INTRODUCTION TO PROGRAMMING

PDP-8 Family Computers

Prepared by
The Software Writing Group
Programming Department
Digital Equipment Corporation

SMALL COMPUTER HANDBOOK SERIES

First Printing, January 1969
Second Printing, July 1969

Foreword

The data processing industry has expanded so rapidly during the past twenty years that there has always been a severe shortage of trained personnel. Many new job openings are created each year because new ways for using the computer are continually being developed. Consequently, the computer industry has a real problem training programmers and engineers fast enough to fill these new jobs.

Fortunately, this situation is improving, primarily because our universities, high schools and training schools have greatly expanded their abilities and facilities to train students in data processing. We are especially heartened by the great strides being made at the secondary school level. Computers are now an integral part of modern life; high school students are using computers to learn to solve Algebra I problems; engineers are using the computer as an "electronic slide rule"; chemists are using the computer to aid in analysis and control of chemical processes; many businesses are using computers to process payroll records, control inventories, and many other applications.

We anticipate that this book will be useful to both teachers and students as a training text and reference handbook. For users of our small computers, it will also be the basic programming reference source for use in conjunction with the many small computer systems of the PDP-8 family produced by DEC.

Introduction to Programming was prepared by the Software Writing Group in the Programming Department at DEC, with the assistance of many others, including instructors in our Training Department, many DEC and user programmers who have reviewed the manuscript, and teachers who are presently using a PDP-8 in their computer sciences courses.

We are most grateful to everyone who has contributed.

Kenneth H. Olsen
President
Digital Equipment Corp.

Preface

This textbook is an introduction to programming digital computers, particularly Digital Equipment Corporation's PDP-8 family of computers (often referred to simply as PDP-8[1]). Over 3,000 of these small, general-purpose computers have proven their versatility in hundreds of different applications.

Introduction to Programming can be used by students, programming trainees, and experienced programmers. It offers two approaches to learning computer programming: (1) learning to program in machine language, that is, learning to write programs using the actual instructions that the computer was built to perform and (2) learning to program in a common programming language that uses many English words and standard mathematical notation. Although easier to learn, common languages, such as FORTRAN, ALGOL or FOCAL, are not as efficient as machine languages, in terms of program execution times and core memory requirements. Of course, machine language and common language programming are complementary, so it is useful to learn both.

Teachers and programming students will find Chapters 1 through 5 useful as an introductory text to machine language programming. Chapters 6 through 9 are devoted to the descriptions, uses, and operating procedures for PDP-8 system software; this software has proven its ability to simplify the tasks of writing, editing, assembling, compiling, debugging, and running user programs.

Chapter 6 describes PDP-8 system software and provides detailed operating procedures for the Symbolic Editor, the assemblers, and other commonly used software. These operating procedures can be used by the computer operator or programmer independent of the remainder of the book.

Chapters 7 and 8 describe the Disk Monitor System and the TSS/8 Time-Sharing System.

---

[1] PDP stands for *Programmed Data Processors* and is a trade mark of Digital Equipment Corporation.

Chapter 9 is a complete student's text on the use of FOCAL (FOrmula CALculator), a conversational interpreter for solving numerical problems. FOCAL language consists of short, easy-to-learn, imperative English statements. FOCAL puts the full calculating power and speed of the computer at the user's fingertips, providing an easy way of simulating mathematical models, plotting curves, handling sets of simultaneous equations in n-dimensional arrays, and much more.

Scientific programming is explained in Chapter 10 along with a detailed special program designed to gather physiological data.

After becoming familiar with PDP-8 programming, the user may wish to join DECUS (Digital Equipment Computer Users Society). DECUS is a user's organization that exchanges ideas and programs; it is described in Chapter 11 together with a list of programs available from DECUS.

A detailed index/glossary, a summary of instructions, answers to selected exercises, and tables of conversion codes are included at the back of the book. More experienced programmers will find the book and its index useful as a reference guide.

Preface to the Second Printing

The text is unchanged, except for the correction of minor errors. Most of these errors were reported by diligent readers, to whom we are most grateful. As we expect to improve this book in future revisions, all readers are earnestly requested to send corrections and comments to:

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

Contents

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 and in Chapter 10, 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 and 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 then 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:

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

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

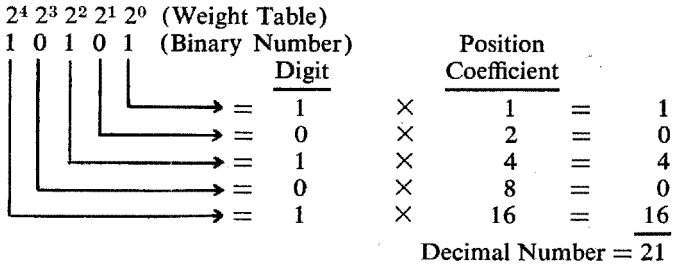
COUNTING IN BINARY NUMBERS

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

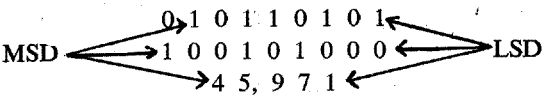
By convention, weighting values are always arranged in the same manner; the highest on the extreme left and the lowest on the extreme right. Therefore, the position coefficient begins at 1 and increases from right to left. This convention has two very practical advantages. The first advantage is that 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

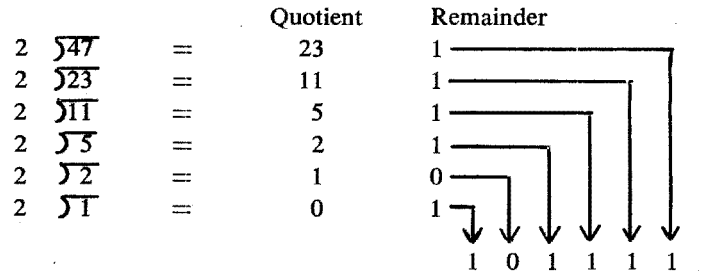
Therefore, $42_{10} = 101010_2$.

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



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

EXERCISES

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

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

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

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

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

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

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

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, $.6_{10}$ (6×10^{-1}) is equivalent to 6 divided by 10^1 or 6/10, and also $.3_8$ (3×8^{-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

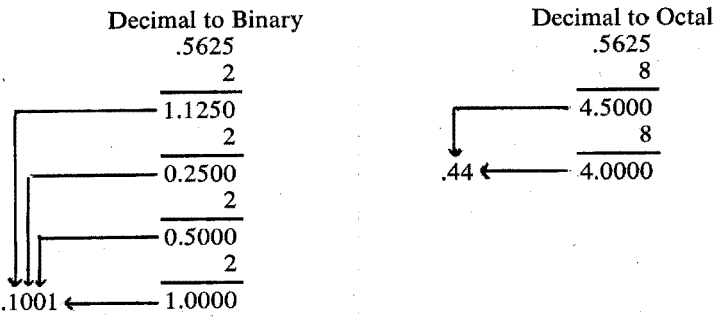
with coefficients of 0 or 1, since they are the only coefficients used in this system. In the octal system, the coefficients 0 through 7 are used. The following example and explanation will show the conversion of the decimal fraction .5625 to binary.

	.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.

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

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

$$
\begin{array}{r}
11.1 \\
100\overline{)1110.0} \\
100 \\
\hline
110 \\
100 \\
\hline
100 \\
100 \\
\hline
0
\end{array}
$$

$11.1_2 = 3.5_{10}$

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.

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

$$
\begin{array}{r}
31 \\
2\overline{)62} \\
6 \\
\hline
02 \\
2 \\
\hline
0
\end{array}
$$

$31_8 = 25_{10}$

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

$$
\begin{array}{r}
66 \\
22\overline{)1714} \\
154 \\
\hline
154 \\
154 \\
\hline
0
\end{array}
$$

EXERCISES

a. Perform the following binary additions.

1. $\begin{array}{r} 10110 \\ +101 \\ \hline \end{array}$

2. $\begin{array}{r} 100 \\ +10 \\ \hline \end{array}$

3. $\begin{array}{r} 11011 \\ +0010 \\ \hline \end{array}$

4. $\begin{array}{r} 10110111 \\ +\quad 1 \\ \hline \end{array}$

5. $\begin{array}{r} 1101 \\ 101 \\ +11 \\ \hline \end{array}$

6. $\begin{array}{r} 101 \\ 1 \\ +110 \\ \hline \end{array}$

7. $\begin{array}{r} 1110 \\ 100 \\ +11 \\ \hline \end{array}$

8. $\begin{array}{r} 1111 \\ 101 \\ +1000 \\ \hline \end{array}$

9. $\begin{array}{r} 110111 \\ 100100 \\ +110001 \\ \hline \end{array}$

10. $\begin{array}{r} 100111 \\ 111001 \\ +101101 \\ \hline \end{array}$

11. $\begin{array}{r} 11011001 \\ 10010011 \\ +11100011 \\ \hline \end{array}$

12. $\begin{array}{r} 11011011 \\ 10111011 \\ 00101011 \\ 01010111 \\ +01111101 \\ \hline \end{array}$

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

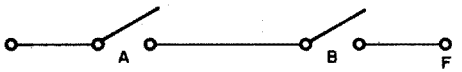
Multiplicand
Multiplier) Product

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

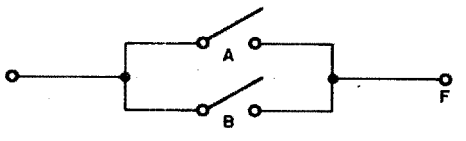
When the AND operation is applied to binary numbers, a binary 1 will appear in the result if a binary 1 appeared in the corresponding position of the two numbers.

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.

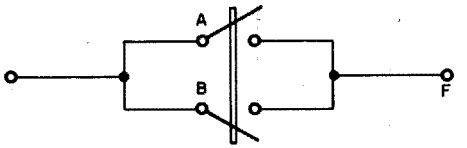
$$\begin{aligned} A &= 011\ 010\ 011\ 111 \\ B &= 100\ 110\ 010\ 011 \\ A+B &= 111\ 110\ 011\ 111 \end{aligned}$$

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.

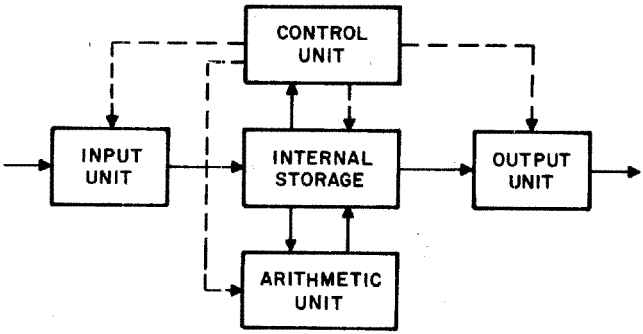


Figure 1-1. PDP-8 General Organization

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.

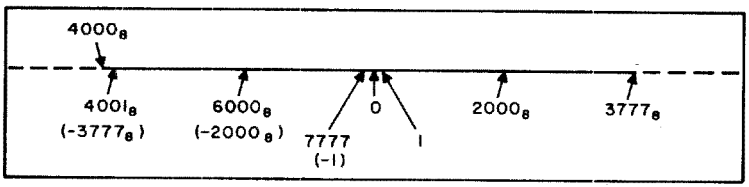


Figure 1-2. PDP-8 Octal Number Line

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.

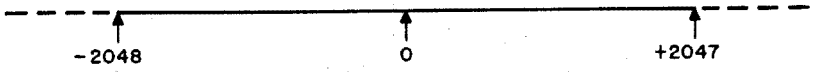


Figure 1-3. PDP-8 Decimal Number Line

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. Further description of the floating point system is contained in Chapter 6.

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.

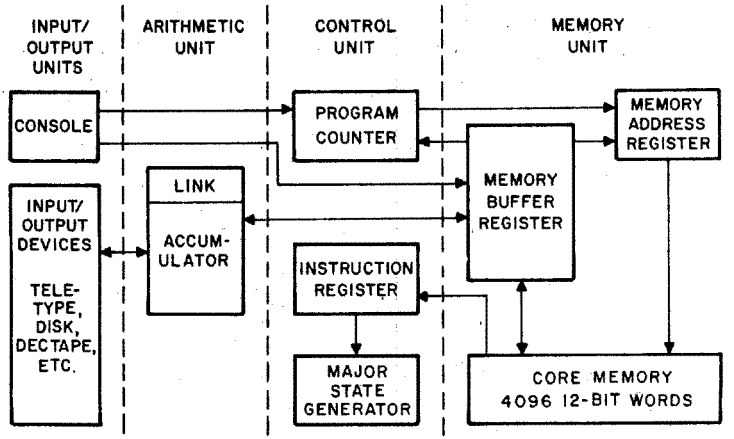


Figure 2-1. Block Diagram of the PDP-8

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.

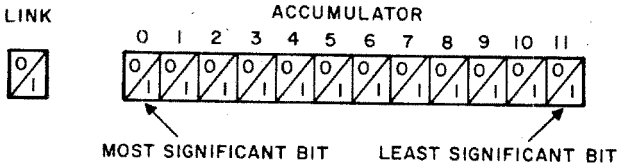
ACCUMULATOR (AC)

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.

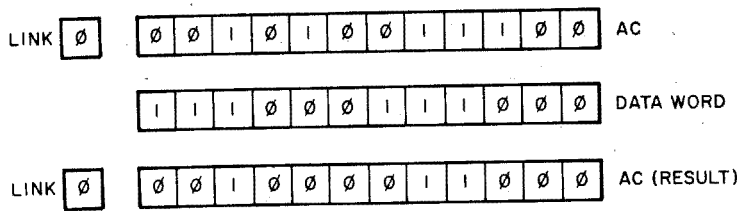
Instruction	Mnemonic[2]	Octal Value	Memory Cycles[1]
Logical AND	AND	0nnn	2
Two's Complement Add	TAD	1nnn	2
Deposit and Clear the Accumulator	DCA	3nnn	2
Jump	JMP	5nnn	1
Increment and Skip if Zero	ISZ	2nnn	2
Jump to Subroutine	JMS	4nnn	2

[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.

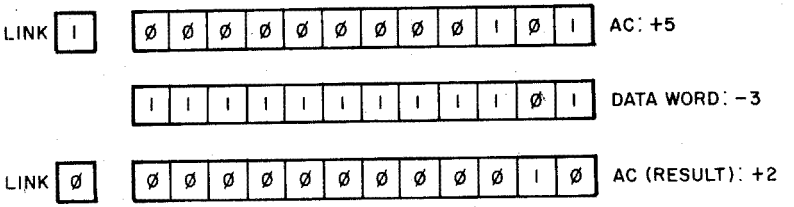


The following points should be noted with respect to the AND instruction:

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.

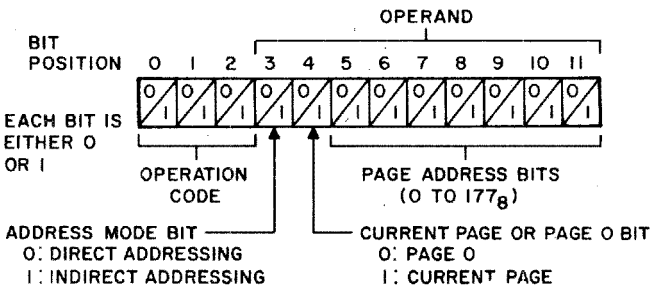


Figure 2-3. Format of a Memory Reference Instruction

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.

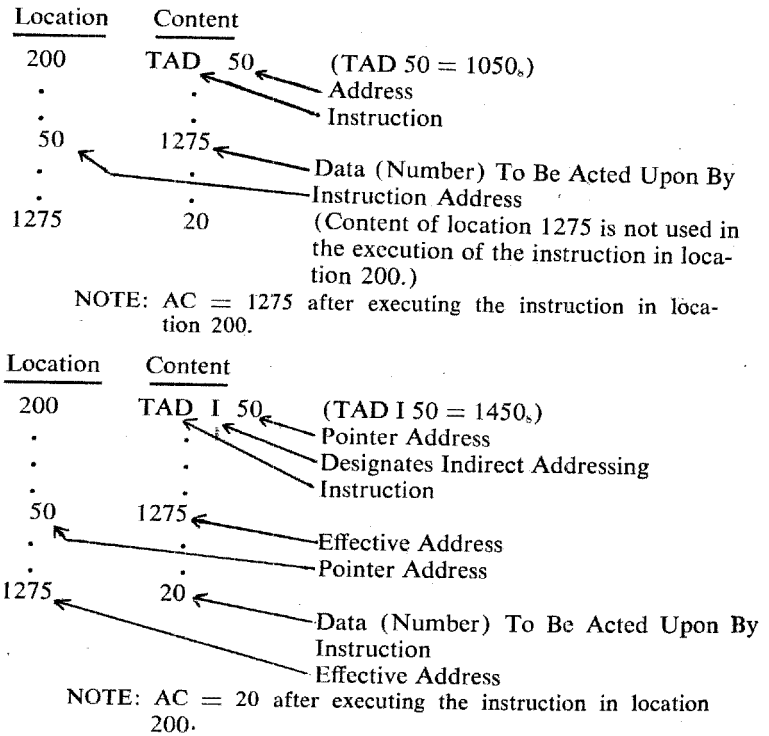
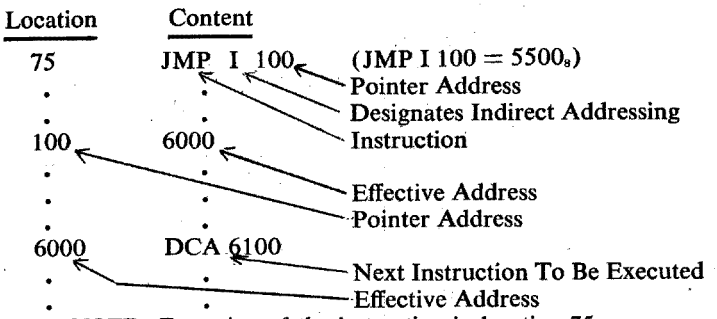


Figure 2-4. Comparison of Direct and Indirect Addressing

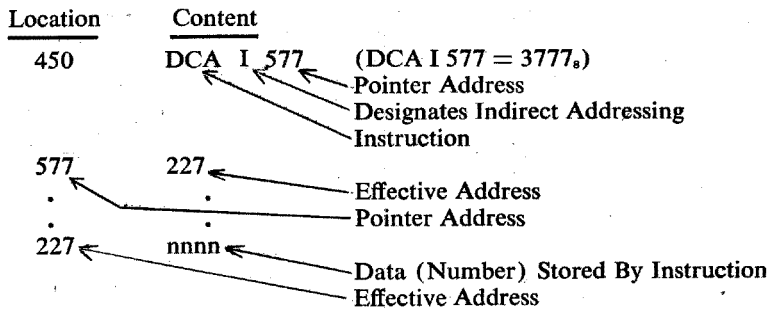
The following three examples illustrate some additional ways in which indirect addressing can be used. As shown in example 1, indirect addressing makes it possible to transfer program control off page 0 (to any desired memory location). (Similarly, indirect addressing makes it possible for other memory 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.)

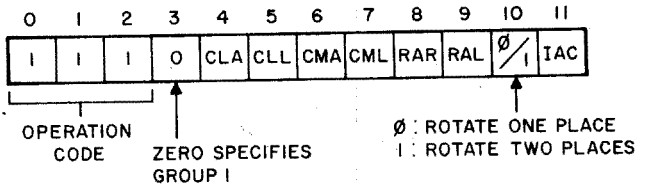
NOTES:
1. Execution of the instruction in location 207 causes the address 210 to be stored in location 2000 and the instruction in location 2001 to be executed next. Execution of the subroutine proceeds until the last instruction (JMP I 2000) causes control to be transferred back to the main program,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

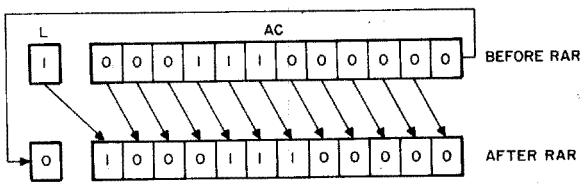
CLA *Clear the accumulator.* If bit 4 is a 1, the instruction sets the accumulator to all zeroes.

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

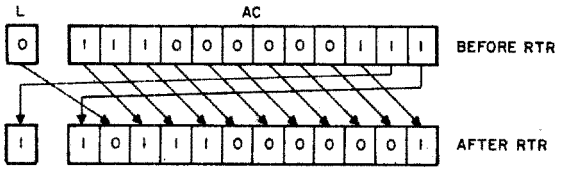
CMA *Complement the accumulator.* If bit 6 is a 1, the accumulator is set to the 1's complement of its original value; that is, all 1's become 0's, and all 0's become 1's.

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

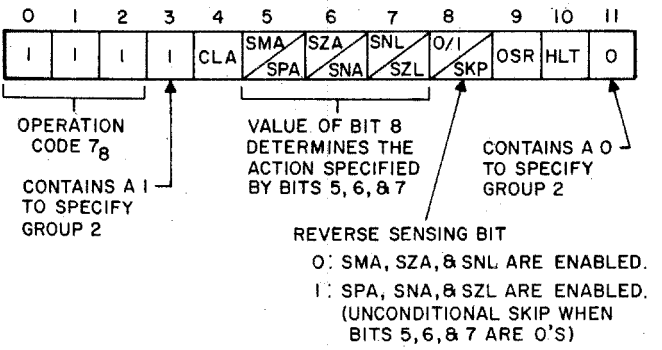
[1] Mnemonic code is meaningful to and translated by an assembler into binary code.

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

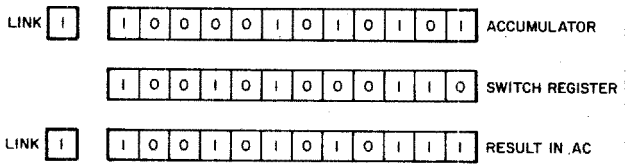
SNA *Skip on nonzero accumulator.* If bit 6 is a 1 and bit 8 is a 1 also, the next instruction is skipped if the accumulator is not zero.

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

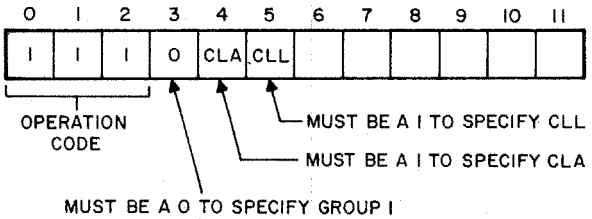
Because PDP-8 instructions of Group 1 and Group 2 are determined by bit assignment, these instructions may be combined, or microprogrammed, to form new instructions enabling the computer to do more operations in less time.

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.

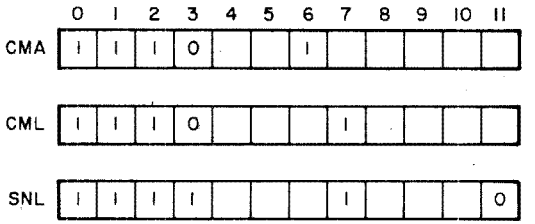
Illegal Combinations

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.
STA	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.

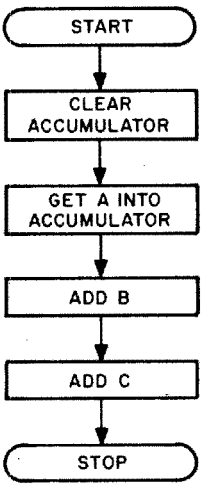
When a more complex problem is to be solved by a computer, the program involves many steps, and writing it often becomes long and confusing. A method for solving a problem which is written in words and mathematical equations is extremely hard to follow, and coding computer instructions from such a document 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

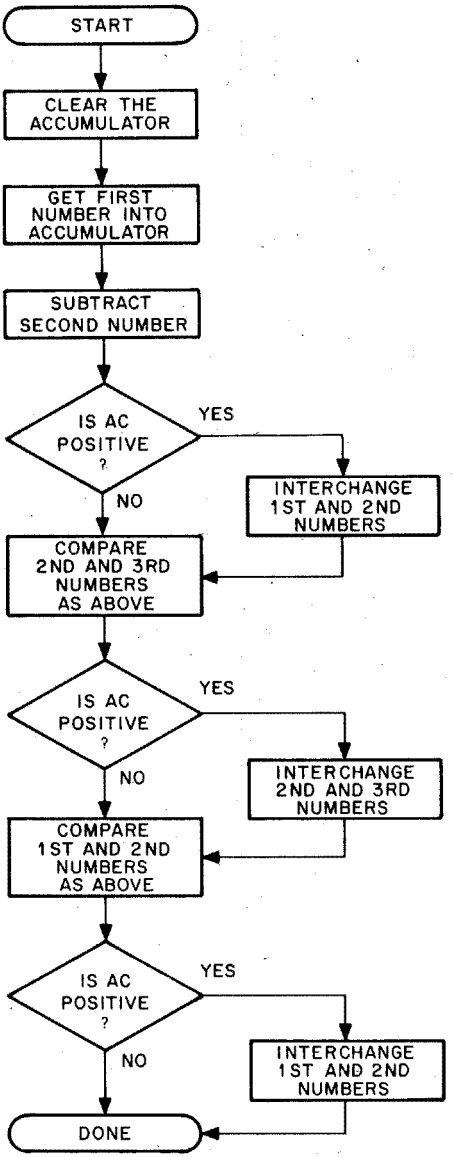
Example 1 is an illustration of straight-line programming. As the flowchart shows, there is a straight-line progression through the processing steps with no change in course. The value of X, which is equal to A+B+C is in the accumulator when the program stops.



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 or TAD A
↑ SPACE ↑ 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)			

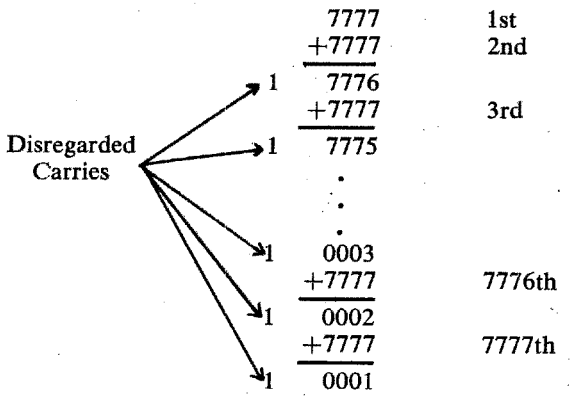
Note that the number to be subtracted (subtrahend) is brought into the accumulator, complemented (1's complement) and incremented by 1 (to form the 2's complement). (The 2's complement could be obtained directly through the one microinstruction CIA.) The number from which A is to be subtracted (minuend) 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 -8×10^6 to 8×10^6. 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 subroutine)
            .
            .
            .
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.

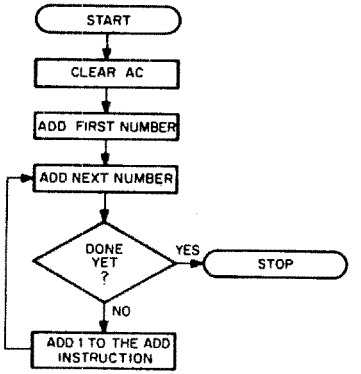
8. An interesting modification of the previous program is achieved by defining a "new operation" MLTPLY by including in the coding the statement MLTPLY=JMS I 30. The assembler would make a replacement such that any time the programmer writes MLTPLY, the computer would perform a jump to the subroutine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

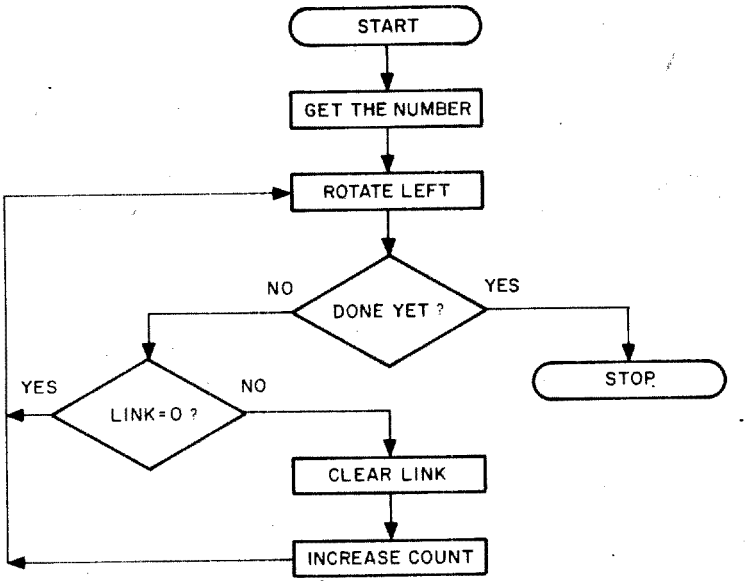
The reader should note that the first eight instructions of the second example are concerned with intializing the program. This intializing enables the stored program to be restarted several times and still operate on the correct locations. If the program had merely incremented locations K4000 and K4200 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
            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

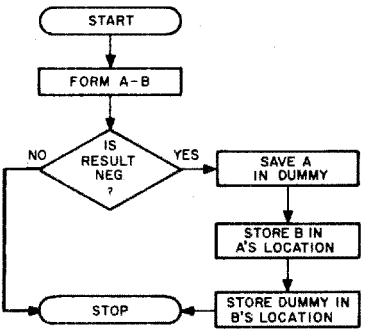
10^{-3} seconds) has occurred. If, as in the example above, the value of CONST is 6030_8, this loop will be entered 1000_{10} times giving a total delay of 18.432 seconds. For any given purpose, a desired delay of from milliseconds to seconds can be obtained precisely by varying the values of CONST and 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.



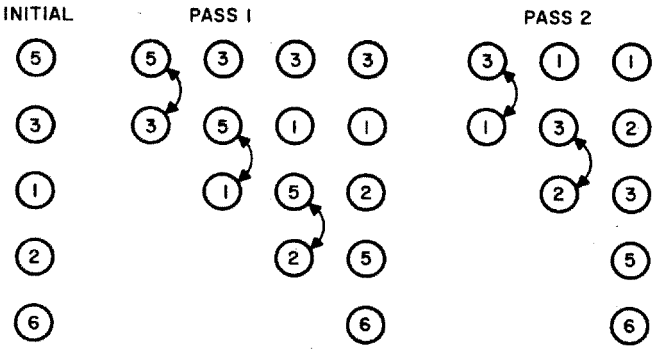
```
*200
TEST,   CLA CLL
        TAD B          /SUBTRACT B
        CIA            /FROM A
        TAD A          /HERE.
        SMA CLA
        HLT            /STOP HERE IF A IS GREATER
                        OR EQUAL

        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.

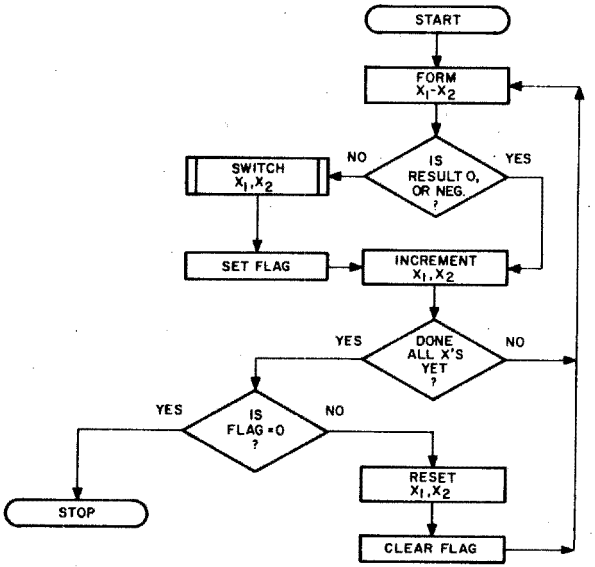
The concept illustrated by the above example can be included in a larger program that will take a set of elements and arrange them in increasing order. The following important concepts should be learned from the example.

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,
B,          } substitute any numbers for A and 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
$
```

Chapter 4

System Description and Operation

The PDP-8 system is composed of the computer console, a Teletype console (usually an Automatic Send Receive Model 33) 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/I and PDP-8/L specifically as representative of the PDP-8 computer family. This chapter also provides an introduction to the more common peripheral devices which may be included in a PDP-8 system. Chapter 5 describes the programmed control of peripheral devices and the means for transferring information between peripheral equipment and the central processor.

CONSOLE OPERATION

The operator console allows manual control of the computer and provides the most elementary means of storing a program in memory. It is a collection of switches and indicator lamps which enable the programmer to examine the contents of locations in memory, alter the contents of memory locations, and determine the current status of a running program.

Console Components

The PDP-8/I operator console is shown in Figure 4-1. The PDP-8/L operator console is significantly different from the PDP-8/I console and is shown in Figure 4-2. The following discussion applies to both console diagrams except where differences are noted. For reference purposes, the switches and indicators are identified in the following tables.

Switch	Explanation
POWER	This key-operated switch controls the computer's primary power supply. This switch has "on" and "off" positions. In the PDP-8/L, however, the switch includes a third position which performs the function of the PANEL LOCK switch, below.
PANEL LOCK	This key-operated switch disables all console switches except the switch register. (This feature is provided in the PDP-8/L by the third position of the POWER switch.)
START	The START switch initiates execution of the computer program beginning in the location currently held in the PROGRAM COUNTER register. The START switch clears the ACCUMULATOR, LINK, MEMORY BUFFER and instruction registers.
LOAD ADD	The LOAD ADD (load address) switch sets the contents of the switch register (SR) into the PROGRAM COUNTER (PC). (The PC is an internal register which is not displayed on the PDP-8/L console.) The computer should be halted before operating this switch.
DEP	The DEP (deposit) switch sets the contents of the switch register into the location currently specified in the PROGRAM COUNTER. Depressing this switch also results in the PROGRAM COUNTER being incremented by 1, and the address of the location previously specified by the PC is displayed in the MEMORY ADDRESS register. Successive operation of the switch stores contents of the switch register in successive locations because the PROGRAM COUNTER is incremented each time the DEP switch is depressed.
EXAM	The EXAM (examine) switch displays the contents of the location specified by the PROGRAM COUNTER in the MEMORY BUFFER and ACCUMULATOR displays. Depressing this switch also results in the PROGRAM COUNTER being incremented by 1; thus, repeated manipulation of the EXAM switch displays the contents of successive memory locations.
CONT	The CONT (continue) switch initiates execution of the stored program at the location specified by the PROGRAM COUNTER. The CONT switch does not clear any active registers. (The START switch does.)

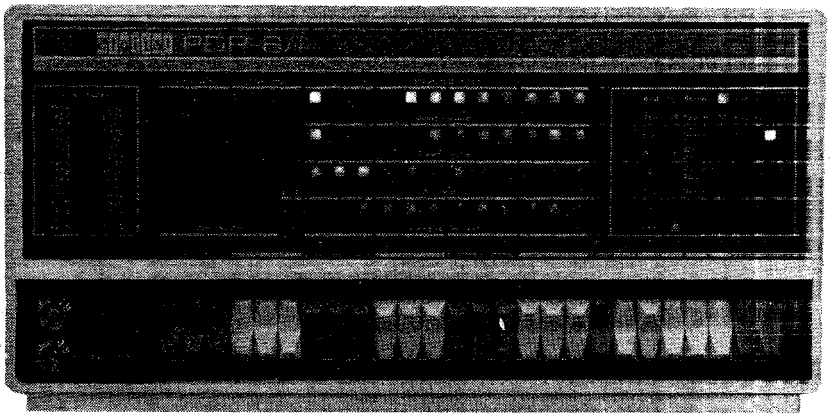


Figure 4-1. PDP-8/I Computer Console

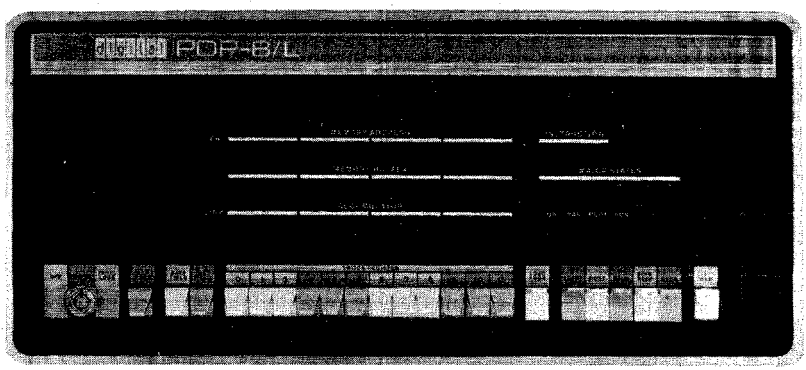


Figure 4-2. PDP-8/L Computer Console

Switch	Explanation
STOP	The STOP switch halts program execution at the end of the instruction in progress.
SING STEP	When the SING STEP (single step) switch is set, the computer executes instructions one memory cycle at a time for each depression of the CONT switch.
SING INST	When the SING INST (single instruction) switch is set, the computer will execute one instruction at a time for each depression of the CONT switch. This switch is not present on the PDP-8/PDP-8/L console.
SWITCH REGISTER (SR)	The SWITCH REGISTER is a set of twelve toggle switches used to specify binary numbers which are loaded into registers when other console switches are operated. LOAD ADD sets the contents of the SR into the PROGRAM COUNTER. DEP sets the contents of the SR into memory through the MEMORY BUFFER register. The twelve positions represent a 12-bit binary word. PDP-8/I — When the top of a switch is out, it represents a binary 1 and is considered set; conversely, when the bottom of the switch is out it represents a binary 0 and is not set. PDP-8/L, PDP-8/S, PDP-8 — When the switch is up, it represents a binary 1 and is considered set; conversely, when the switch is down it represents a binary 0 and is not set.
MEM PROT	This switch is provided on the PDP-8/L only. When set (depressed), the MEM PROT (memory protect) switch prevents the storing or changing of information in the upper 200_8 locations (7600 to 7777) of core memory (the upper 200_8 locations of field 1 in an 8K computer). NOTE: JMS, DCA, ISZ instructions and all input transfers are not permitted, thus protecting programs or data stored in this area.
DATA FIELD (DF) and INST FIELD (IF)	These two console elements (switches and displays) are enabled and used when the basic computer is equipped with extended memory. Their function is described later in this chapter.

Indicator Display	Explanation
PROGRAM COUNTER (PC)	The contents of this 12-bit display represent the address of the next instruction to be executed. This display is not present on the PDP-8/L console.
MEMORY ADDRESS (MA)	Contents represent the address of the word being obtained from or stored in memory. After depressing the DEP or EXAM switches, the contents represent the address of the word previously read or written.
MEMORY BUFFER (MB)	The contents of this 12-bit display represent the word currently being obtained from or stored in memory.
ACCUMULATOR (AC)	This display indicates the current contents of the ACCUMULATOR. After depressing the DEP or EXAM switch, this display indicates the contents of the location whose address is displayed in the MA.
LINK (L)	This display indicates the contents of the LINK, a 1-bit register which serves as an extension of the ACCUMULATOR.
MULTIPLIER QUOTIENT (MQ)	This display is activated by the EAE option described later in this chapter. The option is not available on the PDP-8/L.
Instruction and Status Indicators	These indicators are located in the upper right of the console. The display indicates the status of the program being executed and the operation code of the instruction being executed.

Manual Program Loading

Having written a program, the programmer must store the instructions in memory before they can be executed. The octal value of the instructions may be directly loaded into memory from the computer console. Once the POWER switch has been turned on to energize the

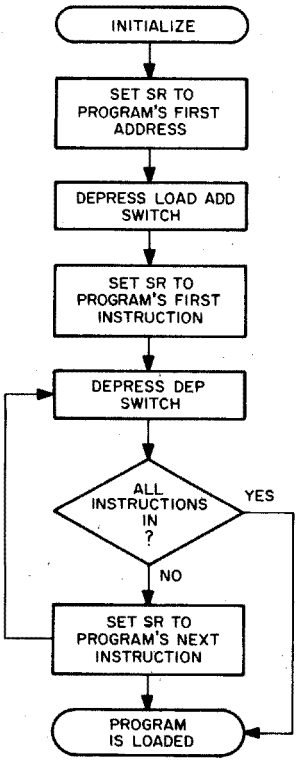
computer, the programmer loads the instructions into memory using the console switches. The location in which the instruction is to be stored is identified by setting the SWITCH REGISTER to the desired address, and then operating the LOAD ADD switch. The address will be placed in the program counter register and displayed in the PROGRAM COUNTER indicator.

To load an instruction or data into the location specified by the PROGRAM COUNTER, a similar procedure is followed. The SWITCH REGISTER is set to the binary representation of the instruction (or data) and then the DEP switch is depressed. The instruction (or data) is displayed in the MB and AC displays and the address of the location is displayed in the MA display. Loading a program into sequential memory locations is simplified by the fact that the DEP switch will automatically increment the program counter register by 1. Thus, once an initial address is specified, instructions and data may be loaded into sequential memory locations by alternately setting the binary representations on the SWITCH REGISTER and operating the DEP switch.

Once loaded into memory, a program may be checked by using the EXAM switch. To check the contents of a location, the desired address is entered on the SWITCH REGISTER and the LOAD ADD switch is depressed. The EXAM switch is used to display the contents of the specified location in the MEMORY BUFFER display. Because operation of the EXAM switch will increment the program counter by 1 (as does the DEP switch), sequential locations may be examined by repeated use of the EXAM switch, once the initial location is specified using the SWITCH REGISTER and LOAD ADD switch. Thus, a stored program may be checked to see that it was correctly loaded by displaying each program location.

To run a stored program, the binary value of the starting address of the program is toggled on the SWITCH REGISTER and then the LOAD ADD is depressed. (The starting address appears in the PC.) Depressing the START switch causes the computer to execute the program beginning with the instruction specified by the address in the PROGRAM COUNTER register. A program may be manually halted by the operation of the STOP switch.

Console switch positioning and procedures for initializing the console are outlined below. Figures 4-3 through 4-5 summarize the procedures for loading, checking, and running a program.



CONSOLE SWITCH POSITIONING

PDP-8/I — When the top of a switch is out, it represents a binary 1 and is considered set; conversely, when the bottom of the switch is out it represents a binary 0 and is not set.

PDP-8/L, PDP-8/S, PDP-8 — When the switch is up it represents a binary 1 and is considered set; conversely, when the switch is down it represents a binary 0 and is not set.

INITIALIZING THE CONSOLE

1. Computer POWER is on.
2. PANEL LOCK is off.
3. SWITCH REGISTER equals zero.
4. Both SING STEP and SING INST are not set.
5. All peripheral devices turned off.

Figure 4-3. Manually Loading a Program

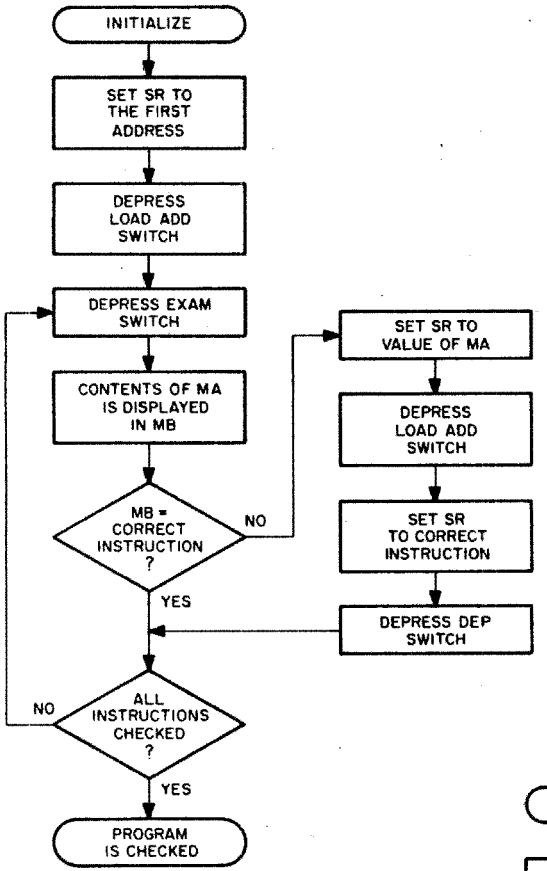


Figure 4-4. Checking a Stored Program

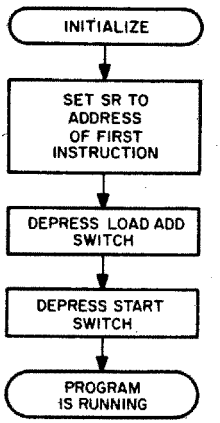


Figure 4-5. Running a Stored Program

TELETYPE OPERATION

The ASR 33 Teletype console is the basic input/output device for the PDP-8 computer family. It consists of a printer, keyboard, paper tape reader, and paper tape punch. The Teletype unit can operate under program control or under manual control. Programmed operation of the Teletype unit is described in detail in Chapter 5. Operation of the Teletype unit as an independent device for generating paper tapes is described later in this section.

Teletype Unit Components

The ASR 33 Teletype unit commonly used in conjunction with the PDP-8 computer family is pictured in Figure 4-6 with the major features noted.

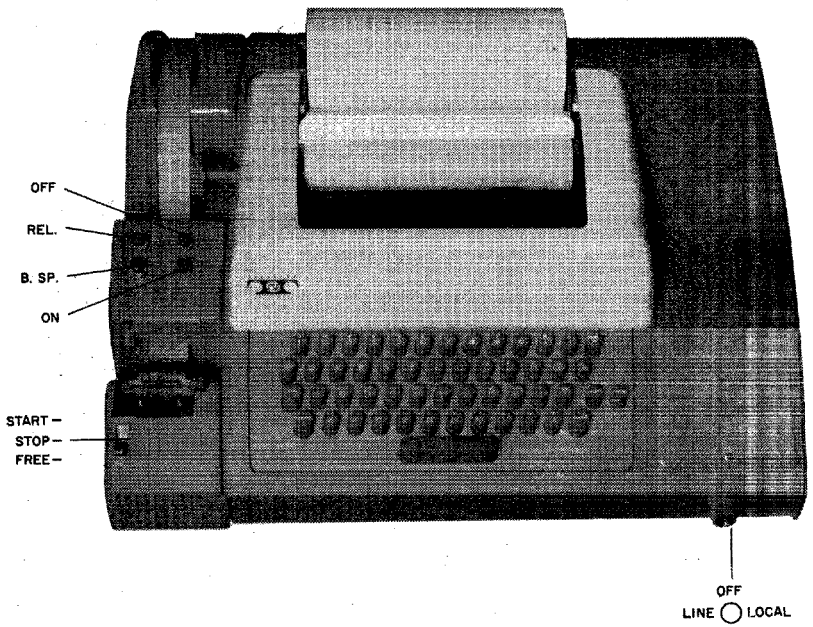


Figure 4-6. ASR 33 Teletype Console

The components of the Teletype unit and their functions are described in the following paragraphs.

CONTROL KNOB

The control knob of the ASR 33 Teletype console (see Figure 4-6) 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.

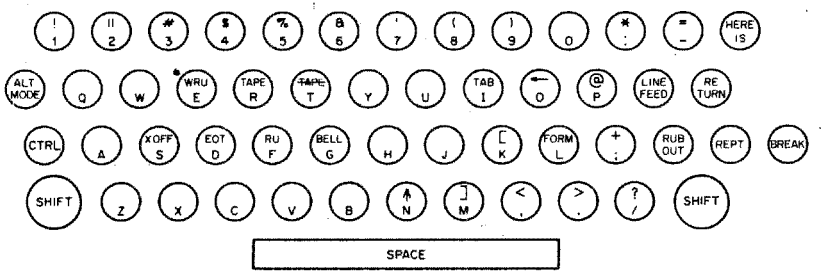


Figure 4-7. Teletype Keyboard

KEYBOARD

The Teletype keyboard shown in Figure 4-7 is similar to a typewriter keyboard, except that some nonprinting characters are included as upper case elements. For typing characters or symbols, such as $, %, #, which appear on the upper portion of numeric keys and certain alphabetic keys, the SHIFT key is held depressed while the desired key is operated.

Designations for certain (nonprinting) operational functions are shown on the upper part of some alphabetic keys. By holding the CTRL (control) key depressed and then depressing the desired key, these functions are activated. Table 4-1 lists several commonly used keys that have special functions in the symbolic language of PDP-8 family computers.

PRINTER

The printer provides a typed copy of input and output at ten characters per second maximum rate. When the Teletype unit is online (LINE), the copy is generated by the computer; when the Teletype unit is offline (LOCAL), the copy is automatically generated whenever a key is struck.

PAPER TAPE READER

The paper tape reader is used to input into memory data punched on eight-channel perforated paper tape at a maximum rate of ten characters per second. The reader control positions are shown in Figure 4-6 and 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.

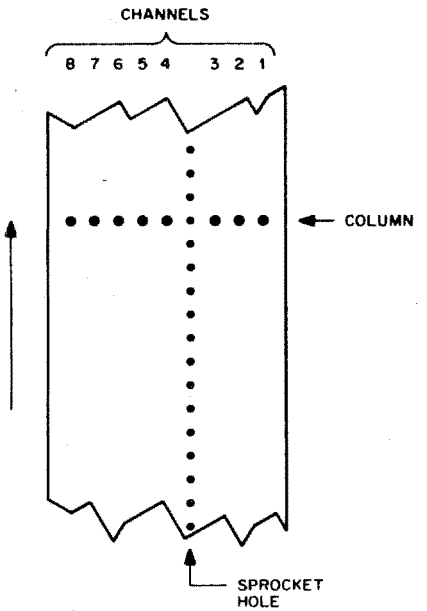
Table 4-1. Special Keyboard Functions

Key	Function	Use
SPACE	space	used to combine and delimit symbols or numbers in a symbolic program
RETURN	carriage return	used to terminate line of symbolic program
HERE IS	blank tape	used for leader/trailer (effective only in LOCAL)
RUBOUT	rubout	used for deleting characters, punches all channels on paper tape
CTRL/REPT/P	code 200	used for leader/trailer of binary program paper tapes (keys must be released in reverse order: P, REPT, CTRL)
LINE FEED	line feed	follows carriage return to advance printer one line

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 4-6 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 family computers accept eight-channel paper tape.

Generating a Symbolic Tape

The previously described components may be used to generate a symbolic program paper tape through the following procedure.

When switched to LOCAL, the Teletype unit is independent of the computer and functions like an electric typewriter. Any character struck on the keyboard is printed, and also punched on paper tape if the tape punch is ON. Each character struck on the keyboard is represented in code by one row of holes and spaces according to the ASCII code described in the following section and given in Appendix B.

A section of leader-trailer code several inches long is punched at the beginning of the symbolic tape, by pressing the HERE IS key on the Teletype keyboard. The symbolic program is then carefully typed, following the conventions used in PDP-8 symbolic programs as described in Chapter 3.

A typing error can be corrected using the B.SP. button of the paper tape punch and the RUBOUT key on the Teletype keyboard. The B.SP. button backspaces the paper tape one column for each depression of the button, and the RUBOUT key perforates all eight channels of a column (this perforation is ignored by the computer). Therefore, errors are removed by backspacing the tape to the error and typing rubouts over the error and all following characters. After typing rubouts, the correct infomation must be typed beginning where the error occurred.

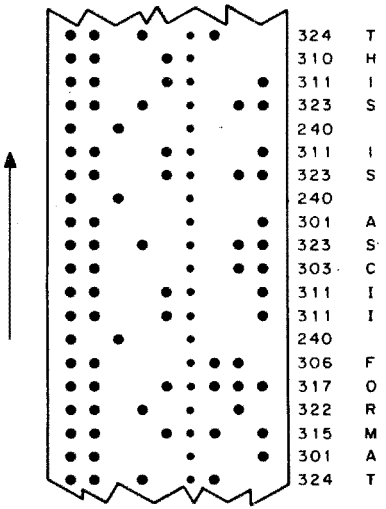
Once the symbolic tape is punched, more leader-trailer tape is generated by striking the HERE IS key. The tape is removed from the punch unit by tearing against the plastic cover of the punch. The symbolic program thus generated is the input to the assembler described in Chapter 6.

The program may be *listed* (typed out) by placing the paper tape in the paper tape reader. This is done by releasing the plastic cover of the reader unit and placing the eight-channel tape over the reader head with the smaller sprocket holes over the sprocket wheel, and replacing the cover. If the Teletype control is switched to LOCAL and the reader is switched to START, the tape will advance over the reader head and a printed copy of the program will be typed on the Teletype printer. If the tape punch is also ON, a duplicate of the tape will be generated at the same time.

Paper Tape Formats

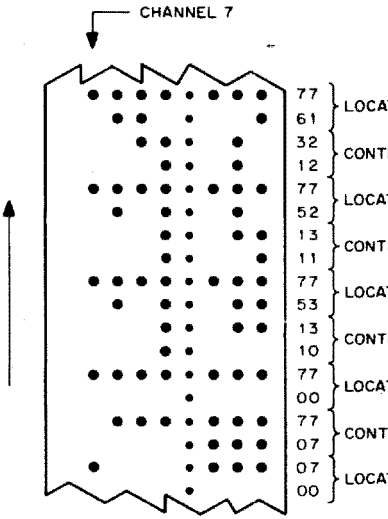
Manual use of the toggle switches on the operator console is a tedious and inefficient means of loading a program. This procedure is necessary in some instances, however, because the PDP-8 family of computers must be programmed before any form of input to the memory unit is possible. For example, before any paper tape can be used to input information into the computer, the memory unit must have a stored program which will interpret the paper tape format for the computer. This loader program must be stored in memory with the console switches. A loader program consists of input instructions to accept information from the Teletype paper tape reader and instructions to store the incoming data in the proper memory locations.

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



ASCII FORMAT

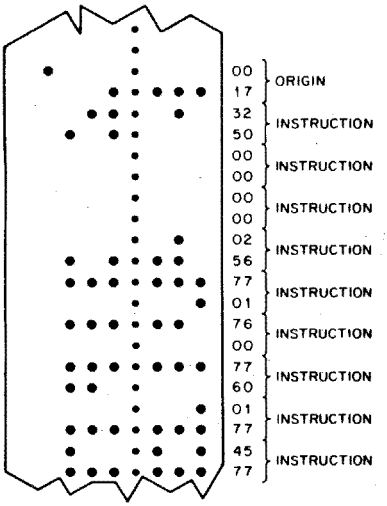
The USA Standard Code for Information Interchange (ASCII) format uses all eight channels[1] of the paper tape to represent a single character (letter, number, or symbol) as shown in the diagram at left. The complete code is given in Appendix B.



RIM (READ IN MODE) FORMAT

RIM format tape uses adjacent columns to represent 12-bit binary information directly. Channels 1 through 6 are used to represent either address or information to be stored. A channel 7 punch indicates that the adjacent column and the following column are to be interpreted as an address specifying the location in which the information of 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).

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



BIN (BINARY) FORMAT

BIN format tape is similar to RIM format except that only the first address of consecutive locations is specified. An address is designated by a channel 7 punch and information following an address is stored in sequential locations after the designated address, until another location is specified as an origin. The tape leader/trailer for BIN format tape must be punched in channel 8 (octal 200) only.

Paper Tape Loader Programs

The three previously described paper tape formats are each used for a separate purpose in conjunction with PDP-8 family computers. The ASCII format is used to represent symbolic programs on paper tapes, which are then used as input to the assembler. As described in Chapters 2 and 3, the assembler translates the mnemonic instructions and symbolic addresses into binary instructions and absolute addresses. Once this translation has been performed by the assembler, a binary format tape is generated.

The binary format tape is the common means of loading an assembled program into the core memory of a PDP-8 family computer. The *BIN (Binary) loader* is the program used to load these binary format paper tapes. Program instructions are stored in successive locations beginning with an origin which is signaled by a channel 7 punch on the paper tape. The BIN loader is a lengthy program requiring 83 memory locations. As an alternative to manually entering the contents of all 83 locations, the RIM (Read In Mode) format is used.

The *RIM loader* is simpler than the BIN loader because the memory unit is supplied with a location for each incoming instruction. It consists of 17 instructions which must be toggled into memory. The BIN loader is punched in RIM format, and is loaded by the RIM loader; but it is used to load tapes punched in the BIN format, which is the output of the assembly program.

The RIM loader is listed in Table 4-2. The instructions are toggled in, and checked by following the flowcharts given in Figures 4-3 and

4-4. The instructions are given for use with both the low speed reader (included in the Automatic Send Receive Model 33 Teletype) and for the high speed reader (an option described later in this chapter).

Table 4-2. RIM (Read In Mode) Loader

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.

The RIM loader will load into memory any program punched on paper tape in the RIM format. Because paper tapes in the RIM and BIN formats cannot be used until the user understands the material in Chapter 6, further discussion of the use of paper tape loaders is contained in that chapter.

PERIPHERAL EQUIPMENT AND OPTIONS

PDP-8 family 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 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 new capabilities of which the programmer should be aware. The purpose and features of each of these options is described in the following paragraphs.

All of the options listed above and many others may be directly connected to the PDP-8/I. The PDP-8/L may be directly equipped with the high speed paper tape reader and punch unit only. Other

peripherals must be interfaced to the PDP-8/L through an I/O conversion panel and/or a peripheral expansion panel. EAE is not available with the PDP-8/L.

High Speed Paper Tape Reader and Punch Unit

Loading a long paper tape program into the PDP-8 core memory with the low-speed reader of the ASR 33 Teletype unit is very time consuming. Punching a long program on paper tape from an assembly program likewise is very slow. If handling lengthy paper tapes is commonly required, much computer time is wasted while these low-speed I/O devices read or punch data. The high-speed paper tape reader and punch unit, shown in Figure 4-8, performs paper tape input and output at a considerably faster rate. It is of great value in a system that relies on paper tape as a primary medium of data and program storage.

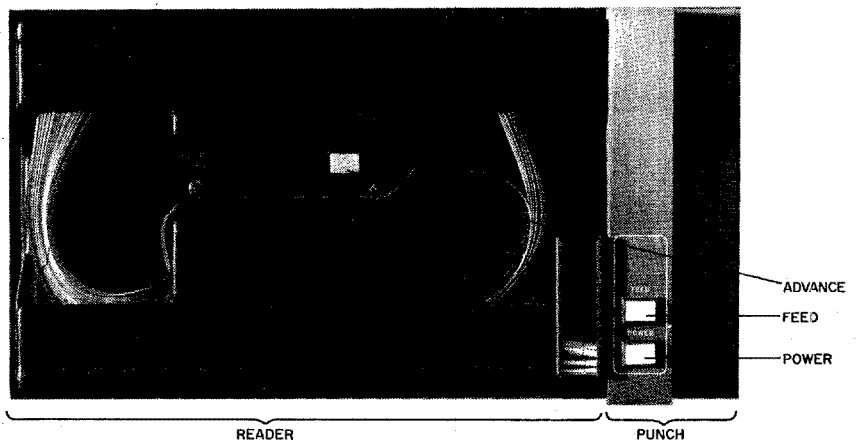


Figure 4-8. High-Speed Paper Tape Unit

The high-speed paper tape reader is used to input data into core memory from eight-channel, fan-folded (non-oiled), perforated paper tape. The reader inputs information photoelectrically at a rate of 300 characters per second (ASR 33 reader inputs at ten characters per second maximum). Primary power is applied to the reader when the computer console POWER switch is on. The reader is controlled by the computer, although the operator may indirectly control the reader from the keyboard through the computer. Tape may be advanced, without being recorded by the photoelectric sensors, by pressing the white *advance* button.

Paper tapes are manually positioned in the high speed reader with the following steps.

1. The paper tape is placed in the right-hand bin such that the beginning of the tape will pass over the sensors first.
2. Several folds of leader tape are placed in the left-hand bin with the tape passing under the tape retainer cover.
3. The retainer cover is closed over the tape such that the feed holes are engaged in the teeth of the sprocket wheel.
4. Tape is advanced and read by programmed computer instructions.

Once the paper tape has been properly placed in the reader and the leader/trailer has been positioned as outlined in the preceding steps, the tape is normally read under control of system software.

The high speed paper tape punch is used to record computer output on eight-channel, fan-folded paper tape at 50 characters per second. All characters are punched under program control from the computer. Primary power is available to the punch when the computer console POWER switch is turned on. Power is supplied when the POWER button is depressed on the punch unit itself. In addition to the POWER button, a FEED button is located on the punch enclosure to advance feed-hole-only punched tape for leader/trailer purposes.

The loader programs, symbolic assemblers, Symbolic Editor, and other system software presented in Chapter 6 include instructions for using the high speed reader/punch, as well as for using the Teletype reader/punch. (By incorporating the appropriate instructions for the high speed unit (see Appendix D), the user may write his own I/O routines for this device as outlined in Chapter 5.)

Extended Memory

The PDP-8 family of computers have a memory unit composed of 12-bit magnetic core locations. The basic configuration stores 4,096 12-bit words; however, the memory unit of the PDP-8/I, -8/S, or -8 can be expanded into a maximum storage of 32,768 words by adding 4,096-word memory modules. The PDP-8/L may be expanded to 8,192 words. Each module is called a *field,* with field 0 being the original 4,096 words and other fields designated 1, 2, . . ., 7. (The PDP-8/L can have only fields 0 and 1.)

Expansion of the basic memory introduces *data field* and *instruction field* registers into the memory unit. Related to these registers are the DATA FIELD and INST FIELD switches and displays of the computer console. Since the 12-bit word of the PDP-8 is capable of representing only 4,096 locations uniquely, the data field and instruction

field registers are used to designate the field of 4,096 words which contains a particular address.

INSTRUCTION FIELD

The content of the instruction field register determines the instruction field (field of 4,096 words) that the instructions are to be taken from. Any *directly addressed* AND, TAD, ISZ, or DCA instruction will obtain its operand from the instruction field. In indirectly addressed instructions, however, the pointer address is taken from the instruction field, but the operand (specified by the effective address) is obtained from the data field.

DATA FIELD

The content of the data field register specifies the data field (field of 4,096 words) from which operands (specified by the effective address) are taken in *indirectly addressed* AND, TAD ISZ or DCA instructions. (The pointer addresses are obtained from the instruction field.)

INITIAL FIELD ASSIGNMENTS

The original setting of the data field and instruction field registers are by the DATA FIELD and INST FIELD switches of the computer console (see Figure 4-1). Thus, to run a program beginning in location 200 of field 1 and operating on data in field 0, the INST FIELD (IF) switches are set to 001, and the DATA FIELD (DF) switches set to 000 (on the PDP-8/L, set to 1 and 0 respectively). The switch register is then set to 200_8. When the LOAD ADD switch is operated, the values for the data field and instruction field are entered as well as the starting address. When the START switch is depressed, the program beginning in location 200 of field 1 is executed.

A common use of extended memory is the storage of system software. For example, the Binary Loader may be stored in field 1. By setting IF = 1 and DF = 0, the Binary Loader runs in field 1, but deposits the program (which is simply data to the loader) in field 0.

CHANGING FIELD ASSIGNMENTS

The instructions for extended memory (see Appendix D) may be used to change instruction and data fields during the execution of a program. The instructions are written in the following format.

Symbolic Instruction	Explanation
CIF+30	change to instruction field 3
CDF+10	change to data field 1

The field being changed to is specified by adding its value to the second octal digit position of the change field instruction. The CDF instruction

causes all future indirectly-addressed operands to be taken from the specified data field. The CIF (change instruction field) instruction does not take effect immediately, but waits for a JMP or JMS instruction to be encountered in the program execution. When the JMP or JMS is encountered, control is transferred to the new instruction field. Thus, if the instruction field is originally field 0, the following instructions transfer control to field 1.

```
CIF+10
JMP 20
```

The next instruction to be executed in the program is contained in location 20 of field 1. If the JMP is indirectly-addressed, the pointer word is obtained from the old field, but the JMP is to a location in the new field.

DECtape System

DECtape is an option of the PDP-8 family of computers which serves as an auxiliary magnetic tape storage facility and updating device. The standard DECtape transport unit is pictured in Figure 4-9. The DECtape system stores and retrieves information at fixed positions on magnetic tape. The advantage of DECtape over conventional magnetic tape is that information is stored at *fixed* positions which may be addressed. Allocation of fixed, addressable positions for information storage is a unique feature of DECtape storage facilities, while conventional magnetic tape stores information in sequential (not directly addressable), variable-length positions. DECtape incorporates timing and mark information to reference the fixed positions. The 10-channel DECtape records five channels of information: a timing channel, a mark channel, and three information channels. These five channels are duplicated on the remaining five nonadjacent channels to minimize any possibility of loss of information from the other channels. The DECtape is organized in blocks of data words with control words to identitfy each block. The tape is bidirectional; that is, it can be written or read in the forward or reverse direction. Information should be read in the same direction that it was written.

The DECtape control unit performs the transfer of information between the PDP-8 and the transport unit. The control can operate as many as eight separate DECtape transport units.

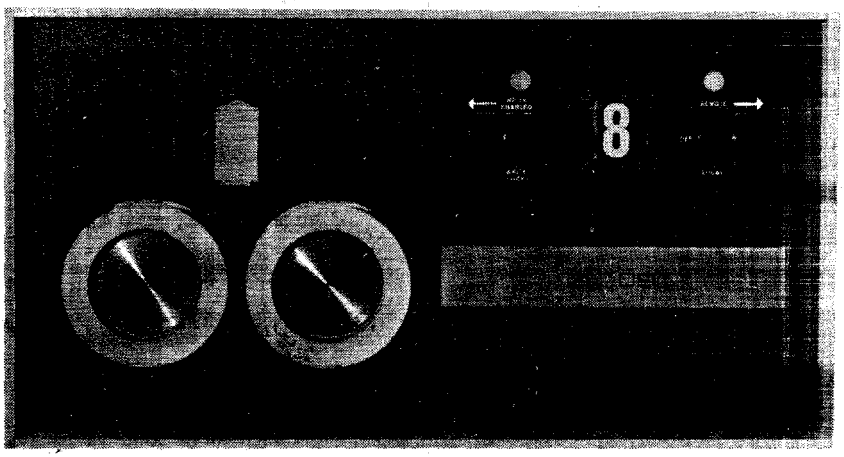
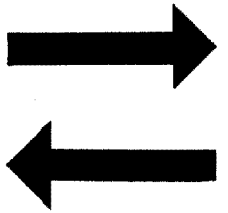


Figure 4-9. DECtape Transport Unit

The DECtape transport unit shown in Figure 4-9 is a bidirectional magnetic tape transport which reads and writes the 10-channel magnetic tape. Tape movement can be controlled by programmed instructions from the computer or by the manual operation of switches located on the front panel of the transport. Data is transferred only under program control.

The transport controls are identified below.

Transport Control	Explanation
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 switch identifies the transport to the control unit. NOTE: Position 8 on the Unit Selector switch corresponds to DECtape unit 0.



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 REMOTE and WRITE ENABLED lamps indicate whenever their respective conditions are present.

DECdisk System

The DECdisk file is a fast, random-access, bulk storage device for the PDP-8 computer family. It has a considerably faster access time than DECtape and offers storage of 32,768 words on each disk (as many as four disks are possible). The DECdisk comprises a storage unit with electronics to perform the read and write functions, and the computer interface logic which participates in the transfer of information between the central PDP-8 and the DECdisk unit.

The storage unit contains a motor-driven, nickel/cobalt-plated disk. The disk has 16 data tracks, with 2,048 words per track. There are two timing tracks plus two spares. Data is recorded on a single disk by fixed position read/write heads. Transfer of information between the DECdisk and the PDP-8 is controlled by programmed instructions as outlined in Chapter 5.

A fast, convenient, keyboard-oriented Disk/DECtape Monitor is available for use with the PDP-8 family computers to allow the programmer to efficiently control the flow of programs through any PDP-8 having a DECdisk or DECtape. This monitor system is described in more detail in Chapter 7.

Extended Arithmetic Element

The EAE option of the PDP-8 computer family (not available on the PDP-8/L) provides circuitry to perform arithmetic operations which can not be directly performed with the basic PDP-8 instruction set. The option includes microinstructions to perform multiplication and division. Other microinstructions perform arithmetic and logical shifts and normalize both positive and negative two's complement numbers.

The option provides a 12-bit multiplier quotient register (MQ) which is used in conjunction with the AC to perform direct multiplication and division. The content of this register is displayed on the PDP-8 console.

The EAE option is essentially an increase in instruction capability. The instructions, which are microprogrammable, are included in Appendix D.

NOTE TO READER

The following exercises are intended for readers who have access to a computer of the PDP-8 family. Readers who do not presently have access to such a computer should begin study of Chapter 5.

Upon completion of the following exercises, readers with access to a PDP-8 would benefit from reading the sections of Chapter 6 which describe loaders, the Symbolic Editor, and the PAL III Symbolic Assembler. Although knowledge of this material is not necessary to understand the subject matter of Chapter 5, this knowledge will facilitate the running of programs presented therein.

EXERCISES

1. Toggle into memory and run the programs written in Chapter 2, for exercises 6 and 10.
2. Toggle into memory the RIM Loader (Table 4-2) using the console switches. Verify the contents of the registers 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 by a HLT. After seeing the switch register activate the CONT key.) Translate the program into octal and toggle it into memory. Verify that it works properly.

Chapter 5

Input/Output Programming

Being able to program a computer to do calculations is of little use if there is no way of getting the results of calculations from the machine. Likewise, the programmer often must supply the computer with information to be processed. A programmer must be provided with the means to transfer information between the computer and the peripheral devices that supply input or that serve as a means of output.

Before a transfer of information can be executed, a control function must be supplied to specify when the exchange will occur, with what peripheral device the exchange will occur, and where in core storage the information will be stored (or obtained from). In general, this control function may be served by either the PDP-8 or the peripheral device itself.

There are three basic methods for the transfer of information between input/output (I/O) devices and the PDP-8. The first two methods provide for PDP-8 control over the transfer. One method is *programmed transfer,* in which instructions are included at some point in the program to accept or transmit information. Thus, programmed transfers are program initiated and are under program control.

Information may also be transferred through *program interrupt,* a standard feature of the PDP-8 computer family that provides for devices to signal the PDP-8 when they are ready to transfer information; the program will then interrupt its normal flow and jump to a routine to process the information, after which it will return to the point in the main program at which it was interrupted. Thus, program interrupt transfers are device initiated but are under program control.

These first two methods (i.e., programmed transfers and program interrupt) use the accumulator as the *buffer,* or storage area in the computer, for all data transfers. Therefore, only one 12-bit word of input or output may be transferred at one time by a programmed transfer, or by program interrupt.

The third method of information transfer is *data break,* an option for the PDP-8/L but standard for other PDP-8 computers. Data break is essentially device controlled and allows for direct exchange of large quantities of information between the device and the PDP-8 memory. It differs from the previous two types of transfer in that there are no program instructions to handle the transfer and the accumulator is not used as a buffer. Data break transfers are device initiated and device controlled.

INPUT/OUTPUT INSTRUCTIONS

As the name implies, programmed transfers of information are accomplished with a set of program instructions. The instructions are similar to the operate microinstructions in that there is no need to specify an address in memory. The operation code 6_8 is used to specify an input/output transfer (IOT) instruction. All programmed transfers are between the accumulator and the device. Since many different devices could be connected to one computer and each device may at some time transfer information, the instruction must identify the proper device for each transfer. The instruction must also specify the exact nature of the function to be performed.

IOT Instruction Format

An IOT instruction is a 12-bit word that is in the following format. The first three bits represent the operation code 6_8. The remaining nine bits may be either binary 0's or 1's.

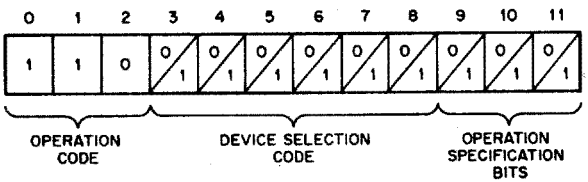


Figure 5-1. The IOT Instruction

The IOT instruction is divided into three parts: operation code, device selection code, and operation specification bits.

Device Selection

The device selection code is transmitted to *all* peripheral equipment whenever the IOT instruction is executed. A *device selector* within each peripheral device monitors the device codes. When the device selector recognizes a device code as the device's assigned code, the device receives the last three bits of the instruction. Each of the last three bits specifies an action associated with the device. When one of the last three bits is set to a 1, the specified action is performed. Since there are three bits, only three different actions can be specified for each device code, although microprogramming is possible. When more instructions are necessary for a given device, more than one code is assigned to the device.

Checking Ready Status

Because there is a great difference in the processing speed of a computer and the speed of most peripheral devices, the computer must check the readiness of a device before any transfer of information is performed. The input device must signal the computer that it has completely assembled the information and is now ready to transfer the information to the computer memory. The output device must signal its readiness to accept the next piece of information from the computer. Without such signals, the computer would input and output information at a faster rate than the device could process it and some information would be lost.

To prevent any loss of information, the computer program checks the ready status of the transmitting or receiving device as part of preparing for a normal data transfer. The ready status is usually checked with a skip instruction such that if the device is ready, the following instruction is skipped. The ready status is signaled through a system of *flags,* which are 1-bit registers within the device. All I/O devices have a device flag which is set to a 1 when the device is ready; that is, when it can be used (if it is an output device), or when it has information (if it is an input device). If the flag is cleared (set to 0), the device is *busy*. If a program initiates a device action, the flag associated with that device will be set to a 1 when the device action is completed.

Instruction Uses

In general, for each device there are three instructions:

1. An instruction to transfer information and 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 the device flag.

The above instructions may be microprogrammed. In particular, the instructions to clear the flag and to operate the device often are combined.

The specific instructions for devices are given in the following sections. The Teletype unit is described in depth to explain the fundamentals of programming data transfers. The general techniques developed for the Teletype unit may be extended to handle other devices.

ASCII Code

The ASCII (U.S.A. Standard Code for Information Interchange) is presented in Table 5-1. Many of the programs written in this chapter use this code to transmit information to the PDP-8. The fact that the ASCII code for the octal digits 0 through 7 is the sum of that digit plus 260_8 should be observed.

PROGRAMMING THE TELETYPE UNIT

One of the most common I/O devices is the Teletype unit, which contains a keyboard, printer, paper tape reader, and paper tape punch. The Teletype unit can use either the keyboard or the paper tape reader to input information to the computer and can use either the printer or the paper tape punch to accept output information from the computer. The Teletype unit is therefore assigned two device codes.

Teletype Input/Output Transfer Instructions

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.

Table 5-1. The 8-Bit ASCII[1] Code

Character	8-Bit Octal	Character	8-Bit Octal
A	301	!	241
B	302	"	242
C	303	#	243
D	304	$	244
E	305	%	245
F	306	&	246
G	307	'	247
H	310	(	250
I	311	)	251
J	312	*	252
K	313	+	253
L	314	,	254
M	315	—	255
N	316	.	256
O	317	/	257
P	320	:	272
Q	321	;	273
R	322	<	274
S	323	=	275
T	324	>	276
U	325	?	277
V	326	@	300
W	327	[	333
X	330	\	334
Y	331	]	335
Z	332	↑	336
0	260	←	337
1	261	Leader/Trailer	200
2	262	LINE FEED	212
3	263	Carriage RETURN	215
4	264	SPACE	240
5	265	RUBOUT	377
6	266	Blank	000
7	267	BELL	207
8	270	TAB	211
9	271	FORM	214

[1] An abbreviation for USA Standard Code for Information Interchange.

KEYBOARD/READER INSTRUCTIONS

The instruction format for the keyboard/reader is shown in Figure 5-2. The mnemonic instructions generated by bits 9, 10, and 11 are noted. The sequence in which the mnemonic instructions are executed when microprogrammed is noted below.

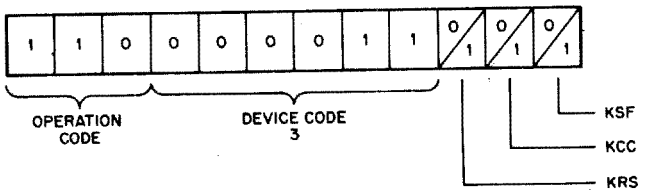


Figure 5-2. Teletype Keyboard/Reader Instructions

Sequence	Mnemonic	Octal	Effect
1	KSF	6031	Skip the next instruction when the keyboard buffer register is loaded with an ASCII symbol (causing the keyboard flag to be raised).
2	KCC	6032	Clear AC, clear keyboard flag.
3	KRS	6034	Transfer the contents of the keyboard buffer into the AC.
2,3	KRB	6036	Transfer the contents of the keyboard buffer into the AC, clear the keyboard flag.

The fourth instruction (KRB) is a microprogrammed combination of the mnemonics KCC and KRS. If the paper tape reader is loaded with a paper tape and switched to START, the KRB instruction accepts one character from the reader.

A program using the above instructions to read in one ASCII character from the keyboard or paper tape reader is shown in Figure 5-3. Note that this program does *not* type the character on the teleprinter, it merely stores the ASCII code for the character in the location STORE.

```
*200
INPUT,      KCC             /CLEAR KEYBOARD FLAG
            JMS LISN
            DCA STORE
            HLT
LISN,       0
            KSF             /SKIP ON KEYBOARD FLAG
            JMP .-1
            KRB             /READ KEYBOARD BUFFER
            JMP I LISN
STORE,      0
$
```

Figure 5-3. Coding to Accept One ASCII Character

The main program begins with KCC. In general, the main program should begin by clearing the flags of all devices to be used later in the program. If the above program is started at location 200, it will proceed to the KSF, JMP .—1 loop, and stay in this loop endlessly until a key on the Teletype unit is pressed or a paper tape is loaded into the reader. When the ASCII code for the character is assembled in the keyboard/reader buffer register, the flag will be set to a 1 and the program will skip out of the loop. The contents of the buffer will be transferred into the accumulator, and the buffer and flag will be cleared.

PRINTER/PUNCH INSTRUCTIONS

The instruction format for the Teletype printer/punch IOT instructions is given in Figure 5-4. The mnemonic instructions generated by bits 9, 10, and 11 are discussed on the following page.

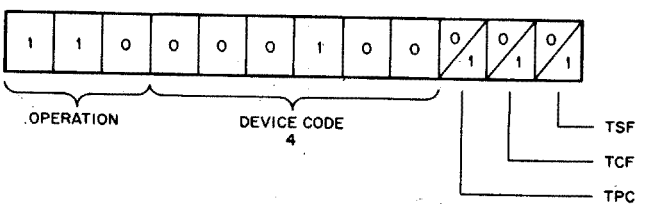


Figure 5-4. Teletype Printer/Punch Instructions

Sequence	Mnemonic	Octal	Effect
1	TSF	6041	Skip the next instruction if the printer flag is set to 1.
2	TCF	6042	Clear the printer flag.
3	TPC	6044	Load the printer buffer register with the contents of the AC, select and print the character. (The flag is raised when the action is completed.)
2,3	TLS	6046	Clear the printer flag, transfer the contents of the AC into the printer buffer register, select and print the character. (The flag is raised when the action is completed.)

The last instruction is a microprogrammed combination of TPC and TCF, such that the flag is cleared, the character is printed, and then the flag is again raised. Whenever the paper tape punch is turned on, the character is punched on paper tape as well as printed on the teleprinter.

Figure 5-5 illustrates a program to print out one ASCII character which is held in a memory location.

```
*200
OUTPUT,  CLA CLL
         TLS
         TAD HOLD
         JMS TYPE
         HLT
TYPE,    0
         TSF              /SKIP ON TELEPRINTER FLAG
         JMP .-1
         TLS              /PRINT THE CHARACTER
         CLA CLL
         JMP I TYPE
HOLD,    0
$
```

Figure 5-5. Coding to Print One ASCII Character

The program in Figure 5-5 begins by clearing the accumulator and executing a TLS instruction (which has the effect of clearing the printer buffer), after which the printer flag will be set, thereby signifying readiness to accept a character. If the initial TLS instruction were not

executed, the flag would not be raised (the START key clears all flags), and the program would remain in the TSF, JMP .—1 loop endlessly. In the previous case, however, the program uses the printer with a cleared accumulator such that no character is printed. However, the flag is set when this action is complete enabling the printing of meaningful information in the TYPE subroutine. The TYPE subroutine clears the accumulator since the TLS instruction does not. It is advisable to clear the accumulator after any subroutine unless meaningful data is contained in it.

Format Routines

Input and output routines are very often written in the form of subroutines, as the TYPE subroutine in the previous example. The example in Figure 5-6 is a carriage return/line feed subroutine that calls the TYPE subroutine to execute a carriage return and line feed on the printer, thus advancing to a new line for the printing of information.

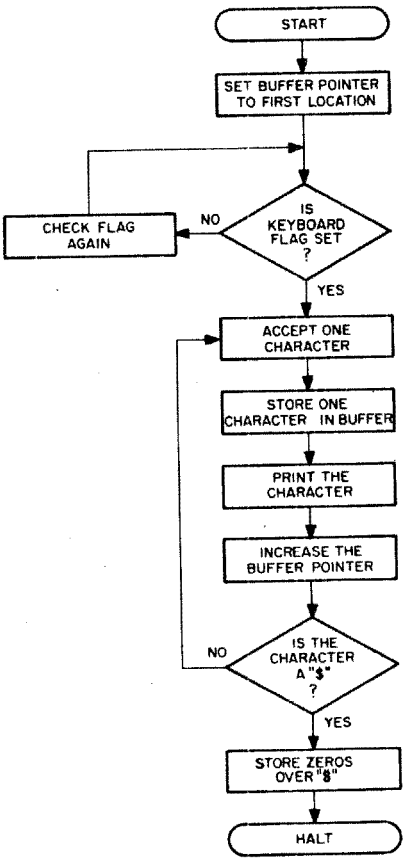
```
CRLF,   0
        TAD K215
        JMS TYPE
        TAD K212
        JMS TYPE
        JMP I CRLF
K215,   215             /ASCII FOR CARRIAGE RETURN
K212,   212             /ASCII CODE FOR A LINE FEED
TYPE,   0
        TSF
        JMP .—1
        TLS
        CLA CLL
        JMP I TYPE
```

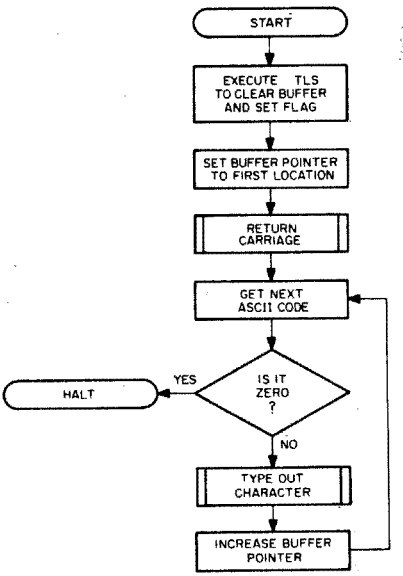
Figure 5-6. Carriage Return/Line Feed Subroutine

Subroutines similar to the one in Figure 5-6 could be written to tab space the carriage a given number of spaces, or to ring the bell of Teletype Model ASR 33 by using the respective codes for these nonprinting characters. Such subroutines, if commonly used in a program, should be placed on page 0 (or else a pointer word to the subroutine should be placed on page zero) to facilitate reaching the routine from all memory locations.

Text Routines

The examples in Figures 5-3 and 5-5 may be expanded to accept and type more than one character. Figures 5-7 and 5-8 illustrate one expansion. These two programs are compatible in that the characters accepted by the first program are typed out by running the second program. The program to accept characters, in Figure 5-7, will continue to accept character input until a dollar sign ($) is struck on the keyboard, at which time the program will store all zeros in the next location and then halt. The program in Figure 5-8 will type the characters whose ASCII code was stored by the first program. The second program will halt when a location with contents equal to zero is reached. Both programs use locations beginning with 2000 as the buffer for the storage of ASCII characters. The following flowcharts introduce the techniques used in the program coding.





```
*5000
START,  CLA CLL
        TAD BUFF        /SET UP BUFFER SPACE.
        DCA BUFFPT
LISN,   KSF             /KEYBOARD STRUCK YET?
        JMP .-1         /NO: CHECK AGAIN.
        KRB             /YES: READ CHARACTER.
        TLS             /ACKNOWLEDGE IT ON PRINTER.
        DCA I BUFFPT    /STORE CHARACTER.
        TAD I BUFFPT    /CHECK FOR TERMINAL $.
        TAD MDOLAR
        SNA
        JMP DONE        /CHARACTER IS A $.
        ISZ BUFFPT      /CHARACTER IS NOT A $.
        JMP LISN        /GET ANOTHER CHARACTER.
DONE,   CLA CLL         /STORE Ø IN LAST LOCATION.
        DCA I BUFFPT
        HLT
BUFF,   2000
BUFFPT, Ø
MDOLAR, 7534
$
```

Figure 5-7. Routine to Accept and Store ASCII Characters

```
*5200
START,  CLA CLL
        TLS             /TLS TO SET PRINTER FLAG.
        TAD BUFF        /SET UP BUFFER SPACE.
        DCA BUFFPT
        JMS CRLF        /RETURN CARRIAGE.
CHRTYP, TAD I BUFFPT    /GET A CHARACTER.
        SNA             /IS IT ALL ZEROS?
        HLT             /YES: STOP.
        JMS TYPE        /NO: TYPE OUT THE CHARACTER.
        ISZ BUFFPT      /INCREMENT BUFFER POINTER.
        JMP CHRTYP      /TYPE ANOTHER CHARACTER.
CRLF,   Ø               /CARRIAGE RETURN & LINE FEED.
        TAD K215        /TYPE CR FIRST.
        JMS TYPE
        TAD K212        /TYPE LINE FEED.
        JMS TYPE
        JMP I CRLF
TYPE,   Ø               /SUBROUTINE TO TYPE CHARACTER.
        TSF             /PRINTER READY YET?
        JMP .-1         /NO: CHECK AGAIN.
        TLS             /YES: TYPE CHARACTER.
        CLA             /CLEAR ASCII FROM AC.
        JMP I TYPE
BUFF,   2000
BUFFPT, Ø
K215,   215
K212,   212
$
```

Figure 5-8. Routine to Print Stored ASCII Characters

The program to print characters may be specialized to print a specific word as in the program of Figure 5-9. The example is a subroutine which uses autoindex registers in place of the ISZ instruction. The subroutine types "HELLO!"

```
HELLO,  0                /HELLO SUBROUTINE
        CLA CLL
        TLS              /TLS TO SET PRINTER FLAG.
        TAD CHARAC       /SET UP INDEX 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 MAIN PROGRAM.
TYPE,   0                /TYPE SUBROUTINE
        TSF
        JMP .-1
        TLS
        CLA
        JMP I TYPE
CHARAC, .                /USED AS INITIAL VALUE OF IR1
        310              /H
        305              /E
        314              /L
        314              /L
        317              /O
        241              /!
M6,     -6
COUNT,  0
IR1=10
```

Figure 5-9. Routine to Print One Word

Numeric Translation Routines

The ASCII codes of octal numbers may be transmitted to the PDP-8 memory as in the program of Figure 5-3. However, the ASCII code for the number must be converted to true octal representation before the computer may use this input. For example, 6 is represented by the ASCII code 266. When the Teletype key is struck for 6, the code 266 is transmitted to the computer upon execution of the KRB instruction. To remove the 260 from the coded number to obtain the octal number itself, two methods could be used.

The first method is to examine the binary form of the ASCII code.

000 010 110 110

Setting the first eight bits to zero by using the AND instruction with the appropriate mask results in the binary value for 6. The appropriate mask for this purpose is 17_8 as shown below.

Instruction	Operation	Comment
	000 010 110 110	ASCII Code 266 in accumulator
AND MASK	000 000 001 111	MASK: 17_8
	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. The instruction TAD M260 is used for this operation as shown in the following example.

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.

Programs to accept and store an octal digit, using the LISN subroutine previously given, are shown in Figure 5-10.

```
*200
/USING AND
 /INSTRUCTION
NUMIN,  KCC
        JMS LISN
        AND MASK
        DCA HOLD
        HLT
HOLD,   0
MASK,   17
$
```

```
*200
/USING TAD
/INSTRUCTION
NUMIN,  KCC
        JMS LISN
        TAD M260
        DCA HOLD
        HLT
HOLD,   0
M260,   7520
$
```

```
/LISN SUBROUTINE
LISN,   0
        KSF
        JMP .-1
        KRB
        JMP I LISN
```

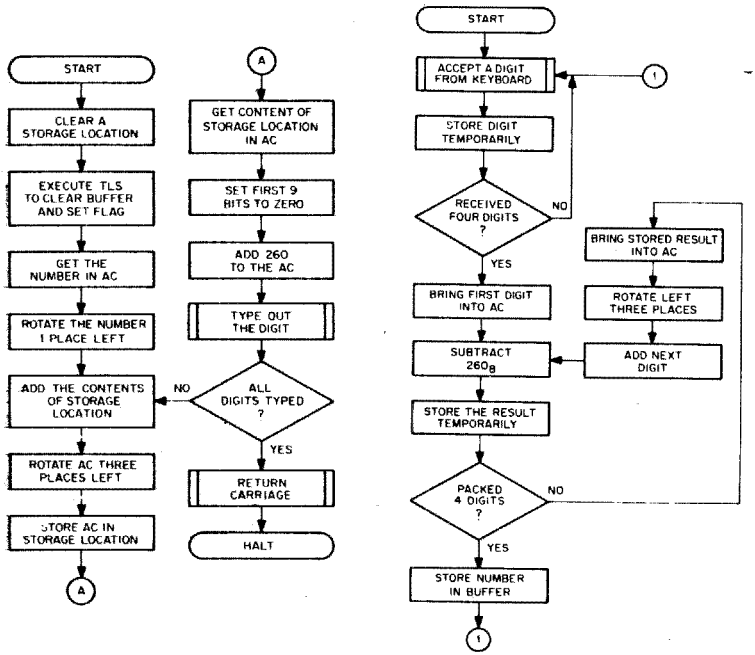
Figure 5-10. Two Methods of Converting ASCII to Binary

When an octal digit stored in memory is to be printed on the teleprinter, the octal number 260 must be added to it. The routine in Figure 5-11 uses the TYPE subroutine previously introduced to print out the binary number 7.

```
*200                                    /TYPE SUBROUTINE
NUMOUT,     CLA CLL                     TYPE, 0
            TLS                         TSF
            TAD NUMBER                  JMP .-1
            TAD K260                    TLS
            JMS TYPE                    CLA CLL
            HLT                         JMP I TYPE
NUMBER,     7
K260,       260
$
```

Figure 5-11. Routine to Type One Stored Digit

The routines presented thus far have been designed to handle only single-digit octal numbers. The PDP-8 core storage location, however, is able to represent octal numbers up to four digits. The routine in Figure 5-12 takes a number that is stored in a PDP-8 location and prints it on the printer as four octal digits. The routine in Figure 5-13 accepts four octal digits from the keyboard and converts them to one octal number and stores it in a location, after which it accepts another four digits, etc. The following flowcharts illustrate the procedure used in the programs.



```
*200
START,  CLA CLL
        DCA STORE       /CLEAR OUT STORAGE LOCATION.
        TLS             /TLS TO SET PRINTER FLAG.
        JMS CRLF        /RETURN CARRIAGE.
        TAD M4          /SET LOCATION TO COUNT NUMBER
        DCA DIGCTR      /OF TYPED DIGITS.
        TAD NUMBER      /GET NUMBER TO BE TYPED.
        RAL             /ROTATE ONE PLACE (INTO LINK).
UNPACK, TAD STORE       /ADD STORED LOCATION TO AC.
        RAL             /ROTATE THREE
        RTL             /PLACES LEFT.
        DCA STORE       /STORE ROTATED NUMBER.
        TAD STORE       /MASK OUT ALL BUT
        AND MASK7       /FIRST THREE BITS OF NUMBER,
        TAD K260        /ADD IN 260,
        JMS TYPE        /AND TYPE 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
        JMP .-1
        TLS
        CLA
        JMP I TYPE
CRLF,   0               /CARRIAGE RETURN & LINE FEED
        TAD K215
        JMS TYPE
        TAD K212
        JMS TYPE
        JMP I CRLF
NUMBER, 1234            /NUMBER TO BE TYPED
MASK7,  7
M4,     -4
DIGCTR, 0
STORE,  0
K212,   212             /ASCII FOR LF
K215,   215             /ASCII FOR CR
K260,   260
$
```

Figure 5-12. Routine to Type a 4-Digit Number

```
*200
START,  CLA CLL
        TLS                 /TLS TO SET PRINTER FLAG.
        TAD K1777           /SET INDEX REGISTER FOR
        DCA IR1             /STORING PACKED NUMBERS.
        JMS CRLF            /RETURN CARRIAGE.
NXTNUM, TAD M4              /SET COUNTER FOR 4 DIGITS.
        DCA COUNTR
        TAD K350            /SET UP TEMPORARY STORAGE
        DCA TEMP            /FOR THE ASCII INPUTS.
NXTDIG, JMS LISN            /GET CHARACTER FROM KEYBOARD.
        DCA I TEMP          /STORE IT TEMPORARILY.
        ISZ TEMP            /INCREMENT STORAGE LOCATION.
        ISZ COUNTR          /GOT 4 DIGITS YET?
        JMP NXTDIG          /NO: GET ANOTHER.
        JMS CRLF            /YES: RETURN CARRIAGE
        JMS PACK            /AND PACK THE 4 DIGITS.
        DCA I IR1           /STORE PACKED NUMBER.
        JMP NXTNUM          /GET A NEW NUMBER.
PACK,   0                   /PACK SUBROUTINE.
        DCA STORE           /CLEAR OUT STORAGE LOC.
        TAD M4              /SET COUNTER FOR
        DCA COUNTR          /4 DIGITS.
        TAD K350            /SET POINTER TO
        DCA TEMP            /ASCII INPUT CHARACTERS.
PAKDIG, TAD STORE           /GET STORE IN AC.
        CLL RAL             /ROTATE INTO CLEARED LINK.
        RTL                 /ROTATE IT TWICE MORE.
        TAD I TEMP          /ADD NEXT STORED CHARACTER.
        TAD M260            /STRIP OFF THE 260.
        DCA STORE           /STORE STRIPPED NUMBER.
        ISZ TEMP            /INCREMENT POINTER TO ASCII.
        ISZ COUNTR          /PACKED 4 DIGITS?
        JMP PAKDIG          /NO: PACK NEXT DIGIT.
        TAD STORE           /YES: TAKE PACKED NUMBER
        JMP I PACK          /BACK TO MAIN PROGRAM.
CRLF,   0                   /CARRIAGE RETURN LINE FEED.
        TAD K215
        JMS TYPE
        TAD K212
        JMS TYPE
        JMP I CRLF
```

Figure 5-13. Routine to Pack and Store 4-Digit Octal Numbers

```
LISN,   0                   /SUBROUTINE TO ACCEPT ASCII.
        KSF
        JMP .-1
        KRB
        TLS
        JMP I LISN
TYPE,   0                   /SUBROUTINE TO TYPE ASCII.
        TSF
        JMP .-1
        TLS
        CLA
        JMP I TYPE
K1777,  1777
IR1=10
M4,     7774
COUNTR, 0
K350,   350                 /TEMPORARY STORAGE (350-353).
TEMP,   0
STORE,  0
M260,   7520
K215,   215
K212,   212
$
```

Figure 5-13. (cont.) Routine to Pack and Store 4-Digit Octal Numbers

Sample Program

The previously described routines for typing text and numeric translation are combined in the following program example which is similar to the final program of Chapter 3. This program performs the same numeric sort; however, the numbers to be placed in order are supplied from the keyboard.

Any number of elements may be supplied; the end of input is signaled by typing a dollar sign ($). The program includes routines to exclude any nonoctal digits from input and type a question mark. Only positive octal numbers ($0\text{-}3777_8$) are allowed as input to the program.

The program is presented in four illustrations. The following flowchart diagrams the program.

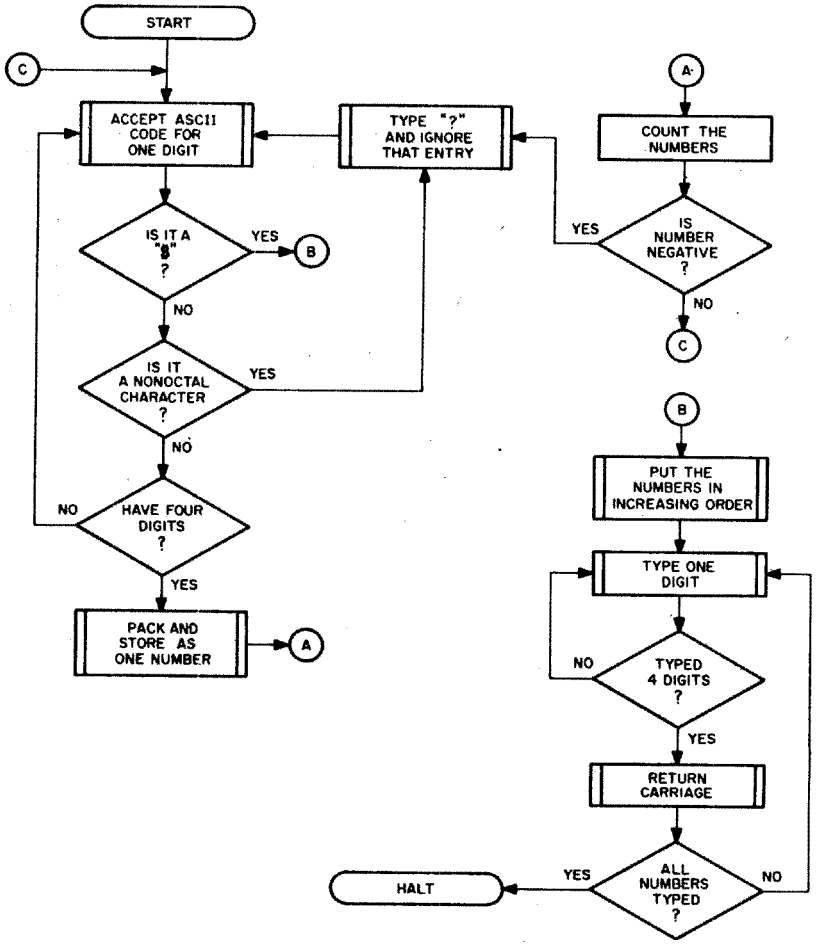


Figure 5-14A. Sample Program: Flowchart

```
*200
/INITIALIZATION
START,  CLA CLL
        TLS                 /TLS TO SET PRINTER FLAG.
        TAD BUFF            /SET UP STORAGE AREA.
        DCA BUFFPT
        DCA AMOUNT          /SET AMOUNT TO 0.
/ACCEPT ONE DIGIT.
ACCEPT, JMS CRLF            /RETURN CARRIAGE.
        TAD M4              /SET UP  COUNTER
        DCA DIGCTR          /FOR 4 DIGITS.
        TAD TEMP1           /SET A POINTER TO
        DCA TEMP            /TEMPORARY INPUT STORAGE.
NEWDIG, JMS LISN            /GET A CHARACTER.
        DCA I TEMP          /STORE IT.
/CHECK THE CHARACTER.
CHECK,  TAD I TEMP
        TAD MDOLAR          /IS CHARACTER A $ ?
        SNA CLA
        JMP ORDER           /YES: ORDER INPUT.
        TAD I TEMP          /NO: CHECK FOR OCTAL INPUT.
        TAD M260            /IS ASCII LESS THAN 260?
        SPA
        JMP ERROR           /YES: ERROR.
        TAD M10             /NO: SUBTRACT 10.
        SMA CLA
        JMP ERROR           /ASCII IS GREATER THAN 267.
        ISZ TEMP            /INCREMENT STORAGE POINTER.
        ISZ DIGCTR          /4 DIGITS YET?
        JMP NEWDIG          /NO: GET ANOTHER.
/YES: PACK THE 4 DIGITS INTO ONE NUMBER.
PACK,   TAD TEMP1           /SET POINTER TO STORAGE LOC.
        DCA TEMP
        DCA HOLD            /CLEAR LOCATION HOLD.
        TAD M4              /SET COUNTER FOR 4 DIGITS.
        DCA DIGCTR
DIGPCK, TAD HOLD            /CONTENTS OF 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 AC IN HOLD.
        ISZ TEMP            /INCREMENT STORAGE POINTER.
        ISZ DIGCTR          /PACKED 4 DIGITS YET?
        JMP DIGPCK          /NO: PACK ANOTHER.
        TAD HOLD            /YES: STORE PACKED NUMBER.
        DCA I BUFFPT
        TAD I BUFFPT        /NEGATIVE INPUT?
        TAD K4000
        SMA CLA
        JMP ERROR           /YES: REJECT ENTRY.
        ISZ AMOUNT          /NO: COUNT THE ENTRIES.
        ISZ BUFFPT          /SET UP FOR  A NEW ENTRY.
        JMP ACCEPT          /GET A NEW ENTRY.
```

Figure 5-14B. Sample Program: Initialization and Input Coding

```
/PUT THE NUMBERS IN INCREASING ORDER.
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 THE POINTERS
         DCA X1           /(X1 AND X2) TO THE
         TAD BUFF         /PROPER DATA LOCATIONS.
         IAC              /X2=X1+1
         DCA X2
TEST,    TAD I X2         /COMPARE X1 AND X2.
         CIA
         TAD I X1
         SMA SZA CLA      /REVERSE ENTRIES IF
         JMS REVERSE      /X2 IS LESS THAN X1.
         ISZ X1           /INCREMENT THE POINTERS.
         ISZ X2
         ISZ TALLY        /DONE COMPARING YET?
         JMP TEST         /NO: COMPARE MORE ENTRIES.
         TAD FLAG         /YES: IS FLAG SET?
         SZA CLA
         JMP ORDER        /YES: MAKE ANOTHER PASS.
         JMP PRINT        /NO: TYPE THE ORDERED DATA.
REVERSE,Ø                 /SUBROUTINE TO SWITCH X'S.
         TAD I X1
         DCA HOLD
         TAD I X2
         DCA I X1
         TAD HOLD
         DCA I X2
         CLA CLL CMA      /SET FLAG WHENEVER
         DCA FLAG         /A SWITCH IS MADE.
         JMP I REVERSE
```

Figure 5-14C. **Sample Program: Ordering Coding**

```
/PRINT OUT THE ORDERED NUMBERS
PRINT,  JMS CRLF          /RETURN THE CARRIAGE.
        TAD BUFF          /SET THE BUFFER POINTER.
        DCA BUFFPT
        TAD AMOUNT        /SET LIMIT FOR OUTPUT.
        CIA
        DCA PRNTCT
ANOTHR, JMS CRLF          /RETURN CARRIAGE.
        TAD M4            /COUNT THE DIGITS OUTPUT.
        DCA DIGCTR
        DCA HOLD          /CLEAR HOLD LOCATION.
        TAD I BUFFPT      /GET A CHARACTER.
        CLL RAL           /ROTATE INTO CLEARED LINK.
MORE,   TAD HOLD          /ADD HOLD TO AC.
        RAL               /ROTATE THREE TIMES LEFT.
        RTL
        DCA HOLD          /STORE AC IN HOLD.
        TAD HOLD
        AND MASK7         /MASK OUT FIRST 9 BITS.
        TAD K260
        JMS TYPE          /TYPE OUT ONE DIGIT.
        ISZ DIGCTR        /TYPED 4 DIGITS?
        JMP MORE          /NO: TYPE ANOTHER.
        ISZ BUFFPT        /YES: INCREMENT BUFFER LOC.
        ISZ PRNTCT        /TYPED ALL ENTRIES?
        JMP ANOTHR        /NO: TYPE ANOTHER ENTRY.
        JMS CRLF          /YES: RETURN CARRIAGE AND
        JMP START         /ACCEPT MORE NUMBERS TO SORT.
ERROR,  CLA
        TAD QUEST
        JMS TYPE
        JMP ACCEPT        / DISREGARDS ILLEGAL ENTRY.
END=.
```

Figure 5-14D. Sample Program: Output Coding

```
*100
TYPE,   0               /TYPE OUTPUT SUBROUTINE.
        TSF
        JMP .-1
        TLS
        CLA
        JMP I TYPE
CRLF,   0               /CARRIAGE RETURN&LINE FEED.
        TAD K215
        JMS TYPE
        TAD K212
        JMS TYPE
        JMP I CRLF
LISN,   0               /LISN INPUT SUBROUTINE.
        KSF
        JMP .-1
        KRB
        TLS
        JMP I LISN
BUFF,   END
BUFFPT, 0
M4,     7774
DIGCTR, 0
TEMP1,  .+2
TEMP,   0
        0
        0
        0
        0
MDOLAR, 7534
M10,    -10
K4000,  4000
HOLD,   0
M260,   -260
AMOUNT, 0
FLAG,   0
TALLY,  0
X1,     0
X2,     0
PRNTCT, 0
MASK7,  7
K260,   260
K212,   212
K215,   215
QUEST,  277
$
```

Figure 5-14E. Sample Program: Subroutines and Constants Coding

PROGRAM INTERRUPT FACILITY

The running time of programs using input and output routines is primarily made up of the time spent in waiting for the device to accept or transmit information. Specifically, this time is spent in loops such as:

```
TSF
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 can be removed and useful routines can be included to use this 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 family computers.

The following two instructions control the interrupt facility.

Mnemonic	Octal	Operation
ION	6001	Turn interrupt facility on.
IOF	6002	Turn interrupt facility off.

The program interrupt facility allows a running program to proceed until a peripheral device connected to the interrupt facility sets its ready flag. The running program is often referred to as the *background program.* Whenever a flag is set to 1 by a device that is connected to the interrupt facility, the PDP-8 completes execution of the instruction in progress and then acknowledges the interrupt. The interrupted computer will automatically execute a JMS 0 instruction. The result of this action is that the program counter register, which contains the address of the next instruction to be performed in the main program, is stored in location 0. The instruction in location 1 is then performed, which usually initiates a service routine for the peripheral device.

The service routine, sometimes called the *foreground program,* is usually contained elsewhere in memory and an indirect jump to the start of the program is contained in location 1. The service routine is terminated by a JMP I 0 instruction, to return to the background program.

The occurrence of an interrupt disables the facility from further interrupts. The ION instruction must be included in the interrupt service routine to enable further interrupts. This is usually done immediately before returning to the background program. *The ION instruction does not take effect until one instruction following it has been completed.*

Thus, the ION instruction is usually followed by the JMP I 0.

SERVICE ROUTINE	INTRTE,	—	
	.		
	.		
	.		
	EXIT,	ION	Enables interrupt facility
	.	JMP I 0	again after execution of
	.		JMP I 0.
	.		
	0000,	0202	
	0001,	JMP INTRTE	
	.		
	.		
	0200,	ISZ COUNT	
	0201,	TAD A	Interrupt request occurred during the execution of this instruction; caused a JMS 0 to be executed immediately after completion of the TAD A instruction execution.
	0202,	RAL	Interrupt is turned on before this instruction is executed.

Programming an Interrupt

The program presented in Figure 5-15 includes a program to rotate one bit through the accumulator as the background program. The foreground program, which is initiated by the service routine, accepts ASCII characters from the Teletype unit and, upon receipt of the ASCII code for a period, prints out the characters which have been stored.

The program begins with an *initialization routine* to set up buffer space to store the incoming characters and to set 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 but 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. The accumu-

lator and link displays will exhibit a quickly rotating light while waiting for an interrupt to initiate the foreground program.

The first instruction to be executed after an *interrupt request* is an automatic JMS 0, thereby storing the return address to the background program in location 0. The program then executes the instruction in location 1 which is an indirect jump to the service routine location 2 which contains SERV).

Since an interrupt occurs during the running of the background program, the *service routine* must save the active registers of the background program. The service routine stores the link and accumulator, so that they may be restored after the service routine is completed.

The service routine must determine the source of the interrupt request by determining which device flag is set and then jump to a routine to service the appropriate device. The service routine ends with a HLT, which would be encountered only if the service routine is entered and neither flag is set—a condition that should never exist.

The *keyboard input routine* is entered when the keyboard flag is set. The flag is cleared to prevent further interruptions when the interrupt system is re-enabled. If the mode is not set for input, program control is transferred to the background program. Otherwise, the program accepts the character (KRB), acknowledges its receipt by printing it on the printer (TLS), and stores it in the buffer. (Notice that no KSF, JMP .—1 loop is necessary.) The routine then checks for the ASCII code for a period, returning to the background program if it is 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, an interrupt will be requested when this action is complete, and the stored ASCII codes will be typed out by the printer output routine.)

The *printer output routine* is entered when its flag is set. The routine clears the device flag and checks for output mode. When in output mode, the routine prints one character from the buffer. (Notice that 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 returns to the background program.

The *exit routine* to return to the background program must restore the link and accumulator to the values at the time of interrupt. The program turns the interrupt on (ION) and then returns to the rotate program via a JMP I 0 instruction (location 0 contains the value of the program counter when the interrupt occurred). The ION instruction does not take effect until the instruction following it has been executed.

The *constants* used by the routines conclude the program listing.

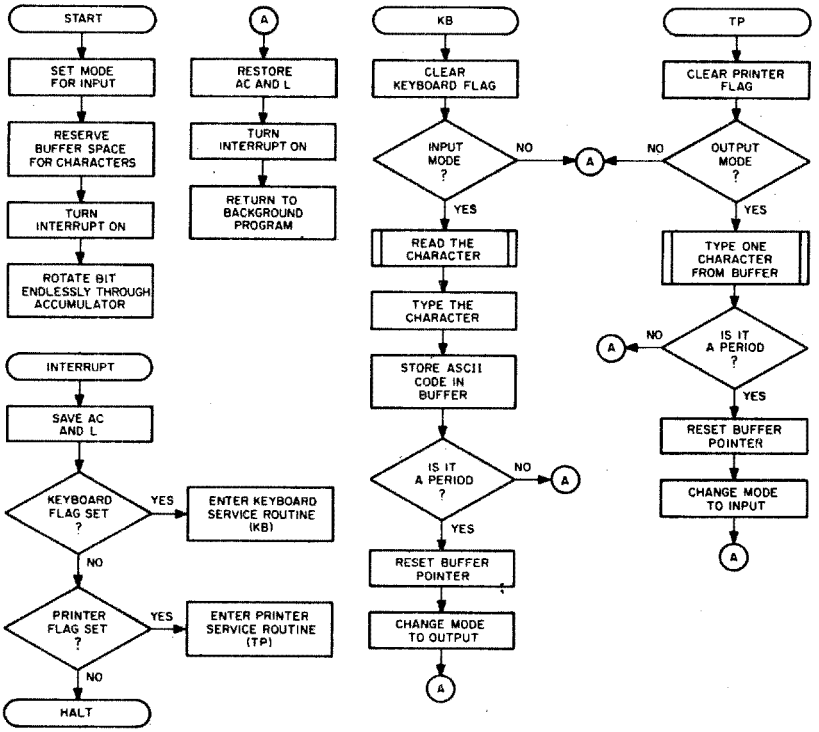


Figure 5-15A. Program to Operate on Program Interrupt Facility: Flowchart

```
*0
/FIRST INSTRUCTIONS AFTER AN INTERRUPT.
        0                   /STORES RETURN ADDRESS.
        JMP I 2             /JUMP TO SERVICE ROUTINE.
        SERV
*200
/INITIALIZATION ROUTINE.
START,  CLA CLL
        DCA MODE            /SET MODE FOR INPUT.
        TAD K1777
        DCA BUFFER          /SET UP BUFFER SPACE.
        ION                 /TURN ON INTERRUPT FACILITY.
/BACKGROUND PROGRAM
ROTATE, CLA CLL IAC         /SET ONE BIT IN AC.
        ISZ COUNT           /ROTATING DELAY INSTRUCTIONS.
        JMP .-1
        ISZ COUNT
        JMP .-1
        RAL                 /ROTATE BIT LEFT.
        JMP ROTATE+1        /EXECUTE DELAY INSTRUCTIONS.
/SERVICE ROUTINE
SERV,   DCA AC              /SAVE ACCUMULATOR.
        RAL
        DCA L               /SAVE LINK.
        KSF                 /KEYBOARD INTERRUPT?
        SKP                 /NO: CHECK PRINTER.
        JMP KB              /YES: SERVICE KEYBOARD.
        TSF                 /PRINTER INTERRUPT?
        SKP                 /NO: SKIP PRINTER ROUTINE JMP.
        JMP TP              /YES: SERVICE THE PRINTER.
        HLT                 /FATAL HALT IF NO FLAG SET.
/KEYBOARD INPUT ROUTINE
KB,     KCC                 /CLEAR KEYBOARD FLAG.
        TAD MODE            /INPUT MODE?
        SZA CLA
        JMP EXIT            /NO: RETURN TO BACKGROUND.
        ISZ BUFFER          /YES: INCREMENT BUFFER.
        KRB                 /READ THE CHARACTER.
        TLS                 /ACKNOWLEDGE ON THE PRINTER.
        DCA I BUFFER        /STORE CHARACTER.
        TAD I BUFFER        /IS CHARACTER A PERIOD?
        TAD MPER
        SZA CLA
        JMP EXIT            /NO: RETURN TO BACKGROUND.
        TAD K1777           /YES: RESET BUFFER
        DCA BUFFER          /TO TYPE THE CHARACTERS.
        CLA CMA
        DCA MODE            /SET MODE FOR OUTPUT AND
        JMP EXIT            /RETURN TO BACKGROUND.
```

Figure 5-15B. Program to Operate on Program Interrupt Facility: Coding

```
/PRINTER OUTPUT ROUTINE
TP,     TCF                 /CLEAR PRINTER FLAG.
        TAD MODE            /OUTPUT MODE?
        SNA CLA
        JMP EXIT            /NO: RETURN TO BACKGROUND.
        ISZ BUFFER          /YES: INCREMENT BUFFER.
        TAD I BUFFER        /GET CHARACTER FROM BUFFER.
        TLS                 /TYPE IT OUT.
        TAD MPER            /IS CHARACTER A PERIOD?
        SZA CLA
        JMP EXIT            /NO: RETURN TO BACKGROUND.
        DCA MODE            /YES: SET MODE FOR INPUT,
        TAD K1777           /RESET BUFFER, AND
        DCA BUFFER
        JMP EXIT            /RETURN TO BACKGROUND.
/ROUTINE FOR RETURNING TO BACKGROUND PROGRAM.
EXIT,   TAD L               /RESTORE LINK.
        CLL RAR
        TAD AC              /RESTORE ACCUMULATOR.
        ION                 /RE-ENABLE INTERRUPT FACILITY.
        JMP I Ø             /Ø CONTAINS RETURN ADDRESS.
COUNT,  Ø
MODE,   Ø
K1777,  1777
BUFFER, Ø
AC,     Ø
L,      Ø
MPER,   -256
$
```

Figure 5-15B (cont.) Program to Operate on Program Interrupt Facility: Coding

Advanced Use of the Program Interrupt Facility

The following paragraphs entitled "Multiple Device Interrupt Programming" and "A Software Priority Interrupt System" are intended for programmers making extensive use of the program interrupt facility. The reader who merely desires a general knowledge of the interrupt facility may omit these topics from his reading.

MULTIPLE DEVICE INTERRUPT PROGRAMMING

Many programming applications use the interrupt facility to service several devices. For example, a PDP-8 may use the program 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 the interrupt request (i.e., which device flag is set). The following example is an instruction sequence which uses dummy skip instructions to determine the device requesting the interrupt.

```
DASF
SKP
JMP SERVA       /DEVICE A REQUESTED THE INTERRUPT
DBSF
SKP
JMP SERVB       /DEVICE B REQUESTED THE INTERRUPT
DCSF
SKP
JMP SERVC       /DEVICE C REQUESTED THE INTERRUPT
  .
  .
  .
DNSF
SKP
JMP SERVN       /DEVICE N REQUESTED THE INTERRUPT
```

The dummy skip instructions (DASF, DBSF, etc.) are skip-on-flag instructions for each of the devices in the interrupt system. (Usually, these instructions skip the next instruction if the device flag is set to a 1. Instructions for some devices, however, may skip if the flag is a 0, i.e., skip-on-non-flag. Instructions of this type should *not* be followed by the unconditional SKP instruction.) Because of the predominance of SKP instructions, the instruction sequence which determines the source of an interrupt request is often called a *skip chain.*

As the previous example implies, the skip chain may be enlarged to test for almost any number of device flags. However, an important limitation upon the size of a skip chain is imposed by the devices which the chain serves. The chain must be traversed and the device serviced before the desired information is lost. High-speed devices, such as magnetic tapes, disks, or drums, must be serviced quickly or the information will not be available. On the other hand, low-speed devices, such as paper tape readers and punches, have a relatively long period of time in which they may be serviced before any loss of information occurs. The programmer must take these time factors into account whenever he programs a multiple device interrupt system.

High-speed devices should be tested and serviced first by the skip chain. Thus, the chain should begin by checking the device flags of the high-speed devices of the system, such as DECtape and DECdisk, and conclude by checking the flags of the low-speed devices such as the Teletype keyboard or paper tape readers and punches. Thus, a high-speed device is not required to wait while a long set of device flags is checked.

In the case of two interrupt requests occurring simultaneously, the high-speed device is serviced first. The second device may be serviced simply by returning to the background program and waiting for another request, or it may be serviced by checking the flags through a skip chain before returning to the background program.

A SOFTWARE PRIORITY INTERRUPT SYSTEM

A service routine may be written in such a way that a priority of device interrupts is established through software. The programmer sets priorities by allowing the service routine for a particular device to be interrupted by the interrupt request of a higher priority device. This may be necessary in a system that includes high-speed devices that retain information for a short time and that require immediate attention. A lower-priority device would be serviced by a routine that would re-enable the interrupt facility at its beginning. The higher-priority device would not re-enable the interrupt until it had completed its task.

The service routine of a multiple interrupt system must include instructions to save the contents of the accumulator and link for each interrupt. The contents of the program counter must also be removed from location 0 before a new interrupt occurs. To save the contents of these active registers, the programmer must establish a "push down" list of accumulators, links, and program counters. The instructions to handle such a list could be the following.

```
SERV,  DCA I ACPTR      /SAVE AC
       RAL
       DCA I LPTR       /SAVE L
       TAD 0
       DCA I PCPTR      /SAVE PROGRAM COUNTER
       ISZ ACPTR        /INCREMENT POINTERS FOR
       ISZ LPTR         /NEXT USE.
       ISZ PCPTR
```

The above instructions would then be followed by the chain of instructions to test the flags and jump to the service routine for the device whose flag is set.

The service routine for a high-priority device in such a system is similar to the preceding program examples, in that the interrupt facility is not enabled until processing is complete.

The low-priority device would be serviced by a routine which would re-enable the interrupt system immediately after clearing its device flag. A low-priority keyboard routine might start as follows.

```
KB,   KCC          /CLEAR DEVICE FLAG.
      ION          /TURN INTERRUPT ON.
      TAD MODE     /SERVICE PROGRAM WILL
                   /PROCEED UNTIL
        .          /INTERRUPTED OR COMPLETED.
        .
        .
```

Thus, the keyboard could be interrupted during the process of transmitting a character. The service routine would save the status of the routine by storing the program counter accumulator and link in the push-down list described before. After the high priority device had been serviced, the following exit routine would return control to the low priority device.

```
EXIT,  IOF          /INTERRUPTS NOT ALLOWED.
       CLA CMA      /FOLLOWING INSTRUCTIONS WILL
       TAD ACPTR    /DECREASE THE POINTERS
       DCA ACPTR    /BY 1 AND THEN
       CLA CMA      /USE THEM.
       TAD LPTR
       DCA LPTR
       CLA CMA
       TAD PCPTR
       DCA PCPTR
       TAD I PCPTR
       DCA 0
       TAD I LPTR
       CLL RAR
       TAD I ACPTR
       JMP I 0
```

Through this approach, multiple interrupts could occur and would be serviced on a priority basis specified by the programmer.

Program Interrupt Demonstration Program

The program presented in Figures 5-16A through 5-16F is a demonstration program to run on the program interrupt facility. It contains a bit rotating program, the speed and direction of which is determined by the switch register settings. (This same program is presented in Exercise 10 of Chapter 3.) The foreground program is the ordering program which was given in Figure 5-14. This program has the capacity to accept 4-digit positive octal input from the Teletype keyboard, automatically terminating each 4-digit number with a carriage return and line feed. Upon receipt of a typed dollar sign ($), the program will place the data in increasing order, and type the ordered data on the printer. The program will not accept negative numbers or nonoctal digits.

The example is useful as a demonstration and illustration of the power of program interrupt as the computer will seem to be performing two tasks at the same time. The programmer knows that this is not possible and that the two tasks are sharing the computer time; however, the appearance indicates simultaneous actions.

When an *interrupt request occurs,* locations 0, 1, and 2 of page 0 provide the storage for the program counter and the jump to the service routine.

The *constants* which are stored on page 0 include four "software switches" to record the conditions within the running program. MODE is used to specify the input or output status of the running program. SW1 is used to signal the input of the first digit of a new number (0) or the input of successive digits of a continuing number (1). SW2 is used to control the input and output of data by separating each number with a carriage return and line feed. SW3 is used to bypass the output mode and allow the typing of carriage returns, line feeds, and question marks (to denote errors of input) during the input of data.

Other constants include pointers which permit off-page jumps to routines elsewhere in memory. The constant BUFF is a pointer to the storage area for the packed input numbers. It contains END which is defined as END = . in the last line of Figure 5-16F. Thus, the buffer is all of memory following the last instruction.

```
*0
/FIRST INSTRUCTIONS AFTER AN INTERRUPT.
        0
        JMP I 2
        SERV
*50
/CONSTANTS STORED ON PAGE 0.
MODE,   0       /INPUT=0; OUTPUT=-1.
SW1,    0       /NUMBER STATUS SWITCH
SW2,    0       /OUTPUT:CR=0,LF=-1,DATA=1.
SW3,    0       /MODE BYPASS SWITCH
AC,     0       /SAVE AC AND
L,      0       /L DURING AN INTERRUPT.
PRINTR, TP      /FOLLOWING ARE POINTERS FOR
KEYBRD, KB      /THE RESPECTIVE ROUTINES.
ORDPTR, ORDER
EXITPT, EXIT
M7000,  1000    /ORDER SUB-PROGRAM CONSTANTS
BUFF,   END
BUFFPT, 0
M4,     7774
DIGCTR, 0
TEMP1,  .+2
TEMP,   0
        0
        0
        0
        0
MDOLAR, 7534
M10,    -10
HOLD,   0
HOLDL,  0
M260,   7520
AMOUNT, 0
FLAG,   0
TALLY,  0
X1,     0
X2,     0
PRNTCT, 0
K7,     7
K260,   260
K212,   212
K215,   215
QUEST,  277
K4000,  4000
```

Figure 5-16A. Program Interrupt Demonstration Program (Constants Located on Page 0)

```
/SUBROUTINES STORED ON PAGE Ø.
CR,     TAD K215        /CARRIAGE RETURN ROUTINE
        TLS
        CLA CMA
        DCA SW2         /SET SW2 FOR A LF.
        JMP I EXITPT
LF,     TAD K212        /LINE FEED ROUTINE
        TLS
        CLA
        TAD SW3
        SNA CLA
        JMP SW2SET
        DCA SW3         /TURN OFF MODE BYPASS.
        DCA SW2         /SET SW2 FOR CR.
        JMP I EXITPT
SW2SET, CLA CLL IAC
        DCA SW2         /SET SW2 FOR DATA.
        JMP I EXITPT
```

Figure 5-16B. Program Interrupt Demonstration Program (Subroutines Located on Page 0)

The location TEMP is used as a pointer to the four locations after it; these locations are used for the storage of incoming ASCII codes.

The *subroutines* CR and LF type the carriage returns and line feeds when called for by the setting of SW2.

The program begins in location 200 as shown in Figure 5-16C. The *initialization routine* sets each of the software switches to zero. After the initialization is completed, the interrupt facility is turned on and the background program is started.

The *rotate subprogram* begins by checking the setting of the switch register to determine the direction of rotation. The value of bit 0 specifies a rotate right when it is a 0, or a rotate left when it is a 1. The last nine bits of the switch register determine the speed of the rotation. They are stored in COUNT and determine the number of passes through the ISZ COUNTR, JMP .—1 loop.

The BEGIN routine determines the speed and direction and the GO routine establishes the bit position after each check of the switch register setting. The INSTR routine executes the delay and the rotation of the bit.

```
*200
/NEXT INSTRUCTIONS INITIALIZE THE PROGRAM.
/FURTHER INITIALIZATION DONE BY RESTART.
START,  IOF      /INTERRUPT OFF DURING INITIALIZATION.
        CLA CLL
        DCA MODE
        DCA SW1
        DCA SW2
        DCA SW3
        TAD BUFF
        DCA BUFFPT
        DCA AMOUNT
        ION
/ROTATE SUB-PROGRAM BEGINS HERE.
ROTATE, CLA CLL CML
BEGIN,  DCA SAVEAC
        RAL
        DCA SAVEL
        TAD K7000       /ALWAYS SET BITS 0,1 AND 2.
        OSR
        DCA COUNT       /MAX COUNT IS -1000.
        OSR
        RAL             /PUT BIT 0 IN LINK.
        SZL CLA
        JMS LEFT
        JMS RIGHT
        CLL
GO,     TAD SAVEL
        RAR
        TAD SAVEAC
INSTR,  HLT             /OVERWRITTEN BY RAR OR RAL.
        ISZ COUNTR
        JMP INSTR+1
        ISZ COUNT
        JMP INSTR+1
        JMP BEGIN
SAVEAC, 0
SAVEL,  0
K7000,   7000
COUNTR, 0
COUNT,  0
/SUBROUTINES TO DETERMINE DIRECTION.
LEFT,   0
        ISZ LEFT        /SKIP INSTR AFTER JMS LEFT.
        TAD KRAL
        DCA INSTR
        JMP I LEFT      /STORE 'RAL' IN 'INSTR'.
RIGHT,  0
        TAD KRAR
        DCA INSTR       /STORE 'RAR' IN 'INSTR'.
        JMP I RIGHT
KRAR,   RAR
KRAL,   RAL
```

Figure 5-16C. Program Interrupt Demonstration Program (Initialization Routine and Rotate Subprogram)

```
/SKIP CHAIN TO SERVICE ROUTINES
SERV,    DCA AC            /SAVE AC AND L
         RAL               /DURING AN INTERRUPT.
         DCA L
         TSF               /IS INTERRUPT CAUSED
         SKP               /BY TELETYPE PRINTER?
         JMP I PRINTR      /SERVICE TELETYPE PRINTER.
         KSF               /IS INTERRUPT CAUSED
         SKP               /BY KEYBOARD?
         JMP I KEYBRD      /SERVICE KEYBOARD.
         HLT               /SHOULD NEVER REACH HERE.
/ORDER SUB-PROGRAM
ORDER,   CLA CLL
         TAD AMOUNT        /SET TALLY FOR COMPARISONS.
         CIA
         IAC
         DCA TALLY
         DCA FLAG          /CLEARS FLAG FOR EACH PASS
         TAD BUFF
         DCA X1
         TAD BUFF
         IAC
         DCA X2
TEST,    TAD I X2
         CIA
         TAD I X1          /IS X2 LESS THAN X1?
         SPA SNA CLA
         JMP INCPTR        /NO: DON'T REVERSE.
REVERSE, TAD I X1          /YES: REVERSE X2 AND X1.
         DCA HOLD
         TAD I X2
         DCA I X1
         TAD HOLD
         DCA I X2
         CLA CLL CMA       /SET FLAG TO SIGNAL
         DCA FLAG          /THAT A REVERSE WAS DONE
INCPTR,  ISZ X1            /INCREMENT X POINTERS.
         ISZ X2
         ISZ TALLY         /COMPARED ALL ENTRIES?
         JMP TEST          /NO: COMPARE NEXT X'S.
         TAD FLAG          /YES: ORDER DONE YET?
         SZA CLA
         JMP ORDER         /NO: MAKE ANOTHER PASS.
         CLA CMA
         DCA MODE          /YES: SET OUTPUT MODE.
         TAD BUFF          /SET POINTER TO FIRST ENTRY.
         DCA BUFFPT
         DCA SW1           /CLEAR NUMBER STATUS SWITCH.
         TAD AMOUNT        /SET A TALLY FOR OUTPUT.
         CIA
         DCA PRNTCT
         TLS               /TO TRIGGER NEXT INTERRUPT.
         JMP I EXITPT
```

Figure 5-16D. Program Interrupt Demonstration Program (Skip Chain to Service Routines and Order Subprogram)

The service routines are reached through the *skip chain* in Figure 5-16D. These service routines are located on a separate memory page and are reached through pointer words stored on page 0.

The *order subprogram* is initiated when input is complete. The routine sorts the positive octal numbers stored in the buffer into increasing order. The technique is the same as that used previously in the program of Figure 5-14.

These routines conclude the instructions placed on page 1 of memory. The service routines begin on the next memory page.

```
*400
KB,     KCC
        TAD MODE        /INPUT MODE DOES NOT
        SZA CLA         /HONOR KEYBOARD REQUEST.
        JMP EXIT
        TAD SW1
        SZA CLA         /CHECK FOR A NEW NUMBER
        JMP CNTDIG      /OR A CONTINUED DIGIT.
        TAD M4
        DCA DIGCTR
        TAD TEMP1
        DCA TEMP
CNTDIG, KRS             /READ KEYBOARD CHARACTER.
        TLS             /TYPE IT ON PRINTER.
        DCA I TEMP      /STORE DIGIT TEMPORARILY.
CHECK,  TAD I TEMP
        TAD MDOLAR      /CHECK FOR TERMINAL $
        SNA CLA
        JMP I ORDPTR
        TAD I TEMP
        TAD M260        /ASCII LESS THAN 260?
        SPA
        JMP ERROR       /YES: ERROR.
        TAD M10         /NO: SUBTRACT 10.
        SPA             /GREATER THAN 267?
        JMP LEGAL       /NO: DIGIT IS LEGAL.
ERROR,  CLA IAC         /NOT AN OCTAL NUMBER.
        DCA SW3         /SET TO TYPE ?,CR,LF.
        DCA SW1         /SET FOR A NEW NUMBER.
        TLS
```

Figure 5-16E. Program Interrupt Demonstration Program (Keyboard Service Routine)

The service routines begin in location 400 as presented in Figure 5-16E. The *keyboard service routine* honors an interrupt only if the mode is set to 0. The routine uses SW1 to specify number status, resetting the digit counter if a new number is started. The routine accepts the incoming digit and types it on the printer.

The input of a non-octal character or terminal symbol ($) is checked after each character is received. Whenever the terminal $ is received by the program, control is transferred to the order subprogram. If the character is not an octal digit, the program types a question mark and

```
        JMP  EXIT
LEGAL,  CLA  CMA          /SET SW1 TO SIGNAL
        DCA  SW1          /A CONTINUED NUMBER.
        ISZ  TEMP
        ISZ  DIGCTR       /HAVE 4 DIGITS?
        JMP  EXIT         /NO: GET NEXT DIGIT.
PACK,   TAD  TEMP1        /YES: PUT NUMBER TOGETHER.
        DCA  TEMP
        DCA  HOLD
        TAD  M4
        DCA  DIGCTR
DIGPCK, TAD  HOLD         /NEXT 7 INSTRUCTIONS
        RAL  CLL          /COMBINE THE 4 OCTAL DIGITS
        RTL               /INTO ONE MEMORY WORD.
        TAD  I TEMP
        TAD  M260
        DCA  HOLD
        ISZ  TEMP
        ISZ  DIGCTR
        JMP  DIGPCK       /PACK ANOTHER DIGIT.
        TAD  HOLD
        DCA  I BUFFPT     /STORE THE OCTAL NUMBER.
        TAD  I BUFFPT     /CHECK FOR NEGATIVE ENTRY.
        TAD  K4000
        SPA  CLA
        JMP  NOTNEG       /ENTRY IS LEGAL.
        IAC               /SET SW3 TO TYPE A "?".
        JMP  DISALO       /DISALLOW NEGATIVE ENTRY.
NOTNEG, ISZ  BUFFPT
        ISZ  AMOUNT       /COUNT THE ENTRIES.
        CLA  CMA          /TYPE A CR,LF
DISALO, DCA  SW3          /AFTER THE ENTRY.
        DCA  SW1          /CLEAR SW1 FOR NEXT PASS.
        TLS
        JMP  EXIT
```

Figure 5-16E (cont.). Program Interrupt Demonstration Program (Keyboard Service Routine)

ignores the whole entry in which it occurred. The program requests a new 4-digit number by typing a carriage return and line feed.

When four digits have been received, the pack routine combines the four ASCII codes into one octal number and stores the number in the buffer. If the number is negative, the entry is disallowed. (The next packed number will be deposited in the same location, thus destroying the negative number.) A running count of the entries is kept and later used when ordering is performed. The software switches are finally set to return the carriage for the next number to be input.

The *printer service routine* in Figure 5-16F is used to output the ordered numbers and to type carriage returns, line feeds and question marks during input. The "mode bypass switch", SW3, is used in conjunction with the subroutines on page 0 to type carriage returns and line feeds. If the program is in output mode, the results of the sorting will be typed by the printer service routine. The stored numbers will be unpacked, translated into ASCII codes and typed out.

```
/PRINTER SERVICE ROUTINE
TP,     TCF
        TAD SW3          /CHECK MODE-BYPASS-SWITCH.
        SNA CLA
        JMP MODCHK       /NO BYPASS, CHECK MODE.
        TAD SW3
        SPA CLA
        JMP RETLF        /MODE-BYPASS, DO A CR & LF.
        TAD QUEST        /MODE BY PASS SET FOR "?".
        TLS
        CLA CMA          /AFTER TYPING THE ? SET SW3.
        DCA SW3          /TO TYPE THE CR &LF.
        JMP EXIT
MODCHK, TAD MODE         /CHECK MODE.
        SNA CLA
        JMP EXIT         /INPUT MODE: IGNORE REQUEST!
RETLF,  TAD SW2          /OUTPUT MODE: BEGIN OUTPUT.
        SNA CLA
        JMP CR           /FIRST PASS,CARRIAGE RETURN.
        TAD SW2
        SPA CLA
        JMP LF           /LINE FEED ON SECOND PASS.
DATA,   TAD SW1          /PRINT DATA ON THIRD PASS.
        SZA CLA          /NEW NUMBER?
        JMP DIGTYP       /NO: TYPE ANOTHER DIGIT.
        TAD M4           /YES: RESET DIGIT COUNTER.
        DCA DIGCTR
        DCA HOLD         /CLEAR THESE LOCATIONS.
        DCA HOLDL
```

Figure 5-16F. Program Interrupt Demonstration Program (Printer Service, Exit and Restart Routines)

When the action of either service routine is completed, the *exit routine* returns control to the background program until the next interrupt occurs. This routine restores the accumulator and link and turns the interrupt on before jumping to the interrupted background program.

The *restart routine* initializes the software switches for the ordering of a new set of data input from the keyboard. The mode is set for input and control returns to the background program until new input is supplied.

```
          TAD I BUFFPT     /GET NUMBER TO BE PRINTED.
DIGTYP,   TAD HOLDL
          CLL RAL          /ROTATE INTO THE LINK.
          TAD HOLD         /THESE TWELVE INSTRUCTIONS
          RAL              /PRINT OUT THE NEXT DIGIT.
          RTL
          DCA HOLD
          RAR
          DCA HOLDL
          TAD HOLD
          AND K7
          TAD K260
          TLS
          CLA CMA          /SET FOR ANOTHER DIGIT.
          DCA SW1
          ISZ DIGCTR       /WORD COMPLETE?
          JMP EXIT         /NO: GET ANOTHER DIGIT.
          CLA              /YES: SIGNAL A NEW NUMBER.
          DCA SW1
          DCA SW2          /TYPE A CR & LF.
          ISZ BUFFPT
          ISZ PRNTCT       /ALL NUMBERS PRINTED?
          JMP EXIT         /NO: WAIT FOR NEW NUMBER.
RESTART,  CLA CLL          /YES: SET UP FOR NEW INPUT.
          DCA MODE
          DCA SW1
          DCA SW2
          CMA              /SET BYPASS SWITCH
          DCA SW3          /TO TYPE A CR & LF.
          TAD BUFF
          DCA BUFFPT
          DCA AMOUNT
          JMP EXIT
/ROUTINE TO RETURN TO ROTATE PROGRAM
EXIT,     CLA CLL
          TAD L
          RAR
          TAD AC
          ION
          JMP I 0          /0 CONTAINS RETURN ADDRESS.
END=.
$
```

Figure 5-16F (cont.). Program Interrupt Demonstration Program (Printer Service, Exit and Restart Routines)

DATA BREAK

Programmed transfers of data, including program interrupt transfers, pass through the accumulator. The accumulator must therefore be cleared while the transfer is performed. 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 (standard on all PDP-8 family computers except the PDP-8/L). Use of these facilities permits an external device to insert or extract words from the computer memory, bypassing all program control. Because the computer program has no cognizance of the transfers made in this manner, the program must check for the presence of this data prior to its use. The data break is particularly well-suited for devices that transfer large amounts of data in block form, for example, random access disk files, high-speed magnetic tape systems, or high-speed drum memories.

The data break facility allows a peripheral device to transfer information directly with the PDP-8 core memory on a "cycle stealing" basis. Input/output equipment operating at high speed can transfer information with the computer through the data break facility more efficiently than through programmed means. In contrast to programmed operations, the data break facilities permit an external device to control information transfers.

Data breaks are of two basic types: single-cycle and 3-cycle. In a single-cycle data break, registers in the device specify the core memory address of each transfer and count the number of transfers to determine the end of the block. In the 3-cycle data break, two computer memory locations perform these functions, simplifying the device interface by omitting the two hardware registers.

In general terms, to initiate a data break transfer of information, the control must do the following tasks.

1. Specify the affected address in core memory.
2. Provide the data word.
3. Indicate direction of data word transfer.
4. Indicate a single-cycle or 3-cycle break.
5. Request the data break.

Single-cycle data break is a device-controlled transfer of information which steals one memory cycle from the PDP-8. When a device that is connected to the data break facility wishes to transfer a word of information, it requests a data break. The PDP-8 computer completes the current instruction and enters the break state. For one memory cycle, the device has access to the PDP-8 and transfers the data word into (or out of) the memory unit at a location specified by the device control. The device controls the number of words to be transferred and the locations in memory to be affected by the transfer. The device continues to request breaks and transfers one word per break until the block transfer is complete.

The 3-cycle data break facility provides a current address register and a word count register in core memory for each connected device, thus eliminating the necessity for registers in the device control. When several devices are connected to the facility, each is assigned a different pair of core locations for word count and current address, allowing interlaced operations of the devices. The device specifies the location of these registers in memory. Since these instructions are in memory, they may be loaded and unloaded without using IOT instructions.

The 3-cycle data break facility performs the following sequence of operations.

1. An address is read from the device to indicate the location of the word count register. This location specifies the number of words in the block yet to be transferred. The address is always the same for a given device.
2. The content of the specified word count register is read from memory and is incremented by 1. To transfer a block of n words, the word count is set to —n during the programmed initialization of the device. When this register is incremented to 0, a pulse is sent to the device to terminate the transfer.
3. The location after the word count register contains the current address register for the device transfer. The content of this register is set to 1 less than the location to be affected by the next transfer. To transfer a block beginning at location A, the register is originally set to A-1.
4. The content of the current address register is incremented by 1 and then used to specify the location affected by the transfer.

After the transfer of information has been accomplished through the data break facility, input data (or new output data) is processed, usually through the program interrupt facility. An interrupt is requested when the data transfer is completed and the service routine will process the information.

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 20 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.

NOTE TO READER

This chapter concludes the introduction to machine-language programming on the PDP-8. The remaining chapters deal primarily with software supplied by Digital Equipment Corporation as an aid to the programmer. Chapter 6 describes the software available to assist the PDP-8 programmer in writing machine-language programs. Later chapters describe specialized software systems and a conversational language, FOCAL, which can be used to write programs on the PDP-8.

The programming of advanced input/output devices and options (e. g., EAE, DECtape, DECdisk, A/D Converters) are beyond the scope of this publication. The applicable PDP-8 Users Handbook should be consulted when programming these devices.

Chapter 6

Operating the System Software

This chapter contains brief descriptions of the PDP-8 family systems programs and selected program operating procedures. The major portion of the chapter is devoted to detailed operating procedures for the most frequently used system software—the loaders, symbolic editor, the assemblers, the dynamic debugging programs, and FOCAL.

DESCRIPTIONS

A comprehensive package of system software accompanies each computer in the PDP-8 family. (Software is the collection of programs and routines associated with the computer.) The package contains many programs and routines furnished on punched paper tape or stored on DECtape in binary coded format, as well as associated manuals and documents describing the use and operation of each program and routine.

The system software, supplied by Digital Equipment Corporation, allows the programmer to write, edit, assemble, compile, debug, and run his programs, making the full data processing capability of the computer immediately available. System software comes from past, present, and continuing programming efforts of DEC programmers and users (see Chapter 11).

The system programs furnished with each computer in the PDP-8 family are those capable of operating with that specific computer and its I/O devices. If, for example, a computer has 4K of core memory, the set of programs in the accompanying software package are those designed to run in 4K of core memory, and the package accompanying computers with 8K of core memory include those programs capable of operating in 8K of core memory, i.e., both 4K and 8K programs, and so on.

Many of the frequently used software programs are briefly described on the following pages.

Symbolic Editor

The Symbolic Editor is a program which allows the programmer to prepare, edit, and generate a symbolic program tape online from the Teletype keyboard. Using Editor, the programmer may enter his symbolic program into core from the keyboard or paper tape reader, and then issue certain commands to edit his program. When used properly, Editor can substantially ease the labor of writing and editing symbolic programs and reduce the number of passes necessary to correct symbolic program tapes. Symbolic Editor is further described later in this chapter and in the DEC manual entitled *Symbolic Editor,* Order No. DEC-08-ESAB-D.

PAL III Symbolic Assembler

The PAL III Symbolic Assembler is a two-pass assembler which translates symbolic programs written in the PAL III symbolic language into binary-coded programs, producing the binary tape acceptable to the computer. The assembler offers an optional third pass which produces an octal/symbolic printout and/or punchout of the assembled program. The assembler is described in more detail later in this chapter and in *PAL III Symbolic Assembler,* Order No. DEC-08-ASAC-D.

MACRO-8 Symbolic Assembler

The MACRO-8 Symbolic Assembler accepts symbolic programs written in the MACRO-8 symbolic language and translates them into binary-coded programs in two passes. An optional third pass is available for an octal/symbolic assembly program listing. MACRO-8 is compatible with PAL III and has the following alditional features: user-defined macros, double-precision integers, floating-point constants, arithmetic and Boolean operators, literals, text facilities, and automatic off-page linkage generation. The MACRO-8 Symbolic Assembler is discussed later in this chapter and in *MACRO-8 Assembler,* Order No. DEC-08-CMAA-D.

8K SABR Symbolic Assembler

SABR is an advanced one-pass assembler for use with 8K to 32K words of core. The SABR language is similar to the assemblers above with many additional features, and differs from them in its operating procedures, pseudo-ops, assembled output (relocatable binary code), and execution of assembled programs. SABR is also used with 8K FORTRAN. For a complete description, refer to *8K SABR Assembler,* Order No. DEC-08-ARXA-D.

FORTRAN Compilers and Operating Systems

FORTRAN (FORmula TRANslation) is a problem-oriented language written mostly in mathematical terms and some English words. It

is especially suited for solving equations and making other mathematical calculations. For those who elect to write their programs in FORTRAN, DEC has 4K FORTRAN and 8K FORTRAN. Each is briefly explained below.

4K FORTRAN

4K FORTRAN consists of a compiler, debugging aid, and operating system. The one-pass compiler translates FORTRAN symbolic-language statements into binary code and produces a binary tape. The debugging aid (Symbol Print) lists the variables used and their locations in core and indicates the section of core used by the compiled program. The operating system loads and executes the compiled program. In addition, the operating system contains an extensive library of arithmetic function subprograms and I/O routines. Useful error messages are printed on the teleprinter when any error is detected by the compiler or operating system. For additional information, refer to *4K FORTRAN,* Order No. DEC-08-AFC0-D.

8K FORTRAN

8K FORTRAN consists of a one-pass compiler, the one-pass 8K SABR assembler, a linking loader, and an operating system, as well as a comprehensive library of subprograms. During compilation, the symbolic program is compiled into nonexecutable binary code. During assembly, a relocatable binary program tape is produced. The linking loader is used to convert the relocatable binary code into absolute binary code for execution under control of the operating system. Meaningful error messages are typed on the teleprinter as they are detected. For additional information, refer to *8K FORTRAN,* Order No. DEC-08-KFXB-D.

ALGOL-8

ALGOL (for ALGOrithmic Language) is one of the most widely used international programming languages. The language emphasizes formal, well-defined procedures for solving problems with computers and is the standard language of the scholarly Association for Computing Machinery (ACM).

The ALGOL-8 compiler conforms to SUBSET ALGOL 60 as approved by the International Federation of Information Processing Societies (IFIPS) with additional restrictions. Further information is provided in *ALGOL-8* Programmer's Reference Manual, Order No. DEC-08-KAYA-D.

FOCAL

FOCAL (for FOrmula CALculator) is an online, conversational interpreter designed to be used as a tool by students, engineers, and scientists in solving a wide variety of their problems. The language consists of short imperative English statements and mathematical expressions in standard notation. FOCAL puts the full calculating power and speed of the computer at the user's fingertips without the user having to master the intricacies of machine-language programming; in fact, the user need know nothing at all about computers.

Using FOCAL, a FORTRAN-like program can be entered from the keyboard and immediately executed, with the interpretive features taking care of editing, compiling, and executing the stored program.

Procedures for loading and getting "online" with FOCAL are described later in this chapter, and a thorough description of the FOCAL language is given in Chapter 9. (FOCAL is also described in a separate manual, Order No. DEC-08-AJAD-D.)

BASIC-8

BASIC-8 is a modified version of the very popular algebraic language developed at Dartmouth College. The BASIC-8 language is composed of easy-to-learn English statements and mathematical expressions. It is ideally suited for the classroom as well as the office or laboratory. (BASIC-8 operates with TSS/8, described in Chapter 8.)

Disk Monitor System

The Disk Monitor System is a keyboard-oriented system containing a monitor and a comprehensive software package. The package includes a FORTRAN Compiler, Program Assembly Language (PAL-D), Editor program (Editor), Peripheral Interchange Program (PIP), and Dynamic Debugging Technique (DDT-D) program. These system programs simplify the user's task of editing, assembling, compiling, debugging, loading, saving, calling and running his own programs. The system is modular and open ended, permitting the user to construct the software required in his particular environment and to have full access to his disk for storage and retrieval of his programs.

Chapter 7 is devoted to the Disk Monitor System, and it is further described in *Disk Monitor System,* Order No. DEC-D8-SDAB-D.

TSS/8

TSS/8 (Time-Sharing System for the PDP-8/I and PDP-8 computers) is a general-purpose, stand-alone, time-sharing system. TSS/8

offers each of up to 16 users a comprehensive library of programs which provide facilities for compiling, assembling, editing, loading, saving, calling, debugging, and running user programs online. Any of these library programs can be called into use by typing, in response to Monitor's invitation (the dot), the command R and the assigned name of the program. For example, .R FOCAL brings the FOCAL program into core from the disk and automatically executes FOCAL so that it begins typing out its initial dialogue (see Chapters 8 and 9).

The heart of this time-sharing system is a complex of programs called Monitor. Monitor coordinates the operations of the various units, allocates the time and services of the computer to users, and controls their access to the System. By segregating the central processing operations from the time-consuming interactions with the human users, the computer can in effect work on a number of programs simultaneously. The executions of various programs are interspersed without interfering with one another and without detectable delays in the responses to the individual users. See Chapter 8 or *Time-Sharing System—TSS/8 Monitor,* Order No. DEC-T8-MRFB-D.

Loaders

A loader is a short program or routine which, when in core, enables the computer to accept and store in core other programs. DEC offers the programmer the following assortment of loaders to use, depending on his preference and system configuration.

Read-In Mode (RIM) Loader—used to load into core programs punched on paper tape in RIM format, primarily the Binary Loader. RIM consist of 17 instructions which are toggled into core using the console switches.

Binary (BIN) Loader—used to load into core programs punched on paper tape in binary format, which includes the programmer's binary tapes and most of DEC's system software. BIN is on punched paper tape in RIM format.

HELP Loader—used to load into core the RIM and BIN Loaders. HELP is in two parts: the first part consists of 11 instructions which are toggled into core using the console switches; the second part is the HELP Bootstrap Loader punched on paper tape (containing the RIM and BIN Loaders), which is loaded into core using the low-speed paper tape reader.

TC01 Bootstrap Loader—used to load into core the DECtape Library System programs. The loader is a 20-instruction program which can be toggled into core using the console switches or it may

be on paper tape in RIM format and therefore read into core using the RIM Loader.

The four loaders above are covered in greater detail in the *System User's Guide,* Order No. DEC-08-NGCB-D. RIM and BIN are also described under "Operating Procedures" in this chapter.

Disk System Binary Loader—is a keyboard-oriented loader used to input assembled binary programs into core. This loader is explained in Chapter 7 and in *Disk Monitor System,* Order No. DEC-D8-SDAB-D.

Dynamic Debugging Programs

Dynamic debugging programs are service programs that allow the programmer to run his binary program on the computer. From the Teletype keyboard, the programmer can control program execution, examine registers and change their contents, make alterations to the program, and much more. DDT-8 and ODT-8 are the two dynamic debugging programs included in the system software package.

DDT-8 (Dynamic Debugging Technique)—allows the programmer to do all the things mentioned in the preceding paragraph by communicating with his object program using either the mnemonic coding of the symbolic program or the octal coding of the binary program. DDT-8 is described in more detail later in this chapter and in *DDT-8,* Order No. DEC-08-CDDA-D.

ODT-8 (Octal Debugging Technique)—allows the programmer to do all the things mentioned above by communicating with his object program using the octal representation of his binary program. ODT-8 occupies less core storage than DDT-8 and can be loaded in upper memory *or* lower memory, depending on where the binary program resides. If the programmer's program uses floating-point numbers, the low version of ODT-8 *must* be used when debugging his program (DDT-8 does not interpret floating-point numbers). ODT-8 is described in more detail later in this chapter and in *ODT-8* Order No. DEC-08-COCO-D.

Library of Utility Subroutines

Utility subroutines are short routines for performing such tasks as printout or punchout of core memory content in octal, decimal, or binary form, as specified by the programmer. Other tasks include octal or decimal data transfers and binary-to-decimal, decimal-to-binary, and paper tape conversions. These subroutines can be incorporated in the user's program or run independently.

A complete set of standard diagnostic programs is provided to simplify and expedite system maintenance; these are called MAINDEC programs or routines. MAINDECs permit the programmer to effectively test the operation of the computer for proper core memory functioning and proper execution of instructions. They also enable performance checking of standard and optional peripheral devices. A list of software documentation may be obtained from the DEC Program Library.

Library of Mathematical Subroutines

The system software package also includes a set of mathematical function routines to perform the following operations in both single and double precision: addition, subtraction, multiplication, division, square root, sine, cosine, arctangent, natural logarithm, and exponentiation. These routines are incorporated in the programmer's symbolic program as needed and are executed in response to an indirect JMS instruction to the desired routine. (See *Program Library Math Routines*, Order No. DEC-08-FFAC-D.)

Also included in the software package is a floating-point system to enable the programmer to concentrate on the logic of his computation rather than on decimal points. The system maintains a constant number of significant digits throughout the computation, thereby enhancing the accuracy of the result.

Floating-point notation is particularly useful for computations involving numerous multiplications and divisions where magnitudes are likely to vary widely and where only crude predictions can be made as to the amount of variation involved. The floating-point system allows storage of very large or very small numbers by storing only the significant digits together with the exponent for that number. The system is constructed as a self-contained package which includes its own input, arithmetic, and output routines. It allows the programmer to use floating-point arithmetic without having to construct his own arithmetic subroutines.

When loaded in core, the starting address of the floating-point system is stored in absolute location 0007 and is activated in response to a JMS I 7 instruction. Then, the appropriate floating-point subroutines are called by the subsequent instructions in the program, as shown in the following example.

```
JMS I 7          /indirect jump to floating-point system,
FGET A           /gets the floating-point number A
FADD B           /and adds it to floating-point number B
```

```
FPUT C          /and puts the result in floating-point
FEXT            /location C, and then exits the floating-
                /point system, returning to the main
                /program
```

For more detail, see *Floating-Point System,* Order No. Digital 8-5-S.

Availability of Software

System program tapes and manuals are available from DEC Sales Offices and the central Program Library. In addition, a large variety of programs written by users of PDP-8 family computers are available through DECUS (see Chapter 11).

OPERATING PROCEDURES

Initializing the System

Before using the computer system, it is good practice to initialize all units. To initialize the system, ensure that all switches and controls are as specified below.

1. Main power cord is properly plugged in.
2. Teletype is turned OFF.
3. Low-speed punch is OFF.
4. Low-speed reader is set to FREE.
5. Computer POWER key is ON.
6. PANEL LOCK is unlocked.
7. Console switches are set to
   DF=000 IF=000 SR=0000
   SING STEP and SING INST are not set.
8. High-speed punch is OFF.
9. DECtape REMOTE lamps OFF.

The system is now initialized and ready for your use.

Loaders

READ-IN MODE (RIM) LOADER

When a computer in the PDP-8 family is first received, it is nothing more than a piece of hardware; its core memory is completely demagnetized. The computer "knows" absolutely nothing, not even how to

receive input. However, the programmer knows from Chapter 4 that he can manually load data directly into core using the console switches.

The RIM Loader is the very first program loaded into the computer, and it is loaded by the programmer using the console switches. The RIM Loader instructs the computer to receive and store, in core, data punched on paper tape in RIM coded format (see Chapter 4). (RIM Loader is used to load the BIN Loader described below.)

There are two RIM loader programs: one is used when the input is to be from the low-speed paper tape reader, and the other is used when input is to be from the high-speed paper tape reader. The locations and corresponding instructions for both loaders are listed in Table 6-1.

The procedure for loading (toggling) the RIM Loader into core is illustrated in Figure 6-1.

Table 6-1. 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

After RIM has been loaded, it is good programming practice to verify that all instructions were stored properly. This can be done by performing the steps illustrated in Figure 6-2, which also shows how to correct an incorrectly stored instruction.

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

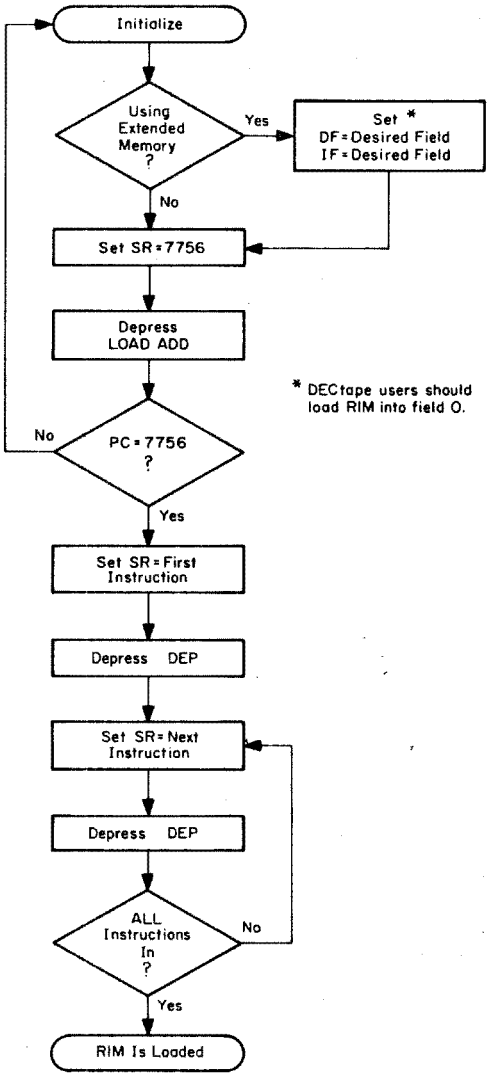


Figure 6-1. Loading the RIM Loader

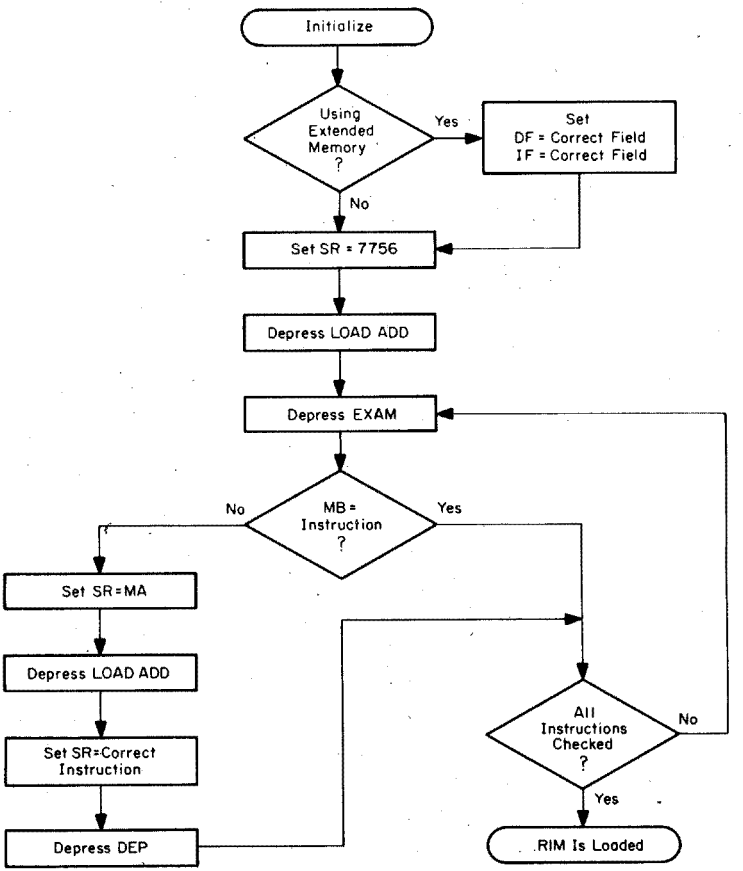


Figure 6-2. Checking the RIM Loader

BINARY (BIN) LOADER

The BIN 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 the programs furnished in the software package (excluding the loaders and certain subroutines) and the programmer's binary tapes.

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. Figure 6-3 illustrates the steps necessary to properly load BIN. And when loading, the input device (low- or high-speed reader) must be that which was selected when loading RIM.

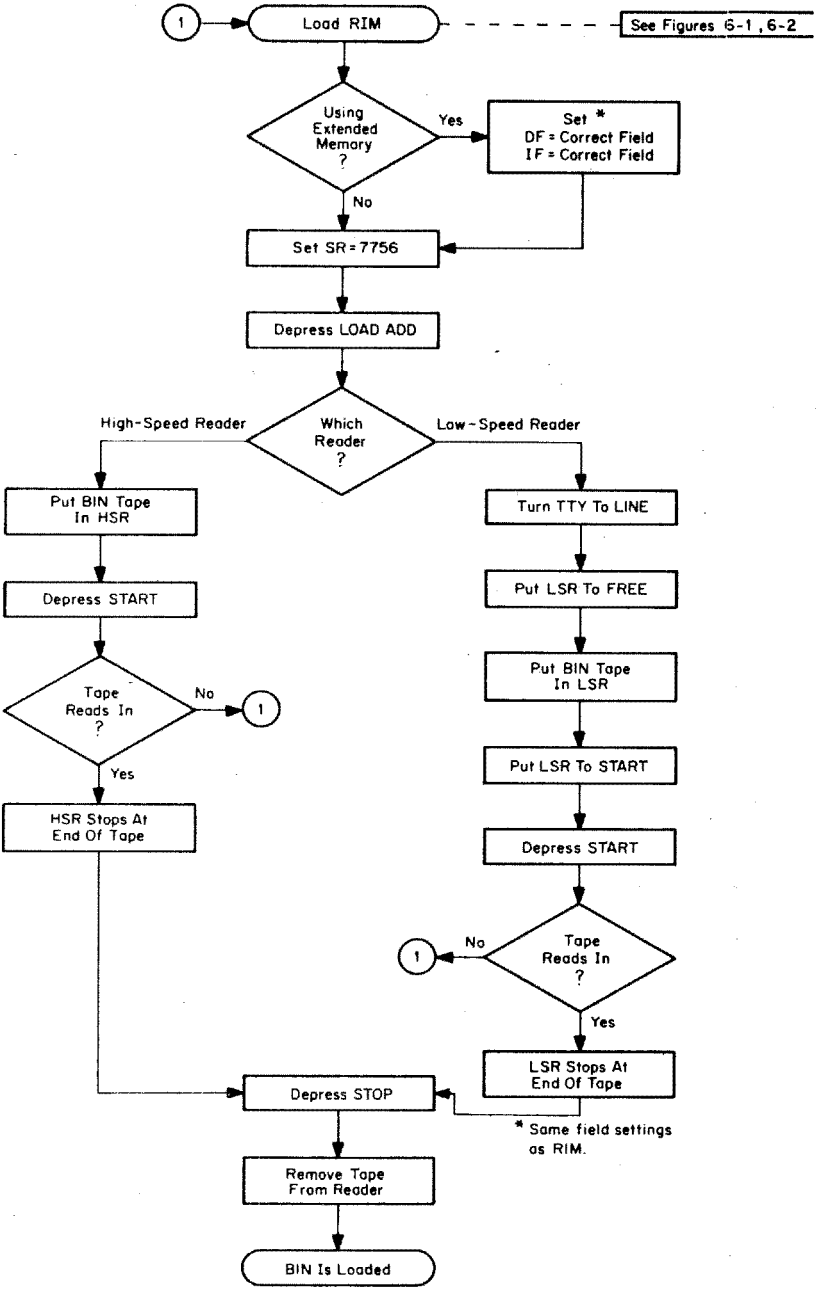


Figure 6-3. Loading the BIN Loader

When stored in core, BIN resides on the last page of core, occupying absolute locations 7625 through 7752 and 7777.

BIN was purposely placed on the last page of core so that it would always be available for use—the programs in DEC's software package do not use the last page of core (excluding the Disk Monitor, discussed in Chapter 7). The programmer must be aware that if he writes a program which uses the last page of core, BIN will be wiped out when that program runs on the computer. When this happens, the programmer must load RIM and then BIN before he can load another binary tape.

Figure 6-4 illustrates the procedure for loading binary tapes into core.

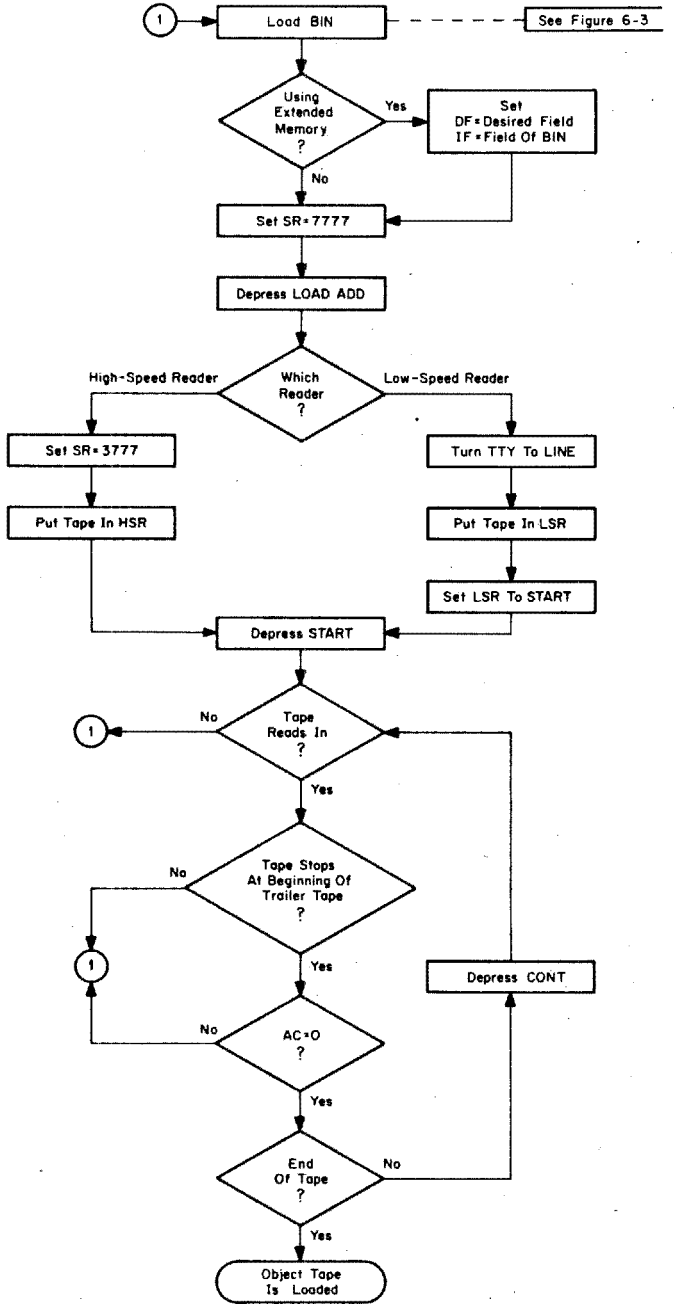


Figure 6-4. Loading A Binary Tape Using BIN

Symbolic Editor

The Symbolic Editor is a service program which allows the programmer to write and prepare symbolic programs and to generate a symbolic program tape of his programs. Editor is very flexible in that the programmer can type his symbolic program online from the Teletype keyboard, thus storing it directly into core memory. Then, using certain Editor commands, the programmer can have his program listed (printed) on the teleprinter for visual inspection.

Editor also allows the programmer to add, correct, and delete any portion of his symbolic program. When the programmer is satisfied that his program is correct and ready to be assembled or compiled, Editor can be commanded to generate a symbolic program tape of the stored program.

The Symbolic Editor program is usually issued on punched paper tape in binary-coded format. Therefore, it is loaded into core memory using the BIN Loader. When in core, Editor is activated for use by setting the switch register (SR) to 0200 (the starting address) and depressing the LOAD ADD (load address) and then START switches. Editor responds with a carriage return/line feed sequence on the Teletype.

Initially, Editor is in command mode, that is, it is ready to accept commands from the programmer; anything typed by the programmer is interpreted as a command to Editor. Editor accepts only legal commands, and if the programmer types something else, Editor ignores the command and types a question mark (?).

When not in command mode, Editor is in text mode, that is, all characters typed from the keyboard or tapes read in on the tape reader are interpreted as text to be put into the text buffer in the manner specified by a preceding Editor command. Figure 6-5 illustrates how the programmer can transfer Editor from one mode to the other.

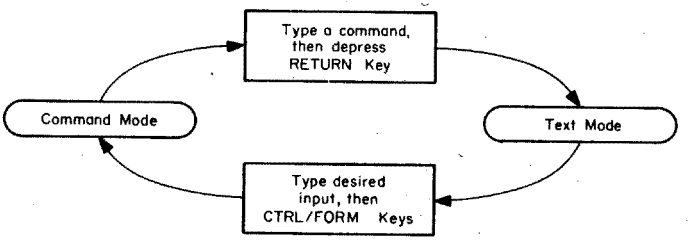


Figure 6-5. Transition Between Editor Modes

Seven of Editor's basic commands are briefly described below.

Command	Meaning
A	Append incoming text from the *keyboard* into the text buffer immediately following the text currently stored in the buffer.
R	Read incoming text from the *tape reader* and append it to the text currently stored in the buffer.
L	List entire text buffer; the programmer can specify one line or a group of lines.
C	Change a line; the programmer precedes the command with the decimal line number or line numbers of the lines to be changed.
I	Insert into text buffer; the programmer specifies the decimal line number in his program where the inserted text is to begin.
D	Delete from text buffer; the programmer specifies the line or group of lines to be deleted.
P	Punch text buffer; the programmer can specify one line, a group of lines, or the entire text buffer.

All commands are executed when the RETURN key is depressed except the P command. To execute the P command, press the RETURN key on the Teletype, turn on the punch, and press the CONT (continue) switch on the computer console.

The above commands are only the seven basic commands. A summary of all commands is provided in Table 6-4 at the end of this section.

WRITING A PROGRAM

Now that you have some idea of what you can do with Editor and what Editor can do for you, we will write and edit a short program, explaining each step in the comments to the right of the printout.

The example program finds the larger of two numbers and halts with the number displayed in the accumulator (AC). The program is written in PAL III, to be assembled using the PAL III Assembler described later in this chapter.

The programmer loads Editor using the BIN loader (see Figure 6-4). Editor is then activated by loading the starting address (0200_8) and depressing the LOAD ADD and START switches. After Editor responds with a carriage return/line feed, the programmer types A and RETURN key. Editor is now in text mode, that is, subsequent characters typed are appended to the text buffer. The programmer now types the symbolic program. (Block indenting is facilitated using the CTRL/TAB key, which Editor has programmed to indent in ten-character increments.)

```
A
                *200
                CLA             /CLEAR AC
                TAD NUMB        /GET B
                CMA
                CMA             /1's COMP B
                IAC             /-B
                TAD NUMA        /ADD -B + A
                SMA.            /IF -B LARGER
                JMP .+4         /JUMP 4 LOCATIONS
                CLA             /CLEAR AC
                TAD NUMB        /GET B
                HLT             /B IS LARGER
                CLA             /CLEAR AC
                TAD NUMA        /GET A
                HLT             /A IS LARGER
NUMB,           0000
NUMA,           0000
                $
```

Visual inspection reveals that we have errors in lines 4, 16, and 17. (Editor maintains a line number count in decimal, with the first line typed being 1 and our last line being 18.) Line 4 can be removed using the D (Delete) command, and lines 16 and 17 can be corrected using the C (Change) command. However, Editor is presently in text mode, and in order to issue another command Editor must be transferred to command mode. This is done when the programmer types CTRL/FORM (depress and hold down the CTRL key while typing the FORM key).

CTRL/FORM (nonprinting)	The programmer types CTRL/FORM; Editor responds with CR/LF and rings the teleprinter bell, indicating that it is in command mode.
4D	The programmer types 4D and the RETURN key; Editor responds with a CR/LF and the line is deleted.
15, 16C	The programmer types 15, 16C and the RETURN key, informing Editor that lines 15 and 16 (formerly 16 and 17) are to be changed.
	Editor responds with a CR/LF, transfers to text mode, and waits for the programmer to change the lines.
NUMA, 1111 NUMB, 0011	The programmer types NUMA, 1111 and NUMB, 0011.

The symbolic program should now be correct. However, it is good programming practice to check the program after editing; this can be done using the L (List) command, but since only original lines 4, 16, and 17 were changed it is not necessary to have the whole program listed. The programmer can command Editor to list lines 4 through 17.

CTRL/FORM (nonprinting)	The programmer types CTRL/FORM to return Editor to command mode; Editor responds with CR/LF and rings the bell, and waits for the next command.
4, 17L	The programmer types 4, 17L and the RETURN key; Editor types lines 4 through 17.

```
        CMA             /1's COMP B
        IAC             /-B
        TAD NUMA        /ADD -B + A
        SMA             /IF -B LARGER
        JMP .+4         /JUMP 4 LOCATIONS
        CLA             /CLEAR AC
        TAD NUMB        /GET B
        HLT             /B IS LARGER
        CLA             /CLEAR AC
        TAD NUMA        /GET A
        HLT             /A IS LARGER
NUMA,   1111
NUMB,   0011
        $
```

The changes were accepted properly. The symbolic program is correct and ready to be punched on paper tape.

GENERATING A PROGRAM TAPE

Before issuing the P (Punch) command, Editor must be in command mode. Figure 6-6 illustrates the procedures required to generate a symbolic program tape using Editor.

CTRL/FORM (nonprinting) ?	The programmer types CTRL/FORM; Editor responds with a question mark, indicating that Editor *was* in command mode.
P	The programmer commands Editor to punch the entire text buffer by typing P and the RETURN key.

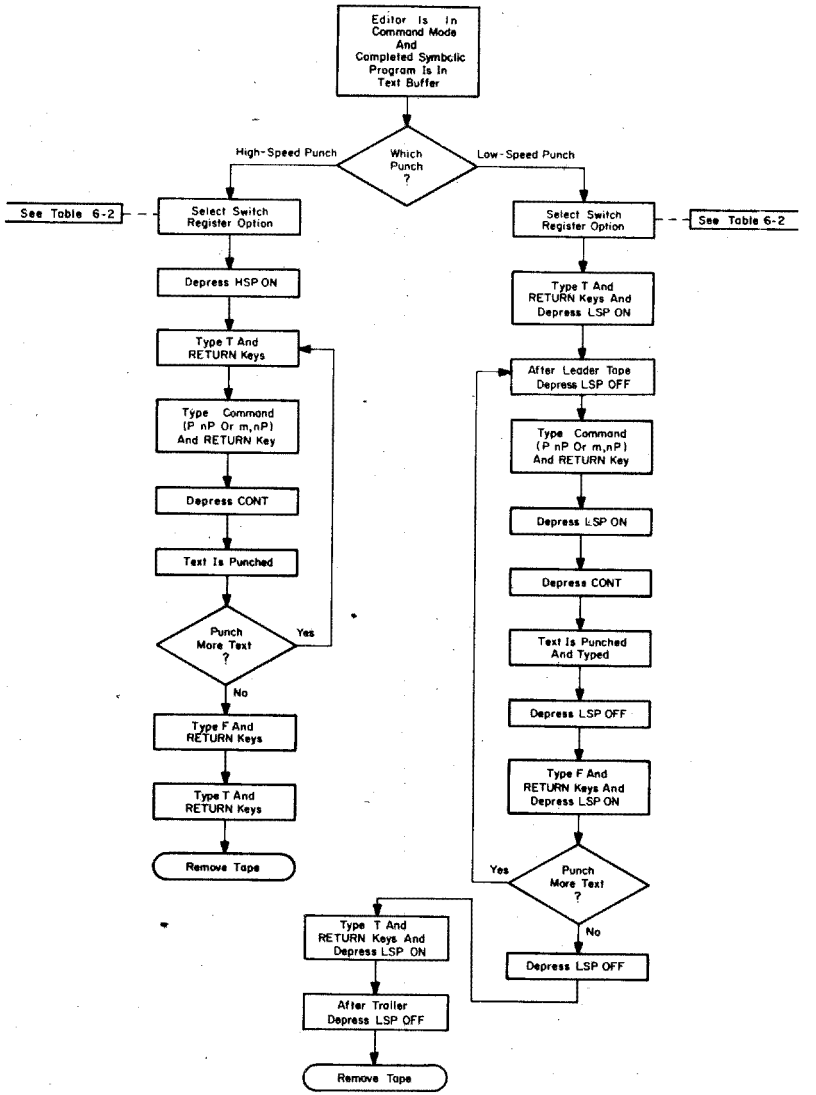


Figure 6-6. Generating a Symbolic Tape Using Editor

When Editor recognizes a P command it waits for the programmer to specify the low- or high-speed punch. If the programmer wants the program punched and typed, he sets SR bit 10 to 0 and the program will be punched on the low-speed punch and simultaneously typed on the teleprinter. If the programmer wants only a program tape and if he

has a high-speed punch available, he sets SR bit 10 to 1 and the program will be punched on the high-speed punch. For the purposes of this discussion, a printed program listing is desired, so the low-speed punch is specified. The programmer turns on the low-speed punch and depresses the CONT switch on the computer console, and Editor begins punching and typing the contents of the entire text buffer.

An image of the stored symbolic program has been punched and typed by Editor.

If the programmer stops the computer, e.g., purposely or accidentally turning the computer off, he may restart Editor at location 0200 or 0177 without disturbing the text in the buffer. Editor can also be restarted at location 0176; however, all text currently in the buffer is wiped out. Therefore, the programmer can restart at location 0176 to re-initialize for a new program.

SEARCH FEATURE

A very convenient feature available with Editor is the search feature, which allows the programmer to search a line of text for a specified character. When the programmer types a line number followed by S, Editor waits for the user to type in the character for which it is to search. The search character is not echoed (printed on the teleprinter). When Editor locates and types the search character typing stops, and Editor waits for the programmer to either type new text and terminate the line with a RETURN key or to use one of the following special keys.

1.	←	to delete the entire line to the left,
2.	RETURN	to delete the entire line to the right,
3.	RUBOUT	to delete from right to left one character for each RUBOUT typed (a \ is echoed for each RUBOUT typed),
4.	LINE FEED	to insert a carriage return/line feed (CR/LF) thus dividing the line into two,
5.	CTRL/FORM	to search for the next occurrence of the search character, and/or
6.	CTRL/BELL	to change the search character to the next character typed by the programmer.

INPUT/OUTPUT CONTROL

Switch register options are used with input and output commands to control the reading and punching of paper tape. The options available to the programmer are shown in Table 6-2. These options are used in conjunction with the "Select Switch Register Option" operation in Figure 6-6.

Table 6-2. Input/Output Control

SR Bit	Position	Function
0	0	Input text as is
	1	Convert all occurrences of 2 or more spaces to a tab
1	0	Output each tab as 8 spaces
	1	Tab is punched as tab/rubout
2	0	Output as specified
	1	Suppress output*
10	0	Low-speed punch and Teleprinter
	1	High-speed punch
11	0	Low-speed reader
	1	High-speed reader

*Bit 2 allows the user to interrupt any output command and return immediately to command mode; when desired, merely set bit 2 to 1.

ERROR DETECTION

Editor checks all commands for nonexistent information and incorrect formatting. When an error is detected, Editor types a question mark (?), and ignores the command. However, if an argument is provided for a command that doesn't require one, the argument is ignored and the command is executed properly.

Editor does not recognize extraneous and illegal control characters; therefore, a tape containing these characters can be cleaned up or corrected by merely reading the tape into Editor and punching out a new tape.

SUMMARY OF SPECIAL KEYS AND COMMANDS

Using special keyboard keys and commands, the programmer controls Editor's operation. Certain keys have special meaning to Editor, of which some can be used in either command or text mode. The mode of operation determines the function of each key. The special keys and their function are shown in Table 6-3.

Table 6-3. Special Keys

Key	Command Mode	Text Mode
RETURN	Execute preceding command	Enter line in text buffer
←	Cancel preceding command (Editor responds with a ? followed by a carriage return and line feed)	Cancel line to the left margin
RUBOUT	same as ←	Delete to the left one character for each depression; a \ (backslash) is echoed (not used in Read (R) command)
CTRL/FORM	Respond with question mark and remain in command mode	Return to command mode and ring teleprinter bell
• (period)	Value equal to decimal value of current line (used alone or with + or — and a number, e.g., .+8)	Legal text character
/	Value equal to number of last line in buffer; used as an argument	Legal text character
LINE FEED	List next line	Used in Search (S) command to insert CR/LF into line
ALTMODE	List next line	
>	List next line	
<	List previous line	
=	Used with . or / to obtain their value	
:	Same as = (gives value of legitimate argument)	
CTRL/TAB		Produces a tab which on output is interpreted as 10 spaces or a tab/rubout, depending on SR option

Editor commands are given when in command mode. There are three basic types of commands: Input, Editing, and Output. Table 6-4 contains a summary of Editor commands and their function.

Table 6-4. Summary of Commands

Type	Command	Function
Input	A	Append incoming text from keyboard into text buffer
	R	Append incoming text from tape reader into text buffer
Editing	L	List entire text buffer
	nL	List line n
	m,nL	List lines m through n inclusively
	nC	Change line n
	m,nC	Change lines m through n inclusively
	I	Insert before first line
	nI	Insert before line n
	K	Delete entire text buffer
	nD	Delete line n
	m,nD	Delete lines m through n inclusively
	m,n$kM	Move lines m through n to before line k
	G	Print next tagged line (if none, Editor types ?)
	nG	Print next tagged line after line n (if none, ?)
	S	Search buffer for character specified after RETURN key and allow modification (search character is not echoed on printer)
	nS	Search line n, as above
	m,nS	Search lines m through n inclusively, as above
Output	P	Punch entire text buffer
	nP	Punch line n
	m,nP	Punch lines m through n inclusively
	T	Punch about 6 inches of leader/trailer tape
	F	Punch a FORM FEED onto tape
	N	Do P, F, K, and R commands

m and n are decimal numbers, and m is smaller than n; K is a decimal number.

The P and N commands halt the Editor to allow the programmer to select I/O control; press CONT to execute these commands.

Commands are executed when the RETURN key is depressed, excluding the P and N commands.

Symbolic Assemblers

A symbolic assembler is a service program that translates symbolic programs into binary-coded programs which can be loaded and run on the computer. In other words, the programmer writes the symbolic program using symbols which are meaningful to him, and then an assembler is used to translate the symbols into binary code, which is meaningful to the computer. The computer knows only yes or no, plus

or minus, voltage or no voltage, magnetized or demagnetized, i.e., one of two conditions (states) which we simplify as 1 or 0. Therefore, the assembler translates the programmer's symbols into 1's and 0's which are meaningful to the computer.

Because assemblers are vital to the efficient operation of computers in the PDP-8 family, DEC presently offers four (PAL III, MACRO-8, PAL-D, and 8K SABR), and this list is destined to grow with time as other versions are needed to assemble other programming languages. This section includes general descriptions of PAL III and MACRO-8. PAL-D incorporates most of the features of both PAL III and MACRO-8 and is used only in the Disk Monitor System. So, if you learn PAL III and MACRO-8, you have only to learn the few exceptions of PAL-D and then you know all three.

First PAL III is discussed and then MACRO-8. MACRO-8 is compatible with PAL III except for some additional features, therefore, in the section on MACRO-8 emphasis is on the additional features and exceptions.

PAL III SYMBOLIC ASSEMBLER

The PAL III Symbolic Assembler (PAL stands for Program Assembly Language) is an indispensable service program used to translate symbolic programs, which are written in the PAL III language, into binary-coded programs (binary programs). Having progressed to this section of the Handbook, *you are by now familiar with the PAL III programming language; because the symbolic language used in the preceding chapters is PAL III.* In this section, the PAL III Assembler is used to assemble the example program written using Editor (see "Writing a Program," above).

PAL III is a two-pass assembler with an optional third pass, i.e., the symbolic program tape must be passed through the assembler two times to produce the binary-coded tape (binary tape), and the optional third pass produces a complete octal/symbolic program listing which can be typed and/or punched if desired. A brief explanation of the three passes is given below.

Pass 1. The assembler reads the symbolic program tape and defines all symbols used and places these user symbols in a symbol table for use during Pass 2. The assembler checks for undefined symbols and certain other errors and types an error message on the teleprinter when an error is detected.

Pass 2. The assembler rereads the symbolic program tape and generates the binary tape using the symbols defined during Pass 1. When the low-speed punch is used, meaningless characters will be typed on the

teleprinter, and these should be ignored by the programmer. The assembler checks illegal referencing during this pass and types an error message on the teleprinter when any is detected.

Pass 3. The assembler reads the symbolic program tape and types and/or punches the octal/symbolic program assembly listing. This listing thoroughly documents the assembled program and is useful when debugging and modifying the program.

The meaningless characters, error messages, and octal/symbolic program listing will be shown later in this section.

PAL III accepts symbolic program tapes from either the low-speed or high-speed reader and produces the binary tapes on either the low-speed or high-speed punch.

During assembly, the programmer communicates with PAL III via the switches on the computer console. Switch options are used to specify which pass the assembler is to perform and which reader and punch the assembler should accept input from and punch out on.

ASSEMBLING A SYMBOLIC PROGRAM. Earlier in this chapter, the programmer wrote a PAL III symbolic program and generated the symbolic program tape using Editor. That symbolic program can now be assembled to produce a binary program using PAL III. A listing of the symbolic program follows.

```
        *200
        CLA             /CLEAR AC
        TAD NUMB        /GET B
        CMA             /1'S COMP B
        IAC             /-B
        TAD NUMA        /ADD -B + A
        SMA             /IF -B LARGER
        JMP .+4         /JUMP 4 LOCATIONS
        CLA             /CLEAR AC
        TAD NUMB        /GET B
        HLT             /B IS LARGER
        CLA             /CLEAR AC
        TAD NUMA        /GET A
        HLT             /A IS LARGER
NUMA,   1111
NUMB,   0011
        $
```

First, PAL III must be loaded into core memory, and since PAL III is on punched paper tape in binary-coded format, it is loaded into core memory using the BIN Loader (see Figure 6-4 for loading procedures).

With PAL III in core, we are ready to assemble the symbolic program. Figures 6-7 and 6-8 illustrate the procedures for assembling with PAL III using the low-speed reader/punch and high-speed reader/punch, respectively. In these flowcharts, the switch register options are set for the appropriate reader/punch.

The low-speed reader and punch (LSR and LSP) are used in the following assembly (see Figure 6-7).

Initializing and Starting

Load PAL III into core memory using BIN.
Set SR=0200 and depress LOAD ADD.
Turn TTY to LINE and put symbolic program tape in LSR.

Entering Pass 1

Set SR=2200 and set LSR to START.
Depress LSP to ON and depress START.

```
NUMA    0215
NUMB    0216
```

Error messages would be typed now.
Symbol table concludes Pass 1.

Entering Pass 2

Put symbolic program tape in LSR.
Set SR=4200 and set LSR to START.
Depress LSP to ON and depress CONT.

```
BB:
  8 8
=*
  <:
<)
```

Disregard meaningless characters while object tape is being punched.
Error messages would be typed now.

Entering Pass 3

Put symbolic program tape in LSR.
Set SR=6200 and set LSR to START.
Depress LSP to ON and depress CONT.
The octal/symbolic program listing is being typed and punched.

```
                        *200
0200    7200            CLA         /CLEAR AC
0201    1216            TAD NUMB    /GET B
0202    7040            CMA         /1'S COMP B
0203    7001            IAC         /-B
0204    1215            TAD NUMA    /ADD -B + A
0205    7500            SMA         /IF -B LARGER
0206    5212            JMP .+4     /JUMP 4 LOCATIONS
0207    7200            CLA         /CLEAR AC
0210    1216            TAD NUMB    /GET B
0211    7402            HLT         /B IS LARGER
0212    7200            CLA         /CLEAR AC
0213    1215            TAD NUMA    /GET A
0214    7402            HLT         /A IS LARGER
0215    1111    NUMA,   1111
0216    0011    NUMB,   0011
NUMA    0215
NUMB    0216
```

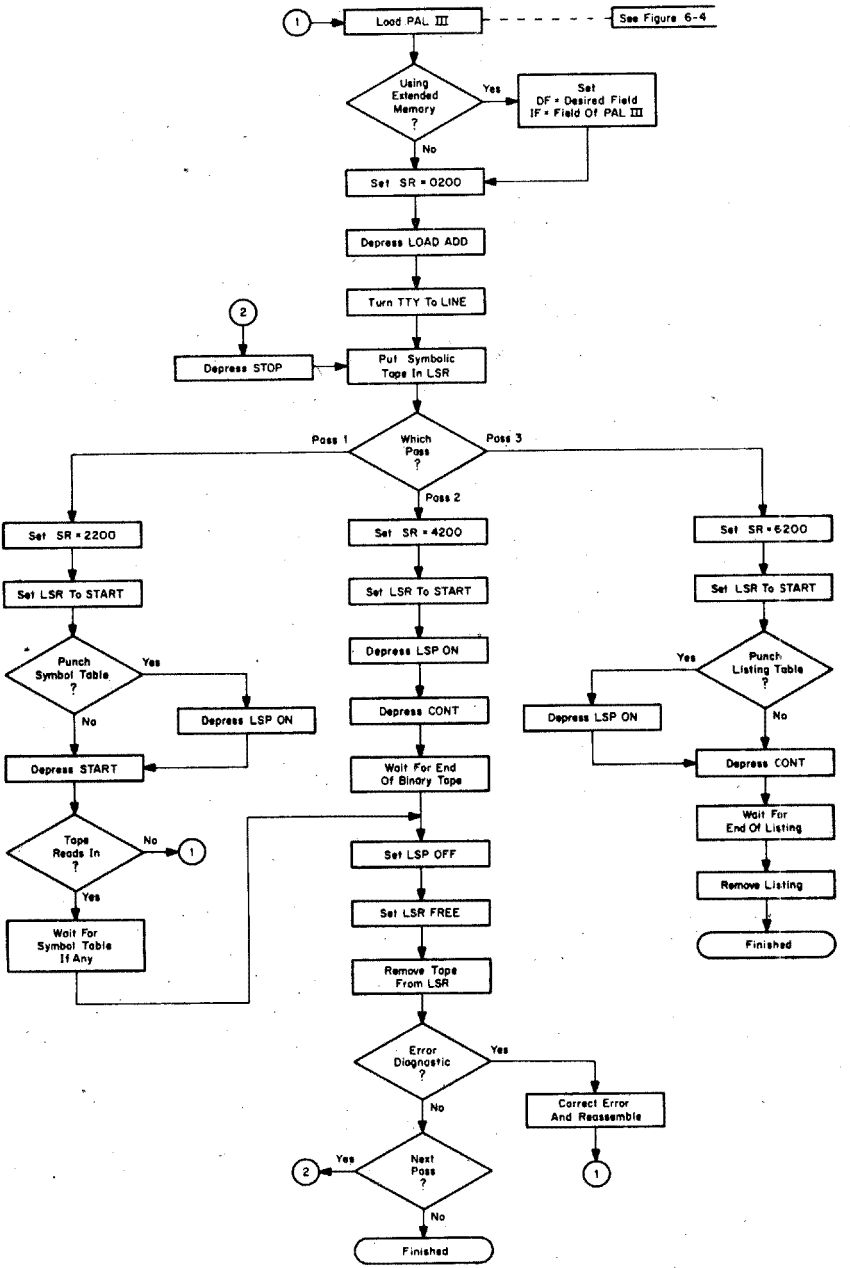


Figure 6-7. Assembling with PAL III Using Low-Speed Reader/Punch

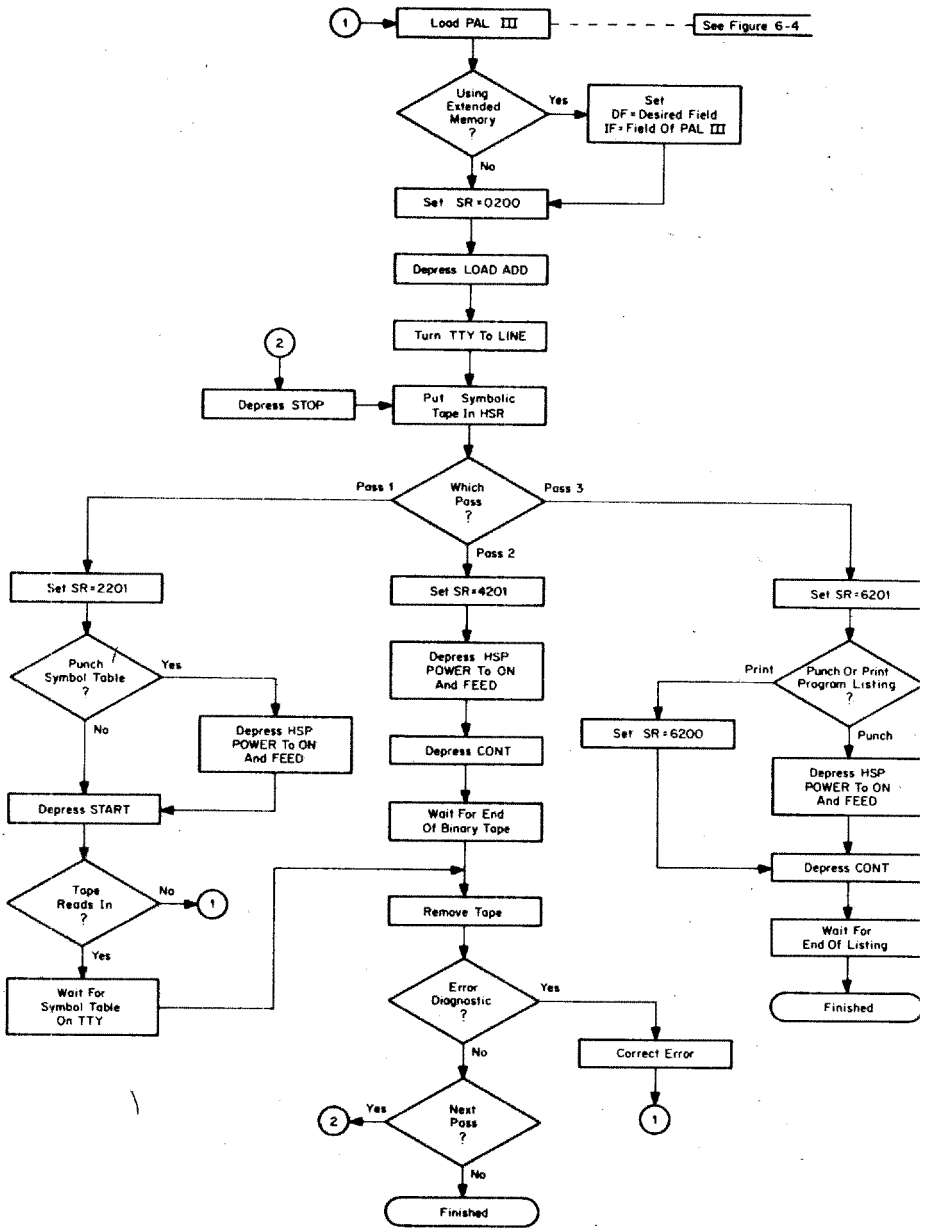


Figure 6-8. Assembling with PAL III Using High-Speed Reader/Punch

The tape produced during Pass 2 is the binary tape, which is loaded into core memory using the BIN Loader. The symbol table tape produced during Pass 1, the binary tape produced during Pass 2, and the octal/symbolic program listing produced during Pass 3 are used when debugging the program.

PSEUDO-OPERATORS. When writing a PAL III symbolic program, the programmer can use pseudo-operators (pseudo-ops) to direct the assembler to perform certain task or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the binary program, others direct the assembler on how to proceed with the assembly. Pseudo-ops are maintained in the assembler's permanent table,[1] which can be altered using certain psuedo-ops.

When using more than one memory bank, the pseudo-op FIELD instructs the assembler to output a field setting:

FIELD n — where n is an integer, a previously defined symbol, or a symbolic expression within the range 0 through 7, inclusive.

Integers used in a symbolic program are usually taken as octal numbers. However, if the programmer wishes to have certain numbers treated as decimal, he may use the pseudo-op DECIMAL, and then return to the original radix with the pseudo-op OCTAL.

DECIMAL — all integers in subsequent coding are taken as decimal until the occurrence of the pseudo-op OCTAL,

OCTAL — which resets the radix to its original base.

In an indirect address instruction, the special symbol "I" between the operation code and the operand, or address field, is another pseudo-op. (Indirect addressing was covered in Chapter 2.)

When a symbolic program is very long, it is often desirable to punch the symbolic program on two or more physical lengths of tape. The last instruction of each section must be the pseudo-op PAUSE, and the last instruction of the program must be a dollar sign ($).

PAUSE — stops the assembler, but the current pass is not terminated. When the programmer is ready to assemble the next section, he has only to depress CONT on the computer console. (PAUSE is normally used only at the physical end of tape.)

[1]The permanent symbol table contains operation codes (MRI's, IOT's, microinstructions, and pseudo-ops) and their octal equivalents; it is a permanent part of the assembler. An external symbol table contains user-assigned symbols and their corresponding absolute addresses; it is created during assembly and punched on the binary tape during Pass 2.

The last three pseudo-ops are used to alter the permanent symbol table. They are:

EXPUNGE	erases the entire permanent symbol table excluding the pseudo-ops.
FIXMRI	meaning FIX Memory Reference Instructions. The pseudo-op FIXMRI must be followed by one space, the symbol for the MRI to be defined, an equal sign, and the octal value of the symbol to the immediate left of the equal sign.
FIXTAB	meaning FIX the current permanent symbol TABle. All symbols that have been defined before the occurrence of this pseudo-op are made part of the permanent symbol table.

Therefore, with these three pseudo-ops the programmer can alter the permanent symbol table to contain only those symbols he needs to assemble his symbolic program, which in turn provides more core for the external symbol table.

OUTPUT CONTROL. Output is controlled by the setting of switch register bit 11. When the assembler first enters a pass, it "looks" at the state of SR bit 11 and outputs as specified. The SR bit 11 options are:

Bit 11=0	Output on teleprinter and low-speed punch.
Bit 11=1	Output on high-speed punch.

ERROR MESSAGES. The assembler is constantly checking for assembly errors, and when any is detected, an error message is printed on the teleprinter. Error messages are printed in the following format.

xx yyyyyy AT nnnn

where xx is the error message (see below), yyyyyy is the symbol or octal value of the symbol of the error occurring AT location nnnn.

Assembly errors are checked for during Pass 1 and Pass 2 only. The error codes are listed below.

Pass 1:	IC	Illegal Character
	RD	Redefinition
	DT	Duplicate Tag
	ST	Symbol Table Full
	UA	Undefined Address
Pass 2:	IR	Illegal Reference

For a thorough description of PAL III, see *PAL III Symbolic Assembler,* Order No. DEC-08-ASAC-D.

MACRO-8 SYMBOLIC ASSEMBLER

The MACRO-8 Symbolic Assembler is a service program used to translate symbolic programs that are written in the MACRO-8 symbolic language into binary programs. The MACRO-8 language can be generally considered as PAL III with the following additional features:

1. User-Defined Macros—Groups of computer instructions required for the solution of a specific problem can be defined by the user as a macro instruction (explained later).
2. Double Precision Integers—Positive or negative double precision integers are allotted two consecutive core locations.
3. Floating-Point Constants—The format and rules for defining these constants are compatible with the format used by the Floating-Point System.
4. Operators—Symbols and integers may be combined with a number of operators.
5. Literals—Symbolic or integer literals (constants) are automatically assigned.
6. Text Facility—There are text facilities for single characters and blocks of text.
7. Link Generation—Links are automatically generated for off-page references.

With these additional features, it is clear that the MACRO-8 Assembler requires more core memory than the PAL III Assembler. Therefore, programs originally coded to be assembled by PAL III might have too many user symbols to be assembled by MACRO-8 (it was necessary to decrease the size of the user's symbol table to incorporate the additional features). However, the programmer can, using the appropriate pseudo-op, increase or decrease the size of the permanent symbol table if desired.

MACRO-8 is a two-pass assembler with an optional third pass which produces a octal/symbolic program assembly listing. There are two versions of MACRO-8.

The *Low Version* uses the low-speed reader for all input and the teleprinter and low-speed punch for all output.

The *High Version* uses the high-speed reader for all input, the high-speed punch for binary output (and, if desired, the program listing tape), and the teleprinter and low-speed punch for output of error diagnostics, symbol table, and the third pass program listing.

The three passes of MACRO-8 are identical to those of PAL III, in fact, the example program assembled by PAL III in the preceding section could be assembled by MACRO-8, and the printed output would be the same for both assemblers.

MACROS. When writing a program, it often happens that certain coding sequences are used several times with just the arguments changed. If so, it is convenient if the entire sequence can be generated by a single statement. To do this, the coding sequence is defined by dummy arguments as a macro. A single statement referring to the macro by name, along with a list of real arguments, will generate the correct sequence in line with the rest of the coding.

The macro name must be defined before it is used. The macro is defined by means of the pseudo-operator DEFINE followed by the macro's name and a list of dummy arguments. For example,

```
DEFINE MOVE DUMMY1 DUMMY2
< CLA
TAD DUMMY1
DCA DUMMY2
TAD DUMMY2>
```

The actual choice of symbols used as dummy arguments is arbitrary; however, they may not be defined or referenced prior to the macro definition. The actual definition of the macro must be enclosed in angle brackets (< >).

The above definition of the macro MOVE could have been coded as follows:

```
DEFINE MOVE ARG1 ARG2
<CLA; TAD ARG1; DCA ARG2; TAD ARG2>
```

When a macro name is processed by the assembler, the real arguments will replace the dummy arguments. For example, assuming that the macro MOVE has been defined as above,

```
*400
A, 0                0400    0000
B, -6.              0401    7772
MOVE A, B           0402    7200
$                   0403    1200
                    0404    3201
                    0405    1201
```

A macro need not have any arguments. A sequence of coding to rotate the contents of the accumulator and link six places to the left might be stated as follows.

```
DEFINE ROTL6
<RTL; RTL; RTL>
```

A macro is referenced by giving the macro name, a space, and then the list of real arguments, separated by commas. There must be at least as many arguments in the macro reference as in the corresponding macro definition. For a detailed list of other macro restrictions, see *MACRO-8 Assembler,* Order No. DEC-08-CMAA-D.

LITERALS. Since the symbolic expressions which appear in the address part of an instruction usually refer to the address of locations containing the quantities being operated upon, the programmer must explicitly reserve the locations holding his constants. The MACRO-8 programming language provides a means for using a constant directly. Suppose, for example, that the programmer has an index which is incremented by two. One way of coding this operation would be as follows.

```
        . . .
        CLA
        TAD INDEX
        TAD C2
        DCA INDEX
        . . .
C2,     2
```

Using a literal, this would become

```
        . . .
        CLA
        TAD INDEX
        TAD (2)
        DCA INDEX
        . . .
```

The left parenthesis is a signal to the assembler that the expression following is to be evaluated and assigned a location in the constants table of the current page. This is the same table in which the indirect address linkages are stored. In the above example, the quantity 2 is stored in a location in a list beginning at the top of the memory page (page address 177), and the instruction in which it appears is encoded with an address referring to that location. A literal is assigned to storage the first time it is encountered, and subsequent references will be to the same location.

If the programmer wishes to assign literals to page 0 rather than the current page, he may use square brackets, [and], in place of the parentheses. However, in both cases, the right or closing member may be omitted, as in the example below.

Literals may be nested. For example:

```
*200
TAD (TAD (30
```

will generate

```
0200    1376
. . .   . . .
0376    1377
0377    0030
```

This type of nesting may be carried to as many levels as desired.

PSEUDO-OPS. MACRO-8 has available a number of useful pseudo-ops, which are listed and briefly explained below.

PAGE	When used without an argument, the current location counter is reset to the first location on the next succeeding page. With an argument (PAGE n), the current location counter is reset to the first location on the specified page, page n.
DECIMAL	When this pseudo-op occurs, all integers encountered in subsequent coding will be taken as decimal until the occurrence of OCTAL,
OCTAL	which will reset the radix to its original base.
PAUSE	When several tapes are to be assembled together, each except the last (which ends with $) should have as its last symbol the pseudo-op PAUSE. This causes the assembler to stop processing and halt the computer. After placing a new tape in the tape reader, assembly can be continued by depressing CONT on the computer console.
EXPUNGE	When used, the entire permanent symbol table (excluding pseudo-ops) is erased. This pseudo-op is used when the programmer wishes to provide more core for the user program symbols. Then, with FIXTAB,
FIXTAB	the programmer can build a customized permanent symbol table, containing only those permanent symbols required for his user program.
DEFINE	Used to define macros. This pseudo-op was covered under "Macros," above.

The preceding pseudo-ops and a few others are described in detail in the *MACRO-8 Assembler* manual.

OFF-PAGE REFERENCING. During assembly, the page bits of the address field are compared with the page bits of the current location counter (see "Page Addressing" in Chapter 2). If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an off-page reference is being attempted (see "Indirect Addressing" in Chapter 2). If the reference is to an address not on the page where the instruction will be located, the assembler will set the indirect bit (bit 3) and an indirect address linkage will be automatically generated on the current memory page.

Although the assembler will recognize and automatically generate an indirect address linkage when necessary, the programmer may still indicate an explicit indirect address using the special symbol "I" between the operation code and the address field.

As can be seen by comparing Figures 6-9 and 6-10 to Figures 6-7 and 6-8, the assembly procedures for MACRO-8 and PAL III are similar but certainly not identical.

SWITCH REGISTER OPTIONS. As the assembler begins to enter each pass, it "reads" or "looks at" or "senses" (take your choice) certain bits of the switch register which tell it how to proceed. The optional settings of the switch register are shown in the table below.

Table 6-5. Switch Register Options

SR Bit	Position	Meaning
0-11	0	Enter next pass.
0	1	Erase symbol table, excluding permanent symbol, and enter Pass 1 (the programmer then depresses STOP and CONT).
1	1	Enter Pass 2 to generate another binary tape.
2	1	Enter Pass 1 without erasing defined symbols.
3	1	Enter Pass 3.
10	1	Delete double precision integer and double precision floating-point processors (this increases the symbol table size by 100_8 symbols).
11	1	Delete macro and number processors (this increases the symbol table size by 175_8 symbols).

Switch register bits 10 and 11 are sensed by MACRO-8 when it enters Pass 1, and remembers these settings throughout the assembly. If these bits are set to 1, MACRO-8 would have to be reloaded to handle subsequent programs that use macros, double precision integers, or floating-point numbers.

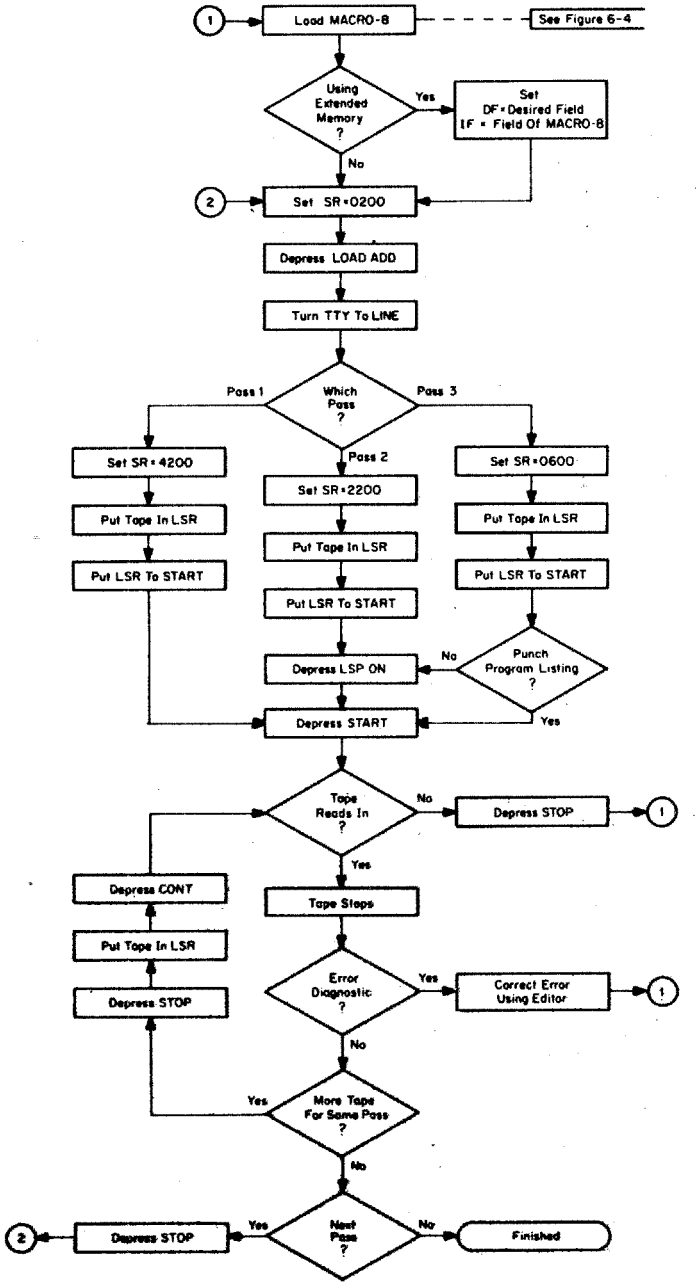


Figure 6-9. Assembling a MACRO-8 Symbolic Program Using the Low-Speed Reader/Punch

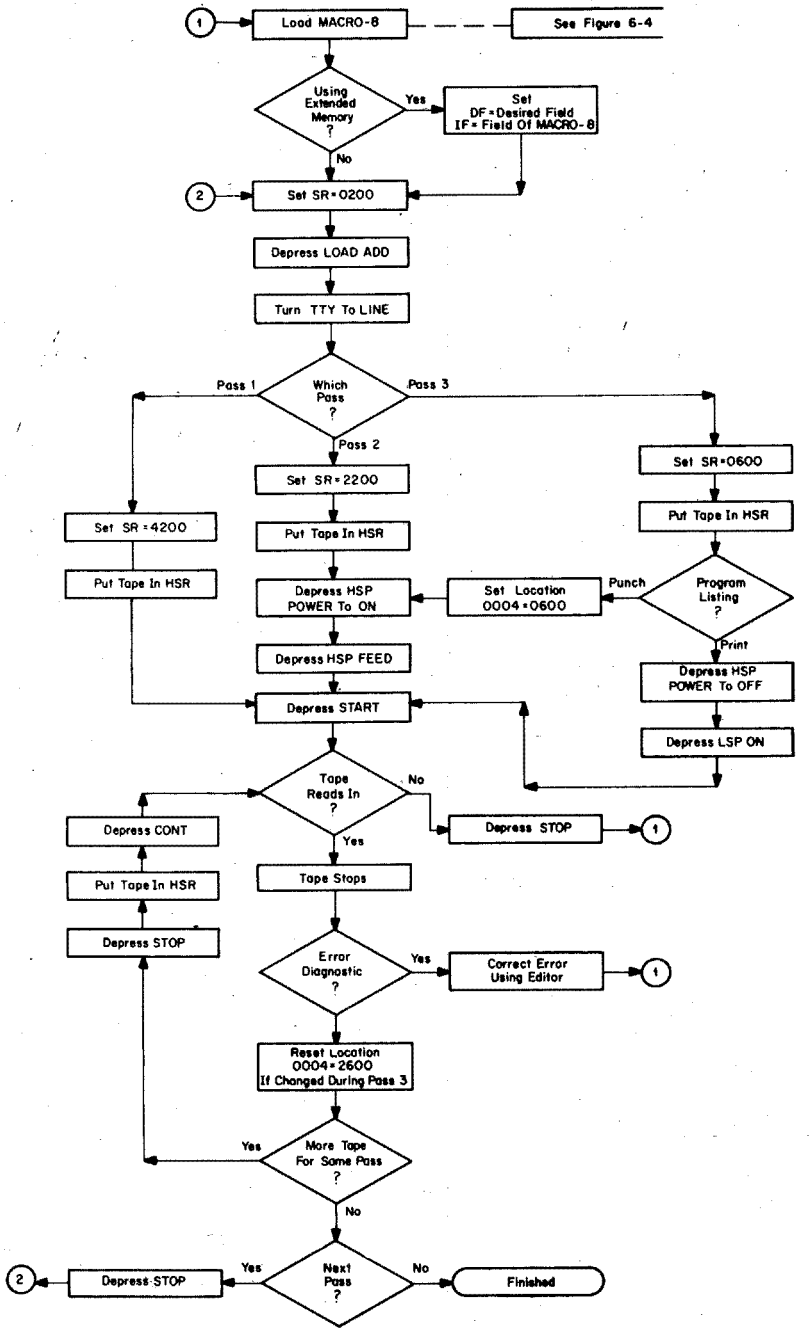


Figure 6-10. Assembling a MACRO-8 Symbolic Program Using the High-Speed Reader/Punch

When assembling using the high-speed reader, the Pass 3 program listing is normally output on the teleprinter and low-speed punch. However, when assembling a long program, the programmer may wish to have the third pass listing output on the high-speed punch. This can be done by changing the contents of location 0004 from 2600 to 0600 (see Figure 6-10). It is advised that this change *not* be made until Pass 3, so that Pass 1 and 2 diagnostics will be printed.

ERROR MESSAGES. The assembler is constantly checking for assembly errors, and when one is detected, an error message is printed on the teleprinter. Error messages are printed in the following format.

xx yyyyyy

where xx is a two-letter code which specifies one of the types of errors listed below, and yyyyyy is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic tag (if there was one) on the current page.

Following is a descriptive list of the error messages, some are printed out during Pass 1, others during Pass 2, and some of which are printed out during both passes.

Error Code	Meaning
BE	MACRO-8 internal tables have overlapped
IC	Illegal character
ID	Illegal redefinition of a symbol
IE	Illegal equal sign
II	Illegal indirect address
IM	Illegal format in a macro definition
LG	Link generated to off-page address*
MP	Missing parameter in macro call
PE	Current, nonzero page exceeded
SE	Symbol table exceeded
US	Undefined symbol
ZE	Page zero exceeded

*This is to inform the user of off-page references which may not be an error. This diagnostic can be suppressed to speed up Pass 2 assembly by setting location 1234 to 7200 prior to entering Pass 1.

For a thorough description of MACRO-8, see *MACRO-8 Assembler,* Order No. DEC-08-CMAA-D.

CONCLUSION

The programmer decides which assembler is needed to translate his symbolic program into a binary program. The decision depends on such things as the amount of core memory available, the size of the symbolic program and the number of user symbols it contains, and whether the symbolic program contains macros, literals, etc.

Dynamic Debugging Programs

Included in the System Software package are two dynamic debugging programs: DDT-8 (Dynamic Debugging Technique) and ODT-8 (Octal Debugging Technique). 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, make alterations to the program, and much much more.

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, not logical errors. Syntax errors include undefined tags, misspelled tags and operation codes, and incorrect format (e.g., omission of a required operand). Logical errors are detected only when the program is running on the computer.

DEBUGGING WITHOUT DDT-8 OR ODT-8

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) by 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; 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, of course, while sitting at his desk using the program assembly listing, mentally execute his program. This method is frequently used with very short programs, very short only—human memory can not retain every step and instruction in even a fairly short program; it can not match 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-8 and ODT-8 are such debugging programs.

DEBUGGING WITH DDT-8

Using DDT-8, the programmer can run his program on the computer, control its execution, and make corrections to the program by typing commands to DDT-8 and altering his program from the Teletype keyboard. 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-8, 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 quickly 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 new program.
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-8 program, which is loaded into core memory using BIN (see Figure 6-4).

To begin the debugging run, first ascertain that BIN is in memory, then load the programmer's binary program and the binary DDT-8 program tape (see Figure 6-4 for loading procedures). Figure 6-11 illustrates the procedures for loading and executing DDT-8.

Core memory now contains the DDT-8 program, the user's program, and a table of permanent symbol definitions. This table includes the definitions for all of the memory reference instructions, microinstructions, the ten basic IOT instructions, and the combined operations CIA and LAS.

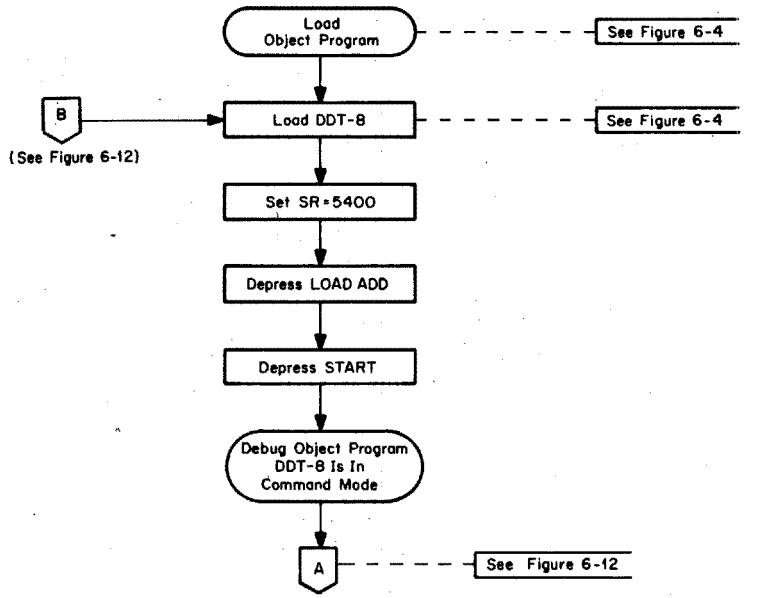


Figure 6-11. Loading and Executing DDT-8

The programmer may communicate with his program using either the mnemonic coding of the symbolic program or the octal coding of the binary program. Therefore, since DDT-8 is to perform all translation between binary and symbolic representation, it must have access in memory to the user's symbol definitions. The external symbol table tape (containing the user's symbol definitions) produced during assembly at the end of Pass 1 is now loaded into memory by DDT-8. Figure 6-12 illustrates the steps required in loading the Pass 1 symbol table tape.

The external symbols of the user's program are stored in memory immediately below DDT-8's permanent symbol table (mentioned in the paragraph below Figure 6-11). These symbols, and any others which may be entered from the console constitute the external symbol table. Additional symbols may be appended to the external symbol table by performing the steps illustrated in Figure 6-13.

A new external symbol tape can be generated off-line using the Teletype keyboard and low-speed punch merely as a tape preparation facility. The symbol tape can then be loaded into memory using DDT-8 as shown in Figure 6-12.

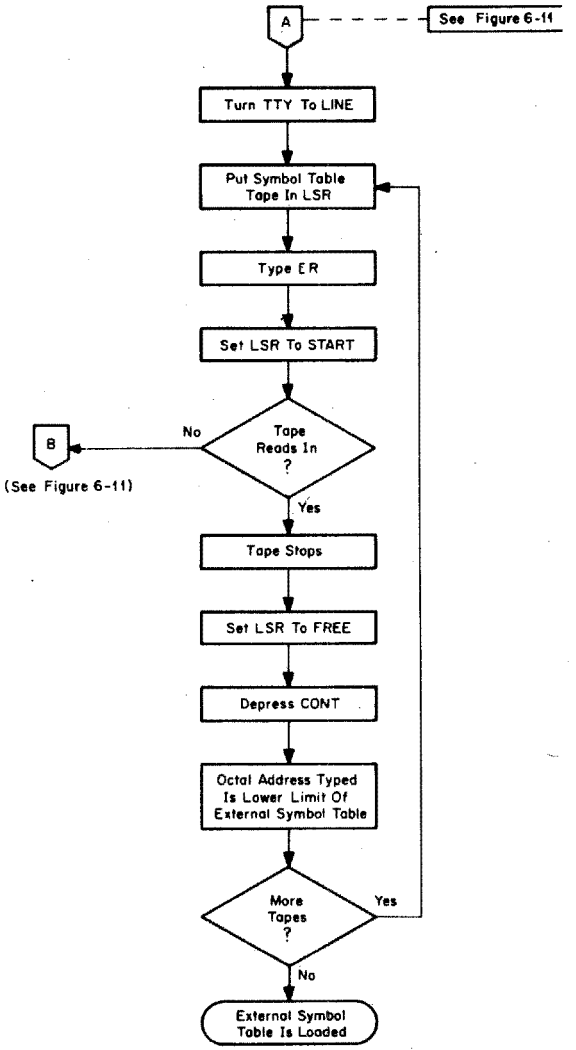


Figure 6-12. Loading the External Symbol Table Tape Using DDT-8

After loading the external symbol table tape, the address typed out will be the lowest memory location which is occupied by a symbol definition. The Programmer is now ready to begin debugging using DDT-8.

The example program to be checked out is a subroutine which accumulates the sum of the first n integers. The program is shown below.

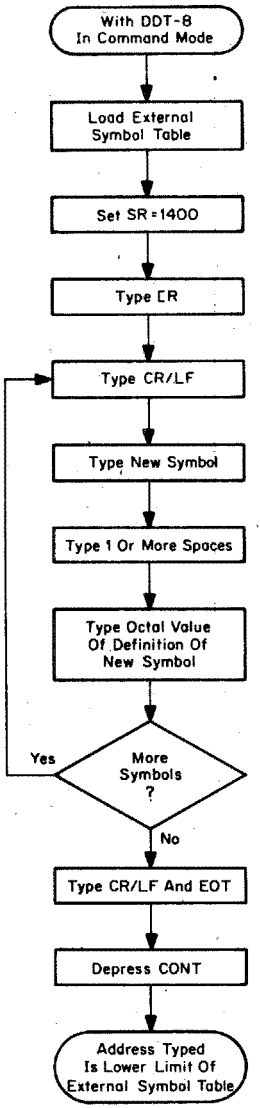


Figure 6-13. Appending New Symbols to External Symbol Table

```
/INTEGER SUMMATION SUBROUTINE
INTSUM,  0
         CLA
         TAD I INTSUM    /GET COUNTER
         DCA N           /SAVE IT
         DCA PSUM        /CLEAR PSUM
LOOP,    TAD PSUM        /GET SUM
         TAD INT         /ADD INTEGER
         DCA PSUM        /SAVE NEW SUM
         ISZ N           /DECREMENT COUNTER, N=0?
         JMP LOOP        /NO, NOT FINISHED
         TAD PSUM        /YES, FINISHED. AC=SUM
         ISZ INTSUM      /GET OVER ARG.
IEXIT,   JMP I INTSUM    /RETURN
N,       0
PSUM,    0
         *300
ITEST,   CLA             /INTSUM TEST PROGRAM
         JMS INTSUM
INT,     0               /PUT INTEGER HERE
RTN,     HLT
         $
```

For testing purposes, a short calling sequence has been included which provides the integer limit of the sum as an item of data. The first task is to place an integer in the core location that holds this datum, namely, the location labeled INT in the calling program. By typing the address of the core location (which in this case can be done by typing the address tag), followed by a slash, the programmer indicates to DDT-8 that he wishes to examine the contents of that location. Thus, he types

```
INT/
```

and DDT-8 responds to the slash by typing an expression which has the value of the contents of the specified location. In this case, the contents of INT = 0, and the line now appears as follows.

```
INT/AND 0000
```

NOTE: In the examples, information typed by DDT-8 is underlined to indicate a distinction from information typed by the programmer. In actual operation no underlining is present.

After typing the contents of the specified core location, DDT-8 types five spaces and waits. The location is now *open,* which means that its contents are available for modification. The programmer decides that the first test integer is 10. This must be an octal integer since DDT-8

performs no decimal arithmetic. With the location open, the programmer types the number 10. Then, to *close* the location, he types a carriage return (RETURN key) immediately after the number.

```
INT/AND 0000      10      (RETURN key was typed after 10)
```

Further access to this location is now denied until the programmer opens it again.

Having provided his data, the programmer is ready to start the program. If it works, it should stop almost immediately with the sum of the first 10_8 integers, which is 100_8, displayed in the accumulator (AC) lights. To start the program, the programmer types the following command.

```
ITEST[G
```

The left bracket ([) is printed when the ALT MODE key is typed; its function here is to identify the succeeding character as a DDT-8 command. The letter G specifies the action to be performed, which in this case causes DDT-8 to transfer control to the test program at location ITEST.

The programmer has typed the G command; his program starts executing. Immediately he observes that something is wrong—the program did not stop almost instantaneously, but ran for a very short, but observable, time. The contents of the AC lights are definitely *not* equal to 100_8.

The programmer restarts DDT-8 after each program execution, that is, he sets the SR to 5400 and then presses LOAD ADD and START switches. He must also open the PSUM and N Locations to restore them to zero before continuing debugging.

At this point, the programmer knows something is wrong, but he is not sure where the error lies. If he could interrupt the program *during* its operation, he might get a better idea of the nature of the problem. For instance, if he could verify that the data was transferred to the subroutine correctly, he could eliminate the calling sequence as a source of error.

The DDT-8 facility which allows the programmer to interrupt the operation of the program at any time is called the *breakpoint*. As its name implies, it allows him to break into the program sequence at some point and return control to DDT-8. He can specify a breakpoint by typing the address of the instruction where he wants to interrupt the sequence, and after this address he types the breakpoint command ([B). If he requests a breakpoint at location

INTSUM+3

the program will be interrupted when the datum is in the AC, but before it is deposited in the working location n. The breakpoint command would follow and the statement would appear as follows.

INTSUM+3[B

When this command is given, the information is retained by DDT-8 until the programmer executes the program. At that time, the sequence of operations performed my DDT-8 is:

1. The contents of location INTSUM+3 are saved in a special location within DDT-8.
2. In place of the instruction in location INTSUM+3, DDT-8 substitutes the instruction JMP I 4. Location 4 contains the address of a special breakpoint handling subroutine within DDT-8.
3. After the breakpoint has been set up, DDT-8 passes control to the programmer's program.

To ascertain that the error did not destroy the item of data in the calling program, check it by opening the location.

INT/AND 0010

The programmer then opens the PSUM and N Locations and restores them to zero.

Having ascertained that the datum is correct, start the program again.

ITEST[G

Almost immediately, the breakpoint is encountered and control returns to DDT-8. When the breakpoint occurs, DDT-8 saves the contents of the AC. It then types the address of the breakpoint, a right parenthesis, and the contents of the AC which have been saved.

INTSUM+0003)0010

The programmer sees that the transfer is correct. Therefore he restarts DDT-8 to continue debugging.

In similar fashion, the programmer moves the breakpoint to the end of the subroutine at location IEXIT. He discovers that at this point the error has manifested itself. He knows now that the trouble is in the initialization of the main loop. The breakpoint is set at LOOP to discover that the datum is placed in location n as desired. Now the breakpoint is moved to the end of the loop.

```
LOOP+3[B
ITEST[G
LOOP+0003)0010
```

At the end of the first pass through the loop, the contents of the AC are equal to the starting value of n. At this point, however, the contents of n itself have just been changed. If the subroutine is working properly, the contents of n should now be equal to 7770. The programmer investigates:

```
N/AND 0010
```

The programmer now realizes what he did wrong. In his attempt to save space by using the datum as a counting index, he forgot that the ISZ instruction increments the contents of negative value. Therefore, he must negate the counter by inserting the CIA instruction as shown below.

```
INTSUM,   0
          CLA
          TAD I INTSUM      /GET COUNTER
          CIA               /NEGATE COUNTER
          DCA N             /SAVE IT
```

and the program will run properly. Realizing this, the programmer concludes the debugging session by typing

```
[B
```

to remove the breakpoint. When the breakpoint is removed, the original contents of the break location is restored.

The example debugging session above was 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.

DEBUGGING NOTES. For DDT-8 to be useful, the programmer should be aware of and consider certain factors. Namely, the following:

1. Do not open any symbol table location.
2. The symbol table tape is loaded using the low-speed reader only.
3. Each user symbol occupies four locations in the symbol table storage area.
4. Input is interrupted when the symbol table storage area is full.
5. To enter a combined operate class IOT instruction into an open location, the combination must contain no more than two mnemonics, the second of which must be CLA. Any other combination is illegal and therefore ignored.
6. DDT-8 is restarted at its starting address, 5400, unless the programmer wishes to restart before he has punched a complete tape with checksum. In which case, he *must* restart at location 5401, and the checksum is preserved.

DDT-8 offers a very convenient feature, the word search feature, which allows the programmer to look at the contents of one, a group, or all locations in his program. A word search is indicated to DDT-8 when the programmer types the upper and lower limits of the search using the [L and [U commands and then the word search command, nnnn[W (nnnn is the word being searched). Address and location contents are printed as symbolic expressions or octal integers, according to the mode (symbolic or octal) specified by the programmer using the [S or [O command. The search *never* alters the contents of any location examined. The word search feature is detailed in *DDT-8,* Order No. DEC-08-CDDA-D.

ERROR DETECTION. DDT-8 is constantly watching for certain errors, those listed below.

1. Undefined symbol or illegal symbol
2. Illegal character
3. Undefined control command
4. Off-page addressing

When an error is detected, DDT-8 types a question mark (?) on the teleprinter and ignores all the information typed between the point of the error and the previous tab or carriage return.

SUMMARY OF SPECIAL KEYS AND COMMANDS. The programmer controls DDT-8 from the keyboard using special keys and commands. Certain keys have special meaning to DDT-8, and they must

be typed in certain formats in order to be acceptable to DDT-8. Tables 6-6 and 6-7 summarize the special keys and commands.

Table 6-6. DDT-8 Special Keys

Key	Meaning
(space)	Separation character
+ (plus)	Specifies address arguments relative to symbols
− (minus)	Same as +
. (period)	Current location; used in address arguments
= (equal)	Type last quantity as an octal integer
RETURN	Make modifications, if any, and close location
LINE FEED	Make modifications, if any, close location and open next sequential location
/ (slash)	Location examination character; when following the address location, the location is opened and its contents printed
↑ (up-arrow)	When following a location printout, the location addressed therein is opened
← (back-arrow)	Delete the line currently being typed

Table 6-7. DDT-8 Commands

Command	Meaning
Mode Control	
[O	Set DDT-8 to type out in octal
[S	Set DDT-8 to type out in symbolic
Input	
[R	Read symbol tape into external table from LSR, or define new symbol from keyboard
Program Examination and Modification	
nnnn/	Open location nnnn (nnnn may be octal or symbolic)
RETURN	Close location currently open; enter modification, if any
LINE FEED	Close location currently open and open next sequential location; enter modification, if any
↑ (SHIFT/I)	Close location currently open and open location address therein; enter modification, if any

Table 6-7. DDT-8 Commands (Cont.)

Command	Meaning
Breakpoint Insertion and Control	
[B	Remove current breakpoint
nnnn[B	Insert a breakpoint at location nnnn
nnnn[G	Go to location nnnn and start program execution
n[C	Continue from breakpoint, execute breakpoint n times and return control to programmer. If n is absent, it is assumed to be 1.
Word Search	
nnnn[W	Begin word search for all occurrences of expression nnnn masked by the contents of M between the limits imposed by L and U. M, L, and U are locations within DDT-8 which may be opened, modified and closed exactly as any general location in the user's program.
Output	
[T	Punch leader/trailer tape
mmmm;nnnn[P	Punch binary tape from memory bounded by addresses mmmm and nnnn
[E	Punch end-of-tape (i.e., checksum and trailer)
Address Tags	
[A	Accumulator storage (at breakpoint)
[L	Lower limit of search
[U	Upper limit of search
[M	Mask; used in search
[Y	Link storage (at breakpoint)

NOTE: The character [is generated by depressing ALT MODE on the keyboard.

For a thorough description of DDT-8, see *DDT-8,* Order No. DEC-08-CDDA-D.

DEBUGGING WITH ODT-8

Using ODT-8, the programmer can run his binary program on the computer, control its execution, and make alterations to his program by typing on the Teletype keyboard. ODT-8 has the same capabilities as DDT-8 except that the programmer must reference his program using its octal representation instead of mnemonic symbols, and ODT-8 commands are formulated differently.

ODT-8 occupies less core memory than DDT-8, and can be loaded into either lower (SA = 1000) or upper (SA = 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-8 should be loaded in the upper pages of memory, and vice versa. As with DDT-8, the user program can not occupy (overlay) any location used by ODT-8, including the breakpoint location which is location 0004 on page zero.

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-8 program (either high or low version).

To begin the debugging run, first ascertain that BIN is in core memory, then load the programmer's binary tape followed by the binary ODT-8 tape (see Figure 6-4 for loading procedures). Figure 6-10 illustrates the procedures for loading and executing ODT-8.

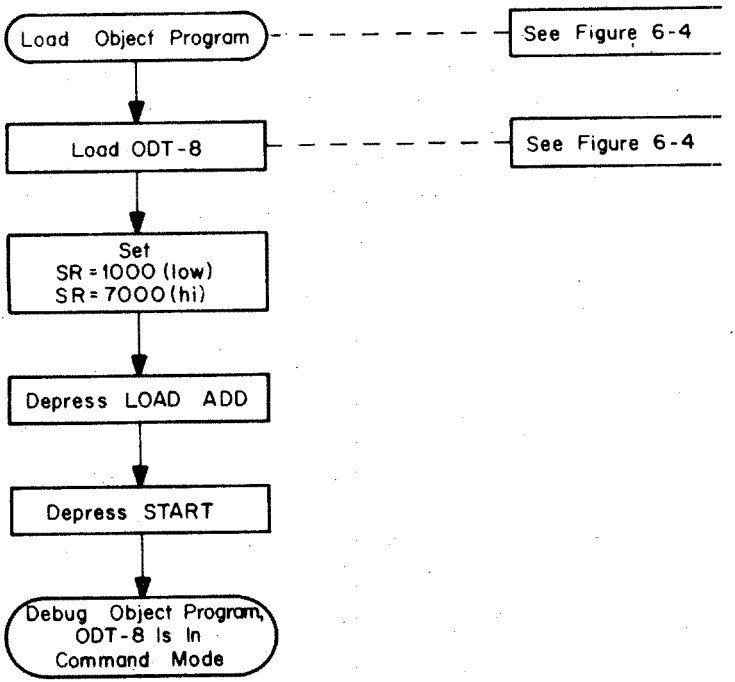


Figure 6-14. Loading and Executing ODT-8

The example program debugged earlier in this chapter under "Debugging with DDT-8" could be debugged here. However, since the only difference would be the appearance of the debugging commands, we need only show the command formats and give a brief explanation of each, which can be found in Table 6-8.

Table 6-8. ODT-8 Commands

Command	Meaning
/	Reopen last named location; modify if desired
nnnn/	Open location nnnn; modify if desired
RETURN	Close currently opened location (typing another command also closes the currently opened location)
LINE FEED	Close currently opened location and open next sequential location
↑ (SHIFT/N)	Close currently opened location, take contents as MRI and open and type contents of referenced location
← (SHIFT/O)	Close currently opened location and open indirectly, i.e., the content of the opened location is interpreted as the address of the location whose content is to be typed and available for modification.
nnnnG	Go to location nnnn and transfer control to programmer for modification if desired.
B	Remove previously established breakpoint and restore breakpoint to original status.
nnnnB	Establish a breakpoint at location nnnn.
A	Open location containing the contents of the accumulator (AC).
LINE FEED	Open location containing the contents of the link.
C	Continue (proceed) from a breakpoint, i.e., execution begins with the breakpoint instruction.
nnnC	Continue (proceed) from a breakpoint and iterate nnn (octal) times through breakpoint.
M	Open search mask, i.e., open for modification the location containing the current value of the search mask (mask is initially set to 7777).
LINE FEED	Open lower search limit (initially set to 0001; change by typing new lower limit after ODT-8 has typed 0001).
LINE FEED	Open upper search limit (initially set to SA of ODT-8, 1000 or 7000; change by typing new upper limit after ODT-8 has typed initial limit).
nnnnW	Word search, i.e., using the mask and lower and upper limits, ODT-8 searches for instruction nnnn.

Table 6-8. ODT-8 Commands (Cont.)

Command	Meaning
T	Punch leader/trailer tape, i.e., type T and *then* turn punch ON; turn punch OFF when sufficient tape has been punched, then depress STOP on the console. Restart ODT-8 at appropriate SA.
mmmm;nnnnP	Punch binary core image of locations mmmm through nnnn. After executing this command by typing the RETURN key, the computer halts for the programmer to turn the punch ON and to depress CONT on the console. When specified image is punched, turn punch OFF.
E	Punch checksum and trailer tape. After executing this command by typing the RETURN key, the computer halts for the programmer to turn the punch ON and to depress CONT on the console. When sufficient trailer tape has been punched, depress STOP on the console and restart ODT-8 at appropriate SA.

The address of the current or last location opened is remembered by ODT-8 even after the commands G, C, B, T, E, and P, and may be reopened merely by typing / (the slash command).

After debugging his program, the programmer can command ODT-8 to produce a binary tape of the debugged program, which can be loaded by the BIN Loader and run on the computer. The P or punch command is used with the low-speed punch only. Figure 6-15 illustrates the procedure for generating the binary coded object tape using the high-speed punch.

Typing the G command alone is an error, it must be preceded by an octal address to which control will be transferred. If not preceded by an address, a question mark will not be typed but it will cause control to be transferred to absolute 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, which means the tape cannot be loaded and run properly.

DEBUGGING NOTES. Only one breakpoint may be in effect at one time, therefore, requesting a new breakpoint removes the previously existing breakpoint. A breakpoint must not be set to any location in the program which is altered during execution; if so set, the alteration will destroy the breakpoint.

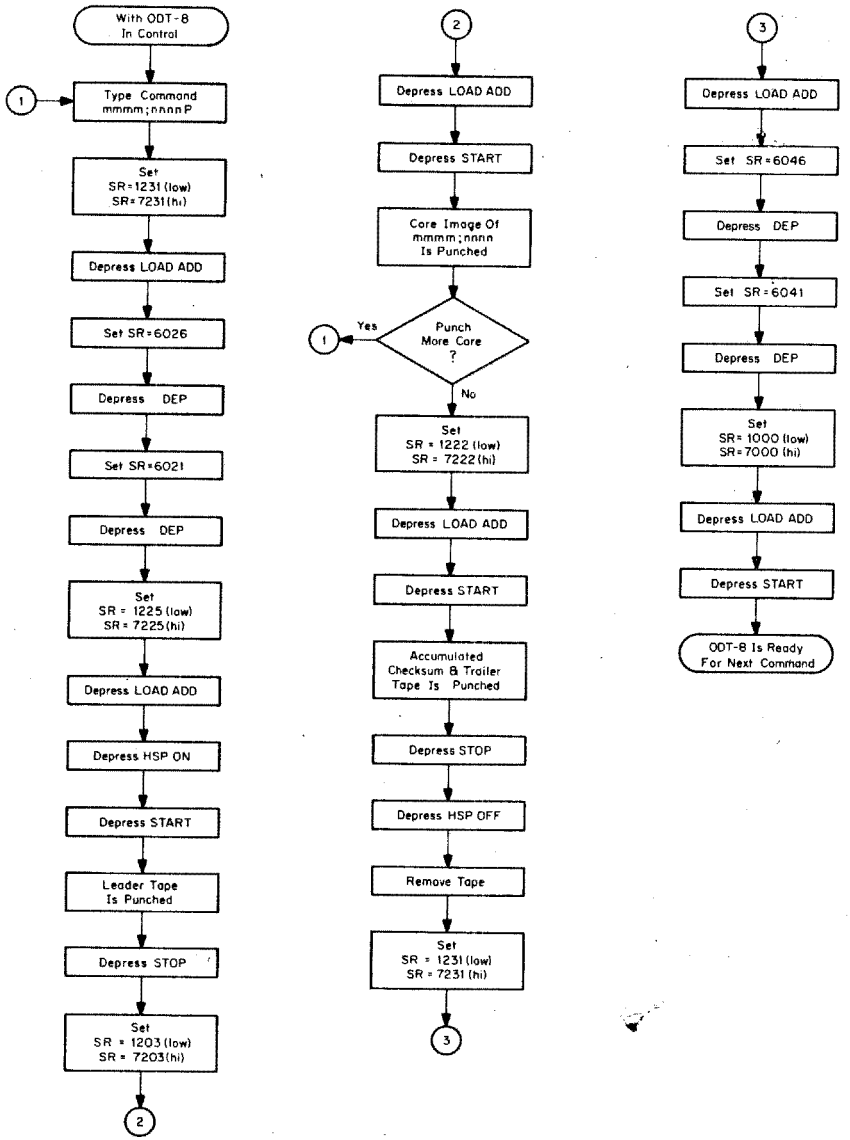


Figure 6-15. Generating Binary Tape Using High-Speed Punch

An open location can be modified by typing the desired octal instruction and closing the location. Any octal number from 1 to 4 digits in length is a legal input, but typing a fifth digit is an error and will cause the entire modification to be ignored and a question mark to be typed by ODT-8.

ERROR DETECTION. The only legal inputs to ODT-8 are control commands and octal digits. Any other characters will cause the character or line to be ignored and a question mark (?) to be typed out by ODT-8.

For a more detailed discussion, see *ODT-8,* Order No. DEC-08-COCO-D.

FOCAL

FOCAL (Formula CALculator) is an online, conversational interpreter designed to help scientists, engineers, and students solve complex numerical problems. The language consists of short easy to learn imperative English statements. Mathematical expressions are typed in standard notation.

With FOCAL, the programmer has the full calculating power and speed of the computer at his fingertips. It is used for simulating mathematical models, for curve plotting, for handling sets of simultaneous equations in n-dimensional arrays, and many other kinds of problems.

The procedure for loading the binary-coded FOCAL tape is illustrated in Figure 6-16. When FOCAL is properly loaded in core, the programmer sets the switch register to 0200 (the starting address of FOCAL), depresses the LOAD ADDress switch and then the START switch, and FOCAL will respond with its initial dialogue (see Chapter 9). The initial dialogue is a question and answer sequence, with FOCAL asking questions and the user providing the answers.

After the programmer has answered the initial dialogue questions, FOCAL will type an asterisk, indicating that it is ready to accept commands from the programmer.

The FOCAL language and its numerous features are explained in Chapter 9.

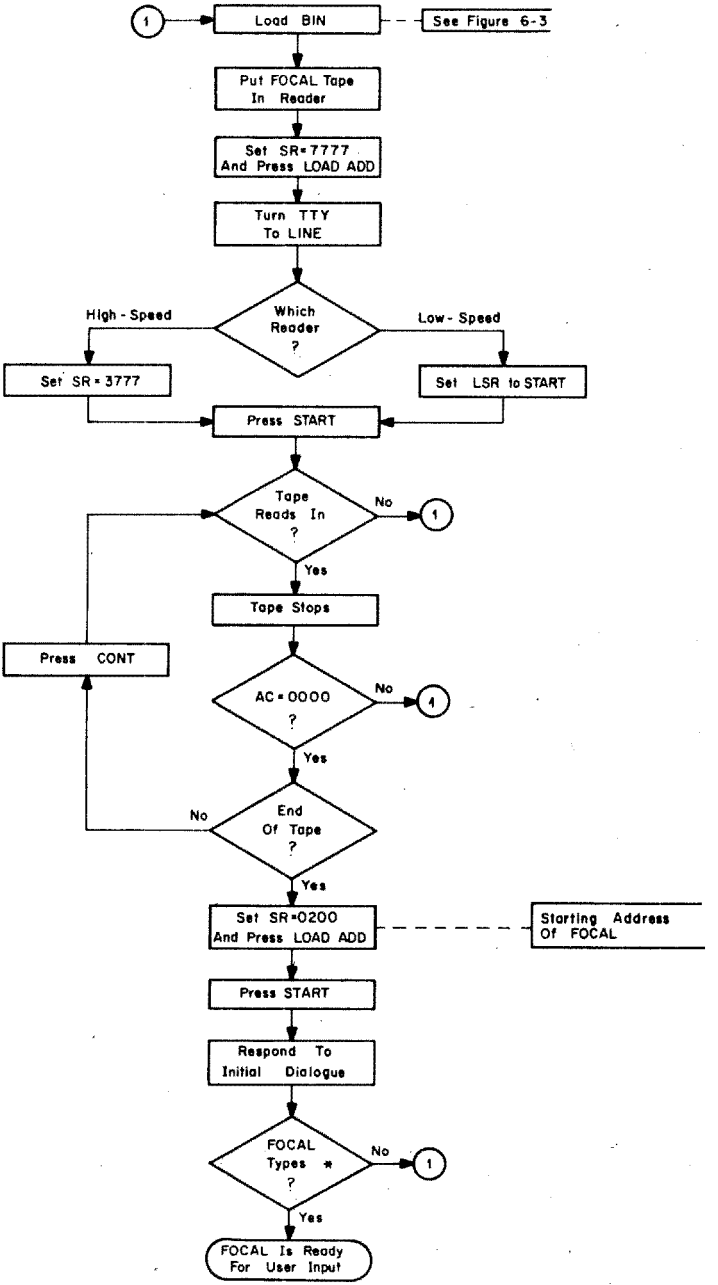


Figure 6-16. Loading and Executing FOCAL

Chapter 7

Disk Monitor System

The PDP-8 Disk Monitor System is designed for any computer in the PDP-8 family having at least one DECdisk. This system consists of a keyboard-oriented Monitor, which enables the user to efficiently control the flow of programs through his PDP-8, and a comprehensive software package, which includes a FORTRAN Compiler, Program Assembly Language (PAL-D), Edit program (Editor), Peripheral Interchange Program (PIP), and Dynamic Debugging Technique (DDT-D) program. Also provided is a program (Builder) for generating a customized monitor according to the user's particular machine configuration (amount of core, number of disks or DECtapes, etc.).

The system is modular and open ended, permitting the user to construct the software required in his environment, and allows the user full access to his disk (or DECtape)—referred to as the *system device*—for storage and retrieval of his programs. By typing appropriate commands to the Monitor, the user can *load* a program (construct it from one or more units of binary coding previously punched out on paper tape or written on the disk by the Assembler, and assign it core), *save* it (write it out, with an assigned starting address, on the system device), and later *call* it (read it back into core from the system device) for execution.

GENERAL DESCRIPTION

The PDP-8 Disk Monitor System permits the user to control the flow of programs through his computer and takes full advantage of the extended memory capabilities of disk or DECtape. In addition to the Monitor, the system also contains a library of system programs. Together, they provide the user with the capabilities of compiling, assembling, editing, loading, saving, calling, and debugging his own programs.

Monitor Residence

Monitor, as well as system and user programs, is stored on and retrieved from the user's *system device*. To obtain a working Monitor, the user must first build his own customized version, via the easy-to-use dialogue technique of the System Builder program and store this version on his system device. Following this, the user then creates his System Program Library on the system device. Both of these procedures are described in Appendix A of the *Disk Monitor* manual (DEC-D8-SDAB-D).

In core, the resident part of Monitor (called *head of monitor*) resides in the top page (locations 7600 through 7777) of field 0. The starting address of Monitor is 7600; 7642 is the entry address to the system I/O routine, which performs all reading and writing on the system device. Nonresident portions of Monitor, such as those routines which perform SAVEs and CALLs, are automatically called in as needed, and in core they share the area from location 7000 through 7577. (These portions disappear after use, leaving this area for the user.)

System Modes

At any point in time, the system is running in one of two modes: *Monitor mode* or *user mode*. Monitor mode is entered (1) whenever the Monitor is started (see "Initializing the Monitor") or (2) when CTRL/C (↑ C) is typed while running any system program. Monitor mode is signalled by the Monitor typeout of a dot (•). At both Monitor and system program time, Monitor is able to sense a ↑ C typein, causing the system to enter Monitor mode, return to Monitor at location 7600, and respond with a dot (•) typeout. At this point, the user can issue any Monitor command via the Teletype keyboard.

User mode is present whenever the system is executing a system or user program. System programs signal user mode by responding with an asterisk (*) typeout.

SYSTEM PROGRAM LIBRARY

The Monitor System's library of programs presently consists of the Peripheral Interchange Program (PIP), Disk System Editor (Editor), PAL-D Disk Assembler (PAL-D), 4K Disk FORTRAN (FORTRAN D), and Dynamic Debugging Technique for Disk (DDT-D), and this list is destined to lengthen with time.

To load a program using the Monitor System, the Loader makes certain queries to which the user must type a reply. The queries are the same for all programs. The user's replies will vary, however, depending on the particulars of the program being loaded.

When loading a program into core, the user should first check to see whether Monitor is in core. This is done by typing ↑ C (CTRL key and then the C key). The ↑ C will not echo (not print on the teleprinter). If Monitor is in core, it will respond by typing a period (.) at the left margin of the teleprinter paper. If a period is not typed in response to ↑ C, Monitor is not in core.

The library system includes the Disk System Binary Loader (LOAD) which is automatically saved on the disk at Build time. Disk System Binary Loader is described in "Loading Programs—Disk System Binary Loader."

The user may save any program on the disk by responding to the last period typed by Monitor with the word SAVE, a 1-to-4-character name of the program, the type of program (user or system), whether it's a one or more page save, and the location of its starting address, as is thoroughly described in "Saving Programs (SAVE Command)."

After each program is saved on the system device, it may be called (*i.e.,* transferred from the disk into core) merely by responding to Monitor (to a period) with the four characters designated as the name of that program, as explained in "Calling a Program (CALL Command)."

The programs in the System Library are described below. Further information concerning these programs may be found in the *Disk Monitor* manual or in the other manuals referred to.

Peripheral Interchange Program

The Peripheral Interchange Program (PIP) performs general utility operations, such as listing the contents of specified directories, deleting unwanted files from the system device, and transferring files between devices, and copying specified files. PIP enables the user to do any of the above operations merely by typing commands from the teleprinter keyboard.

A summary of PIP options follows.

L	List entire system directory
B	Copy a binary file
D	Delete a file to be specified
F	Copy a FORTRAN binary file
M	Move directory to safe disk
P	Protect disk 1 (blocks 0-176)
R	Restore directory from safe disk
S	Copy a system file[1]
U	Copy a user file[1]
RETURN or A	Copy a ASCII file

Disk System Editor

The Disk System Editor enables the user to generate and edit symbolic programs online from the teleprinter keyboard. The symbolic program may be either printed on the teleprinter, punched on paper tape using the high- or low-speed punch, or stored on the system device as a user program.

Editor operates either in command or text mode. In command mode, all typed input is interpreted as a command instructing Editor to perform a certain operation or to allow the user to perform an operation on the text stored in the buffer. In text mode, all typed input is interpreted as text to replace, to be inserted into, or to be appended to the contents of the text buffer.

The command language of the Disk System Editor is identical to that described in the *PDP-8 Symbolic Editor* manual (DEC-08-ESAB-D) with a few exceptions.

A summary of Editor commands follows. Each command is terminated by pressing the RETURN key, which initiates execution of the command.

Function	Command	Meaning
Read	R	Read incoming text and append to buffer until a form feed is encountered
Append	A	Append incoming text to any already in the buffer until a form feed is encountered.
List	L	List the entire buffer.
	nL	List the line n.
	m,nL	List lines m through n.

[1]User system files may not be copied onto paper tape.

Function	Command	Meaning
Proceed	P	Proceed and output the entire contents of the buffer and return to command mode.
	nP	Output line n, followed by a form feed.
	m,nP	Output lines m through n, followed by a form feed.
Trailer	T	Punch four inches of trailer.
Next	N	Punch the entire buffer and a form feed; kill the buffer and read next page.
	nN	Repeat the above sequence n times.
Kill	K	Kill the buffer.
Delete	nD	Delete line n.
	m,nD	Delete lines m through n.
Insert	I	Insert before line 1 all text until a form feed is encountered.
	nI	Insert before line n until a form feed is encountered.
Change	nC	Delete line n and replace it with any number of lines from the keyboard until a form feed is encountered.
	m,nC	Delete lines m through n, replace from keyboard as above until form feed is encountered.
Move	m,n$kM	Move and insert lines m through n before line k.
Get	G	Get and list the next line beginning with a tag.
	nG	Get and list the next line after line n which begins with a tag.
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, allow modification.
End file	E	Process the entire file (perform enough NEXT commands to pass the remaining input to the output file) and create an end-of-file indication; legal only for output to the system device. If the low-speed paper tape reader is used for input while performing an E command, the paper tape reader will eventually run out of tape, and at this point typing a form feed will allow the command to be completed.

PAL-D Disk Assembler

PAL-D, the acronym for Program Assembly Language for the Disk system, is the symbolic assembly program designed primarily for the 4K PDP-8 family of computers with disk or DECtape.

The PAL-D Assembler performs many useful functions, making machine language programming easier, faster, and more efficient. Basically, the Assembler processes the user's source program statements by translating mnemonic operation codes into the binary codes needed in machine instructions, relating symbols to numeric values, assigning absolute core addresses for program instructions and data, and preparing an output listing of the program which includes notification of any errors detected during the assembly process.

The Assembler is thoroughly documented in the *PAL-D Disk Assembler Programming Manual* (Doc. No. DEC-D8-ASAA-D).

FORTRAN-D for the Disk System

FORTRAN-D (FORmula TRANslation for the Disk System), is an expanded version of standard PDP-8 FORTRAN designed for PDP-8 family computers with disk or DECtape units. FORTRAN-D is described in detail in the *4K FORTRAN Programming Manual* (DEC-08-AFAC-D).

FORTRAN-D contains a compiler, an operating system, and two programs to aid debugging. The FORTRAN compiler is used to convert a source program into an object program. The FORTRAN operating system is used to execute the object program.

This version of FORTRAN is designed to facilitate user/system communication by allowing the user to type appropriate commands from the teleprinter keyboard, eliminating the need to toggle input using the switch registers.

FORTRAN statements specify the computations required to carry out the processes of the FORTRAN program. There are four types of statements provided for by the FORTRAN language:

a. Arithmetic statements define a numerical calculation.
b. Control statements determine the sequence of operation in the program.
c. Specification statements define the properties of variables, functions, and arrays appearing in the source program. They also enable the user to control storage allocation.
d. Input/output statements are used to transmit information between the computer and related input/output devices.

The compiler consists of a loader (FORT) and the main portion of the compiler (.FT.). This version of the compiler differs from the

standard PDP-8 4K FORTRAN compiler in the following ways.

a. It uses the disk or DECtape unit (whichever is the system device) during its operation.
b. It will compile programs which have been stored on the system device or programs which have been prepared on punched paper tape.
c. It will generate a FORTRAN binary output file either on the system device or on punched paper tape.
d. Significant improvements have been employed with the READ and WRITE statements.
e. Input and output devices are determined using the Command Decoder.
f. It is possible to terminate compilation at any time by typing ↑ C, thus returning control to Monitor.
g. Within certain restrictions, a program compiled with output to the system device may be executed immediately when the user types ↑ P after compilation of the program.
h. Statement numbers need not be delimited by a semicolon, unless the user wishes them to be employed for appearance.
i. Statements without preceding numbers must be preceded by a space, a tab, or a semicolon.

Debugging Aid (Symbolprint). Symbolprint (STBL) is a program which may be used with the FORTRAN compiler. Its purpose is to help the user create and debug his FORTRAN programs by providing certain information about the compiler-generated interpretive code. Symbolprint may be used only immediately after a program has been compiled using the Disk FORTRAN compiler.

Symbolprint provides the following information:

a. The core limits of the interpretive code.
b. A list of variable names and their corresponding locations (the symbol table).
c. A list of statement numbers and their corresponding locations (the statement number table).

Operating System. The FORTRAN operating system consists of a loader (FOSL) and the interpreter and arithmetic subroutine package (.OS.). This version of FOSL differs from the paper tape FORTRAN operating system in the following ways.

a. It will load and execute programs which have been compiled and saved on the system device or programs which have been compiled on paper tape.

b. FOSL may be called directly by the compiler when a program has been compiled and saved on the system device. This is referred to as compile-and-go mode.
c. FOSL is able to recognize READ and WRITE statements which may read and write data in ASCII format on either the low-speed paper tape reader/punch, the high-speed paper tape reader/punch, or the system device.
d. The execution of a FORTRAN program may be interrupted by the user at any time by typing ↑ C; control will be returned to Monitor.

Debugging Aid (Diagnose). Diagnose (DIAG) is a basic system program which helps the user debug his FORTRAN program. It is intended to be used in conjunction with the PDP-8 4K FORTRAN Operating System and revised FORTRAN Symbolprint. Diagnose provides the following information.

a. If stack overflow or underflow has occurred, it will type a message indicating which of the five run-time stacks caused the error.
b. It will type a message indicating the contents of the current location counter (CLC).
c. If the counter stack is nonempty, it will type the contents of that stack.
d. If location zero is nonzero, it will type the contents of that location (minus one), indicating the point at which some FORTRAN systems error occurred.

Dynamic Debugging Technique for the Disk System

The Dynamic Debugging Technique for the Disk System (DDT-D) is used for online checking, testing, and altering object programs by typing from the teleprinter keyboard. When debugging online, the user checks his program at the computer, controlling its execution, and making corrections or changes to his program while it is running on the computer.

When using DDT-D, the user should have a listing of his program and its symbols so that he can update the program listing as corrections and changes are made to his program. The user may refer to variables and tags by their symbolic names or by their octal values.

DDT-D operates as described in *DDT Programming Manual* (Doc. No. DEC-08-CDDA-D), except where that manual differs from this one, in which case this manual has precedence.

DDT-D can be considered as being in three sections.

a. DDT Proper (.DDT)	A slightly modified version of DDT-8 (low); occupies core locations 200 through 4577 and the three breakpoint locations.
b. Driver (DDT)	Resident in core with its origin set above DDT proper (above 4577); it is a 2-page program plus a 1-page once-only program, and it contains breakpoint insertion and removal logic, overlay routines, continuation iteration count and control, and breakpoint list.
c. User Core Image File (.SYM)	Occupies same storage area as DDT proper and is used for swapping DDT proper and the user program to and from the system device.

DDT-D is an expanded version of DDT-8 with the following exceptions.

a. Three breakpoints (as opposed to only one in DDT-8)
b. No punching (program may be output on the system device)
c. No switch options (user direction is via keyboard)
d. No halts (continues when user types ↑ P)

EQUIPMENT REQUIREMENTS

The minimum equipment requirements of the PDP-8 Disk Monitor System are as follows.

A basic PDP-8 family computer with 4K of core
Teletype
3-Cycle Data Break (Option required with PDP-8/S)
At least one DF32 Random Access DECdisk File or a TC01 DECtape Control with a TU55 DECtape transport. The DECtape must have timing and mark tracks written on it prior to use.

NOTE
The system will recognize up to 32K of core, up to four disks (1 Type DF32 and 3 Type DS32's), up to eight DECtape transports (TC01 Control only) and a high-speed paper-tape reader/ punch.

INITIALIZING THE MONITOR

The following discussion assumes that the user has built a customized Monitor via the System Builder and has stored it on his system device; it further assumes that the user has bootstrapped this customized Monitor into core and that the Monitor is currently intact. If this is not the case, such a Monitor must be built and bootstrapped into core by the steps outlined in the *Disk Monitor System* manual (DEC-D8-SDAB-D).

Monitor start is at location 7600. A jump to this location can be made by either (1) stopping the machine, setting the switches to 7600, and pressing LOAD ADD (load address) and START, or (2) typing ↑C when in Monitor mode or when a system program (or any user program which includes coding to sense a ↑C typein) is running.[1]

Monitor start performs the following actions.

a. Saves the coding from location 7200 through 7577 in the first two scratch blocks on the system device.
b. Reads blocks 1 and 2 (containing the rest of Monitor) from the system device into these locations.
c. Transfers control to Monitor, which responds with a carriage return, line feed, and a dot.

A Monitor restart can be performed by typing RUBOUT to Monitor. A Monitor restart performs the same actions as described above except for Subparagraph a. A common use for RUBOUT is to terminate a command string when the operator has discovered that he has made a mistake. The command string is ignored, and Monitor responds as described in Subparagraph c. The user core image on the system device is not changed by RUBOUT (it is changed, however, by ↑C).

COMMAND STRINGS

The user types commands in the form of *command strings* to direct Monitor, or a system program, to perform some action. Command strings are simple in format and afford the user an easy means of communicating with the system.

Monitor indicates its readiness to accept a command string by typing a dot, and at this point, the user can type some Monitor command, such as CALL or SAVE.

Unless otherwise indicated, all command strings are terminated by the RETURN key, which echoes as a carriage return, line feed.

[1]A start instruction (ST=7600) is issued when running Loader causes a jump to 7600 after loading has been performed. Certain errors also cause a jump to this location.

System programs indicate their readiness to receive information by typing either an asterisk or a query. The most common queries are as follows.

*OUT- Requests that the user specify one output device name. In the case of disk or DECtape the filename to be assigned to the output data must also be specified (see "Filenames").

*IN- Requests that the user specify one or more (up to five) input device names. For disk and DECtape, filenames of input files must also be specified (see Filenames).

*OPT- Requests that the user specify one option or switch, entered as a single alphanumeric character; see the appropriate reference for options available in each system program.

This communication between the system and the user is handled by a portion of Monitor known as the Command Decoder.[1] Command Decoder is called into core by the system when needed and occupies any four contiguous pages of core.

Command String Format

Command strings are composed of a few basic elements and follow certain rules of punctuation. Their basic elements are as follows.

a. Device names
b. Filenames
c. Punctuation
d. Special characters

Each of these elements is described in the following paragraphs.

Device Names. Device names permitted in command strings are as follows.

Dn: DECtape unit, if both disk and DECtape are present in the system (n=unit number, 0 through 7)

S: System device (disk or DECtape unit 0)

R: High-speed paper tape equipment (reader or punch)

T: Low-speed paper tape equipment on the Teletype (reader or punch)

Filenames. Filenames are limited to four characters in length and can be composed of any combination of alphanumeric characters or special characters[2] with the following exceptions.

a. Imbedded spaces cannot appear in a filename (they are ignored).[3] However, trailing spaces are permitted.

[1]Command Decoder is a system program (.CD.) which is saved on the system device at Build time.

[2]Although both printing and nonprinting keyboard characters are allowable, printing characters are recommended.

[3]Note that Monitor is given the filename EX C; one reason for this unconventional use of an imbedded blank is to protect Monitor from accidental destruction by the user (e.g., deletion via PIP).

b. A filename cannot be one of the following words or symbols.

CALL SAVE ! , ; :

Extensions to the filenames specified by the user are automatically appended by the system. They are used internally by the system and cannot be referred to or modified by the user.

SYS (n)	Saved system program file in core bank n.
USER (n)	Saved user program file in core bank n.
ASCII	Source language program file (input to PAL-D Assembler or FORTRAN Compiler).
BINARY	Binary program file (output from PAL-D Assembler).
FTC BIN	Interpretive binary file (output from FORTRAN Compiler).

Filenames (and extensions) are meaningful only for file-structured devices (disk and DECtape). If they are specified for other devices, they are ignored. Both the filename and extension name appear on directory listings produced by the list feature in PIP.[1]

Example:

```
NAME  TYPE      BLK
 8D
PIP  . SYS (0)   0015
SRC1 . ASCII     0007
BIN  . BINARY    0001
SRC1 . USER (0)  0001
```

Punctuation. Punctuation within command strings is as follows.

, Used to separate device names, when more than one is given in a command string. The comma is also used to separate core references in a SAVE command string, when more than one contiguous area of core is specified.

; Precedes the entry specification in a SAVE command.

: Terminates each device name. The colon is also used following the filename in a SAVE command to indicate that the file is to be saved as a user program.

- Separates the beginning and ending addresses of a contiguous core area specification in a SAVE command.

! Follows the filename in a SAVE command when a file is to be saved as a system program.

[1]"8D" in example means VERSION 8, change D.

Special Characters. Special characters are used as described below.

↑C	If given while the system is in Monitor mode or a system program is running, control is returned to Monitor start (location 7600). Monitor responds with a dot. ↑C is typed by holding down the CTRL key and striking ↑C. ↑C does not echo (does not print).
↑P	Typed in response to a ↑ typeout. Instructs the system to proceed with the next operation.[1] ↑P is typed by holding down the CTRL key and striking P. ↑P does not echo (does not print).
RETURN	Carriage return terminates current command string input. When typed alone, in response to a system query, it indicates that the user does not desire to specify the item (e.g., device name) requested. Echoes as carriage return, line feed.
RUBOUT	Causes the current command string to be ignored, and the system returns to the beginning of the command string and is ready to receive a new command. RUBOUT does not echo.

Examples of Command Strings

These examples illustrate the elements and rules explained above. Samples of both Monitor commands and system program commands are given.[2]

Monitor Commands

.CALL PRG1	Call the user program file, PRG1, from the system device into core for execution.
.SAVE PALD! 0-7577; 6200	Save a program, previously loaded by Loader into locations 0 through 7577 or core, on the system device as a system program (!). Assign a starting address of 6200 and a filename, PALD.

SYSTEM PROGRAM COMMANDS

*IN-S:PRO2	Use the file PRO2 on the system device as the input file.
*IN-S:TST1,R:	Use the file TST1 on the system devise and one file from the high-speed paper tape reader as the input files.
*OUT-D5:SPEC	Write the output file on DECtape unit No. 5 and assign it the filename SPEC.

[1]↑P can also be used to prematurely terminate certain operations while in progress (e.g., the typing out of a file directory by the list option in PIP).
[2]In all examples, system response (typeout) is underlined for clarity.

LOADING PROGRAMS—DISK SYSTEM BINARY LOADER[1]

The Disk System Binary Loader takes as input the binary coding produced by the PAL-D Assembler and loads it into core in executable form. When loading is completed, Loader "disappears" after first entering the loaded program at the starting address typed by the user just prior to loading (see "Loader Operating Procedures" below). Loader accepts input from the system device or paper tape.

Loader requires one pass for any program which does not load above location 6777 (field 0). Loader uses core from location 167 through 177 and 6000 through 7577, and the resident portion of Monitor occupies the remainder of field 0. One-pass loading reads input files only once.

Two passes are required for all other programs (i.e., programs loading above 6777). In two-pass loading, programs can be loaded in all of field 0, except locations 7600 through 7777.[2] Two-pass loading requires that input paper tapes be read through the reader twice.

Loader Operating Procedures

.LOAD	Direct Monitor to bring Loader from the system device into core for execution.
*IN-	Loader requests source of input(s). Type one or more device names, separated by commas. If an input device is a file-structured device (S:, Dn:) include filename(s). Up to five files can be specified.
Examples	
*IN-R:,R:,R:	Input three tapes from the paper tape reader.
*IN-S:INPT	Input the file INPT from the system device.
*IN-S:BIN2,R:	Input the file BIN2 from the system device and one tape from the paper tape reader.
*IN-S:BIN1,S:BIN2	Input the files BIN1 and BIN2 from the system device.

[1]The Disk System Binary Loader is a system program saved on the disk at Build time. It is called by the user in the same manner as any system program. It occupies locations 7000-7577 and has a starting address of 7000.

[2]In 8K and larger systems, Loader sets up locations 7574 through 7577 to perform a start in fields other than 0. It is the user's responsibility to protect these locations if he wants to start in other than field 0.

*	If device(s) are valid and filenames (if any) are actually found on the system device, Loader responds with one asterisk for each correct input.
*OPT-	Loader requests mode desired (one-pass or two-pass).
Examples	
*OPT-1	One-pass loading desired; no programs are loaded above location 6777.
*OPT-2 (or anything else)	Two-pass loading desired; programs can be loaded above location 6777.
*ST =	Loader requests the starting address to which control is to be transferred when loading is completed. The address is typed in the form fnnnn where f = field number[1] (omitted if field 0), and nnnn = location within field
Examples	
*ST =	Load into field 0.
*ST = 7600	Return to Monitor after loading.
*ST = 0	
*ST = 30225	Load into field 3. Jump to location 255, field 3, after loading.
*ST = 10000	Load into field 1. Return to Monitor after loading into field 1. Loader now types a series of up-arrows, one at a time, as explained below. Following each up-arrow typeout, the user is required to perform one or more actions.
↑↑↑	First up-arrow: Loader is ready to load. If paper tape input, put the tape in the reader. Type ↑ P.[2]

[1]The f-digit forces Loader to start loading into the specified field until a "field setting" is found in the input file or tape.
[2]If Teletype paper tape equipment is used, type ↑ P *before* turning on the reader.

Second up-arrow: End of pass 1. If operating in one-pass mode, type ↑P to jump to previously specified starting address.

If operating in two-pass mode, type↑P.

The next two up-arrows appear only if operating in two-pass mode.

Third up-arrow: Reload paper tape input for pass 2. Type ↑P.

Fourth up-arrow: End of pass 2. Type ↑P to jump to previously specified starting address.

Multiple Input Files

An up-arrow is typed out as the processing of each input file is completed. If paper tape input, insert the next file in the reader and type ↑P. Repeat the above step until all files given in response to the *IN-request have been processed.

If in two-pass mode, each tape must be entered twice, in the order

T1, T2, T3, ... T1, T2, T3, ...

After all files have been entered the required number of times, type ↑P to jump to the previously specified starting address.

NOTE

After each input paper tape is read, the high-speed paper tape version of Loader loops until the user types ↑P to continue. However, the low-speed paper tape version *halts*. Thus, when using the Teletype paper tape equipment for input, the user need not type ↑P but *press* CONT *on the console* and *start* the paper tape reader.

At this point, Loader disappears and control is transferred to the previously specified starting address.

The loading parameters for system programs are given in Table 7-1.

Loader Error Messages

An illegal checksum error condition causes Loader to type

?

and return to Monitor after the user types ↑P or ↑C. Error messages for illegal filenames or devices are as specified in "System Error Messages."

SAVING PROGRAMS (SAVE COMMAND)

The SAVE command enables the user to write core images of system or user programs from core onto his system device for subsequent call-in (CALL) and execution. For example, a program which has been loaded by Disk System Binary Loader can be stored on the system device by the SAVE command. Or, a previously saved program which has been called in and modified by DDT can be stored in its updated version on the system device, overlaying the old version if desired.

Core images can be saved in units of one or more pages, each page occupying one block on the system device. If a core specification (see below) addresses only a portion of a page, the entire page is written out. For example, the core specification 45-150 is treated as though it were 0-177. Core areas to be saved may be contiguous or noncontiguous as desired by the user. Up to 32_{10} core specifications, in any combination of monotonically increasing single-page or multiple-page requests, can be entered in a single SAVE command.

SAVE Command Format

_.SAVE filename { ! / : } core-specifications, . . .; entry-point

SAVE	Directs Monitor to call in the nonresident SAVE routine.
filename	The filename (program name) to be assigned to the file on the systems device. This name will be used to call the file later when the user wants to read in and execute the program. Restrictions on the formation of filenames can be found in "Command Strings" under "Filenames." Any previously saved program with the same "filename" and having the same extension will be automatically overwritten.
! or :	! is typed immediately after the filename of a file if the user desires to save it as a *system program* (e.g., PIP). A program saved in this manner can be called in by simply typing its name to Monitor (the word CALL is not required) _filename An extension name of .SYS is automatically appended to the filename.

: is typed immediately after the filename of a file if the user desires to save it as a *user program*. A program saved in this manner can be called in and executed later via the CALL command

.CALL filename

An extension name of .USER is automatically appended to the filename.

core-specifications

Up to 32 specifications can be entered in a single SAVE command. Each core specification is separated from the following one by a comma. The last core specification in the series is followed by a semicolon. Addresses are expressed in octal.

Single-page core specification

fnnnn

where

f= field number (can be omitted if field 0).

nnnn= any location within the page which the user desires to save.

Multiple-page core specification

When a user wishes to save a core area of several contiguous pages, he can type a multiple-page core specification in the format.

fnnnn$_1$-nnnn$_2$

where

f=field number (can be omitted if field 0).

nnnn$_1$= any location within the first page of the series of contiguous pages to be saved.

nnnn$_2$=any location within the last page of the series of contiguous pages to be saved.

The following rules apply.

a. The beginning address of a multiple-page request must be smaller than the ending address (nnnn$_1$ must be smaller than nnnn$_2$).

b. Both addresses must be in the same field.

c. The field number (f) must be within the range of your system; however, no check for the validity of this number is performed at SAVE time.

entry-point — The entry point of the saved program, in the format

fnnnn (see explanations above)

An entry point of 0 causes a return to Monitor at CALL time, regardless of the field into which the program was saved.

NOTE
The *last* nonzero field number encountered in a SAVE command string is remembered and prefixed to all other addresses in the command string. (Remember: only one field can be referred to in each command string.)

Example: The following entries are identical in meaning.

SAVE PRGA: 10000-10777, 11400, 1600-17777; 10200

SAVE PRGA: 30000-777, 51400, 26000-7777; 10200

SAVE PRGA: 10000-777,1400, 6000-7777; 200

SAVE PRGA: 0-777, 1400, 6000-7777; 10200

In each of these examples, all addresses are treated as being *field 1*, because the last five-digit entry seen contained a most significant digit 1.

SAVE command strings for system programs are given in Table 7-2.

SAVE Command Processing

A list of the required pages is constructed from the information typed by the user and a block requirement count is kept. When the user types the terminating carriage return (RETURN), allowing the SAVE process to begin, a directory name search on the system device is initiated. If a file having the same name as the filename in the SAVE command is found, it is replaced by the file now being saved. If no such file is found, a new file is created. Next, a storage availability search finds a sufficient number of available blocks on the system device to satisfy the block requirement count. (See above.) These block numbers are stored in a corresponding block list; the blocks are then filled with the contents of the pages to be saved. When the SAVE process is completed, control returns to Monitor (7600).

CALLING A PROGRAM (CALL COMMAND)

Once a file has been loaded and saved, it can be called into core as desired. There are two types of CALL command strings: one for system programs and the other for user programs.

The CALL command string format for system programs (programs saved by a SAVE command string in which the filename was followed by a !) is

.filename

where filename is the same as the one used in the SAVE command string which saved it.

The CALL command string format for user programs (programs saved by a SAVE command string in which the filename was followed by a :) is

.CALL filename

When a program is called, a directory name search is performed on the system device. Associated with the directory entry is the entry point of the program and information concerning file protection and memory extension. If the appropriate directory name entry is found and the file has the proper extension (.SYS or .USER), calling proceeds. If not, the calling process is terminated, ? is typed, and control is returned to Monitor.

Table 7-1. Loading Parameters for System Programs

Name	Core Limits	Entry Point	Pass
PIP	0-5177	1000	1
EDIT	0-3177	2600	1
PALD	0-3377, 3600-4377, 4600, 5200, 6200-6577, 7000-7577	6200	2
FORT	0-1777	200	1
.FT.	200-7377	--	2
STBL	600-777	600	1
FOSL	0-1577	200	1
.OS.	0-5177	--	1
DIAG	200-1177	200	1
.DDT	200-4577	--	1
.SYM	200-4577	--	-
DDT	7200-7577	7200	2

Table 7-2. SAVE Commands for System Programs

Name	SAVE Command String
PIP	SAVE PIP!0-5177;1000
EDIT	SAVE EDIT!0-3177;2600
PALD	SAVE PALD!0-7577;6200
FORT	SAVE FORT!0-1777;200
.FT.	SAVE .FT.!200-7377;0
STBL	SAVE STBL!600;600
FOSL	SAVE FOSL!0-1577;200
.OS.	SAVE .OS.!0-5177;0
DIAG	SAVE DIAG!200-1177;200
.DDT	SAVE .DDT!200-4577;0
.SYM	SAVE .SYM!200-4577;0
DDT	SAVE DDT!7200-7577;0
	(User may assemble anywhere above location 4577)

SYSTEM ERROR MESSAGES

As an input command string is being typed, Monitor recognizes any incorrect syntax and remembers it. When the user types a carriage return, Monitor responds with a ? to indicate invalid input.

Error messages output by Command Decoder are given in Table 7-3.

Table 7-3. System Error Messages

Message	Meaning
?	Illegal syntax or miscellaneous error condition
D	Directory on the systems device is full
E	Too many inputs or outputs were entered
I	No such inputs
S	System I/O failure

Monitor-time read or write errors cause a halt to occur. Persistence of this condition indicates a hardware failure, as the system I/O routine attempts to read or write three times before halting.

I/O PROGRAMMING

The modular concept of input/output (I/O) handling of the disk system provides for easy maintenance and programming. The system device I/O is found in the following places (all I/O routines must be in field 0).

a. Top page of field 0 (location 7642) which is the I/O routine used by all system programs for normal I/O. A copy of this page is on block 0 of the system device. Block 0 of each DECtape is the DECtape I/O routine.

b. Interrupt versions of disk and DECtape routines are found in PIP.

c. Paper tape I/O is handled by individual programs.

The basic I/O routine (see "Normal," below) is called as shown in Table 7-4. It is called in two ways, as determined by bit 2 of the function word.

Normal. The I/O routine returns to JMS +6 (normal) or JMS +5 (error). For example, the following routine would read consecutive blocks from a file on the system device. The routine is initialized by putting the first block number of the desired file into location LINK.

If an attempt is made to read past the last block of the file, an exit will be made to a routine called ENDFIL.

```
GETBLK,  0
         TAD LINK        /GET LINK FROM LAST READ
         SNA             /IS THIS END OF FILE
         JMP ENDFIL      /YES
         DCA BLOK
         JMS 7642        /CALL DISK I/O ROUTINE
         3               /FUNCTION=READ
BLOK,    0
         BUFFAD          /BUFFER ADDRESS
LINK,    0
         JMP ERROR       /ERROR RETURN
         JMP I GETBLK
```

Indirect. The I/O routine returns to the 12-bit address in the error return word (normal or the 12-bit address in ERROR).

Table 7-4 Calling Sequence for Disk System I/O Routine

Calling Sequence	Explanation
JMS I SYSIO	Location SYSIO points to I/O
FUNCT	Function word[1]
BLOCK	Block to be accessed
CORE	Low-order core address
LINK	Filled by READ, used by WRITE
ERROR	Error return here
	Normal return here

[1]Function word:		
	Bits 0-1	unused
	Bit 2	=0, normal return =1, indirect return at end of read/ write to address +1 in error return
	Bits 3-5	unit no. if DECtape
	Bits 6-8	memory field
	Bits 9-11	function: READ = 3; WRITE = 5

Chapter 8

Time-Sharing System

INTRODUCTION

The Time-Sharing System for the PDP-8/I and PDP-8 computers (TSS/8) is a general purpose stand-alone time-sharing system offering each TSS/8 user a comprehensive library of system, service, and utility programs. These programs provide facilities for compiling, assembling, editing, loading, saving, calling, and debugging user programs online. Also included are two conversational language programs, FOCAL-8 and BASIC-8.[1]

By segregating the central processing operations from the time-consuming interactions with the human users, the computer can in effect work on a number of programs simultaneously. Giving only a fraction of a second at a time to each program or task, the computer can deal with many users seemingly at once, as if each user has the computer to himself. The executions of various programs are interspersed without interfering with one another and without detectable delays in the responses to the individual users.

Core and Disk Allocation

TSS/8 requires a minimum of 12K words (three fields) of core memory. The first 8K (Fields 0 and 1) are shared by the various Monitor subprograms. The other 4K (Field 2) and any additional core memory are shared by the users of the system.

[1] BASIC-8 is a modified version of the algebraic language originally developed at Dartmouth College.

The disk area is divided as follows.

Monitor Area	The first 16K of the disk is occupied by the Monitor subprograms. These subprograms are swapped into core memory as needed.
User Swapping Area	This area consists of a 4K track for each user in the system. When a user is temporarily swapped out of core memory to allow other users to have their turn, his program is stored on his 4K track.
File Storage Area	The remaining disk area is used for storing system and user program files.

Monitor Functions

The heart of this time-sharing system is a complex of subprograms called Monitor. Monitor coordinates the operations of the various programs and Teletype consoles, allocates the time and services of the computer to users, and controls their access to the system. The scheduling function includes scheduling the user's requests, transferring control of the central processor from one user to another, moving (swapping) programs between core memory and disk, and managing the user's private files.

TSS/8 User and Console

A TSS/8 user is any person running a computer program within TSS/8. He has an account number and password assigned to him which identifies him to TSS/8. This account number and password are assigned to the user by the person responsible for the system, and it identifies his files on a permanent basis. When the user is using the system, this number also identifies his console, his individual disk swapping track (a contiguous 4K section of disk in the User Swapping Area) for temporary storage of active programs by the Monitor, and whatever other facilities he may be using at any given time.

While the user is logged into the system, he owns at least one Teletype console (see Chapter 4) and one 4K disk track. The console consists of a keyboard, which allows the user to type information to his user programs and to Monitor, a paper tape reader and punch for paper tape input to and output from core, and a teleprinter, which furnishes typed copy of user input and program and Monitor output.

System Program Library

A comprehensive library of programs is available to all TSS/8 users and is located in the File Storage Area of the disk. The library can consist of any or all of the system software which operates in 4K of core memory. Any program in the library can be called into core from the disk and started merely by typing, in response to the dot typed by Monitor, the command R and the assigned name of the program. For example,

.R FOCAL

brings FOCAL into core from the disk and automatically executes it so that it begins typing out its initial dialogue.

User Programs

When a user program is being run by TSS/8 it is swapped into core memory from the user's disk track. Several programs may be run at virtually the same time by employing the technique of bringing a program into core from the disk, allowing it to execute for a short time, marking the state in which its execution is stopped, returning it to the disk, and picking up the next user program.

User programs are serviced regularly on a round-robin basis. After a user program has been executed, it is placed last in the queue of user programs waiting to run. Each program is allowed to run a fixed interval of time and then it is exchanged for the next user program. If only one program is in a condition to run, it is allowed to run without interruption.

User Files

User files are stored in the File Storage Area of the disk. Using the appropriate Monitor commands, the user can create new files and extend, contract, and delete old files. Files may contain textual information, binary core images, or data in any standard format.

Each user can have access to up to four active files at any time. An internal file number (0 through 3) is associated with each of the user's active files, and commands then operate in terms of the internal file numbers. With this feature, a user may attach one of his files to an internal file number and then load service and utility programs that operate on his file without having to explicitly call the file for each service or utility program used.

The user can protect his files against unauthorized access. He can also specify the extent of access certain other users may have to his files. For example, a user's associates may be permitted to look at the data of certain files but not permitted to alter that data.

System Configuration

Depending on the hardware configuration of a particular TSS/8, there can be from 1 to 16 users working with the system simultaneously. Each user has the following resources: at least 4K words of core memory for execution of programs and a corresponding 4K disk track for temporary storage of his core image when swapped out by Monitor.

The minimum equipment requirements of the system are listed below (see Figure 8-1).

PDP-8/I or PDP-8 with KT08/I Time-Sharing Modifications
MC8/I-A Memory Extension Control and 4096 words
MM8/IA 4096 word memory
RF08 Disk Control
RS08 Disk
PT08 Asynchronous Line Interfaces, Dual (4), Real-Time Clock
PT8/I High-Speed Paper Tape Reader
KE8/I Automatic Multiply-Divider
H961A Option Cabinet

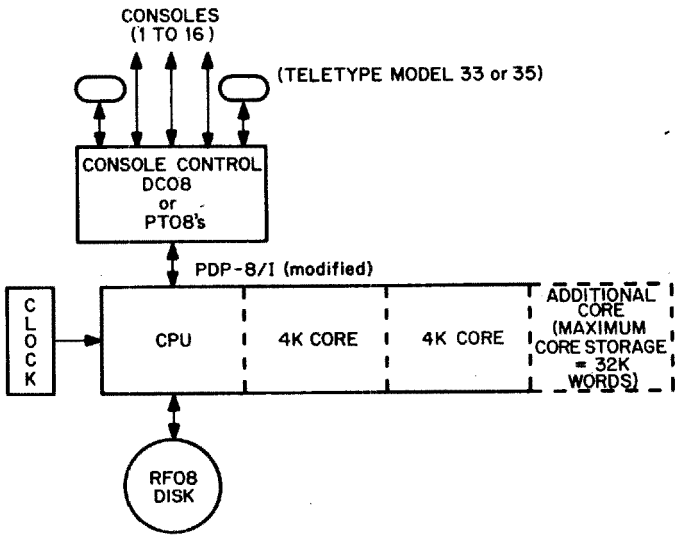


Figure 8-1. System Configuration

The system can have a maximum of 32K of core memory. As additional fields of core memory are added, they permit overlapping the running time of one user program with the swapping time of another, expanding the resident Monitor to buffer a larger number of consoles and reducing the amount of Monitor overlay required, thus increasing operating speed.

With a minimum of 12K of core memory, the following options are available.

1. Up to three DECdisks can be added to increase the number of active users and their file storage.
2. Up to eight DECtapes drives can be added for individual users.
3. Memory Parity
4. Extended Arithmetic Element (PDP-8/I only)
5. Power Failure Protection
6. One High-Speed Paper Tape Punch
7. One DA-10 Interface (PDP-10)
8. One Bit Synchromous Communication Unit (Type 637)
9. One Line Frequency Clock (required with PT08's)
10. Maximum total of 16 active Teletype consoles with appropriate Teletype Controls (PT08's or DC08)

The Data Communication System, Type DC08, is used in full duplex mode. It consists of a Data Line Interface Unit (DL8/I), a Serial Line Multiplexer Unit (685), and other devices connected to form a data link and message switching system between the user consoles and the central processor.

THE TSS/8 MONITOR

The TSS/8 Monitor is composed of the following routines.

Scheduler	Error Message Handler
System Interpreter	Buffer Handler
IOT Trap Handler	DC08 Service
Storage Allocator	Disk Service
Overlay Control	Optional Device Service
File Control	

With the above routines, Monitor provides services which can be divided into three broad categories: *device service, scheduling and running,* and *communication.*

On an interrupt basis (program interrupt—see Chapter 5) the device service routines receive information from all input devices, distribute that information out to the appropriate buffers, and inform the activation routine when a user program must be activated to receive its information. The device service routines also accept output from user programs, buffer it, and send it to output devices whenever the devices are able to receive it.

Scheduling is handled by a round-robin scheduling algorithm with the exception that programs with disk requests pending are run out of turn to optimize disk usage. The scheduling routine decides whether to remove the current program from core, and if so, which program is to be run next.

A user program is, at any point in time, in one of the following states.

Running. The user program is in execution. It continues execution until its quantum has expired or until it issues an I/O request that cannot be satisfied immediately.

Active. The user program is ready to run and will be swapped into core when its turn comes. A program can become active when

a. An output buffer is almost empty, thus assuring continuous output of information,
b. An input buffer is almost full,
c. Input requested by a user has arrived,
d. A user determined activation condition has been satisfied, or
e. The user commands the system to begin execution.

Waiting. In this state the program would be active except that it is waiting for the completion of some I/O request or special condition.

Dormant. The program is not being entered into the round-robin. The user may be communicating with Monitor or the program may be dismissed for a variety of Monitor-determined-reasons (e.g., illegal op code).

A program may be swapped out of core memory for any of the following reasons.

1. The quantum of time has expired. The program moves from the running state to the active state.

2. The program has requested that the quantum of time be terminated. The program moves from the running state to the active state.
3. The program has filled an output buffer, or has requested input, but the input buffer is empty, or has requested a special dismissal condition. The program moves from the running to the waiting state.
4. The program has tried to execute an illegal instruction. The program moves from the running state to the dormant state.

System Interpreter

Monitor checks all incoming characters for the *call* (CTRL/B) character. When the call character is encountered, Monitor routes all subsequent characters up to and including the first carriage return (RETURN key) to its System Interpreter's input buffer. Monitor's System Interpreter performs the following services.

1. Verifies the user's account number and password when he logs in, and provides him with a disk track to store user programs. It releases facilities owned by the user when he logs out.
2. Parcels out extra consoles and input/output devices.
3. Provides commands for creating, opening, and maintaining the user's files.
4. Allows the user to save all or part of his binary user program for future use and reference, and restores it with its state unchanged.
5. Provides accounting of console time and user program run time.
6. Provides the user with information about the state of his program while it is running, as well as information about the state of the character control tables.
7. Provides commands for calling the various utility programs.

Communications to the System Interpreter are automatically duplexed. That is, all characters from a console keyboard to the System Interpreter appear on that console's printer, regardless of the setting of the character control tables (explained later).

The System Interpreter is kept quite busy receiving messages from many keyboards and user programs, therefore, it must be run often. Consequently, it occupies a priority position in the round-robin, being scheduled whenever there are characters in its input buffer.

Phantom Routines

Monitor has two phantom routines: error and file control. They are not phantoms as defined by Webster but they appear to act as though they were. These routines reside on the disk, "jump" into core when needed, perform their assigned tasks, and "fade" back onto the disk.

Phantoms are privileged routines which run in place of a regular user program in the round-robin. The time the phantom takes to perform its service is charged to that user program upon which it is servicing. For example, the error phantom prints all error messages for running user programs, and printing a lengthy error message may require several intervals of time to which the program in error will be charged. The error phantom routine replaces the offending program in the round-robin and prints an error message, thus punishing the user responsible for the error and no one else.

The file control phantom routine handles all modifications to the user's files, such as creating, lengthening, renaming, and destroying files. This phantom routine is brought into operation when a user program executes certain input/output transfer (IOT) instructions or when Monitor is acting for the user program.

Character and Data Flow

When a user logs into the system his account number and password are associated with a job number, and that job number is then associated with a disk swapping track and the console(s) he owns. The account number establishes ownership of the input buffer attached to the user program and the output buffer(s) attached to the printer(s) of his console(s).

Monitor provides for communication among the users, user programs, and Monitor. Data flow is illustrated in Figure 8-2. Communication is provided through the IOT instructions. When a program executes IOT instructions, Monitor picks up locations in the user program as parameters to service routines. These routines may simulate I/O to or from the online device, control or release ownership of devices, handle character transmission, or return information to the user program by filling core locations within the program. Through IOT instructions the user program can make its wants known and Monitor can inform the user program of any unusual conditions in the system.

Characters are generated by users typing at keyboards or by user programs typing out. Characters go into input buffers to be read by user programs or to output buffers to be printed on console printers. The character control tables (Figure 8-2) set up useful character transmission paths, whereby any character source (keyboard and program)

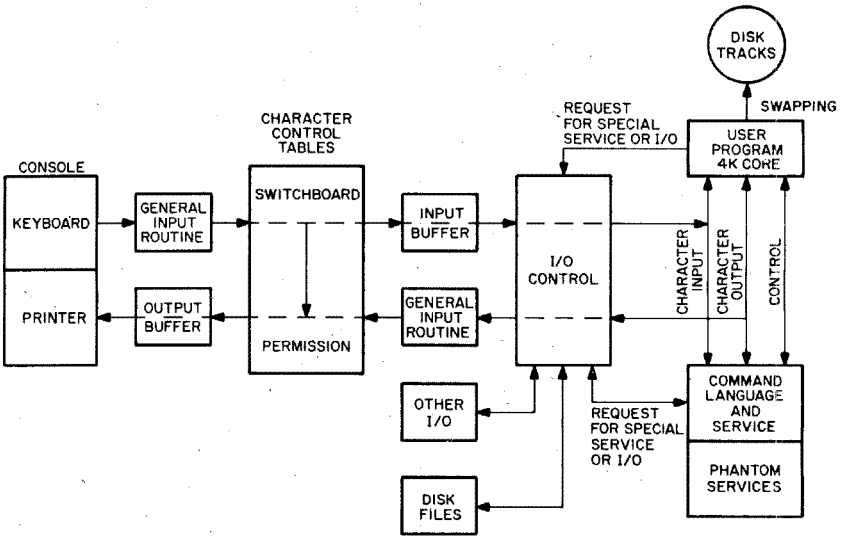


Figure 8-2. Character and Data Flow

may, with permission, send to any character buffer. To insure that unwanted connections are not made, using the appropriate command to Monitor the user owning any character buffer may grant or deny permission for connection into that buffer.

The character control tables are a permission table and a switchboard table. The permission table allows a user to control which character sources may place characters in which of his buffers, and the switchboard table controls the actual routing of these characters. Characters originating from any keyboard or from any user program typing out can, with permission, be placed in any or all of the keyboard input buffers or any or all of the printer output buffers. The user, using appropriate Monitor commands, establishes the actual routing of characters to and from his associated buffers.

The character control feature finds applications in:

1. Duplexing—Characters typed at a keyboard can be printed on other consoles without user program intervention.
2. Interprogram Communication—User programs can communicate with each other and with Monitor. Thus, several user programs may run as one system coordinating their separate tasks through character communication.

3. Interconsole Communication—Users at consoles can set up general links for conferences, teaching, monitoring, or for any number of reasons.
4. Multiple Consoles—User programs may receive characters from and send characters to more than one console. This allows a user program to act as a time-sharing subsystem within TSS/8, controlling its own set of consoles, e.g., as in teaching machine monitors.

MONITOR COMMANDS

A Monitor command is a string of characters terminated by a semicolon (;) or a carriage return (RETURN key). Commands to Monitor are typed on the console keyboard by the user or output by the user program, with each command beginning with a command name. In some cases, the command name is the entire command, in which case it is followed directly by a terminator. In other cases, the command name is followed by a space, one or more parameters, and then a terminator (see example below).

.TIME	(Requests the time of day. This command was terminated by the RETURN key)
.TIME c_1	(Requests the processing time used by job c_1. Terminated by the RETURN key)

Only enough characters need be typed in the command name to uniquely identify the command name as shown in the following example.

LOGI	for LOGIN
LOGO	for LOGOUT

More than one command may be typed on a line, with all but the last command being terminated by a semicolon; the last command is terminated by the RETURN key (see example below). Commands are executed only when the RETURN key is typed, which explains why the last command on a line must be terminated by the RETURN key.

.OPEN c_1 s_1 c_2; LOAD c_1 s_1; START c_1

As shown in the above examples, each command or a line of commands is typed after the dot typed by Monitor. The dot is typed by Monitor when it is ready and available to accept commands from the user.

Parameters may be typed as octal numbers, decimal numbers, character strings, or single letters. In the following descriptions of Monitor commands the parameters are coded as follows.

c_1, c_2 . . .	represent octal numbers
d_1, d_2, . . .	represent decimal numbers
s_1, s_2, . . .	represent character strings
l_1, l_2, . . .	represent single letters

Logging In and Out

Logging in and out of TSS/8 is a function performed by Monitor's System Interpreter routine. When a console is in the free state, a prospective user may attempt to log into the system.

LOGIN c_1 s_1;

This is a request by a user to enter into the system. If the console from which the command is typed is free, there is an available disk track, and the two parameters following the command LOGIN form a legitimate account number and password, then the user will be logged into the system.

c_1 is the user's account number.
s_2 is the user's password.

At the time of login, the switchboard table is initialized to the normal operation setting. However, Monitor diverts all characters typed to the System Interpreter until the user gives a command that indicates otherwise.

LOGOUT;

This is a request by a user to leave the system. It disconnects all consoles and temporary disk track that he owns, places his programs in the free state, and resets the switchboard table. It also writes an account record on the disk showing how much computing time (processing time) and console time was used by the user's program(s).

TIME c_1;

This is a request for Monitor to type out the computing (processing) time used since login. If job c_1 is not specified, the job owning the console is assumed. If requested before login and if no job is specified, the time-of-day is typed. If, at any time, job 0 is specified the time-of-day is typed.

Console Manipulation

ASSIGN K c_i;

This is a request for console number c_i to be added to those consoles already owned by the requesting user.

K denotes console
c_i is the console number

If the console is free, it will be given to the user and the user's switchboard table settings will be augmented by those additional entries which put the new console in the normal state with respect to its own printer and the program on the user's disk track. Cross settings for interconsole connections are left to the user's discretion.

SLAVE c_i;

This command causes console number c_i to be slaved. This means that console number c_i will be unable to communicate with Monitor, that the system will ignore the call (CTRL/B) character from that console. However, the console may serve as an input/output device for the user's program. Monitor checks to make sure that the requesting user owns the console he wishes to slave. It is illegal to slave every console that a user owns.

UNSLAVE c_i;

This command restores a slaved console to normal status. After this command, console number c_i will be able to communicate with Monitor. The user must own the console to unslave it.

RELEASE K c_i;

This command releases (deassigns) console number c_i and puts it in the free state. Monitor checks to make sure that the requesting user owns the console he wishes to release. When the console is released, the switchboard table is reset and the input and output buffers are cleared.

Device Allocation

ASSIGN l_i;

This is a request for access to the device specified by l_i.

l_i = R for high - speed paper tape reader
P for high - speed paper tape punch
C for card reader
L for line printer
I for incremental plotter

If device l_i is not busy, it will be allocated to the user program issuing the command. If device l_i is busy, the job number of the user having possession is returned. A RELEASE l_i; or LOGOUT; will release the device.

ASSIGN D c_1;

This command is the same as ASSIGN l_1;

D denotes DECtape unit

c_1 is the unit number of the DECtape unit

RELEASE l_1;

RELEASE K c_1;

RELEASE D c_1;

Each of these commands will annul the current assignment of the specified device.

File Control

OPEN c_1 s_1 c_2;

This command establishes association between an internal file number and a file. After this command is given, the file of account c_2 with the name s_1 is associated with internal file number c_1. The internal file number specified must be between zero and three inclusive. If c_2 is not specified, the account number of the current user is assumed.

CLOSE c_1 c_2 . . . ;

This command closes the files specified by c_1 c_2 . . .

c_1 c_2 . . . is a list of internal file numbers separated by spaces

After this command is given, no writing can be done on the files specified, and the associations between the internal file numbers and the files are broken.

CREATE s_1;

This command causes Monitor to create a new file which is to have the name s_1, if there is available space in the File Storage Area of disk. At the time of creation, Monitor will enter the name of the new file and the date of creation into the owner's file directory. Example:

CREATE NEWF;	Asks Monitor to create a new file named NEWF. If there is no more space in File Storage, Monitor will so inform the user.

RENAME c_1 s_1;

This command renames a file. The file to be renamed must be already open and associated with internal file number c_1. Its new name will become s_1. Example:

RENAME 1 MSTR;	Assume that file NEWF has been opened to internal file number 1; this command will change the name of that file from NEWF to MSTR.

REDUCE c_1 d_1;

This command reduces the length of a file. The file which is to be shortened must be open and associated with internal file number c_1.

d_1 is the number of segments to be removed from the end of the file

If d_1 is greater than or equal to the number of segments in the file, the file is deleted from the directory. Example:

REDUCE 2 2;	Assume file TT13 is opened to internal file number 2; this command then removes 2 segments from its end and returns those segments to free storage.

EXTEND c_1 d_1;

This command extends the length of a file. The file which is to be lengthened must be open and associated with internal file number c_1.

d_1 is the number of segments to be added to the end of the file

If there is free storage space available, Monitor will lengthen the file as requested.

PROTECT c_1 c_2;

This command changes the file protection mask of a file. The file to be protected must be open and associated with internal file number c_1.

c_2 is the new file protection mask

For file protection, the 12-bit account number is partitioned into a project number (high order 7 bits) and a programmer number (low order 5 bits).

File protection masks (c_2) are assigned as follows:

- $c_2 = 1$ read protect against users whose project number differs from owner's.
- $c_2 = 2$ write protect against users whose project numbed differs from owner's.
- $c_2 = 4$ read protect against users whose project number is same as owner's.
- $c_2 = 10$ write protect against users whose project number is same as owner's.
- $c_2 = 20$ write protect against owner.

c_2 can be the sum of any of the above values.

The protect command is illegal for all users except the owner. Example:

PROTECT 1 3;	Read and write protect internal file 1 against access by any user whose project number differs from the owner's.

F c_1;

This command causes Monitor to print out the current state of the association of the user's internal file number c_1 with the file. The response format is c_2 s_1 c_3 d_1
where

c_2 is the owner's account number.
s_1 is the filename.
c_3 is the protection mask.
d_1 is the number of segments.

Control of User Programs

START c_1;

This command begins execution of a user program at location c_1. In addition, the command resets the switchboard table so that all characters typed from either a keyboard or program directed into the user program's input buffer are no longer intercepted by the System Interpreter.

START;

This command restarts a user program. If Monitor has been called during the execution of the user program, the complete state of the program is saved including the location of the next instruction to be executed. When the START command is given, the program's state is restored and the program continues execution where it left off. As in the START c_1; command, characters intended for the user program's input buffer are no longer intercepted by the System Interpreter.

DEPOSIT c_1 c_2 . . . c_n;

This command stores c_2 in location c_1; c_3 in location $c_1 + 1$; . . .;c_n in location $c_1 + n - 1$.

n is equal to or less than 10 (decimal).

A user can load a binary user program using the DEPOSIT command, although it is much easier using DDT-8. This command is useful when making small patches to stored programs. The address c_1 must be an absolute address within the user's 4K core area.

EXAMINE c_1 d_1;

This command causes Monitor to type the contents of the d_1 locations starting at location c_1.

d_1 is equal to or less than 10 (decimal).

If d_1 is not specified, the contents of location c_1 is typed. The address c_1 must be an absolute address within the user's 4K core area.

Saving and Restoring Binary User Programs. The SAVE, LOAD, R, and RUN commands leave file s_1 open and assign it internal file number 3, and turn the user's interrupt system off.

SAVE c_1 s_1;

SAVE c_1 s_1 c_2;

SAVE c_1 s_1 c_2 c_3;

SAVE c_1 s_1 c_2 c_3 c_4;

These commands write portions of the user's core image onto a file whose owner's account number is c_1, filename is s_1 (s_1 must have been previously CREATEd), and

- c_2 is the file address of the first word to be written; if not specified, the entire 4K is written on the first 4096 words of the file.
- c_3 is the core address of the first word to be written; if not specified, all 4096 words are written.
- c_4 is the core address of the last word to be written; if not specified, 7777 is assumed.

If c_1 is not specified, the account number of the current user is assumed.

Example:

```
SAVE NEWF;          Writes core words 0 through 7777 or words 0
                    through 7777 of file NEWF.
```

LOAD c_1 s_1;

LOAD c_1 s_1 c_2;

LOAD c_1 s_1 c_2 c_3;

LOAD c_1 s_1 c_2 c_3 c_4;

These commands read certain portions of the file whose owner's account number is c_1 and whose filename is s_1, into core. c_1 may be omitted if it is the same as the account number under which the user is logged in.

- c_2 is the file address of the first word to be read; if not specified, words 0 through 7777 are read into words 0 through 7777.
- c_3 is the core address of the first word to be loaded; if not specified, all words 0 through 7777 are loaded.
- c_4 is the core address of the last word to be loaded; if not specified, 7777 is assumed.

Example:

```
LOAD NEWF 5 10 17;  Loads words 5 through 14 into words 10
                    through 17 respectively.
```

R s_1;

This command is equivalent to

OPEN 3 s_1 2; LOAD 2 s_1; START 0

which loads program s_1 from the system library (account 2) and starts the program running.

RUN s_1;

This command is equivalent to

OPEN 3 s_1; LOAD s_1; START 0

RUN c_1 s_1;

This command is equivalent to

OPEN 3 s_1 c_1; LOAD c_1 s_1; START 0

S;

This command stops the execution of the user program, saves its complete state, and sets the switchboard table so that all characters directed to the program's input buffer will be intercepted by System Interpreter.

WHERE;

This command causes Monitor to type out the current state of a user program's accumulator, link, program counter, and switch register, multiplier quotient, and step counter.

USER;

This command causes Monitor to type out the number of the job and devices owned by the user.

USER c_1;

This command causes Monitor to type out the numbers of the devices owned by user c_1. If job 0 is specified, the numbers of unassigned devices are typed.

SWITCH c_1;

This command sets the user's switch register to c_1.

BREAK c_1;

This command sets the user's keyboard delimiter mask to the value c_1.

Permission and Switchboard Tables

Using Monitor commands the user may set, reset, and read any given bit in either the permission table or the switchboard table.

SET l_1 l_2 c_1 l_3 c_2

This command sets a bit in either the permission or switchboard table to a 1.

l_1 = P denotes permission table
S denotes switchboard table

l_2 = K denotes keyboard
P denotes user program

c_1 is the octal number of the keyboard or program

l_3 = I denotes input buffer
O denotes output buffer

c_2 is the octal number of the buffer which is to receive the characters from the character source

Monitor checks for appropriate ownership to decide whether it will allow the connection to be made. The ownership of the character source is used to determine legality in setting the switchboard table while the ownership of the character sink is used in setting the permission table. Examples:

SET P K 13 O 12;	Requests that keyboard number 13 be given permission to write on the printer (output buffer) of console 12.
SET S P 4 I 16;	Informs Mnoitor that job number 4 would like to type characters into the input buffer of user program 16. This request will be granted only if user 16 has previously given permission by SET P P 4 I 16; or by the equivalent IOT instruction (SSP).

RESET l_1 l_2 c_1 l_3 c_2;

This command is identical in all respects to the SET command above except that the bit indicated by the parameters will be cleared to zero.

READ l_1 l_2 c_1 l_3 c_2;

This command uses the same parameters to specify a bit in either the permission or switchboard table as in the SET and RESET commands above. Monitor will type out the value of the bit.

DUPLEX;

This command is a shorthand command to set the switchboard table so that characters typed on the keyboard from which this command is issued will appear on the printer of that console.

Example:

DUPLEX;	If the keyboard issuing this command is number 16, the command is equivalent to RESET S K 16 O 16;

UNDUPLEX;

This command is a shorthand command to undo the effects of the DUPLEX command.

ALLOW c_1;

This command is shorthand to indicate that a user is giving permission for keyboard number c_1 to place characters in the output buffer of his console.

Example:

ALLOW 4;	Assume that this command was issued from keyboard number 7, then it is equivalent to SET P K 4 O 7;

LINK c_1;

This command is given by a user who wishes to communicate with a console other than the one at which he is sitting. This command

is legal only if the owner of that console has set the permission table so that he will accept characters from the requesting console into his printer's output buffer. The command is shorthand for the command ALLOW c_1; followed by the command to set the switchboard table so that characters from the requesting console will be placed in output buffer number c_1. Example:

LINK 16; — Assume that the console which issued this command was number 7, then this command would be legal only if the owner of console number 16 had previously set the permission table to allow keyboard 7 access to that output buffer. He could have done this with ALLOW 7; or by SET P K O 16; or by an equivalent IOT instruction executed by his program. If that permission has been granted, the LINK command is equivalent to the following two switchboard commands: SET P K 16 O 7; (ALLOW 16) and SET S K 7 O 16;

Inter-System Communication

In those TSS/8 systems having a local connection to a PDP-10 Time-Sharing System or a Synchronous Data Communication System (Type 637), the input to the user's input buffer and program output is scanned for the character sequences CTRL/B CTRL/X and CTRL/B CTRL/Y, respectively. All characters up to the next CTRL/B are diverted to the PDP-10 or 637 System, whichever the case may be. Characters from the PDP-10 and 637 System are directed into the user's input buffer.

INPUT/OUTPUT TRANSFER INSTRUCTIONS

Whenever a user program executes an input/output transfer (IOT) instruction (an instruction of the form 6XXX) the system traps the instruction and transfers control to a system service or simulation routine. These routines accomplish special tasks, set up parameters for the system, and perform input/output for the user program.

IOT instructions may be separated by function into three types:

1. Input/output instructions available on the standard PDP-8/I without a Time-Sharing Monitor—When a user program executes one of these IOTs, TSS/8 simulates an input/output function similar to the function of the IOT instruction on the standard PDP-8/I. Some standard PDP-8/I IOT instructions are illegal in TSS/8 (see Appendix C of *Time-Sharing System —TSS/8 Monitor,* Order No. DEC-T8-MRFB-D).

2. IOT instructions to request input/output service from TSS/8 unavailable on the standard PDP-8/I—These include requests for DECdisk, DECtape, high-speed paper tape reader and punch, card reader, and console character handling.

3. IOT instructions which call subroutines to set user parameters or to alter the time-sharing environment for a particular user program—An alteration may, for example, include requests to add or release facilities or to change mode of character handling.

An IOT instruction usually acts as a subroutine call. Therefore, depending on the specific IOT instruction, parameters may be loaded into the accumulator (AC) before execution of the IOT. In some cases, the parameter in the AC acts as a pointer to a parameter block in the user's program. If the system has to return information to the user's program, it returns that information in the AC or in a block of locations in the user's program (the beginning address of this block is in the AC).

In general, nearly all TSS/8 console commands have corresponding IOTs which allow the user the same features under program control which he has with the console commands.

ERROR MESSAGES

User Program

Error messages are typed on the user's printer by the error phantom routine when error conditions occur in a running user program. If the user program is not enabled for system error interrupts, the error messages are typed in the following format.

s_1 FOR USER c_1.
AC = c_2, L = c_3, PC = c_4
INSTR = c_5

s_1 is a string describing the nature of the illegal instruction c_5 that user program c_1 has executed at location c_4. At the time the illegal instruction was executed the value of the accumulator was c_2 and the value of the link was c_3. For example, an error message might appear as follows:

ILLEGAL IOT FOR USER 2 AC = 1520, L = 1, PC = 3552 INSTR = 6025	The user program has executed an IOT which the system regards as illegal. The illegality may be for one of two reasons: 1. The IOT itself may be illegal. 2. The parameters of a legal IOT may invalidate it.

System Interpreter

The following error messages result from illegal requests to the System Interpreter. They are printed by the error phantom on the console.

s_1 ?	The System Interpreter does not understand the command. s_1 = command
ILLEGAL REQUEST	The user has requested an illegal service. This error usually results when some parameter has been given an incorrect value or the request refers to a facility not owned by the user.
SWITCHBOARD ERROR	The user has attempted to set a bit in the permission table not owned by him; or the user has attempted to make a connection in the switchboard table for which permission has not been granted.
CONSOLE IN USE LOGOUT TO RELEASE	The user has tried to log in on a console which is already in use; or he has attempted to add a console to those he already owns.
LOGIN PLEASE	The user has attempted to use a console which is not logged in.
UNAUTHORIZED ACCOUNT	The user has attempted to log into the system with an invalid account number or name.
FULL	This message may appear on an attempt to log into TSS/8 from a console. It means that all disk swapping tracks are in use.
c_1 HAS IT	Job number c_1 has the request device; the request for its acquisition cannot be granted.
FAILED BY d_1	The user has attempted to extend a file beyond the available disk storage. d_1 segments of the user's request could not be added.

FILE NOT OPEN	The user has attempted to manipulate a file which is not associated with an internal file number.
DIRECTORY FULL	The user's file directory has no room for new file names.
PROTECTION VIOLATION	The user has attempted to use a file in a manner contrary to the file protection code specified for that file.
FILE NOT FOUND	The system could not find the desired file in the specified directory.
DISK FULL	The user had attempted to create a new file when there is no room on the disk for it.

CONCLUSION

The above is merely an overview of TSS/8. There are many other aspects not mentioned here. For a thorough coverage, see *Time-Sharing System—TSS/8 Monitor,* Order No. DEC-T8-MRFB-D.

Chapter 9

FOCAL Programming

FOCAL (FOrmula CALculator) is an online, conversational, interpretive language for the PDP-8 family of computers. It is designed to help students, engineers, and scientists solve numerical problems. The language consists of short, easy-to-learn, imperative English statements. Mathematical expressions are typed in standard notation. The best way to learn the FOCAL language is to sit at the Teletype console and try the commands, starting with the examples given in this chapter.

FOCAL puts the full calculating power and speed of the computer at your fingertips. FOCAL is an easy way of simulating mathematical models, plotting curves, handling sets of simultaneous equations in n-dimensional arrays, and much more. A few of the many kinds of problems that have been solved by FOCAL are described under "Examples of FOCAL Programs." The user can become acquainted with many applications of FOCAL by duplicating the example programs using different variables.

This chapter describes the features of FOCAL, 8/68, which is issued on tape DEC-08-AJAC-PB.

EQUIPMENT REQUIREMENTS

FOCAL operates on a 4K PDP-8 family computer with an ASR 33 Teletype, and with or without a high-speed reader/punch, analog-to-digital converter (189), oscilloscope display (34D), or any other DEC peripherals with the appropriate program overlays available from DEC.

GETTING ONLINE WITH FOCAL

The FOCAL program is furnished to the user on punched paper tape in binary-coded format. Therefore, it is loaded into core using the Binary Loader program, as described in Chapter 6.

The Initial Dialogue

After FOCAL has been loaded and started, it begins typing out its initial dialogue, giving the user the options of retaining certain groups of mathematical functions. If these functions are not needed, the user answers FOCAL's questions by typing NO and the RETURN key, and FOCAL erases those functions from core, thus the user gains additional core storage for use by his programs.

Samples of the initial dialogue are shown below.

```
CONGRATULATIONS!!
YOU HAVE SUCCESSFULLY LOADED 'FOCAL' ON A PDP-8/I  COMPUTER.

SHALL I RETAIN LOG, EXP, ATN ?:YES

PROCEED.
*
```

With the above response, all mathematical functions are retained, and the user has about 720 locations available for his programs.

```
CONGRATULATIONS!!
YOU HAVE SUCCESSFULLY LOADED 'FOCAL' ON A PDP-8/I  COMPUTER.

SHALL I RETAIN LOG, EXP, ATN ?:NO
SHALL I RETAIN SINE, COSINE ? :YES

PROCEED.
*
```

When the user answers NO to the first question FOCAL asks a second question. The above response leaves about 975 locations available for the user's programs.

```
CONGRATULATIONS!!
YOU HAVE SUCCESSFULLY LOADED 'FOCAL' ON A PDP-8/I  COMPUTER.

SHALL I RETAIN LOG, EXP, ATN ?:NO
SHALL I RETAIN SINE, COSINE ? :NO

PROCEED.
*
```

The above response erases all mathematical functions from core, giving the user about 1105 locations for use by his programs.

A simple FOCAL program which determines the number of core locations available for the user's programs and a formula for calculating the length of a user program are given under "Estimating the Length of a User's Program."

In the second line of the initial dialogue, FOCAL identifies the type of computer being used—PDP-8/I, PDP-8/L, PDP-8, etc. FOCAL concludes the initial dialogue by telling the user to PROCEED, followed by an *, and waits for user input.

THE FOCAL LANGUAGE

When the initial dialogue is concluded, FOCAL types

```
*
```

indicating that the program is ready to accept commands from the user. Each time the user completes typing a Teletype line and terminates it by depressing the RETURN key or after FOCAL has performed a command, an asterisk is typed to tell the user that FOCAL is ready for another command.

Simple Commands

One of the most useful commands in the FOCAL language is TYPE. To FOCAL this means "type out the result of the following expression." When you type (following the asterisk which FOCAL typed),

```
*TYPE 6.4318+8.1346
```

and then press the RETURN key, FOCAL types

```
=   14.5664*
```

Another useful command is SET, which tells FOCAL "store this symbol and its numerical value. When I use this symbol in an expression, insert the numerical value." Thus, the user may type,

```
*SET A=3.14159; SET B=428.77;  SET C=2.71828
*
```

The user may now use these symbols to identify the values defined in the SET command. Symbols may consist of one or two alphanumeric characters. The first character must be a letter, but must *not* be the letter F.

```
*TYPE A+B+C
=   434.6300*
```

Both the TYPE and SET commands will be explained more fully in their respective sections of this chapter.

FOCAL is always checking user input for invalid commands, illegal formats, and many other kinds of errors, and types an error message indicating the type of error detected. In the example,

```
*HELP
?03.31
*TYPE 2++4
?07.<0
*
```

HELP is not a valid command and two plus signs (double operators)

are illegal. The complete list of error messages and their meanings is given under "Error Diagnostics."

Output Format

FOCAL is originally set to produce results showing up to eight digits, four to the left of the decimal point (the integer part) and four to the right of the decimal point (the fractional part). Leading zeros are suppressed, and spaces are shown instead. Trailing zeros are included in the output, as shown in the examples below.

```
*SET A=77.77; SET B=1111.1111;  SET C=39
*TYPE A,B,C
=  77.7700=  1111.1100=  39.0000*
```

The results are calculated to *six significant digits*. Even though a result may show more than six digits, *only* six are significant, as shown above in SET B = 1111.1111, which FOCAL typed as = 1111.1100.

The output format may be changed if the user types

```
TYPE %x.yz,
```

where x is the total number of digits to be output and yz (always two digits, i.e., 01, 08, 12, etc.) is the number of digits to the right of the decimal point. x and yz are positive integers, and the value of x cannot exceed 19. When first loaded, FOCAL is set to produce output having eight digits, with four of these to the right of the decimal point (%8.04). For example, if the desired output format is mm.nn, the user may type

```
*TYPE %4.02, 12.22+2.37
```

and FOCAL will type

```
=  14.59*
```

Notice that the format operator (%x.yz,) must be followed by a comma.

In the following examples, the number 67823 is typed out in several different formats.

```
*SET A=67823
*TYPE %6.01, A
=  67823.0*
*TYPE %5, A
=  67823*
*TYPE %8.03, A
=  67823.000*
```

If the specified output format is too small to contain the number, FOCAL automatically prints the number in floating-point format, as explained below. If the specified format is larger than the number, FOCAL inserts leading spaces:

```
*TYPE %7, 67823
=  67823*
```

Leading blanks and zeros in integers are always ignored by FOCAL.

```
*TYPE %8.04, 0016, 0.016, ., 007
=  16.0000=  0.0160=  0.0000=  7.0000*
```

Floating-Point Format

To handle much larger and much smaller numbers, the user may request output in exponential form, which is called floating-point or E format. This notation is frequently used in scientific computations, and is the format in which FOCAL performs its internal computations. The user requests floating-point format by including a % followed by a comma, in a TYPE command. From that point on, until the user again changes the output format, results will be typed out in floating-point format.

```
*TYPE %, 11
=  0.110000E+02*
```

This is interpreted as .11 times 10^2, or simply 11. Exponents can be used to ± 616. The largest number that FOCAL can handle is $+0.999999$ times 10^{616}, and the smallest is -0.999999 times 10^{-616}.

To demonstrate FOCAL's power to compute large numbers, you can find the value of 300 factorial by typing the following commands. (The FOR statement, which will be explained later, is used to set I equal to each integer from 1 to 300.)

```
*SET A=1
*FOR I=1,300; SET A=A*I
*TYPE %, A                    (wait for FOCAL to
=  0.306051E+615*             calculate the value)
```

Arithmetic Operations

FOCAL performs the usual arithmetic operations of addition, subtraction, multiplication, division, and exponentiation. These are written by using the following symbols:

Symbol	Math Notation	FOCAL	
↑Exponentiation	3^3	3↑3	(Power must be a positive integer)
*Multiplication	$3 \cdot 3$	3*3	
/Division	$3 \div 3$	3/3	
+Addition	3+3	3+3	
−Subtraction	3−3	3−3	

These operations may be combined into expressions. When FOCAL evaluates an expression, which may include several arithmetic operations, the order of precedence is the same as that in the list above. That is, exponentiation is done first, followed by multiplication, division, addition and subtraction. Addition and subtraction have equal priority. Expressions with the same precedence are evaluated from the left to right.

A+B*C+D is A+(B*C)+D not (A+B)*(C+D) nor (A+B)*C+D
A*B+C*D is (A*B)+(C*D) not A*(B+C)*D nor (A*B+C)*D
X/2*Y is $\frac{X}{2Y}$
2↑2↑3 is 4^3 not 2^8

Expressions are combinations of arithmetic operations or functions which may be reduced by FOCAL to a single number. Expressions may be enclosed in properly paired parentheses, square brackets, and angle brackets (use the enclosures of your choice for clarity; FOCAL is impartial and treats them all in the same way).
For example,

```
*SET A1=(A+B)*<C+D>*[E+G]
*
```

The [and] enclosures are typed using SHIFT/K and SHIFT/M, respectively.

Expressions may be nested. FOCAL computes the value of nested expressions by doing the *innermost* first and then working outward.

```
*TYPE %, [2+(3-<1*1>+5)+2]
=  0.110000E+02*
```

Note that the result is typed in floating-point format.

More About Symbols

The value of a symbolic name or identifier is not changed until the expression to the right of the equal sign is evaluated by FOCAL. There-

fore, the value of a symbolic name or identifier can be changed by redefining it in terms of itself (i.e., in terms of its current value).

```
*SET A1=3↑2
*SET A1=A1+1
*TYPE %2, A1
=  10*
```

NOTE
Symbolic names or identifiers must *not* begin with the letter F.

The user may request FOCAL to type out the values of all of the user-defined identifiers, in the order of definition, by typing a dollar sign ($).

```
*TYPE %6.05, $
```

The user's symbol table is typed out like this

```
A@(00)= 0.306051E+615
B@(00)= 1111.11
C@(00)= 39.0000
I@(00)= 301.000
A1(00)= 10.0000
D@(00)= 0.00000
E@(00)= 0.00000
G@(00)= 0.00000
*
```

If an identifier consists of only one letter, an @ is inserted as a second character in the symbol table printout, as shown in the example above. An identifier may be longer than two characters, but only the first two will be recognized by FOCAL and thus stored in the symbol table.

Subscripted Variables

FOCAL always allows identifiers, or variable symbols, to be further identified by subscripts (range ±2047) which are enclosed in parentheses immediately following the identifier. A subscript may also be an expression:

```
*SET A1(I+3*J)=2.71; SET X1(5+3*J)=2.79
```

The ability of FOCAL to compute subscripts is especially useful in generating arrays for complex programming problems. A convenient way to generate linear subscripts is shown under "Simultaneous Equations and Matrices."

The ERASE Command

It is useful at times to delete all of the symbolic names which you have defined in the symbol table. This is done by typing a single com-

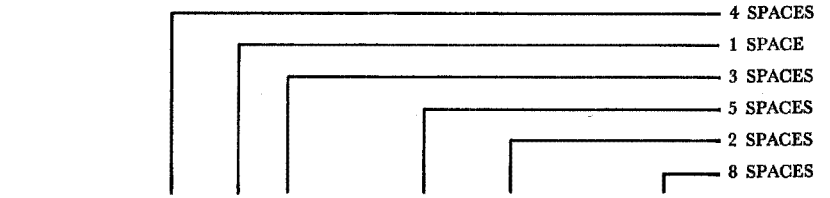
mand: ERASE. Since FOCAL does not clear the user's symbol table area in core memory when it is first loaded, it is good programming practice to type an ERASE command before defining any symbols.

Handling Text Output

Text strings are enclosed in quotation marks (". . .") and may include most Teletype printing characters and spaces. The carriage return, line feed, and leader-trailer characters are not allowed in text strings. In order to have FOCAL type an automatic carriage return/line feed at the end of a text string, the user inserts an exclamation mark(!).

```
*TYPE "ALPHA"!"BETA"!"DELTA"!
ALPHA
BETA
DELTA
*
```

To get a carriage return without a line feed at the end of a text typeout, the user inserts a number sign (#) as shown below.




```
*TYPE !"        X Y    Z"#"       +   ="#"          /"!
    X+Y ≠ Z
*
```

The number sign operator is useful in formatting output and in plotting another variable along the same coordinate (an example is given under "Intercept and Plot of Two Functions").

Indirect Commands

Up to this point we have discussed commands which are executed immediately by FOCAL. Next we shall see how indirect commands are written.

If a Teletype line is prefixed by a line number, that line is not executed immediately, instead, it is stored by FOCAL for later execution, usually as part of a sequence of commands. Line numbers must be in the range 1.01 to 15.99. The numbers 1.00, 2.00 etc., are illegal line numbers; they are used to indicate an entire group of lines. The number to the left of the point is called the *group number;* the number to the right is called the *step number*. For example,

```
*ERASE
*1.1 SET A=1
*1.3 SET B=2
*1.5 TYPE %1, A+B
*
```

Indirect commands are executed by typing the GOTO or DO commands.

The GOTO command causes FOCAL to start the program by executing the command at a specified line number. If the user types

```
*GOTO 1.3
= 2*
```

FOCAL started executing the program at the second command in the example above, so that the variable "A" was not previously defined and therefore has a value of zero.

The GO command causes FOCAL to go to the lowest numbered line to begin the program. If the user types a direct GO command after the indirect commands above, FOCAL will start executing at line 1.1.

```
*GO
= 3*
```

The DO command is used to transfer control to a specified step, or group of steps, and then return automatically to the command immediately following the DO command.

```
*ERASE ALL
*1.1 SET A=1; SET B=2
*1.2 TYPE " STARTING "
*1.3 DO 3.2
*2.1 TYPE " FINISHED "
*3.1 SET A=3; SET B=4
*3.2 TYPE %1, A+B
*GO
 STARTING = 3 FINISHED = 7*
```

When the DO command at line 1.3 was reached, the command TYPE %1, A+B was performed and then the program returned to line 2.1.

The DO command can also cause FOCAL to jump to a group of commands and then return automatically to the normal sequence, as shown in the example below.

```
*ERASE ALL
*1.1 TYPE "A "
*1.2 TYPE "B "
*1.3 TYPE "C "
*1.4 DO 5.0
*1.5 TYPE " END"; GOTO 6.1
```

```
*5.1 TYPE "D "
*5.2 TYPE "E "
*5.3 TYPE "F "
*6.1 TYPE ". "
*GO
A B C D E F  END. *
```

When the DO command at line 1.4 was reached, FOCAL executed lines 5.1, 5.2, and 5.3 and then returned to line 1.5.

An indirect command can be inserted in a program by using the proper sequential line number. For example,

```
*ERASE ALL
*4.8 SET A=1; SET B=2
*6.3 TYPE %5.4, B/C+A
*4.9 SET C=1.31*.29
*GO
= 6.2645*
```

where line 4.9 will be executed before line 6.3 and after line 4.8. FOCAL arranges and executes indirect commands in numerical sequence by line number, starting with the smallest line number and going to the largest.

Error Detection

During execution, FOCAL checks for a variety of errors. When an error is detected FOCAL stops execution, types a ? followed by an error message, types an *, and waits for more user input. When the error occurs in a direct statement, FOCAL types the error message immediately after the user terminates that line (direct statements are executed immediately after the line terminator). For example,

```
*SET A=2; PET B=4; TYPE A + B
?03.31
*
```

PET is not a FOCAL command. Therefore, FOCAL issued the error code ?03.31 which means that an illegal command was used. (See "Error Diagnostics" for a list of all error messages and their meanings.)

When an error occurs in an indirect statement the error message is typed when FOCAL encounters that statement during execution. And in addition to the error code FOCAL types the line number of the line containing the error. For example,

```
*1.10 SET A=2; TYPE "A", A, !
*1.20 SET B=4; TYPE "B", B, !
*1.30 GO TO 1.10
*1.40 TYPE "A+B", A+B
*GO
```

```
A=   2.0000
B=   4.0000
?01.89 @ 01.30
*
```

FOCAL executed lines 1.10 and 1.20 and then recognized that GO TO is a 1-word command and should have been written GOTO. Therefore, it issued the error message, meaning GOTO was not used as one word at line number 1.30.

Corrections

If the user types the wrong character, or several wrong characters, he can use the RUBOUT key, which echoes a backslash (\) for each RUBOUT typed, to erase one character to the left each time the RUBOUT key is depressed. For example,

```
*ERASE ALL
*1.1  P\TYPE X-Y
*1.2  SET $=13\ \ \ \X=13
*WRITE
C—FOCAL., 1968
01.10  TYPE X-Y
01.20  SET X=13
*
```

The left arrow(←) erases everything which appears to its *left* on the same line, except when being used to correct a value typed after a colon (:) in response to an ASK command (see "ASK").

```
*1.3 TYPE A, B, C←
*WRITE
C—FOCAL., 1968
01.10  TYPE X-Y
01.20  SET X=13
*
```

A line can be overwritten by repeating the same line number and typing the new command.

```
*14.99 SET C9(N+3)=15
*
```

is replaced by typing

```
*14.99 TYPE C9/Z5-2
*WRITE 14.99
14.99 TYPE C9/Z5-2
*
```

A line or group of lines may be deleted by using the ERASE command with an argument. For example, to delete line 2.21, the user types.

```
*ERASE 2.21
*
```

To delete all of the lines in group 2, the user types

```
*ERASE 2.0
*
```

The user's entire symbol table is erased from memory whenever a line number is retyped or the ERASE command is given. Since FOCAL does not zero memory when loaded, it is good practice to ERASE before defining symbols. The command ERASE ALL erases all user input, i.e., program text and variables. Therefore, the ERASE ALL command should be given *before* writing a new program.

The MODIFY command is another valuable feature, especially in editing. It may be used to change any number of characters in a particular line, as explained under "MODIFY."

Alphanumeric Numbers (Using Letters as Numbers)

Numbers must start with a numeral but may contain letters. FOCAL interprets as a number any character string beginning with a numeral, 0 through 9. An alphanumeric number is a string of alphanumeric characters (excluding symbols) which starts with a number. For example,

```
0ABC     23CAT     9XYZ
```

Each letter in an alphanumeric number is taken as a number, with each letter A through Z having the value of 1 through 26 respectively, except for E which has special meaning and is explained below.

A = 1	J = 10	S = 19
B = 2	K = 11	T = 20
C = 3	L = 12	U = 21
D = 4	M = 13	V = 22
E = (exponentiation)	N = 14	W = 23
F = 6	O = 15	X = 24
G = 7	P = 16	Y = 25
H = 8	Q = 17	Z = 26
I = 9	R = 18	

An easy way to give FOCAL numerical valued letters is to start with the number 0, as in the following example.

```
*TYPE %2, 0AB
= 12*
```

Since after 0, A=1 and B=2, therefore, AB=12. Also,

```
*TYPE OAB+OC
= 15*
```

Since after O, A=1, B=2, and C=3, then 12+3 = 15. Therefore,

```
*TYPE 0XYZ+1
= 2677*
```

because

```
X =  24
Y =   25
Z =    26
+1  +   1
      2677
```

The above example can be solved using the following algorithm.

(X times 10^2) + (Y times 10^1) + (Z times 10^0) + 1 = 2677

or

$(24 \times 100) + (25 \times 10) + (26 \times 1) + 1 = 2677$

Taken as a numeral, the letter E has special meaning. It denotes exponentiation, where the subsequent alphanumerics are taken as the exponent of the preceding alphanumerics.

```
*TYPE %8, OAEC
=       1000*     (A↑10^c=1↑10^3)
*TYPE OAEG
= 10000000*     (A↑10^c=1 × 10^7)
```

Only one E is allowed in any one alphanumeric number.

Alphabetic characters may be used when assigning numerical values to identifiers or variables in response to an ASK statement. An example of this use can be found in lines 3.20 and 3.30 of "Intercept and Plot of Two Functions."

FOCAL COMMANDS

TYPE

The TYPE command is used to request that FOCAL compute and type out a text string, the result of an expression, or the value of an identifier. For example

```
4.14 TYPE 8.1+3.2 - (29.3*5)/2.5↑7
4.15 TYPE (2.2+3.5)*(7.2/3)/59.1↑3
*
```

Several expressions may be computed in a single TYPE command, with commas separating each expression.

```
*ERASE
*9.19 TYPE %4.01, A1*2, E+2↑5, 2.51*81.1
*DO 9.19
=    0.0=     32.0=     204*
```

The output format may be included in the TYPE statement as shown in the example above and as explained under "Handling Text Output."

The user may request a typeout of all identifiers which he has defined by typing TYPE $ and a carriage return. This causes FOCAL to type out the identifiers with their values, in the order in which they were defined. The $ may follow other statements in a TYPE command, but *must be the last operation on the line.*

```
*ERASE
SET L=33; SET B=87; SET Y=55; SET C9=91
*TYPE $
L@(00)=      33.0
B@(00)=      87.0
Y@(00)=      55.0
C 9 (00)=    91.0
*
```

Any text string enclosed in quotation marks may be included in a TYPE command. A carriage return may replace the terminating quotation mark, as shown below:

```
*1.2 TYPE "X SQUARED =
*
```

A text string or any FOCAL command or group of commands may not exceed the capacity of a Teletype line, which is *72 characters* on the ASR33 Teletype. A line may not be continued on the following line. To print out a longer text, each line must start with a TYPE command.

NOTE: For extremely large numbers, there *may be* some input/output conversion error. For example,

```
*TYPE 10↑616
= 0.999959E+616*
```

Exponent overflow is *not* detected:

```
*TYPE 10↑617
= 0.957418E−616
```

Several operations are useful in formulating output.

1. FOCAL does *not* automatically perform a carriage return after executing a TYPE command. The user may insert a carriage return/line feed, by typing an exclamation mark (!).
2. To insert a carriage return without a line feed, the user types a number sign (#).
3. Spaces may be inserted by enclosing them in quotation marks.
4. An expression may be enclosed in question marks to avoid repeating it in quotes (see "Using the Trace Feature").

ASK

The ASK command is normally used in indirect commands to allow the user to input data at specific points during the execution of his program. The ASK command is written in the form,

```
*11.99 ASK X, Y, Z,
*
```

When step 11.99 is encountered by FOCAL, it types a colon (:). The user then types a value in any format for the identifier, followed by a terminator, which may be space, comma, carriage return, or ALT MODE. FOCAL then types another colon and the user types a value for the next identifier. This continues until all the identifiers or variables in the ASK statement have been given values,

```
*11.99 ASK X, Y, Z
*DO 11.99
:5, :4, :3,*
```

where the user typed 5, 4, and 3 as the values, respectively, for X, Y, and Z.

The ALT MODE, when used as a terminator, is nonspacing and leaves the previously defined variable *unchanged,* as shown below.

```
*SET A=5
*ASK A
:123*       (user depressed the ALT MODE Key after typing 123)
*TYPE A
=      5*
```

ALT MODE is frequently used when the user does not wish to change the value of one or more identifiers in an ASK command.

```
*11.99 ASK X, Y, Z
*DO 11.99
:5, :4, :3,*
*DO 11.99     (User did not wish to enter new value for Y, so he typed
:8, ::10,*    ALT MODE in response to second colon.)
*TYPE X, Y, Z
=      8=    4=    10*
```

FOCAL recognizes the value when its terminator is typed. Therefore, a value can be changed but only *before* typing its terminator. This is done by typing a left arrow (←) immediately after the value, and then typing the correct value followed by its terminator. This is the exception to the use of the left arrow, as explained under "Corrections."

Text strings and format control characters may be included in an ASK statement by enclosing the string in quotation marks.

```
*1.10 ASK "HOW MANY APPLES DO YOU HAVE?" APPLES
*DO 1.10
HOW MANY APPLES DO YOU HAVE? :25
*
```

The identifier AP (FOCAL recognizes the first two characters only) now has the value 25.

Alphabetic Responses

Alphabetic characters may be used to assign *numerical* values to identifiers or variables:

```
*1.1 ASK A; TYPE %4, A
*DO 1.1
:ABCD = 1234*
```

Where the user typed ABCD and FOCAL typed the numerical value of ABCD, as was explained under "Alphanumeric Numbers."

WRITE

A WRITE command without an argument (or with the argument *ALL*) causes FOCAL to type out all indirect statements which the user has typed. Indirect statements are those preceded by a line number.

When the user types

```
*WRITE
*WRITE ALL
```

or simply

```
*W
```

and the RETURN key, FOCAL types out a copy of all previously typed indirect statements.

A specific line or group of lines may be typed out with the WRITE command using arguments:

```
*WRITE 2.1          (FOCAL types line 2.1)
*WRITE 2.0          (FOCAL types all group 2 lines)
```

SET

The SET command is used to define identifiers. When FOCAL executes a SET command, the identifier and its value is stored in the user's symbol table, and that value will be substituted for the identifier when the identifier is encountered in the program.

```
*ERASE ALL
*3.4 SET A=2.55; SET B=8.05
*3.5 TYPE %, A+B
*GO
= 0.106000E+02*
```

An identifier may be set equal to previously defined identifiers, which appear in arithmetic expressions.

```
*3.7 SET G=(A+B)*2.2↑5
*
```

ERASE

An ERASE command without an argument is used to delete all identifiers, with their values, from the symbol table.

If the ERASE command is followed by a group number or a specific line number, a group of lines or a specific line is deleted from the program.

```
*ERASE 2            (deletes all group 2 lines)
*ERASE 7.11         (deletes line 7.11)
*
```

The ERASE ALL command erases *all* of the user's input (i.e., symbol table entries and commands).

In the following example, an ERASE command is used to delete line 1.50.

```
*ERASE ALL
*1.20 SET B=2
*1.30 SET C=4
*1.40 TYPE B+C
*1.50 TYPE B—C
*ERASE 1.50
*WRITE ALL
C-FOCAL, 8/68
01.20 SET B=2
01.30 SET C=4
01.40 TYPE B+C

*
```

GO

The GO command requests that FOCAL execute the program, starting with the lowest numbered line. The remainder of the program will be executed in line number sequence. Line numbers must be in the range 1.01 to 15.99. The GO command cannot be given indirectly.

GOTO

The GOTO command causes FOCAL to transfer control to a specific line in the indirect program. It *must* be followed by a specific line number. After executing the command at the specified line, FOCAL continues to the next larger line number, executing the program sequentially. GOTO is a single word.

```
ERASE ALL
*1.1 TYPE "A"
*1.2 TYPE "B"
*1.3 TYPE "C"
*1.4 TYPE "D"
*GOTO 1.2
BCD*
```

DO

The DO command transfers control momentarily to a single line, a group of lines, or the entire indirect program. If transfer is made to a single line, the statements on that line are executed, and control is transferred back to the statement following the DO command. Thus,

the DO command makes a subroutine of the commands transferred to, as shown in this example,

```
*ERASE ALL
*1.1 TYPE "X"
*1.2 DO 2.3; TYPE "Y"
*1.3 TYPE "Z"
*2.3 TYPE "A"
*GO
XAYZA*
```

If a DO command transfers control to a group of lines, FOCAL executes the group sequentially and returns control to the statement following the DO command.

If DO is written without an argument or the user writes DO ALL, FOCAL executes the entire indirect program.

With arguments, DO commands cause specified portions of the indirect program to be executed as closed subroutines. These subroutines may also be terminated by a RETURN command.

If a GOTO or an IF command is executed within a DO subroutine, two actions are possible:

1. If a GOTO or IF command transfers to a line *inside* the DO group, the remaining commands in that group will be executed as in any subroutine before returning to the command following the DO.
2. If transfer is to a line *outside* the DO group, *that line* is executed and control is returned to the command following the DO; unless that line contains another GOTO or IF.

```
*ERASE ALL
*1.1 TYPE "A"; SET X=-1; DO 3.1; TYPE "D"; DO 2
*1.2 DO 2.2
*
*2.1 TYPE "G"
*2.2 IF (X)2.5,2.6,2.7
*2.5 TYPE "H"
*2.6 TYPE "I"
*2.7 TYPE "J"
*2.8 TYPE "K"
*2.9 TYPE %2.01, X; TYPE " "; SET X=X+1
*
*3.1 TYPE "B"; GOTO 5.1; TYPE "F"
*
```

```
*5.1 TYPE "C"
*5.2 TYPE "E"
*5.3 TYPE "L"
*GO
```

(FOCAL types the answer)

```
ABCDGHIJK=—1.0      IGIJK=0.0      BCEL*
```

IF

In order to transfer control after a comparison, FOCAL contains a conditional IF statement. The normal form of the IF statement consists of the word IF, *a space,* a parenthesized expression or variable, and three line numbers in order, separated by commas. The expression is evaluated, and the program transfers control to the first line number if the expression is *less than zero,* to the second line number if the expression has *a value of zero,* or to the third line number if the value of the expression is *greater than zero.*

The program below transfers control to line number 2.10, 2.30, or 2.50, according to the value of the expression in the IF statement.

```
*2.1 TYPE "LESS THAN ZERO"; QUIT
*2.3 TYPE "EQUAL TO ZERO"; QUIT
*2.5 TYPE "GREATER THAN ZERO"; QUIT
*IF (25—25)2.1,2.3,2.5
EQUAL TO ZERO*
```

The IF statement may be shortened by terminating it with a semicolon or carriage return after the first or second line number. If a semicolon follows the first line number, the expression is tested and control is transferred to that line if the expression is less than zero. If the expression is not less than zero, the program continues with the next statement,

```
*2.20 IF (X)1.8; TYPE "Q"
*
```

In the above example, when line 2.20 is executed, if X is less than zero, control is transferred to line 1.8. If not, Q is typed out.

```
*3.19 IF (B)1.8,1.9
*3.20 TYPE B
*
```

In this example, if B is less than zero, control goes to line 1.8; if B is equal to zero, control goes to line 1.9; if B is greater than zero,

control goes to the next statement, which in this case is line 3.20, and the value of B is typed. The expression must be enclosed in parentheses, but other enclosures may be used within the expression.

RETURN

The RETURN command is used to exit from a DO subroutine. When a RETURN command is encountered during execution of a DO subroutine, the program exits from its subroutine status and returns to the command following the DO command that initiated the subroutine status.

QUIT

A QUIT command causes the program to halt and return control to the user. FOCAL types an asterisk and the user may type another command. This command is suggested as the formal end of any program.

Comment

Beginning a command string with the letter C will cause the remainder of that line to be ignored so that comments may be inserted into the program. Such lines will be skipped over when the program is executed, but will be typed out by a WRITE command. A program that is well documented with comments is much more meaningful and easier to understand than one without comments.

FOR

This command is used in setting up program loops and iterations. The general format is

FOR A=B, C, D; (command)

The identifier A is initialized to the value B, then the command following the semicolon is executed, at least once. After the command has been executed, the value of A is incremented by C and compared to the value of D. If A is less than or equal to D, the command after the semicolon is executed again. This process is repeated *until A is greater than D,* and FOCAL goes to the next sequential line.

The identifier A must be a single variable. B, C, and D may be either expressions, variables, or numbers. If comma and the value C are omitted, it is assumed that the increment is 1. If C,D is omitted, it is handled like a SET statement and no iteration is performed.

The computations involved in the FOR statement are done in floating-point arithmetic, and it may be necessary, in some circumstances, to account for this type of arithmetic computation.

Example 1 below is a simple example of how FOCAL executes a FOR command. Example 2 shows the FOR command combined with a DO command.

Example 1:

```
*ERASE ALL
*1.1 SET A=100
*1.2 FOR B=1,1,5; TYPE %5.02, "B IS " B+A,!
*GO
B IS =  101.00
B IS =  102.00
B IS =  103.00
B IS =  104.00
B IS =  105.00
*
```

Example 2:

```
*1.1 FOR X=1,1,5; DO 2.0
*1.2 GOTO 3.1
*
*2.1 TYPE !"          " %3, "X "X
*2.2 SET A=X+100.000
*2.3 TYPE !"          " %5.02, "A "A
*
*3.1 QUIT
*GO

     X =   1
     A =  101.00
     X =   2
     A =  102.00
     X =   3
     A =  103.00
     X =   4
     A =  104.00
     X =   5
     A =  105.00*
```

MODIFY

Frequently, only a few characters in a particular line require changes. To facilitate this job, and to eliminate the need to replace the entire line, the FOCAL programmer may use the MODIFY command. Thus, in order to modify the characters in line 5.41, the user types MODIFY 5.41. This command is terminated by a carriage return whereupon the program waits for the user to type that character in the position in which he wishes to make changes or additions. This character is not

printed. After he has typed the search character, the program types out the contents of that line until the search character is typed.

At this point, the user has seven options:

1. Type in new characters in addition to the ones that have already been typed out.
2. Type a form-feed (CTRL/L); this will cause the search to proceed to the next occurrence, if any, of the search character.
3. Type a CTRL/BELL; this allows the user to change the search character just as he did when first beginning to use the MODIFY command.
4. Use the RUBOUT key to delete one character to the left each time RUBOUT is depressed.
5. Type a left arrow (←) to delete the line over to the left margin.
6. Type a carriage return to terminate the line at that point, removing the text to the right.
7. Type a LINE FEED to save the remainder of the line.

The ERASE ALL and MODIFY commands are generally used only in immediate mode since they return to command mode upon completion. (The reason for this is that internal pointers may be changed by these commands.)

During command input, the left arrow will delete the line numbers as well as the text if the left arrow is the right most character on the line. During MODIFY the line numbers *cannot* be changed.

Notice the errors in line 7.01 below.

```
*7.01 JACK AND BILL W$NT UP THE HALL
*MODIFY 7.01
JACK AND B\JILL W$\ENT UP THE HA\ILL
*WRITE 7.01
07.01 JACK AND JILL WENT UP THE HILL
*
```

To modify line 7.01, a B was typed by the user to indicate the character to be changed. FOCAL stopped typing when it encountered the search character, B. The user typed the RUBOUT key to delete the B, and then typed the correct letter J. He then typed the CTRL/BELL keys followed by the $, the next character to be changed. The RUBOUT deleted the $ character, and the user typed an E. Again a search was made for an A character. This was changed to I. A LINE FEED was typed to save the remainder of the line.

NOTE

When the MODIFY command is used the values in the user's symbol table are reset to zero. Therefore, if the user defines his symbols in direct statements and then uses a MODIFY command, the values of his symbols are *erased* and *must be redefined.*

However, if the user defines his symbols by means of indirect statements prior to using a MODIFY command, the values will not be erased because these symbols are not entered in the symbol table until the statements defining them are executed.

Using the Trace Feature

The trace feature is useful in checking an operating program. Those parts of the program which the user has enclosed in question marks will be printed out as they are executed by FOCAL.

In the following example, parts of three lines will be printed.

```
*ERASE ALL
*1.1 SET A=1
*1.2 SET B=5
*1.3 SET C=3
*1.4 TYPE %2, ?A+B-C?,!
*1.5 TYPE ?B+A/C?,!
*1.6 TYPE ?B-C/A?
*GO
A+B-C=  3
B+A/C=  5
B-C/A=  2*
```

When only one ? is inserted the trace feature becomes operative when FOCAL encounters the ? during execution, and the program is printed out from that point until another ? is encountered (the program may loop through the same ?), until an error is encountered (execution stops and an error message is typed), or until program completion.

```
*ERASE ALL
*1.1 ?SET A=0B; TYPE %3, A!
*1.2 FOR B=1,1,4; TYPE B+A!
*GO
SET A=0B; TYPE %3, A!
=   0 FOR B=1,1,4; TYPE B+A!
=   1 TYPE B+A!
=   2 TYPE B+A!
=   3 TYPE B+A!
=   4*
```

In this example, FOCAL encountered the ? as it entered line 1.1 and traced the entire program. In the following example an error has been inserted in the FOR statement—FOCAL will detect it.

```
*ERASE ALL
*1.1 SET A=0B; TYPE %3, A!
*1.2 FOR B=1,1:4; TYPE B+A!
*GO?
C—FOCAL SET A=0B; TYPE %3, A!
=      0 FOR B=1,1:?07.14 @ 01.20
*
```

GO? traced the entire program under the same conditions as explained above when only one ? is inserted.

Mathematical Functions

Functions are provided to give extended arithmetic capabilities and to give the potential for expansion to additional input/output devices. A standard function call consists of four letters *beginning with the letter F* and followed by a parenthetical expression.

```
FSGN(A—B*2)
```

There are three basic types of functions, two of which are included in the basic FOCAL program.

The first type contains integer part, sign part, and absolute value functions.

The second type, the extended arithmetic functions, are loaded at the option of the user. They will consume approximately 800 locations of the users program storage area. These arithmetic functions are adapted from the extended arithmetic functions of the PDP-8 three-word floating-point package and are fully described in the *Floating Point System* manual, Order No. Digital-8-5-S.

The third type are the input/output functions. These include a non-statistical random number generator (FRAN). This function uses the FOCAL program itself as a table of random numbers. An expanded version could incorporate the random number generator from the DECUS library. Following are examples of the functions now available.

1. The square root function (FSQT) computes the square root of the expression within parentheses.

```
*TYPE %2, FSQT(4)
=  2*
*TYPE FSQT(9)
=  3*
*TYPE FSQT(144)
=  12*
```

2. The absolute value function (FABS) outputs the absolute or positive value of the number in parentheses.

```
*TYPE FABS(−66)
= 66*
*TYPE FABS(−23)
= 23*
*TYPE FABS(−99)
= 99*
```

3. The sign part function (FSGN) outputs the sign part (+ or −) of a number and the integer part becomes a 1.

```
*TYPE FSGN(4−6)
=− 1*
*TYPE FSGN(4−4)
=  1*
TYPE FSGN(−7)
=− 1*
```

4. The integer function (FITR) has the value of the integer part of any number:

```
*TYPE FITR(5.2)
=  5*
```

It may be used for truncating:

```
*TYPE FITR(55.66+.5)
= 56*
*TYPE FITR(77.434+.5)
= 77*
```

For negative numbers, FITR gives the next smaller integer:

```
*TYPE FITR(−4.1)
=− 5*
```

5. The random number generator function (FRAN) computes a nonstatistical pseudo-random number between ±1.

```
*TYPE %, FRAN( )
=−0.250000E+00*
*TYPE FRAN( )
=−0.623535E+00*
```

6. The exponential function (FEXP) computes e to the power within parentheses. (e=2.718281)

```
*TYPE FEXP(6.66953E—1)
= 0.194829E+01*
*TYPE FEXP(.666953)
= 0.194829E+01*
*TYPE FEXP(1.23456)
= 0.343687E+01*
*TYPE FEXP(—1.)
= 0.367879E+00*
```

7. The sine function (FSIN) calculates the sine of an angle in radians.

```
*TYPE %, FSIN(3.14159)
= 0.238419E—05*
*TYPE FSIN(1.400)
= 0.985450E+00*
```

Since FOCAL requires that angles *must be expressed in radians,* to find a function of an angle in degrees, the conversion factor, $\pi/180$, must be used. To find the sine of 15 degrees,

```
*SET PI=3.14159; TYPE FSIN(15*PI/180)
= 0.258819E+00*
*TYPE FSIN(45*3.14159/180)
= 0.707106E+00*
```

8. The cosine function (FCOS) calculates the cosine of an angle in radians.

```
*TYPE FCOS(2*3.141592)
= 0.100000E+01*
*TYPE FCOS(.50000)
= 0.877582E+00*
*TYPE FCOS(45*3.141592/180)
= 0.707107E+00*
```

9. The arc tangent function (FATN) calculates the angle in radians whose tangent is the argument within parentheses.

```
*TYPE FATN(1.)
= 0.785398E+00*
*TYPE FATN(.31305)
= 0.303386E+00*
*TYPE FATN(3.141592)
= 0.126263E+01*
```

10. The logarithm function (FLOG) computes the natural logarithm ($\log_e$) of the number within parentheses.

```
*TYPE FLOG(1.00000)
= 0.000000E+00*
*TYPE FLOG(1.98765)
= 0.686953E+00*
*TYPE %5.03, FLOG(2.065)
= 0.725*
```

Reading From the High-Speed Reader

Up to this point we have considered only one source of input to FOCAL programs—the Teletype keyboard. We learned that when FOCAL types an *, it is ready to accept user commands.

The user can switch the input device to the high-speed paper tape reader by typing an asterisk in the first position, or immediately following the line number in a statement. The following statements cause FOCAL to *read* a tape from the high-speed reader.

```
**;      The first asterisk is that typed out by FOCAL, informing the user that a command may be input from the keyboard. The second asterisk typed by the user, switches input to the high-speed reader.
*1.21*   The user types * after the line number.
```

To switch back to the keyboard, the user types another *. If there is no tape in the high-speed reader, or when an "end of tape" condition is reached, FOCAL automatically switches back to keyboard input.

This feature is useful for loading FOCAL programs and for inputting large amounts of data during execution of a FOCAL program.

When the following statement is executed FOCAL accepts four pieces of data from the high-speed reader.

```
*1.10*; FOR I=1,4; ASK HR(I)
*1.11*
*DO 1.10
: : : : *
```

The user typed an asterisk after line 1.11 to return control to the keyboard. If the tape contains fewer than four pieces of data, the remaining pieces will be taken from the keyboard.

Generating Program Tapes

To generate a program tape of the user's FOCAL program on the low-speed paper tape punch, the user should

1. Respond to * by typing WRITE ALL (do not depress the RETURN key).
2. Depress low-speed punch to ON.
3. Generate leader tape (depress the SHIFT, REPT, and P keys in that order; release in reverse order).
4. Depress RETURN key.

When the user's FOCAL program has been typed and punched, the user should

5. Generate trailer tape (depress the SHIFT, REPT, and P keys in that order; release in reverse order).
6. Depress low-speed punch to OFF.
7. Remove and label the punched paper tape.

The user's FOCAL program is still in the computer. FOCAL is waiting for user input.

EXAMPLES OF FOCAL PROGRAMS

The following programs reveal some of FOCAL's features in various applications. The examples show that FOCAL finds practical application in any situation:

FORTRAN-type problems are handled easily with little programming time.

FOCAL's easy-to-learn language allows the user to concentrate more on his problem than on programming.

FOCAL as a tool is easier to learn and use than a slide rule or any other desk calculator, and it offers vastly more than any combination of previous problem solving tools.

FOCAL can calculate complex problems and print and/or display the results in "one fell swoop."

The example programs included in this section are maintained on punched paper tape. Each program was loaded into core using FOCAL's high-speed paper tape reader input feature. The WRITE command was then used to get the program printout for inclusion here. Then the GO command was issued to execute each program.

When using the WRITE command, FOCAL immediately identifies the version of the FOCAL tape being used—in this case, C-FOCAL, 8/68. The C preceding FOCAL , 8/68 is the *comment* line indicator.

Table Generation Using Functions

The ability to evaluate simple arithmetic expressions and to generate values with the aid of library functions is one of the first benefits to be obtained from learning the FOCAL language. In this example, a table of the sine, cosine, natural logarithm, and exponential values is generated for a series of arguments. As one becomes familiar with these and other library functions, it becomes easy to combine them with the standard arithmetic operations of addition, subtraction, multiplication, division, and exponentiation. The user is then able to evaluate any given formula for a single value or for a range of values as in this example.

Although FOCAL allows the typing of more than one command per line, each command in this example has been typed on a separate line to maintain clarity and because of the length of several of the commands. In this example, line 01.05 outputs the desired column headings. Line 01.10 is the loop to generate values for I, beginning with the value 1.00000 and continuing in increments of .00001 up through the value 1.00010; the DO 2.05 command at the end of this second line causes line 02.05 to be executed for each value of I. Line 02.05 is the command to evaluate the various library functions for the I arguments; the %7.06 specifies that all output results up to the next % symbol

are to appear in fixed-point format with one digit position to the left of the decimal and six digit positions to the right: the second % symbol reverts the output mode back to floating point for the remaining values — FLOG(I). and FEXP(I). Line 01.20 (optional) returns control to the user.

Several techniques can be noted in line 02.05 of this example.

1. FOCAL commands can be abbreviated to the first letter of the command followed by a space, as shown by the use of T instead of TYPE. This technique can be used to shorten command strings.
2. Arguments can be enclosed in various ways: ( ), < >, [ ]. This ability is useful in matching correctly when a number of such enclosures appear in a command.
3. Spaces can be inserted in an output format by enclosing the appropriate number of spaces within quotation marks. Such use of spacing is recommended to improve the readability of the output results.
4. FOCAL presently allows accuracy of six significant digits, which makes possible the use of very small loop increments (in this example, .00001); this should eliminate the need to interpolate between table values of trigonometry functions in most cases.

```
C-FOCAL , 8/68

01.05 T "      I          SINE       COSINE           LOG              E"!
01.10 FOR I=1,.00001,1.0001; DO 2.05
01.20 QUIT

02.05 T %7.06,I," ",FSIN(I)," ",FCOS<I>," ",%,FLOG(I)," ",FEXP(I),!
*
*GO
      I          SINE       COSINE           LOG              E
= 1.000000   = 0.841471   = 0.540303   = 0.000000E+00   = 0.271828E+01
= 1.000010   = 0.841476   = 0.540295   = 0.977507E-05   = 0.271831E+01
= 1.000020   = 0.841481   = 0.540286   = 0.195501E-04   = 0.271834E+01
= 1.000030   = 0.841487   = 0.540278   = 0.293249E-04   = 0.271836E+01
= 1.000040   = 0.841492   = 0.540270   = 0.390997E-04   = 0.271839E+01
= 1.000050   = 0.841497   = 0.540262   = 0.488744E-04   = 0.271842E+01
= 1.000060   = 0.841503   = 0.540254   = 0.586490E-04   = 0.271844E+01
= 1.000070   = 0.841508   = 0.540245   = 0.684235E-04   = 0.271847E+01
= 1.000080   = 0.841513   = 0.540237   = 0.781980E-04   = 0.271849E+01
= 1.000090   = 0.841519   = 0.540229   = 0.879723E-04   = 0.271852E+01
= 1.000100   = 0.841524   = 0.540221   = 0.977465E-04   = 0.271855E+01
*
```

Addition Exerciser Using Functions

FOCAL randomly selects pairs of 1- or 2-digit positive integers and sets them up in an addition problem. The user types the answer to the addition problem. FOCAL then checks the user's answer against its own and tells the user whether he was right or wrong. If the user's answer is correct, FOCAL types another addition problem. If the user's

answer is wrong, FOCAL gives him two more tries at the problem. If after three tries the user still has not typed the correct answer, FOCAL suggests that the user consult his teacher, gives the correct answer to the problem, and goes on to another problem.

This program can be used as a drill for the young student of addition, and with a few modifications it can be extended to give only subtraction problems or randomly vary between addition and subtraction.

The program uses three functions in line 01.10.

```
C-FOCAL , 8/68

01.05 TYPE "HELLO, PLEASE ADD THE FOLLOWING SETS OF NUMBERS."!
01.10 SET A=FABS(FITR<100*FRAN[]>); SET B=FABS(FITR<99*FRAN[]>)
01.20 TYPE %7, A,!B,!"---------"!
01.30 ASK REPLY,!
01.40 IF (REPLY-A-B) 2.1,1.5,2.1
01.50 SET WR=0;TYPE "THAT IS CORRECT."!
01.60 GOTO 1.1

02.10 SET WR=WR+1; IF (WR-2) 2.2,2.2,3.1
02.20 T "SORRY, TRY AGAIN,"!; GOTO 1.2

03.10 T "IF YOU ARE HAVING TROUBLE, ASK YOUR TEACHER FOR HELP."!
03.20 TYPE "THE CORRECT ANSWER IS "A+B,!
03.30 GOTO 1.1
*
*GO
HELLO, PLEASE ADD THE FOLLOWING SETS OF NUMBERS.
=       0
=       0
---------
:1

SORRY, TRY AGAIN,
=       0
=       0
---------
:0

THAT IS CORRECT.
=      32
=       0
---------
:32

THAT IS CORRECT.
=      28
=       5
---------
:33

THAT IS CORRECT.
```

```
=       8
=       4
---------
:84

SORRY, TRY AGAIN.
=       8
=       4
---------
:0

SORRY, TRY AGAIN.
=       8
=       4
---------
:48

IF YOU ARE HAVING TROUBLE, ASK YOUR TEACHER FOR HELP.
THE CORRECT ANSWER IS =       12
=       7
=       0
---------
:10

IF YOU ARE HAVING TROUBLE, ASK YOUR TEACHER FOR HELP.
THE CORRECT ANSWER IS =        7
=       6
=      81
---------
:87

THAT IS CORRECT.
=       4
=       8
---------
:12

THAT IS CORRECT.
=       9
=       7
---------
:17

SORRY, TRY AGAIN.
=       9
=       7
---------
:16

THAT IS CORRECT.
```

Finding Roots of a Quadratic Equation

This program uses the square root function to analyze a quadratic equation of the form:

$$Y = AX^2+BX+C$$

The analysis is followed by output of the roots and/or a comment concerning the nature of the roots. After the output the program restarts and accepts new values for A, B, and C. Some interesting features are:

1. It is not phased by a negative discriminant.
2. Input is set up so that if the space bar is used as a terminator, A, B, and C will be printed in a row format which is suggestive of their format in the equation.
3. If A is equal to zero, FOCAL types an error message. Therefore, division by zero is not possible.

```
C-FOCAL , 8/68

01.10 ASK ! ?A B C?; SET ROOT=B'2-4*A*C
01.20 IF (A) 1.4,1.3,1.4
01.30 TYPE ! "THIS IS A FIRST DEGREE EQUATION" !; GOTO 1.10
01.40 TYPE %6.03, ! " THE ROOTS ARE"; IF (ROOT) 1.7,1.6
01.50 TYPE !,(-B+FSQT<ROOT>)/2*A,!,(-B-FSQT<ROOT>)/2*A; GOTO 1.1
01.60 TYPE ! -B/2*A,!; GOTO 1.10
01.70 TYPE " IMAGINARY"!, -B/2*A," + (",FSQT(-ROOT)/2*A,")" "*I"
01.80 TYPE !,-B/2*A," - (",FSQT[-ROOT]/2*A,")*I",!; GOTO 1.10
*
*GO

A :4 B :2 C:8
 THE ROOTS ARE IMAGINARY
=-  0.250  +  (=   1.392)*I
=-  0.250  -  (=   1.392)*I

A :0 B :2 C:4
THIS IS A FIRST DEGREE EQUATION

A :1 B :4 C:4
 THE ROOTS ARE
=-  2.000

A :1 B :-2 C:4
 THE ROOTS ARE IMAGINARY
=   1.000  +  (=   1.732)*I
=   1.000  -  (=   1.732)*I

A :4 B :6 C:8
 THE ROOTS ARE IMAGINARY
=-  0.750  +  (=   1.199)*I
=-  0.750  -  (=   1.199)*I

A :1 B :6 C:8
 THE ROOTS ARE
=-  2.000
=-  4.000
```

Square Completer Using the FOR Command

The program directs the FOCAL user to specify the values for A, B, and C for the parabolic function of the form:

$$F(X)=AX^2+BX+C$$

The program quickly manipulates these values and prints an expression of the same function in the form:

$$F(X)=A(X-K)^2+C-K^2$$

where in line number 01.20, below, K is SET equal to $-B/2A$, the value of the parabola's axis of symmetry.

It is brought out in many high school algebra classes that this is a handy way to visualize a parabolic function since it expresses, in easily readable forms, various characteristics of the graph of the function. The program is designed to do the complicated arithmetic involved in completing the square of a parabolic function.

```
C-FOCAL , 8/68

01.09 TYPE !,"INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C",!
01.10 ASK ?A B C?; IF (A) 1.2,1.4;
01.20 SET K=-B/2*A; SET C=C/A-K↑2
01.30 TYPE %4.02, !," F(X)",A,"*(X-",K,")↑2",C,!; GOTO 1.09
01.40 TYPE !, "THAT IS NO QUADRATIC!",!!; GOTO 1.09
*
*GO

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :1 B :1 C:1
 F(X)=  1.00*(X-=- 0.50)↑2=  0.75

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :3 B :6 C:3
 F(X)=  3.00*(X-=- 1.00)↑2=  0.00

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :3 B :18 C:24
 F(X)=  3.00*(X-=- 3.00)↑2=- 1.00

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :1 B :1 C:12
 F(X)=  1.00*(X-=- 0.50)↑2= 11.75

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :AB B :CD C:FG
 F(X)= 12.00*(X-=- 1.42)↑2=  3.58

INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :0 B :1 C:B
THAT IS NO QUADRATIC!


INPUT A,B,C SUCH THAT F(X)=2X↑2+BX+C
A :2 B :4 C:6
 F(X)=  2.00*(X-=- 1.00)↑2=  2.00
```

Interest Payment Program

This is an example of a business-oriented FOCAL program. It is designed to give a complete picture of the payments which will be made on a loan, with interest, on an installment plan basis.

Under program control, the computer requests as input the amount of a loan, the percentage of interest on that loan, and the length of time over which the loan is to be paid. The computer then calculates and types the amount of monthly payments to be paid, the total amount of interest which will be paid, and a table showing interest paid, amount applied to principle, and balance due after each payment.

```
C-FOCAL , 8/68

01.10 ASK !!%7.02, "ENTER INTEREST IN PERCENT " J,!
01.14 SET J=J/100
01.16 ASK "ENTER AMOUNT OF LOAN " A,!
01.20 ASK "NUMBER OF YEARS " N,!
01.24 ASK "NUMBER OF PAYMENTS PER YEAR " M,!!
01.30 SET N=N*M; SET I=J/M
01.34 SET B=1+I
01.40 SET R=A*I/(1-1/B↑N)
01.42 TYPE "MONTHLY PAYMENT " R,!
01.48 TYPE "TOTAL INTEREST  " R*N-A,!!

02.05 SET B=A
02.10 TYPE " INTEREST       APP TO PRIN              BALANCE",!
02.12 SET L=B*I; SET P=R-L
02.16 SET B=B-P
02.18 TYPE L, "       "P,"             "B,!
02.20 IF (B-R) 2.24,2.24,2.12
02.24 TYPE B*I,"       "R-B*I,! "LAST PAYMENT!" B*I+B,!
02.30 QUIT
*
*GO

ENTER INTEREST IN PERCENT :5
ENTER AMOUNT OF LOAN :2345.00
NUMBER OF YEARS :4
NUMBER OF PAYMENTS PER YEAR :3

MONTHLY PAYMENT =   217.23
TOTAL INTEREST  =   261.75

 INTEREST        APP TO PRIN             BALANCE
=    39.08       =   178.15             =  2166.85
=    36.12       =   181.12             =  1985.74
=    33.10       =   184.13             =  1801.61
=    30.03       =   187.20             =  1614.40
=    26.91       =   190.32             =  1424.08
=    23.74       =   193.50             =  1230.58
=    20.51       =   196.72             =  1033.86
=    17.23       =   200.00             =   833.87
=    13.90       =   203.33             =   630.53
=    10.51       =   206.72             =   423.81
=     7.06       =   210.17             =   213.65
=     3.56       =   213.67
LAST PAYMENT!=   217.21
*
```

Temperature Conversion Using the FOR Command

The ability for loop parameters to be negative, zero, fractional, or expressions, provides power beyond many other similar languages in simplifying the routine's structure. It also reemphasizes the flexibility and control over FOCAL programs at the time they are run.

Measurement system conversions are time consuming in many lines of work. A short FOCAL program, such as the one illustrated in the following example, eliminates hours of repeated calculations. In this particular example, the problem is to convert temperatures from degrees Fahrenheit to degrees Centigrade, using the formula:

$$T°C=5/9(T°F-32)$$

This routine is quite similar in structure to the "Table Generation" example. The one basic difference is that here the user can input the loop parameters which govern the generation of the output. Thus, provision has been made for output of properly labeled requests for starting, ending, and incrementing values and their input for use by the program.

```
C-FOCAL , 8/68

02.10 ASK !,"FROM ",START,"  TO ",END,"  DEGREES FAHRENHEIT",!
02.20 ASK "      IN INCREMENTS OF ",INCR," DEGREES",!!
02.30 TYPE "THE APPROXIMATE FAHRENHEIT TO CENTIGRADE CONVERSIONS ARE:"
02.40 FOR T=START,INCR,END; TYPE !; DO 2.50
02.45 QUIT
02.50 TYPE " ",T," FAHRENHEIT DEG.  ",(T-32)*5/9," CENTIGRADE DEG."
*
*GO

FROM :-40   TO :80   DEGREES FAHRENHEIT
     IN INCREMENTS OF :20  DEGREES

THE APPROXIMATE FAHRENHEIT TO CENTIGRADE CONVERSIONS ARE:
 =-    40.00 FAHRENHEIT DEG.  =-    40.00 CENTIGRADE DEG.
 =-    20.00 FAHRENHEIT DEG.  =-    28.89 CENTIGRADE DEG.
 =      0.00 FAHRENHEIT DEG.  =-    17.78 CENTIGRADE DEG.
 =     20.00 FAHRENHEIT DEG.  =-     6.67 CENTIGRADE DEG.
 =     40.00 FAHRENHEIT DEG.  =      4.45 CENTIGRADE DEG.
 =     60.00 FAHRENHEIT DEG.  =     15.56 CENTIGRADE DEG.
 =     80.00 FAHRENHEIT DEG.  =     26.67 CENTIGRADE DEG.*
```

One-Line Function Plotting

This example demonstrates the use of FOCAL to present, in graphic form, some given function over a range of values. In this example, the function used is

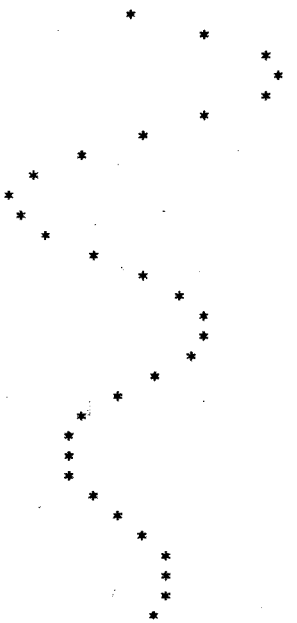
$$y = 30 + 15(\text{SIN}(x))e^{-.1x}$$

with x ranging from 0 to 15 in increments of .5. This damped sine wave has many physical applications, especially in electronics and mechanics (for example, in designing the shock absorbers of a car).

In the actual coding of the example, the variables I and J were used in place of x and y, respectively; any two variables could have been used. The single line 08.01 contains a set of nested loops for I and J. The J loop types spaces horizontally for the y coordinate of the function; the I loop prints the * symbol and the carriage return and line feeds for the x coordinate. The function itself is used as the upper limit of the J loop, again showing the power of FOCAL commands.

The technique illustrated by this example can be used to plot any desired function. Although the * symbol was used here, any legal FOCAL character is acceptable.

```
C-FOCAL , 8/68

08.01 F I=0,.5,15; T "*",!; F J=0,30+15*FSIN(I)*FEXP[-.1*I]; T " "
*
*DO 8.01
*
```

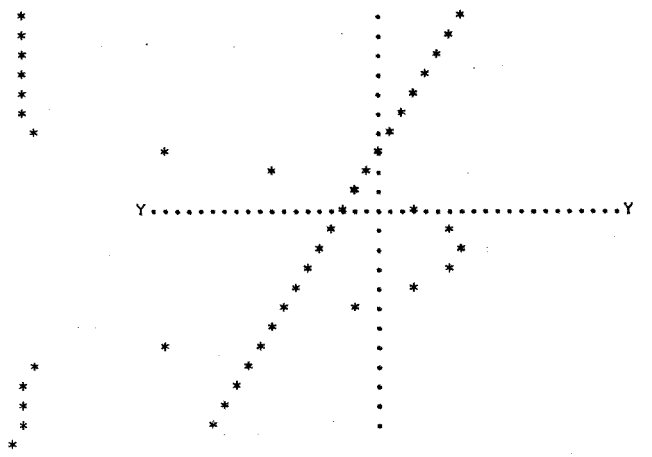


Intercept and Plot of Two Functions

Values are first computed and printed for two monotonic functions. Then these curves are plotted within specified limits. Nonmonotonic functions must be plotted using the method of residuals.

```
C-FOCAL , 8/68

01.02 ASK "LOWER LIMIT",LL,!"UPPER LIMIT",UL,!"INCREMENT",IN,!
01.10 SET Y1=0; SET Y2=0;
01.20 FOR X=LL,IN,UL; SET Y1=-X-3; SET Y2=3+4*X-X↑2; DO 2.0

02.10 IF (Y2-Y1) 2.4,2.2,2.4
02.20 TYPE !! "THE POINT OF INTERSECTION IS ",!; GOTO 2.3
02.30 TYPE "X1",X,"   ","Y1",Y1,!,"X2",X,"   ","Y2",Y2,!!; RETURN
02.40 TYPE "X1",X,"   ","Y1",Y1,!,"X2",X,"   ","Y2",Y2,!!

03.10 TYPE "DO YOU WANT A PLOT?"
03.20 ASK "(TYPE Y FOR YES. TYPE N FOR NO) ",AN,!!
03.30 IF (AN-25) 9.1,4.1

04.10 FOR X=LL,IN,UL; DO 5.0

05.01 IF (X) 5.1,5.02,5.1
05.02 TYPE "          Y.....Y",#
05.10 FOR Y=0,30; TYPE " "
05.20 TYPE ".",#
05.30 FOR Y=0,30+(-X-3); TYPE " "
05.40 TYPE "*",#
05.50 FOR Y=0,30+(3+4*X-X↑2); TYPE " "
05.60 TYPE "*",!
05.70 RETURN

09.10 QUIT
*
*GO
LOWER LIMIT:-10
UPPER LIMIT:10
INCREMENT:1
X1=-   10.00   Y1=    7.00
X2=-   10.00   Y2=-  137.00

X1=-    9.00   Y1=    6.00
X2=-    9.00   Y2=-  114.00

X1=-    8.00   Y1=    5.00
X2=-    8.00   Y2=-   93.00

X1=-    7.00   Y1=    4.00
X2=-    7.00   Y2=-   74.00

X1=-    6.00   Y1=    3.00
X2=-    6.00   Y2=-   57.00

X1=-    5.00   Y1=    2.00
X2=-    5.00   Y2=-   42.00

X1=-    4.00   Y1=    1.00
X2=-    4.00   Y2=-   29.00

X1=-    3.00   Y1=    0.00
X2=-    3.00   Y2=-   18.00

X1=-    2.00   Y1=-   1.00
X2=-    2.00   Y2=-   9.00
```

```
THE POINT OF INTERSECTION IS
X1=-     1.00    Y1=-     2.00
X2=-     1.00    Y2=-     2.00

X1=      0.00    Y1=-     3.00
X2=      0.00    Y2=      3.00

X1=      1.00    Y1=-     4.00
X2=      1.00    Y2=      6.00

X1=      2.00    Y1=-     5.00
X2=      2.00    Y2=      7.00

X1=      3.00    Y1=-     6.00
X2=      3.00    Y2=      6.00

X1=      4.00    Y1=-     7.00
X2=      4.00    Y2=      3.00

X1=      5.00    Y1=-     8.00
X2=      5.00    Y2=-     2.00


THE POINT OF INTERSECTION IS
X1=      6.00    Y1=-     9.00
X2=      6.00    Y2=-     9.00

X1=      7.00    Y1=-    10.00
X2=      7.00    Y2=-    18.00

X1=      8.00    Y1=-    11.00
X2=      8.00    Y2=-    29.00

X1=      9.00    Y1=-    12.00
X2=      9.00    Y2=-    42.00

X1=     10.00    Y1=-    13.00
X2=     10.00    Y2=-    57.00

X1=     11.00    Y1=-    13.00
X2=     11.00    Y2=-    57.00

DO YOU WANT A PLOT?(TYPE Y FOR YES. TYPE N FOR NO) :Y
```



Plotting on the Oscilloscope

This is an example of using the FDXS and FDIS functions for plotting on the oscilloscope. The functions are used in the SET command of statement number 01.40 as shown below, which is equivalent to

SET H=FDXS(X)−FDIS(Y)

where X and Y are the x and y coordinates of the point to be plotted on the scope.

The program will plot a sine wave with a user-determined number of complete cycles (Q).

```
C-FOCAL , 8/68

01.10 ASK "NUMBER OF CYCLES" Q
01.20 F I=0.,50; S X(I)=20*I; S Y(I)=(FSIN(3.14159*I/<25/Q>)+1)*500
01.30 TYPE !! "READY TO PLOT" ,!
01.40 FOR I=0.,50; SET H=FDXS(X(I))-FDIS(Y<I>)
01.50 GOTO 1.4
*
*GO
NUMBER OF CYCLES:2
READY TO PLOT
```

Formula Evaluation for Circles and Spheres

In this example, FOCAL is used to calculate, label, and output the following values for an indefinite number of radii typed in by the user.

Given: radius(R)	
Program calculates:	circle diameter 2R
	circle area πR^2
	circle circumference $2\pi R$
	sphere volume $4/3\pi R^3$
	sphere surface area $4\pi R^2$

Although the American system of inches is used in this example, conversions to other systems (metric, for example) could be very easily incorporated into the program, thus eliminating any need for hand-calculated conversions.

The program is very straightforward. ASK is used to allow the user to type in the radius value to be used in the calculations. SET is used to supply the value of π (PI). TYPE is used for all calculations and output. Note that if a value (such as PI in this example) is to be entered once and then used in repeated calculations, it should be entered by a SET command which is outside the calculation loop, otherwise, the variable would be set at the beginning of each pass through the loop. However, if the value of the variable changes during each iteration, then it must be calculated either by a SET or TYPE command within the loop.

The use of the GOTO command (line 01.60) results in an infinite loop of lines 01.10 through 01.60. This technique is used when the number of desired repetitions is not known. The looping process can be terminated at any time by typing CTRL/C. If, however, the number of desired repetitions is known (e.g., 10), the following method can be used.

```
*SET PI=3.14159
*1.1 ASK ...
  .
  .
  .
*1.6 TYPE !!!!!          (Eliminate GOTO 1.1)
*
*FOR I=1,10; DO 1        (Direct command; causes all
                          steps in group 1 to be ex-
                          ecuted 10 times)
```

The ability to choose between these methods provides great flexibility in actually running FOCAL programs.

```
C-FOCAL , 8/68 .

01.01 SET PI=3.141592
01.10 ASK "A RADIUS OF", R, " INCHES"
01.20 TYPE %8.04, !, " GENERATES A CIRCLE OF:", !
01.21 TYPE "           DIAMETER", 2*R, " INCHES", !
01.30 TYPE "           AREA", PI*R↑2, " SQUARE INCHES", !
01.35 TYPE "           CIRCUMFERENCE", 2*PI*R, " INCHES", !
01.40 TYPE !, " AND A SPHERE OF:", !
01.49 TYPE "           VOLUME", (4/3)*PI*R↑3, " CUBIC INCHES", !
01.50 TYPE "           AND SURFACE AREA", 4*PI*R↑2, " SQUARE INCHES"
01.60 TYPE !!!; GOTO 01.10
*
*GO
A RADIUS OF:1.414  INCHES
 GENERATES A CIRCLE OF:
          DIAMETER=    2.8280 INCHES
          AREA=    6.2813 SQUARE INCHES
          CIRCUMFERENCE=    8.8844 INCHES

 AND A SPHERE OF:
          VOLUME=   11.8423 CUBIC INCHES
          AND SURFACE AREA=   25.1252 SQUARE INCHES


A RADIUS OF: ...
```

Simultaneous Equations and Matrices

Many disciplines use subscripted variables for vectors in one, two, or more dimensions to store and manipulate data. A common use is the 2-dimensional array or matrix for handling sets of simultaneous equations. For example,

Given:
$$1X_1 + 2X_2 + 3X_3 = 4$$
$$4X_1 + 3X_2 + 2X_3 = 1$$
$$1X_1 + 4X_2 + 3X_3 = 2$$

Find: the values of X_1, X_2, and X_3 to satisfy all three equations simultaneously.

The solution can be reduced to simple mathematics between the various elements of the rows and columns until correct values of X are found, as shown in Example 3 below.

Since FOCAL uses only a single subscript, the handling of two or more dimensions requires the generation of a *linear subscript* which represents the correct position if it were stored in normal order; i.e., leftmost subscript moving fastest.

IN ONE DIMENSION

ARRAY( )		
	A	0
	B	1
	C	2
	D	3
	E	4

Element D could be represented as ARRAY(3); any element in this array can be represented by a subscript in the range 0 through 4. The first element in an array always has a subscript of 0.

IN TWO DIMENSIONS

ARRAY(row, column) or A(I,J)

This must be reduced to the form A(G), where G is a function of I and J; that is, A(I,J) = A(G). Consider the diagram

	J = 0	1	2
I = 0	0	5	10
1	1	6	11
2	2	7	12
3	3	8	13
4	4	9	14

The numbers along the outside edges of the box above are the 2-dimensional subscripts; the numbers inside the box are the linear subscripts.

Thus each combination of I and J can be given a unique value, e.g., for I=2 and J=1 the element is 7.

Notice that for a constant I, increasing the value of J by one increases the value of the linear subscript by five. Similarly, for a constant J, increasing the value of I by one increases the linear subscript by three.

The array, above, has five rows and three columns, so two values can be defined:

IMAX = 5 and JMAX = 3

The total number of elements is IMAX * JMAX = 15. To generate the number G in any box, using the corresponding values of I and J, the formula is

G =I + IMAX * J or A(G)

which is equivalent to A(I+IMAX*J). The example of solving simultaneous equations, above, uses this algorithm for subscripts merely by replacing I, IMAX, and J with J, L, and K, respectively, so as to form the equation

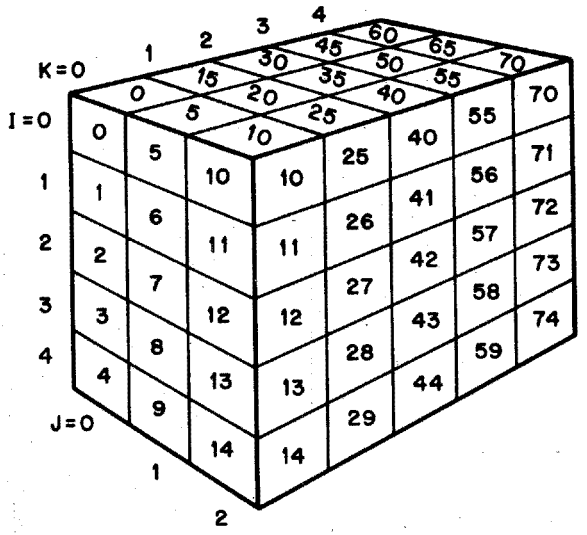
A(J+L*K)

Each element in a 2-dimensional array represents an area.

IN THREE DIMENSIONS

ARRAY(row, column, plane) = A(I,J,K) = A(G)

In a 3-dimensional array, each array represents a volume. Three dimensions can be illustrated as a cube.



This cube has dimensions of five rows, three columns, and five planes; thus, IMAX = 5, JMAX = 3, and KMAX = 5. Each plane is numbered exactly as in the 2-dimensional example, except with the addition of 15 times K (with K = the number of planes back from the first) to each subscript in the first plane.

For example,

Upper lefthand square, back one plane from the first=15
I=0, J= 0, K=1; I+(IMAX*J)+(IMAX*JMAX*K)=15=G
or
A(0,0,1)=A(15)

IN FOUR DIMENSIONS

ARRAY (row, column, plane, cube)=A(I,J,K,L)=A(G)

Assign the values for IMAX, JMAX, KMAX; a method similar to the one used above yields

G=I+(IMAX*J)+(IMAX*JMAX*K)+(IMAX*JMAX*KMAX*L)

This process can be extended indefinitely to n-dimensions!

Example 1:

```
*FOR J=0,4; TYPE %2,I; FOR I=0,2; TYPE J+5*I

=  0=   5=  10
=  1=   6=  11
=  2=   7=  12
=  3=   8=  13
=  4=   9=  14*
```

Example 2:

```
C-FOCAL , 8/68

01.05 TYPE "ENTER 3 ROWS AND 4 COLUMNS OF NUMBERS."!
01.10 FOR J=0,2; TYPE !; FOR K=0,3; ASK NO(J+3*K)
01.15 SET MAX=NO(0)
01.20 FOR J=0,2; FOR K=0,3; DO 02.00
01.25 TYPE !, "LARGEST NUMBER IS ", MAX; QUIT

02.05 IF (MAX-NO(J+3*K)) 2.10; RETURN
02.10 SET MAX=NO(J+3*K); RETURN
*
*GO
ENTER 3 ROWS AND 4 COLUMNS OF NUMBERS.

:0 :5 :8 :9
:1 :2 :3 :4
:9 :8 :7 :6
LARGEST NUMBER IS =    9.00000*
*
*GO
ENTER 3 ROWS AND 4 COLUMNS OF NUMBERS.

:A :B :C :D
:A :B :C :D
:A :B :C :D
LARGEST NUMBER IS =    4.00000*
*
*GO
ENTER 3 ROWS AND 4 COLUMNS OF NUMBERS.

:A :B :C :D
:4 :3 :2 :1
:A :4 :C :2
LARGEST NUMBER IS =    4.00000*
*
```

Example 3:

```
C-FOCAL , 8/68

01.02 TYPE !"ROUTINE TO SOLVE MATRIX EQ. AX=B FOR X"!
01.04 ASK "ENTER DIMENSION OF A, THEN
01.05 TYPE !"ENTER COEFF'S A(J,K)...A(J,N) AND B(J)"!
01.10 ASK L,!; SET N=L-1; SET I=-1
01.11 FOR K=0,N; SET R(K)=K+1
01.12 FOR J=0,N; TYPE !; FOR K=0,L; ASK A(J+L*K)
01.14 SET M=1E-6
01.16 FOR J=0,N; FOR K=0,N; DO 4.0
01.17 SET R(P)=0
01.18 FOR K=0,L; SET A(P+L*K)=A(P+L*K)/M
01.20 FOR J=0,N; DO 5.0
01.22 SET I=I+1
01.23 IF (I-N) 1.14,1.26,1.14
01.26 FOR J=0,N; FOR K=0,N; DO 7.0
01.28 FOR K=0,N; TYPE !%2,"X("K,")  ",%8.05,X(K)
01.29 TYPE !!; QUIT

04.05 IF (R[J]) 0,4.3,4.1
04.10 IF (FABS<A[J+L*K]>-FABS[M]) 4.3;
04.20 SET M=A(J+L*K)
04.22 SET P=J; SET Q=K
04.30 RETURN

05.10 IF (J-P) 5.2,5.4,5.2
05.20 SET D=A(J+L*Q)
05.30 FOR K=0,L; SET A(J+L*K)=A(J+L*K)-A(P+L*K)*D
05.40 RETURN

07.10 IF (1E-6-FABS[A<J+L*K>]) 7.2; RETURN
07.20 SET X(K)=A(J+L*L)
*
*GO

ROUTINE TO SOLVE MATRIX EQ. AX=B FOR X
ENTER DIMENSION OF A, THEN
ENTER COEFF'S A(J,K)...A(J,N) AND B(J)
:3

:1  :2  :3  :4
:4  :3  :2  :1
:1  :4  :3  :2
X(=  0)  =    0.00000
X(=  1)  =-   1.00000
X(=  2)  =    2.00000

*
```

Demonstration Dice Game

Sooner or later, people who have access to a computer will try to "match brains" with it or use it for their own enjoyment. Such pastimes are usually keyboard oriented and FOCAL lends itself nicely to these ends. The following example uses the random number generator, FRAN( ), to produce dice combinations, plus IF logic to check bets and winning combinations.

Lines beginning with a C indicate that the line is to be treated as a comment and is not to be interpreted or executed. If a comment statement is preceded by a statement number, the line is stored as part of the program but does not affect the program logic.

The random number generator must be modified for use with statistical or simulation programs to achieve true randomness. However, it is sufficiently random in its present form for most applications.

NOTE
We naturally cannot assume any responsibility for the use of this or any similar routines.

```
C-FOCAL , 8/68

01.10 SET B=0;TYPE !!"DICE GAME"!,"HOUSE LIMIT OF $1000
01.13 TYPE ".  MINIMUM BET IS $1"!!
01.20 ASK "YOUR BET IS"A;IF (1000-A) 3.1
01.22 IF (A-1)3.4,1.26,1.26
01.26 IF (A-FITR(A))3.5,1.3,3.5
01.30 ASK M;DO 2;SET D=C;DO 2;TYPE " ";SET D=D+C
01.32 IF (D-7)1.42,3.2,1.42
01.40 IF (D-2)1.5,3.3,1.5
01.42 IF (D-11)1.4,3.2,1.4
01.50 IF (D-3) 1.6,3.3,1.6
01.60 ASK M;DO 2;SET E=C;DO 2;TYPE " ";SET E=E+C
01.72 IF (E-7) 1.74,3.3,1.74
01.74 IF (E-D)1.6,3.2,1.6

02.10 SET C=FITR(10*FABS(FRAN()));IF (C-6)2.2,2.2,2.1
02.20 IF (C-1)2.1;TYPE %1," "C;RETURN

03.10 TYPE "HOUSE LIMITS ARE $1000"!!; GOTO 1.2
03.20 S B=B+A;T %7,!"YOU WIN. YOUR WINNINGS ARE      "B,!!;GOTO 1.2
03.30 S B=B-A;T %7,!"SORRY, YOU LOSE. YOUR WINNINGS ARE "B,!!;GOTO 1.2
03.40 TYPE "MINIMUN BET IS $1"!!;GOTO 1.2
03.50 TYPE "NO PENNIES, PLEASE"!!;GOTO 1.2
*
*C  ONCE YOU PLACE A BET, FOLLOW THE OTHER COLONS WITH A
*C   CARRIAGE RETURN TO INDICATE THE COMMAND "ROLL THE DICE".
*
```

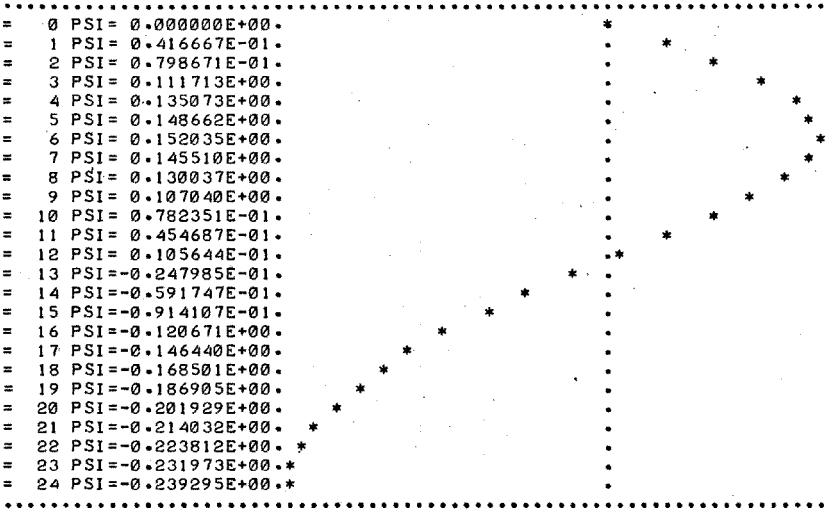
```
*GO


DICE GAME
HOUSE LIMIT OF $1000.  MINIMUM BET IS $1

YOUR BET IS:.50 MINIMUN BET IS $1

YOUR BET IS:15 :
  = 6   = 3 :
  = 1   = 4 :
  = 4   = 5
YOU WIN. YOUR WINNINGS ARE       =       15

YOUR BET IS:5 :
  = 2   = 2 :
  = 6   = 1
SORRY, YOU LOSE. YOUR WINNINGS ARE =      10

YOUR BET IS:3 :
  = 6   = 5
YOU WIN. YOUR WINNINGS ARE       =       13

YOUR BET IS:  I'LL QUIT WHILE I'M AHEAD. THANKS!
```

Schroedinger Equation Solver

The program is designed to aid the user in searching for possible energy-states of an electron in a potential well. This is one of the most complex equations yet written in FOCAL. It calculates and plots the energy levels of an electron within specified boundary conditions.

```
C-FOCAL , 8/68

01.01 T !,"SCHROEDINGER EQUATION SOLVER -",!
01.02 T !,"          -DELSQUARED PSI + AX * PSI = E * PSI",!!
01.03 A "TILTED SQUARE WELL PROBLEM WITH WIDTH",X0,!
01.08 A "WELL TILT SLOPE A",A1,!,"TRIAL ENERGY E",B1,!
01.09 A "NUMBER OF STEPS",NT,!
01.11 S VF=0; S SL=1
01.70 S P(0)=0; S DX=X0/NT; S P(1)=SL*DX; S R0=0
01.75 S VF=0
01.80 S P0=0
01.90 F N=0,1,NT-2; D 6
01.93 T !,"PSI ZEROS"%2.0, P0
01.95 GOTO 7.02

05.10 T !,%3.0, PX," PSI",%,P(PX),"."
05.20 S PZ=FITR(PM*SC); S PE=FITR<(P[PX]+PM)*SC>
05.30 F X=1,1,PZ; T " "
05.40 T ".",#; F X=1,1,PE+24; T " "
05.50 T "*"; R

06.10 S P(N+2)=<(-B1+A1*DX*[N+1])*DX↑2+2>*P(N+1)-P(N)
06.20 I (NT-N-2) 12.90,6.9,6.3
06.30 S RB=P(N+2)*P(N+1); I (RB) 6.4,6.4,6.9
06.40 S P0=P0+1; R
06.90 CONTINUE

07.02 S CF=(P<NT>/P<1>)↑2; T "  CONV IND"%, CF
07.05 A "  NEW E?"NY
07.07 I (NY-9) 7.9,7.08,7.9
07.08 I (VF) 7.09,7.8,7.09
07.09 I (CF-100) 7.1,7.1,7.8
07.10 S R2=P(NT)*VF; I (R2) 7.73,7.80,7.85
07.73 S DB=-0.5*DB; GOTO 7.85
07.80 S DB=0.1
07.85 S B1=B1*(1+DB); T B1; S VF(NT); G 1.80
07.90 DO 14; GOTO 12.01

12.01 T !,!,"EIGEN E"B1; S HP=B1/(A1*X0)
12.20 T "  EN/MAX POT"HP,!
12.90 QUIT

14.10 S PM=0; S PP=0; F PX=1,1,NT; D 15
14.20 S PS=PM+PP; S SC=45/PS
14.30 T !!!; F PX=1,1,70; T "."
14.40 F PX=0,1,NT; D 5
14.50 T !;F PX=1,1,70; T "."
14.60 T !!; R

15.10 I (P[PX]) 15.2,15.9,15.5
15.20 I (PM+P<PX>) 15.3,15.4,15.4
15.30 S PM=FABS(P[PX])
15.40 RETURN
15.50 I (P<PX>-PP) 15.9,15.9,15.6
15.60 S PP=P(PX)
15.90 RETURN
*
```

```
*GO

SCHROEDINGER EQUATION SOLVER -

          -DELSQUARED PSI + AX * PSI = E * PSI

TILTED SQUARE WELL PROBLEM WITH WIDTH:1

WELL TILT SLOPE A:50

TRIAL ENERGY E:50

NUMBER OF STEPS:24

PSI ZEROS=  1  CONV IND= 0.329830E+02  NEW E?:Y

```




```
EIGEN E= 0.500000E+02  EN/MAX POT= 0.100000E+01
*
```

SUMMARY OF COMMANDS, OPERATIONS, AND FUNCTIONS

Command	Abbreviation	Example of Form	EXPLANATION
ASK	A	ASK X,Y,Z	FOCAL types a colon for each variable; the user types a value to define each variable.
COMMENT	C	COMMENT	If a line begins with the letter C, the remainder of the line will be ignored.
CONTINUE	C	C	Dummy line.
DO	D	DO 4.1	Execute line 4.1; return to command following DO command,
		DO 4.0	Execute all group 4 lines; return to command following DO command, or when a RETURN is encountered.
ERASE	E	ERASE	Erases the symbol table.
		ERASE 2.0	Erases all group 2 lines.
		ERASE 2.1	Deletes line 2.1.
		ERASE ALL	Deletes all user input.
FOR	F	FOR i=x,y,z;(commands)	Where the command following is executed at each new value.
		FOR i=x,z;(commands)	x=initial value of i
			y=value added to i until i is greater than z.
GO	G	GO	Starts indirect program at lowest numbered line number.
GO?	G?	GO?	Starts at lowest numbered line number and traces entire indirect program until another ? is encountered, until an error is encountered, or until completion of program.
GOTO	G	GOTO 3.4	Starts indirect program (transfers control to line 3.4). Must have argument.
IF	I	IF (X)Ln, Ln, Ln IF (X)Ln, Ln; (commands)	Where X is a defined identifier, a value, or an expression, followed by three line numbers.
		IF (X)Ln; (commands)	If X is less than zero, control is transferred to the first line number.
			If X is equal to zero, control is to the second line number.
			If X is greater than zero, control is to the third line number.
MODIFY	M	MODIFY 1.15	Enables editing of any character on line 1.15 (see below).
QUIT	Q	QUIT	Returns control to the user.
RETURN	R	RETURN	Terminates DO subroutines, returning to the original sequence.
SET	S	SET A=5/B*C;	Defines identifiers in the symbol table.
TYPE	T	TYPE A+B—C;	Evaluates expression and types out = and result in current output format.
		TYPE A—B, C/E;	Computes and types each expression separated by commas.
		TYPE "TEXT STRING"	Types text. May be followed by ! to generate carriage return-line feed, or # to generate carriage return.
WRITE	W	WRITE WRITE ALL	FOCAL types out the entire indirect program.
		WRITE 1.0	FOCAL types out all group 1 lines.
		WRITE 1.1	FOCAL types out line 1.1.

To set output format,	TYPE %x.y	where x is the total number of digits, and y is the number of digits to the right of the decimal point.
	TYPE %6.3, 123.456	FOCAL types: = 123.456
	TYPE %	Resets output format to floating point.
To type symbol table,	TYPE $	Other statements may not follow on this line

To input from high-speed paper tape reader,

**	The second * was typed by the user. Input is from the high-speed reader until occurrence of next *. FOCAL types * for each line number read in from the reader.
1.10;	User typed 1.10*;. Input is taken from the high-speed reader until occurrence of next *.

Modify Operations

After a MODIFY command, the user types a search character, and FOCAL types out the contents of that line until the search character is typed. The user may then perform any of the following optional operations.

1. Type in new characters. FOCAL will add these to the line at point of insertion.
2. Type a CTRL/L. FOCAL will proceed to the next occurrence of the search character.
3. Type a CTRL/BELL. After this, the user may change the search character.
4. Type RUBOUT. This deletes characters to the left, one character for each time the user strikes the RUBOUT key.
5. Type ← . Deletes the line over to the left margin, but not the line number.
6. Type RETURN. Terminates the line, deleting characters over to the right margin.
7. Type LINE FEED. Saves the remainder of the line from the point at which LINE FEED is typed over to the right margin.

The Trace Feature

Special Character	Example of Form	Explanation
?	?...? or ?...	Those parts of the program enclosed in question marks will be printed out as they are executed. If only one ? is inserted, the trace feature becomes operative, and the program is printed out from that point until another ? is encountered, until an error is encountered, or until program completion.

Summary of Functions

Square Root	FSQT(x)	where x is a positive number or expression greater than zero.
Absolute Value	FABS(x)	FOCAL ignores the sign of x.
Sign Part	FSGN(x)	FOCAL evaluates the sign part only, with 1.0000 as integer.
Integer Part	FITR(x)	FOCAL operates on the integer part of x, ignoring any fractional part.
Random Number Generator	FRAN( )	FOCAL generates a random number.
Exponential Function (e^x)	FEXP(x)	FOCAL generates e to the power x. (2.71828^x)
Sine	FSIN(x)	FOCAL generates the sine of x in radians.
Cosine	FCOS(x)	FOCAL generates the cosine of x in radians.
Arc Tangent	FATN(x)	FOCAL generates the arc tangent of x in radians.
Logarithm	FLOG(x)	FOCAL generates $\log_e(x)$.
Analog-to-Digital	FADC(n)	FOCAL reads from an analog-to-digital channel, the value of the function is that integer reading.

Scope Functions

FDIS(y)	Displays y coordinate on scope and intensifies x-y point.
FDXS(x)	Displays x coordinate on scope.

Other functions are available from DEC for use with such peripherals as incremental plotter, card reader, etc.

Special Characters

1. Mathematical operators:
   - ↑ Exponentiation
   - * Multiplication
   - / Division
   - + Addition
   - − Subtraction
2. Control characters:
   - % Output format delimiter
   - ! Carriage return and line feed
   - # Carriage return
   - $ Type symbol table contents
   - ( ) Parentheses
   - [ ] Square brackets
   - < > Angle brackets

     } (mathematics)
   - " " Quotation marks (text string)
   - ? ? Question marks (trace feature)
   - * Asterisk(s) (high-speed reader input)
3. Terminators:
   - SPACE key (names)
   - RETURN key (lines)
   - ALTMODE key (with ASK statement)

     } (nonprinting)
   - , Comma (expressions)
   - ; Semicolon (commands and statements)

Error Diagnostics

Error messages are typed in the following format:

?nn.nn @ nn.nn (error code @ line number)

Error Code	Meaning
?00.00	Manual start from console
?01.00	Interrupt from keyboard via CTRL/C
?01.35	Group zero is an illegal line number
?01.43	Illegal step or line number
?01.89	GOTO not used as *one* word or bad argument in IF
?01.;2	Line number too large
?01.;3	Double periods in line number
?02.48	Nonexistent line referenced by DO
?02.63	Nonexistent group referenced by DO
?02.81	Storage filled by push-down list
?03.09	Nonexistent line used or a tight loop
?03.31	Illegal command used
?04.07	No space after IF or illegal format
?04.35	Left of = in error for FOR or SET
?04.48	Excess right parenthesis
?04.56	Illegal terminator in FOR
?05.63	Bad argument to MODIFY
?06.13	Illegal use of function or number
?06.64	Storage filled by variables
?07.14	Operator missing or double E
?07.34	No operator before parenthesis
?07.<0	Double operators
?07.;1	No argument given after function call
?07.;8	Illegal function name
?08.50	Parentheses do not match
?09.16	Bad argument in ERASE
?09.50	Maximum group number exceeded
?11.20	Input buffer has overflowed
?12.83	Storage filled by text
?20.41	Logarithm of zero requested
?23.35	Literal number is too large
?26.91	Negative exponent used
?26.96	Exponent is too large
?28.58	Division by zero requested
?30.48	Imaginary square roots required
?31.<7	Illegal character or unavailable command or unavailable function used

NOTE:
The above diagnostics apply only to the version of FOCAL, 8/68, issued on tape DEC-08-AJAC-PB.

ESTIMATING THE LENGTH OF USER'S PROGRAM

FOCAL requires five words for each identifier stored in the symbol table, and one word for each two characters of stored program. This may be calculated by

$$5s + \frac{c}{2} \cdot 1.01 = \text{length of user's program}$$

where s = Number of identifiers defined
c = Number of characters in indirect program

If the total program area or symbol table area becomes too large, FOCAL types an error message.

FOCAL occupies core locations $1\text{-}3200_8$ and $4600_8\text{-}7576_8$. This leaves approximately 700_{10} locations for the user's program (indirect program, identifiers, and push-down list). The extended functions occupy locations 4600-5377. If the user decides not to retain the extended functions at load-time, there will be space left for approximately 1100_{10} characters for the user's program.

The following routine allows the user to find out how many core locations are left for his use.

```
*FOR I=1,300; SET A(I)=I
?06.64                                  (disregard error code)
*TYPE %4, I*5, " LOCATIONS LEFT "
=   720 LOCATIONS LEFT   *
```

CALCULATING TRIGONOMETRIC FUNCTIONS IN FOCAL

Function	FOCAL Representation	Argument Range	Function Range
Sine	FSIN(A)	0<|A|<10↑4	0<|F|<1
Cosine	FCOS(A)	0<|A|<10↑4	0<|F|<1
Tangent	FSIN(A)/FCOS(A)	0<|A|<10↑4 |A|=(2N+1)π/2	0<|F|<10↑6
Secant	1/FCOS(A)	0<|A|<10↑4 |A|=(2N+1)π/2	1<|F|<10↑6
Cosecant	1/FSIN(A)	0<|A|<10↑4 |A|=2Nπ	1<|F|<10↑6
Cotangent	FCOS(A)/FSIN(A)	0<|A|<10↑4 |A|=2Nπ	0<|F|<10↑440
Arc sine	FATN(A/FSQT(1−A↑2)	0<|A|<1	0<|F|<π/2
Arc cosine	FATN(FSQT(1−A↑2)/A)	0<|A|<1	0<|F|<π/2
Arc tangent	FATN(A)	0<A<10↑6	0<F<π/2
Arc secant	FATN(FSQT(A↑2−1))	1<A<10↑6	0<F<π/2
Arc cosecant	FATN(1/FSQT(A↑2−1))	1<A<10↑300	0<F<π/2
Arc cotangent	FATN(1/A)	0<A<10↑615	0<F<π/2
Hyperbolic sine	(FEXP(A)−FEXP(−A))/2	0<|A|<700	0<|F|<5*10↑300
Hyperbolic cosine	(FEXP(A)+FEXP(−A))/2	0<|A|<700	1<F<5*10↑300
Hyperbolic tangent	(FEXP(A)−FEXP(−A))/ (FEXP(A)+FEXP(−A))	0<|A|<700	0<|F|<1
Hyperbolic secant	2/(FEXP(A)+FEXP(−A))	0<|A|<700	0<F<1
Hyperbolic cosecant	2/(FEXP(A)−FEXP(−A))	0<|A|<700	0<|F|<10↑7
Hyperbolic cotangent	(FEXP(A)+FEXP(−A))/ (FEXP(A)−FEXP(−A))	0<|A|<700	1<|F|<10↑7
Arc hyperbolic sine	FLOG(A+FSQT(A↑2+1))	−10↑5<A<10↑600	−12<F<1300
Arc hyperbolic cosine	FLOG(A+FSQT(A↑2−1))	1<A<10↑300	0<F<700
Arc hyperbolic tangent	(FLOG(1+A)−FLOG(1−A))/2	0<|A|<1	0<|F|<8.31777
Arc hyperbolic secant	FLOG((1/A)+FSQT((1/A↑2)−1))	0<|A|<1	0<F<700
Arc hyperbolic cosecant	FLOG((1/A)+FSQT((1/A↑2)+1))	0<|A|<10↑300	0<|F|<1400
Arc hyperbolic cotangent	(FLOG(X+1)−FLOG(X−1))/2	1<A<10↑616	0<F<8

Chapter 10

PDP-8 Computers in the Sciences

As a leader in the field of small, general purpose computers, Digital is well aware of the significant impact of small computers on the sciences, particularly in the way scientists work in their laboratories and in the way engineers design and build instruments, machines, and control systems. Scientists use PDP-8 family computers as personal, powerful tools; engineers build them in as components. More computers in the PDP-8 family have been sold for these applications than any other computer in the world. Designed particularly for online applications, PDP-8 computers embody design features that make them simple and straightforward to interface and program for any application.

OFFLINE AND ONLINE USES

Before we discuss in detail the scientific applications of computers in the PDP-8 family, let us define two basic concepts in this regard, *offline* and *online*.

A computer used to analyze data that has previously been recorded is said to be used *offline*. In contrast, a computer used by a scientist to directly collect, sample, and analyze data while an experiment is in process is said to be an *online* computer.

There are two types of online usage. In the first type, the computer—usually a small model—is physically located in the laboratory. In the second type, where the amount of data to be processed is too large for a small computer to handle, the scientist connects his experiment to a larger, remotely located computer; this larger computer is usually capable of handling several users simultaneously, i.e., on a time-shared basis. A variation of this second type of online usage is to connect a

small computer situated in the laboratory to a larger, time-shared computer at a remote location.

Offline use of a computer reduces computation time as compared to manual calculations, but offers few other advantages. On the other hand, an online computer permits the scientist to participate to a greater degree in his experiment. The results which he constantly receives during the course of his experiment allow him to make certain choices on the spot. He may call for a more intensive investigation of interesting results, extend the scope of his experiment on the basis of developing information, or even instruct the computer to look for and investigate interesting phenomena automatically during the experiment.

DATA COLLECTION

Frequently, an experiment involves the measuring of a "signal" that is continuous, although it may vary in amplitude, frequency, or both. This signal must be converted into an electronic signal before it can be processed by any type of computer, whether analog or digital; this conversion is usually performed by a transducer or a potentiometer, and, if necessary, the signal is then amplified by a high-gain amplifier. In the case of digital computers, the analog signal thus produced must be further converted by connecting it to the input channel of an analog-to-digital (A/D) converter to digitize the signal. The A/D converter changes a signal varying continuously with time to a series of discrete numeric values. This conversion, known as "sampling a signal," can be performed at many points in time during the experiment.

In addition to converted analog inputs as described above, a digital computer can accept digital or contact-closure information. Process-related on/off signals such as alarms, limit indications, and selector switch settings can then be correlated with the analog data. Direct digital transducers (e.g., digital shaft position encoders) can interface to digital input channels without the intermediate A/D conversion.

Once the inputs are brought into the computer, the data can be verified, manipulated, formatted, and displayed in a variety of ways. The arbitrary voltage or frequency readings from the A/D converter can be compared against limits and converted to meaningful units (e.g., 3.26 millivolts from one sensor may correspond to +132°F; the same voltage from another sensor might represent 84 gallons per minute). Through programming, input data can be compared, sorted, arranged in labeled columns along with the time of reading, and displayed on a

Teletype printer or cathode ray tube. Data can be accumulated and then written out on some peripheral medium for later analysis offline. Communications linkages allow data to be sent to remote processors as it is collected.

DATA ANALYSIS

The computer can be used to analyze collected data to obtain arithmetic means, medians, variances, deviations, correlations, regressions, factors, etc., between one group of responses and another. Large statistical evaluations can be made either online or offline, depending upon their complexity. A set of data can be compared with a previously collected data set, a user determined standard, or a component within the sample itself. The user can specify the number of places of accuracy he requires and have all calculations performed to this limit.

DATA DISPLAY

Using an online computer, the experimenter can display his data while the experiment is running. Data can be displayed in a meaningful format such as columnar listings on the line printer or as graphs on an incremental plotter or cathode ray tube oscilloscope.

COMPUTER INSTRUMENTATION CONTROL

To illustrate the automation of instruments in the lab by a computer, the following elemental example of data collection, analysis, and instrument control is given. The application consists of controlling the pressure of a liquid within a pipeline, as shown in Figure 10-1. There are four valves available for controlling the flow. A pressure gauge transmits the current pressure. Assuming that the pressure is to be always kept below 20 lb/sq.in., the flow chart in Figure 10-1 shows the processing involved.

LABORATORY AUTOMATION

In laboratory work, the computer offers many advantages besides just those of faster numeric computation. It can be programmed to make judgments based on data as it is received. For example, it can skim areas of little interest and concentrate on more promising areas. The results of a general scan can be viewed by the experimenter while a more detailed scan of some particular area is being performed. Also, procedures can be modified or even terminated prematurely as the progress of the experiment dictates.

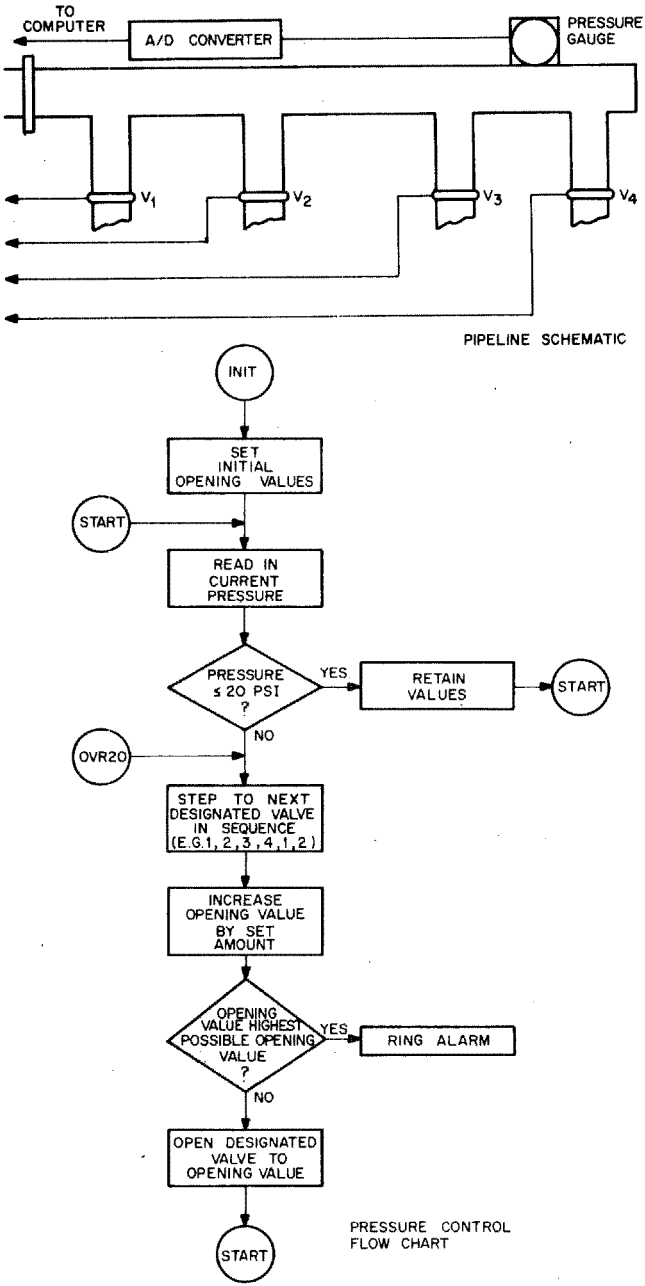


Figure 10-1. Example of Computer Instrumentation Control

Such flexibility is not possible with fixed-program devices; these devices cannot vary their operations even though meaningful data is not being collected. An instrument under the control of a computer, on the other hand, can be made to find the proper areas of study. In addition, a readout device may be used to inform the experimenter of what is happening and signal him immediately of any unusual results.

Computerizing instrumentation is a major step forward in laboratory automation, but the loop between input and results can be closed even further. For example, in the clinical laboratory, personnel shortages and increasing hospital populations have placed an unprecedented burden on the pathologist. This burden has been relieved by a clinical computer, which can receive testing instructions, start and stop instrumentation, analyze the data, log the data in a precise format, correlate the data with other physiological information, send the correct information to a ward, coordinate with other departments, and keep accurate and complete records.

The PDP-8 family of computers has become increasingly important in this field of analytical instrumentation in the laboratory. Currently, PDP-8 computers are being used in a variety of applications because of their ability to process large numbers of chemical analyses repetitively on samples with a standard matrix and a relatively narrow variation between samples. A sampling of these applications is given below.

Gas-Liquid Chromatography

The GLC-8 computer-based system can simultaneously handle 20 laboratory gas or liquid chromatographs. The system automatically detects peaks and shoulders, calculates peak areas and peak retention times, allocates peak overlap areas, corrects for baseline shift and applies response factors. Using stored analysis tables as internal standards, peaks can be identified and the percentage of each component determined. GLC-8 prints out a complete analysis report, including retention time, peak area, component percentage and tolerance. Analysis time is saved, human errors reduced, and efficiency of instrument use greatly increased. Types and sampling rates of chromatographs can be mixed, analysis tables can be calibrated and updated automatically, and instantaneous analysis reports to remote locations can be provided. GLC-8 is designed primarily for use with gas or liquid chromatographs, but may be used with other analytical instruments, such as amino-acid analyzers, that use the same method of analysis.

Pulse Height Analysis

PHA-8 is a complete computer system for single- or multi-parameter analysis. It can gather, store, display, and analyze energy or time-of-

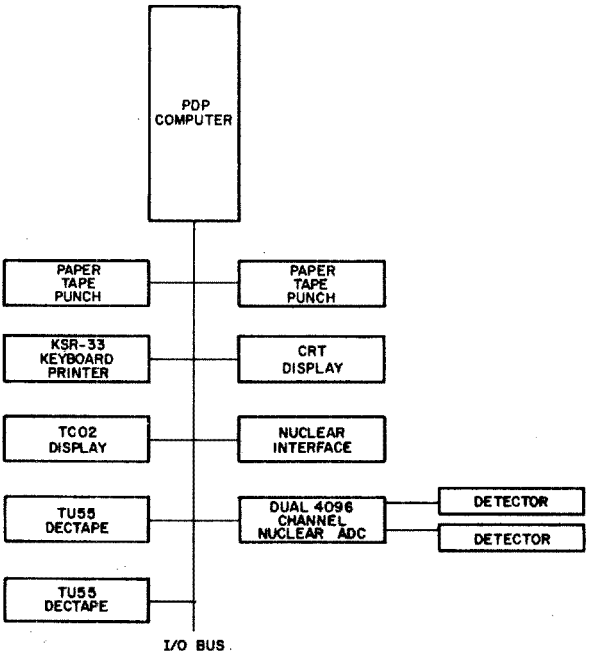


Figure 10-2. Computer-Based Pulse Height Analysis System

flight spectra and record the results on a variety of output devices. Using analog-to-digital converters, PHA-8 can be configured with 4,096 to 32,768 channels for spectra storage. With optional mass storage devices, systems can be set up to run multiple experiments automatically, store results from each experiment, and retrieve them upon command. Background counts can be subtracted, spectra compared, one spectrum subtracted from another, and energy calculations performed. Oscilloscope subsystems may be used to generate contours, isometrics, and area-of-interest displays. The computer can also be used to rotate axes, integrate areas between markers and under peaks, calibrate energy versus channel number, and fit curves.

Mass Spectroscopy

The extremely high data rates required in high resolution mass spectroscopy are easily handled by PDP-8 systems. With sampling rates above 20KHz, the computer can scan 500 or more peaks with from 20 to 50 points per peak. The computer can also be used to control the sweep of the magnet, providing lower data rates and increased accuracy when desired. Centroid calculations, to determine peak positions accurately, may be performed while the scan is taking place.

In low and medium resolution mass spectroscopy, where the analyst has much slower data rates, the computer can generate a mass map to determine integer mass units. The computer can be connected to magnetic deflection, time-of-flight (TOF), or quadropole mass spectrometers to observe ion current versus voltage potential.

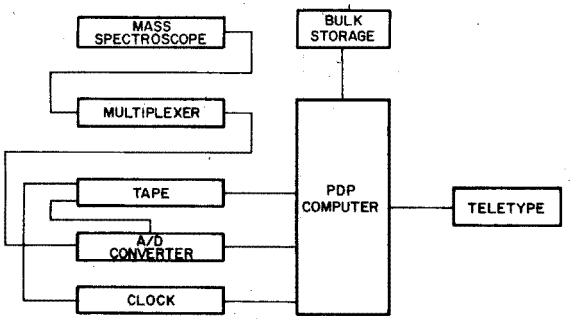


Figure 10-3. Computer-Based System for Mass Spectroscopy

Nuclear Magnetic Resonance (NMR) Spectroscopy

Adding a PDP-8 computer-based system to an NMR spectrometer significantly benefits the analyst in the collection, analysis, and presentation of the data. In the data collection phase, the computer system provides for more capability than a hard-wired signal averager, allowing online selection and examination of key areas of interest and improved sensitivity. In the analysis phase, corrections may be applied, and Fourier analyses performed directly without prior storage or readout of the data. The computer can also be used to reduce the spectra to compact form so that they may be stored, retrieved, and compared with other data. Spin simulation may be calculated on relatively complex spectra.

Absorption Spectroscopy

PDP-8 computer-based systems decrease noise by mathematical smoothing, locate the position of maximum absorption, and automatically record the absorption value. With the computer, the analyst can easily apply background and frequency corrections: 100% and zero line, wave number, ordinate (percentage transmittance), slit function, and tracking error. The computer permits the analyst to determine integrated band densities, resolve poorly separated bands, and synthesize hypothetical spectra—all while the sample is being run. The computer also performs difference analyses (subtracting one spectrum from another), automatically replots spectra, tabulates peak data, and identifies peaks by searching a spectrum library in the computer's memory.

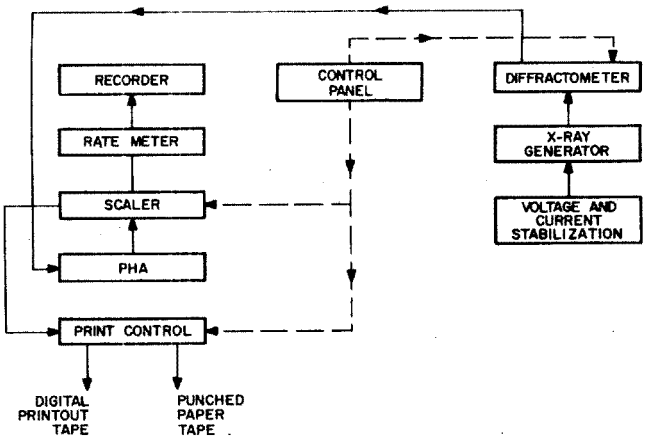


Figure 10-4. Computer-Based X-Ray Diffraction System

X-Ray Diffraction

A PDP-8 computer-based system enables the analyst to obtain more reliable data by allowing him to correct errors caused by accidental changes in experimental setups; it also corrects for occasional slippage of the sample as it is held in the path of the X-ray. To do this, the computer is directed to analyze the data as it is received and to com-

pare it with previously established standards stored in memory. New measurements that deviate from the stored information generate new input control signals used in positioning the crystal.

Time-of-Flight Analysis

PDP-8 systems are available for performing time-distribution studies for three types of experimentation: velocity measurements to determine energies, life-time measurements to determine identities of unstable particles, and time-coincidence measurements to seek correlations between events occurring near each other in time. All systems provide scalars that allow them to collect data at high neutron counting rates, to process several events during each cycle of the accelerator, and to eliminate cumbersome deadtime corrections. Time-measuring sections are available for low-energy particle experiments with channel widths down to 0.1 microsecond. For high-energy applications, there are total systems that include commercially available time-measuring sections.

OTHER APPLICATIONS

The efficiency, accuracy and speed of any instrumental analysis can be significantly increased by including a PDP-8 computer in the experimental system to control the sequence of events and record and analyze the accumulated data. A PDP-8 computer can be easily interfaced to a wide range of analytical instruments, including tensile testers, electron microprobes, ultracentrifuges, microdensitometers, liquid scintillation counters, and emission spectrographs. Extensive uses in the physical, life, and natural sciences have included the following applications.

Field	Typical Applications
PHYSICS	High- and low-energy studies. Coupled to flying spot scanners, nuclear detectors and counters, mass spectrometers, bubble chambers, X-ray diffractometers, and accelerators.
NUCLEAR REACTIONS	Used at a radiation lab online for readout, display, and graph plotting. Can monitor a dual system at a nuclear reactor site.

Field	Typical Applications
GAMMA-RAY SPECTROSCOPY	Used at a testing station in conjunction with an oscilloscope and light pen, receiving gamma-ray spectra from high resolution detectors and A/D converters.
ELECTROCHEMISTRY	Used to investigate electrochemical phenomena at the electrode-electrolyte interface and to determine the chemical composition of steel samples.
CRYSTALLOGRAPHY	Sets individual crystal angles of diffractometer, records X-ray reflection data, and plots crystal structure on CRT.
NEUROLOGY	Used to study and determine power spectra of finger and hand tremors, interactions of communities of neurons, time-interval histograms from discharges in cerebellar units, and the effect of electrical stimulation upon activity in the human ulnar nerve. In brain mapping, single cell activity within the brain of a monkey is recorded and impedance measurement data from the brain is digitized to map brain structures on a scope.
CARDIOLOGY	Used to average and store changes in transmembrane voltage of cardiac muscle cells after external electrical stimulation, to study the hydrodynamics and transmission characteristics of the mammalian arterial system, to evaluate by Fourier analysis complex pressure pulses in the aorta of mammals, to measure microcirculation, to evaluate velocity profiles and viscuous losses in blood vessel walls during pulsatile flow, and to evaluate pressure flow relations and propagation of pressure pulses in arterial trees of a live subject.
CLINICAL MEDICINE	Used to perform routine data processing, record handling, and statistical research, to analyze ECG's to aid the brain surgeon

Field	Typical Applications
	in the operating room in locating the cerebral area in which to work, to study Parkinson's disease, and to aid in endocrine analysis.
IMAGE PROCESSING	Used to analyze photos of cells and chromosomes, to estimate karyotypes, to perform density measurements on autoradiographs, to correlate photos of vocal cord action with the sounds made, and to perform photogrammetric analyses.
BEHAVORIAL SCIENCES	Used to study interresponse times and the social behavior of primates, to study the operant conditioning of pigeons, to control and monitor schedules of reinforcement, to determine auditory, tactile, and visual responses in the human, and to perform online psychological and sequence pattern tests.
EARTH SCIENCES	Used in seismology and paleontology to identify small teleseismic disturbances, to correlate multiband images from airborne or spaceborne cameras, to combine multiple polarization radar images, to group and separate fossils, to aid in undersea drilling by scanning and analyzing electrical signals from sensors located on the rig floor, and to compare and correlate oil-field brines by means of pattern analysis.
OCEANOGRAPHY	Used to process data on sea conditions, to assist in navigation and ship control, to perform oceanographic surveys, to determine the biomass in the oceans, to measure and correlate simultaneously the total magnetic intensity of the earth's field, the gravity, and the sea surface temperature at a specific location while making a profile of the subbottom structure.
ATMOSPHERIC SCIENCES	Used to analyze air polution, to aid in micrometeorological studies, and to analyze fluid mechanic interactions.

Field	Typical Applications
GEOPHYSICS	Used to investigate the ionosphere, to process telemetry data from scientific instrumentation in sounding rockets and balloons, and to aid in research of solar phenomena.
SPACE SCIENCES	Used to check out guidance and control equipment for the United States' largest rockets, to perform online testing of space satellite hardware, to decode data from orbiting satellites, to check out scientific payloads, to develop a solar wind spectrometer, to process photographic and other signals from lunar and planetary space probes, to handle radio telescope control systems and collect and reduce radio astronomy data, and to automate optical telescope operations.

AN EXAMPLE

To illustrate the adaptation of a PDP-8 computer to a scientific application, we have selected as an example the collecting and preprocessing of physiological signals by a computer for subsequent analysis by another computer.[1]

A PDP-8 program was written to receive and preprocess as many as eight physiological signals simultaneously from a monitored patient (or individual signals from up to eight patients). Inputs are analog signals, which are sampled 500 times/sec. and digitized. The program performs a code recognition of each signal and temporarily stores this and subsequent data on a drum until the number points required for the analysis have been accumulated. Concurrently with other instructions, the data break facility allows short blocks of data to be written on the drum or long blocks read back into core. The long-block data (referred to as a "lead"), needed to perform the analysis, is relayed via

[1] The information in this section was extracted from a paper in "DECUS Proceedings, Fall, 1967" submitted by Miss Maxine L. Paulsen of the Medical Systems Development Laboratory, Washington, D.C.

an interface to the other computer, which transfers the data to magnetic tape. The tape is then used as input to a third computer, which consolidates, analyzes, and interprets the signals for each patient. The operations of all three computers are carried on simultaneously once the first input tape is written.

The Preprocessing Hardware

The equipment used in this application is a DIGITAL Preprocessing System F, which includes the following components.

A 12K PDP-8 Processor
One Type DM01 Data Channel Multiplexer
One Type RM08E Serial Magnetic Drum
One Type 139E General Purpose Multiplexer Control
One Type 138E General Purpose A/D Converter
One Programmable Real Time Clock
One Signal Input-Routing Network Package (includes 24 Type A103 Multiplexer Switches and 64 Amplifier Mounting Boards)
One High-Speed 2-Way Interface to the other computer
One Interrupt and Skip Logic feature with 8 Program Flags

Also included in the system, but not used in this particular application, were one Type TC01 DECtape Control Unit, two Type TU55 DECtape Transports, and one PC01 High-Speed Paper Tape Reader and Punch.

Input/Output Specifications for the PDP-8 Program

The input to this program is any physiological signal that is preceded by 3 or 13 BCD digits represented by square waves. Data can be received on up to eight channels simultaneously. The data are digitized to a precision of 10 bits while being sampled at the rate of 500 times/sec. The output from this program is the "long block" transferred to the other computer via the high-speed interface. The other computer inputs the "long block" data, writes it on magnetic tape, and then signals the PDP-8 that it is ready for another block. This long block has two words for channel identification, 13 words for the recognized BCD digits, and 2,032 data points from the signal. These data points are used in the analysis of the signal by the other computer.

Timing and Storage Considerations

An input block size of 128 words was chosen; eight blocks this size (allowing one for each channel) can be transferred from the core to the drum faster than the next eight blocks can be brought in from the A/D converter (it takes the same amount of time to transfer one or eight blocks into core, because of the sampling rate). Two blocks,

128 words each, are reserved in core for each channel and are utilized in a double-buffered manner so that data is being written out of one block onto the drum while at the same time data is being read into the other block from the A/D converter. The block size was chosen from the timing table given in Table 10-1.

Associated with these blocks is a "Block Ready Table," which keeps a record of all blocks to be written on the drum; because of the alternating use of the blocks as described above, only one block for each channel should appear in this table at any one point in time. The entries recorded in the table are in the form xx0c, where xx00 is the starting address in core of the block to be written, and c is the associated channel and is used indirectly to determine the starting location on the drum where the block is to be written. Once the block is written, the entry in the table is replaced by zeros. Two pointers are associated with this table to ensure that the first block read into core is the first block written out on the drum. Pointer 2 gives the location in the table where the next block ready is to be recorded; pointer 3 gives the location in the table which contains the location in core of the next block to be written on the drum. The pointers travel through the table, with pointer 3 following pointer 2, until the end of the table is reached; then the pointers circle back to the beginning of the table again (see Figure 10-5).

For each channel, sixteen 128-word blocks are accumulated on the drum in consecutive locations to make up a "lead" to be read back into core, and then transferred to the other computer. As in the preceding discussion, there is an associated table, called a "Lead Ready Table," used to keep a record of all "leads" ready to be transferred. This table has 64 entries (the number of leads that can be stored on the drum—a maximum of eight leads per channel). The entries in this table are of the form yyf0, where yy00 is the starting location on the drum of the lead, and f is the field (f is set-to 4 for field 0 to avoid having an entry of 0000 when the lead starts at location 0000, field 0). When a lead has been read into PDP-8 core from the drum and transferred to the other computer, that entry in the table is set to 0000. As with the other table, two pointers are also associated with this table. Pointer 0 points to the location in the table where the next lead to be completed is to be recorded; pointer 1 points to the location in the table which contains the location on the drum of the next lead to be read and transferred. This arrangement ensures that the leads are transferred in the same order they are completed (see Figure 10-5).

Program Initialization

The program begins by clearing all peripheral equipment flags. Next, the usual initialization, such as clearing tables, setting counters and initial exits, etc., is done. Then, the drum flag is set by writing a sector on the drum. The clock is set to interrupt 500 times/sec. The program halts and, when everything is ready, the CONT (continue) switch is pressed. The clock is started, the interrupt turned on, and the program is sent to the write routine, where the first interrupt will occur (see Figure 10-6).

Program Interrupt Service Routines

The interrupt is off during interrupt servicing.

Answer Interrupt Routine. The clock interrupt flag is cleared, the contents of the accumulator are saved, and the return from interrupt is set up (see Figure 10-7).

A/D Service Routines (one for each channel). A value from the signal is digitized (0 through 10 bits) and stored, right adjusted. When all channels have been serviced, the program goes on to test the eight words just read in (see Figure 10-7).

Test Routines or Path Selector (one for each channel).

EXIT1 – "No data" path. Originally set to come to this exit. As soon as a nonzero value is received on this channel, EXIT2 is set (see Figure 10-8).

EXIT2 – "Code recognition" path. While set to this path, the data values are examined point by point, and the BCD digits represented by the square waves are recognized. After the required number of digits have been found and stored in the first block for a "lead," EXIT3 is set (see Figure 10-8).

EXIT3 – "Data store" path. The first block of data has two words for channel identification; the next 13 words are for the BCD digits just recognized; the remaining 113 words are filled with data points from the signal. When these 16 blocks (a "lead") have been input, EXIT2 is set to wait for the next BCD code. As each block is filled, it is recorded in the "Block Ready Table," pointer 2 is incremented, and the functions of Block 1 and Block 2 are switched (see Figure 10-8).

Exit from Interrupt Service. When each of the eight words is processed, the interrupt is turned on, and the program returns to where it was previously interrupted (see Figure 10-8).

Program I/O Routines

Drum Write. By using pointer 3, the program checks the "Block Ready Table" to see if there are any blocks to write. If not, the program goes to the Drum Read routine. If there are, the block is written from the starting core location xx00 given in the table to a location on the drum found indirectly by using the channel number c, which is also found in the table entry. After writing, the entry in the table is set to 0000 and pointer 3 is incremented. The drum location for this channel is appropriately advanced. When 16 blocks have been written, a "lead" has been stored on the drum. It is recorded in the "Lead Ready Table" in the location indicated by pointer 0; then pointer 0 is incremented. The program then goes back to the beginning of this routine to see if there are any more blocks to write. See Figure 10-9.

Drum Read. By using pointer 1, the program checks the "Lead Ready Table" to see if there are any leads to read and transfer. If not, the program goes to the Drum Write routine. If a "lead" is ready, and the last transfer is completed, the "lead" is read in from location and field yyf0 of the drum into the space reserved for a "lead" in core. Then, this entry in the "Lead Ready Table" is set to 0000, and pointer 1 is incremented. See Figure 10-9.

Transfer to Other Computer. The program sets up and initiates the transfer of the "lead" from the PDP-8 core to the core of the other computer, then goes to the Drum Write routine. See Figure 10-9.

Table 10-1. Example Timing Table

Block Size (# words)	Drum Transfer Time Core Drum (max. access time = 17.3 ms)		Data Input Time A/D Core
	1 Channel	8 Channels	1-8 Channels
16	17.3 + 0.25 = 17.55 ms	140.4 ms	32 ms
64	17.3 + 1.00 = 18.3 ms	146.4 ms	128 ms
128[1]	17.3 + 2.00 = 19.3 ms	154.4 ms	256 ms
256	17.3 + 4.00 = 21.3 ms	170.4 ms	512 ms
512	17.3 + 8.00 = 25.3 ms	202.4 ms	1024 ms
1024	17.3 + 16.00 = 33.3 ms	266.4 ms	2048 ms

[1]The input size used in this application, because 8 blocks in the least number of blocks that can be written on the drum in less time (154.4 ms) than it takes to read them into core (256 ms).

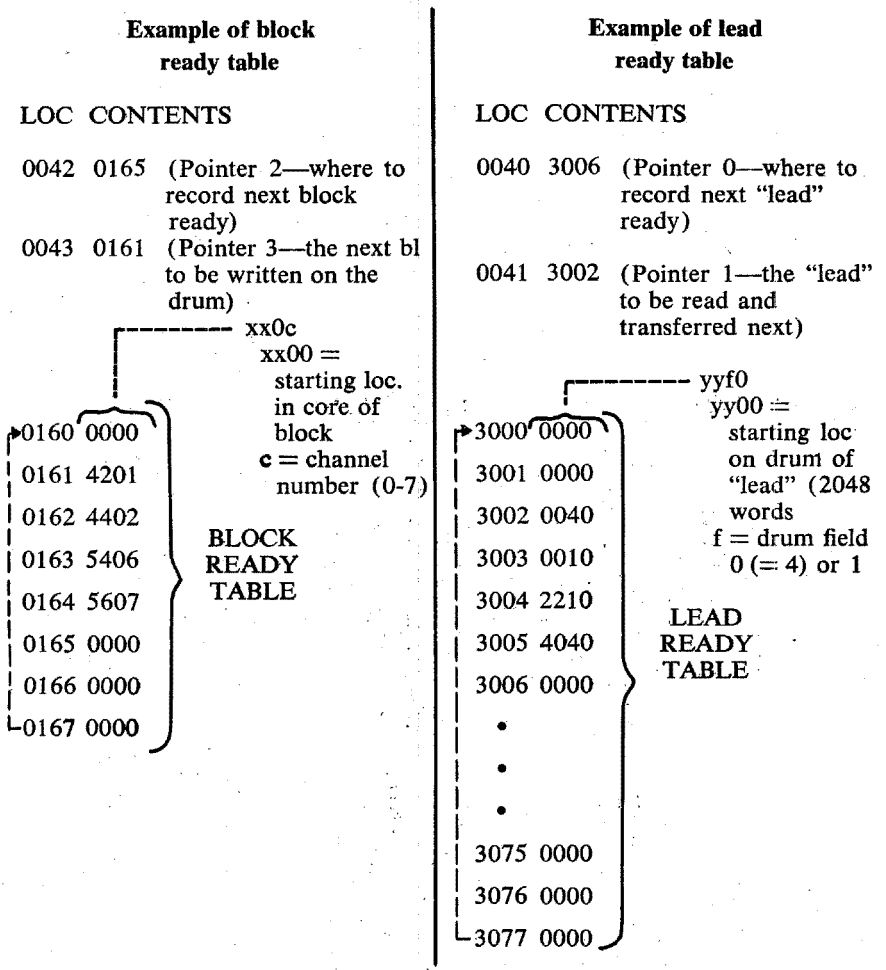


Figure 10-5. Examples of "Block Ready" and "Lead Ready" Tables

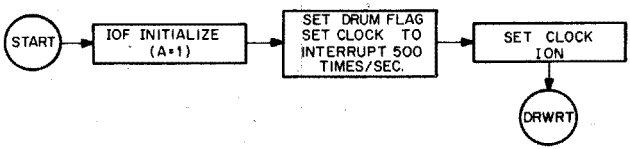


Figure 10-6. Example of Initialization

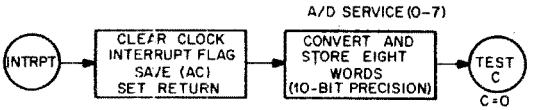


Figure 10-7. Example of Interrupt Answering and A/D Service Routines

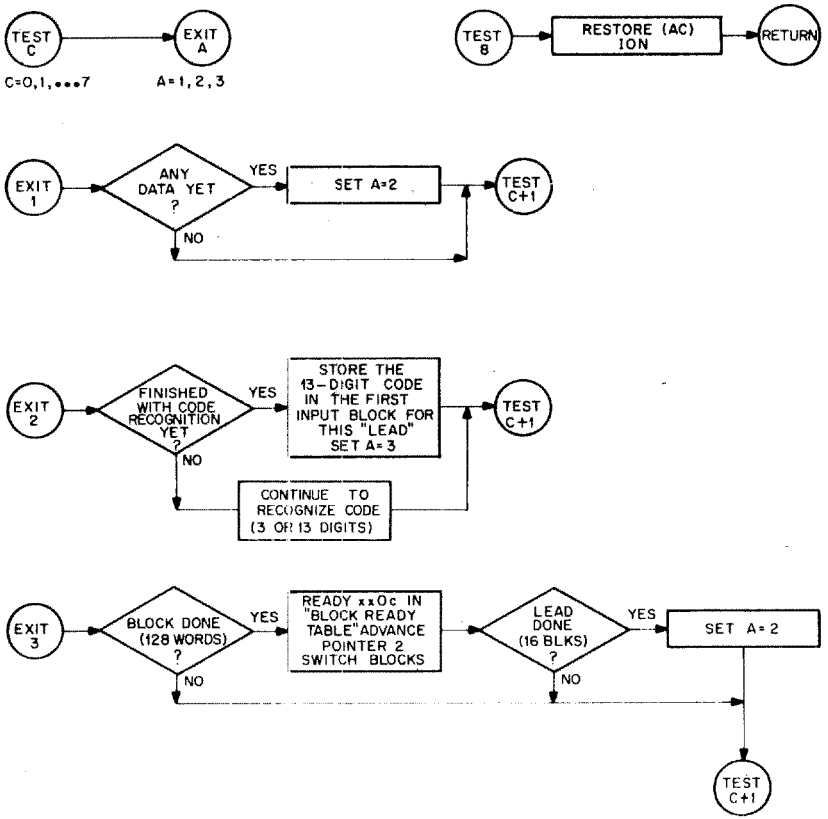


Figure 10-8. Interrupt Service

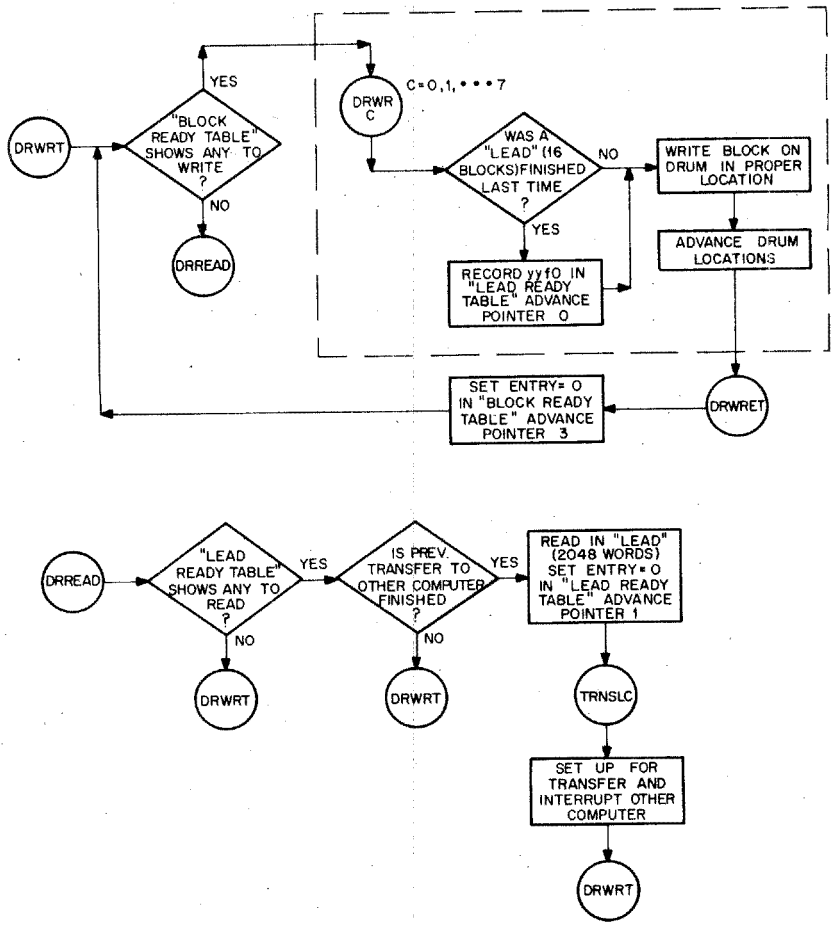


Figure 10-9. I/O Routines (Interrupt is on)

Chapter II

Digital Equipment Computer Users Society

OBJECTIVES

Digital Equipment Computer Users Society (DECUS) was established to advance the effective use of Digital Equipment Corporation's computers and peripheral equipment. It is a voluntary, non-profit users group supported by DEC, 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 publishes a newsletter, DECUSCOPE, every two months, sponsors technical symposia twice a year (Spring and Fall), maintains a program library, and publishes proceedings of its symposia.

DECUS PROGRAM LIBRARY

The DECUS Program Library is one of the major functions of the users group. It is maintained and operated separately from the DEC library and contains programs contributed by users. Programs are available for the PDP-8/I,-8/L,-8/S, and -8. (Programs are also available for the PDP-1, PDP-4/7/9, PDP-6/10, PDP-5, LINC, and LINC-8.) The library contains many types of programs, such as executive routines, editors, debuggers, special functions, games, maintenance, and various other classes of programs.

Programs for the PDP-8 family and the PDP-5 are listed later in this chapter together with abstracts of a few programs that are frequently requested by users.

Another feature of the program library is "Programs Available from Authors." This was initiated in order to allow users access to programs which are not fully documented or fully debugged but which are working to a certain extent. These programs are announced periodically in DECUSCOPE with information as to where they may be obtained. The Program Library Catalog also contains a section describing these programs.

Forms and information for submitting programs to the library may be obtained from the Executive Secretary. Any user may submit a program which he feels will be of use to others. Specifications for programs to be submitted are fairly simple. It is required, however, that documentation and operating instructions be clear. An object and symbolic tape of each program submitted is desirable.

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.

Programs submitted to the library are reviewed by the Programming Committee before being placed in the library. A review checklist is sent out with each program for evaluation by the user. Noteworthy comments and suggestions on DECUS programs are published in the newsletter. Certification of programs is under the jurisdiction of the Programming Committee.

The library presently contains 351 programs. In 1967, 8815 programs were issued to requestors, and 127 programs were submitted to the library.

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. The majority of articles are contributed by the users who are invited to submit ideas, programming notes, letters, and application notes for publication. DECUSCOPE is mailed every two months to members and others interested in DEC's computers and DECUS. Circulation reached 3,200 copies per issue in April, 1968.

Forms for submitting material to DECUS (newsletter or library) are available from the Executive Secretary.

ACTIVITIES

Two nation-wide symposia are held each year—one in the *SPRING* and the other in the *FALL*. Regional seminars and workshops are also held periodically. The proceedings and papers presented at the symposia and seminars are published shortly after each meeting and are sent automatically to meeting attendees and 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 Secretary. Specific details on this interchange program are available from the DECUS office.

DECUS encourages subgrouping of users with common interests. Special interest groups, such as the following, have been formed.

European Users have formed a group electing a committee to formally organize meetings annually in Europe. They have held three meetings to date with proceedings being published for each meeting.

The *Education Sub-Group,* organized in the early months of 1968, held their first technical session and sub-group workshop at the DECUS Spring 1968 Symposium in Philadelphia. Enthusiasm ran high, and chairwoman Mrs. Judith Edwards of Computer Instruction NETWORK is highly confident that they will be an extremely active and productive group.

Users in the *Biomedical* field have expressed the desire to hold meetings specific to their field. Sessions with papers presented in this area were held at two DECUS meetings, and a separate Biomedical Seminar was held in New York City in 1967. A formal sub-group with a chairman has not been formed at this time, but it is hoped that such a group will be organized in the near future.

Information on how to join or formally organize a sub-group may be obtained by contacting the DECUS Executive Secretary.

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.

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 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. The delegate receives DECUS literature automatically and library programs upon request.

A three-ring binder containing information and forms pertinent to the users group is sent to each delegate upon acceptance into the Society. The binder provides a convenient means for maintaining and updating DECUS literature. It contains such material as bylaws, newsletters, library catalog, forms for submitting and requesting material, indexes of newsletters and proceedings, etc.

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. They receive on an automatic basis only the newsletter and Program Library Catalog. They can, however, receive other DECUS material on a request basis.

Written application indicating desire to join must be submitted to the Executive Secretary 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.

MEMBERSHIP — APRIL 1968

Installation Delegates — 1136 Individual Members — 1238

EXECUTIVE BOARD, POLICIES AND ADMINISTRATION

The Society's policies are formulated by an Executive Board elected by vote of Installation Member delegates.

The board consists of a President, Executive Secretary, Recording Secretary, and Standing Committee Chairmen. In addition, a non-voting representative of Digital Equipment Corporation is a member of the board. The DECUS president for the preceding year is also included as a non-voting member of the Executive Board.

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 Secretary.

DECUS PROGRAM LIBRARY CATALOG

The DECUS Program Library Catalog contains lists and abstracts of all programs available from the DECUS Library. Programs for all DECs computers are included. The catalog is divided into three sections: category index, abstracts and numerical index, and "Programs Available from Authors." The category index for PDP-5/8 users programs and selected program abstracts are included below. Each member of DECUS receives a catalog automatically. Additional copies may be requested. The catalog is updated periodically and new additions to the library are published in DECUSCOPE.

PDP-5, PDP-8, -8/S, -8/I, -8/L Category Index

Category	DECUS No.	Title
Executive Routines, Assemblers and Compilers	5-13	PDP-5 Assembler (for use on IBM 7044/7094)
	5/8-18A,B,C,	Binary Tape Disassembly Programs
	5/8-20	Remote Operator FORTRAN System
	5/8-28a*	PAL III Modifications - Phoenix Assembler
	5/8-45*	PDP-5/8 Remote and Time-Shared System
	5/8-46a	Utility Programs for the PDP-5 and PDP-8
	8-59	PALDT—PAL Modified for DECtape (552 Control)
	5/8-64	DECtape Programming System
	8-67	PAL Modified for DECtape Input
	8/8S-77	PDP-8 Dual Process System
	8-82	Library System for 580 Magnetic Tape (Preliminary Version)
	8-84	One-Pass PAL III
	8-91	MICRO-8: An On-Line Assembler
	8-102	A LISP Interpreter for the PDP-8
	8-110	Directory Print (DIREC)
	8-115	Double Precision Integer Interpretive Package
	8-116	PDP-8/Automatic Tape Control (Type 57A) Library System
	8-122	SNAP (Simplified Numerical Analysis Program)
	8-123	UNIDEC Assembler
	8-124*	PDP-8 Assembler for IBM 360/67
	8-125	PDP-8 Relocatable Assembler for IBM 360/67
	6/8-12	PDP-8 Assembler for PDP-6

Category	DECUS No.	Title
Editors	5-24	Vector Input/Edit
	8-52	Tiny Tape Editor
	8-66	Editor Modified for DECtape
	8-97	GOOF
	8-101	Symbolic Editor With View
Debuggers	5/8-1.1	BPAK—A Binary Input/Output Package
	5-2.1	OPAK—An On-Line Debugging Program
	5-11	PDP-5 Debug System
	8-19a	DDT-UP—Octal-Symbolic Debugging Program
	5/8-33	Tape to Memory Comparator
	5-36	Octal Memory Dump Revised
	5-41	Breakpoint
	5/8-55	PALEX—An On-Line Debugging Program for the PDP-5/8
	5-63	SBUG - 4
	8-56	Fixed Point Trace No. 1
	8-57	Fixed Point Trace No. 2
	8-78	Diagnose: A Versatile Trace Routine for the PDP-8 Computer with EAE
	8/8S-83A&B	Octal Debugging Package (With and Without Floating Point)
	8-89	XOD—Extended Octal Debugging Program
	8-95	TRACE for EAE
	8-105	D-BUG
	8-111	DISKLOOK
Punch and Loaders	5-3	A Binary Relocatable Loader with Transfer Vector Options for the PDP-5
	5-12	Pack-Punch Processor and Reader for the PDP-5
	8-26A	Compressed Binary Loader (CBL)
	8-26B	CBC (BIN to CBL) and CONV (CBL to BIN)
	8-26C	XCBL—Extended Memory CBL Loader
	8-26D	XCBL Punch Program
	5/8-27&27a	Bootstrap Loader and Absolute Memory Clear
	8-47	ALBIN—A PDP-8 Loader for Relocatable Binary Programs
	5/8-48	Modified Binary Loader MKIV
	8-106	Readable Punch
	8-120	Disk/DECtape FAILSAFE
Duplicators and Verifiers	5-16	Tape Duplicator for the PDP-5
	5-22	DECtape Duplicate
	5/8-33	Tape to Memory Comparator
	5/8-53	COPCAT (DECtape Copy 552)
	8-113	Conversion of Friden (EIA) to ASCII
Arithmetic Routines Elementary Functions, Numerical Input/Output	5-4	Octal Typeout of Memory Area with Format Option
	5-6	BCD to Binary Conversion of 3-Digit Numbers
	5/8-7	Decimal to Binary Conversion by Radix Deflation on PDP-8
	5-8	PDP-5 Floating Point Routines
	5/8-21	Triple Precision Arithmetic Package
	5/8-29	BCD to Binary Conversion Subroutine (73.6 usec)
	5/8-35	Binary Coded Decimal to Binary Conversion Subroutine and Binary to Binary Coded Decimal Subroutine (Double Precision)
	5/8-38	FTYPE—Fractional Signed Decimal Type-In
	5/8-39	DSDPRINT, DDTYPE—Double-Precision Signed Decimal Input-Output Package
	5-42	Alphanumeric Input
	5/8-43	Unsigned Octal-Decimal Fraction Conversion
	8-44	Modification to the Fixed Point Output in the PDP-8 Floating Point Package
	8-60	Square Root Function by Subtraction Reduction
	8-61	Improvement to Digital 8-9-F Square Root
	5/8-65	LESQ29 and LESQ11
	8-72*	Matrix Inversion—Real Numbers
	8-73	Matrix Inversion—Complex Numbers
	8-74	Solution of System of Linear Equations: AX = B, By Matrix Inversion and Vector Multiplication
	8-75	Matrix Multiplication—Including Conforming Rectangular Matrices

PDP-5, PDP-8, -8/S, -8/I, -8/L Category Index (cont.)

Category	DECUS No.	Title
	8-80	Determination of Real Eigenvalues of a Real Matrix
	8-93	CHEW—Convert Any BCD to Binary-Double Precision
	8-96	J Bessel Function (FORTRAN)
	8-100	Double Precision Binary Coded Decimal Arithmetic Package
	8-103 A*	Four Word Floating Point Function Package
	8-103 B	Four Word Floating Point Rudimentary Calculator
	8-103 C	Four Word Floating Point Output Controller with Rounding
	8-103 D	Additional Instructions for Use with Four Word Floating Point Package
	8-114	Rounded Decimal Output Modification for PDP-8 FORTRAN
	8-115*	Double Precision Integer Interpretive Package
	8-118	General Linear Regression
Special Functions	8/8S-76	PDP NAVIG 2/2
Displays	5/8-23A	PDP-5/8 Oscilloscope Symbol Generator (4 × 6 Matrix)
	5/8-23B	PDP-5/8 Oscilloscope Symbol Generator (5 × 7 Matrix)
	8-99A	Kaleidoscope
	8-99B	Kaleidoscope - 338
	8-107	CHESSBOARD for the PDP-8/338
	8-108	Increment Mode Compiler—INCMOD (338)
	8-109	SEETXT Subroutine (338)
Text Manipulation	8-121	DECtape Handler (552 DECtape)
Probability and Statistics	5/8-9*	Analysis of Variance PDP-5/8
	5-25	A Pseudo Random Number Generator
Scientific and Engineering Applications	8-49	Relativistic Dynamics
	8-65	A Programmed Associative Multichannel Analyser
	5/8-69	LESQ29 and LESQ11
	5/8-90	Histogram on Teletype
	8-92	Analysis of Pulse-Height Analyser Test Data With a Small Computer
	8-117	A PDP-8 Interface for a Charged-Particle Nuclear Physics Experiment
	8-118	General Linear Regression
Hardware Control	5/8-17	Drum Transfer Routine for Use on the PDP-5/8
	5-30	GENPLOT - General Plotting Subroutines
	5-31	FORPLOT - FORTRAN Plotting Program for PDP-5
	5-37	Transfer II
	5-40	ICS DECtape Routines (One-Page)
	8-58	One-Page DECtape Routines (552 Control)
	8-70	EAE Routines for FORTRAN Operating System
	8-82	Library System for 580 Magnetic Tape (Preliminary Version)
	8-104	Card Reader Subroutine for the PDP-8 FORTRAN Compiler
	8-120	DISK/DECtape FAILSAFE
	8-121	DECtape Handler (552 DECtape)
Games and Demonstrators	5/8-14	Dice Game for the PDP-5 or PDP-8
	5/8-15	ATEPO (Auto Test in Elementary Programming and Operation of a PDP-5/8 Computer)
	5/8-54	Tic-Tac-Toe Learning Program
	8-71	Perpetual Calendar
	8-79	TIC-TAC-TOE (Trinity College Version)
	8-94A & B	BLACKJACK
	8-98	3D DRAW for 338
	8-99A	Kaleidoscope
	8-99B	Kaleidoscope - 338
	8-107	CHESSBOARD for the PDP-8/338
	8-108	Increment Mode Compiler, INCMOD (338)
	8-112	Sentence Generator
	8-119	Off-Line TIC-TAC-TOE (PAL)
Desk Calculators	5-5	Expanded Adding Machine
Maintenance	5-10	Paper Tape Reader Test
Miscellaneous	5/8-18A,B,C	Binary Tape Disassembly Program
	5/8-32a	Program to Relocate and Pack Programs in Binary Format
	5-34	Memory Halt—A PDP-5 Program to Store Halt in Most Memory

Category	DECUS No.	Title
	5/8-50	Additions to Symbolic Tape Format Generator
	5/8-51	Character Packing and Unpacking Routines
	8-68a	LABEL for PDP-8
	8-81	A BIN or RIM Format Data or Program Tape Generator
	5/8-85	Set Memory Equal to Anything
	8-87	XMAP
	8-88	DECtape Symbolic Format Generator
	8-112	Sentence Generator

NOTE: An asterisk beside the DECUS No. indicates that the program abstract is included on the following pages.

Abstracts of Frequently Requested Programs

The following program abstracts are representative examples taken from the DECUS Program Library Catalog. They have been selected from among the programs most frequently requested by users without regard for their relative merits.

DECUS No. 5/8-9

Analysis of Variance PDP-5/8

Henry Burkhardt, Digital Equipment Corporation, Maynard, Massachusetts

An analysis of variance program for the standard PDP-5/8 configuration. The output consists of:

A. For each sample:
   1) sample number
   2) sample size
   3) sample mean
   4) sample variance
   5) sample standard deviation

B. The grand mean

C. Analysis of Variance Table:
   1) the grand mean
   2) the weighted sum of squares of class means about the grand mean
   3) the degrees of freedom between samples
   4) the variance between samples
   5) the pooled sum of squares of individual values about the means of their respective classes
   6) the degrees of freedom within samples
   7) the variance within samples
   8) the total sum of squares of deviations from the grand mean

9) the degrees of freedom
10) the total variance
11) the ratio of the variance between samples to the variance with samples.

This is the standard analysis of variance table that can be used with the F test to determine the significance, if any, of the differences between sample means. The output is also useful as a first description of the data.

All arithmetic calculations are carried out by the Floating Point Interpretive Package (Digital-8-5-S).

DECUS No. 5/8-21

Triple Precision Arithmetic Package for the PDP-5 and the PDP-8

Joseph A. Rodnite, Information Control Systems, Ann Arbor, Michgan

An arithmetic package to operate on 36-bit signed integers. The operations are add, subtract, multiply, divide, input conversion, and output conversion. The largest integer which may be represented is 2^{35}-1 or 10 decimal digits. The routines simulate a 36-bit (3 word) accumulator in core locations 40, 41, and 42 and a 36-bit multiplier quotient register in core locations 43, 44, and 45. Aside from the few locations in page 0, the routines use less core storage space than the equivalent double-precision routines.

DECUS No. 5/8-28a

PAL III Modifications—Phoenix Assembler

Terrel L. Miedaner, Space Astronomy Laboratory, Madison, Wisconsin

This modification of the PAL IIII Assembler speeds up assembly on the ASR-33/35 and operates only with this I/O device. Operation is essentially the same as PAL III, except that an additional pass has been added, Pass 0. This pass, started in the usual manner, but with the switches set to zero, reads the symbolic tape into a core buffer area. Subsequent passes then read the tape image from storage instead of from the Teletype.

DECUS No. 5/8-45

PDP-5/8 Remote & Time-Shared System

James Miller, Dow Badische Chemical Company, Freeport, Texas

A time-shared programming system which allows remote stations immediate access to the computer and a wide selection of programs.

DECUS No. 8-72

Matrix Inversion—Real Numbers

A. E. Sapega, Trinity College, Hartford, Connecticut

The program inverts a matrix, up to size 12 × 12, of real numbers. The algorithm used is the Gauss-Jordan method. A unit vector of appropriate size is generated internally at each stage. Following the Gauss sweep-out, the matrix is shifted in storage, another unit vector is generated, and the calculation proceeds.

Other Programs Needed: FORTRAN Compiler and FORTRAN Operating System

Storage: This program uses essentially all core not used by the FORTRAN Operating System

Execution Time: Actual computation takes less than 10 seconds. Data read-in and read-out may take up to five minutes.

DECUS No. 8-103 A

Four Word Floating Point Function Package

D.A. Dalby, Bedford Institute of Oceanography, Dartmouth, Nova Scotia, Canada

This program package, written for use with Digital's Four Word Floating Point Package (DEC-08-FMHA-PB), includes subroutines to evaluate square, square root, sine, cosine, arctangent, natural logarithm, and exponential functions.

DECUS No. 8-115

Double Precision Integer Interpretive Package

Roger E. Anderson, Lawrence Radiation Laboratory, Livermore, California

This program is a Double Precision Integer Interpretive Package similar in operation to the Floating Point Package (Digital 8-5-S). It consists of addition, subtraction, multiplication, division, load, store, jump and branch subroutines coupled to an interpreter. It allows direct and indirect addressing in the normal assembly language manner. The operation is faster and more compact than the collected individual double precision subroutines. The program requires fourteen words on page zero and an additional two pages of memory.

Minimum Hardware: Basic PDP-5, -8, -8/S or -8/I

DECUS No. 8-124

PDP-8 Assembler for IBM 360/50 and up

V. Michael Powers, University of Michigan, Ann Arbor, Michigan

The 360/PDP-8 Assembler is a collection of programs written mostly in FORTRAN IV (G) which operate on the IBM 360/67. It assembles programs for PDP-5 and PDP-8 computers. Once a program has been assembled, it may be punched on cards, saved in a file, or transmitted through the Data Concentrator over data lines. It is also possible to obtain binary paper tapes by use of the Data Concentrator.

The Assembler follows the PAL III operation code and addressing conventions. The input format and program listing conventions are slightly different from those of PAL III, because it is organized around a line format, while PAL III is organized around a paper tape format.

NOTE: Source deck and documentation only available.

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
12. 10 111 011 110
14. 111 110 101
16. 1 110 001 011
18. 110 110 000 000
20. 11 110 101 111

b. 2. 5
4. 94
6. 31
8. 55
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
10. 7777
12. 7664
14. 255
16. 2372

b. 2. 111 011 110
4. 1 000
6. 101 100 010 100
8. 1 001 000 001
10. 110 100
12. 111 111 110 101
14. 100 101 011 010
16. 1 010 100 011

c. 2. 40
4. 1104
6. 3
8. 2500
10. 6005
12. 7777

d. 2. 31
4. 512
6. 482
8. 4095
10. 2431
12. 174

Answers to selected exercises on page 1-26

a. 2. 110
4. 10 111 000
6. 1 100
8. 11 100
10. 10 001 101
12. 1 010 010 101

b.	*One's Complement*	*Two's Complement*
2.	101 000 100 000	101 000 100 001
4.	111 111 111 111	000 000 000 000
6.	111 011 011 011	111 011 011 100
8.	011 111 111 111	100 000 000 000
10.	011 110 011 001	011 110 011 010
12.	000 000 000 000	000 000 000 001

c. 2. 101 000 101
4. 11 001 110 101

d. 2. 110 110 010
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 on page 2-28

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 on page 3-34

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,          5703        /-1084 DECIMAL
COUNT,          0
CONST1,         44          /36 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 on page 4-23

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
                    /IN AC
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 5

Answers to selected exercises on page 5-43

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* 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	↑	336	36
0	260	60	←	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	

*An abbreviation for USA Standard Code for Information Interchange.

Card Reader Code

The following table gives the octal representation of the internal (binary) codes for the listed punch combinations. These internal codes are generated by the card reader and are transmitted to the PDP-8 upon execution of the appropriate IOT instruction. Any combination of punches which is not shown in the table is invalid, and the card reader will not detect invalid combinations.

Internal Code	Card Code Zone	Num.	IBM 29 Keyboard Character	IBM 26 Keyboard Character
00	NONE		SPACE	SPACE
01	—	1	1	1
02	—	2	2	2
03	—	3	3	3
04	—	4	4	4
05	—	5	5	5
06	—	6	6	6
07	—	7	7	7
10	—	8	8	8
11	—	9	9	9
12	—	8•2	: Colon*	None Assigned
13	—	8•3	# Number Sign	= Equal Sign
14	—	8•4	@ At Sign	' Apostrophe
15	—	8•5	' Apostrophe*	None Assigned
16	—	8•6	= Equal Sign*	None Assigned
17	—	8•7	" Quotation Mark*	None Assigned
20	0	—	0	0
21	0	1	/ Slash	/ Slash
22	0	2	S	S
23	0	3	T	T
24	0	4	U	U
25	0	5	V	V
26	0	6	W	W
27	0	7	X	X
30	0	8	Y	Y
31	0	9	Z	Z

*Not all available IBM 29 keyboard arrangements contain these graphic characters.

Internal Code	Card Code Zone	Num.	IBM 29 Keyboard Character	IBM 26 Keyboard Character
32	0	8•2	None Assigned	None Assigned
33	0	8•3	, Comma	, Comma
34	0	8•4	% Percent	(Parenthesis
35	0	8•5	_ Underscore*	None Assigned
36	0	8•6	> Greater Than*	None Assigned
37	0	8•7	? Question Mark*	None Assigned
40	11	—	— Minus or Hyphen	— Minus or Hyphen
41	11	1	J	J
42	11	2	K	K
43	11	3	L	L
44	11	4	M	M
45	11	5	N	N
46	11	6	O	O
47	11	7	P	P
50	11	8	Q	Q
51	11	9	R	R
52	11	8•2	! Exclamation*	None Assigned
53	11	8•3	$ Dollar Sign	$ Dollar Sign
54	11	8•4	* Asterisk	* Asterisk
55	11	8•5	) Parenthesis*	None Assigned
56	11	8•6	; Semicolon*	None Assigned
57	11	8•7	¬ Logical NOT*	None Assigned
60	12	—	& Ampersand	+ Plus
61	12	1	A	A
62	12	2	B	B
63	12	3	C	C
64	12	4	D	D
65	12	5	E	E
66	12	6	F	F
67	12	7	G	G
70	12	8	H	H
71	12	9	I	I
72	12	8•2	¢ Cent Sign*	None Assigned

*Not all available IBM 29 keyboard arrangements contain these graphic characters.

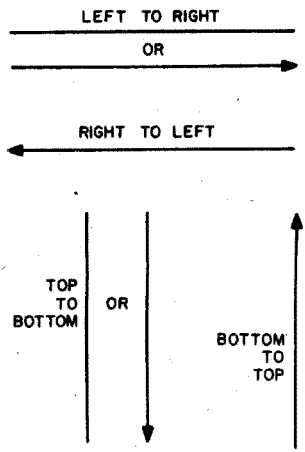
Internal Code	Card Code Zone	Num.	IBM 29 Keyboard Character	IBM 26 Keyboard Character
73	12	8•3	. Period	. Period
74	12	8•4	< Less than	) Parenthesis
75	12	8•5	(Parenthesis*	None Assigned
76	12	8•6	+ Plus Sign*	None Assigned
77	12	8•7	| Vertical Bar*	None Assigned

*Not all available IBM 29 keyboard arrangements contain these graphic characters.

Appendix C
Flowchart Guide

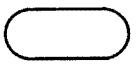
The following is a partial list of flowchart symbols which can be used to diagram the logical flow of a program. The symbols may be made sufficiently large to include the pertinent information.

REPRESENTATION OF FLOW



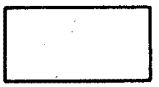
The direction of flow in a program is represented by lines drawn between symbols. These lines indicate the order in which the operations are to be performed. Normal direction of flow is from left to right and top to bottom. When the flow direction is not from left to right or top to bottom, arrowheads are placed on the reverse direction flowlines. Arrowheads may also be used on normal flow lines for increased clarity.

TERMINAL



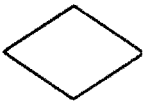
The oval symbol represents a terminal point in a program. It can be used to indicate a start, stop, or interrupt of program flow. The appropriate word is included within the symbol.

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.

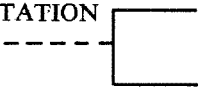
PREDEFINED PROCESS

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 shown below 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

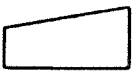


An addition of descriptive comments or explanatory notes for clarification is included within this symbol.

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

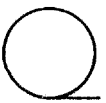


This symbol may be used to represent the manual input of information by means of on-line keyboards, switch settings, etc.

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
Tables Of Instructions

The instruction tables of this appendix apply in general to all PDP-8 family computers, except where differences are noted for specific machines.

PDP-8/I Memory Reference Instructions[1]

Mnemonic Symbol	Operation Code	Direct Addr.		Indirect Addr.		Operation
		States Entered	Execution Time (μsec)	States Entered	Execution Time (μsec)	
AND Y	0	F, E	3.0	F, D, E	4.5	Logical AND. The AND operation is performed between the content of memory location Y and the content of the AC. The result is left in the AC, the original content of the AC is lost, and the content of Y is restored. Corresponding bits of the AC and Y are operated upon independently.
TAD Y	1	F, E	3.0	F, D, E	4.5	Two's complement add. The content of memory location Y is added to the content of the AC in two's complement arithmetic. The result of this addition is held in the AC, the original content of the AC is lost, and the content of Y is restored. If there is a carry from AC0, the link is complemented.

[1]All MRI's operate on all PDP-8 family computers; execution times, however, apply to PDP-8/I.
[2]Y is the address of a memory location.

PDP-8/I Memory Reference Instructions (continued)

Mnemonic Symbol	Operation Code	Direct Addr.		Indirect Addr.		Operation
		States Entered	Execution Time (μsec)	States Entered	Execution Time (μsec)	
ISZ Y	2	F, E	3.0	F, D, E	4.5	Increment and skip if zero. The content of memory location Y is incremented by one. If the resultant content of Y equals zero, the content of the PC is incremented and the next instruction is skipped. If the resultant content of Y does not equal zero, the program proceeds to the next instruction. The incremented content of Y is restored to memory. If
DCA Y	3	F, E	3.0	F, D, E	4.5	Deposit and clear AC. The content of the AC is deposited in core memory at address Y and the AC is cleared. The previous content of memory location Y is lost.
JMS Y	4	F, E	3.0	F, D, E	4.5	Jump to subroutine. The incremented content of the PC is deposited in core memory location Y, and the next instruction is taken from core memory location Y + 1.
JMP Y	5	F	1.5	F, D	3.0	Jump to Y. Address Y is set into the PC so that the next instruction is taken from core memory address Y. The original content of the PC is lost.

PDP-8/I Group 1 Operate Microinstructions[1]

Mnemonic Symbol	Octal Code	Sequence	Operation
NOP	7000	—	No operation. Causes a 1.5 μsec 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.
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 AC 11.
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.

[1]Group 1 operate microinstructions operate on all PDP-8 family computers. (See Appendix E for the event times of each model.)

PDP-8/I Group 2 Operate Microinstructions[1]

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 = 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, 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 $>$ 0 and if L = 0.
SMA SZA	7540	1	Skip if AC $<$ 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 $>$ 0, then clear AC.

[1]Group 2 microinstructions operate on all PDP-8 family computers.

PDP-8/I Extended Arithmetic Element[1] Microinstructions

Mnemonic Symbol	Octal Code	Sequence	Operation
MUY	7405	3	Multiply. The number held in the MQ is multiplied by the number held in core memory location PC + 1 (or the next successive core memory location after the MUY Command). At the conclusion of this command the most significant 12 bits of the product are contained in the AC and the least significant 12 bits of the product are contained in the MQ.
DVI	7407	3	Divide. The 24-bit dividend held in the AC (most significant 12 bits) and the MQ (least significant 12 bits) is divided by the number held in core memory location PC + 1 (or the next successive core memory location following the DVI command). At the conclusion of this command the quotient is held in the MQ, the remainder is in the AC, and the L contains a 0. If the L contains a 1, divide overflow occurred so the operation was concluded after the first cycle of the division.
NMI	7411	3	Normalize. This instruction is used as part of the conversion of a binary number to a fraction and an exponent for use in floating-point arithmetic. The combined content of the AC and the MQ is shifted left by this one command until the content of AC0 is not equal to the content of AC1, to form the fraction. Zeros are shifted into vacated MQ11 positions for each shift. At the conclusion of this operation, the step counter contains a number equal to the number of shifts performed. The content of L is lost.
SHL	7413	3	Shift arithmetic left. This instruction shifts the combined content of the AC and MQ to the left one position more than the number of positions indicated by the content of core memory at address PC + 1 (or the next successive core memory location following the SHL command). During the shifting, zeros are shifted into vacated MQ11 positions.
ASR	7415	3	Arithmetic shift right. The combined content of the AC and the MQ is shifted right one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the ASR command). The sign bit, contained in AC0, enters vacated positions, the sign bit is preserved, information shifted out of MQ11 is lost, and the L is undisturbed during this operation.

[1]This option is not available with the PDP-8/L.

PDP-8/I Extended Arithmetic Element Microinstructions (continued)

Mnemonic Symbol	Octal Code	Sequence	Operation
LSR	7417	3	Logical shift right. The combined content of the AC and MQ is shifted left one position more than the number contained in memory location PC + 1 (or the next successive core memory location following the LSR command). This command is similar to the ASR command except that zeros enter vacated positions instead of the sign bit entering these locations. Information shifted out of MQ11 is lost and the L is undisturbed during this operation.
MQL	7421	2	Load multiplier quotient. This command clears the MQ, loads the content of the AC into the MQ, then clears the AC.
SCA	7441	2	Step counter load into accumulator. The content of the step counter is transferred into the AC. The AC should be cleared prior to issuing this command or the CLA command can be combined with the SCA to clear the AC, then effect the transfer.
SCL	7403	3	Step counter load from memory. Loads complement of bits 7 through 11 of the word in memory following the instruction into the step counter.
MQA	7501	2	Multiplier quotient load into accumulator. The content of the MQ is transferred into the AC. This command is given to load the 12 least significant bits of the product into the AC following a multiplication or to load the quotient into the AC following a division. The AC should be cleared prior to issuing this command or the CLA command can be combined with the MQA to clear the AC then effect the transfer.
CLA	7601	1	Clear accumulator. The AC is cleared during sequence 1, allowing this command to be combined with the other EAE commands that load the AC during sequence 2 (such as SCA and MQA).
CAM	7621	1, 2	Clear accumulator and multiplier quotient.

Basic IOT Microinstructions

Mnemonic	Octal	Operation
Program Interrupt		
ION	6001	Turn interrupt on and enable the computer to respond to an interrupt request. When this instruction is given, the computer executes the next instruction, then enables the interrupt. The additional instruction allows exit from the interrupt subroutine before allowing another interrupt to occur.
IOF	6002	Turn interrupt off i.e. disable the interrupt.
High Speed Perforated Tape Reader and Control		
RSF	6011	Skip if reader flag is a 1.
RRB	6012	Read the content of the reader buffer and clear the reader flag. (This instruction does not clear the AC.)
RFC	6014	Clear reader flag and reader buffer, fetch one character from tape and load it into the reader buffer, and set the reader flag when done.
High Speed Perforated Tape Punch and Control		
PSF	6021	Skip if punch flag is a 1.
PCF	6022	Clear punch flag and punch buffer.
PPC	6024	Load the punch buffer from bits 4 through 11 of the AC and punch the character. (This instruction does not clear the punch flag or punch buffer.)
PLS	6026	Clear the punch flag, clear the punch buffer, load the punch buffer from the content of bits 4 through 11 of the accumulator, punch the character, and set the punch flag to 1 when done.
Teletype Keyboard/Reader		
KSF	6031	Skip if keyboard flag is a 1.
KCC	6032	Clear AC and clear keyboard flag.
KRS	6034	Read keyboard buffer static. (This is a static command in that neither the AC nor the keyboard flag is cleared.)
KRB	6036	Clear AC, clear keyboard flag, and read the content of the keyboard buffer into the content of AC 4-11.
Teletype Teleprinter/Punch		
TSF	6041	Skip if teleprinter flag is a 1.
TCF	6042	Clear teleprinter flag.
TPC	6044	Load the TTO from the content of AC 4-11 and print and/or punch the character.
TLS	6046	Load the TTO from the content of AC 4-11, clear the teleprinter flag, and print and/or punch the character.
Oscilloscope Display Type VC8/I and Precision CRT Display Type 30N		
DCX	6051	Clear X coordinate buffer.
DXL	6053	Clear and load X coordinate buffer.

Basic IOT Microinstructions (continued)

Mnemonic	Octal	Operation
DIX	6054	Intensify the point defined by the content of the X and Y coordinate buffers.
DXS	6057	Executes the combined functions of DXL followed by DIX.
DCY	6061	Clear Y coordinate buffer.
DYL	6063	Clear and load Y coordinate buffer.
DIY	6064	Intensify the point defined by the content of the X and Y coordinate buffers.
DYS	6067	Executes the combined functions of DYL followed by DIY.
Oscilloscope Display Type VC8/I		
DSB	6075	Set minimum brightness.
DSB	6076	Set medium brightness.
DSB	6077	Set maximum brightness.
DSB	6074	Zero brightness.
Precision CRT Display Type 30N		
DLB	6074	Load brightness register (BR) from bits 9 through 11 of the AC.
Light Pen Type 370		
DSF	6071	Skip if display flag is a 1.
DCF	6072	Clear the display flag.
Memory Parity Type MP8/I		
SMP	6101	Skip if memory parity error flag = 0.
CMP	6104	Clear memory parity error flag.
Automatic Restart Type KP8/I		
SPL	6102	Skip if power is low.
Memory Extension Control Type MC8/I		
CDF	62N1	Change to data field N. The data field register is loaded with the selected field number (0 to 7). All subsequent memory requests for operands are automatically switched to that data field until the data field number is changed by a new CDF command.
CIF	62N2	Prepare to change to instruction field N. The instruction buffer register is loaded with the selected field number (0 to 7). The next JMP or JMS instruction causes the new field to be entered.
RDF	6214	Read data field into AC 6-8. Bits 0-5 and 9-11 of the AC are not affected.
RIF	6224	Same as RDF except reads the instruction field.
RIB	6234	Read interrupt buffer. The instruction field and data field stored during an interrupt are read into AC 6-8 and 9-11 respectively.
RMF	6244	Restore memory field. Used to exit from a program interrupt.

Mnemonic	Octal	Operation
Data Communications Systems Type 680		
TTINCR	6401	The content of the line select register is incremented by one.
TTI	6402	The line status word is read and sampled. If the line is active for the fourth time, the line bit is shifted into the character assembly word. If the line is active for a number of times less than four, the count is incremented. If the line is not active, the active/inactive status of the line is recorded.
TTO	6404	The character in the AC is shifted right one position, zeros are shifted into vacated positions, and the original content of AC11 is transferred out of the computer on the Teletype line.
TTCL	6411	The line select register is cleared.
TTSL	6412	The line select register is loaded by an OR transfer from the content of AC5-11, then the AC is cleared.
TTRL	6414	The content of the line select register is read into AC5-11 by an OR transfer.
TTSKP	6421	Skip if clock 1 flag is a 1.
TTXON	6424	Clock 1 is enabled to request a program interrupt and clock 1 flag is cleared.
TTXOF	6422	Clock 1 is disabled from causing a program interrupt and clock 1 flag is cleared.
Incremental Plotter and Control Type VP8/I		
PLSF	6501	Skip if plotter flag is a 1.
PLCF	6502	Clear plotter flag.
PLPU	6504	Plotter pen up. Raise pen off of paper.
PLPR	6511	Plotter pen right.
PLDU	6512	Plotter drum (paper) upward.
PLDD	6514	Plotter drum (paper) downward.
PLPL	6521	Plotter pen left.
PLUD	6522	Plotter drum (paper) upward. (Same as 6512.)
PLPD	6524	Plotter pen down. Lower pen on to paper.
Serial Magnetic Drum System Type 251		
DRCR	6603	Load the drum core location counter with the core memory location information in the accumulator. Prepare to read one sector of information from the drum into the specified core location. Then clear the AC.
DRCW	6605	Load the drum core location counter with the core memory location information in the accumulator. Prepare to write one sector of information into the drum from the specified core location. Then clear the AC.
DRCF	6611	Clear completion flag and error flag.

Mnemonic	Octal	Operation
DREF	6612	Clear the AC then load the condition of the parity error and data timing error flip-flops of the drum control into accumulator bits 0 and 1 respectively to allow programmed evaluation of an error flag.
DRTS	6615	Load the drum address register with the track and sector address held in the accumulator. Clear the completion and error flags, and begin a transfer (reading or writing). Then clear the AC.
DRSE	6621	Skip next instruction if the error flag is a 0 (no error).
DRSC	6622	Skip next instruction if the completion flag is a 1 (sector transfer is complete).
DRCN	6624	Clear error flag and completion flag, then initiate transfer of next sector.
Serial Magnetic Drum System Type RM08		
DRCR	6603	Load the drum core location counter with the core memory location information in the accumulator. Prepare to read one sector of information from the drum into the specified core location. Then clear the AC.
DRCW	6605	Load the drum core location counter with the core memory location information in the accumulator. Prepare to write one sector of information into the drum from the specified core location. Then clear the AC.
DRCF	6611	Clear completion flag and error flag.
DRES	6612	Clear the AC then load the condition of the parity error and data timing error flip-flops of the drum control into accumulator bits 0 and 1 respectively to allow programmed evaluation of an error flag. The contents of the drum sector counter are transferred into bits AC 6-11.
DRTS	6615	Load the drum address register with the track and sector address held in the accumulator. Clear the completion and error flags, and begin a transfer (reading or writing). Then clear the AC.
DRSE	6621	Skip next instruction if the error flag is a 0 (no error).
DRSC	6622	Skip next instruction if the completion flag is a 1 (sector transfer is complete).
DRFS	6624	Loads the drum field register with the contents of the accumulator bits 10 and 11. Loads the sector number register with the contents of the accumulator bits 0-5, to specify the number of sectors to be transferred. Loads the three most significant bits of the drum core location register (DCL_{0-2}) with the contents of the AC bits 6, 7, 8 to specify the core memory block to be used during the drum transfer.

Basic IOT Microinstructions (continued)

Mnemonic	Octal	Operation
Random Access Disc File (Type DF32)		
DCMA	6601	Clears memory address register, parity error and completion flags. This instruction clears the disc memory request flag and interrupt flags.
DMAR	6603	The contents of the AC are loaded into the disc memory address register and the AC is cleared. Begin to read information from the disc into the specified core location. Clears parity error and completion flags. Clears interrupt flags.
DMAW	6605	The contents of the AC are loaded into the disc memory address register and the AC is cleared. Begin to write information into the disc from the specified core location. Clears parity error and completion flags.
DCEA	6611	Clears the disc extended address and memory address extension register.
DSAC	6612	Skips next instruction if address confirmed flag is a 1. (AC is cleared.)
DEAL	6615	The disc extended address extension registers are cleared and loaded with the track data held in the AC.
DEAC	6616	Clear the AC then loads the contents of the disc extended address register into the AC to allow program evaluation. Skip next instruction if address confirmed flag is a 1.
DFSE	6621	Skips next instruction if parity error, data request late, or write lock switch flag is a zero. Indicates no errors.
DFSC	6622	Skip next instruction if the completion flag is a 1. Indates data transfer is complete.
DMAC	6626	Clear the AC then loads contents of disc memory address register into the AC to allow program evaluation.
Automatic Line Printer and Control Type 645		
LSE	6651	Skip if line printer error flag is a 1.
LCB	6652	Clear both sections of the printing buffer.
LLB	6654	Load printing buffer from the content of AC 6-11 and clear the AC.
LSD	6661	Skip if the printer done flag is a 1.
LCF	6662	Clear line printer done and error flags.
LPR	6664	Clear the format register, load the format register from the content of AC 9-11, print the line contained in the section of the printer buffer loaded last, clear the AC, and advance the paper in accordance with the selected channel of the format tape if the content of AC 8 = 1. If the content of AC 8 = 0, the line is printed and paper advance is inhibited.

Mnemonic	Octal	Operation
DECtape Transport Type TU55 and DECtape Control Type TC01		
DTRA	6761	The content of status register A is read into AC0-9 by an OR transfer. The bit assignments are: AC0-2 = Transport unit select number AC3-4 = Motion AC5 = Mode AC6-8 = Function AC9 = Enable/disable DECtape control flag
DTCA	6762	Clear status register A. All flags undisturbed.
DTXA	6764	Status register A is loaded by an exclusive OR transfer from the content of the AC, and AC10 and AC11 are sampled. If AC10 = 0, the error flags are cleared. If AC11 = 0, the DECtape control flag is cleared.
DTSF	6771	Skip if error flag is a 1 or if DECtape control flag is a 1.
DTRB	6772	The content of status register B is read into the AC by an OR transfer. The bit assignments are: AC0 = Error flag AC1 = Mark track error AC2 = End of tape AC3 = Select error AC4 = Parity error AC5 = Timing error AC6-8 = Memory field AC9-10 = Unused AC11 = DECtape flag
DTLB	6774	The memory field portion of status register B is loaded from the content of AC6-8.
Card Reader and Control Type CR8/I		
RCSF	6631	Skip if card reader data ready flag is a 1.
RCRA	6632	The alphanumeric code for the column is read into AC6-11, and the data ready flag is cleared.
RCRB	6634	The binary data in a card column is transferred into AC0-11, and the data ready flag is cleared.
RCSP	6671	Skip if card reader card done flag is a 1.
RCSE	6672	Clear the card done flag, select the card reader and start card motion towards the read station, and skip if the reader-not-ready flag is a 1.
RCRD	6674	Clear card done flag.
Automatic Magnetic Tape Control Type TC58		
MTSF	6701	Skip on error flag or magnetic tape flag. The status of the error flag (EF) and the magnetic tape flag (MTF) are sampled. If either or both are set to 1, the content of the PC is incremented by one to skip the next sequential instruction.
MTCR	6711	Skip on tape control ready (TCR). If the tape control is ready to receive a command, the PC is incremented by one to skip the next sequential instruction.

Basic IOT Microinstructions (continued)

Mnemonic	Octal	Operation
MTTR	6721	Skip on tape transport ready (TTR). The next sequential instruction is skipped if the tape transport is ready.
MTAF	6712	Clear the status and command registers, and the EF and MTF if tape control ready. If tape control not ready, clears MTF and EF flags only.
— —	6724	Inclusively OR the contents of the command register into bits 0-11 of the AC.
MTCM	6714	Inclusively OR the contents of AC bits 0-5, 9-11 into the command register; JAM transfer bits 6, 7, 8 (command function).
MTLC	6716	Load the contents of AC bits 0-11 into the command register.
— —	6704	Inclusively OR the contents of the status register into bits 0-11 of the AC.
MTRS	6706	Read the contents of the status register into bits 0-11 of the AC.
MTGO	6722	Set "go" bit to execute command in the command register if command is legal.
— —	6702	Clear the accumulator.

General Purpose Converter and Multiplexer Control Type AF01A

ADSF	6531	Skip if A/D converter flag is a 1.
ADVC	6532	Clear A/D converter flag and convert input voltage to a digital number, flag will set to 1 at end of conversion. Number of bits in converted number determined by switch setting, 11 bits maximum.
ADRB	6534	Read A/D converter buffer into AC, left justified, and clear flag.
ADCC	6541	Clear multiplexer channel address register.
ADSC	6542	Set up multiplexer channel as per AC 6-11. Maximum of 64 single ended or 32 differential input channels.
ADIC	6544	Index multiplexer channel address (present address + 1). Upon reaching address limit, increment will cause channel 00 to be selected.

Guarded Scanning Digital Voltmeter Type AF04A

VSEL	6542	The contents of the accumulator are transferred to the AF04A control register.
VCNV	6541	The contents of the accumulator are transferred to the AF04A channel address register. Analog signal on selected channel is automatically digitized.
VINX	6544	The last channel address is incremented by one and the analog signal on the selected channel is automatically digitized.

Basic IOT Microinstructions (continued)

Mnemonic	Octal	Operation
VSDR	6531	Skip if data ready flag is a 1.
VRD	6532	Selected byte of voltmeter is transferred to the accumulator and the data ready flag is cleared.
VBA	6534	BYTE ADVANCE command requests next twelve bits, data ready flag is set.
VSCC	6571	SAMPLE CURRENT CHANNEL when required to digitize analog signal on current channel repeatedly.

Appendix E

Legal Microinstruction Combinations

The following tables identify the legal operate microinstruction combinations for the PDP-8 family computers. It should be noted that each possible pair of mnemonics within a microprogrammed instruction should be checked for legality.

GROUP 1 MICROINSTRUCTION COMBINATIONS

										Logical Event Times		
	CLA	CLL	CML	CMA	RAR	RAL	RTR	RTL	IAC	8/I, 8/L	8/S	8
CLA	—	All	All	All	All	All	All	All	All	1	1	1
CLL	All	—	All	All	All	All	All	All	All			
CML	All	All	—	All	All	All	All	All	All	2	2	2
CMA	All	All	All	—	8 8/I 8/L	8 8/I 8/L	8 8/I 8/L	8 8/I 8/L	All			
RAR	All	All	All	8 8/I 8/L	—	None	None	None	8/I 8/L	3	3	3
RAL	All	All	All	8 8/I 8/L	None	—	None	None	8/I 8/L			
RTR	All	All	All	8 8/I 8/L	None	None	—	None	8/I 8/L			
RTL	All	All	All	8 8/I 8/L	None	None	None	—	8/I 8/L			
IAC	All	All	All	All	8/I 8/L	8/I 8/L	8/I 8/L	8/I 8/L	—	4		

GROUP 2 MICROINSTRUCTION COMBINATIONS

		SMA	SZA	SNL	SPA	SNA	SZL	SKP	CLA	OSR	HLT	Logical Event Times (All PDP-8 computers)
OR Group (Skip if *any*)	SMA	—	All	None	None	None	None	None	All	All	All	
	SZA	All	—	All	None	None	None	None	All	All	All	
	SNI	All	All	—	None	None	None	None	All	All	All	
AND Group (Skip if *all*)	SPA	None	None	None	—	All	All	None	All	All	All	1
	SNA	None	None	None	All	—	All	None	All	All	All	
	SLZ	None	None	None	All	All	—	None	All	All	All	
	SKP	None	None	None	None	None	None	—	All	All	All	
	CLA	All	All	All	All	All	All	All	—	All	All	2
	OSR	All	All	All	All	All	All	All	All	—	All	3
	HLT	All	All	All	All	All	All	All	All	All	—	4

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
1 073 741 824	30	0.000 000 000 931 322 574 615 478 515 625
2 147 483 648	31	0.000 000 000 465 661 287 307 739 257 812 5
4 294 967 296	32	0.000 000 000 232 830 643 653 869 628 906 25
8 589 934 592	33	0.000 000 000 116 415 321 826 934 814 453 125
17 179 869 184	34	0.000 000 000 058 207 660 913 467 407 226 562 5
34 359 738 368	35	0.000 000 000 029 103 830 456 733 703 613 281 25
68 719 476 736	36	0.000 000 000 014 551 915 228 366 851 806 640 625

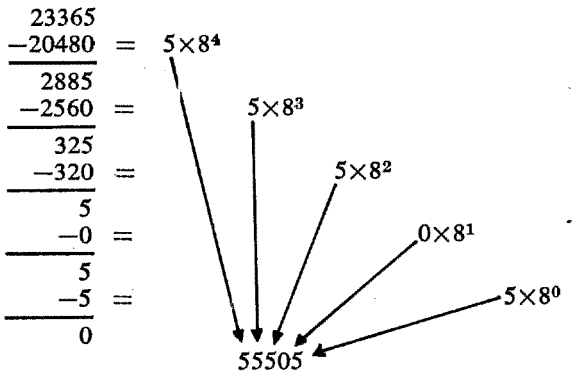
Octal-Decimal Conversion

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.

16384
0
128
56
7
16575

Thus, 40277_8 is converted to 16575_{10}.

To convert a number from decimal to binary, the highest number which appears in the table is subtracted from the number. The highest number which can be subtracted from the remainder is subtracted until the number is reduced to 0. The octal number is obtained by recording the multipliers of the numbers which were subtracted in the proper digit positions. To convert 23365_{10} to its octal equivalent, the following procedure is followed.



Thus, 23365_{10} is converted to 55505_8.

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	.230468	.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	1.000 000 000 000 000 000 000	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

$$0+1=\begin{matrix}0+0=0\\1+0=1\\1+1=10\end{matrix} \qquad 0\times 1=\begin{matrix}0\times 0=0\\1\times 0=0\\1\times 1=1\end{matrix}$$

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

π =	3.11037 552421$_8$	e =	2.55760 521305$_8$	γ =	0.44742 147707$_8$
π^{-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$_8$

Common Abbreviations

The following list of abbreviations includes many abbreviations used in conjunction with the PDP-8 family of computers.

Abbreviation	Meaning
AC	Accumulator
ADDR	Address
B. SP.	Back Space
BIN	Binary
CLC	Current Location Counter
CONT	Continue
CR	Carriage Return
CR/LF	Carriage Return-Line Feed
CTRL/L	Control/L (represents holding down the CTRL key while depressing the L key or the key following the slash)
DEC	Digital Equipment Corporation
DEP	Deposit
DF	Data Field
EAE	Extended Arithmetic Element
EXAM	Examine
IF	Instruction Field
INST	Instruction
L	Link
LF	Line Feed
LOAD ADD	Load Address
LOC	Location
LSP	Low-Speed Punch
LSR	Low-Speed Reader
HSP	High-Speed Punch
HSR	High-Speed Reader
KBRD	Keyboard
PC	Program Counter
PROG	Program
MA	Memory Address
MB	Memory Buffer
MQ	Multiplier Quotient
REL	Release
RIM	Read-In Mode
SA	Starting Address
SHIFT/P	Shift/P (similar to CTRL/L)
SING INST.	Single Instruction
SING STEP	Single Step
SR	Switch Register
SW	Console Switches
TTY	Teletype

Index/Glossary

E

G

H

I

N

O

T

U

W

NOTES