCOPE is the Society's technical newsletter, published since April, 1962. The aim of this informal news "scope" is to facilitate the interchange of information. Society members are invited to submit ideas, programming notes, letters, and application notes for publication. DECUSCOPE is mailed as issued to all members. Circulation reached 16,500 in 1972.

All submissions to DECUSCOPE should be sent to the Editor, DECUSCOPE, at the DECUS Office in Maynard or Geneva.

## ACTIVITIES

Two nation-wide symposia are held each year—one in the spring and the other in the fall. Seminars are also held annually in Europe, Canada, and Australia. The proceedings and papers presented at the symposia and seminars are published shortly after each meeting and are sent automatically to meeting attendees and, for a nominal charge, upon request to others.

DECUS sponsored the first workshop meeting of the Joint Users Group of the Association for Computing Machinery in April, 1966, and has actively participated in workshops held each year since. The purpose of the Joint Users Group meetings is to establish means for intercommunication among user groups.

DECUS is also a member of the Joint User Group Library Catalog Project sponsored by JUG. This catalog contains lists of programs available from several major user groups. Members of the participating user groups will be eligible to request program documentation from other groups through their Program Interchange Chairman, i.e., for DECUS members, the DECUS Executive Director. Specific details on this interchange program are available from the DECUS office.

DECUS encourages subgrouping of users with common interests. Local User Groups (LUG's) have formed in many areas for the purpose of providing closer communication between users in a specific field or area. Special Interest Groups (SIG's) have formed for the purpose of providing closer communication between users of a specific product or application area. They provide valuable group feedback to the manufacturer on specific programming and hardware needs. Current Special Interest Groups are: PDP-6/10 Mainframe Group, Education SIG, Physics SIG, Graphics SIG, PS/8 SIG, PDP-8 Newspapers Users Group, LUDEC—Laboratory Users of Digital Equipment Computers, and BATCH Users SIG.

## MEMBERSHIP

Membership in DECUS is voluntary and does not require the payment of dues. Members are invited to take an active interest in the Society by contributing to the program library, to DECUSCOPE, and by participating in its meetings and symposia. There are two types of membership in DECUS: Installation Membership and Individual Membership.

## MEMBERSHIP—DECEMBER 1972

Installation Delegates—7,052 Individual Members—8,243

**Installation Membership**

An organization which has purchased or has on order a computer manufactured by Digital Equipment Corporation is automatically eligible for installation membership in DECUS. Membership status is acquired by submitting a written application to the Executive Director or European Secretary for approval by the DECUS Board.

An organization may appoint one delegate for each DEC computer owned. The delegate should be one who is immediately concerned with the operation of the computer he represents and who is willing to take an active part in DECUS activities. He is entitled to vote on all DECUS policies and during the election of officers.

**Individual Membership**

There are two classes for individual membership:

1. Individuals desiring membership in DECUS who are employed at an installation but are not appointed delegates.
2. Individuals who have a direct interest in DECUS or DEC computers but are not employees of a DECUS installation member.

An individual member is not entitled to vote on DECUS policies or during elections. Written application indicating desire to join must be submitted to the DECUS office for approval by the DECUS Board. There is no limit to the number of individual members that may join from either an installation or a non-installation.

## EXECUTIVE BOARD, POLICIES AND ADMINISTRATION

The Society's policies are formulated by an Executive Board elected by vote of Installation Member delegates.

The Administrative office is located at Digital Equipment Corporation, Maynard, Massachusetts, 01754, and all correspondence should be directed to the attention of the DECUS Executive Director.

The European Regional Administrative office is located at Digital Equipment S.A., 81 Route de l'Aire, CH-1211 Geneva 26, Switzerland.

## DECUS PROGRAM LIBRARY CATALOG

The DECUS Program Library Catalog contains detailed information on programs available from the DECUS Library. Catalogs are divided according to computer line. A member must request the catalog of his interest. Periodic updates are sent automatically.

# index/glossary

F

G



H



I

## N

Q

T

## NOTES

DIGITAL EQUIPMENT CORPORATION, Corporate Headquarters: Maynard, Massachusetts 01754, Telephone: (617) 897-5111

SALES AND SERVICE OFFICES

DOMESTIC — ARIZONA, Phoenix and Tucson • CALIFORNIA, Los Angeles, Monrovia, Oakland, Ridgecrest, San Diego, San Francisco (Mountain View), Santa Ana, Sunnyvale and Woodland Hills • COLORADO, Englewood • CONNECTICUT, Fairfield and Meriden • DISTRICT OF COLUMBIA, Washington (Latham, Md.) • FLORIDA, Orlando • GEORGIA, Atlanta • ILLINOIS, Chicago (Rolling Meadows) • INDIANA, Indianapolis • IOWA, Bettendorf • KENTUCKY, Louisville • LOUISIANA, Metairie (New Orleans) • MASSACHUSETTS, Marlborough and Waltham • MICHIGAN, Detroit (Farmington Hills) • MINNESOTA, Minneapolis • MISSOURI, Kansas City and St. Louis • NEW HAMPSHIRE, Manchester • NEW JERSEY, Fairfield, Metuchen and Princeton • NEW MEXICO, Albuquerque • NEW YORK, Albany, Huntington Station, Manhattan, Rochester and Syracuse • NORTH CAROLINA, Durham/Chapel Hill • OHIO, Cleveland, Columbus and Dayton • OKLAHOMA, Tulsa • OREGON, Portland • PENNSYLVANIA, Philadelphia (Bluebell) and Pittsburgh • TENNESSEE, Knoxville • TEXAS, Austin, Dallas and Houston • UTAH, Salt Lake City • WASHINGTON, Bellevue • WISCONSIN, Milwaukee (Brookfield) •

INTERNATIONAL — ARGENTINA, Buenos Aires • AUSTRALIA, Adelaide, Brisbane, Canberra, Melbourne, Perth and Sydney • AUSTRIA, Vienna • BELGIUM, Brussels • BOLIVIA, La Paz • BRAZIL, Puerto Alegre, Rio de Janeiro and São Paulo • CANADA, Calgary, Halifax, Montreal, Ottawa, Toronto and Vancouver • CHILE, Santiago • DENMARK, Copenhagen • FINLAND, Helsinki • FRANCE, Grenoble and Paris • GERMANY, Berlin, Cologne, Hannover, Hamburg, Frankfurt, Munich and Stuttgart • HONG KONG • INDIA, Bombay • INDONESIA, Djakarta • ISRAEL, Tel Aviv • ITALY, Milan and Turin • JAPAN, Osaka and Tokyo • MALAYSIA, Kuala Lumpur • MEXICO, Mexico City • NETHERLANDS, Utrecht • NEW ZEALAND, Auckland • NORWAY, Oslo • PHILIPPINES, Manilla • PUERTO RICO, Santurce • SINGAPORE • SPAIN, Barcelona and Madrid • SWEDEN, Gothenburg and Stockholm • SWITZERLAND, Geneva and Zurich • TAIWAN, Taipei and Taoyuan • UNITED KINGDOM, Birmingham, Bristol, Dublin, Edinburgh, Leeds, London, Manchester and Reading • VENEZUELA, Caracas • YUGOSLAVIA, Ljubljana •