

VOLUME

1

**Introduction to Programming
and
On-Line Operations**

- | Chapter | Part One |
|----------------|--|
| 1. | Computer Fundamentals |
| 2. | Programming Fundamentals |
| 3. | Elementary Programming Techniques |
| 4. | System Description and Operation |
| 5. | Input/Output Programming |
| | Part Two |
| 6. | On-Line Operations |
| 7. | Paper Tape System |
| 8. | Disk Monitor System |
| 9. | 8K Programming System |
| 10. | TSS-8 Time Sharing System |
-

VOLUME

2

Programming Languages

- | Chapter | |
|----------------|---|
| 11. | FOCAL |
| 12. | BASIC |
| 13. | 4K Assemblers |
| 14. | 8K Assemblers |
| 15. | 8K FORTRAN |
| 16. | Floating Point Package
and Math Routines |

PROGRAMMING LANGUAGES

PDP-8 Family Computers

Prepared by
The Software Writing Group
Programming Department
Digital Equipment Corporation

PDP-8 HANDBOOK SERIES

First Printing, May 1970

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

Copyright © 1970
Digital Equipment Corporation

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

DEC	PDP
FLIP CHIP	FOCAL
DIGITAL	COMPUTER LAB

Foreword

This volume is a collection of programming language texts which DEC makes available for users of the PDP-8 small computer series. It is an extension of our companion volume, "Introduction to Programming," which is now widely available. This 2-volume set contains most of the standard systems programming information which PDP-8 programmers will need.

We have brought all of these documents together because we believe that the availability of compact, low-cost, rapid access, information sources (such as these paperback books), is an essential catalyst to assist the continuing rapid growth of data processing, and its many new applications. The need for computer programming skills, such as facility in these languages, will increase during the decade ahead.

I would like to thank the many systems programmers, engineers, scientists, teachers and students, who, over the years, have contributed to the development and improvement of these programs and documents.



President,
Digital Equipment Corporation

Additional copies may be ordered from DEC Program Library,
Bldg. 3-5, Maynard, Mass. 01754.

\$2.00 : quantity discounts available..

Preface

Like its companion volume, *Introduction to Programming*, this book is designed as a portable, compact source of information. Before you choose a language, we suggest you spend a few moments studying the way information is organized and indexed.

To find a major section of this book quickly, you should first go to the index tab guide on the back cover. Bend the book slightly, place your finger on the corresponding black printed tab, and open the book to the first page of the desired section.

For more detailed information, go to the Master Index/Glossary at the back of the book. The glossary terms are set in italics, and include most of the specialized terminology used in *Programming Languages*.

In general, this book is concerned with the languages only. Although we show, by many examples, how the languages may be used, we do not attempt to cover programming techniques in a detailed or complete manner. Nor do we discuss program preparation, loading, or testing. These activities are discussed in *Introduction to Programming*, and will be explained more fully in forthcoming editions of that handbook. Operating procedures for these programs are also contained in the series of System User's Guides which are available from the DEC Program Library, Maynard, Mass., 01754.

Many other programming languages have been implemented on the PDP-8 series, and some of these are available through the DEC User's Society (DECUS). For example, DECUS can send you a copy of a 4K ALGOL compiler which was developed at the University of Grenoble in France. Or you might want to use a LISP interpreter from the Technical University of Delft, in Holland. To find out more about these and other programs which DECUS members exchange, write to DECUS, Maynard, Mass., 01754.

■ We invite you to help us to improve the next edition of this handbook. If you should find an error, an ambiguity, or misleading information, send us a note. We would also like to have your critical evaluation of this book, and the ease (or difficulty) you experienced in learning a programming language. The address is:

Manager, Software Writing Group
Digital Equipment Corporation
Building 12-2
Maynard, Mass. 01754

Introduction

These PDP-8 languages may present a selection problem for some users. Which of these languages should *you* learn, or to state the problem differently, which language is best for the programs you will write.

If you want to pick a language which is easy to learn, start with FOCAL or BASIC. These are conversational, interactive languages which are popular in high schools and colleges. They are recommended for non-mathematical programmers and for busy engineers, scientists, accountants and others who need a very high speed, supercalculator.

It is generally agreed that the most efficient computer programs, both in terms of storage use and speed of execution, are programs written in machine language: in this case the hard-wired instructions which the PDP-8 can perform. If you are planning to write a program which will be used very often, or which must operate in a small number of core locations, you probably should write your program for assembly by PAL III, MACRO, PAL-D or SABR.

FORTRAN is the world's most popular programming language, and is considered primarily a scientific language. On the PDP-8 there is a basic 4K FORTRAN system (not included in this volume), and a more powerful 8K system (see Chapter 15).

Comparing Programming Languages

All computer languages have certain things in common. It is helpful to understand what these things are if one wishes to choose among or learn several different languages.

Every language has commands for communicating the desires of the user to the computer. All computer commands can be placed into four categories:

1. Input and output commands to transmit raw information;
2. Computational commands to manipulate the information, thus producing new information;

3. Control commands to tell the program the order of executing other commands;
4. Pseudo commands that do none of the other jobs but merely tell the computer things it needs to know, like where the end of a program is.

Every language has data structures to hold the information that will be communicated and manipulated. All data structures may be placed into four categories.

1. Variables to hold numbers;
2. Text to hold the commands (also called programs);
3. Strings to hold symbolic information;
4. Stacks to hold control information.

Every language implementation may be classified by its primary modes of operation.

1. Compilers take user-oriented commands and translate them into machine-oriented commands;
2. Incremental compilers translate only those statements actually executed;
3. Semi-compilers produce an intermediate code that is then interpreted.
4. Interpreters scan the user oriented commands and call appropriate subroutines.
5. Assembly language is essentially machine oriented.

Future Developments

If new programming languages continue to proliferate at the present fast rate, we can expect to see more powerful conversational systems such as FOCAL coming into general use.

There will be many other special purpose languages, and adaptations of common languages. These are used for numerical machine control, typesetting, laboratory automation, library indexing, information management, inventory control and many other applications.

Entirely new fields of study, such as mathematical linguistics, which attempts to explore the common characteristics of both natural and machine languages, and microprogramming, which is an interesting way of programming at the hardware pulse level, will provide new ideas for future programming languages.

Contents

Foreword	iii
Preface	iv
Introduction	vi
Chapter 11 FOCAL	11-1
A conversational language used on-line with the computer offering facilities for performing a wide variety of mathematical calculations. FOCAL can be used as a simple but powerful desk calculator or as a sophisticated language for writing programs.	
Chapter 12 BASIC	12-1
A conversational language for performing mathematical calculations. BASIC, originally developed at Dartmouth College, is easy for the beginning programmer to learn and use.	
Chapter 13 4K Assembly Programs	13-1
PAL III	13-7
The basic assembler for PDP-8 family computers having paper tape peripheral devices. Introduces concepts basic to the use of the advanced assemblers.	
MACRO	13-41
Similar to PAL III. Assembles user defined macro instructions. Provides double precision integers, floating point constants, a text facility, literals, Boolean and arithmetic operators, and automatic off-page linkage generation.	
4K PAL-D	13-63
Similar to PAL III and MACRO. Contains all of the features of MACRO excluding user defined macros. Includes a larger symbol table.	

Chapter 14	8K Assembly Programs	14-1
	8K PAL-D	14-5
	Similar to 4K PAL-D, but designed to operate in either 8K or 12K of core. Additional pseudo-operators provide features such as pagination of listings, conditional assembly, and binary output control.	
	SABR	14-9
	A one-pass assembler adapted to extended memory computer configurations. Produces relocatable binary code with automatically generated off-page and off-field linkages. Furnished with special Linking Loader.	
Chapter 15	8K FORTRAN	15-1
	A FORTRAN compiler coupled with the SABR assembler and Linking Loader to produce relocatable binary code. An extensive library of arithmetic and trigonometric subroutines supplements the compiler.	
Chapter 16	Floating Point System and Math Routines	16-1
	Floating Point Package for the PDP-8 family of computers has its own input/output and arithmetic routines. Also contains facilities for extended precision and for use of the Extended Arithmetic Element.	
Appendix A2	Permanent Symbol Table	A2-1
Appendix B2	ANSCHII Code	B2-1
Appendix C2	Loading Procedures	C2-1
Index Glossary	Index-1

Chapter II

FOCAL

CONTENTS

Part One

Getting Started in Programming	11-5
Communicating With the Computer	11-5
High-speed Calculations	11-6
Enclosures	11-6
Another Command: SET	11-7
The Talking Typewriter	11-7
Keeping Track of the Decimal Point	11-8
Correcting Mistakes	11-9
Summary of Part One	11-9

Part Two

Sequential Commands	11-11
Indirect Commands	11-11
GO	11-11
GOTO	11-11
DO	11-12
RETURN	11-13
QUIT	11-13
COMMENT	11-13
WRITE	11-13
More About Symbols	11-13
Subscripted Variables	11-14
Error Detection in Indirect Statements	11-15
Corrections	11-15
ERASE	11-16
ASK	11-17
IF	11-18
FOR	11-20
MODIFY	11-21
Using the Trace Feature (Debugging)	11-23
Mathematical Functions	11-24
Square Root, Absolute Value, Sign, Integer, Ran- dom Number Generator, Exponential, Logarithm, Arc Tangent, Sine, Cosine	11-25
Calculating Trigonometric Functions in FOCAL ...	11-28

PROGRAMMING TECHNIQUES	11-29
FOCAL SYSTEMS	11-30
Multi-user Segments	11-31
QUAD (Four-user FOCAL)	11-31
LIBRA (Seven-user FOCAL)	11-32
Utility Package	11-34
CLINE Graphics Package	11-34
Current FOCAL Tapes and Documents	11-39
ESTIMATING PROGRAM LENGTH	11-40
LOADING PROCEDURES	11-42
Paper-Tape System	11-42
FOCAL Loading Procedure	11-42
Restart Procedure	11-43
Saving FOCAL Programs	11-43
Multi-user Systems	11-44
LIBRA	11-44
DISKIN	11-45
QUAD	11-46
Utility Package	11-46
Graphics Package: CLINE, PLOTR, GRAPH	11-47
Disk Monitor System	11-47
4K FOCAL	11-47
8K FOCAL	11-49
QUAD Error Procedures	11-51
EQUIPMENT REQUIREMENTS	11-53
THE INITIAL DIALOGUE	11-53
EXAMPLES OF FOCAL PROGRAMS	11-55
 Part Three	
Summary of Commands, Operations, and Error Messages .	11-72
FOCAL COMMAND SUMMARY	11-72
OPERATIONS	11-74
ERROR MESSAGES	11-75

Part One

Getting Started in Programming

This is the starting point for learning how to solve numerical problems using the FOCAL¹ (Formulating On-line Calculations in Algebraic Language) system on PDP-8 series computers.

We will assume that you will type your programs on a Teletype terminal, which has a keyboard much like a standard typewriter. There are some important differences, which we will point out as we go along.

FOCAL is called a *conversational* language because the system reacts immediately to the things that you do.

Let's start by turning on the Teletype.

COMMUNICATING WITH THE COMPUTER

When you are ready to start, you must turn the round switch on the front of your Teletype to LINE. This connects you to the computer.

Next, press down on the Control (marked CTRL) key and at the same time, press on the C key. If you are now connected to the computer, FOCAL will respond with "?01.00*". The "?01.00" is a coded message from FOCAL meaning the FOCAL program is loaded into the computer, and the asterisk means FOCAL is ready for your next command. To help you decipher other coded messages, we have included a list of these codes, and the meaning of each, in the back of this chapter. Generally speaking, if you write a command which FOCAL cannot interpret, or if you break any of the programming rules for writing FOCAL statements, you will get a coded error message.

Operating and loading procedures are included at the end of this chapter.

¹ Trademark, Digital Equipment Corp.

HIGH-SPEED CALCULATIONS USING THE TYPE COMMAND

You only need to learn one FOCAL command, TYPE (abbreviated T), to do calculations such as the following.

$$10^3 \times \frac{3}{10} + 21 - 2$$

to do this in FOCAL, you type

```
*T 10↑3 *3/10 + 21-2
= 319.0000
```

(and after you press the RETURN key FOCAL computes this result. Every line must end with a RETURN.)

This example shows the arithmetic operations performed by FOCAL. These are done from left to right except that exponentiation (\uparrow) is done first, then multiplication (*), then division (/), then addition (+) or subtraction (-).

This means that

$$6 + 6 * 2$$

(which is 18 because multiplication is done before addition)

is not $(6 + 6) * 2$

(which is 24)

ENCLOSURES

To make sure that the computer performs these operations in the order you want, you can place them inside parenthesis marks. When the computer sees an expression enclosed in parentheses, it does that first. If the statement includes parentheses with parentheses (nesting), it computes the innermost first.

$$7 + (6/3) - (5 \uparrow 2) * 3$$

In this example, the computer first computes the values of the expressions enclosed in parentheses: $(6/3)$ is 2, and $(5 \uparrow 2)$ is 25. Then $7 + 2 - 25 * 3 = -66.0000$.

You can also use square brackets, [and], and angle brackets, < and >, to enclose expressions. All of these enclosure symbols

are evaluated equally, but the innermost will always be done first. They must always be used in pairs. The [and] enclosures are typed using SHIFT/K and SHIFT/M, respectively.

ANOTHER COMMAND: SET

This useful command tells FOCAL "store this symbol and its numerical value. When I use this symbol in an expression, insert the numerical value."

```
*SET PI=3.14159; SET E=2.71828; SET R=9.12739
```

Symbols may consist of one or two alphanumeric characters. The first character must be a letter, *but must not be the letter F*.

Just for practice, let's use FOCAL to calculate the volume of a sphere which has a radius of 9.12739. (We're going to use two of the symbols we have just defined in the SET commands above.)

The formula is $V = \frac{4}{3}\pi r^3$

which we can type like this:

```
R↑3*PI*4/3
```

You might be interested in running a timing test to show how long it takes to do such calculations by hand, with a calculator, and with FOCAL.

THE TALKING TYPEWRITER

To make the output of your program absolutely clear to other people, it is sometimes useful to give FOCAL certain messages or column headings. We call these character strings. These messages are enclosed in quotation marks.

```
*SET E=2.71828
*SET PI=3.14159
*TYPE "PI TIMES E" PI*E
```

and FOCAL types out

```
PI TIMES E=      8.5397*
```

You are not allowed to use the carriage return, line feed or leader-trailer characters in these character strings. But you can tell FOCAL to do a carriage return/line feed by inserting an ex-

clamation mark (!). You can get a carriage return by inserting a number sign (#).

```
*TYPE "JOHN"! "BILL"! "FRED"! "SAM"! "RICHARD"  
JOHN  
BILL  
FRED  
SAM  
RICHARD  
*
```

Spaces can be used in character strings as needed.

KEEPING TRACK OF THE DECIMAL POINT

FOCAL results are accurate to six significant digits. As we have shown in the examples so far, FOCAL assumes at the start that you want to see your results with 4 digits (or spaces) to the left of the decimal point and 4 digits to the right of the decimal point. This is called fixed-point notation.

You can change the output format within a TYPE statement by typing "%x.y" where x is the total number of digits to be output, and y is the number of digits to the right of the decimal point. Both x and y are positive integers equal to or less than 31. If y is a single digit, it must be preceded by 0. For example, %6.02 indicates four digits to the left and two to the right of the decimal point.

If your results exceed the format you have specified, FOCAL gives you results in floating-point format, like this:

$$= \pm 0.xxxxxxE \pm Z$$

where Z is an exponent of 10.

To switch to floating-point, you include a percent sign (%) followed by a comma, in a TYPE command.

```
*TYPE %,11  
=0.110000E+02*
```

which is 0.11 times 10^2 , or 11. The largest number that FOCAL can handle is + 0.999999E + 615, and the smallest is - 0.999999 E + 615.

CORRECTING MISTAKES

If you should strike the wrong key, you can delete it by striking the RUBOUT key. Each time you strike RUBOUT, another previously typed character will be deleted. When you strike RUBOUT, FOCAL echoes back a backslash (\) to tell you how many characters you deleted.

SUMMARY OF PART ONE

In this first part you have learned how one FOCAL command TYPE, is used to evaluate expressions, to type out character strings enclosed in quotes, and to use symbols (defined in SET commands) in expressions.

In the second part you will learn the other commands, and the use of line numbers to write a sequence of FOCAL statements. As you learn these techniques, you will be advancing rapidly in the art of computer programming.

Part Two

Sequential Commands

INDIRECT COMMANDS

Up to this point, only commands which are executed *immediately* by FOCAL have been discussed. If a Teletype line is prefixed by a line number, that line is not executed immediately; instead, it is stored in the computer's memory for later execution, usually as part of a sequence of commands.

Line numbers must be in the range from 1.01 to 31.99. The numbers 1.00, 2.00, etc., are illegal line numbers; they are used to identify the entire group. The number to the left of the point is called the group number; the number to the right is called the step number.

```
*1.1 SET A=1  
*1.3 SET B=2  
*1.5 TYPE %1, A+B
```

Indirect commands are executed by typing GO, GOTO, or DO commands which may be direct or indirect.

GO Command

The GO command causes FOCAL to go to the lowest numbered line to begin executing the program. If the user types a direct GO command after the indirect commands in the example above, FOCAL will carry out the command at line 1.1, and then the others, sequentially.

```
*GO  
=3*
```

GOTO Command

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 higher line. The GOTO causes a program *branch*; we have *jumped* from one sequence of lines to another. Sometimes we merely jump back and repeat a sequence of commands. This technique of repeating sequences is called *iteration*, and it is often used by experienced computer programmers.

```
GOTO 1.3  
= 2*
```

DO Command

The DO command is used to transfer control to a specified step, or group of steps, and then return automatically to the command following the DO command.

```
*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  
*CC  
STARTING= 3 FINISHED= 7*
```

If a DO command is written without an argument, FOCAL executes the entire indirect program.

```
*1.1 SET A=1  
*1.3 SET B=2  
*1.5 TYPE %1, A+B  
*DO
```

The following example causes a programming loop, which could be terminated by inserting line 1.5 QUIT, see below.

```
*1.1 SET A=1  
*1.2 TYPE A  
*1.3 DO 2.0  
*1.4 TYPE "FINISHED"  
  
*2.1 SET A=A-1  
*2.2 TYPE A
```

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, explained below.

RETURN Command

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 Command

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.

COMMENT Command

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.

WRITE Command

The WRITE command without an argument can be used to cause FOCAL to print out the entire indirect program, allowing the user to visually check it for errors.

A group of line numbers, or a specific line, may be typed out with the WRITE command using arguments, as shown below.

*7.97 WRITE 2.0	(FOCAL types all group 2 lines)
*7.98 WRITE 2.1	(FOCAL types line 2.1)
*7.99 WRITE	(FOCAL types all numbered lines)
*	

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. Therefore, before it is evaluated, the value of a symbolic name or identifier can be changed by retyping the identifier and giving it a new 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 all of the user defined identifiers, in the order of definition, by typing a dollar sign (\$) after a TYPE command.

```
*TYPE %6.05,$
```

The user's symbol table is typed out like this

```
A@(00)= 0.336051E+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. Notice that for numbers with more than one integer part, the output format operator %6.05 is ignored so that the whole number can be printed.

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*5)=2.71; SET X1(K+3*J)=2.79
```

The ability of FOCAL to compute subscripts is especially useful in generating arrays for complex programming problems.

When FOCAL types out symbol subscripts, only two digits are shown in the range 00-99. Despite this, subscripts up to ± 2047 may be used in calculations, but such programs are probably too long to fit in memory.

ERROR DETECTION IN INDIRECT STATEMENTS

When an error occurs in an indirect statement, the error message is typed out when the statement is encountered during program execution. In addition to the error code, FOCAL types the line number containing the error, as shown in the following example.

```
*1.10 SET A=2; TYPE "A",A,!
*1.20 SET B=4; TYPE "B",B,!
*1.30 GOTO 1.01
*1.40 TYPE "A+B",A+B
*GO
A=    2.0000
B=    4.0000
?03.05 @ 01.30
*
```

FOCAL executes lines 1.1 and 1.2 and then recognizes that line 1.3 is an illegal command. Therefore it issued the error message to show you that an illegal command was used.

To pinpoint an error in line 3.3, for example, type "DO 3.3?" and the program will be traced until the error is found.

CORRECTIONS

If the user types the wrong character, or several wrong characters, he can use the RUBOUT key as we explained in Part One, 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 S=13\X=13
*WRITE
C-FOCAL,1969

01.10 TYPE X-Y
01.20 SET X=13
*
```

The left arrow (←) erases everything which appears to its left on the same line.

```
*WRITE  
C-FOCAL,1969
```

```
01.10 TYPE X-Y  
01.20 SET X=13  
*
```

A line can be corrected by retyping the 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  
*
```

ERASE Command

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  
*
```

Used alone, without an argument, the ERASE command causes FOCAL to erase the user's symbols. Since FOCAL may not zero memory when loaded, it is good practice to ERASE ALL before starting a new program.

Typing WRITE after making corrections causes FOCAL to

print the indirect program as altered. This is useful for checking commands and for obtaining a "clean" program printout.

ASK Command

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 line 11.99 is encountered by FOCAL, it types a colon (:). The user then types a value in any format for the first identifier, followed by a terminator.² FOCAL then types another colon and the user types a value for the second 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.

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 in the previous section on corrections.

The ALT MODE key, when used as a terminator, is nonspacing and leaves the previously defined variable unchanged, as shown below.

```
*SET A=5  
*ASK A  
:123*  
*TYPE A  
= 5.0*
```

(user depressed the ALT MODE
key after typing 123)

² Terminators are space, comma, ALT MODE, and RETURN keys.

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
:8,:10,*
*TYPE X,Y,Z
=      8=      4=      10*
```

(User did not wish to enter new value for Y, so he typed ALT MODE in response to second colon.)

A text string may be included in an ASK statement by enclosing the string in quotation marks, just as in the TYPE command.

```
*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.

IF Command

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 the three line numbers 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 IF expression or variable must be enclosed in parentheses.

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 out.

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 command 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  
*  
*2.1 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 GIJK= 0.0 GJK= 1.0 BCEL*
```

FOR Command

This command is used for convenience 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. When 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, at which time 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 the comma and value C are omitted, it is assumed that the increment is one. 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 Command

Frequently, only a few characters in a particular line require changing. To facilitate this job, and to eliminate the need to retype 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 because they return to command mode upon completion.

During command input, the left arrow will delete the line numbers as well as the text if the left arrow is the rightmost character on the line.

Notice the errors in line 7.01 below.

```
*7.01 JACK AND BILL WSNT UP THE HALL
*MODIFY 7.01
JACK AND BAJILL WS\ENT UP THE HANILL
*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.

Caution

When any text editing is done, 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.

Notice in the example below that the values of A and B were set using direct statements. The use of the MODIFY command reset their values to zero and listed them after the defined symbols.

```
*ERASE ALL
*SET A=1
*SET B=2
*1.1 SET C=3
*1.2 SET D=4
*1.3 TYPE A+B+C+D; TYPE !; TYPE $
*MODIFY 1.1
SET      C=3\5
*GO
= 9.00
C@(00)= 5.00
D@(00)= 4.00
A@(00)= 0.00
B@(00)= 0.00
*
```

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.

In the following example, parts of 3 lines are 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 as 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 en-

countered (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!
= 0FOR 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.

MATHEMATICAL FUNCTIONS

The functions are provided to improve and simplify arithmetic capabilities and to give potential for expansion to additional input/output devices. A standard function call consists of four (or fewer) letters beginning with the letter F and followed by a parenthetical expression.

```
FSGN(A-B*2)
```

There are three basic types of functions: simple, extended, and I/O. The first type contains integer part, sign part, absolute value and square root functions.

In the second type, the extended arithmetic functions, are loaded at the option of the user. They compute logarithms, exponentials, arctangents, sines, and cosines.

The input/output functions are the third type. These include a nonstatistical random number generator (FRAN) whose value ranges from .5 to .9. There are also functions available to control scopes and analog-to-digital converters.

Square Root

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*
```

Absolute Value

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*
```

Sign

The sign part function (FSGN) outputs the sign part (+ or -) of a number with a value of 1.

```
*TYPE FSGN(4-6)
=- 1*
*TYPE FSGN(4-4)
= 1*
*TYPE FSGN(-7)
=- 1*
```

Integer

Integer part function (FITR) outputs the integer part of a number.

```
*TYPE FITR(5.2)
= 5*
*TYPE FITR(55.66)
= 55*
*TYPE FITR(77.434)
= 77*
*TYPE FITR(-4.1)
=- 4*
```


Random Number Generator

The random number generator function (FRAN) computes a nonstatistical pseudo-random number between 0.5000 and 0.9999.

```
*TYPE %, FRAN( )  
= 0.607295E+00*  
*TYPE FRAN( )  
= 0.737615E+00*
```

Exponential

The exponential function (FEXP) computes e to the power within parentheses. ($e = 2.718281$)

```
*TYPE FEXP(.666953)  
= 0.194829E+01*  
*TYPE FEXP(1.23456)  
= 0.343687E+01*  
*TYPE FEXP(-1.)  
= 0.367879E+00*
```

Logarithm

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 %, FLOG(2.065)  
= 0.725*
```

Arc Tangent

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*
```

Sine

The sine function (FSIN) calculates the sine of an angle in radians.

```
*TYPE %, FSIN(3.14159)
= 0.333786E-05*
*TYPE FSIN(1.400)
= 0.985448E+00*
```

Since FOCAL requires that angles 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*
```

Cosine

The cosine function (FCOS) calculates the cosine of an angle in radians.

```
*TYPE FCOS(2*3.141592)
= 0.999996E+00*
*TYPE FCOS(.50000)
= 0.877582E+00*
*TYPE FCOS(45*3.141592/180)
= 0.707107E+00*
```

These trigonometric functions may be combined to calculate other functions as shown in the following table.

Table 11-1

Calculating Trigonometric Functions in FOCAL

Function	FOCAL Representation	Argument Range	Function Range
Sine	FSIN(A)	$0 \leq A < 10 \uparrow 4$	$0 \leq F \leq 1$
Cosine	FCOS(A)	$0 \leq A < 10 \uparrow 4$	$0 \leq F \leq 1$
Tangent	FSIN(A)/FCOS(A)	$0 \leq A < 10 \uparrow 4$ $ A \neq (2N+1)\pi/2$	$0 \leq F < 10 \uparrow 6$
Secant	1/FCOS(A)	$0 \leq A < 10 \uparrow 4$ $ A \neq (2N+1)\pi/2$	$1 \leq F < 10 \uparrow 6$
Cosecant	1/FSIN(A)	$0 \leq A < 10 \uparrow 4$ $ A \neq 2N\pi$	$1 \leq F < 10 \uparrow 6$
Cotangent	FCOS(A)/FSIN(A)	$0 \leq A < 10 \uparrow 4$ $ A \neq 2N\pi$	$0 \leq F < 10 \uparrow 440$
Arc sine	FATN(A/FSQT(1-A ²))	$0 \leq A < 1$	$0 \leq F \leq \pi/2$
Arc cosine	FATN(FSQT(1-A ²)/A)	$0 < A \leq 1$	$0 \leq F \leq \pi/2$
Arc tangent	FATN(A)	$0 \leq A < 10 \uparrow 6$	$0 \leq F < \pi/2$
Arc secant	FATN(FSQT(A ² -1))	$1 \leq A < 10 \uparrow 6$	$0 \leq F < \pi/2$
Arc cosecant	FATN(1/FSQT(A ² -1))	$1 < A < 10 \uparrow 300$	$0 < F < \pi/2$
Arc cotangent	FATN(1/A)	$0 < A < 10 \uparrow 615$	$0 < F < \pi/2$
Hyperbolic sine	(FEXP(A)-FEXP(-A))/2	$0 \leq A < 700$	$0 \leq F \leq 5 * 10 \uparrow 300$
Hyperbolic cosine	(FEXP(A)+FEXP(-A))/2	$0 \leq A < 700$	$1 \leq F \leq 5 * 10 \uparrow 300$
Hyperbolic tangent	(FEXP(A)-FEXP(-A))/(FEXP(A)+FEXP(-A))	$0 \leq A < 700$	$0 \leq F \leq 1$
Hyperbolic secant	2/(FEXP(A)+FEXP(-A))	$0 \leq A < 700$	$0 < F \leq 1$
Hyperbolic cosecant	2/(FEXP(A)-FEXP(-A))	$0 < A < 700$	$0 < F < 10 \uparrow 7$
Hyperbolic cotangent	(FEXP(A)+FEXP(-A))/(FEXP(A)-FEXP(-A))	$0 < A < 700$	$1 \leq F < 10 \uparrow 7$
Arc hyperbolic sine	FLOG(A+FSQT(A ² +1))	$-10 \uparrow 5 < A < 10 \uparrow 600$	$-12 < F < 1300$
Arc hyperbolic cosine	FLOG(A+FSQT(A ² -1))	$1 \leq A < 10 \uparrow 300$	$0 \leq F < 700$
Arc hyperbolic tangent	(FLOG(1+A)-FLOG(1-A))/2	$0 \leq A < 1$	$0 \leq F < 8.3177$
Arc hyperbolic secant	FLOG((1/A)+FSQT((1/A ²)-1))	$0 < A \leq 1$	$0 \leq F < 700$
Arc hyperbolic cosecant	FLOG((1/A)+FSQT((1/A ²)+1))	$0 < A < 10 \uparrow 300$	$0 \leq F < 1400$
Arc hyperbolic cotangent	(FLOG(X+1)-FLOG(X-1))/2	$1 < A < 10 \uparrow 616$	$0 \leq F < 8$

PROGRAMMING TECHNIQUES

To decrease program length, maximize available core area, and assist in preparing complex routines, the experienced programmer can implement the following suggestions:

1. All commands can be abbreviated to their first letter.
2. A string of commands, except WRITE, RETURN, MODIFY, QUIT, T \$, and ERASE, can be combined on one line (up to 72 characters), with each command separated by a semicolon.
3. When creating a lengthy program, it is a good programming practice to leave free line numbers scattered throughout the body of the program. This will permit insertion of additional commands without complicated referencing routines. Remember that programs are executed sequentially by line number; consequently, an addition to the program placed physically at the end will be executed in turn. Line numbers must be in the range 1.01 to 31.99.
4. Some programs may require a keyboard response of YES or NO to a question asked during program execution. The answer typed to the question determines the next command to be executed (for example, in the initial dialogue). For this purpose, alphanumeric numbers are used in an IF statement to direct the execution.

```
*1.1 TYPE "DO YOU WANT A LINE?",!  
*1.2 ASK "TYPE YES OR NO",ANS,!  
*1.3 IF (ANS-ØYES)2.1.2.2,2.1  
*  
*2.1 QUIT  
*2.2 TYPE "-----",!  
*2.3 GOTO 1.1  
*GO  
DO YOU WANT A LINE?  
TYPE YES OR NO:YES  
  
-----  
DO YOU WANT A LINE?  
TYPE YES OR NO:NO  
  
*
```

If the user types the answer YES, the identifier ANS is given the alphanumeric value of YES. When the IF statement

is executed, the parenthetical express in ANS-0YES equals zero, and the command at line 2.2 is executed. If the user types YES in answer to the ASK question, then when its alphanumeric value is substituted in the parenthetical expression, the expression will not equal zero and line 2.1 will be executed. Note that for YES/NO responses, the sign of the parenthetical expression is irrelevant; only its zero or non-zero value is of interest.

5. To avoid filling storage with the push-down list during long routines, it is helpful to limit the number of levels of nested expressions in a command. Use of abbreviations and limited number of variable names will maximize storage space. An FCOM function to increase variable storage is explained in DEC-08-AJBB-DL, Advanced FOCAL Technical Specifications.

FOCAL SYSTEMS

The user who has mastered the fundamental FOCAL system can appreciate the expanded FOCAL capabilities. Primarily, there are two ways to increase FOCAL's powers:

1. Share FOCAL on a single computer with more than one person (multi-user segments)
2. Expand a single user system to allow longer programs, improved accuracy, and graphic display (additional segments).

Table 11-2 describes all segments of FOCAL. Each is available on binary coded paper tape. A simple one-pass loading procedure adds these extra capabilities to the FOCAL system.

Table 11-2

FOCAL Segments

Segment Name	Function
<u>Interpreter System</u>	
FOCAL and FLOAT	The interpreter, Teletype input/output handler, and modified floating-point package.
INIT	The binary tape for the initial dialogue program.

Table 11-2 (Continued)

FOCAL Segments

Segment Name	Function
Additional Segments	
Utility Package { <ul style="list-style-type: none"> 4-WORD 8K 	Extended accuracy overlay to FLOAT (gives 10 digits). Allows one user to take advantage of an 8K PDP-8.
Graphics Package { <ul style="list-style-type: none"> CLINE PLOTR GRAPH 	Permits scope to interact with FOCAL to display vectors, arcs and cursors. For use with an incremental plotter. For use with KV8/I.
Multi-User Segments	
LIBRA	Allows multiple users (up to seven) to run and save FOCAL programs on an 8K PDP-8 with Disk.
QUAD	Allows multiple users (up to four) to share FOCAL on an 8K PDP-8.

Multi-User Segments

FOCAL can be shared simultaneously by more than one user by parcelling computer time among the various users. Such a system, referred to as time-sharing, permits one computer to serve several persons, allowing each user to feel he has the system all to himself. No detectable delays occur under normal operating conditions. With a very heavy workload, some users may detect only a slight delay, less than a second, in response to their commands to FOCAL.

The two multi-user systems associated with FOCAL are detailed below.

QUAD (Four-User FOCAL)

QUAD permits from one to four persons to use FOCAL simultaneously on an 8K PDP-8, -8/L, or -8/I Computer. Up to four Teletype consoles and appropriate PT08 (or DC02 for 8/L) communicating units are required.

LIBRA (Seven-User FOCAL)

LIBRA allows up to seven persons to use FOCAL efficiently on one 8K PDP-8, 8/I, or 8/L Computer. LIBRA requires, in addition to from one to seven Teletype consoles, appropriate PT08's or DC02's, and at least one disk (RF08 or DF32). There are two versions of LIBRA available, depending on the user's disk system, i.e., RF08 or DF32 version. A disk initialization routine, DISKIN, prepares the disk for use by LIBRA. With LIBRA, user programs can be saved, retrieved, or deleted from the disk by library capabilities, i.e., each program is assigned a name by the user, and a three-word command tells LIBRA what to do with that program. In all cases, the name of a program must be one to four characters. A directory of saved (stored) user programs can also be listed by LIBRA.

LIBRA Commands

There are four LIBRA commands to perform the LIBRARY functions.

1. To save a program on the disk, use the command
LIBRARY SAVE name

This will store the entire program on the disk for future use.

2. To call a stored program from the disk, the command
3.10 LIBRARY CALL name

will bring the named program into the user's area. If this is followed by:

3.11 CONTINUE

Execution begins at the first line of the called program, as if the user had typed a GO command.

3. If a stored program is no longer needed, it may be removed from the disk by

LIBRARY DELETE name

4. The user may wish to have LIBRA list the names of all the programs it has stored on the disk. Use

LIBRARY LIST

to obtain the directory of stored program names. Note that the LIBRARY LIST command destroys any program in the active user's area by an ERASE ALL.

While using these library commands to the disk, very few errors are possible. When saving a program (refer to subparagraph (1) above), LIBRA may find a program with an identical name in its

directory list. Because each stored program must have a different name, the present program cannot be stored until the user gives it a new name. Also, the directory may be full already; therefore, this program cannot be stored.

After the program has been stored and the user wants to call or delete it from the disk, the only error possible is that LIBRA may find no such program name in its directory. Check your typing to be sure you spelled the program name correctly. The error codes for the above have the same format as normal FOCAL error code. They are listed in the summary of error messages.

Common Storage Function

LIBRA has swapping abilities which permit users to trade programs and data. The FCOM function allows a program to pass up to five arguments to another program. It is used as follows:

FCOM (J,Z) stores element Z in array element J
FCOM (J) retrieves array element J

Index J has a range of $0 < J < 4$.

The FCOM function is explained fully in LIBRA System Specifications (DEC-08-AJCA-DL).

Limitations on Focal with Libra

When operating at full capacity, LIBRA places only a few limitations on FOCAL, none of which interfere with normal system operation. These limitations are:

1. The command for the high-speed reader is inoperable.
2. The FADC (analog to digital conversion) and FDIS (display) functions may be used by only one user at any given time.
3. None of the additional FOCAL segments (see below) are compatible with LIBRA.
4. The use of the trace feature should be limited to prevent delaying execution of other users' commands. Trace only as few characters as are necessary. To stop a program while using the trace feature, it may be necessary to depress CTRL/C more than once.
5. The search character of the MODIFY command is echoed.

LIBRA is described more fully in *LIBRA System Specifications* (DEC-08-AJCA-DL).

ADDITIONAL SEGMENTS

FOCAL's capabilities can be expanded to provide greater accuracy, larger user program size, increased variable storage, and handle various graphic devices. These features are provided by overlay tapes as described below. The powers and uses of the segments are described below.

Utility Package

The utility package includes the following.

4WORD

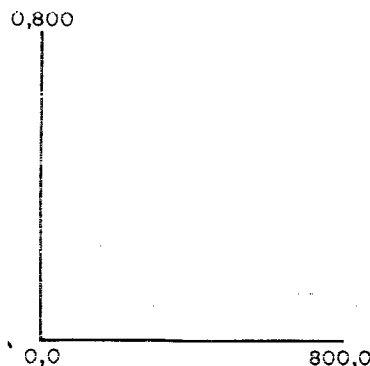
To increase FOCAL's accuracy to 10 digits for arithmetic operations. Because of this increased accuracy, there is a small decrease in the number of program variables that can be stored. Note that extended functions, trigonometric and exponential, are accurate to 6 places.

8K

To increase program size, the 8K overlay activates an additional 4K of core memory. This permits significantly longer programs to be used with no decrease in the number of program variables that can be stored. The user's system must have 8K hardware for this segment. The 8K system has all the capabilities of 4K FOCAL, with the exception that the MODIFY command and other text changes do not erase variables. Only an ERASE command will clear the storage area in 8K FOCAL.

CLINE Graphics Package

By interfacing a PDP-8 system with a VC8 control unit which will handle a variety of display devices (34D, Tektronix 611) and loading the CLINE overlay, vectors and arcs are displayed for visual inspection. The coordinate systems vary for these instruments; programs in this manual are based on a 34D scope, the coordinate system of which is:



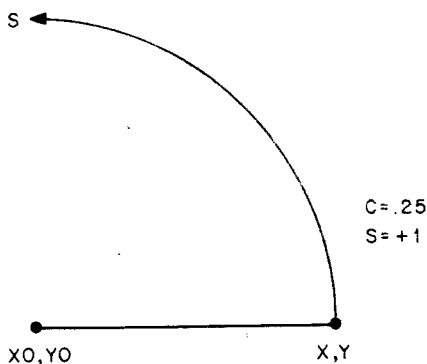
CLINE is especially useful for numerical control. CLINE can produce two basic types of lines: vectors and arcs; each type has a fundamental command string and the two can be combined at any time. CLINE will display a vector, the starting and ending points of which (X0, Y0 and X,Y, respectively) have been defined. To display a line, a DO 17 command must be incorporated into the program after each set of ending points has been assigned. The following four lines must be added to the program to display a line:

```
*16.2 SET P=X-X0; SET Q=Y-Y0; SET R=FSQT(Q^2+P^2)
*16.3 SET Z=FDIS(6.3*R*C,P,Q,X0,Y0,S/R)
*16.4 SET X0=X; SET Y0=Y
*17.1 DO 16.2; SET Z=FDIS(R,P/R,Q/R,X0,Y0,0); DO 16.4
```

Note that line 17.1 resets the values of X0 and Y0 to the previous ending points, X, Y, by a reference to line 16.4; thus, the end of one line automatically becomes the start of the next line. To start the next line from a different point, assign new values to both the starting and ending points.

To display an arc, the following variables must be defined:

X0, Y0 center
 X,Y starting point ($\sqrt{(X-X0)^2 + (Y-Y0)^2}$ =radius)
 C circumference, where
 C = .5 is a semicircle; C = 1 is a full circle
 S direction, where
 S = +1 counterclockwise; S = -1 clockwise



After assigning values to the above variables and creating the routine to perform, increment, and restart the display pattern, a

DO 16 command must be put in the program to perform the actual display function evaluation for each set of values and then project the results on the scope. The group 16 commands for arc generation are as follows:

```
*16.2 SET P=X-X0; SET Q=Y-Y0; SET R=FSQT(Q*2+P*2)
*16.3 SET Z=FDIS(6.3*R*C,P,Q,X0,Y0,S/R)
*16.4 SET X0=X; SET Y0=Y
```

As with line drawing, the last values assigned to X and Y, now the starting point of the arc, become the values of X0 and Y0, which here define the center of the next arc to be drawn.

For example, the following routine will display an arc enclosed within a square on a 34D scope.

```
*1.10 SET C=.5; SET S=1; SET A=800
*1.20 SET X0=0; SET Y0=0
*1.30 SET X=A; SET Y=Y0; DO 17; SET Y=A; DO 17; SET X=0; DO 17
*1.40 SET Y=0; DO 17
*1.50 SET X0=400; SET Y0=400; SET Y=Y0; SET X=200; DO 16
*1.60 GOTO 1.20
*16.2 SET P=X-X0; SET Q=Y-Y0; SET R=FSQT(Q*2+P*2)
*16.3 SET Z=FDIS(6.3*R*C,P,Q,X0,Y0,S/R)
*16.4 SET X0=X; SET Y0=Y
*17.1 DO 16.2; SET Z=FDIS(R,P/R,Q/R,X0,Y0,0); DO 16.4
*GO
```

Lines 1.1 and 1.5 defined a semicircle to be displayed inside the box described in lines 1.2, 1.3, and 1.4. When line 1.6 is reached, CLINE will first display the box, and then jump to the semicircle. The above program produces the pattern shown in Figure 11-1.

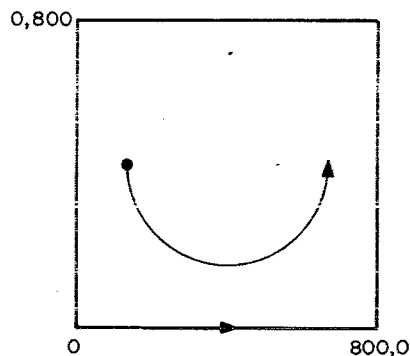


Figure 11-1 CLINE Example

PLOTR

An incremental plotter can be added to a graphics system by using the PLOTR overlay instead of CLINE (refer to Table 11-3). The three original CLINE functions (FDIS group) are available to the user. This system permits the user to design and debug a picture on a scope with CLINE before obtaining a hard copy with PLOTR.

An incremental plotter can be added to a graphics system by using the PLOTR overlay with CLINE (refer to Table 11-3). The three original CLINE functions (FDIS group) are available to the user. The PLOTR system permits the user to design and debug a picture on a scope before obtaining a hard copy.

GRAPH

A powerful graphics function is available for the appropriate KV 8/I device with a joystick cursor. The overlay, GRAPH, contains an X-Coordinate Cursor Read function, FCOM(0), and a Y-Coordinate Cursor Read function, FCOM(1). These two functions can be combined with the other GRAPH functions which are in the form FX(), for complete graphic control, including display of vectors and arcs, use of the cursors, and erasure of the screen. The interrupt button causes execution of group 31.

Table 11-3

Graphics Systems

Name	Function Call	Results	Device
built in	FDIS (X,Y)	points	VC8 and display LAB-8 (AX08)
CLINE	FDIS (R,P/R, Q/R,X,Y,0) FDIS (6.3*R*C, P,Q,X,Y,S/R) FDIS (, , X,Y)	vectors arcs points	VC8 and display
CLINE and LAB-8 (manual patches)	same as CLINE	same as CLINE	LAB-8 (AX08)

Table 11-3 (Continued)

Graphics Systems

Name	Function Call	Results	Device
PLOTR	same as CLINE	same as CLINE	Incremental Plotter and VC8
GRAPH	FCOM (0)	value is cursor x coordinate	H306 joystick, KV8 and display
	FCOM (1)	value is cursor y coordinate	H306 joystick, KV8 and display
	group 31 called by interrupt bar	synchronizes x, y cursors with external events	KV8 and display
	FX (0,441,X,Y, X0,Y0)	vectors	KV8 and displays
	FX (C*64,211, X,Y,X0,Y0)	arcs	
FX (, 4, , ,)	erases screen		
FX (, 1400, , ,)	displays cursor		

WORKABLE OVERLAY COMBINATIONS

These overlays can be combined with 4K FOCAL to produce a system with vastly increased powers that will optimize the individual user's system. Table 11-4 shows the workable FOCAL segment combinations. The initial dialogue and Disk Monitor System are included in these combinations to illustrate the many sets possible.

In addition to directing graphics instruments, FOCAL can direct various other instruments using special programs written by the user. The extended functions³ option allows the user to tailor the

³ See DECUS Document FOCAL-17, How to Write New Subroutines. FOCAL system to his own specifications.

Table 11-4

Allowable FOCAL Systems

- 1 — Must be loaded into field one
- 0 — Must be loaded into field zero
- X — Cannot be accepted
- Y — Command may be used if disk system is built
- N — Command is illegal
- * — Command is different

Binary Segment	Allowed Combinations are Indicated by Entries in Vertical Columns	Minimum Hardware Required
FOCAL	0 0 0 0 1 1 1 1	4K
INIT (optional)	0 0 0 0	
4WORD	0 0 1 1	4K
8K	0 0	8K
QUAD (non-8/S)	0 0 0 0	8K/PT08s
LIBRA (non-8/S)	0 0	8K/PT08s/DF32
CLINE (optional)	0 0	Graphics Terminal
PLOT (calcomp)	0 0	
GRAPH (KV 8/I)	0 0	
LIBRARY COMMAND (for Disk Monitor)	Y Y Y Y N N * *	DF32
FOCAL is always loaded first in the proper field.		

Current FOCAL Tapes and Documents

The following program tapes and documents comprise the FOCAL 1969 software package currently offered by DEC. This list is subject to revision at any time. The last two letters in these product codes stand for: D, document; PB, paper tape; LA, listing.

FOCAL-8 Manual	DEC-08-AJAD-D
FOCAL, 1969 + INIT (4K, INIT)	DEC-08-AJAE-PB
Listing (Includes Utility Overlays)	DEC-08-AJAE-LA
Utility Overlays for FOCAL, 1969 Tape (4WORD, 8K)	DEC-08-AJ1E-PB
Advanced FOCAL Technical Specifications	DEC-08-AJBB-DL

Graphic Overlays for FOCAL, 1969 (CLINE, PLOTR, GRAPH)	DEC-08-AJ2E-PB
Listing (of above)	DEC-08-AJ2E-LA
Extended Functions for FOCAL, 1969 (REPLACE, REMOVE)	DEC-08-AJ4E-PB
Multi-user Overlays for FOCAL, 1969 (LIBRA.DF32, DISKIN.DF32)	DEC-08-AJ5E-PB
Listing (of above)	DEC-08-AJ5E-LA
Multi-user Overlays for FOCAL, 1969 (LIBRA.RF08, DISKIN.RF08)	DEC-08-AJ6E-PB
Listing (of above)	DEC-08-AJ6E-LA
Four User Overlay for FOCAL, 1969 (QUAD.PT08)	DEC-08-AJ7E-PB
Listing (of above)	DEC-08-AJ7E-LA
Four User Overlay for FOCAL, 1969 (QUAD.DC02)	DEC-08-AJ8E-PB
Listing (of above)	DEC-08-AJ8E-LA

All the above articles may be purchased from the DEC Program Library, Bldg. 3-5, Maynard, Mass. 01754.

ESTIMATING PROGRAM LENGTH

FOCAL requires five words for each identifier stored in the symbol table, and one word for each two characters of stored program. This can be calculated by

$$5s + \frac{c}{2} 1.01 = \text{length of user's program}$$

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_8 through 3200_8 and 4600_8 through 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
?06.54
*TYPE %4,I*5,"LOCATIONS LEFT" (disregard error code)
= 705LOCATIONS LEFT*
```

A LOCATIONS command can be given with a paper-tape system to determine how much space remains in core for user programs and variables. Execution of this command causes FOCAL to print four octal numbers (core memory works on a base 8 number system) representing the following locations within core:

- | | | | |
|-----------------------------|---|---|---|
| 1. start of text buffer | } | } | space for user's program |
| 2. end of text buffer | | | |
| 3. end of variable list | } | } | space for storing variables assigned during program |
| 4. bottom of push-down list | | | |
| | | } | space for subroutines |

The LOCATIONS command permits the user to optimize his available storage space and to determine program length. If an 8K paper-tape system is being used, the values of 1 and 2 point to field 1. Locations 3 and 4 always point to field 0. The LOCATIONS command, for example, can be used after the three possible initial dialogue responses to indicate how much core each allows the user.

Dialogue response	Yes/Yes	No/Yes	No/No
Locations	*L	*L	*L
	3206	3206	3206
	3217	3217	3217
	3217	3217	3217
	4617	5177	5377

To get another * in order to continue with FOCAL after it has printed the four locations, the paper-tape system user must put 5177 in location 7600. If this is neglected, a manual restart is necessary.

Disk Monitor System users also have a command that indicates storage allocation: LINK. The LINK command for the Disk Monitor System is more limited than the LOCATIONS command for the paper tape system. LINK must be used only to return to the Disk Monitor; it cannot be used arbitrarily to determine core al-

location. LINK types out four locations, in the same fashion as the LOCATIONS command, but then types a period, indicating that control has been transferred to the Disk Monitor. A command to the disk must then follow.

When storing a program on the disk, the LINK command maximizes storage space by specifying the exact amount of memory that is filled by text, variables, and subroutines. The core locations printed out by LINK are used in calls to the Disk Monitor.

LOADING PROCEDURES

Read-In Mode (RIM) Loader

The RIM Loader is used to load the Binary Loader. For a detailed description see Appendix C2.

Binary (BIN) Loader

See the description of the BIN Loader in Appendix C2.

USING THE BINARY LOADER

The BIN Loader will be used to load a variety of programs with the following general procedures:

1. Put tape-to-be-loaded into the reader with leader-trailer code over the read head. Set reader control lever to "START."
2. Set SWITCH REGISTER to 7777 (all switches up).
3. Set MEM PROTECT down (if present).
4. Set INST FIELD to the field address of the BIN Loader (zero, or all down for the usual case).
5. Set DATA FIELD to the address of the field into which the tape-to-be-loaded is going. Depress LOAD ADDRESS.
6. If the high-speed reader is to be used, put down bit 0.
7. Depress LOAD ADDRESS; depress START switch.
8. Tape should begin reading in; if not, reload the BIN Loader and try again.
9. When tape stops check contents of ACCUMULATOR. If it has any lights (0-11) lit, go back to step 1 above.

Paper-Tape System

FOCAL LOADING PROCEDURE

The Binary Loader is used to load FOCAL. Check to see if

the Binary Loader is in core. If location 7777 contains 5301, the Binary Loader is probably in core; if not, load again.

The procedure for loading FOCAL is detailed below.

1. Load the FOCAL binary tape into field zero as described under the Binary Loader.
2. The FOCAL tape will halt once while loading. If all lights in the ACCUMULATOR are out, depress the CONTINUE key and the tape will finish loading. If any lights are lit in the ACCUMULATOR, the FOCAL program has been loaded incorrectly; return to Step 1 immediately preceding.
3. Set INST FIELD and DATA FIELD to zero (down). Set MEM PROTECT up (if present).
4. Place 0200 (the starting address of FOCAL) in the SWITCH REGISTER, and depress the LOAD ADDRESS key.
5. Press the START key. The FOCAL Initial Dialogue will begin. Answer the questions.

FOCAL RESTART PROCEDURE

Two methods for restarting the system are outlined below.

Hit the CTRL/C key at any time.⁴ FOCAL will type ?01.00 indicating a keyboard restart, and an asterisk on the next line indicating it is ready for user input.

OR

1. Depress the STOP switch.
2. Put 0200 in the SWITCH REGISTER (only switch #4 is up).
3. Depress the LOAD ADDRESS switch.
4. Depress the START switch.
5. FOCAL will then type *?00.00 indicating a manual restart, and an asterisk on the next line, indicating it is ready for user input.

SAVING FOCAL PROGRAMS

To save a FOCAL program on-line, proceed as follows:

1. Respond to "*" by typing WRITE ALL (do not depress the RETURN key yet).

⁴ CTRL/C indicates holding down the Control key while depressing the "C" key. This convention is used throughout the loading procedures.

2. Turn on low-speed tape punch by pushing down the "ON" button.
3. Type several "@" signs to get leader tape (simultaneously hold down the "SHIFT", "REPT", and "P" keys in that order, release in the reverse order).
4. Depress the "RETURN" key.

When the program has been printed and punched out:

5. Type several more "@" signs to get trailer tape.
6. Turn off the tape punch.

The user may now continue with another FOCAL program. The previous FOCAL program still in the computer.

Multi-User Systems

LIBRA LOADING PROCEDURE

LIBRA is loaded over FOCAL. For this system, FOCAL is loaded into field 1. The Binary Loader, however, must be in field 0. Do not load FOCAL's Initial Dialogue. The Disk "WRITE LOCK" switch must be off.

1. Load the FOCAL binary segment into field 1 using the description in the section on the Binary Loader.
2. Load the LIBRA binary segment into field 0 as described under the Binary Loader.
3. When the tape halts, depress the CONTINUE switch. The tape will continue to load in. Load DISKIN (the binary segment after LIBRA) the same way.
4. Set INST FIELD and DATA FIELD switches to zero.
5. Place 200 in the SWITCH REGISTER and depress the LOAD ADDRESS key.
6. Press the START key. The DISKIN Initial Dialogue will begin. Answer the questions as in the section on DISKIN Initial Dialogue, described below.
7. FOCAL/LIBRA is now ready to be shared by seven users.

LIBRA STOP AND RESTART PROCEDURE

To stop the system during operation, hold down switch 11 until the system halts. To resume operation, press CONTINUE or go to step 4 of the LIBRA LOADING PROCEDURE.

NOTE

After running LIBRA or DISKIN, two locations in the Binary Loader must be restored before the Loader can be used again. Load 1355 into location 7750, and load 5743 into location 7751.

DISKIN INITIAL DIALOGUE

The disk initialize overlay, like LIBRA, has two versions: one to clear a DF32 Disk System, and the other for a RF08 Disk System. DISKIN does not save the contents of the disk.

The overlay is short and can be quickly loaded with BIN. It will not destroy the LIBRA system that is currently in core. This allows the disk to be cleared at any time without reloading the entire system.

1. Load the DISKIN overlay tape into field 0 using the Binary Loader as described in the section on USING THE BINARY LOADER WITH FOCAL.
2. Start at 0200 in field 0 as in steps 4 and 5 of Using the Binary Loader.
3. The overlay types "FOCAL DISK (DF32) INITIALIZE?" for a DF32 System and "FOCAL DISK (RF08) INITIALIZE?" for an RF08 System.
4. To clear the disk directory, type Y. Any other answer causes the program to go to step 9, below.
5. The overlay types the question "NO. DISK SURFACE?" meaning the number of disks to be used.
6. Type 1, 2, 3, or 4 for the number of disk surfaces to be used. Any other answer goes back to step 5.
7. The overlay then types the number of free blocks available for use in the form XXX FREE BLOCKS, where XXX is the octal number of programs that can be stored according to the number of surfaces selected.
8. The overlay then cleans the directory and confirms by typing "DIRECTORY WRITTEN".
9. The overlay then types "SWAP AREA INITIALIZE?".

10. If the swapping areas are to be initialized, type "Y". Any other answer returns control to FOCAL.
11. The overlay cleans the swapping areas and confirms success by typing "SWAP" AREAS WRITTEN" and returns control to FOCAL.
12. The message "TO FOCAL" is typed before entering the FOCAL system.
13. Typing a CTRL/C aborts the dialogue and transfers to FOCAL.
14. The message "DISK WRITE ERROR" indicates that the disk is not operational or write locked. Remedy the cause of the error and reload the initialize overlay.

QUAD LOADING PROCEDURE

1. Load the first Section of the FOCAL tape into field 1 as described in USING THE BINARY LOADER. When the tape halts the first time, remove it from the reader.
2. Place the QUAD tape in the reader. Hit the CONTINUE key. The QUAD tape will read in.
3. When the tape halts, put 0200 in the SWITCH REGISTER and set INST FIELD and DATA FIELD switches to 0; depress the LOAD ADDRESS key.
4. Depress START key. QUAD will respond with ?00.00 on all four Teletypes. FOCAL is ready for use at each of the four Teletypes.
5. The program can be restarted as outlined in FOCAL RE-START PROCEDURE.
6. The program may only be stopped after typing CTRL/C on each user Teletype.

UTILITY PACKAGE LOADING PROCEDURE

1. Load and Start FOCAL plus INIT as described under FOCAL LOADING PROCEDURE. Complete the Initial Dialogue.
2. Press the STOP key.

3. If 10 digit accuracy is desired, load 4WORD binary tape (the first segment) into field zero as described in the section on the Binary Loader.
4. If long programs are to be run (up to 8000 characters), load 8K (the 2nd segment) into field one as in the section on the Binary Loader.
5. LOAD ADDRESS 200 as per steps 4 and 5 under the Binary Loader.
6. Press the START key.

GRAPHICS PACKAGE LOADING PROCEDURE

This set of three segments permits program control of vector generation and other graphic controls for different devices. The three successive binary sections are called CLINE, PLOTR, and GRAPH, respectively.

1. Load and Start FOCAL as described under FOCAL LOADING PROCEDURES.
2. Hit the STOP key.
3. Load the desired binary segment into field zero using the Binary Loader as described in the section on USING THE BINARY LOADER.
 - a. Use CLINE if using a VC8/I (or 34D).
 - b. Use PLOTR if using an incremental plotter.
 - c. Use GRAPH if using a KV8/I.
4. Set SWITCH REGISTER to 0200; set DATA FIELD and INST FIELD to zero; press LOAD ADDRESS; press START.

Disk Monitor System

Use one of the following sections.

4K FOCAL WITH DISK MONITOR

Having built the Disk Monitor System, copy the FOCAL tape onto the disk using PIP, and then load and start FOCAL on the disk system by using the LOAD command. (Refer to DEC-D8-SDAB-D, Disk Monitor System, Programmer's Reference Manual.) Give a starting address of 200.

1. Complete the Initial Dialogue; press STOP; LOAD ADDRESS 7600; press START.

2. Return to Disk Monitor by the FOCAL command "L".
3. Initialize the Disk Files by typing the following SAVE commands after the period typed by the Monitor.

```
.SAVE START ! 4600-7577;200
.SAVE FOCAL ! 0-3377;
```

4. To run a FOCAL program, call FOCAL from the Disk.

```
.FOCAL
.START
?00.00      (FOCAL prints the error code for a con-
*           sole restart and an asterisk to indicate it is
           ready to accept commands.)
```

5. To save a program, it must be given a name. Note that page 0 is also being saved.

```
*LOCATIONS                                (User)
  (returns command to Disk Monitor)
3206 (a)
3217 (b)                                (FOCAL)
3217 (c)                                (Disk Monitor)
4577 (d)
.SAVE(Name): 0,(a) - (b);                (User)
```

6. To continue to use the same program, add the command

```
.START
?00.00
*
```

After FOCAL types the error code and asterisk, the user can continue with the same (saved) FOCAL program.

7. To run a program that has been stored (as previously described), LOAD ADDRESS 7600, press START, and type the following routine:

```
.FOCAL
.CALL(name)
.START (line feed will not occur)
?00.00 (FOCAL types the error code for a manual
       restart and an asterisk to indicate it is ready
       to accept commands.)
```

FOCAL WITHOUT SOME EXTENDED FUNCTIONS UNDER THE DISK MONITOR

To use FOCAL without some of the extended functions, load FOCAL as under FOCAL Loading Procedures but give a starting address of 7600. Then issue the following commands to the Disk:

```
.SAVE FOCAL ! 0-7577;200  
.FOCAL
```

Now configure the system you want to use by answering the dialogue's questions.

8K FOCAL WITH DISK MONITOR

1. Save 8K FOCAL by issuing the following commands after the "L" command (see step 8 of 4K FOCAL WITH DISK MONITOR, preceding).

```
.SAVE ST8K!    (d) - 7577;200  
.SAVE FCL8!    0-3377  
.SAVE NUL8!    10100;10113 (to initialize program-  
                    text area)  
.SAVE NAME:    10100-(b);10113
```

2. To run a FOCAL program requiring 8K of memory, get FOCAL from the Disk.

```
.FCL8  
.NUL8  
.ST8K  
?00.00  
*
```

The above 3 commands to DISK are the appropriate starting sequence for a FOCAL program.

3. To save procedure for a finished 8K program is similar to 4K

```
.SAVE NAME:    1 (a) -1 (b);10113
```

4. Add the following command to save a set of variables in field 0.

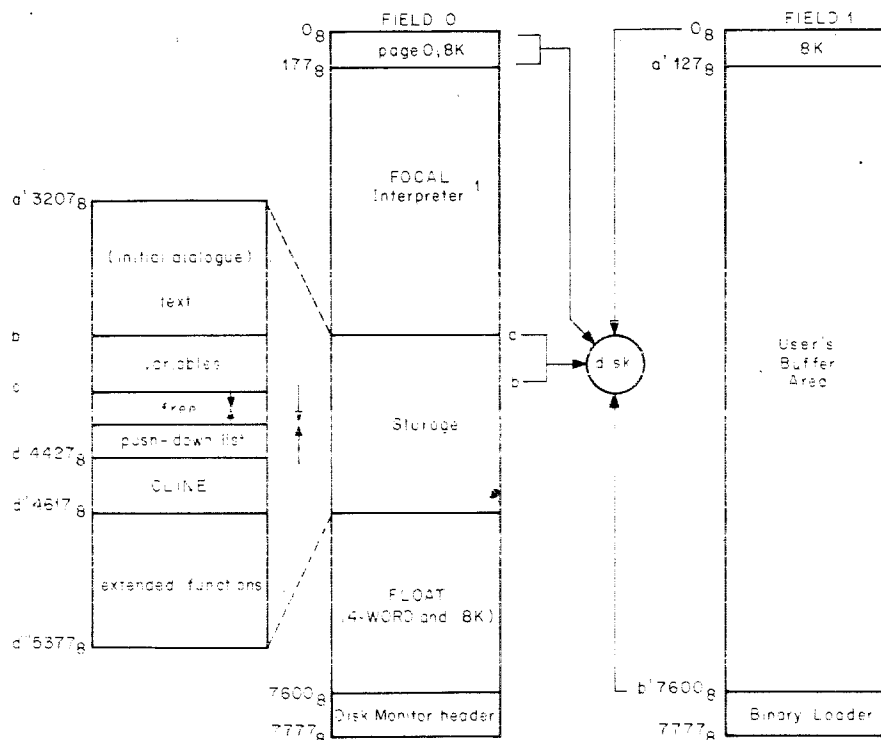
```
.SAVE DAT8:0,3000 - (c);
```


The .SAV DAT8 command stores a set of data (variables) located in field 0.

To set up a new program with a particular data set, type:

```
.FCL8
.CALL DAT8
.CALL NAM8
.ST8K
?00.00
*
```

Refer to DEC-08-SDAB-D, Disk Monitor System Programmer's Reference Manual for additional information.



a,b,c,d above point to the values indicated by the LOCATIONS command.

- Start of text buffer.
- End of text buffer.
- End of variable list.
- Bottom of push-down list. Value depends on which functions are retained and which additional system segments are used.

a' and b' point to the corresponding locations in field 1. The FOCAL interpreter is in field 1 when using QUAD.

Figure 11-8 Core Map for FOCAL with Disk Monitor System

QUAD ERROR PROCEDURES

If the four user FOCAL system appears to fail in some way, determine which of the four circumstances outlined below is applicable. Then try suggested procedure A under that category. If this does not restore operation, then try procedure B. If the system still does not respond correctly, as a last resort reload the overlay. Keep a log of all procedures tried.

1. If interrupt is OFF, and RUN is ON:
 - A. Hit STOP; note the contents of the PC; put that number into Sense Switches and hit LOAD ADDRESS; hit START.
 - B. If the system has Power-Fail/Auto-Restart, turn the key to OFF; turn the key to ON.
2. If interrupt is OFF, and RUN is OFF:
 - A. Try 1.B.
 - B. If the system has a disk, there may be a disk error flag raised; try 1.A above.
3. If the Interrupt is ON:
 - A. Type Control/C on all Teletypes.
 - B. Load Address 200, and START.
4. If the Teletypes occasionally produce garbage:
 - A. Adjust the timing on the Teletype control module.
 - B. Reload the overlay.
 - C. Call field service.

QUAD TELETYPE PROBLEMS

Since both the paper tape reader and the Teletype keyboard are perceived by the computer as similar input channels, some conflicts have been known to arise. While it is important for the user to have control of the computer from the Teletype, he may temporarily lose that control if:

1. The reader runs so fast as to fill up a program input buffer. The program will then remove the keyboard momentarily from the interrupt bus in order to process the input buffer; the user could prevent the buffer from filling by manually stopping the tape at intervals.

2. The user types so fast as to fill up an input buffer or holds down the REPEAT and RETURN keys. At this point the program will ignore a character coming from the Teletype, possibly reading it as a garbled character.

In the QUAD system there is a procedure for the user to follow that will prevent buffer overflow problems: the user may type a CTRL/R to prevent character echoing. The reduced load on the program permits FOCAL to keep up with the input buffer.

The more general, but costlier, solution is to install the so-called "XON + XOFF" control. This device, when installed on a Teletype, permits discrimination between keyboard and reader.

Other problems which may arise are documented below with suggestions for their solution.

3. When a modem (device which interfaces data between the computer and the Teletype) is attached to the computer, the program does not have control over the reader, which causes the reader to look like a fast typist as in example 2, above. The same solutions may be used to alleviate the problem.
4. There is another unique problem which arises with the input of the carriage return character because the carriage return is echoed by FOCAL as a "carriage return/line feed/asterisk" combination (a three-for-one expansion). *Do not hold down the REPEAT and RETURN keys* as this will fill the output buffer three times as fast as the input buffer. If this happens, FOCAL puts the user into an output-wait status. This reverts to example 1 which disables the user from the interrupt bus.
5. When long program tapes are read, the carriage return characters may cause the output buffer to become full as in example 4. The solution here is the same as for example 2; the user must follow the procedure of typing a CTRL/R before reading in the program tape.

6. When reading in data tapes, it is also possible to encounter the three-for-one problem as in example 4. The solution is to use CTRL/R to turn off the echo. In certain severe cases where much output is interspersed between input data, it will be necessary to stop the tape manually before the output buffers fill.

Once the user has disabled the echo feature of FOCAL and read in his tape, the echoing can be again enabled by typing CTRL/T.

If, for some reason, the program is manually stopped by the user or by a power failure, it is possible for user programs to become garbled. The solution is to install the Power-Fail/Auto-Restart option on the computer. This simple addition to the system turns QUAD into a Failsafe/Turnkey system. It is then possible to turn the key to PANEL LOCK and remove it. This prevents manual interference and allows the system to be activated by control of the power source as easily as controlling a light bulb.

EQUIPMENT REQUIREMENTS

FOCAL operates on a 4K PDP-8/I, -8/L, -8/S, -8, -12, -5, or LINC-8 Computer with a 33 ASR Teletype. Optional equipment includes an analog-to-digital converter and an oscilloscope display.

THE INITIAL DIALOGUE

After FOCAL has been loaded and started, FOCAL identifies itself and the type of computer being used; FOCAL then types the options available to the user for retention of the 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.

Alternate initial dialogues are shown below.

```
CONGRATULATIONS!!  
YOU HAVE SUCCESSFULLY LOADED 'FOCAL,1969' ON A PDP-8 COMPUTER.  
SHALL I RETAIN LOG, EXP, ATN ?:YES  
PROCEED.  
*
```

When the user answers YES to the above question, all mathematical functions are retained, and the user has approximately 700₁₀ locations available for his programs.

CONGRATULATIONS!!
YOU HAVE SUCCESSFULLY LOADED 'FOCAL,1969' ON A PDP-8 COMPUTER.

SHALL I RETAIN LOG, EXP, ATN ?:NO

SHALL I RETAIN SINE, COSINE ?:YES

PROCEED.

*

If the user answers NO to the first question, FOCAL asks a second question. A YES answer to the second question leaves approximately 900₁₀ locations available for the user's programs.

CONGRATULATIONS!!
YOU HAVE SUCCESSFULLY LOADED 'FOCAL, 1969' ON A PDP-8 COMPUTER.

SHALL I RETAIN LOG, EXP, ATN ?:NO

SHALL I RETAIN SINE, COSINE ?:NO

PROCEED.

*

A NO answer to the second question erases all extended functions from core, giving the user 1100₁₀ locations for use with his programs. To determine the exact number of locations available, use the LOCATIONS command.

Note that logarithm, arctangent and exponential functions cannot be retained without the sine and cosine. Refer to DEC-08-AJBB-DL, *Advanced FOCAL Technical Specifications* for another way to eliminate the extended functions.

After the initial dialogue has been answered, FOCAL automatically erases it from core. FOCAL concludes the initial dialogue by telling the user to PROCEED followed by an * and waits for user input.

EXAMPLES OF FOCAL PROGRAMS

Disclaimer

The FOCAL-8 demo programs that follow are examples or suggestions of procedures for writing games, quizzes and problem solving routines. These routines are in no way meant to be considered a final product of development.

Addition Exerciser

ABSTRACT: This is an educational routine designed for children in elementary school. The purpose and result of this routine complement each other. First, the purpose of this routine is to quiz the child in basic addition, so that he may learn to associate numbers and quantities at a more rapid pace. The result of this is that the student is introduced to the computer at an early age. He will eventually conclude that he can not only learn and have fun with the computer, but he may also conclude that it is a very applicable tool. And there is always that chance that later in life he may remember his past experience.

OPERATIONAL PROCEDURES:

1. "Addition Exerciser" is loaded via FOCAL-8.
2. Type "GO" and execution begins.
3. A sample run follows.

```
*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969

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.
=      60
=      73
-----

:133

THAT IS CORRECT.
=      89
=      53
-----

:152

SORRY, TRY AGAIN,
=      89
=      53
-----
```

Prime Factors of Positive Integers

ABSTRACT: After receiving a positive integer as input this FOCAL-8 routine will dump on the Teletype all the prime factors of the specified integer.

OPERATIONAL PROCEDURES:

1. "Prime Factors of Positive Integers" is loaded by FOCAL-8.
2. Type "GO" and respond to the request for a positive integer and the prime factors will be typed.
3. A sample run follows.

```
*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969

Ø1.1Ø ASK !!"A POSITIVE INTEGER >1 PLEASE" N ,!!!;SET DI=2;SET PH=Ø
Ø1.11 IF (FITR(N)-N) 1.1;IF (N-1) 1.1;SET P=N
Ø1.2Ø IF (P/DI-FITR(P/DI)) 1.4,1.3,1.4
Ø1.3Ø TYPE "PRIME FACTOR" DI,!!;SET P=P/DI;GOTO 1.2
Ø1.4Ø IF (1-PH) 1.1,1.5;SET PH=1;SET DI=DI+1;GOTO 1.2
Ø1.5Ø SET DI=DI+2;IF (DI-P) 1.6,1.6; TYPE !"DONE"!!;GOTO 1.1
Ø1.6Ø IF (DI-FSQRT(FABS(N))) 1.2,1.2;SET DI=P;GOTO 1.2
*
*GO
```

A POSITIVE INTEGER >1 PLEASE:5

PRIME FACTOR= 5

DONE

A POSITIVE INTEGER >1 PLEASE:63

PRIME FACTOR= 3

PRIME FACTOR= 3

PRIME FACTOR= 7

DONE

A POSITIVE INTEGER >1 PLEASE:47

PRIME FACTOR= 47

DONE

Repeating Decimals

ABSTRACT, This FOCAL-8 routine computes and types the repeating decimals that appear in a fraction. The user must input the numerator and the denominator respectively.

If the output "appears" to be repeating for a line or two, interrupt the output by typing a CONTROL/C (\uparrow C). FOCAL will give an error message and an asterisk (*). Type "GO" if you wish to continue.

OPERATIONAL PROCEDURES:

1. Load "Repeating Decimals" with FOCAL-8, type "GO".
2. Input the numerator and denominator followed by a carriage return. And the results will be typed on the Teletype.
3. A sample run follows.

```
*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969
```

```
01.05 ASK " ENTER NUMERATOR AND DENOMINATOR "A,B,!"
01.10 SET Z=5
01.20 IF (B-A)1.4,1.3; TYPE " 0 ."; GOTO 2.1
01.30 TYPE !"1!";QUIT
01.40 TYPE !"THIS PROGRAM ONLY EVALUATES FRACTIONS<1!";QUIT
```

```
02.10 SET N=10
02.20 IF (N*A-B) 2.3,4.1,4.1
02.30 SET N=10*N
02.40 TYPE 0.0;D 6
02.50 GOTO 2.2
02.87
```

```
04.10 SET C=1
04.20 IF (N*A-C*B) 5.1
04.30 SET C=C+1
04.40 GOTO 4.2
```

```
05.10 TYPE %1,C-1; DO 6
05.20 SET A=N*A-(C-1)*B
05.30 IF (-A) 5.5;TYPE !; QUIT
05.50 IF (A-B) 2.1,1.3,1.4
```

```
06.10 IF (Z-20) 6.2; SET Z=0; TYPE !
06.20 SET Z=Z+1;RETURN
```

```
*
*GO
ENTER NUMERATOR AND DENOMINATOR :1 :4
0 .= 2= 5
```

```
*GO
ENTER NUMERATOR AND DENOMINATOR :134 :250
0 .= 5= 3= 6
```

```
*GO
```


ENTER NUMERATOR AND DENOMINATOR :1 :7

0 . = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8
= 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7
= 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4
= 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8
= 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7
= 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4
= 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8
= 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7
= 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4
= 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8
= 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7
= 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4 = 2 = 8 = 5 = 7 = 1 = 4

⋮

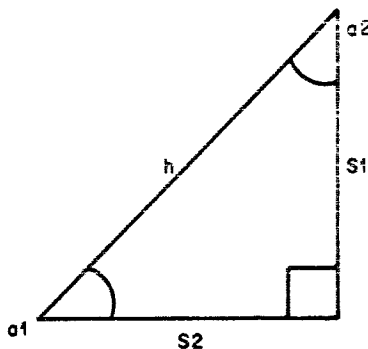
= 5 = 7 = ?00.00 @ 04.30

*

(Output continues until stopped from the keyboard)

Right Triangle

ABSTRACT: Given the length of the first side and the degrees of the adjacent angle, this routine computes the hypotenuse, the length of the second side, and the number of degrees for the other angle.



OPERATIONAL PROCEDURES:

1. "Right Triangle" is loaded by FOCAL-8. Type "GO", supply the length of S1 and the degrees of the adjacent angle. The results will be returned on the Teletype.
2. A sample run follows.

*ERASE ALL

*

*WRITE ALL

C-FOCAL, 1969

01.10 ASK "SIDE S1 EQUALS" S1

01.20 A " ADJACENT ANGLE A2 EQUALS" A2; TYPE "DEGREES"!!!

```

01.30 S RATIO=3.141592/180; SET A1=90-A2
01.40 SET HYP=S1/FSIN(A1*RATIO); SET S2=FSQT(HYP*2-S1*2)
01.50 T "SIDE S2 ", S2,!, "HYPOTENUSE", HYP,!
01.60 T "ANGLE A1", A1, !
*
*GO
SIDE S1 EQUALS:4
ADJACENT ANGLE A2 EQUALS:35
DEGREES

SIDE S2 = 2.801
HYPOTENUSE= 4.883
ANGLE A1= 55.00
*

```

Roots of a Quadratic

ABSTRACT: Given values a, b, c, of a first degree quadratic equation, this FOCAL-8 program computes the roots of the equation.

Based on the quadratic equation theorem: given $ax^2+bx+c=0$, then $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Then the following principles are applied:
if: a, b, and c are real then:

1. if $b^2 - 4ac$ is positive—then the roots are real and unequal.
2. if $b^2 - 4ac$ is 0—then the roots are real and equal
3. if $b^2 - 4ac$ is negative—then the roots are imaginary and unequal.

OPERATIONAL PROCEDURES:

1. Load "Roots of a Quadratic" with FOCAL-8.
2. Type "GO" and input the values of a, b, and c, and execution begins.
3. A sample run follows.

```

*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969

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 1.1
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 1.1
01.70 TYPE " IMAGINARY "!, -B/2*A," + (",FSQT(-ROOT)/2*A," )" "*I"
01.80 TYPE !, -B/2*A," - (",FSQT(-ROOT)/2*A," )" "*I",!;GOTO 1.1
*
*GO

```

```

A : 5 B : 6 C : 3
THE ROOTS ARE IMAGINARY
-- 0.600 + (= 0.490)*I
-- 0.600 - (= 0.490)*I

```

Perpetual Calendar

ABSTRACT: Given month/date/year, the "Perpetual Calendar" will type the day of the week.

OPERATIONAL PROCEDURES:

1. The "Perpetual Calendar" is loaded by FOCAL-8.
2. Type "GO" respond to the dialogue and your answer is typed back immediately.
3. A sample run follows.

```

*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969

01.10 ASK !"WHAT IS THE DATE ? (MM/DD/YYYY) 'M,K,C,!'
01.20 S C=C/100;S D=FITR(.1+100*(C-FITR(C)));S C=FITR(C)
01.30 S M=M-2; IF (M) 5.4, 5.4; GOTO 5.5

05.40 S M=M+12;S D=D-1;I (-D)5.5,5.5;S D=99;S C=C-1
05.50 S X=FITR<FITR[2.6*M-.2]+K+D+FITR[D/4]+FITR(C/4)-2*C>
05.60 IF (X-6) 5.7,5.7;S X=X-7;G 5.6
05.70 T !"THE DAY IS "; DO 6.1
05.80 IF (M*1E6+K*1E4+C - Q )5.9,5.85,5.9
05.85 T " , TODAY !"
05.90 T !!!; GOTO 1.1

06.10 I (X)6.26,6.2;I (X-2)6.21,6.22,6.15
06.15 I (X-4)6.23,6.24;I (X-6)6.25,6.26;
06.20 T "SUNDAY
06.21 T "MONDAY
06.22 T "TUESDAY
06.23 T "WEDNESDAY
06.24 T "THURSDAY
06.25 T "FRIDAY
06.26 T "SATURDAY
06.50 ASK M,K,C;DO 1.2;D 1.3; SET Q=M*1E6+K*1E4+C;GOTO 1.1
*
*GO

WHAT IS THE DATE ? (MM/DD/YYYY) :4/:3/:1970

THE DAY IS FRIDAY

WHAT IS THE DATE ? (MM/DD/YYYY) :10/:15/:70

THE DAY IS WEDNESDAY

WHAT IS THE DATE ? (MM/DD/YYYY) :?00.00 @ 01.10
*

```

King of Sumeria

ABSTRACT: The "King of Sumeria" is a game which challenges your ability to foresee the consumer market. Hamurabi, your servant, will state the following facts about last year, and you must decide the number of acres you will need, and how many bushels of grain you expect to distribute as food. You will base your decisions on these facts:

1. Number of people who died of starvation.
2. Number of new people that came to the city.
3. Number of acres owned by the city.
4. Number of bushels harvested per acre.
5. Total number of bushels that were harvested.
6. Number of bushels that were destroyed.
7. Number of bushels currently in storage.

Then based on your decisions, Hamurabi will state a new report of the above information.

OPERATIONAL PROCEDURES:

1. Load FOCAL-8, without extended functions.
2. Load the "King of Sumeria", following the loading instructions for paper tape.
3. Type "GO" and the game begins. (A sample run follows)

*ERASE ALL

*

*WRITE ALL

C-FOCAL,1969

01.10 S P=95;S S=2800;S H=3000;S E=200;S Y=3;S A=1000;S I=5;S Q=1

02.10 S D=0

02.20 D 6;T !!!"LAST YEAR"!D," STARVED,

02.25 T !I," ARRIVED,";S P=P+I;I (-Q)2.3

02.27 S P=FITR(P/2);T !!!**PLAGUE**!

02.30 T !"POPULATION IS"P,!!"THE CITY OWNS

02.35 T A," ACRES."!!!;I (H-1)2.5;T "WE HARVESTED

02.40 D 3.2

02.50 T !" RATS ATE "E," BUSHELS, YOU NOW HAVE

02.60 T !S," BUSHELS IN STORE."!

03.10 D 6;D 8;S Y=C+17;T "LAND IS TRADING AT

03.20 T Y," BUSHELS PER ACRE;"S C=1

03.30 D 4.3;A " BUY?"!Q;I (Q)7.2,3.8

03.40 I (Y*Q-S)3.9,3.6;D 4.6;G 3.3

03.50 D 4.5;G 3.3

03.60 D 3.9;G 4.8

03.70 S A=A+Q;S S=S-Y*Q;S C=0

03.80 A !"TO SELL?"!Q;I (Q)7.2,3.9;S Q=-Q;I (A+Q)3.5

03.90 S A=A+Q;S S=S-Y*Q;S C=0

04.10 T !"BUSHEL'S TO USE
 04.11 A " AS FOOD?"!Q;I (Q)7.2;I (Q-S)4.2,4.7;D 4.6;G 4.1
 04.20 S S=S-Q;S C=1
 04.30 A !"HOW MANY ACRES OF LAND DO YOU WISH TO
 04.35 A !"PLANT WITH SEED? "D
 04.40 I (D)7.2;I (A-D)4.45;I (FITR(D/2)-S-1)4.65;D 4.6;G 4.3
 04.45 D 4.5;G 4.3
 04.50 D 7;T A," ACRES."!
 04.60 D 7;D 2.6
 04.65 I (D-10*P-1)5.1;D 7;T P," PEOPLE."!;G 4.3
 04.70 D 4.2
 04.80 D 6;T "YOU HAVE NO GRAIN LEFT AS SEED !!"!S D=0

 05.10 S S=S-FITR(D/2);D 8;S Y=C;S H=D*Y
 05.20 D 8;S E=0;I (FITR(C/2)-C/2)5.3;S E=S/C
 05.30 S S=S-E+H;D 8;S I=FITR(C*(20*A+S)/P/100+1);S C=FITR(Q/20)
 05.40 S Q=FITR(10*FABS(FRAN()));I (P-C)2.1;S D=P-C;S P=C;G 2.2

 06.10 T !!"HAMURABI: "%5

 07.10 I (C)7.2;S C=C-1;D 6;T "BUT YOU HAVE ONLY";R
 07.20 D 6;T !"GOODBYE!"!!!;Q

 08.10 S C=FITR(5*FABS(FRAN()))+1
 *

*
*GO

HAMURABI:

LAST YEAR
 = 0 STARVED,
 = 5 ARRIVED,
 POPULATION IS= 100

THE CITY OWNS= 1000 ACRES.

WE HARVESTED= 3 BUSHELS PER ACRE;
 RATS ATE = 200 BUSHELS, YOU NOW HAVE
 = 2800 BUSHELS IN STORE.

HAMURABI: LAND IS TRADING AT = 21 BUSHELS PER ACRE;
 HOW MANY ACRES OF LAND DO YOU WISH TO BUY?

:0
TO SELL?

:0
BUSHELS TO USE AS FOOD?
:2000

HOW MANY ACRES OF LAND DO YOU WISH TO
 PLANT WITH SEED? :800

HAMURABI:

LAST YEAR
= 0 STARVED,
= 8 ARRIVED,
POPULATION IS= 108
THE CITY OWNS= 1000 ACRES.

WE HARVESTED= 4 BUSHELS PER ACRE;
RATS ATE = 0 BUSHELS, YOU NOW HAVE
= 3600 BUSHELS IN STORE.

HAMURABI: LAND IS TRADING AT = 22 BUSHELS PER ACRE;
HOW MANY ACRES OF LAND DO YOU WISH TO BUY?
:30
BUSHELS TO USE AS FOOD?
:2040
HOW MANY ACRES OF LAND DO YOU WISH TO
PLANT WITH SEED? :900

HAMURABI:

LAST YEAR
= 6 STARVED,
= 10 ARRIVED,
POPULATION IS= 112

THE CITY OWNS= 1030 ACRES.

WE HARVESTED= 5 BUSHELS PER ACRE;
RATS ATE = 0 BUSHELS, YOU NOW HAVE
= 4950 BUSHELS IN STORE.

HAMURABI: LAND IS TRADING AT = 20 BUSHELS PER ACRE;
HOW MANY ACRES OF LAND DO YOU WISH TO BUY?
:70
BUSHELS TO USE AS FOOD?
:2300
HOW MANY ACRES OF LAND DO YOU WISH TO
PLANT WITH SEED? :950

HAMURABI:

LAST YEAR
= 0 STARVED,
= 12 ARRIVED,
POPULATION IS= 124

THE CITY OWNS= 1100 ACRES.

WE HARVESTED= 4 BUSHELS PER ACRE;
RATS ATE = 194 BUSHELS, YOU NOW HAVE
= 4381 BUSHELS IN STORE.

HAMURABI: LAND IS TRADING AT = 21 BUSHELS PER ACRE;
HOW MANY ACRES OF LAND DO YOU WISH TO BUY?
?00.00 @ 03.30

Towers of Hanoi

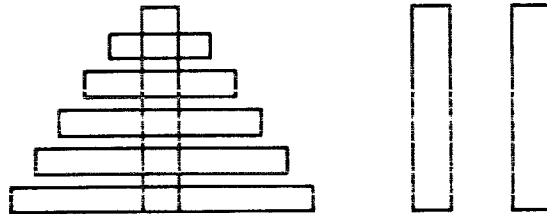
ABSTRACT: A Challenging Game in "FOCAL" and an example of recursive programming.

ORIGIN OF GAME:

According to legend, there exists a secret society of monks living far underground beneath the city of Hanoi. They possess three large stacks or towers on which different size disks may be placed.

Moving one at a time and never placing a larger upon a smaller disk, they are endeavoring to move the tower of disks from the left stack to the right stack. The legend says that when they have finished moving this tower of 64 disks, the world will end!

What is the minimum number of moves they will have to make?



Using this program you can try your hand at a small stack or watch the computer solve it.

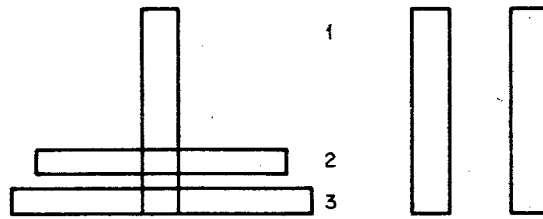
METHOD OF OPERATION:

The program is written in the FOCAL language and will run on a 4K PDP-8. To start the program, type "GO", followed by a carriage return. Type a space following any response made to a question asked by the program. To terminate the program at any time, type a CTRL/C. The program must be run *without* extended functions.

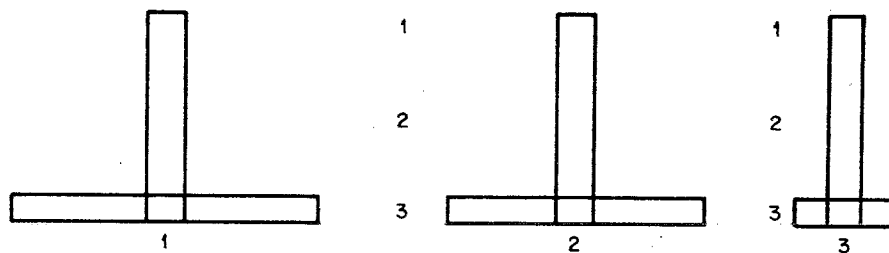
The program first asks for the number of disks (3 to 8). It then asks for the kind of output desired as either a plot of the disk positions or as a list of the moves. The final question is used to determine whether you will make the moves manually, or if the program should proceed automatically (0 to 1).

A move is selected by determining the stack out (SO) and the vertical disk number out (NO) and the stack in (SI) and the disk number in (NI). Error checking is not performed.

Thus the next move is this example



is SO:1 NO:2 SI:2 NI:3



ALGORITHM:

The stacks are represented internally as an array (SS). The value of a member of this array represents the size of the disk in that position.

DISK:

1	0	0	0
2	0	0	0
3	3	2	1

STACK: 1 2 3

EXPLANATION OF "TOWERS OF HANOI:" PROGRAM:

Group 1 —Main Program

- Ask for number of disks to be removed.
- Initialize the stacks.
- Move the stack (DO 2).

Group 2 —Move a specified stack.

- Save move request.
- To move this stack first move all but one disk (i.e. NO-NO-1).

If the resultant stack has an odd number of disks in it,
then move it to temporary storage
(i.e. $SI = 6 - SO - SI$).

If no disk remains to be moved in the output stack,
return. (line #2.3).

Find the first free position on the input stack. (line
#2.5).

Move the remaining stack and the bottom piece
(#2.6). Then move that stack back onto the
bottom piece (#2.7; 2.8).

Restore move request and return (#2.9).

Group 5 —User must specify which stack for output (“SO”) and
which disk to move (“NO”). No check is made if he
cheats. (This means the possibility of inventing new
games, by the way.)

A check is made to see if he has finished (#5.4; 5.5).

Group 6 —Execute a single move.

“SO” and “NO” and “SI” and “NI” are used to effect
the transfer of a piece.

Group 23—Plot the status of the board.

Each piece has a size value kept the stack array $SS(I)$.
The stacks are scanned (line #23.1) and posi-
tions on each stack checked.

*ERASE ALL

*

*WRITE ALL

C-FOCAL, 1969

```
01.05 T !" TOWERS OF HANOI."!;E
01.10 A " NO. OF DISKS? "N,!
01.20 F I=1,N;S SS(I)=I
01.30 S SO=1;S SI=3
01.40 S NO=N;S NI=N;S I=0
01.45 A "MOVES#0, PLOTS#1 ? "MOVE,!
01.46 IF (MOVE)ERR,1.47; DO 23
01.47 ASK "AUTO#0, MANUAL#1 ? ",A,!
01.50 I (-A)S,1;D 2;T !!!"DONE !!!";0

02.20 I [SS<(SO-1)*N+NO-1>JER,2.95;
02.30 S I=I+1;S NO(I)=NO;S SO(I)=SO;S SI(I)=SI
02.50 S SI=6-SO-SI;S NO=NO-1;D 3;S TE(I)=NI;D 2
02.60 S SI=SI(I);S NO=NO+1;D 3;D 6
02.70 S SO=6-SO-SI;S NO=TE(I); DO 3; DO 2
02.80 S SI=SI(I);S SO=SO(I);S NO=NO(I);S I=I-1
02.90 R
02.95 D 3;D 6;R

03.10 S NI=N
03.20 I [SS<(SI-1)*N+NI]JER,3.3;S NI=NI-1;G 3.2
03.30 R
```

```

05.10 A ? SO NO ?! ? SI NI ?! ; D 6
05.30 S A=0
05.40 F I=1,N*2;S A=A+SS(I)
05.50 I (-A) 5.1;T !"WELL DONE!"!;Q

```

```

06.10 S DO=(SO-1)*N+NO
06.20 S DI=(SI-1)*N+NI
06.30 S SS(DI)=SS(DO)
06.40 S SS(DO)=0
06.50 I (MOVE)E,6.7;DO 23;R
06.70 T !%2,?SO, NO,!SI, NI,?!

```

```

23.10 F J=1,N;T !;F K=0,70;DO 23.3
23.20 T !!!!!;R
23.30 IF [K-15+SS(J)*2]23.6;IF [-K+15+SS(J)*2]23.6;T ""#
23.60 IF [K-35+SS(J+N)*2]23.7;IF [-K+35+SS(J+N)*2]23.7;T ""#
23.70 IF [K-55+SS(J+N+N)*2]23.8;IF [-K+55+SS(J+N+N)*2]23.77;T ""#
23.77 S K=100;R
23.80 T " "
*

```

*GO

TOWERS OF HANOI.
NO. OF DISKS? :3
MOVES#0, PLOTS#1 ? :1

```

##### #
##### #
##### #

```

AUTO#0, MANUAL#1 ? :0

```

# #
##### #
##### #

```

```

# #
# #
##### #

```

```

# #
# ##### #
##### #

```

```

# #
# ##### #
##### #

```

```

# #
# #
##### #

```

```
#  
#  
#####
```

```
#  
#  
#
```

```
#  
#####  
#####
```

```
#  
#  
#
```

```
#  
#  
#
```

```
#####  
#####  
#####
```

DONE !

*GO

TOWERS OF HANOI.
NO. OF DISKS? :4
MOVES#0, PLOTS#1 ? :0
AUTO#0, MANUAL#1 ? :0

SO,= 1 NO,= 1!
SI,= 2 NI,= 4

SO,= 1 NO,= 2!
SI,= 3 NI,= 4

SO,= 2 NO,= 4!
SI,= 3 NI,= 3

SO,= 1 NO,= 3!
SI,= 2 NI,= 4

SO,= 3 NO,= 3!
SI,= 1 NI,= 3

SO,= 3 NO,= 4!
SI,= 2 NI,= 3

SO,= 1 NO,= 3!
SI,= 2 NI,= 2

SO,= 1 NO,= 4!
SI,= 3 NI,= 4

SO,= 2 NO,= 2!
SI,= 3 NI,= 3

SO,= 2 NO,= 3!
SI,= 1 NI,= 4

SO,= 3 NO,= 3!
SI,= 1 NI,= 3

SO,= 2 NO,= 4!
SI,= 3 NI,= 3

SO,= 1 NO,= 3!
SI,= 2 NI,= 4

SO, = 1 NO, = 4!
SI, = 3 NI, = 2

SO, = 2 NO, = 4!
SI, = 3 NI, = 1

DONE !

Return on Investment

ABSTRACT: The book, *Managerial Finance* by Weston Brigham, defines "Return on Investment" (or "Internal Rate of Return") as the interest rate that causes "the present value of the expected future receipts" to be equal to "the present value of the investment outlay" (page 148). This equality desired could also be called "Discounted Cash Flow Back" to be equal to "The Present Value of Capital Employed".

OPERATIONAL PROCEDURES:

1. Type "GO" and answer the following:
 - a. size of periods (e.g. .25)
 - b. number of years
 - c. amount to be depreciated
 - d. Immediate Expense (tax deductible)
 - e. additional working capital (e.g. inventory)

This is followed by a period-by-period estimate of savings (or income) of expenses. A negative number placed in the SAVE(T) column will cause the prior year's savings of expenses to be repeated automatically for the remainder of the periods.

2. Assumptions: All assumptions may be changed in the example program.
 - a. Tax rate is taken as 57% per year (line #12.5 in the sample program)
 - b. Declining balance depreciation is used. (something on the order of straight-line depreciation when it becomes faster; lines 5.1 and 5.2).
 - c. In computing present value, a discount factor is computed assuming daily compound interest and distributed receipts (line 5.4).
 - d. Annual compound interest may be substituted by:

$$(5.4 \text{ SET } DI = 1/(1+K) T)$$

*ERASE ALL

*

*WRITE ALL

C-FOCAL, 1969

```
02.20 A "SIZE OF PERIODS,YRS.";E
02.30 A Z," NUMBER OF YEARS"N
02.40 A !"AMOUNT TO BE DEPRECIATED"A
02.50 S N=N/Z;S R=.57*Z
02.55 A !"IMMEDIATE EXPENSES"E,!"WORKING CAPITAL
02.60 A WC,!" T SAVE(T) EXPENSE(T)
02.70 F T=1,N;D 3
02.80 D 4;R

03.20 I <ST>3.3;T !%4.02,T," "A S(T);I <-S(T)>3.4,3.4;S ST=-1
03.30 S S(T)=S(T-1);S E(T)=E(T-1);R
03.40 A " " E(T)

04.10 SET K=.25
04.20 SET BA=A;SET Y=A+WC+E*(1-R);SET X=0;FOR T=1,N;DO 5
04.30 SET Z=FABS(X/Y)
04.40 IF <FABS(Z-1)*K-.0001>4.8;SET K=K*Z;GOTO 4.2
04.80 T !!"%6.02" R.O.I."K*100," %
04.91 T !!" PROFIT(BEFORE) (AFTER) CASH
04.93 T !!" PERIOD DEPREC. TAXES TAXES FLOW FACTOR
04.94 T " VALUE"!
04.95 S BA=A;F T=0,N;S X=0;D 5;D 6

05.10 S DE=FEXP(-2*T/N)
05.20 IF <DE-(1-T/N)>5.3;S DE=1-T/N
05.30 S DE=BA-A*DE;S BA=BA-DE
05.40 S DI=FEXP(K/2-K*T)
05.50 S X=X+[S(T)-E(T)]*(1-R)*DI
05.60 S Y=Y-DE*R*DI
05.70 S Z=S(T)-E(T);S Y=Y+Z*(FSGN(Z)-1)/(,0)*1.25;T

06.10 S Z=X/(1-R)*DI
06.20 T T,DE,Z,(1-R)*Z,X/DI+DE,%6.04,DI,%6.02,X+DE*DI,!
*
```

Stock Market Commissions

ABSTRACT: During a stock purchase through a broker, a commission will be charged based on a series of rates for units of 100 shares (even lots) and a definite set of charges for smaller units (odd lots).

This program accepts a "buy" or "sell" indication plus the number and price of the shares involved. Given these facts, it then computes the net you must "pay" or "receive".

OPERATIONAL PROCEDURES:

1. "Stock Market Commissions" is loaded by FOCAL-8.
2. Type "GO" and respond to the dialogue and execution begins.
3. A sample run follows.

*ERASE ALL
*
*WRITE ALL
C-FOCAL,1969

01.05 E
01.10 A !!!**** BUY OR SELL?"OR
01.20 A !"HOW MANY " ?SHARES PRICE ?,!
01.40 T %8.02,?PRICE*SHARES?," \$"
01.45 S ODD=SHARES-FITR(SHARES/100)*100
01.50 I (-OD) 2.05;
01.55 T !!!"ROUND LOTS
01.60 S AM=PRICE*SHARES
01.70 I (AM-400) 1.73 ;I (AM-2400) 1.75;I (AM-5000) 1.77;C
01.71 S CO=AM*.001+39;G 1.8
01.73 S CO=AM*.020+ 3;G 1.8
01.75 S CO=AM*.010+ 7;G 1.8
01.77 S CO=AM*.005+19;G 1.8
01.80 T "COMMISSION IS "CO,!
01.85 I (FABS(OR-0BUY)),1.86;S NET=QU+AM-OC-CO;T "INCOME";G 1.87
01.86 SET NET=QU+AM+OC+CO ;T "OUTGO
01.87 IF (CO+OC-6) 1.9; IF (<OC+CO>/<OD+SH>-1.50) 1.88,1.9,1.9
01.88 T "IS ",?NET ?," \$" ,! ; GO
01.90 A "EXCEPTIONAL COMMISSION " CO;G 1.85

02.05 T !!!"ODD LOTS
02.10 SET BROKER=.125; IF (PRICE-55) 2.2; SET BR=.250
02.20 S SH=SH-ODDS
02.30 S QU=OD*PR
02.40 IF (QU-400)2.47;IF (QU-2400)2.45;IF (QU-5000)2.43
02.41 S CO=QU*.001+37;G 2.8
02.43 S CO=QU*.005+17;G 2.8
02.45 S CO=QU*.010+5;G 2.8
02.47 S CO=QU*.020+1;G 2.8
02.80 T "COMMISSION ON "%3,0D," ODD SHARES IS "%7.02,CO+BR*OD
02.90 S OC=CO+BR*ODDS ; IF (OF-0BUY) 3.1, 2.9 , 3.1
02.91 T !" OUTGO", OC+QU,!
02.93 IF (SH)E,0,1.55

03.10 T !" INCOME "QU-OC,!
03.20 GOTO 2.93
*
*GO

**** BUY OR SELL?:BUY
HOW MANY SHARES :120 PRICE :22.50
PRICE*SHARES= 2700.00 \$

ODD LOTS COMMISSION ON = 20 ODD SHARES IS = 12.00
INCOME = 438.00

ROUND LOTS COMMISSION IS = 29.50
OUTGO IS NET = 2741.50 \$

**** BUY OR SELL?:?00.00 @ 01.10

(Program continues until stopped from the keyboard.)

Part Three

Summary of Commands, Operations, and Error Messages

FOCAL COMMAND SUMMARY

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 lines
DO	D	DO 4.1	Execute line 4.1; return to command following DO command.
		DO 4.0	Execute all group 4 lines.
		DO ALL	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) IF (X) Ln; (commands)	Where X is a defined identifier, a value, or an expression, followed by one to three line numbers. 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.
LIBRARY CALL	L C	LIBRARY CALL name	Calls stored program from the disk.
LIBRARY DELETE	L D	LIBRARY DELETE name	Removes program from the disk.
LIBRARY LIST	L L	LIBRARY LIST	Types directory of stored program names.
LIBRARY SAVE	L S	LIBRARY SAVE name	Saves program on the disk.
LINK	L	L	For disk monitor system; FOCAL types 4 locations indicating start and end of text area, end of variable list and bottom of push-down list.
LOCATIONS	L	L	For paper-tape system; types same locations as LINK.
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 test. May be followed by ! to generate carriage return-line feed, or # to generate carriage return.
WRITE	W	WRITE	FOCAL types out the entire indirect program.
		WRITE ALL	
		WRITE 1.0	FOCAL types out all group 1 lines.
		WRITE 1.1	FOCAL types out line 1.1.

FOCAL OPERATIONS AND THEIR SYMBOLS

Mathematical operators:

↑ Exponentiation
* Multiplication
/ Division
+ Addition
- Subtraction

Control Characters:

% Output format delimiter
! Carriage return and line feed
Carriage return
\$ Type symbol table contents
() Parentheses
[] Square brackets } (mathematics)
< > Angle brackets }
" " Quotation marks (text string)
? ? Question marks (trace feature)
* Asterisk (high-speed reader input)

Terminators:

SPACE key (names)
RETURN key (lines) (nonprinting)
ALT MODE key (with ASK statement)
, Comma (expressions)
; Semicolon (compounds and statements)

FOCAL'S FUNCTIONS

FSQT()	Square Root	FCOS()	Cosine
FABS()	Absolute Value	FATN()	Arctangent
FSGN()	Sign Part of the Expression	FLOG()	Naperian Log
FITR()	Integer Part of the Expression	FDIS()	Scope Functions
FRAN()	A Random number Generator	FADC()	Analog to Digital Input Function,
FEXP()	Natural Base to the Power	FNEW()	User Function
FSIN()	Sine	FCOM()	Storage Function

FOCAL'S ERROR DIAGNOSTICS †

Code	Meaning
?00.00	Manual start given from console.
?01.00	Interrupt from keyboard via CTRL/C.
?01.40	Illegal step or line number used.
?01.78	Group number is too large.
?01.96	Double periods found in a line number.
?01.:5	Line number is too large.
?01.;4	Group zero is an illegal line number.
?02.32	Nonexistent group referenced by 'DO'.
?02.52	Nonexistent line referenced by 'DO'.
?02.79	Storage was filled by push-down-list.
?03.05	Nonexistent line used after 'GOTO' or 'IF'.
?03.28	Illegal command used.
?04.39	Left of " = " in error in 'FOR' or 'SET'.
?04.52	Excess right terminators encountered.
?04.60	Illegal terminator in 'FOR' command.
?04.:3	Missing argument in display command.
?05.48	Bad argument to 'MODIFY'.
?06.06	Illegal use of function or number.
?06.54	Storage is filled by variables.
?07.22	Operator missing in expression or double 'E'.
?07.38	No operator used before parenthesis.
?07.:9	No argument given after function call.
?07.;6	Illegal function name or double operators.
?08.47	Parentheses do not match.
?09.11	Bad argument in 'ERASE'.
?10.:5	Storage was filled by text.
?11.35	Input buffer has overflowed.
?20.34	Logarithm of zero requested.
?23.36	Literal number is too large.
?26.99	Exponent is too large or negative.
?28.73	Division by zero requested.
?30.05	Imaginary square roots required.
?31.< 7	Illegal character, unavailable command, or unavailable function used.

† For FOCAL, 1969 only.

Chapter 12

BASIC

CONTENTS

Introduction to BASIC Programming	12-5
About Computing	12-5
How to Use This Chapter	12-7
Fundamentals of Programming in BASIC	12-9
An Example Program and Output	12-9
REM Statement	12-9
Numbers	12-9
Variables	12-11
LET Statement	12-11
Arithmetic Operations	12-12
Parentheses and Spaces	12-13
Functions	12-14
More Complex Functions	12-15
RANDOMIZE Statement	12-17
User Defined Functions	12-19
Input/Output Statements	12-22
READ Statement	12-22
DATA Statement	12-23
RESTORE Statement	12-24
INPUT Statement	12-25
PRINT Statement	12-27
TAB Function	12-30
Subscripts and Loops	12-31
Subscripted Variables	12-31
DIM Statement	12-33
Loops	12-34
FOR Statement	12-34
NEXT Statement	12-35
Nesting Loops	12-36
Transfer of Control	12-37
Unconditional Transfer—GOTO Statement	12-37
Conditional Transfer	12-38
IF-THEN and IF-GOTO Statements	12-38

Subroutines	12-39
GOSUB Statement	12-39
RETURN Statement	12-40
STOP and END Statements	12-41
Nesting Subroutines	12-41
Errors and How to Make Corrections	12-42
Single Letter Corrections	12-42
Erasing a Line	12-42
Erasing a Program in Core	12-43
Stopping a Run	12-44
Running a BASIC Program	12-45
LOGIN Procedure	12-45
Initial Dialogue	12-45
RUN Command	12-46
Editing Phase	12-47
SAVE Command	12-47
REPLACE Command	12-48
UNSAVE Command	12-48
LIST Command	12-48
DELETE Command	12-49
NEW or OLD Command	12-49
CATALOG Command	12-49
BYE Command	12-50
ALTMODE Key	12-50
Punching a Paper Tape	12-50
Reading and Listing a Paper Tape	12-51
Transferring a File to Paper Tape or DECTape from Disk	12-52
Implementation Notes	12-52
Advanced BASIC	12-52
EDIT Command	12-53
COMPILE Command	12-54
File Extensions	12-55
Summary of BASIC Statements	12-56
Summary of BASIC Edit and Control Commands	12-57
Summary of BASIC Error Messages	12-60

INTRODUCTION TO BASIC PROGRAMMING

BASIC is an easy to learn, conversational, computer language for scientific, business, and educational applications. It is used to solve both simple and complex mathematical problems and is directed from the user's Teletype.¹

In writing a computer program, it is necessary to use a language or vocabulary that the computer will recognize. There are many computer languages and BASIC is one of the simplest because of the small number of readily learned commands needed, its easy application in solving problems, and its practicality in an educational environment.

BASIC is similar to other programming languages in many respects and is aimed at facilitating communication between the user and the computer. The novice computer user will benefit from reading the entire chapter from the beginning. The user who is already familiar with a language such as FOCAL or FORTRAN should first turn to the language summary at the end of the chapter.

As a BASIC user, you type in your computational procedure as a series of numbered statements, making use of common English words and familiar mathematical notation. You can solve almost any problem by spending an hour or so learning the necessary elementary commands. After becoming more experienced, you can add the more advanced techniques needed to perform more intricate manipulations and to express your problem more efficiently and concisely. Once you have entered your statements, you give a RUN command. This command initiates the execution of your program and causes the return of your results almost instantly.

About Computing

As we approach a computer terminal, there is a certain way we attack a problem. It is not enough to understand the technical commands of a computer language, we must also be able to correctly and adequately express the problem to be solved. For this reason it will be helpful to outline the process of setting up a problem for computer solution.

The first step is to define the problem to be solved in detail. Understand each fact and possibility within the problem before

¹ At present BASIC is available only on the TSS/8 system, but plans are being implemented to provide BASIC for individual use and batch processing. A stand-alone BASIC system called POLY BASIC is presently available from DECUS, the DEC User's Society.

attempting to go any further. Problems to be solved with BASIC are generally of a level which admit to fairly straightforward analysis.

In computing there is always more than one correct way of approaching a given problem. Generally a standard mathematical method for solution can be found, or a method developed. Programs using the same method can still be written in more than one correct way.

For some complicated programs a flowchart is useful. A flowchart is a diagram which outlines the procedure for solving the problem, step by step.

Having a diagram of the logical flow of a problem is a tremendous advantage to you when determining the mathematical techniques to be used in solving the problem, as well as when you write the BASIC program. In addition, the flowchart is often a valuable aid when checking the written program for errors.

A flowchart is a collection of boxes and lines. The boxes indicate, in a general fashion, what is to be done; the lines indicate the sequence of the boxes. The boxes have various shapes representing the type of operation to be performed in the program (input, computation, etc.). Appendix C in Volume 1 of this set is a guide to the standard flowchart symbols and procedures.

Following satisfactory completion of a flowchart, you proceed to write the program. To do this you need to understand the various instructions and capabilities of the BASIC language. The rest of this chapter is designed to teach you how to write programs in the BASIC language in a minimal amount of time.

Once the correct procedure has been coded it is time to try it on the computer. At this point it is possible the program will not work perfectly as originally written. BASIC will locate any mistakes the programmer has made in typing his program and print appropriate error messages to help him correct them. It is important to understand that even if the program does run, the results will only be correct if the problem has been correctly analyzed and proper code written to achieve the correct solution. The computer can only do what you tell it to do. If you have unknowingly told the computer to do something other than what you wanted it to do, the results will be accurate according to the information the computer processed. The computer cannot know what you really want, only what you have told it.

How to Use This Chapter

The most straightforward treatment of the BASIC programming language will be obtained by reading this chapter from the beginning. Examples are taken directly from Teletype output so that the reader will become familiar with the computer output and formats. Once you have mastered the principles of BASIC language, you will most likely only need to refer to the summaries found at the end of the chapter.

Detailed examples appear and may be run on the computer as a first exercise before attempting an original program.

The early sections of this chapter contain directions on how to write a BASIC program. The section on Implementation Notes is recommended for every reader. Once you have written several BASIC programs you will find the section on Advanced BASIC helpful; reading that section too early in your programming experience may be confusing. As soon as you are ready to try running a BASIC program on the computer turn to the section on Running a BASIC program.

```
10  REMARK - PROGRAM TO TAKE AVERAGE OF
15  REMARK - STUDENT GRADES AND CLASS GRADES
20  PRINT "HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT";
30  INPUT A,B
40  LET I=0
50  FOR J=1 TO A-1
55  LET V=0
60  PRINT "STUDENT NUMBER =";J
75  PRINT "ENTER GRADES"
76  LET D=J
80  FOR K=D TO D+(B-1)
81  INPUT G
82  LET V=V+G
85  NEXT K
90  LET V=V/B
95  PRINT "AVERAGE GRADE =";V
96  PRINT
99  LET Q=Q+V
100 NEXT J
101 PRINT
102 PRINT
103 PRINT "CLASS AVERAGE =";Q/A
104 STOP
140 END
RUN
```

Figure 12-1 An Example BASIC Program

```
HOW MANY STUDENTS, HOW MANY GRADES PER STUDENT? 5,4
STUDENT NUMBER = 0
ENTER GRADES
? 78
? 86
? 88
? 74
AVERAGE GRADE = 81.5

STUDENT NUMBER = 1
ENTER GRADES
? 59
? 86
? 70
? 87
AVERAGE GRADE = 75.5

STUDENT NUMBER = 2
ENTER GRADES
? 58
? 64
? 75
? 80
AVERAGE GRADE = 69.25

STUDENT NUMBER = 3
ENTER GRADES
? 88
? 92
? 85
? 79
AVERAGE GRADE = 86

STUDENT NUMBER = 4
ENTER GRADES
? 60
? 78
? 85
? 80
AVERAGE GRADE = 75.75

CLASS AVERAGE = 77.6

READY
```

Figure 12-1 An Example BASIC Program (continued)

FUNDAMENTALS OF PROGRAMMING IN BASIC

An Example Program and Output

At this point the program in Figure 12-1 may mean little to you, although the output (following the word RUN) should be fairly clear. One of the first things you notice about the program is that each line begins with a number. BASIC requires that each line be numbered with an integer from 1 to 2046. When the program is ready to be run, BASIC executes the statements in the order of their line numbers, regardless of the order in which you typed the statements. This allows the later insertion of a forgotten or new line. The programmer is, therefore, advised to leave gaps in his numbering on the first typing of a program. Numbering by fives or tens is a common practice.

The next thing we notice about the program is that each line begins with a word, a command to the computer to tell it what to do with the information on that line. BASIC does not understand the statement $V=0$ unless we write `LET V=0`. Once we understand the usage of these commands we are able to describe our problem to the computer.

REM Statement

The `REM` or `REMARK` statement allows the programmer to insert notes to himself or anyone who will read the program later. The form is:

(line number) REM (message)

Everything following `REM` is ignored by the computer. In Figure 12-1, line 10 is a remark describing what the program does. It is often useful to put the name of the program and information on what the program does at the beginning for future reference. Remarks throughout the body of a long program will help later debugging by explaining the function of each section of code within the whole program.

Numbers

In BASIC, as in all languages, there are conventions to be learned. The most important initial concepts are (1) how do we express a number to the computer and (2) how do we create algebraic symbols.

BASIC treats all numbers as decimal numbers, which is to say that it assumes a decimal point after an integer, or accepts any

number containing a decimal point. The advantage of treating all numbers as decimal numbers is that the programmer can use any number or symbol in any mathematical expression, knowing that the computer can combine the numbers given. (In some languages integers must be used separately from decimal numbers.)

A third form (other than integers and real numbers) we use in expressing numbers to the computer is called exponential form. In this form a number is expressed as a decimal number times some power of 10. For example:

$$23.4E2 = 2340$$

The E can be read as “times 10 to the—power” depending upon the positive or negative integer following E. A number can be expressed in exponential form by the programmer anywhere in his program. You may input data in any form. Results of computations are printed out as decimal numbers if they are in the range $.01 \leq N < 1,000,000$. Outside this range numbers are automatically printed out in E format. The computer handles seven significant digits in normal operation and input/output, as seen below:

<u>Value Typed In</u>	<u>Value Typed Out by BASIC</u>
.01	.01
.0099	9.900000E-3
999999	999999
1000000	1.000000E+6

The computer automatically omits printing leading and trailing zeros in integer and decimal numbers and formats all exponential numbers in the following form:

(sign) digit . six digits E \pm exponent value

For example:

−3.470218E+8 is equal to −347,021,800
 7.260000E−4 is equal to .000726

All letters are printed as capitals at the Teletype console. Therefore, a convention used by programmers, and which occurs on Teletype output, is that to distinguish zeros from the letter “oh” we slash zeros (\emptyset). This enables accurate input to the computer

(when you are typing a program previously written down) and ease of understanding in reading computer output (in which zeros are all slashed). Notice that unlike a typewriter, the letter "el" does not produce the number one (1) on the console keyboard. All numbers are on the top row of the keyboard. Notice also that the computer will not insert commas into large numbers, as we are accustomed to doing (i.e., 1,742,300 is printed as 1742300).

Variables

A variable in BASIC is an algebraic symbol for a number, and is formed by a single letter or a letter followed by a digit. For example:

Acceptable Variables

I
B3
X

Unacceptable Variables

2C — a digit cannot begin
a variable
AB — two or more letters
cannot form a vari-
able

We assign values to variables by either inputting these values or indicating them in a LET statement.

LET Statement

Before examining the LET statement we should first clarify the meaning of the equal sign (=). For example, the command:

```
10 LET A = B + C
```

tells the computer to add the values of B and C and store the result in a variable called A (The number 10 is the line number mentioned earlier).

```
20 LET D = 7.2
```

means to store the value 7.2 in the variable D.

```
30 LET D = 406
```

causes the value of D which was 7.2 (above) to be changed to 406.

The equal sign means replacement rather than equality. In algebra the formula:

$$X = X + 1$$

is meaningless, but when we say:

```
10 LET X = X + 1
```

we mean "add one to the current value of X and store the result back in the variable X."

Values of variables can be reassigned throughout the program as the programmer wishes. The equal sign, then shows a replacement relationship where the expression after the equal sign is evaluated and replaces the old value (if any) of the variable indicated.

The LET statement is of the form:

(line number) LET (variable) = (formula)

where a formula is either a number, another variable, or an arithmetic expression. The LET statement is the most elementary BASIC statement, used when computation is to be performed or, to put it more generally, whenever a new value is assigned to a variable.

Arithmetic Operations

Looking at the console keyboard we can find some of the usual arithmetic symbols (+, -, and =). BASIC can perform addition, subtraction, multiplication, division and exponentiation as well as other more complicated operations explained later. Each mathematical formula fed to the computer must be on a single line, with a line number and an appropriate command. The five operators used in writing most formulas are:

<u>Symbol Operator</u>	<u>Meaning</u>	<u>Example</u>
+	Addition	A + B
-	Subtraction	A - B
*	Multiplication	A * B
/	Division	A / B
↑	Exponentiation (Raise A to the Bth power)	A ↑ B

In BASIC, the mathematical formula:

$$A = 7 \left(\frac{B^2 + 4}{X} \right)$$

would be written:

```
10 LET A = 7 * ((B↑2 + 4)/X)
```

How does the computer know what operation to perform first? There are conventions built into computer languages; BASIC performs arithmetic operations with the order of evaluation indicated below:

1. Parentheses receive the top priority. Any expression within parentheses is evaluated before an unparenthesized expression.
2. In absence of parentheses the order of priority is:
 - a. Exponentiation
 - b. Multiplication and Division
 - c. Addition and Subtraction
3. If 1 or 2 does not clear ambiguity, the order of evaluation is from left to right as we would read the formula.

So in the example above, $B \uparrow 2$ is evaluated first, then $(B \uparrow 2 + 4)$ and then $((B \uparrow 2 + 4)/X)$, finally $7 * ((B \uparrow 2 + 4)/X)$. Keeping the conventions above in mind, $A \uparrow B \uparrow C$ will be evaluated as $(A \uparrow B) \uparrow C$, likewise $A/B * C$ is evaluated as $(A/B) * C$.

Parentheses and Spaces

Use of parentheses allows us to change the order of priority of evaluation in rule 2 above. They also prevent any confusion or doubt on our part as to how the expression is evaluated. To make a formula easier to write as well as read, it is frequently a good idea to provide more parentheses than strictly required. For example, which is easier to read:

$$\begin{aligned} &A * B \uparrow 2 / 7 + B / C * D \uparrow 2 \\ &(A * B \uparrow 2) / 7 + (B / C) * D \uparrow 2 \\ &((A * B \uparrow 2) / 7) + ((B / C) * D \uparrow 2) \\ &(((A * (B \uparrow 2)) / 7) + ((B / C) * (D \uparrow 2))) \end{aligned}$$

Each of the above formulas will be executed the same way, but

which makes the most sense to the programmer reading it, or perhaps trying to make corrections later? On the other hand, which has superfluous parentheses not required for clarity?

Spaces may also be used freely to make formulas easier to read.

```
10 LET B = D*2 + 1
```

instead of:

```
10LETB=D*2+1
```

Functions

BASIC performs several mathematical calculations for the programmer, eliminating the need for tables of trig functions, square roots and logarithms. These functions have a three letter call name, (the argument X can be a number, variable, formula, or another function) and are written as follows:

<u>Functions</u>	<u>Meaning</u>
SIN(X)	Sine of X (where X is expressed in radians) is returned.
COS(X)	Cosine of X (where X is expressed in radians) is returned.
TAN(X)	Tangent of X (where X is expressed in radians) is returned.
ATN(X)	Arctangent of X is returned as an angle in radians
EXP(X)	e^x (where $e = 2.712818$) is returned.
LOG(X)	Natural logarithm of X, $\log_e X$, is returned.
ABS(X)	Absolute value of X, $ X $, is returned.
SQR(X)	Square root of X, \sqrt{X} , is returned.

These functions are built into BASIC and can be used in any statement as part of a formula. For example:

```
10 LET A = SIN(ABS(X))/2
```

will cause A to be set equal to one half the value of the sine of the absolute value of X.

MORE COMPLEX FUNCTIONS

Three other functions are available, and although they are not as readily useful to the beginning programmer, they will become so as skill in designing program logic increases.

Sign Function, SGN(X)

The sign function returns the value +1 if X is a positive value, 0 if X is 0, and -1 if X is negative. For example: $SGN(3.42) = 1$, $SGN(-42) = -1$, and $SGN(23-23) = 0$.

```
10 REM- SGN FUNCTION EXAMPLE
20 READ A,B
25 PRINT "A="A,"B="B
30 PRINT "SGN(A)="SGN(A),"SGN(B)="SGN(B)
40 PRINT "SGN(INT(A))="SGN(INT(A))
50 DATA -7.32, .44
60 END
RUN
```

```
A=-7.32          B= .44
SGN(A)=-1       SGN(B)=1
SGN(INT(A))=-1
```

READY

Integer Function, INT(X)

The integer function returns the value of the greatest integer not greater than X. For example $INT(34.67) = 34$. INT can be used to round numbers to the nearest integer by asking for $INT(X+.5)$. For example: $INT(34.67+.5) = 35$. INT can also be used to round to any given decimal place, by asking for

$$INT(X*10^{\uparrow}D+.5)/10^{\uparrow}D$$

where D is the number of decimal places desired, as in the following program:

In order to obtain random digits from 0 to 9, change line 40 to read:

```
40 PRINT INT(10*RND(0)),
```

and tell BASIC to run the program again. This time the results will look as follows:

```
RANDOM NUMBERS
2           2           6           9           2
7           6           8           7           6
0           9           9           1           8
7           7           4           1           8
5           5           9           2           3
0           5           6           6           1
```

READY

It is possible to generate random numbers over any range. For example, if the range (A,B) is desired, use:

$$(B-A)*RND(\emptyset)+A$$

to produce a random number in the range $A < n < B$.

RANDOMIZE STATEMENT

If you want the random number generator to calculate different random numbers every time the program is run, BASIC provides the RANDOMIZE statement. RANDOMIZE is normally placed at the beginning of a program which uses random numbers (the RND function). When executed, RANDOMIZE causes the RND function to choose a random starting value, so that the same program run twice will give different results.

For example:

```
10 RANDOMIZE
20 PRINT RND(0)
30 END
```

will print a different number each time it is run. For this reason, it is a good practice to debug a program completely before inserting the RANDOMIZE statement. (RANDOMIZE uses the low order 12 bits of the time of day as a starting value, thus there are 4096 distinct starting points.)

The form of the statement is as follows:

(line number) RANDOMIZE
or (line number) RANDOM (abbreviated form)

To demonstrate the effect of the RANDOMIZE statement on two runs of the same program, we insert the RANDOMIZE statement as statement 15 below:

```
15 RANDOM
20 FOR I=1 TO 5
25 PRINT "VALUE" I "IS" RND(0)
30 NEXT I
35 END
RUN
```

```
VALUE 1 IS .7004438
VALUE 2 IS .6706673
VALUE 3 IS .7200098
VALUE 4 IS .2840528
VALUE 5 IS .2242288
```

READY

RUN

```
VALUE 1 IS .59055
VALUE 2 IS .3409859
VALUE 3 IS .7309656
VALUE 4 IS .3169203
VALUE 5 IS .3228311
```

READY

Clearly, the output from each run is different.

USER DEFINED FUNCTIONS

In some programs it may be necessary to execute the same mathematical formula in several different places. BASIC allows the programmer to define his own functions and call these functions in the same way he would call the square root or trig functions.

These user defined functions consist of a three-letter function name, the first two letters of which should be FN.

We define the function once at the beginning of the program before its first use. The defining or DEF statement is formed as follows:

(line number) DEF FNA(X) = formula(X)

where A is any letter. The argument (X) must be the same on each side of the equal sign and may consist of one or more variables. For example:

```
10 DEF FNA(S) = S+2
```

will cause a later statement:

```
20 LET R = FNA(4)+1
```

to be evaluated as R=17.

The two following programs

Program #1:

```
10 DEF FNS(A) = A+A
20 FOR I=1 TO 5
30 PRINT I, FNS(I)
40 NEXT I
50 END
```

Program #2:

```
10 DEF FNS(X) = X*X
20 FOR I=1 TO 5
30 PRINT I, FNS(I)
40 NEXT I
50 END
```

both cause the same output:

RUN

1	1
2	4
3	27
4	256
5	3125

READY

The argument in the DEF statement can be seen to have no significance; it is strictly a dummy variable. The function itself can be defined in the DEF statement in terms of numbers, variables, other functions or mathematical expressions. For example:

```
10 DEF FNA(X) = X2+3*X+4
15 DEF FNB(X) = FNA(X)/2 + FNA(X)
20 DEF FNC(X) = SQR(X+4) + 1
```

The statement in which the user defined function appears may have that function combined with numbers, variables, other functions or mathematical expressions. For example:

```
40 LET R = FNA(X+Y+Z)
```

The user defined function can be a function of more than one variable, as shown below:

```
25 DEF FNL(X,Y,Z) = SQR(X2 + Y2 + Z2)
```

A later statement in a program containing the above user defined function might look like the following:

```
55 LET B = FNL(D,L,R)
```

where D, L, and R have some values in the program.

The program in Figure 12-2 contains examples of a multi-variable DEF statement in lines 11, 21, and 31.

```

1  REM    MODULUS ARITHMETIC
10 REM    FIND X MOD M
11 DEF FNM(X,M)=X-M*INT(X/M)
12 REM
20 REM    FIND A+B MOD M
21 DEF FNA(A,B,M)=FNM(A+B,M)
22 REM
30 REM    FIND A*B MOD M
31 DEF FNB(A,B,M)=FNM(A*B,M)
100 PRINT "ADDITION AND MULTIPLICATION TABLES MOD M"
110 PRINT "GIVE ME AN M";
120 INPUT M
130 PRINT
140 PRINT "ADDITION TABLES MOD"; M
150 GOSUB 800
200 FOR I=0 TO M-1
205 PRINT I;" ";
210 FOR J=0 TO M-1
220 PRINT FNA(I,J,M);
230 NEXT J
240 PRINT
250 NEXT I
260 PRINT
270 PRINT
280 PRINT "MULTIPLICATION TABLES MOD" M
290 GOSUB 800
300 FOR I=0 TO M-1
305 PRINT I;" ";
310 FOR J=0 TO M-1
320 PRINT FNB(I,J,M);
330 NEXT J
340 PRINT
350 NEXT I
360 STOP
800 PRINT
810 PRINT TAB(5); 0;
820 FOR I=1 TO M-1
830 PRINT I;
840 NEXT I
850 PRINT
860 FOR I=1 TO 3*M+5
870 PRINT "-";
880 NEXT I
890 PRINT
900 RETURN
999 END
RUN

```

ADDITION AND MULTIPLICATION TABLES MOD M
GIVE ME AN M? 7

Figure 12-2 Modulus Arithmetic

ADDITION TABLES MOD 7

	0	1	2	3	4	5	6
0	0	1	2	3	4	5	6
1	1	2	3	4	5	6	0
2	2	3	4	5	6	0	1
3	3	4	5	6	0	1	2
4	4	5	6	0	1	2	3
5	5	6	0	1	2	3	4
6	6	0	1	2	3	4	5

MULTIPLICATION TABLES MOD 7

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

READY

Figure 12-2 Modulus Arithmetic (continued)

INPUT/OUTPUT STATEMENTS

One of the most important groups of statements is the group of I/O (Input/Output) statements. These I/O statements allow us to bring data into our programs during execution when and from where we choose. Similarly, we can choose the output format which best suits our needs. In the case of the example programs in Figures 12-1 (at the beginning of the chapter), data was typed in at the console keyboard as the computer requested it.

READ Statement

A simple way to put data into a program is with READ and DATA statements. One statement is never used without the other. The READ statement is of the form:

(line number) READ (variables separated by commas)

For example:

```
10 READ A,B,C
```

where A, B, and C are the variables we wish to assign values. In order to assure that all variables are assigned values before computation begins, READ statements are usually placed at the beginning of a program, or at least before the point where the value is required for some computation.

DATA Statement

Now that we have told the computer to read the values for three variables, we must supply those values in a DATA statement of the form:

(line number) DATA (numeric values separated by commas)

For example:

```
70 DATA 1,2,3
```

The DATA statement provides the values for the variables in the READ statement(s). The values must be separated by commas, in the same order as the variables are listed in the READ statement. Thus at execution time A=1, B=2, and C=3 according to the two lines above.

The DATA statement is usually placed at the end of a program before the END statement, so as to be easily accessible to the programmer should he wish to change his values.

A given READ statement may have more or fewer variables than there are values in any one DATA statement. READ causes BASIC to search all available DATA statements, in the order of their line numbers until values are found for each variable. A second READ statement will begin reading values where the first stopped. If at some point in your program you attempt to read data which is not present or if your data is not separated by commas, BASIC will stop and print an OUT OF DATA IN LINE XXXX message at the console, indicating the line which caused the error.

RESTORE Statement

If it should become necessary to use the same data more than once in a program, the RESTORE statement will make it possible to recycle through the DATA statements beginning with the lowest numbered DATA statement. The RESTORE statement is of the form:

(line number) RESTORE

For example:

```
85 RESTORE
```

will cause the next READ statement following line 85 to begin reading data from the first DATA statement in the program, regardless of where the last data value was found.

You may use the same variable names the second time through the data or not as you choose, since the values are being read as though for the first time. In order to skip unwanted values dummy variables must be read. In the following example, BASIC prints:

```
4           1           2           3
```

on the last line because it did not skip the value for the original N when it executed the loop beginning at line 200.

```
10 REM - PROGRAM TO ILLUSTRATE USE OF RESTORE
20 READ N
25 PRINT "VALUES OF X ARE:"
30 FOR I=1 TO N
40 READ X
50 PRINT X,
60 NEXT I
70 RESTORE
185 PRINT
190 PRINT "SECOND LIST OF X VALUES"
200 PRINT "FOLLOWING RESTORE STATEMENT:"
210 FOR I=1 TO N
220 READ X
230 PRINT X,
240 NEXT I
250 DATA 4,1,2
251 DATA 3,4
300 END
RUN
```

```

VALUES OF X ARE:
  1           2           3           4
SECOND LIST OF X VALUES
FOLLOWING RESTORE STATEMENT:
  4           1           2           3

READY

```

INPUT Statement

The second way to input data to a program is with an INPUT statement. This statement is used when writing a program to process data to be supplied while the program is running. The programmer types in the values as the computer asks for them. Depending upon how many values are to be brought in by the INPUT command, the programmer may wish to write himself a note reminding himself what data is to be typed in at what time. In the example program in Figure 12-3 the question is asked at execution time "INTEREST IN PERCENT?", "AMOUNT OF LOAN?", and "NUMBER OF YEARS?" The programmer knows which value is requested and proceeds to type and enter the appropriate number.

```

10 REM - PROGRAM TO COMPUTE INTEREST PAYMENTS
20 PRINT "INTEREST IN PERCENT";
25 INPUT J
26 LET J=J/100
30 PRINT "AMOUNT OF LOAN";
35 INPUT A
40 PRINT "NUMBER OF YEARS";
45 INPUT N
50 PRINT "NUMBER OF PAYMENTS PER YEAR";
55 INPUT M
60 LET N=N*M
65 LET I=J/M
70 LET B=1+I
75 LET R=A*I/(1-1/B^N)
78 PRINT
80 PRINT "AMOUNT PER PAYMENT =";R
85 PRINT "TOTAL INTEREST      =";R*N-A
88 PRINT
90 LET B=A
95 PRINT " INTEREST      APP TO PRIN      BALANCE"

```

Figure 12-3 Interest Example Program

```

100 LET L=B*I
110 LET P=R-L
120 LET B=B-P
130 PRINT L,P,B
140 IF B>=R GOTO 100
150 PRINT B*I,R-B*I
160 PRINT "LAST PAYMENT =" B*I+B
200 END
RUN

```

```

INTEREST IN PERCENT? 9
AMOUNT OF LOAN? 2500
NUMBER OF YEARS? 2
NUMBER OF PAYMENTS PER YEAR? 4

```

```

AMOUNT PER PAYMENT = 344.9617
TOTAL INTEREST      = 259.6932

```

INTEREST	APP TO PRIN	BALANCE
56.25	288.7117	2211.288
49.75399	295.2077	1916.081
43.11182	301.8498	1614.231
36.32019	308.6415	1305.589
29.37576	315.5859	990.0035
22.27508	322.6866	667.317
15.01463	329.947	337.3699
7.590824	337.3708	
LAST PAYMENT = 344.9608		

READY

Figure 12-3 Interest Example Program (continued)

The INPUT statement is of the form:

(line number) INPUT (variables separated by commas)

For example:

```
10 INPUT A,B,C
```

will cause the computer to pause during execution, print a question mark and wait for the user to type in three numerical values separated by commas and entered to the computer by hitting the RETURN key at the end of the list.

As you will notice in Figure 12-3, the question mark is grammatically useful if you care to formulate a verbal question which the

input value will answer. This will be further explained in the section on the PRINT Statement.

The output for the program begins after the word RUN and includes a verbal description of the numbers. This verbal description on the output is optional with the programmer, although it has a definite advantage in ease of use and understanding.

Only one question mark is printed per INPUT statement, so the programmer must be careful to insert the correct number of variables at that point, separating them by commas if more than one are to be typed. When the correct number of variables have been typed, hit the RETURN key to enter them to the computer.

If too few values are listed, the message:

```
MORE?
```

will appear. If too many values are typed, the message:

```
TOO MUCH INPUT, EXCESS IGNORED
```

will be given.

PRINT Statement

The PRINT statement is the output statement for BASIC. Depending upon what follows the PRINT command, we can create numerous different output formats and even plot points on a graph.

In order to skip a line on the output sheet, type only a line number and the command PRINT:

```
10 PRINT
```

When the computer comes to line 10 during the run, the paper on the console will be advanced by one line. In the example program in Figure 12-3, line 53 causes a blank line on the output sheet between the section where the user enters data to the computer and the section where the computer supplies the results of the program.

In order to have the computer print out the results of a computation, or the value of any variable at any point in the program, the user types the line number, the command PRINT, and the variable names separated by commas:

```
10 PRINT A,B,C,D,E
```

This will cause the values of A, C + B, and the square root of A to be printed in the first three of the five fixed format columns (of 14 spaces each) which BASIC uses for most output. For example the statement:

```
10 PRINT A,C+B,SQRT(A)
```

will cause the values of the variables to be printed like this:

```
12.3      12.3      12.3      12.3      12.3
```

where A, B, C, D, E equal 12.3. When more than five variables are listed in the PRINT statement and separated by commas, the sixth value begins a new line of output.

The third possibility for the PRINT statement is to print out a message, or some text. The user may ask that any message be printed by placing the message in quotation marks. For example:

```
10 PRINT "THIS IS A TEST"
```

when line 10 above is encountered during execution the following will be printed:

```
THIS IS A TEST
```

(Going back to the example program in Figure 12-3, notice the function of lines 80, 85, and 90.)

Looking at Figure 12-3 shows that the PRINT statement can combine the second and third options. One PRINT command tells the computer to print:

```
AMOUNT PER PAYMENT = 344.9617
```

The command which did this was line 80:

```
80 PRINT "AMOUNT PER PAYMENT =";R
```

It is not necessary to use the standard five column format for output. A semi-colon (;) will cause the following text or data to be printed following the last character of text or data printed. A

comma (,) will cause a jump to the next of the five output format columns. BASIC allows the user to omit format control characters (,) or (;) between text and data, and assumes a semi-colon. For example:

```
80 PRINT "AMOUNT PER PAYMENT =" R
```

will result in the same output as line 80 above.

In addition to the capabilities already mentioned, the PRINT statement can also cause a constant to be printed at the console. For example:

```
10 PRINT 1.234, SQR(100/4)
```

will cause the following to be printed at execution time:

```
1.234          5
```

Any number present in a PRINT statement will be printed exactly as shown. Any algebraic expression in a PRINT statement will be evaluated with the current value of the variables and the result printed.

In Figure 12-3, line 160 reads:

```
160 PRINT "LAST PAYMENT =" B*I+B
```

and caused the following to be printed upon execution:

```
LAST PAYMENT = 344.9608
```

This demonstrates the omission of the format control character as well as the ability of the PRINT statement to print text and do calculations.

The following example program illustrates the use of the control characters in PRINT statements:

```

10 READ A,B,C
20 PRINT A,B,C,A*2,B*2,C*2
30 PRINT
40 PRINT A;B;C;A*2;B*2;C*2
50 DATA 4,5,6
60 END
RUN

```

```

4           5           6           16           25
36

4 5 6 16 25 36

```

READY

If a number should happen to be too long to be printed on the end of a single line, BASIC automatically moves the number entirely to the beginning of the next line.

TAB Function

When using the PRINT statement thus far we have had to print a blank character wherever we wanted blank space; there was no real control over printing. The TAB function is a more sophisticated technique allowing the user to position the printing of characters anywhere on the Teletype paper line. This line is 72 characters long, and the print positions can be thought of as being numbered from 0 to 71, going from left to right. The TAB function argument can be positive or negative: TAB(-1) causes a tab over to position 71, TAB(3) causes a tab to position 3. (The TAB function can be thought of as operating modulo 72.)

After performing TAB(n), the next character to be printed will be placed in position n. If n is a position to the left of the current position, a carriage return (without a line feed) is used to correctly position the printing head.

For example:

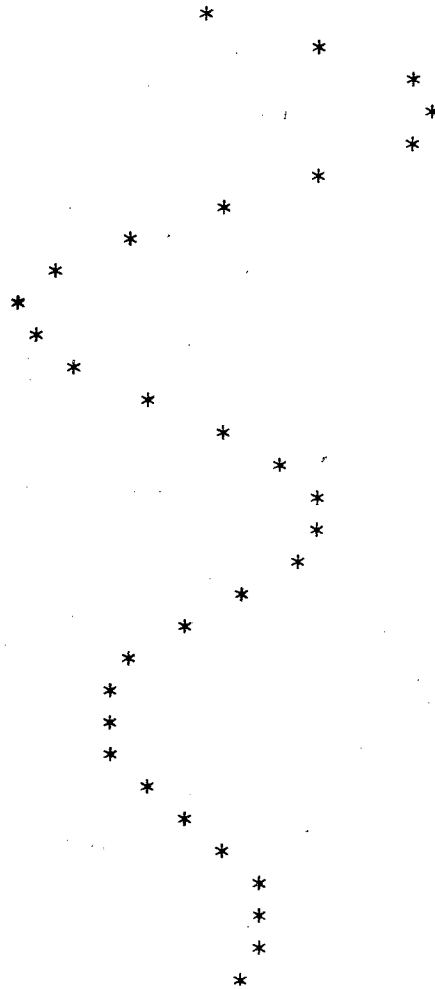
```
10 PRINT "X =";TAB(2);"/";3.14159
```

will print the slash on top of the equal sign, as shown below:

```
X ≠ 3.14159
```


The following is an example of the sort of graph that can be drawn with BASIC using the TAB function:

```
30 FOR X=0 TO 15 STEP .5
40 PRINT TAB(30+15*SIN(X)*EXP(-.1*X));"*"
50 NEXT X
60 END
RUN
```



READY

SUBSCRIPTS AND LOOPS

Subscripted Variables

In addition to simple variable names, there is a second class of variables which BASIC accepts called subscripted variables. Subscripted variables provide the programmer with additional computing capabilities for dealing with lists, tables, matrices, or any set of related variables. In BASIC variables are allowed one or two subscripts. A single letter forms the name of the variable followed by

one or two integers in parentheses, separated by commas, indicating the place of that variable in the list. You can have up to 26 arrays in any program (corresponding to the letters of the alphabet), subject only to the amount of core space available for data storage. For example, a list might be described as A(I) where I goes from 1 to 5 as shown below:

A(1), A(2), A(3), A(4), A(5)

This allows the programmer to reference each of the five elements in the list A. A two dimensional matrix A(I, J) can be defined in a similar manner, but the subscripted variable A can only be used once. A(I) and A(I, J) cannot be used in the same program.

It is possible, however, to use the same variable name as both a subscripted and as an unsubscripted variable. Both A and A(I) are valid variable names and can be used in the same program.

Input can be done easily using subscripted variables, as follows:

```
10 REM - PROGRAM DEMONSTRATING READING OF
11 REM - SUBSCRIPTED VARIABLES
15 DIM A(5), B(2,3)
18 PRINT "A(I) WHERE A=1 TO 5:"
20 FOR I=1 TO 5
25 READ A(I)
30 PRINT A(I);
35 NEXT I
38 PRINT
39 PRINT
40 PRINT "B(I,J) WHERE I=1 TO 2"
41 PRINT "      AND J=1 TO 3:"
42 FOR I=1 TO 2
43 PRINT
44 FOR J=1 TO 3
48 READ B(I,J)
50 PRINT B(I,J);
55 NEXT J
56 NEXT I
60 DATA 1,2,3,4,5,6,7,8
61 DATA 8,7,6,5,4,3,2,1
65 END
RUN
A(I) WHERE A=1 TO 5:
 1  2  3  4  5
B(I,J) WHERE I=1 TO 2
      AND J=1 TO 3:

 6  7  8
 8  7  6

READY
```

DIM STATEMENT

As in the preceding examples, we see that the use of subscripts requires a dimension (DIM) statement to define the maximum number of elements in the array. The DIM statement is of the form:

(line number) DIM $v_1(n_1), v_2(n_2, m_2)$

where v_n indicates an array variable name and n and m are integer numbers indicating the largest subscript value required during the program. For example:

```
10 DIM A(6,10)
```

The first element of every array is automatically assumed to have a subscript of zero. Dimensioning A(6, 10) sets up room for an array with 7 rows and 11 columns. This matrix can be thought of as existing in the following form:

$A_{0,0}$	$A_{0,1}$.	.	.	$A_{0,10}$
$A_{1,0}$	$A_{1,1}$.	.	.	$A_{1,10}$
$A_{2,0}$	$A_{2,1}$.	.	.	$A_{2,10}$
.
.
.
$A_{6,0}$	$A_{6,1}$.	.	.	$A_{6,10}$

as shown in the program below:

```
10 REM - MATRIX CHECK PROGRAM
15 DIM A(6,10)
20 FOR I=0 TO 6
22 LET A(I,0) = I
25 FOR J=0 TO 10
28 LET A(0,J) = J
30 PRINT A(I,J)
35 NEXT J
40 PRINT
45 NEXT I
50 END
RUN
```

0	1	2	3	4	5	6	7	8	9	10
1	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0

READY

Notice that a variable has a value of zero until it is assigned a value.

If the user wishes to conserve core space, and not make use of the extra variables set up within the array, he should, for example, say DIM A(5, 9) which would result in a 6 by 10 array which would be referenced beginning with the A(0, 0) element.

You can define more than one array in a single DIM statement:

```
10 DIM A(20), B(4,7)
```

will dimension both the list A and the matrix B.

A number must be used to define the maximum size of the array. A variable inside the parentheses is not acceptable and would result in an error message by BASIC at run time. The amount of user core not filled by the program will determine the amount of data the computer can accept as input to the program at any one time. In some programs a PROGRAM TOO LARGE message may occur, indicating that core will not hold an array of the size requested. In that event, the user should change his program to process part of the data in one run and the rest later.

Loops

So far in this chapter we have seen FOR and NEXT statements used several times in examples. These two statements define the beginning and end of a loop. A loop being a set of instructions which modifies itself and repeats until some terminal condition is reached.

FOR STATEMENT

The FOR statement is of the form:

(line number) FOR (variable) = (formula) TO (formula)
STEP (formula)

For example:

```
10 FOR K=2 TO 20 STEP 2
```

which will iterate (cycle) through the designated loop using K as 2, 4, 6, 8, . . . , 20 in calculations involving K. When the value 20 is reached the loop is left behind and the program goes to the line following the NEXT statement (described below).

The variable mentioned in the definition must be unsubscripted, although a common use of such loops is to deal with subscripted variables using the FOR variable as the subscript of a previously defined variable. The formulas mentioned in the definition can be real or integer numbers, variables, or expressions.

NEXT STATEMENT

The NEXT statement signals the end of the loop and at that point the computer adds the STEP value to the variable and checks to see if the variable is still less than the terminal value. When the variable exceeds the terminal value control falls through the loop to the following statement.

When control falls through the loop the variable value is one step greater than it was when the loop was last executed. For some programs this information may be useful.

If the STEP value is omitted, +1 is assumed. Since +1 is the usual STEP value, that portion of the statement is frequently omitted.

In the following example we see a demonstration of the last two paragraphs. The loop is executed 10 times, the value of I is 11 when control leaves the loop and +1 is the assumed STEP value.

```
10 FOR I=1 TO 10
20 NEXT I
30 PRINT I
40 END
RUN
```

11

READY

If line 10 had been:

```
10 FOR I=10 TO 1 STEP -1
```

the value printed by the computer would be 0.

The numbers used in the FOR statement can be "formulas" as indicated earlier. A formula in this case can be a variable, a mathematical expression, or a numerical value.

The value of each formula is evaluated upon first encountering

the loop. While the values of the variables, if any, used in evaluating these formulas can be changed within the loop, the values assigned in the FOR statement remain as they were initially defined.

In the last example program the value of I (in line 10) can be successfully changed in the program. The loop:

```
10 FOR I=1 TO 10
15 LET I=10
20 NEXT I
```

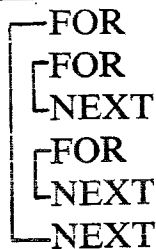
will only be executed once since the value 10 has been reached by the variable I and the termination condition is satisfied.

NESTING LOOPS

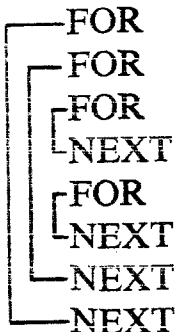
It is often useful to have one or more loops within a loop. This technique is called nesting. Nesting is allowed as long as the field of one loop (the numbered lines from the FOR statement to the corresponding NEXT statement, inclusive) does not cross the field of another loop. A diagram is the best way to illustrate acceptable nesting procedures:

ACCEPTABLE NESTING TECHNIQUES	UNACCEPTABLE NESTING TECHNIQUES
----------------------------------	------------------------------------

Two Level Nesting



Three Level Nesting



A maximum of eight (8) levels of nesting is permitted. Exceeding that limit will result in an ILLEGAL FOR NESTING error message.

If the value of the counter variable is originally set equal to the terminal value, the loop will execute once, regardless of the STEP value. If the starting value is beyond the terminal value, the loop will not execute.

It is also possible to exit from a FOR-NEXT loop without the counter variable reaching the termination value. A conditional transfer may be used to leave a loop. Control may only transfer *into* a loop which had been left earlier without being completed, ensuring that the termination and STEP values are assigned.

TRANSFER OF CONTROL

Certain statements can cause the execution of a program to jump to a different line either unconditionally or depending upon some condition within the program. Looping is one method of jumping to a designated point until a condition is met. The following commands give the programmer additional capabilities in this area.

Unconditional Transfer

The GOTO statement is an unconditional command telling the computer to either jump ahead or back in the program. For example:

```
100 GOTO 50
```

or

```
24 GOTO 78
```

The GOTO statement is of the form:

(line number) GOTO (line number)

When the logic of the program reaches the GOTO statement, the statement(s) immediately following it will not be executed, but the statements beginning with the line number indicated are performed.

The program on the following page never ends; it does a READ, prints something and attempts to do this over and over until it runs out of data, which is sometimes an acceptable, though not advisable, way to end a program:

```

10 REM - PROGRAM ENDING WITH ERROR
11 REM - MESSAGE WHEN OUT OF DATA
20 READ X
25 PRINT "X="X,"X+2="X+2
30 GOTO 20
35 DATA 1,5,10,15,20,25
40 END
RUN

```

```

X= 1          X+2= 1
X= 5          X+2= 25
X= 10         X+2= 100
X= 15         X+2= 225
X= 20         X+2= 400
X= 25         X+2= 625

```

OUT OF DATA IN LINE 20

READY

Conditional Transfer

If a program requires that two values be compared at some point, logic may direct us to different procedures depending on the comparison. In computing we logically test values to see whether they are equal, greater, or less than another value, or a possible combination of the three.

In order to compare values we use a group of mathematical symbols not discussed earlier. These symbols are as follows:

<u>BASIC</u> <u>Symbol</u>	<u>Math</u> <u>Symbol</u>	<u>BASIC</u> <u>Example</u>	<u>Meaning</u>
=	=	A = B	A is equal to B
<	<	A < B	A is less than B
<=	≤	A <= B	A is less than or equal to B
>	>	A > B	A is greater than B
>=	≥	A >= B	A is greater than or equal to B
<>	≠	A <> B	A is not equal to B

IF-THEN AND IF-GOTO

The IF-THEN and IF-GOTO statements both allow the pro-

grammer to test the relationship between two formulas (variables, numbers, or expressions). Providing the relationship we have described in the IF statement is true at that point, control will transfer to the line number indicated. The statements are of the form:

(line number) IF (formula) relation (formula)

{ THEN }
{ GOTO } (line number)

The use of the word THEN or GOTO is the programmer's choice. For example:

```
10 IF A=5 GOTO 70
```

causes transfer from line 10 to line 70 if A is equal to 5. If A is not equal to 5, control passes to the next line of the program following line 10.

SUBROUTINES

When particular mathematical expressions are evaluated several times throughout a program, the DEF statement enables the user to write that expression only once. The technique of looping allows the program to do a sequence of instructions a specified number of times. If the program should require that a sequence of instructions be executed several times in the course of the program, this too is possible. A subroutine is a section of code performing some operation that is required at more than one point in the program. Sometimes a complicated I/O operation for a volume of data, a mathematical evaluation which is too complex for a user defined function, or any number of other processes may be best performed in a subroutine.

GOSUB Statement

Subroutines are placed physically at the end of a program, usually before DATA statements, if any, and always before the END statement. The program begins execution and continues until it encounters a GOSUB statement of the form:

(line number) GOSUB (line number)

where the number after GOSUB is the first line number of the subroutine. Control then transfers to that line in the subroutine. For example:

```
50 GOSUB 200
```

RETURN Statement

Having reached line 50, as shown on the previous page, control transfers to line 200; the subroutine is processed until the computer encounters a RETURN statement of the form:

(line number) RETURN

which causes control to return to the line following the GOSUB statement. Before transferring to the subroutine, BASIC internally records the next line number to be processed after the GOSUB statement; the RETURN statement is a signal to transfer control to this line. In this way, no matter how many subroutines or how many times they are called, BASIC always knows where to go next. The following program demonstrates a simple subroutine:

```
1 REM - THIS PROGRAM ILLUSTRATES GOSUB AND RETURN
10 DEF FNA(X)= ABS(INT(X))
20 INPUT A,B,C
30 GOSUB 100
40 LET A=FNA(A)
50 LET B=FNA(B)
60 LET C=FNA(C)
70 PRINT
80 COSUB 100
90 STOP
100 REM - THIS SUBROUTINE PRINTS OUT THE SOLUTIONS
110 REM - OF THE EQUATION:  $AX^2 + BX + C = 0$ 
120 PRINT "THE EQUATION IS " A "X^2 + " B "X + " C
130 LET D= B*B - 4*A*C
140 IF D<>0 THEN 170
150 PRINT "ONLY ONE SOLUTION... X =" -B/(2*A)
160 RETURN
170 IF D<0 THEN 200
180 PRINT "TWO SOLUTIONS... X =";
185 PRINT (-B+SQR(D))/(2*A) "AND X =" (-B-SQR(D))/(2*A)
190 RETURN
200 PRINT "IMAGINARY SOLUTIONS... X = (";
205 PRINT -B/(2*A) "," SQR(-D)/(2*A) ") AND (";
207 PRINT -B/(2*A) "," -SQR(-D)/(2*A) ")"
210 RETURN
900 END
RUN
```

```
? 1,.5,-.5
THE EQUATION IS 1 *X^2 + .5 *X + -.5
TWO SOLUTIONS... X = .5 AND X =-1
```

```
THE EQUATION IS 1 *X^2 + 0 *X + 1
IMAGINARY SOLUTIONS... X = ( 0 , 1 ) AND ( 0 , -1 )
```

READY

Lines 100 through 210 constitute the subroutine. The subroutine is executed from line 30 and again from line 80. When control returns to line 90 the program encounters the STOP statement and terminates execution. Note that even though the program logically ends with a STOP, the END command must still be present.

For another detailed example of a subroutine, see Figure 12-2.

STOP and END Statements

The STOP statement is used synonymously with the END statement to terminate execution, but the END statement must be the last statement of the entire program. STOP may occur several times throughout the program. No BASIC program will run without an END statement of the form:

(line number) END

The format of the STOP statement is simply:

(line number) STOP

STOP is equivalent to a GOTO nn, where nn is the line number of the END statement.

Nesting Subroutines

More than one subroutine can be used in a single program in which case they can be placed one after another at the end of the program (in line number sequence). A useful practice is to assign distinctive line numbers to subroutines, for example if you have numbered the main program with line numbers up to 199, you could use 200 and 300 as the first numbers of two subroutines.

Subroutines can also be nested, in terms of one subroutine calling another subroutine. If the execution of a subroutine encounters a RETURN statement it will return control to the line following the GOSUB which called that subroutine; therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and have more than one RETURN statement where certain conditions will cause control to reach any one RETURN statement. It is possible to transfer to the beginning or any part of a subroutine; multiple entry points and RETURNS make a subroutine more versatile.

The maximum level of GOSUB nesting is about forty (40)

levels, which should prove more than adequate for all normal uses. Exceeding this limit will result in the message:

```
GOSUB--RETURN ERROR
```

ERRORS AND HOW TO MAKE CORRECTIONS

Single Letter Corrections

Nobody being perfect, we all make typing errors if not logical errors. The first is by far the easier to correct. If you notice an error immediately as you type it, for example:

```
10 LEB
```

instead of LET as you meant to begin the line, hit the RUBOUT key or SHIFT/O (back arrow) once for every character you wish to remove, including spaces. This will result in the printing (by BASIC) of a back arrow to show that the rubout has been accomplished. Make the correction and continue typing as shown below.

```
10 LEB-T A=10*B
```

if that was the intended line. The computer does not even see the mistake; it is erased, except on the console as you typed it. The typed line enters the computer only when you hit the RETURN key. Before that time you can correct errors with the RUBOUT key or SHIFT/O. If you desire a neat, corrected listing at the end of your work, that is possible too. More on that later.

```
20 DEN F---F FNA(X,Y)=X+2+3*Y
```

is the same as:

```
20 DEF FNA(X,Y)=X+2+3*Y
```

to the computer. Notice you erase spaces, as well as printing characters.

Erasing a Line

If at any time you have typed a line and not yet hit the RE-

TURN key, the line can be erased by striking the ALTMODE (ESCAPE on some machines) key. BASIC will echo back:

```
$ DELETED
```

at the end of the line to indicate that the line has been removed. You can continue typing on that line as though it were the start of a new line or hit the RETURN key to start a fresh line.

Once you have hit the RETURN key and have entered a line into the computer it can still be corrected by simply typing the line number and proceeding to retype the line correctly. The old line is automatically deleted as you type the line again, even if it was longer than the new line.

You can delete an entire line by typing the line number and hitting the RETURN key. This removes the entire line and line number from your program.

NOTE

Typing a line number followed by back arrows does not remove that number from the line it identifies. If you accidentally type the line number of a previous line you do not want erased, the RUBOUT key will remove the unwanted line number, leaving the original line intact. For example:

```
10 LET A=4  
10←←20 LET B=A+7
```

will leave line 10 as it is and allow you to type line 20.

Following an attempt to run a program you may receive an error message. Most errors can be corrected by typing the line number, typing the line over again with the correction, and hitting RETURN. The program is then ready to be run again. You can make as many changes or corrections between runs as you wish. (For a more advanced technique in program editing, see the section on Advanced BASIC.)

Erasing a Program in Core

Assuming you have written a program on-line in BASIC, have completed it and now wish to run another program in BASIC, but

do not wish to save the old program, when BASIC types READY, answer:

SCRATCH
or SCR

The SCRATCH command will erase the old program and leave a fresh, blank area in which you can work. Only the abbreviation SCR is necessary. BASIC will again reply READY, and you proceed from this point. The previous program name is maintained for the cleared area. You can, alternatively, reply to READY with NEW, if you wish to create a new program, or OLD, if you wish to recall a saved program for further work. SCRATCH is much faster than NEW or OLD in clearing core.

If, after BASIC types READY, you merely begin typing a new program without clearing core, BASIC will retain the name of the previous program and in effect you will write over that program as though you were changing each single line. However, if you do not remove or type over *all* of the previous line numbers you will discover the unchanged lines appearing in the new program as well. To avoid this, telling BASIC to SCRATCH the old program and create a new program gives you a blank area on which to write.

Stopping a Run

If your program begins to print what you know will be a long list of unwanted output for one reason or another, you can stop a running program by depressing the CTRL (control) key and hitting the C key. CTRL/C will cause ↑C to be printed on the console paper, and will stop execution, returning you to edit mode (BASIC prints READY). You can make changes, save the program, or whatever you wish.

NOTE

The up arrow (↑) in the command ↑BS or ↑C is not to be confused with the up arrow used to express exponentiation. The ↑ indicated on the console keyboard is for raising a number to a power. The ↑C, for example, is a short way of writing CTRL/C where the CTRL key is depressed while the C is struck.

RUNNING A BASIC PROGRAM

LOGIN Procedure

Before you attempt to use TSS/8, someone in charge of the computer will issue you an account number and a password. When you sit down at the Teletype console turn the LINE-OFF-LOCAL knob to LINE. You should see a period on the left margin of the paper, if not, hit the RETURN key and one will appear.

In answer to the dot type:

LOGIN account number password

Enter the three terms with a single space between them and strike the RETURN key. For example:

```
.LOGIN 175 DEMO
```

None of the characters in the line you typed will be printed at the console, in order to preserve the secrecy of these codes. When you successfully log onto the system some opening message will likely be printed ending with another dot. In reply type:

```
R BASIC
```

Initial Dialogue

This puts you in communication with BASIC which will then type out:

```
NEW OR OLD--
```

If you are entering a new program you reply NEW, if calling in an old program you have saved in a file, reply OLD. To enter the command to BASIC, you must strike the RETURN key. BASIC will then reply:

```
NEW PROGRAM NAME--
```

```
OR OLD PROGRAM NAME--
```

as the case may be. You will type in any six-or-less character identifier as your program name. An old program's name must be typed correctly, so it is a good idea to choose an appropriate, easily remembered name. An example of how to call a program which

you had previously saved would look as follows:

```
OLD PROGRAM NAME--PRIME
```

where PRIME is the name of the program.

Programs (called files when they have been saved) may be loaded from another user's account, file protect permitting.² When BASIC asks for the old program name, you may reply:

```
OLD PROGRAM NAME--PRIME 120
```

where PRIME is the name of a program and 120 is the file account number under which PRIME is stored.

If a file exists for use of a large number of programmers it will likely be placed in the System Program Library and may be called by typing the name of the program immediately followed by an asterisk:

```
OLD PROGRAM NAME--PRIME*
```

will call PRIME from the System Program Library.

Following the program name supplied by the user, BASIC then types READY.

At this point you may begin to type in a new program hitting RETURN after each line, or change or run an old program in accordance with the conventions already established.

RUN Command

When your program is ready to be run (be sure there is an END statement), type RUN, press the RETURN key, and the program will attempt to execute. If there is some error in the way you wrote your BASIC code, an error message will be printed, following which you may correct the errors one line at a time. Then type the RUN command again. If the program executes correctly you will obtain whatever printed output you requested. When the END

² When you SAVE a file on disk the protection code of that program allows anyone knowing the account number to access the program. For additional information on file protection codes, see the *TSS/8 User's Guide* (DEC-T8-MRFB-D).

statement is reached, BASIC stops execution and again types READY.

Editing Phase

To simplify matters, we can think of BASIC as having two phases, a run phase and an editing phase. The run phase is the time between when you type RUN and when BASIC types READY; this is the time during which BASIC is compiling and executing your program. Once BASIC has printed READY, it is able to accept commands directly from your Teletype; during this editing phase you can prepare your program and can direct BASIC to perform a variety of services such as the SCRATCH command. (You can force an entry to the editing phase with a CTRL/C.) The commands used in the editing phase can all be abbreviated to three letters, some have arguments, others do not, as explained below.

SAVE COMMAND

When you have completed working on a program, you may save it on disk to call again in the future. To do this type:

SAVE
or SAV

This would use the same name you typed in response to the question NEW PROGRAM NAME--(If you think you might forget it, write the name in a REMARK statement at the beginning of the program.)

It is also possible to say:

SAVE name
or SAV name

where name is not the original name you gave as a reply to NEW PROGRAM NAME--, however, the name you tell BASIC to save is the name you must give to retrieve the program in response to a later query of OLD PROGRAM NAME--

NOTE

Spaces do have significance in program names (i.e., SAVE TIP TOP will be saved as TIP). In general, then, spaces are delimiters for all editing phase commands.

REPLACE COMMAND

If you have called an old program and made some changes in it, you can then return the corrected program to the disk under its old name using the REPLACE command. This command deletes the old program of that name as it enters the new one.

In response to READY, type:

REPLACE
or REP

or, alternatively:

REPLACE name
or REP name

which causes the program presently being worked on to replace the old copy of the same program on the disk. If a program name is indicated, that name is used as the file name.

UNSAVE COMMAND

If you wish to delete a program from your disk storage area, type:

UNSAVE name
or UNS name

The program with the name specified will be deleted from your permanent file. This is done when you no longer plan to use that program. In general, programs which are not going to be run frequently are best stored on paper tape, reserving disk storage for more active programs. It is possible to delete several files with a single UNSAVE command separating the program names with commas.

LIST COMMAND

Once your program works you may discover you have several feet of Teletype paper filled with corrections and other gibberish. To obtain a clean listing of your program, type LIST or LIS followed by the RETURN key. The whole program will be printed. You can then tell the computer to RUN and your output will follow.

For debugging purposes it is sometimes useful to list part of your program. LIST or LIS followed by one line number or two line

numbers separated by a comma will result in BASIC printing either that single line or the lines between and including the two numbers given.

DELETE COMMAND

DELETE or DEL followed by two line numbers separated by a comma will cause all lines between and including the two given to be deleted from the program. If only one line number is given, that line will be deleted. For example:

```
DEL 10,20
```

causes all lines between 10 and 20 inclusive to be deleted.

NEW AND OLD COMMANDS

If you have completed working with one program and have saved that program for future use, you may wish to work on another BASIC program or leave the terminal. If you wish to call an old program, type OLD. To indicate that you wish to begin a new program, type NEW. In either case BASIC will request a program name and, following your reply, type READY. These commands may be used at any time, not only in direct response to the question BASIC asks of NEW or OLD PROGRAM NAME--.

CATALOG COMMAND

If you type CAT or CATALOG followed by the RETURN key, a listing of all program names in your disk file will be printed by BASIC. For example:

```
CATALOG
*****
FOOTBL.BAC
PRIME .BAS
FTBALL.BAS
FOO .BAS
PRINO .BAC
```

```
READY
```

The program names have appended to them the terms .BAC and .BAS which are explained in the section on Advanced BASIC.

BYE COMMAND

When you are ready to leave the Teletype, type BYE and hit RETURN, this will return control to TSS/8 Monitor which prints a dot at the left margin. Then type LOGOUT and hit RETURN. Wait until the computer has finished its concluding message before turning the LINE-OFF-LOCAL knob to OFF.

ALTMODE KEY

Striking the ALTMODE key (which is non-printing and non-spacing) will cause any of the preceding commands (DELETE, LIST, SAVE, etc.) to be erased. ALTMODE must be struck before the RETURN key which enters the command into the computer. If you do change your mind about a command, you can alter it as shown below:

```
SAVE FOOS DELETED
```

BASIC replies \$ DELETED to show that the command has been erased, you may then retype the line.

Punching a Paper Tape

It may be useful in many cases to have a copy of a program you have written in BASIC stored on paper tape. You can create such a copy quite easily. Once you have completed your program to the point that you wish to copy it, punch a listing of it through BASIC. The steps involved are:

1. Type TAPE followed by hitting RETURN. Any characters you type now will not echo on the console or on your tape.
2. Punch the ON button on the tape punch.
3. Type LIST followed by the RETURN key. This causes the program to be listed on paper tape and on the console.
4. Punch the OFF button on the tape punch.

Using LIST when in TAPE mode will result in the following:

1. The word LIST will not echo. No leading spaces are printed before line numbers as in a normal LIST.
2. Blank tape is "printed" before and after the program.

You will notice that when you tear off the tape from the punch

there will be an arrow head on the tape, this shows the direction in which the tape is later to be inserted into the machine.

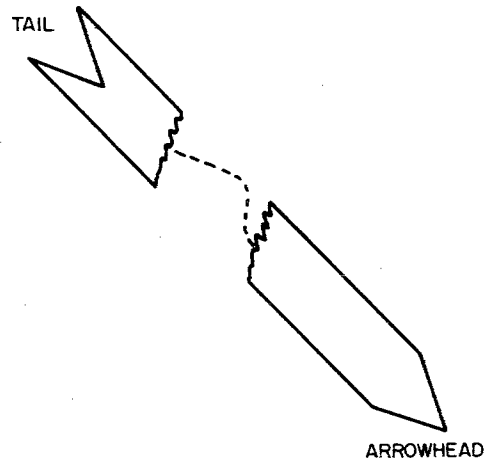


Figure 12-4 Paper Tape Diagram

Once you have finished punching your program you will wish to return to regular operating mode on the computer. During TAPE mode no characters you type will be echoed, RUBOUTs are ignored, as is blank tape. Typing KEY followed by the RETURN key will bring you back to normal operating mode. You may then continue working on that program, call another program, or log out.

A paper tape can be duplicated or copies made by positioning the tape in the reader depress the ON button, turn the LINE-OFF-LOCAL knob to LOCAL, and turn the reader control switch to START. Tape will be reproduced as it is.

Reading and Listing a Paper Tape

To read in a paper tape from the low speed reader on the Teletype, first create a new program name in BASIC and proceed as follows:

1. Position paper tape in the reader head:
 - a. Raise retainer cover,
 - b. Set reader control to FREE,
 - c. Position paper tape with feed holes over the sprocket wheel and the arrow (cut) pointing outward from the console.
2. Type TAPE, hit the RETURN key.

3. Set reader control switch to **START** until listing has been completed. Reader will not stop at blank tape. You must turn the reader control switch to **FREE**.
4. In order to get back into regular operating mode where the characters you type will be echoed at the console, type **KEY** and hit the **RETURN** key.
5. **BASIC** will type **READY**, you can then ask **BASIC** to **LIST** and **RUN** your program.

Transferring a File to Paper Tape or DECTape from Disk

It is not in the scope of this manual to describe the transfer of **BASIC** programs from disk to paper tape or DECTape. If you wish to use these facilities, refer to writeups on **PIP** (Peripheral Interchange Processor) and **COPY** (both found in the *TSS/8 User's Guide*).

IMPLEMENTATION NOTES

The **TSS/8 BASIC** language is compatible with Dartmouth **BASIC** except as noted below:

1. There are no matrix operations.
2. There are no character string instructions.
3. The **ON** statement has not been implemented.
4. **BASIC** has no features which allow reading or writing data files on the disk. (Although programs may be saved on the disk for future use.)
5. All array (subscripted) variables must appear in a **DIM** statement.
6. User defined functions in **DEF** statements need not begin with the letters **FN** as in Dartmouth **BASIC**.
7. User defined functions are restricted to one line.
8. Maximum size of a **BASIC** program can be said to be roughly 350 lines. The exact size of a program that a user can run depends upon several factors: the number and size of arrays, number of nested loops and subroutines, number of variables, and user defined functions. A program using an unusually large number of any of these factors will, of course, have less room in which to run.

ADVANCED BASIC

This section deals with additional features of **BASIC** which,

once you have learned the BASIC language, will make programming somewhat easier.

EDIT Command

Frequently it is only necessary to correct several characters in a line. Rather than retype the entire line, which may be a complex formula or output format, there is a command which allows you to access a single line and search for the character you wish to change. The form of the EDIT command is as follows:

EDI line number
[character]

Notice that the EDIT command may be abbreviated to three letters. It is then followed by the line number of the statement to be changed. Enter the command by striking the RETURN key. At this point BASIC types [and waits for you to type a search character after which BASIC types]. The character you give will be some character which already exists on the line (one of the legal BASIC characters, ANSCI 240 through 336 inclusive on the ANSCII table in Appendix B2). After the search character is typed, BASIC prints out the contents of that line until the search character is printed. At this point printing stops, and the user has the following options:

1. Type in new characters which are inserted following the ones already printed.
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 BELL (CTRL/G); this allows the user to change the search character. BASIC types back another [and the user can specify a new search character.
4. Use the RUBOUT (or SHIFT/O) key to delete one character to the left each time RUBOUT is depressed. RUBOUT echoes as ←.
5. Type the RETURN key to terminate editing of the line at that point, removing any text to the right.
6. Type the ALTMODE key to delete all the characters to the left except the line number.
7. Type the LINE FEED key to terminate editing of the line, saving the remaining characters.

On completion of the EDIT operation, BASIC types READY. Note that line numbers cannot be changed using EDIT, i.e., you cannot search for a line number digit. Any illegal characters will be ignored.

The following example demonstrates the EDIT command where the incorrect line reads as follows:

```
60 PRINT "PI=3.14146 ABOU*!"
```

To edit the line would result in the following output on the Teletype:

```
EDIT 60
[6]60 PRINT "PI=3.14146--59[*] ABOU*-T
READY
LIST 60
60 PRINT "PI=3.14159 ABOUT!"
```

The operations involved in editing the line were as follows: First the number 6 was indicated as the search character. BASIC ignores the line number, but will print it. When the 6 was printed, RUBOUT was struck twice to remove the two incorrect digits and 59 inserted in their place. CTRL/BELL is struck resulting in BASIC accepting another search character. BASIC then prints to the search character * which is removed with a RUBOUT and replaced with T. A LINE FEED is struck to terminate the edit and save the remaining characters.

COMPILE Command

When a program is debugged and working to your satisfaction, it is faster to be able to directly RUN a program without waiting for BASIC to recompile it each time. To enable you to store a compiled program the COMPILE command has been added to BASIC. The form of the command is as follows:

```
COMPILE name
or COM name
```

The program in core will be compiled and saved in the specified

file. **COMPILE** will not overwrite an existing file (it is like **SAVE** in this respect); if the name is in use the error message:

```
DUPLICATE FILE NAME
```

will be printed, and the program will not be compiled.

The compiled program may then be loaded and run in the usual manner. For example:

```
NEW OR OLD--OLD  
OLD PROGRAM NAME--FTBALL*
```

```
READY  
COMPILE FOOTBL
```

```
READY  
OLD  
OLD PROGRAM NAME--FOOTBL
```

```
READY  
RUN
```

In the example above, the programmer told **BASIC** to load a System Library Program file named **FTBALL** into core (the * after **FTBALL** indicates the System Library files). The programmer told **BASIC** to compile the program now in core and store the compiled program in his personal file with the name **FOOTBL**. Once **BASIC** has done this it replies **READY**. The programmer indicates that he wishes to call an old program into core, this old program is the already compiled version of the original program which can be made to execute by giving **BASIC** the **RUN** command.

Compiled **BASIC** files may not be listed or changed in any way; therefore a program should not be saved as a compiled file until it has been completely debugged. If you attempt to list or change a compiled file the error message:

```
EXECUTE ONLY FILE
```

will be printed.

FILE EXTENSIONS

In order for the user to easily tell the difference between com-

piled, uncompiled, and temporary files within your storage area on disk, the following conventions are followed and will help you tell the difference when you run the CATALOG command.

1. SAVE and REPLACE commands will always write out a file with the extension .BAS appended to the file name given by the user.
2. COMPILE will always write out a file with the extension .BAC to the file name.

CATALOG will only list those files which have a .BAS or a .BAC extension. The two BASIC temporary files assigned to each user will be given the extension .TMP.

SUMMARY OF BASIC STATEMENTS

Command	Example of Form	Explanation
LET	LET v=f	Assign the value of the formula f to the variable v.
READ	READ v1, v2, ..., vn	Variables v1 through vn are assigned the value of the corresponding numbers in the DATA string.
DATA	DATA n1, n2, ..., nn	Numbers n1 through nn are to be associated with corresponding variables in a READ statement.
PRINT	PRINT a1, a2, ..., an	Print out the values of the specified arguments, which may be variables, text, or format control characters (, or ;).
GOTO	GOTO n	Transfer control to line n and continue execution from there.
IF-THEN	IF f1 r f2 THEN n	If the relationship r between the formulas f1 and f2 is true then transfer control to line n; if not, continue in regular sequence.
IF-GOTO	IF f1 r f2 GOTO n	Same as IF-THEN
FOR-TO	FOR v=f1 TO f2 STEP f3	Used to implement loops: The variable v is set equal to the formula f1. From this point the loop cycle is completed following which v is incremented after each cycle by f3 until its value is greater than or equal to f2. If STEP f3 is omitted, f3 is assumed to be +1.

SUMMARY OF BASIC STATEMENTS (CONT.)

Command	Example of Form	Explanation
NEXT	NEXT v	Used to tell the computer to return to the FOR statement and execute the loop again until v is greater than or equal to f2.
DIM	DIM v(s) DIM v(s1, s2)	Enables the user to create a table or array with the specified number of elements where v is the variable name and s is the maximum subscript value. Any number of arrays can be dimensioned in a single DIM statement.
GOSUB	GOSUB n	Allows the user to enter a subroutine at several points in the program. Control transfers to line n.
RETURN	RETURN	Must be at the end of each subroutine to enable control to be transferred to the statement following the last GOSUB.
RANDOMIZE	RANDOMIZE RANDOM	Enables the user to obtain an un-reproducible random number sequence in a program using the RND function.
INPUT	INPUT v1, v2, ..., vn	Causes typeout of a ? to the user waits for the user to supply the values of the variables v1 through vn.
REM	REM	When typed as the first three letters of a line allows typing of remarks within the program.
RESTORE	RESTORE	Sets pointer back to the beginning of the string of DATA values.
DEF	DEF FNB (x)= f(x) DEF FNB(x, y)= f(x, y)	The user may define his own functions to be called within his program by putting a DEF statement at the beginning of a program. The function name begins with FN and must have three letters. The function is then equated to a formula f(x) which must be only one line long. Multiple variable function definitions are allowed.

SUMMARY OF BASIC STATEMENTS (CONT.)

Command	Example of Form	Explanation
STOP	STOP	Equivalent to transferring control to the END statement.
END	END	Last statement in every program, signals completion of the program.

Functions

In addition to the usual arithmetic operations of addition (+), subtraction (-), multiplication (*), division (/), and exponentiation (\uparrow); BASIC provides the following function capabilities:

SIN(X)	Sine of X
COS(X)	Cosine of X
TAN(X)	Tangent of X
ATN(X)	Arctangent of X
EXP(X)	e^X ($e=2.712818$)
LOG(X)	Log of X (natural logarithm)
ABS(X)	Absolute value of X ($ X $)
SQR(X)	Square root of X (\sqrt{X})
INT(X)	Greatest integer in X
RND(X)	Random number between 0 and 1 is a repeatable sequence value of X ignored.
SGN(X)	Assign value of +1 if X is positive, 0 if 0, or -1 if negative.
TAB(X)	Controls the position of the printing head on the Teletype.

NOTE: Trig functions use radians.

SUMMARY OF BASIC EDIT AND CONTROL COMMANDS

Several commands for editing BASIC programs and for controlling their execution enable you to: delete lines, list your program, save programs on disk, delete or replace old programs on disk with new programs, call in programs from disk, etc. The commands may be given at any time during the editing phase, and are not preceded by a line number.

Command	Abbreviation	Action
BYE	BYE	Causes an exist to TSS/8 Monitor, user has left BASIC.
CATALOG	CAT	Returns a list of programs which are on file under your account number.

Command	Abbreviation	Action
COMPILE	COM name	BASIC compiles the program in core and stores it on disk with the given name.
DELETE	DEL n n	Delete the line with line number n, an alternate form is to type the line number and the RETURN key.
	DEL n, m	Delete the lines with line numbers n through m inclusive
EDIT	EDI n [c]	Allows the user to search line n for the character c.
KEY	KEY	Return to KEY (normal) mode. (See TAPE)
LIST	LIS LIS n LIS n, m	List the entire program in core. List line n. List lines n through m inclusive.
NEW	NEW	BASIC will clear core and ask for the new program name.
OLD	OLD	BASIC will clear core, ask for the old program name, and retrieve the program from disk, leaving it in core.
REPLACE	REP REP name	Replace the old file on disk with the updated version of the same name currently in core. If a name is not indicated under which BASIC is to store the new version, the old name is retained.
RUN	RUN	Compile and run the program currently in core.
SAVE	SAV SAV name	Save the contents of user core as the file whose name is indicated.
SCRATCH	SCR	Erase the current program from core.
TAPE	TAP	Enter TAPE mode, characters typed will not echo on the console paper.
UNSAVE	UNS name UNS name, . . .	Delete the named program(s) from the disk.
CTRL/C	CTRL/C	Stops a running program, types ↑C and returns to the editing phase. BASIC replies READY.

SUMMARY OF BASIC ERROR MESSAGES

The following error messages may be printed by BASIC during the editing phase:

Message	Explanation
WHAT?	The editor cannot understand the command just given.
BAD FILE NAME	An illegal character was put in the file name.
CAN'T DELETE: name	UNSAVE cannot delete the file with the name given.
DUPLICATE FILE NAME.	BASIC cannot SAVE over an existing file, use a different name, or use the REPLACE command.
ILLEGAL LINE NUMBER	Line number was outside range of 1 to 2046.
CAN'T FIND "name" CAN'T FIND "name" FOR USER n CAN'T FIND "name" IN SYSTEM LIBRARY	The file name given following OLD PROGRAM NAME—cannot be opened. Either it does not exist or it is read protected against this user, as indicated.
EXECUTE ONLY FILE	Attempt to LIST or change a BASIC compiled file.

During input to the editor or when executing an INPUT command the following messages may be printed in response to input:

Message	Explanation
LINE TOO LONG	The line just typed exceeded the available core buffer, retype the line.
\$ DELETED	In response to an ALTMODE character the line has been deleted. Retype the line.
(bell-bell)	Two bells mean that the previous character was illegal, it is automatically deleted.
←	Back arrow is printed any time a RUBOUT or SHIFT/O is used, the previous character is deleted.

The following error messages may be typed out by BASIC following a RUN command:

Message	Explanation
ARRAY USED WITHOUT DIM STATEMENT IN LINE n	All arrays must be previously defined in a DIMENSION statement.
DEF STATEMENT MISSING	A function was called which was not defined in a DEF statement.
DIMENSION TOO LARGE IN LINE n	Self explanatory.
FOR WITHOUT NEXT	Unmatched FOR statement in program.
GOSUB—RETURN ERROR IN LINE n	Either subroutines are nested too deeply, or a RETURN was encountered without a previous GOSUB.
ILLEGAL CHARACTER IN LINE n	Self explanatory.
ILLEGAL CONSTANT IN LINE n	Format of a constant in line n is not valid.
ILLEGAL FOR NESTING IN LINE n	FOR-NEXT loops have been nested too deeply, or NEXT statements were encountered before the FOR was executed.
ILLEGAL FORMAT IN LINE n	Illegal syntax for BASIC statement.
ILLEGAL FORMULA IN LINE n	Error in expression syntax.
ILLEGAL INSTRUCTION IN LINE n	Statement in line n was not a legal BASIC command.
ILLEGAL LINE NUMBER IN LINE n	Line number n is outside the range 1 to 2046.
ILLEGAL VARIABLE IN LINE n	An array variable was used in line n, where it was not permissible.
IMPROPER DIM STATEMENT IN LINE n	Syntax error in DIM statement, or an array name that was previously dimensioned is reused.
MISUSED TAB IN LINE n	The TAB function may appear only in PRINT statements.
NEXT WITHOUGH FOR IN LINE n	Self explanatory.

Message	Explanation
NO END STATEMENT	Self explanatory.
OUT OF DATA IN LINE n	Attempt to do a READ past the available data.
PROGRAM TOO LARGE.	Self explanatory. Try reducing arrays or use fewer variables.
STACK OVERFLOW IN LINE n	Expression too complicated. Try typing it as two separate statements.
SUBSCRIPT ERROR IN LINE n	Negative subscript was calculated for an array.
UNDEFINED LINE NUMBER, LINE n	Tried to reference a line which does not exist.

The following messages are typed out at execution and are non-fatal (i.e., the program continues to execute):

Message	Explanation
IC IN n	Illegal constant in INPUT, retype the value.
LN IN n	An attempt to compute the logarithm of a number less than or equal to zero. The maximum negative number will be used as the result.
MORE?	Response to INPUT did not contain the number of values requested. Respond by supplying the additional values.
OV IN n	Overflow—value is too large for BASIC to use, the largest possible number will be used instead.
PW IN n	Attempt to raise a negative number to a non-integer power. The absolute value raised to the indicated power will be used instead.
SQ IN n	Attempt to compute the square root of a negative number. The square root of the absolute value will be used instead.
TOO MUCH INPUT, EXCESS IGNORED	Response to INPUT contained more values than requested. This message has no effect on the program.

Message	Explanation
UN IN n	Underflow—value is too small for BASIC to use, zero will be used instead.
/φ IN n	Attempt to divide by zero. The largest possible number will be given as a result.

The following errors may occur during any BASIC operation:

Message	Explanation
ABORT ↑ BS •	A non-recoverable disk error has occurred. BASIC halts.
DISK FULL	There is no room left on the disk; delete some files and try again.
ILLEGAL OPERATION IN LINE n	These are failures of BASIC. When they occur they should be reported via a SOFTWARE TROUBLE REPORT.
SYSTEM ERROR	
SYSTEM I-O ERROR	TSS/8 disk I/O failure, try again.

Chapter 13

4K Assemblers

CONTENTS

Introduction to 4K Assemblers	13-5
PAL III Programming	13-7
Character Set	13-7
Numbers	13-8
Format Effectors	13-8
Form Feed	13-8
Tabulations	13-9
Statement Terminators	13-9
Statements	13-10
Labels	13-11
Instructions	13-11
Operands	13-11
Comments	13-11
Coding Practices	13-11
Symbols	13-12
Internal Symbol Representation	13-12
Symbolic Addresses	13-13
Symbolic Instructions	13-15
Symbolic Operands	13-15
Symbol Table	13-16
Direct Assignment Statements	13-16
Expressions	13-18
Address Assignments	13-21
Current Address Indicator	13-22
Indirect Addressing	13-22
Autoindexing	13-23
Instructions	13-24
Memory Reference Instructions	13-24
Microinstructions	13-25
Pseudo-Operators	13-27
Altering the Permanent Symbol Table	13-31
Program Preparation and Assembler Output	13-33
Operating Procedures	13-36
Summary of Diagnostic Messages	13-38

MACRO Programming	13-41
Numbers	13-41
Double Precision Integers	13-41
Floating Point Constants	13-42
Characters	13-44
Expressions	13-44
Origin Setting	13-45
Link Generation	13-46
Literals	13-48
Text Facility	13-51
Single Character Text Facility	13-51
Text Strings	13-51
User Defined Macros	13-52
Defining a Macro	13-53
Restrictions on Macros	13-54
Symbol Table Sizes	13-56
Symbol Table Modification	13-56
Symbol Representation in MACRO and PAL-D	13-57
Pseudo-Ops	13-57
Operating Procedures	13-58
Assembler Output	13-58
Versions of MACRO	13-58
Operating Instructions	13-59
Summary of Error Diagnostics	13-61
4K PAL-D Programming	13-65
Listing Control	13-65
FIELD Pseudo-Op	13-65
Pseudo-Operators	13-66
Program Preparation and Assembler Output	13-66
Under TSS/8	13-67
Under the Disk Monitor System	13-68
Summary of 4K PAL-D Error Diagnostics	13-70

INTRODUCTION TO 4K ASSEMBLERS

This chapter contains a description of each of the 4K Assemblers: the first and most basic of which is PAL III. After you become familiar with the PAL III language (information on which is contained in Volume 1 in this set) you may want to consider the other two PDP-8 Assemblers: MACRO and 4K PAL-D.

MACRO is similar to PAL III with the following additional features: user defined macros, double precision integers, floating point constants, arithmetic and Boolean operators, literals, text facilities, and automatic off-page linkage generation. It is recommended when any of the additional features mentioned are desired, and when a large symbol table is not required.

4K PAL-D is similar to MACRO excluding macros but with a larger symbol table. It is used only in the PDP-8/I Disk Monitor System and the PDP-8 Time Sharing System (TSS/8). It also requires a 4K core memory for operation.

It is assumed in this chapter that you are familiar with the material presented in the first five chapters of Volume 1; the programming language discussed in those chapters is PAL III. Loading procedures and operating instructions for PAL III can also be found in Volume 1.

In Appendix A2 there is a summary of the PDP-8 permanent symbol tables. These symbols are used in each of the languages discussed in this chapter with exceptions as noted.

Several other System Library Programs are useful in assembly language programming. You can use the Symbolic Tape Editor to change, correct or create your program at the Teletype. After assembly, DDT is useful for debugging and communicating with your program. You will find more on these and other useful programs in Volume 1.



PAL III PROGRAMMING

PAL III (an acronym for Program Assembly Language, version III) is a two pass Assembler (with an optional third pass) designed for the 4K PDP-8 family of computers. Your program, written in the PAL III source language, is translated by the Assembler into a binary tape in two passes through the Assembler. The binary tape is loaded by the Binary Loader into the computer for execution.

During the first pass of the assembly, all user symbols are defined and placed in the Assembler's symbol table. During the second pass, the binary equivalents of the input source language are generated and punched. The Assembler's third pass produces a printed assembly listing (a listing of your program's instructions with the location, generated binary, and source code side by side on each line).

The Assembler requires a basic PDP-8 family computer with a 4K core memory, and a Teletype console. The Assembler can use either the high-speed reader, the high-speed punch, or both. The user can change the Assembler's permanent symbol table to reflect his specific machine configuration, as explained under the section on Altering the Permanent Symbol Table.

CHARACTER SET

Legal Characters

The following characters are acceptable to PAL III:

1. The alphabetic characters: A through Z
2. The numeric characters: 0 through 9
3. The special characters:
 - a. Printing characters

+	plus	;	semicolon
-	minus	\$	dollar sign
,	comma	.	period
=	equal sign	/	slash
*	asterisk		
 - b. Nonprinting keyboard characters:
SPACE
TAB
RETURN

4. Ignored characters:
 - FORM FEED
 - blank tape
 - RUBOUT
 - code 200
 - LINE FEED

Illegal Characters

All other characters are illegal (except when used in a comment) and cause the illegal character message:

IC xxxx AT nnnn

during pass 1, where xxxx is the octal value of the offending character and nnnn is the value of the current location counter where it occurred. Illegal characters are ignored and assembly can proceed. The current location counter contains the address in which the next word of object code will be assembled. If the illegal character occurs in the middle of a symbol, the symbol is terminated at that point.

NUMBERS

Any sequence of digits delimited by punctuation characters forms a number. For example:

1
35
4372

The radix control pseudo-ops, OCTAL and DECIMAL, indicate to the Assembler the radix (number base) to be used in number interpretation. The radix is initially set to octal (base 8) and remains octal unless you change it.

The pseudo-op DECIMAL indicates that all numbers which follow are to be interpreted directly as decimal numbers until the occurrence of the pseudo-op OCTAL. The pseudo-op OCTAL indicates that all numbers which follow are to be interpreted as octal until the next occurrence of the pseudo-op DECIMAL. (For an explanation of the internal representation of numbers in the PDP-8, see Volume 1.)

FORMAT EFFECTORS

Form Feed

The form feed code, if present in a PAL III program, will cause the Assembler to output 12 blank lines during the pass 3

Assembly listing. This is useful in creating a page by page listing. form feed is generated by a SHIFT/L.

Tabulations

Tabulations are used in the body of a source program to provide a neat readable listing. Tabs separate fields into columns; for details see the Symbolic Tape Editor Manual. For example, a line written:

```
GO, TAD TOTAL / MAIN LOOP
```

is much easier to read if tabs are inserted to form:

```
GO,      TAD TOTAL      / MAIN LOOP
```

Statement Terminators

Either the semicolon (;) or the RETURN key may be used as a statement terminator. The semicolon is considered identical to a carriage return/line feed except that it will not terminate a line. For example:

```
TAD A      / THIS IS A COMMENT; TAD B
```

The entire expression between the slash (/) and the carriage return is considered a comment. The Assembler ignores the TAD B.

If, for example, the user wishes to write a sequence of instructions to rotate the contents of the accumulator and link six places to the right, it might look like:

```
RTR  
RTR  
RTR
```

The programmer can alternatively place all three instructions on a single line by separating them with the special character semicolon (;) and terminating the line with a carriage return. The above sequence of instructions can then be written:

```
RTR;RTR;RTR
```

These multi-statement lines are particularly useful when setting aside a section of data storage for use during processing. For ex-

ample, a 4-word cleared block could be reserved by specifying either of the following formats:

```
LIST,  0;      0;      0;      0
```

or

```
LIST,  0  
      0  
      0  
      0
```

Either format may be used to input data words which may be in the form of numbers, symbols, or expressions. Symbols and expressions will be explained later. Each of the following lines generate one storage word in the object program:

```
DATA,  7777  
      A+C-B  
      S  
      123+B2
```

STATEMENTS

PAL III source programs are usually prepared on a Teletype, with the aid of the Symbolic Tape Editor, as a sequence of statements. Each statement is written on a single line and is terminated by striking the RETURN key. PAL III statements are virtually format free; that is, elements of a statement are not placed in numbered columns with rigidly controlled spacing between elements, as in punched-card oriented assemblers.

There are four types of elements in a PAL III statement which are identified by the order of their appearance in the statement, and by the separating (or delimiting) character which follows or precedes the element.

Statements are written in the general form:

```
label,      instruction operand      /comment
```

The Assembler interprets and processes these statements, generating one or more binary instructions, data words, or performing an assembly process. A statement must contain at least one of these elements and may contain all four types.

Labels

A label is the symbolic name created by the source programmer to identify the location of the statement in the program. If present, the label is written first in a statement and terminated by a comma.

Instructions

An instruction may be one or more of the mnemonic machine instructions (see Appendix A2), or a pseudo-operation (pseudo-op) which directs assembly processing. The assembly pseudo-ops are described later in this chapter. Instructions are terminated with one or more spaces (or tabs if an operand follows) or with a semicolon, slash, or carriage return.

Operands

Operands are usually the octal or symbolic addresses of the data to be accessed when an instruction is executed, but they can be any expression, or an argument of a pseudo-op. In each case, interpretation of operands in a statement depends on the statement instruction. Operands are terminated by a semicolon, slash, or carriage return.

Comments

Following a slash mark the programmer may add notes to a statement. Such comments do not affect assembly processing or program execution, but are useful in the program listing for later analysis or debugging. The Assembler ignores everything from the slash to the next carriage return. (For an example see the section on Statement Terminators, preceding.)

It is possible to have nothing but a carriage return on a line, resulting in a space in the final listing. An error message is not given.

CODING PRACTICES

A neat printout (or program listing, as it is usually called) makes subsequent editing, debugging, and interpretation much easier than if the coding were laid out in a haphazard fashion. The coding practices listed below are in general use, and will result in a readable, orderly listing.

1. A title comment begins with a slash at the left hand margin.

2. Pseudo-ops may begin at the left margin; often, however, they are indented one tab stop to line up with the executable instructions.
3. Address labels begin at the left margin. They are separated from succeeding fields by a tabulation.
4. Instructions, whether or not they are preceded by a label field, are indented one tab stop.
5. A comment is separated from the preceding field by one or two tabs (as required) and a slash, unless it occupies the whole line, in which case it usually begins with a slash at the left margin.

SYMBOLS

A symbol is a string of letters and digits beginning with a letter and delimited by a non alphanumeric character. Although a symbol may be any length, 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.

Pseudo-ops have fixed meanings, and cannot be redefined by the programmer.

The Assembler has in its permanent symbol table definitions of the symbols for all PDP-8 pseudo-op codes, memory reference, operate and IOT (Input/Out Transfer) instructions, which may be used without prior definition by the user. All other symbols must be defined in the source program. For example:

1. Permanent symbols:

HLT is a symbolic instruction whose value of 7402 is taken by the Assembler from the permanent symbol table.

2. User defined symbols:

A is a user defined symbol. When used as a symbolic address label, its value is the address of the instruction it precedes. This value is assigned by the Assembler. The user may assign values to symbols by using a direct assignment statement of the form $A = 1234$ which will be explained later.

Internal Symbol Representation for PAL III

Each permanent and user defined symbol occupies four words in the symbol table storage area, as shown on the next page.

Word 1	C1	C2	Octal code or address
Word 2	C3	C4	
Word 3	C5	C6	
Word 4			

where C1, C2, . . . , C6 represent the first character, second character, . . . , sixth character respectively. (Remember, symbols consist of from one to six characters.) For a permanent symbol, word 4 contains the octal code of the symbol; for a user defined symbol, word 4 contains the address of the symbol. For example: the permanent symbol TAD is represented as follows:

$$\begin{aligned} \text{Word 1} &= 24_8 \times 100_8 + 01 = 2401_8 \text{ or TA} \\ \text{Word 2} &= 04_8 \times 100_8 + 00 = 0400_8 \text{ or D} \\ \text{Word 3} &= 0000 \\ \text{Word 4} &= 1000 \text{ (octal code for TAD)} \end{aligned}$$

The PAL III Assembler distinguishes between pseudo-ops, memory reference instruction, other permanent symbols, and user defined symbols by their relative position in the symbol table.

Symbolic Addresses

A symbol used as a label to specify a symbolic address must appear as the first term in a statement and must be immediately followed by a comma. When used in this way, a symbol is assigned a value equal to the current location counter and is said to be defined.

A defined symbol can be used as an operand, or as a reference to an instruction. As we explained in Volume 1, the user sets or resets the location counter by typing an asterisk followed by the octal absolute address value in which the next program word is to be stored. If not used, PAL III begins assigning addresses at location 200.

```

TAG,      *300 . /SET LOCATION COUNTER TO 300
          CLA
          JMP A
B,        0
A,        DCA B
          .
          .
          .

```

The symbol TAG (on the preceding page) is assigned a value of 0300, the symbol B a value of 0302, and the symbol A a value of 0303.

If a symbol is defined more than once in this manner, the Assembler will print the duplicate tag diagnostic:

```
DT xxxxxx AT nnnn
```

where xxxxxx is the symbol, and nnnn is the value of the location counter at the second occurrence of the symbol definition. The symbol is *not* redefined. For example:

```
*300
START, TAD A
      DCA COUNTER
CONTIN, JMS LEAVE
      JMP START
A,     -74
COUNTER, 0
START,  CLA CLL
      .
      .
      .
```

The symbol START would have a value of 0300, the symbol CONTIN would have a value of 0302, the symbol A would have a value of 0304, the symbol COUNTER (considered COUNTE by the Assembler) would have a value of 0305. When the Assembler processed the next line it would print (during pass 1):

```
DT START AT 0306
```

Since the first pass of PAL III is used to define all symbols in the symbol table, the Assembler will print a diagnostic if, at the end of pass 1, there are any symbols remaining undefined. For example:

```
*7170
A,     TAD C
      CLA CMA
      HLT
      JMP A1
C,     0
$
```

(The dollar sign must terminate all PDP-8 assembly programs.)

would produce the undefined address diagnostic:

UA xxxx AT nnnn

where xxxxxx is the symbol and nnnn is the location at which it was first seen. The entire user's symbol table is printed in alphabetical order at the end of pass 1. In the case of the above example, this would look as follows:

```
A            71 70
UA  A1    AT    71 73
C            71 74
```

The following are examples of legal symbolic addresses:

ADDR,
TOTAL,
SUM,
A1

The following are examples of illegal symbolic addresses:

AD)M, (contains an illegal character)
7ABC, (first character must be alphabetic)
LA BEL, (must NOT contain imbedded space)

Symbolic Instructions

Symbols used as instructions must be predefined by the Assembler or by the programmer. If a statement has no label, the instruction may appear first in the statement, and must be terminated by a space, tab, semicolon, or carriage return. The following are examples of legal operators:

TAD (a mnemonic machine instruction operator)
PAGE (an Assembler pseudo-op)
ZIP (legal only if defined by the user)

SYMBOLIC OPERANDS

Symbols used as operands normally have a value defined by the user. The Assembler allows symbolic references to instructions or data defined elsewhere in the program. Operands may be numbers or expressions.

```
TOTAL,    TAD AC1+TAG
```

The values of the two symbols AC1 and TAG, already defined by the user, are combined by a two's complement add. This value is used as the address of the operand.

Symbol Table

The Assembler processes user defined symbols in source program statements by adding to its symbol table, which contains all defined symbols along with the binary value assigned to each symbol.

Initially, the Assembler's symbol table contains the mnemonic op-codes of the machine instructions and the Assembler pseudo-op codes, as listed in Appendix A2. As the source program is processed, user defined symbols are added to the symbol table.

If, during pass 1, PAL III detects that the symbol table is full (in other words, there is no more memory space to store symbols and their associated values), the symbol table full diagnostic:

```
ST xxxxxx AT nnnn
```

is printed; xxxxxx is the symbol that caused the overflow condition and nnnn is the current location when the overflow occurred. The Assembler halts and may not be restarted.

More address arithmetic should be used to reduce the number of symbols. It is also possible to segment a program (using the PAUSE pseudo-op) and assemble the segments separately, taking care to generate proper links between the assemblies. PAL III's symbol capacity when using the high-speed reader is 558 symbols. The permanent symbol table contains 80 symbols leaving space for 478 possible user defined symbols. When using the low-speed reader PAL III's symbol capacity is 656 symbols, leaving space for 576 user defined symbols.

Direct Assignment Statements

The programmer inserts new symbols with their assigned values directly into the symbol table by using a direct assignment statement of the form:

```
SYMBOL = VALUE
```

VALUE may be a number or expression. No space(s) or tab(s) may appear between the symbol to the left of the equal sign and the equal sign. For example:

```
A=6  
EXIT=JMP I 0  
C=A+B
```

All symbols to the right of the equal sign must be already defined. The symbol to the left of the equal sign and its associated

value is stored in the user's symbol table. The use of the equal sign does not increment the location counter. It is, rather, an instruction to the Assembler itself.

A direct assignment statement may also equate a new symbol to the value assigned to a previously defined symbol. In this case the two symbols share the same memory location.

```
BETA=17  
GAMMA=BETA
```

The new symbol, GAMMA, is entered into the user's symbol table with the value 17.

The value assigned to a symbol may be changed as follows:

```
ALPHA=5  
ALPHA=7
```

The second line of code shown changes the value assigned to ALPHA from 5 to 7. (This will generate an RD error, as explained below.)

Symbols defined by use of the equal sign may be used in any valid expression. For example:

```
*200  
A=100 /DOES NOT UPDATE CLC  
B=400 /DOES NOT UPDATE CLC  
A+B /THE VALUE 500 IS ASSEMBLED AT LOC. 200  
TAD A /THE VALUE 1100 IS ASSEMBLED AT LOC. 201
```

If the symbol to the left of the equal sign has already been defined, the redefinition diagnostic:

```
RD xxxxxx AT nnnn
```

will be printed as a warning, where xxxxxx is the symbol name and nnnn is the value of the location counter at the point of redefinition. The new value will be stored in the symbol table; for example:

```
CLA=7600
```

will cause the diagnostic:

```
RD CLA AT 0200
```

Whenever CLA is used after this point, it will have the value 7600.

Multiple assignments can be carried to two levels only. Where X is some previously defined symbol or combination of symbols:

```
A=B=X      will assemble, but
A=B=C=X    will not assemble.
```

The expression to the right of the second equal sign must be composed completely of numbers and/or previously defined symbols. If this is not the case, a "pushdown stack overflow" diagnostic of the form:

```
PO  xxxx  AT  nnnn
```

will be generated. This is a non-recoverable error condition which causes the assembly to terminate. Continuation is not possible at this point. The error must be corrected and the assembly restarted.

EXPRESSIONS

Symbols and numbers are combined by arithmetic and logical operators to form expressions. There are three operators:

```
+  plus      Signifies two's complement addition
-  minus     Signifies two's complement subtraction
   space     Space is interpreted in context
```

When a space occurs in an expression that does not contain a memory reference instruction, it means an inclusive OR is to be performed. For example:

```
CLA CLL
```

The symbol CLA has a value of 7200 and the symbol CLL has a value of 7100; CLA CLL would produce 7300. User defined symbols are treated as operate instructions. For example:

```
      A=333
      *222
B,    CLA
```

Possible expressions and their values using the symbols just defined are shown below. Notice that the Assembler reduces each expression to one 4-octal-digit word:

A	0333	
B	0222	
A+B	0555	
A-B	0111	
-A	7445	
1-B	7557	
B-1	0221	
A B	0333	(an inclusive OR is performed)
-71	7707	
etc.		

An expression is terminated by either a comma, carriage return, or semicolon (;). If any information is generated to be loaded, the current location counter is incremented. For example:

```
B-7; A+4; A+B
```

produces three words of information and the current location counter is incremented after each expression. The statement:

```
HALT=HLT CLA
```

produces no information to be loaded (it produces an association in the symbol table) and hence does not increment the current location counter.

```
    *4721  
TEMP,  
TEM2, 0
```

The location counter is not incremented after the line TEMP, and hence the two symbols TEMP and TEM2 are assigned the same value, in this case 4721.

Since a PDP-8 instruction has an operation code of three bits as well as an indirect bit, a page bit, and seven address bits, the Assembler must combine memory reference instructions in a manner somewhat differently from the way in which it combines

operate or IOT instructions. The Assembler differentiates between the symbols in its permanent symbol table and user defined symbols. The following symbols are used as memory reference instructions:

AND	0000	Logical AND
TAD	1000	Two's complement addition
ISZ	2000	Increment and skip if zero
DCA	3000	Deposit and clear accumulator
JMS	4000	Jump to subroutine
JMP	5000	Jump
FADD	1000	Floating addition
FSUB	2000	Floating subtraction
FMPY	3000	Floating multiply
FDIV	4000	Floating divide
FGET	5000	Floating GET
FPUT	6000	Floating PUT

When the Assembler has processed one of these symbols, the space following it acts as an address field delimiter.

```

*4100
JMP A
A,   CLA

```

A has the value 4101, JMP has the value 5000, and the space acts as a field delimiter. These symbols are represented as follows:

```

A      100 001 000 001
JMP    100 000 000 000

```

The seven address bits of A are taken, i.e.:

```
000 001 000 001
```

The remaining bits of the address are tested to see if they are zeros (page zero reference); if they are not, the current page bit is set:

```
000 011 000 001
```

The operation code is then ORed into the JMP expression to form:

```
101 011 000 001
```

or, written more concisely in octal:

5301

In addition to the above tests, the page bits of the address field are compared with the page bits of the current location counter. If the page bits of the address field are nonzero and do not equal the page bits of the current location counter, an out-of-page reference is being attempted and the illegal reference diagnostic is printed on pass 2 or pass 3. For example:

```
A,      *4100
        CLA CLL
        .
        .
        .
        *7200
        JMP A
```

The symbol in the address field of the JMP instruction has a value of 4100 while the location counter (the address where the instruction is placed in memory) has a value of 7200. This instruction is illegal because PAL III does not generate off-page reference, and will be flagged during pass 2 or pass 3 by the illegal reference diagnostic:

```
IR 4100 AT 7200
```

(Note, such a diagnostic would not be generated when using MACRO or 4K PAL-D which both automatically generate off-page references).

ADDRESS ASSIGNMENTS

The PAL III Assembler sets the origin, or starting address, of the source program to absolute location (address) 0200 unless the origin is otherwise specified by the programmer. As source statements are processed, PAL III assigns consecutive memory addresses to the instructions and data words of the object program. This is done by automatically incrementing the current location counter each time a memory location is assigned. A statement which generates a single object program storage word increments the location counter by one. Another statement might generate six storage words, incrementing the location counter by six.

Direct assignment statements and some Assembler pseudo-ops

do not generate storage words and therefore do not affect the location counter.

Current Address Indicator

The special character period (.) always has a value equal to the value of the current location counter. It may be used as any integer or symbol (except to the left of an equal sign). For example:

```
*200  
JMP .+2
```

is equivalent to JMP 0202. Also,

```
*300  
. +2400
```

will produce in location 0300 the quantity 2700. Consider

```
*2200  
CALL=JMS I .  
0027
```

The second line (CALL=JMS I .) does not increment the current location counter, therefore, 0027 is placed in location 2200 and CALL is placed in the user's symbol table with an associated value of 4600 (the octal equivalent of JMS I .).

Indirect Addressing

When the character I appears in a statement between a memory reference instruction and an operand, the operand is interpreted as the address (or location) *containing* the address of the operand to be used in the current statement. Consider:

```
TAD 40
```

which is a direct address statement, where 40 is interpreted as the address on page zero containing the quantity to be added to the accumulator. References to addresses on the current page and to

page zero may be done directly. An alternate way to note the page zero reference is with the letter Z, as follows:

```
TAD Z 40
```

This is an optional notation, not differing in effect from the previous example. Thus, if address 40 contains 0432, then 0432 is added to the accumulator. Now consider:

```
TAD I 40
```

which is an indirect address statement, where 40 is interpreted as the address of the address containing the quantity to be added to the accumulator. Thus, if address 40 contains 0432, and address 432 contains 0456, then 456 is added to the accumulator.

NOTE

Because the letter I is used to indicate indirect addressing, it is never used as a variable. Likewise the letter Z, which is sometimes used to indicate a page zero reference, is never used as a variable.

Autoindexing

Interpage references are often necessary for obtaining operands when processing large amounts of data. The PDP-8 computers have facilities to ease the addressing of this data. When absolute locations 10 to 17 (octal) are indirectly addressed, the content of the location is incremented before it is used as an address and the incremented number is left in the location. This allows the programmer to address consecutive memory locations using a minimum of statements.

It must be remembered that initially these locations (10 to 17 on page 0) must be set to one less than the first desired address. Because of their characteristics, these locations are called autoindex registers. No incrementation takes place when locations 10 to 17 are addressed directly. For example, if the instruction to be

executed next is in location 300 and the data to be referenced is on the page starting at location 5000, we can use autoindex register 10 to address the data as follows:

```

0276    1377    TAD C4777    /=5000-1
0277    3010    DCA 10      /SET UP AUTO INDEX
0300    1410    TAD I 10    /INCREMENT TO 5000
.        .        .        /BEFORE USE AS AN ADDRESS
.        .        .
.        .        .
0377    4777    C4777,4777

```

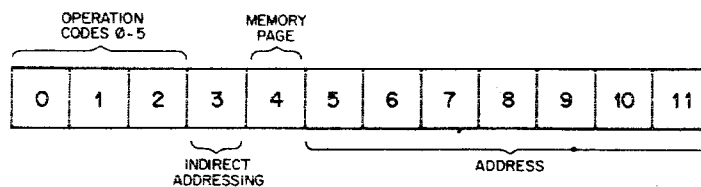
When the instruction in location 300 is executed, the contents of location 10 will be incremented to 5000 and the contents of location 5000 will be added to the contents of the accumulator. When the instruction TAD I 10 is executed again, the contents of location 5001 will be added to the accumulator, and so on.

INSTRUCTIONS

There are two basic groups of instructions: memory reference and microinstructions. Memory reference instructions require an operand; microinstructions do not require an operand.

Memory Reference Instructions

In PDP-8 computers, some instructions require a reference to memory. They are appropriately designated memory reference instructions, and take the following format:



Memory Reference Instruction Bit Assignments

Bits 0 through 2 contain the operation code of the instruction to be performed. Bit 3 tells the computer if the instruction is indirect, that is, if the address of the instruction specifies the location of the address of the operand. Bit 4 tells the computer if the instruction is referencing the current page or page zero. This leaves bits 5 through 11 (7 bits) to specify an address. In these 7 bits,

200 octal or 128 decimal locations can be specified; the page bit increases accessible locations to 400 octal or 256 decimal. For a list of the memory reference instructions and their codes; see Appendix A2.

In PAL III a memory reference instruction must be followed by a space(s) or tab(s), an optional I or Z designation, and any valid expression. In PAL III Memory Reference Instructions may be defined with the FIXMRI instruction, explained under the section on Altering the Permanent Symbol Table. Permanent symbols may be defined using the FIXTAB instruction. In PAL III permanent symbols may be used in address fields as shown below:

```
A=1234
FIXTAB
TAD A
```

Microinstructions

Microinstructions are divided into two groups: operate and Input/Output Transfer (IOT) microinstructions.

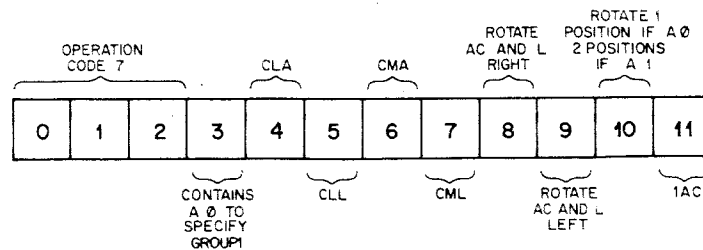
NOTE

If a programmer mistakenly makes an illegal combination of microinstructions, the Assembler will perform an inclusive OR between them; i.e.,

CLL SKP is interpreted as SPA
(7100 7410) (7510)

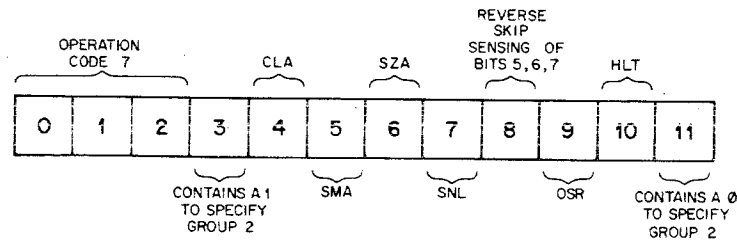
OPERATE MICROINSTRUCTIONS

Within the operate group, there are two groups of microinstructions which cannot be mixed. Group 1 microinstructions perform clear, complement, rotate and increment operations, and are designated by the presence of a 0 in bit 3 of the machine instruction word. (See Permanent Symbol Table list in Appendix A2.)



Group 1 Operate Microinstruction Bit Assignments

Group 2 microinstructions check the contents of the accumulator and link and, based on the check, continue to or skip the next instruction. Group 2 microinstructions are identified by the presence of a 1 in bit 3 and a 0 in bit 11 of the machine instruction word (See Appendix A2).



Group 2 Operate Microinstruction Bit Assignments

Group 1 and Group 2 microinstructions cannot be combined because bit 3 determines either one or the other.

Within Group 2, there are two groups of skip instructions. They can be referred to as the OR groups and the AND group.

<u>OR Group</u>	<u>AND Group</u>
SMA	SPA
SZA	SNA
SNL	SZL

The OR group is designated by a 0 in bit 8, and the AND group by a 1 in bit 8. OR and AND group instructions cannot be combined because bit 8 determines either one or the other.

If the programmer does combine legal skip instructions, it is important to note the conditions under which a skip may occur.

1. OR Group—If these skips are combined in a statement, the inclusive OR of the conditions determines the skip. For example:

SZA SNL

The next statement is skipped if the accumulator contains 0000, or the link is a 1, or both conditions exist.

2. AND Group—If the skips are combined in a statement, the

logical AND of the conditions determines the skip. For example:

```
SNA SZL
```

The next statement is skipped only if the accumulator differs from 0000 and the link is 0.

INPUT/OUTPUT TRANSFER MICROINSTRUCTIONS

These microinstructions initiate operation of peripheral equipment and effect an information transfer between the central processor and the Input/Output device(s).

The Permanent Symbol Table in Appendix A2 contains the commonly used IOTs for the disk, TTY, DECTape, and high speed devices. These and other IOTs are discussed in detail in the *1970 Small Computer Handbook*.

Pseudo-Operators

The programmer uses pseudo-operators to direct the Assembler to perform certain tasks or to interpret subsequent coding in a certain manner. Some pseudo-ops generate storage words in the object program, other pseudo-ops direct the Assembler as to how to proceed with the assembly. Pseudo-ops are maintained in the permanent symbol table.

The function of each PAL III pseudo-op is described below.

INDIRECT ADDRESSING

I Symbolic representation for indirect addressing, must be separated on each side by at least one space.

For example:

```
DCA I ADD
```

The value of the symbol ADD is used as the address of the address in which the contents of the accumulator will be stored.

Z Optional method of denoting a page zero reference.

For example:

```
DCA ADD  
DCA Z ADD
```

The two statements above have the same meaning and generate the same code, where ADD is on page zero.

Both Z and I can be present in the same instruction, separated by at least one space, as follows:

```
DCA Z I ADD
```

which is the same as:

```
DCA I ADD
```

RADIX CONTROL

Numbers used in a source program are initially considered to be octal numbers. However, if the programmer wishes to have certain numbers interpreted as decimal, he can use the pseudo-op DECIMAL.

DECIMAL All following numbers are taken as decimal until the occurrence of the pseudo-op OCTAL.

OCTAL Resets the radix to its original octal base.

EXTENDED MEMORY

When using more than one memory bank, the pseudo-op FIELD instructs the Assembler to output a field setting. This field setting is punched during pass 2 and is recognized by the Binary Loader, which in turn causes all subsequent information to be loaded into the field specified by the expression. A new origin should be specified after using the FIELD pseudo-op.

FIELD n Where n is an integer, a previously defined symbol, or an expression within the range $0 \leq n \leq 7$.

The FIELD pseudo-op causes a field setting (binary word) of the form:

11 xxx 000 where $000 \leq xxx \leq 111_2$

to be output on the binary tape during pass 2. This word is read

by the Loader, which then begins loading information into the new field.

NOTE

CDF and CIF instructions must be used prior to any instruction referencing a location outside of the current field, as shown in the following example:

```

*200
TAD P301
CDF 00
CIF 10
JMS PRINT
CIF 10
JMP NEXT
P301, 301
FIELD 1
*200
NEXT, TAD P302
CDF 10
JMS PRINT
HLT
P302, 302
PRINT, 0
TLS
TSF
JMP .-1
CLA
RDF
TAD P6203
DCA .+1
000
JMP I PRINT
P6203, 6203
```

When FIELD is used, the Assembler follows the new FIELD setting with an origin at location 200. For this reason, if you want to assemble code at location 400 in field 1 you must write:

```
FIELD 1          /CORRECT EXAMPLE
*400
```

The following is *incorrect* and will not generate the desired code:

```
*400          /INCORRECT  
FIELD 1
```

END OF TAPE

The pseudo-op PAUSE signals the Assembler to stop processing the paper tape being read. The current pass is not terminated, and processing continues when the user depresses the CONTINUE key (when using MACRO depress CONTINUE; when using PAL-D, type CTRL/P).

When processing a segmented program, the programmer uses the PAUSE pseudo-op as the last statement of each segment (tape) to halt Assembler processing, giving him time to insert the next segment of his program.

The PAUSE pseudo-op should be used *only* at the physical end of a tape or file and with two or more tapes of one program. When the Assembler reaches a PAUSE statement it does the following:

1. The Assembler stops.
2. This is the physical end of the tape; the Assembler resets the input buffer pointer.
3. Operator intervention is required to put the next tape segment of the program in the reader and press the CONTINUE key.

If a PAUSE is encountered somewhere other than at the physical end of a tape, some of the user code immediately after the PAUSE will not be assembled. This occurs because PAL III has an input buffer to allow maximum use of reader speed. A tape is read in until the buffer is filled or the physical end of the tape is reached. The contents of the buffer are then processed. However, upon recognizing a PAUSE, PAL III resets the buffer to empty and waits for step 3 above.

END OF PROGRAM

The special symbol dollar sign (\$) indicates the end of a program. When the Assembler encounters the dollar sign, it terminates the current pass. The Assembler must read a \$ after each pass before it will correctly proceed with the assembly.

ALTERING THE PERMANENT SYMBOL TABLE

PAL III contains a table of symbol definitions for the PDP-8 and its most common peripheral devices. These are symbols such as TAD, DCA, and CLA, which are used in most PDP-8 Programs. This table is considered to be the permanent symbol table for PAL III; all of the symbols it contains are listed in Appendix A2.

If the user purchases one or more optional devices whose instruction set is not defined among the permanent symbols (for example EAE or an A/D Converter), he would want to add the necessary symbol definitions to the permanent symbol table in every program he assembles. Conversely, the user who needs more space for user defined symbols would probably want to delete all definitions except the ones used in his program. For such purposes, PAL III has three pseudo-ops that can be used to alter the permanent symbol table. These pseudo-ops are recognized by the Assembler only during pass 1. During either pass 2 or pass 3 they are ignored and have no effect.

EXPUNGE Deletes the entire permanent symbol table, except pseudo-ops.

FIXTAB Appends all presently defined symbols to the permanent symbol table. All symbols defined before the occurrence of FIXTAB are made part of the permanent symbol table until the Assembler is reloaded. For example, the PAL III Extended Symbols Tape ends with FIXTAB.

To append the following RF08 disk IOTs to the symbol table, the programmer generates an ANSCII tape of:

```
DCIM=6611
DIML=6615
DIMA=6616
DFSE=6621
DISK=6623
DCXA=6641
DXAL=6643
DXAC=6645
DMMT=6646
FIXTAB
PAUSE
```

The ANSCII tape is then read into core ahead of the symbolic program tape during pass 1. The PAUSE pseudo-op stops assembly, and the Loader waits for the programmer to put the symbolic program tape into the tape reader and press CONTINUE.

Each time the Assembler is loaded, PAL III's permanent symbol table is restored to contain only the permanent symbols shown in Appendix A2.

After altering the symbol table to fit his needs, the user might want to keep PAL III in this state for future use. This can be done by punching a binary of the section of core occupied by PAL III with its new symbol table.

To do this:

1. Read in PAL III and modify symbol table as desired.
2. PAL III's symbol table begins at location 2332 (octal). Count all the symbols in the altered symbol table. Since each symbol and its value require four words multiply this number by 4. Convert this number to octal and add it to 2332 (octal). This number is the upper limit of PAL III. The lower limit is 0001.
3. Using the Binary Punch Routine (DEC-08-YXYA-D), which does a binary core dump to the high-speed or Teletype punch, and the limits as stated in 2 above, punch out the PAL III Assembler.
4. The output of the Binary Punch Routine is the Assembler with the modified symbol table and can be loaded with the Binary Loader. This revised version of the Assembler can thereafter be used instead of the original version.

The third pseudo-op used to alter the permanent symbol table in PAL III (and not present in MACRO or PAL-D) is FIXMRI which may be used only after an EXPUNGE instruction:

FIXMRI Fix memory reference instruction. Memory reference instructions are stored in the permanent symbol table immediately following the pseudo-ops. The letters **FIXMRI** must be followed by one space, the symbol for the instruction to be defined, an equal sign, and the value of the symbol. The pseudo-op must be repeated for each memory reference instruction to be

defined. All memory reference instructions must be defined before the definition of any other symbols.

For example:

```
EXPUNGE
FIXMRI TAD=1000
FIXMRI DCA=3000
CLA=7200
FIXTAB
PAUSE
```

When the preceding program segment is read into the Assembler during pass 1, all symbol definitions are deleted and the three symbols listed are added to the permanent symbol table. Notice that CLA is not a memory reference instruction. This process is often performed to alter the Assembler's symbol table so that it contains only those symbols used at a given installation or by a given program. This may increase the Assembler's capacity for user defined symbols in the program.

PROGRAM PREPARATION AND ASSEMBLER OUTPUT

The source language or symbolic tape is punched in ANSCII code on 8-channel paper tape, using an off-line Model 33 ASR Teletype or the on-line Symbolic Editor. In general, a program should begin with leader code, which may be blank tape, code 200, or RUBOUTs.

Certain codes which the Assembler ignores may be used freely to produce a more readable symbolic program listing. These codes are TAB and LINE FEED. The Assembler also ignores extraneous spaces, carriage return/line feed combinations, and blank tape. When the Assembler encounters a form feed character, it causes 12 blank lines to be output on the listing (in PAL III only).

The two programs below are identical and produce the same binary code. The second, however, was generated using the TAB function of the Symbolic Editor and is easier to read. The first program assembles faster only because there is less paper tape to be read into the computer.

Program #1:

```
*200
/EXAMPLE OF INPUT TO THE FORMAT
/GENERATOR PROGRAM
BEGIN, 0/START OF PROGRAM
KCC
KSF/WAIT FOR FLAG
JMP .-1/FLAG NOT SET YET
KRB/READ IN CHARACTER
DCA CHAR
TAD CHAR
TAD MSPACE/IS IT A SPACE?
SNA CLA
HLT/YES
JMP BEGIN+2/NO:INPUT AGAIN
CHAR,0/TEMPORARY STORAGE
MSPACE,-240/-ANSCII EQUIVALENT
/END OF EXAMPLE
$
```

Program #2:

```
*200
/EXAMPLE OF INPUT TO THE FORMAT
/GENERATOR PROGRAM
BEGIN, 0 /START OF PROGRAM
KCC
KSF /WAIT FOR FLAG
JMP .-1 /FLAG NOT SET YET
KRB /READ IN CHARACTER
DCA CHAR
TAD CHAR
TAD MSPACE /IS IT A SPACE?
SNA CLA
HLT /YES
JMP BEGIN+2 /NO: INPUT AGAIN
CHAR, 0 /TEMPORARY STORAGE
MSPACE, -240 /-ANSCII EQUIVALENT
/END OF EXAMPLE
$
```

The program consists of statements and pseudo-ops; is terminated by the dollar sign (\$), and followed by some trailer code. If the program is large, it can be segmented using the pseudo-op PAUSE, which often facilitates the editing of the source program since each section will be physically smaller.

The Assembler initially sets the current location counter to 0200. This counter is reset whenever the asterisk (*) is processed.

During pass 1, all illegal characters cause a diagnostic to be printed. The tape should be corrected and reassembled.

The Assembler reads the source tape and defines all symbols used. The user's symbol table is printed (or punched) at the end of pass 1. The symbol table is printed in alphabetical order. If any symbols remain undefined, the undefined address diagnostic is printed. If the program listed on the previous page were assembled, the pass 1 symbol table output would be:

```
BEGIN    0200  
CHAR     0213  
MSPACE   0214
```

During pass 2, the Assembler reads the source tape and generates the binary code, using the symbol table equivalences defined during pass 1. The binary tape that is punched may be loaded by the Binary Loader. This binary tape consists of leader code, an origin setting, and data words. At the end of pass 2, a checksum (See Index/Glossary) is punched on the binary tape, and trailer code is generated. During pass 2, the Assembler may diagnose an illegal reference; when using the 33 ASR Punch, the diagnostic is both printed and punched, and is preceded and followed by RUBOUTs. The Binary Loader ignores everything that has been punched on a tape between RUBOUTs.

During pass 3, the Assembler reads the source tape and generates the code from the source statements. The assembly listing is printed (or punched). It consists of the current location counter, the generated code in octal, and the source statement. The symbol table is printed at the end of the pass. If the sample program listed above were assembled, the pass 3 output would be:

```

*200
/EXAMPLE OF INPUT TO THE FORMAT
/GENERATOR PROGRAM
0200      0000      BEGIN,  0          /START OF PROGRAM
0201      6032          KCC
0202      6031          KSF          /WAIT FOR FLAG
0203      5202          JMP  .-1      /FLAG NOT SET YET
0204      6036          KRB          /READ IN CHARACTER
0205      3213          DCA CHAR
0206      1213          TAD CHAR
0207      1214          TAD MSPACE    /IS IT A SPACE?
0210      7650          SNA CLA
0211      7402          HLT          /YES
0212      5202          JMP BEGIN+2  /NO: INPUT AGAIN
0213      0000          CHAR,  0      /TEMPORARY STORAGE
0214      7540          MSPACE, -240  /-ANSCII EQUIVALENT
/END OF EXAMPLE

BEGIN      0200
CHAR       0213
MSPACE    0214

```

OPERATING PROCEDURES

The PAL III Assembler is provided to DEC customers as a binary tape, which is loaded into the PDP-8 memory by means of the Binary Loader, using either the 33 ASR reader or the high-speed reader. The Assembler also uses either the 33 ASR reader or the high-speed reader to read the source language tape, and it uses either the 33 ASR punch or the high-speed punch for output. The selection of I/O devices is made when the Assembler is started. The source language tape must be in the proper reader, with the reader and punch turned on.

When using the high-speed punch, the symbol table is printed on the 33 ASR Teletype if bit 11 of the switch register is a 0. The symbol table is punched on the high-speed punch if bit 11 of the switch register is a 1.

All diagnostics are printed on the 33 ASR (except for the undefined address diagnostic when using the 33 ASR punch, or the high-speed punch if it is included in the machine configuration and turned on. The only diagnostic in pass 2 will be illegal reference. (Since this diagnostic is printed on the 33 ASR, it will also be punched on the binary tape. It will, however, be ignored by the Binary Loader.) The bit 11 switch option can also be used during

pass 3. If the machine is not equipped with a high-speed punch, bit 11 must be set to 0.

In addition to the binary tape of the PAL III Assembler, the user is provided with an ANSCII tape (PAL III Extended Symbols Tape) containing symbol definitions for the instruction sets of the available options to the PDP-8 (card readers, magnetic tapes, and A/D converters). A limited amount of space is available in a 4K system; therefore, expanding the number of permanent symbols that the Assembler recognizes will decrease the maximum number of symbols the user has available.

The following is a description of steps in using the PAL III Assembler (this information can also be found in Volume 1, Chapter 6):

1. Load the Assembler, using either the 33 ASR reader or the high-speed reader.
2. Set 0200 into the switch register; press LOAD ADDRESS.
3. Place the source language tape in the reader, turn on the appropriate reader and the punch.
4. Set bits 0 and 1 of the Switch Register for the proper pass. These settings are:

Bit 0	Bit 1	
0	1	pass 1
1	0	pass 2
1	1	pass 3

Pass 1 is required so that the Assembler can initialize its symbol table and define all user symbols. After pass 1 has been made, either pass 2 or pass 3 can be made.

5. Bit 11 switch options:

pass 1	Bit 11=1	Punch the symbol table on the high-speed punch if it is in the machine configuration.
	Bit 11=0	Print (and punch) the symbol table on the 33 ASR (low-speed punch).
pass 2	11=1	Punch binary tape on high-speed punch.
	11=0	Punch binary tape on low-speed punch.

pass 3 Bit 11=1 Punch the assembly listing tape in ANSCII, on the high-speed punch.
 Bit 11=0 Print the assembly listing on the 33 ASR.

Bit 10 switch options:

pass 3 Bit 10=1 Output TAB (code 211) as 8 space tab stops.
 10=0 Output TAB as TAB RUBOUT (code 211 and 377).

Bit 2 switch options:

passes 1 and 3

 Bit 2=1 Suppress output of symbol table.
 2=0 Output symbol table.

6. Press START to begin pass 1 only. Press CONTINUE to begin passes 2 or 3. The Assembler halts at the end of each pass. Proceed from step 3. If the Assembler has halted because of a PAUSE statement, put the next tape into the reader and press CONTINUE.

Summary of Diagnostic Messages for PAL III

PASS 1 DIAGNOSTICS

The assembler reads the source tape, defines all user symbols, and outputs the user symbol table in alphabetical order. Pass 1 diagnostics are:

IC xxxx AT nnnn Illegal Character

Where xxxx is the value of the illegal character and nnnn is the value of the current location counter when the character was processed.

RD xxxx AT nnnn Redefinition

Where xxxx is the symbol being redefined and nnnn is the value of the current location counter at the point of redefinition. The symbol is redefined.

DT xxxx AT nnnn Duplicate Tag

An attempt is being made to redefine a symbol using the comma. xxxx is the symbol and nnnn is the value of the current location counter at the point of redefinition. The previous

value of the symbol is retained and the symbol is not re-defined.

ST xxxx AT nnnn Symbol Table Full

Where xxxx is the symbol causing the overflow and nnnn is the value of the current location counter at the point of overflow. The Assembler halts and cannot be restarted.

PO xxxx AT nnnn Pushdown List Overflow

This error message occurs in the most recent version of PAL III. An attempt is being made to carry a multiple assignment to more than two levels; i.e., the value of the symbol or expression to the right of the second equal sign was not known to the Assembler when it was encountered in the multiple assignment statement. xxxx is the value of the pushdown stack pointer and, by definition of overflow, this value is the address of the last location in the stack.

The value of the current location counter when overflow is detected is nnn. The Assembler halts at this point without reading more source tape. The CONTINUE key has no effect at this time; however, the Assembler can be restarted at location 200, as indicated in step 2 under Operating Instructions.

UA xxxxxx AT nnnn Undefined Address

Where xxxxxx is the symbol that was used, but never defined, and nnnn is the value of the current location counter when the symbol was first processed. This message is printed with the symbol table at the end of pass 1. The symbol is assigned a value equal to the highest address on the memory page where it was first used.

PASS 2 DIAGNOSTICS

The Assembler reads the source tape, and, using the symbol table defined during pass 1, generates and punches the binary code. This binary tape can then be loaded by the Binary Loader. The pass 2 diagnostic is:

IR xxxx AT nnnn Illegal Reference

Where xxxx is the address being referenced and nnnn is the value of the current location counter. The illegal address is

then treated as if it were on the proper memory page. For example:

```
*7306  
JMP 307
```

would produce:

```
IR 0307 AT 7306
```

and would generate 5307 to be loaded into location 7306.

PASS 3 DIAGNOSTICS

The Assembler reads the source tape and, using the symbol table defined during pass 1, generates and prints the code represented by the source statements. The current location counter, the contents, and the source statement are printed side by side on one line. If bit 11 of the switch register is a 1 and the machine configuration includes the high-speed punch, the assembly listing is punched in ANSCII. The pass 3 diagnostic is Illegal Reference, as in pass 2.

MACRO PROGRAMMING

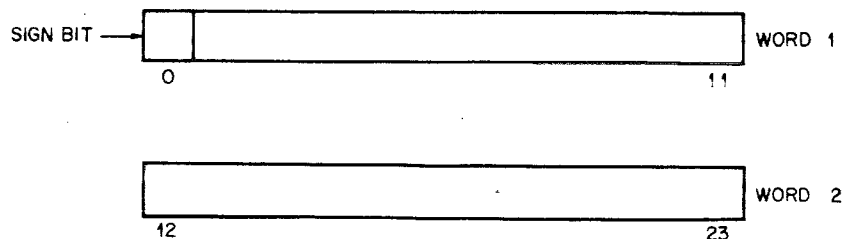
Having mastered the sections on assembly language programming in Volume 1 and the material in this chapter on PAL III, we turn to the MACRO Assembler (sometimes called MACRO-8). MACRO is compatible in most respects with PAL III and has several additional features which will be of interest to more advanced PDP-8 programmers: double precision integers, floating point constants, Boolean operators, link generation, literals, a text facility, and user defined macros.

NUMBERS

The types of numbers allowed in MACRO assemblies are integers, double precision integers, and double precision floating point numbers.

Double Precision Integers

Double precision integers may be positive or negative (stored as two's complement) according to their sign but may not be combined with operators in expressions. They are always taken as *decimal* radix although the current radix of the program is not disturbed. Each double precision integer is allotted two consecutive words with the sign indicated by bit 0 of the first word, as shown below:



The double precision integer mode is entered through the use of the pseudo-op DUBL. All numbers encountered after the occurrence of DUBL are considered double precision integers (stored in

2 words) until an alphabetic character is encountered. Each number is terminated by a carriage return, semicolon (;), or comment. For example:

```

*400
DUBL 679467
      44
      -3
TAG,  CLA
      .
      .
      .

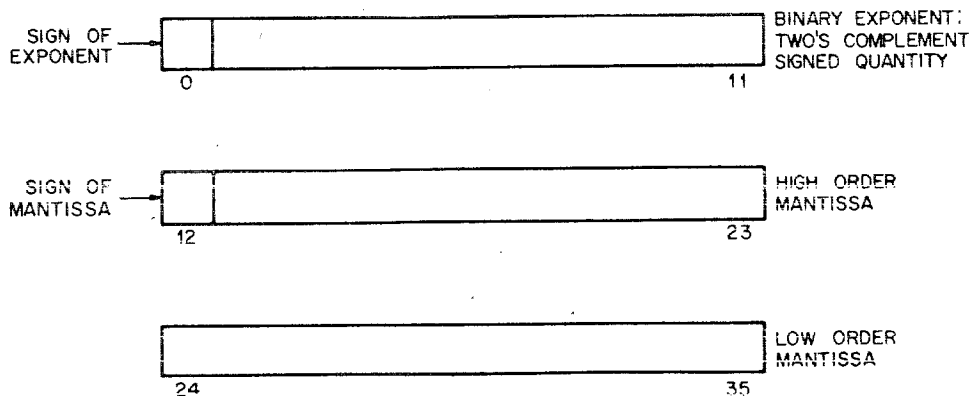
```

The preceding section of code would produce:

Location	Contents	
0400	0245	The numbers indicated under Contents are the octal equivalent of the decimal numbers above. The CLA instruction is given the value 7200 as found in the permanent symbol table. The symbol TAG would have a value of 0406.
0401	7053	
0402	0000	
0403	0054	
0404	7777	
0405	7775	
0406	7200	

Floating Point Constants

Double precision floating point constants may be positive or negative according to their sign but cannot be combined with operators. *Decimal* radix is assumed but the current radix of the program is not altered. Floating point constants are each assigned three words and are stored in normalized form, as shown below:



The exponent is a signed two's complement quantity in one 12-bit word. The signed mantissa is stored in two 12-bit words, maintaining 23 bits of significance, making a total of three words for storage.

The double precision floating point mode is entered through use of the pseudo-op FLTG. All numbers encountered after the use of FLTG will be interpreted as double precision floating point constants until the occurrence of an alphabetic character other than E. The general input format of a floating point number is:

$$\pm ddd.dddE \pm dd$$

where each d is a decimal digit. Any character which is not legally part of the above format (except RUBOUT) terminates input of the number. For example:

```

*400
FLTG  +509.32E-02
      -62.97E04
      1.00E-2
TAG,  CLA

```

would produce upon execution:

<u>Location</u>	<u>Contents</u>
0400	0003
0501	2427
0402	6670
0403	0024
0404	5462
0405	0740
0406	7772
0407	2436
0410	5564
0411	7200

and the symbol TAG would be assigned a value of 0411.

CHARACTERS IN MACRO

In addition to those characters discussed under PAL III, the following characters are used in MACRO:

<u>Symbol</u>	<u>Name</u>	<u>Function</u>
&	ampersand	Combines symbols or numbers (Boolean AND)
!	exclamation point	Combines symbols or numbers (Boolean OR)
"	double quote	Generates ANSCII code
()	parentheses	Defines a literal on the current page
[]	square brackets	Defines a page 0 literal
<>	angle brackets	Defines a macro

EXPRESSIONS IN MACRO

All symbols and numbers (exclusive of pseudo-ops, macro names, and double precision or floating point constants), may be combined with certain arithmetic and logical operators to form expressions. These operators are:

<u>Symbol</u>	<u>Name</u>	<u>Function</u>
+	plus	Signifies two's complement addition modulo 4096 decimal
-	minus	Signifies two's complement subtraction modulo 4096 decimal
!	exclamation point	Signifies Boolean inclusive OR (union)
&	ampersand	Signifies Boolean AND (intersection)
	space	Interpreted in context; can signify an inclusive OR, or act as a field delimiter as in PAL III

Symbols and integers may be combined with any of the above operators. A symbolic expression is evaluated from left to right; grouping of terms (i.e., use of parentheses for grouping) is not permitted. For example:

	<u>A</u>	<u>B</u>	<u>A+B</u>	<u>A-B</u>	<u>A!B</u>	<u>A&B</u>
Value	0002	0003	0005	7777	0003	0002
Value	0007	0005	0014	0002	0007	0005
Value	0700	0007	0707	0671	0707	0000

ORIGIN SETTING IN MACRO

The origin is ordinarily set by use of the special character asterisk (*) as described in PAL III. All symbols to the right of the asterisk must already have been defined. For example, if D has the value 250 then:

*D+10

will set the location counter to 0260.

To ease the programmer's addressing problems, a convention has been defined that divides memory into sections called pages. Each page contains 200 octal locations (128 decimal) numbered 0 to 177 (octal) on that page. There are 40 octal or 32 decimal pages numbered 0 to 37 (octal). Some examples of page numbers and the absolute and relative locations (addresses) are shown below. It must be borne in mind, however, that there is no physical separation of pages in memory.

<u>Page</u>	<u>Absolute Address</u>	<u>Relative Address</u>
0	0—177	0—177
1	200—377	0—177
2	400—577	0—177
36	7400—7577	0—177
37	7600—7777	0—177

To simplify page handling, the pseudo-op PAGE can be used:

PAGE n The PAGE pseudo-op resets the location counter to the first address of page n, where n is an integer, a previously defined symbol, or a symbolic expression.

For example:

```
PAGE 2      sets the location counter to 0400
PAGE 6      sets the location counter to 1400
```

PAGE When used without an argument, **PAGE** resets the location counter to the first location on the next succeeding page. Thus, if a program is being assembled into page 1 and the programmer wishes to begin the next segment on page 2 he need only insert the pseudo-op **PAGE**, as follows:

```
      .
      .
      .
PAGE  JMP  .-7
      CLA
      .
      .
      .
```

LINK GENERATION

In addition to automatically handling symbolic addressing on the current page of core memory, **MACRO** automatically generates links for out-of-page references. **MACRO** compares the page bits of the address field with the page bits of the location counter. If the page bits of the address field are nonzero (not a page 0 reference) and do not equal the page bits of the location counter, an out-of-page reference is being attempted.

If reference is made to an address not on the page where the instruction is located, the Assembler sets the indirect bit (bit 3), and an indirect address linkage will be generated on the current memory page. If the out-of-page reference is already an indirect one, the error diagnostic II (illegal indirect) will be printed during pass 2.

When a link is generated, the LG (link generated) message will be printed on pass 2 (in **MACRO** only, not in **PAL-D**). In the case of several out-of-page references to the same address, the link will be generated only once, but the LG message will be printed each time.

To suppress the LG message when links are generated, make the following change in **MACRO** (only recommended when the pro-

gram has been debugged): change the contents of location 1234 to 7200. Do not make this change unless the LG message is not wanted.

```
      *2117
A,    CLA
      .
      .
      .
      *2600
      JMP A
```

In the example above, the space preceding the user defined symbol A acts as an address field delimiter. The Assembler will recognize that the register labelled A is not on the current page (in this case 2600 to 2777) and will generate a link to it as follows:

1. In location 2600 the Assembler will place the word 5777 which is equivalent to JMP I 2777.
2. In address 2777 (the last available location on the current page) the Assembler will place the word 2117 (the actual address of A).

Although the Assembler will recognize and generate an indirect address linkage when necessary, the programmer may indicate an explicit indirect address by using an explicit indirect address by using the special symbol I. This must be between the instruction code and the address field, as it would be placed in PAL III. The Assembler cannot generate a link for an instruction that is already specified as being an indirect reference. In this case, the Assembler will print the message II (illegal indirect). For example:

```
      *2117
A,    CLA
      .
      .
      .
      *2600
      JMP I A
```

The above coding will not work because A is not defined on the page where JMP I A is attempted.

LITERALS

Symbolic expressions appearing in the operand part of an instruction usually refer to locations containing the quantities being operated upon. Therefore, the programmer must explicitly reserve locations to hold his constants. The MACRO language provides a means (known as literals) 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:

```
*200
.
.
.
CLA
TAD INDEX
TAD C2
DCA INDEX
.
.
.
C2, 2
```

Using a literal, the above coding would be rewritten as:

```
*200
.
.
.
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 word 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 word in the linkage and literals list beginning at the top of the current memory page. The instruction in which the literal appears is encoded with an address referring

to the address of the literal. A literal is assigned to storage the first time it is encountered; subsequent reference to that literal from the current page is made to the same register.

If the programmer wishes to assign literals to page zero rather than to the current page, he may use square brackets, [and], in place of the parentheses. This enables the programmer to reference a single literal from any page of core. For example:

```
*200
TAD [2]
.
.
.
*500
TAD [2]
.
.
.
```

For the first and succeeding times the literal 2 is referenced, identical code is generated to a single location on page zero containing the literal.

Whether on page zero or the current page, the right (closing) member may be omitted. The following examples are acceptable

```
TAD (777
AND [JMP
```

In the second example, the instruction AND [JMP has the same effect as AND [5000.

Literals can be nested, for example:

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

will generate the following:

<u>Location</u>	<u>Contents</u>
0200	1376
0376	1377
0377	0030

This type of nesting can be carried to as many levels as necessary.

Literals are stored on each page starting at page address 177 (relative) and extend toward page address 0 (relative). Only 127 decimal or 177 octal literals may be placed on page zero. If a literal is generated for a nonzero page and the origin is then set to another page, the current page literal buffer is punched out (during pass 2); this does not effect later execution. If the origin is then reset to the previously used page, the same literal will be generated if used again, but it will not destroy previously used literals on that page.

To summarize, literals may take the following forms:

[C	(C
[V	(V
[<u>"</u> C	(<u>"</u> C
[I	(I
[E	(E

Where C is a constant, V is a variable, I is an instruction, and E is an arithmetic expression. [indicates a page 0 reference and (indicates a current page reference.

Arithmetic expressions may consist of constants, variables, and operators but must not include literals. An instruction may contain a literal, for example:

```
TAD      (JMP I [500
```

is valid, while:

```
TAD      (A+(50
```

is not valid. Literals may be used as the address part of a memory reference instruction:

```
TAD      (50
```

or in place of an instruction:

```
(MSG4
```

which causes the location address of the literal (MSG4 to be assembled at the point where it occurs in the program.

NOTE

If a large number of nested literals or particularly large numbers of literals are used, the literal list may be output before the logical end of the page. This will not affect later execution.

TEXT FACILITY

Single Character Text Facility

If a single character is preceded by a double quote ("), the 8-bit value of ANSCII code for the character is inserted instead of interpreting the letter as a symbol. For example:

```
CLA  
TAD      ("A
```

will place the constant 0301 in the accumulator.

Text Strings

A string of text characters can be entered by giving the pseudo-op TEXT followed by a space, any delimiting character, a string of text, and a repetition of the same delimiting character. For example:

```
TEXT      ATEXTA
```

The character codes are stored two per word in ANSCII code that has been trimmed to six bits. Following the last character, a 6-bit zero is inserted as a stop code. The above statement would produce:

```
2405  
3024  
0000
```

The string in the following example:

```
TEXT    /BOB/
```

would produce:

```
0217  
0200
```

The TEXT pseudo-op could also be used as part of a calling sequence to a subroutine:

Example 1:

```
JMS MESS  
TEXT/    /
```

Example 2:

```
JMS MESS  
NOWDS    /NO WDS IN MESSAGE  
ADDRESS  /ADDRESS OF MESSAGE  
.  
.  
.  
ADDRESS,TEXT/    /
```

NOTE

While the TEXT pseudo-op causes characters to be stored in a trimmed code, the use of the single character control (") causes characters to be stored as a full 8-bit ANSCII code.

USER DEFINED MACROS

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

Defining a Macro

The macro name must be defined before it is used. The macro is defined by means of the pseudo-op `DEFINE` followed by the macro name and a list of dummy arguments separated by spaces. For example, a simple macro to move the contents of word A to word B and leave the result in the accumulator could be coded as follows:

```
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 definition of the macro is enclosed in angle brackets.

The above definition of the macro `MOVE` can also be written as follows:

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

The definition of the macro is enclosed in angle brackets, as mentioned above and the semicolon characters indicate the termination of a line of code, as in `PAL III`.

When a macro name is processed by the Assembler, the real arguments replace the dummy arguments. For example, assuming that the macro `MOVE` has been defined above:

```
*400
0
-6
MOVE A,B
```

produces the following code:

<u>Location</u>	<u>Contents</u>
0400	0000
0401	7772
0402	7200
0403	1200
0404	3201
0405	1201

Notice that a macro definition has spaces separating the dummy arguments and the macro call has commas separating the macro arguments.

A macro need not have any arguments. For example, a sequence of coding to rotate the accumulator and link six places to the left might be coded as a macro by means of the following code:

```
DEFINE ROTL  
<RTL;RTL;RTL>
```

The entire macro definition is placed in the macro table, two characters per word, with a dummy argument value replacing the symbolic names. For example:

```
DEFINE LOAD A  
<CLA  
TAD A>
```

is stored, in the macro table, roughly as follows:

|CL|A |TA|D |7700|>00

where the vertical lines indicate successive 12-bit words. Comments and line feeds are not stored.

The macro definition can consist of any valid coding except for TEXT (or " type) statements.

Restrictions on Macros

1. Macros cannot be nested, i.e., another macro name or definition cannot appear in a macro definition and cannot be brought in as an argument at the time a macro is referenced.

2. TEXT (or " type) statements cannot appear in a macro definition.
3. Arguments cannot be another macro name, a TEXT pseudo-op or a " character.
4. The symbols used as dummy arguments must not have been previously defined or referenced.
5. A macro cannot be redefined.

Consider the following macro definition:

```
DEFINE LOOP A B
<TAD A
DCA B
TAD COUNT
ISZ B
JMP .-2>
```

A macro is referenced by giving the macro name, a space, and the list of real arguments, separated by commas. There must be at least as many arguments in the macro call as in the corresponding macro definition. When a macro is referenced, its definition is found, expanded, and the real arguments replace the dummy arguments. The expanded macro is then processed in the normal fashion. For example the macro call:

```
LOOP X, Y2
```

in the context of the program in which it appears, is equivalent to:

```
TAD X
DCA Y2
TAD COUNT
ISZ Y2
JMP .-2
```

The macro table shares the available space (604 decimal words, which is equivalent to 151 symbols) with the symbol table. Thus the programmer must be aware of the amount of room required by his macros and the fact that each symbol occupies four words of memory. Also, the arguments of a macro call are temporarily stored in this buffer space while the macro is being expanded.

SYMBOL TABLE SIZE

To incorporate these new features, it was necessary to decrease the size of the symbol table and because of this, programs that were originally coded to be assembled by PAL III might have too many symbols to be assembled by MACRO. If switch registers 10 and 11 are set to 1 during assembly, MACRO's external (user) symbol table will be extended.

Symbol Table Modification

Because of the small amount of core (604 decimal words) remaining to be used for programmer symbols and the macro table, the following suggestions are offered which allow a particular installation or individual to conserve symbol table space.

By use of the pseudo-ops EXPUNGE and FIXTAB, unnecessary instruction mnemonics can be removed from the symbol table, making more space available for programmer defined symbols and macros. This also decreases assembly time as the unused instruction symbols are not involved in the symbol table searches. The most often used instruction mnemonics should be assembled first, so that they will be in core next to the special characters and pseudo-ops. This is desirable because the symbol search routine starts searching at the top of the table and works down.

At an installation that does not have optional equipment available, the corresponding instruction sets can be removed. A symbolic tape beginning with EXPUNGE, containing all necessary instruction mnemonics, and ending with FIXTAB and \$ could be assembled (only during pass 1 is this necessary) by MACRO prior to any other assemblies. For example:

```
EXPUNGE
AND=0000
TAD=1000
CLA=7200
.
.
.
FIXTAB
$
```

The pseudo-op PAUSE could also be used with the above tape, as the first tape of a multiple tape assembly. See the list of permanent symbols in Appendix A2.

Internal Symbol Representation for MACRO and PAL-D

Each permanent and user defined symbol occupies four words (locations) in the symbol table storage area, as shown below:

	0	1	2	
Word 1			$C1 \times 45_8 + C2$	first 2 characters
Word 2			$C3 \times 45_8 + C4$	second 2 characters
Word 3			$C5 \times 45_8 + C6$	third 2 characters
Word 4				octal code or address

where $C1, C2, \dots, C6$ represent the first character, second character, \dots , sixth character, respectively. (Remember symbols can consist of from one to six characters.) Bits 0 and 1 of word 1, and bit 0 of word 2 are system flags¹ For a permanent symbol, word 4 contains the octal code of the symbol; for a user defined symbol, word 4 contains the address of the symbol. For example, the permanent symbol TAD is represented as follows:

$$\text{Word 1} = 24_8 \times 45_8 + 01 = 1345_8 \text{ or TA}$$

$$\text{Word 2} = 04_8 \times 45_8 + 00 = 224_8 + 4000 = 4224_8 \text{ or D}$$

$$\text{Word 3} = 0000$$

$$\text{Word 4} = 1000 \text{ (octal code for TAD)}$$

Note that the octal code for each character is always scaled by the Assembler so that the character is represented using six bits of a word. For example, ANSCII code for T is 324, it was trimmed to 24; A is 301, it was trimmed to 01, etc. Digits 0 through 9 are scaled to the range 33 through 44.

MACRO PSEUDO OPERATORS

<u>Pseudo-Op</u>	<u>Description</u>
DECIMAL	See PAL III
OCTAL	See PAL III
PAUSE	See PAL III
FIELD	See PAL III

¹ Bit 0 of word 1 signifies a pseudo-op, bit 1 signifies an undefined symbol, bit 0 of word 2 signifies a defined symbol.

<u>Pseudo-Op</u>	<u>Description</u>
Z	See PAL III
I	See PAL III
EXPUNGE	See PAL III and MACRO, sections on Symbol Table Alteration
FIXTAB	See PAL III and MACRO, sections on Symbol Table Alteration
DEFINE	See MACRO, section on User Defined Macros
TEXT	See MACRO, section on Text Facility
FLTG	See MACRO, section on Floating Point Constants
DUBL	See MACRO, section on Double Precision Integer Constants
PAGE	See MACRO, section on Origin Setting
\$	End of pass

MACRO OPERATING PROCEDURES

Assembler Output

MACRO is a two pass Assembler with an optional third pass which produces an octal/symbolic assembly listing. During the first pass, MACRO processes the source tape and places all symbol definitions and macro definitions in its symbol table and macro table, respectively. During the second pass, MACRO processes the source tape and punches the Binary Format Tape. At the end of pass 2, MACRO prints the symbol table (it is also punched if the 33 ASR punch is turned on). This punched table can be read by DDT. The third pass provides a listing of the generated octal code and the original source language.

Versions of MACRO

There are two versions of MACRO which differ with respect to their use of Input/Output equipment: the low speed version uses the 33 ASR reader for all input and the 33 ASR punch for all output; the high speed version uses the photoelectric reader for all input, the high-speed punch for all binary output, and the 33 ASR for printable output such as error printouts, symbol table listing and third pass assembly listing.

NOTE

In the high speed version of MACRO; the high-speed punch may be used as the printable output device by changing the contents of location 0004 to 0600. This is useful for long third pass listings, since the punched output from the high-speed punch can be subsequently listed off-line. It is advised that this change not be made until pass 3, so that pass 1 and pass 2 error messages will come out on the 33 ASR.

Operating Instructions

1. Load MACRO with the Binary Loader.
2. Put the source tape in the reader.
3. Set the switch register to 0200.
4. Depress LOAD ADDRESS.
5. Set switch options (see Table 13-1).
6. Depress START.
7. Turn on the 33 ASR reader (if using low speed version).
8. When MACRO stops reading (after processing a PAUSE statement), place the next tape in the reader and depress CONTINUE. Repeat this step until all tapes have been processed.
9. When MACRO encounters the terminating character, dollar sign (\$), it performs one of the following sets of events depending upon what pass has just been completed (proper operator intervention is then required):

<u>Pass Completed</u>	<u>Events</u>	<u>Operator Intervention</u>
1	Set up symbol table for use in pass 2.	Turn on 33 ASR punch (in high speed version, symbol table is output via 33 ASR). Put source tape in reader, and proceed from step 2 of Operating Procedures above.

2. Terminate current assembly. Punch out page zero constants, checksum, and trailer code on binary tape. Print and punch RUBOUT, the alpha-numerically ordered symbol table, and EOT code, a RUBOUT, and trailer code. Set up for pass 1.
 - a) If pass 3 is desired: (in high speed version the constants of word 0004 could be altered at this point to change output devices) Go to step 2 of Operating Procedures above.
 - b) If pass 3 is not desired: Turn off 33 ASR punch, put next program to be assembled in the reader. Go to step 2 of Operating Procedures, above.
3. Terminate assembly listing. Turn off 33 ASR punch; put next program to be assembled in the reader; hit CONTINUE to enter pass 1.

TABLE 13-1
MACRO SWITCH OPTIONS

<u>Switch Up</u>	<u>Result</u>
None	MACRO enters the next pass as defined in the preceding table. For example: if the previous assembly was terminated during or at the end of pass 1, restarting MACRO with no switches up would cause pass 2 to be entered. MACRO initially starts at pass 1.
0	Restore symbol table to the permanent basic symbols and enter pass 1.
1	Enter pass 2.
2	Enter pass 1 without erasing any previously defined symbols.
3	Enter pass 3. During pass 3, MACRO outputs an octal/symbolic listing of the assembled program. If this pass is terminated before completion, either switch options 0 or 2 may be used to return to pass 1 for subsequent assemblies. MACRO will output as much of the source

Switch Up

Result

- statement (symbolic) as its internal storage capacity will allow. Because of the internal operations during the processing of macro statements, the symbolic output may be meaningless.
- 10 The double precision integer and double precision floating point processors are deleted and may be used for storage of user defined symbols. This increases the size of the symbol table by 64₁₀ symbols.
- 11 The macro processor and the number processors (above) are deleted and can be used for storage of user defined symbols. This increases the size of the symbol table by 125₁₀ symbols.

NOTE: Switches 10 and 11 are sensed whenever pass 1 is entered. MACRO would have to be reloaded to handle subsequent programs that use macros, double precision integers, or floating point numbers.

SUMMARY OF MACRO ERROR DIAGNOSTICS

The format of the error messages is:

ERROR CODE ADDRESS

where ERROR CODE is a two character code which specifies the type of error, and ADDRESS is either the absolute octal address where the error occurred or the address of the error relative to the last symbolic label (if there was one) on that page.

Assembly will continue or can be continued after all errors except SE (Symbol Table Exceeded). If an SE error occurs, the Assembler will halt and cannot be restarted.

Error

Code

Meaning

PE

Current, Non-Zero Page Exceeded

An attempt was made to override a literal with an instruction, or override an instruction with a literal. This can be corrected by decreasing the number of literals on the page, or decreasing the number of instructions on the page.

ZE

Zero Page Exceeded

Same as PE only with reference to page 0.

Error
Code

Meaning

ID **Illegal Redefinition of a Symbol**
An attempt was made to give a previously defined symbol a new value not via =. The symbol was not redefined (This is similar to the Duplicate Tag diagnostic of PAL III.)

IC **Illegal Character**
1. * % ' : ? @ / were processed other than in a comment or a TEXT field. The character is ignored and the assembly continued.
2. A non-valid character was processed. The computer halts with the illegal character displayed in the accumulator. Assembly can be continued by putting the desired character in the switch register and depressing CONTINUE.

IE **Illegal Equals**
An equal sign was used in the wrong context. For example:

```
TAD A+=B  
A+B=C
```

The expression to the left of the equal sign is not a single symbol.

II **Illegal Indirect**
An out of page reference was made, and a link could not be generated because the indirect bit was already set. For example:

```
      *200  
      TAD I A  
      .  
      .  
      .  
      PAGE  
A,      CMA CLL
```

LG **Link Generated**
A warning message; a link was generated for an out-of-page reference at this address. For example:

<u>Error Code</u>	<u>Meaning</u>
	*200
	TAD A
	.
	.
	.
	PAGE
A,	CMA CLL

will result in the following:

<u>Location</u>	<u>Contents</u>
0200	1777
0377	0400
0400	7140

SE Symbol Table Exceeded
 The symbol table overlaps the macro table or vice versa. Assembly is halted and cannot be continued.

IM Illegal Format in a Macro Definition
 The expression after the **DEFINE** pseudo-instruction does not comply with the macro definition position, or structural rules. For example: A macro name is referenced before the macro definition.

US Undefined Symbol
 A symbol has been processed during pass 2 that was not defined by the end of pass 1.

MP Missing Parameter in a Macro Call
 An argument, called for by the macro definition, is missing. For example:

```
DEFINE MAC A B
<TAD A
CIA
DCA B>
MAC SUM
```

BE Two **MACRO** internal tables have overlapped.
 This situation can usually be corrected by decreasing the number of current page literals used prior to this point on the page. If the error persists, please contact the Small Computer Systems Programming Group at Digital Equipment Corporation for assistance.



4K PAL-D PROGRAMMING

The 4K PAL-D Assembler is compatible with the PAL III Assembler. It is also compatible with MACRO with respect to the following features: Boolean operators, linkage generation, literals and a text facility. 4K PAL-D does not have user defined macros, floating point constants, or double precision numbers.

Before reading this section the reader should become familiar with the section on PAL III and read the pertinent sections of MACRO as indicated above. The following information is supplementary to that already mentioned.

A major difference between PAL III and 4K PAL-D is the way in which they recognize Memory Reference Instructions. In PAL-D permanent symbols may not be used in the address field of a memory reference instruction. PAL-D considers a memory reference instruction to be defined as a permanent symbol followed by a space(s) or tab(s) with the address field permitted to be any valid expression not containing a permanent symbol. For example:

```
A=1234  
FIXTAB  
TAD A
```

will not work on 4K PAL-D. The example above would generate the code 1234 by inclusively ORing 1000 and 1234.

LISTING CONTROL

During pass 3, a listing of the source program is printed (or punched). The programmer can, however, control the output of his pass 3 listing by use of the pseudo-op XLIST.

XLIST Those portions of the source program enclosed by XLIST will not appear in the pass 3 listing.

FIELD PSEUDO-OP

In addition to the information on the FIELD pseudo-op provided in the section on PAL-III: when using PAL-D, use of the FIELD pseudo-op causes all literals to be punched.

PAL-D PSEUDO-OPERATORS

<u>Pseudo-Op</u>	<u>Description</u>
FIELD	See PAL III
PAUSE	See PAL III
DECIMAL	See PAL III
OCTAL	See PAL III
Z	See PAL III
I	See PAL III
FIXTAB	See PAL III
EXPUNGE	See PAL III
TEXT	See MACRO, section on Text Facility
PAGE	See MACRO, section on Origin Setting
XLIST	See 4K PAL-D, section on Listing Control
\$	End of Pass.

4K PAL-D PROGRAM PREPARATION AND ASSEMBLER OUTPUT

The information on PAL III Program Preparation and Assembler Output is applicable to PAL-D except as follows:

The symbol table is punched after the listing and is preceded and followed by a small amount of leader/trailer (200) code. The symbol table tape will be punched whether or not a listing is requested, and will appear either on the Teletype punch or on the high-speed punch output, depending upon the device being used.

It is possible to terminate any pass of the assembly by typing a CTRL/P on the console Teletype. ↑P causes PAL-D to go on to the next pass of the assembly. As with the previous version of PAL-D, assembly can be terminated at any time by typing a CTRL/C.

During the listing pass note that blank lines will remain in the listing and the form feed (214 ANSCII) character is ignored.

The 4K PAL-D symbol table has room for 161 symbols in core (about 6 to 8 pages as an average). That number can be expanded as explained in the section about 4K PAL-D on the Disk Monitor System following.

Following pass 2, the binary output can be loaded into core by the Disk Monitor System Binary Loader. Under TSS/8, all three passes are automatically processed.

Under TSS/8

Loading 4K PAL-D in a TSS/8 system is performed by the system manager as described in the *System Manager's Guide*.

Assembling with 4K PAL-D under the TSS/8 Monitor requires no operator intervention between passes. The symbol table is printed at the end of pass 2 and the listing at the end of pass 3. The assembly can be terminated at any point by typing CTRL/C. Control will revert from PAL-D to the TSS/8 Monitor which prints a period at the left-hand margin and waits for the next instruction from the Teletype.

In order to run a PAL-D program on TSS/8 the user types the LOGIN command and his account number and password. He then creates his PAL-D program and saves it as a file using the Symbolic Editor program. When the program is ready to be run, PAL-D is brought into core. The user types:

```
R PALD
```

in reply to Monitor's period. PAL-D signals its presence by requesting an input file name as follows:

```
INPUT:TYPE2
```

The user reply in this case was TYPE2, a user defined name for the source program to be assembled.

PAL-D next requests the name of an output file:

```
OUTPUT:BIN2
```

The user response was BIN2, the name under which the assembled program will be stored.

Optionally, the user can type the RETURN key to specify no output file. This is useful in debugging. A program can be corrected and reassembled any number of times with production of an output file postponed until a satisfactory version is achieved.

PAL-D's final query is whether the user wants a program listing, as follows:

OPTION:

There are two responses: N signifying No and RETURN key signifying Yes. When it receives the last response, PAL-D reads in the user source program from disk and proceeds with the assembly. After assembly, PAL-D returns control to the Monitor which prints a period and waits for the user to supply the next command.

The program can be run by calling the program into core to be run under the direction of the TSS/8 Monitor:

```
.R TYPE2
```

where the period was printed by TSS/8 Monitor and TYPE2 is the name of the output file in which the compiled binary program was stored.

Under the Disk Monitor System

Under the Disk Monitor System the user should first build the Disk Monitor if it is not present on the disk or DECTape, according to the instructions in the *Disk Monitor System Manual*, DEC-D8-SDAB-D.

4K PAL-D is loaded into core as explained in the *Disk Monitor System Manual* using the Binary Loader.

The Assembler is incorporated in the system by loading the paper tape into core using the disk Loader. The Assembler can be saved on the disk or DECTape.

4K PAL-D can be saved on the system device as a system program. This is done by typing the following:

```
.SAVE PALD!0-7577;6200
```

where the period was printed by the Monitor. The user may also want to reserve space on disk for the symbol table by typing, for example:

```
.SAVE .SYM:0-4377;0
```

The limits (0 through 4377) are arbitrary and determine the maximum length of the symbol table. With the SAVE command shown, 4K PAL-D will reserve space for a total of 737 symbols (161 of which are stored in core). The programmer can now type PALD to bring the Assembler into core for use with symbolic source programs, where Monitor types the dot:

.PALD

PAL-D responds with a request for an output file by printing

*OUT-

The user selects the output device by typing one of the following:

T: for the Teletype (low speed reader/punch) or
R: for the high speed reader/punch, or
S: name for output to the system device as file name

PAL-D now prints:

*IN-

and waits for the user to select the input files. Up to five input files can be specified (for example: R:, R:, S: name, R:, R:).

When PAL-D is satisfied that the input file(s) is valid, it will request the third pass listing option by printing:

*OPT-

The user can type:

T Meaning listing and symbols are to be produced on the
 Teletype, or
R Meaning listing and symbols are to be produced on the
 high speed punch, or
Carriage Meaning symbols only (any other character means no
Return third pass is desired).

When the high-speed punch is selected as a listing device, the alphabetic symbol table produced at the end of pass 3 is also produced on the high-speed punch.

PAL-D will now proceed with the assembly, pausing only when user intervention is required (i.e., placing a new paper tape in the reader, turning on the punch, etc.). On these occasions, PAL-D will print an up-arrow (↑) on the Teletype to indicate user intervention is required. When the user has performed the necessary function and is ready to continue with the assembly, he types CTRL/P (which does not echo).

When using the disk as an output device, the compiled binary is ready to be loaded for execution following pass 2.

The symbol table will be output at the end of pass 2 if the third pass has not been requested. If pass 3 is requested, PAL-D will follow the assembly listing with the user's symbol table in alphabetical order (in addition to the assembled binary output).

Assembly can be terminated and control returned to the Monitor at any time by typing CTRL/C. When the assembly is complete, control will automatically be returned to the Monitor.

SUMMARY OF 4K PAL-D ERROR DIAGNOSTICS

PAL-D makes many error checks as it processes source language statements. When an error is detected, the Assembler prints an error message. The format of the error messages is

ERROR CODE ADDRESS


where ERROR CODE is a two letter code which specifies the type of error, and ADDRESS 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.

The programmer should examine each error indication to determine whether correction is required.

4K PAL-D's error messages are listed and explained below.

<u>Error Code</u>	<u>Explanation</u>
BE	Two PAL-D internal tables have overlapped—This situation can usually be corrected by decreasing the level of literal nesting or a number of current page literals used prior to this point on the page.

<u>Error Code</u>	<u>Explanation</u>
DE	Systems device error—An error was detected when trying to read or write the system device, after three failures, control is returned to the Monitor.
DF	Systems device full—The capacity of the systems device has been exceeded; assembly is terminated and control is returned to the Monitor.
IC	Illegal Character—An illegal character was encountered other than in a comment or TEXT field; the character is ignored and the assembly continued.
ID	Illegal redefinition of a symbol—An attempt was made to give a previously defined symbol a new value by other means than the equal sign; the symbol was not redefined.
IE	<p>Illegal equals—An equal sign was used in the wrong context. Examples:</p> <pre>TAD A+=B A+B=C</pre> <p>The expression to the left of the equal sign is not a single symbol or, the expression to the right of the equal sign was not previously defined.</p>
II	<p>Illegal indirect—An off-page reference was made; a link could not be generated because the indirect bit was already set.</p> <p>Example:</p> <pre> *200 TAD I A . . . PAGE A, 7240</pre>
ND	The program terminator, \$, is missing (with TSS/8 only).

- PE Current non-zero page exceeded—An attempt was made to:
1. Override a literal with an instruction, or
 2. Override an instruction with a literal; this can be corrected by
 - (a) Decreasing the number of literals on the page, or
 - (b) Decreasing the number of instructions on the page.
- PH Phase error—PAL-D has received input files in an incorrect order; either program terminator (\$) is missing or misplaced. Assembly is terminated and control is returned to the Monitor.
- SE/ Symbol table exceeded—Assembly is terminated and control is returned to the Monitor; the symbol table may be expanded to contain up to 1184 user symbols by saving a file named .SYM on the system device.
- US Undefined symbol—A symbol has been processed during pass 2 that was not defined before the end of pass 1.
-  ZE Page 0 exceeded—Same as PE except with reference to page 0.

Chapter 14

8K Assemblers

CONTENTS

8K PAL-D Programming	14-5
Character Set	14-5
Pseudo-Operators	14-5
Loading and Operating Procedures	14-7
Saving 8K PAL-D	14-7
Third Pass Listing Option	14-7
Symbol Table	14-7
12K Version of 8K PAL-D	14-7
SABR Programming	14-9
Statements	14-9
The Character Set	14-11
Statement Elements	14-12
Labels	14-12
Operators	14-12
Operands	14-12
Comments	14-16
Incrementing Operands	14-17
Pseudo-operators	14-18
Assembly Control	14-19
Symbol Definition	14-23
Data Generating	14-25
External Subroutine Pseudo-operators	14-26
CALL and ARG	14-26
ENTRY and DUMMY	14-28
RETRN	14-29
Passing Subroutine Arguments	14-30
SABR Operating Characteristics	14-32
Pagewise Assembly	14-32
Multiple Word Instructions	14-34
Run-time Linkage Routines	14-34
Skip Instructions	14-37

Program Address	14-37
The Symbol Table	14-38
The Binary Output Tape	14-38
Loader Relocation Codes	14-39
Sample Assembly Listing	14-43
Loading and Operating SABR	14-46
Paper Tape System	14-46
Disk Monitor System	14-46
Assembly Procedure	14-47
Procedure for use as Fortran Pass 2	14-49
The Linking Loader	14-50
Operation	14-51
Linkage Routine Locations	14-52
Switch Register Options	14-52
Loading the Linking Loader	14-54
Loading Relocatable Programs	14-54
The Disk Linking Loader	14-56
Loading, Saving and Starting LLDR	14-57
Expanded Configuration	14-59
LLDR Functions	14-59
Disk File Assignment Function(D)	14-60
Loading Function (L and O)	14-62
Loading Errors	14-65
Utility Functions (C, M and U)	14-65
Exit Functions (E and S)	14-66
Overlay Loading	14-66
User Program Execution	14-70
Storage Allocation	14-70
Error Messages	14-71
The Subprogram Library	14-74
Input/Output	14-74
Floating Point Arithmetic	14-75
Integer Arithmetic	14-77
Subscripting	14-78
Functions	14-78
DECTape I/O Routines	14-80
Disk I/O Routines	14-82
ODISK and CDISK	14-82
RDISK and WDISK	14-83
Demonstration Program Using Library Routines	14-84

8K PAL-D PROGRAMMING

The 8K PAL-D Assembler is similar to 4K PAL-D. The reader is advised to learn the 4K PAL-D Assembler by studying the appropriate sections of Chapter 13, then return to this section to learn the additional features of 8K PAL-D. These additional features include assembler directives which permit operation of the Assembler to be controlled by the source program, page control, and the ability to expand to run in 12K of core.

CHARACTER SET

In addition to the characters allowed in 4K PAL-D, the following characters are given a special significance in 8K PAL-D: < >

The angle brackets (< >) define the bounds of a conditional statement. The user should be especially cautious not to use angle brackets within a comment in any program containing a conditional assembly statement.

PSEUDO-OPERATORS

In addition to the pseudo-operators allowed in 4K PAL-D, the following pseudo-operators are unique to 8K PAL-D:

RESERVING FREE STORAGE

ZBLOCK n

Where n is an integer, ZBLOCK causes the Assembler to reserve n words of memory containing zeros, starting at the word indicated by the current location counter.

CONDITIONAL ASSEMBLY

IFDEF symbol <statements>

If the symbol indicated is previously defined, assemble the statements contained in the angle brackets. If undefined, ignore these statements. Any number of statements can be contained in the angle brackets and may consist of several lines of code. The format of the IFDEF statement requires a single space before and after the symbol.

IFZERO expression <statements>

If the evaluated (arithmetic or logical) expression is equal to zero, assemble the statements contained within the angle brackets;

if the expression is non-zero, ignore these statements. Any number of statements can be contained in the angle brackets and may consist of several lines of code. The format of the IFZERO statement requires that the expression not contain any imbedded spaces and must have a single space preceding and following it.

BINARY OUTPUT CONTROL

NOPUNCH

Upon encountering this statement the Assembler continues to assemble the code, but ceases binary output.

ENPUNCH

This statement causes the Assembler to resume (or continue) binary output.

These two pseudo-operators are put into the source program and are ignored until pass 2 at which time they direct the Assembler to delete some section of code from the final binary punched tape.

For example, these pseudo-operators could be used where several programs have the same contents on page zero. When these programs are to be loaded and executed together, only one page zero need be punched.

PAGINATION OF OUTPUT LISTINGS

EJECT

The EJECT command causes the listing to jump to the top of the next page. The 8K PAL-D Assembler counts output lines and will format the user's program into neat, even pages with a page eject every 55 lines. If the user requires more frequent paging, he should use the EJECT pseudo-operator. A FORM FEED character within the source program will also cause a page eject.

The pagination process within the 8K PAL-D Assembler causes an output of carriage return/line feed pairs for the 33 ASR Teletype. For users with the 35 ASR Teletype who desire to output a FORM FEED character directly, changes should be made to modify the FORMI subroutine found in the 8K PAL-D listing.

LOADING AND OPERATING PROCEDURES

Saving 8K PAL-D

The 8K PAL-D Assembler is supplied on binary coded paper tape. It is loaded using the Binary Loader as explained in Appendix C2.

The 8K PAL-D Assembler may be saved on the system device as a system program. This is done by typing the following SAVE instruction:

```
.SAVE PAL8! 0-5177,6600-7577;200
```

The Assembler is now saved as a system program. The programmer may now type PAL8, which brings the assembler into core for use with symbolic source programs.

Third Pass Listing Option

Output devices are the same for 8K PAL-D as for 4K PAL-D. When 8K PAL-D requests the input file(s), the user may select up to ten (10) input files. Valid input devices for 8K PAL-D are as follows:

<u>Device Designator</u>	<u>Device</u>
T:	Teletype
R:	High-speed reader/punch
S: name	DF 32 disk
Sn: name	RF 08 disk
D0: name through D7: name	DECTape

Symbol Table

The symbol table for 8K PAL-D provides space for 896 (decimal) user defined symbols. When the SE (symbol table exceeded) error message occurs, assembly is terminated and control is returned to the Monitor. The user file .SYM is not used by 8K PAL-D.

12K VERSION OF 8K PAL-D

The 8K PAL-D Assembler must be reassembled to run in 12K of core. The 12K version has a larger symbol table, but assembles at a slower pace. The changes to be made are documented on page 1 of the 8K PAL-D listing.



SABR PROGRAMMING

SABR (Symbolic Assembler for Binary Relocatable programs) is an advanced, one-pass assembler producing relocatable binary code with automatically generated core page and memory field linkages. It supports an extensive list of pseudo-operators which provide, among other facilities, external subroutine calling with argument passing and conditional assembly.

SABR controls a library of subprograms, any of which can be assembled into user programs. In an optional second pass, SABR produces an octal/symbolic listing of assembled programs.

Relocatable programs assembled with SABR are loaded with the 8K Linking Loader. Both SABR and Linking Loader are incorporated in the 8K Fortran compiler (see Chapter 15). SABR functions as the second pass of the compiler, and linking loader is included as part of the Fortran Operating System.

With the exception of its pseudo-operators, SABR is similar, from a user's standpoint, to the PAL-III assembler which produces location dependent (non-relocatable) binary coding. *Introduction to Programming*—Volume 1 of this set—contains an elementary approach to assembly language programming in Chapters 1-5. Concepts presented in Chapter 13 of this volume are also pre-requisite to the use of SABR.

SABR can be run on any PDP-8/I, -8/L, -8, or -8/S computer (or on the PDP-5 if it has the extended memory control modification) with at least 8K of core storage and a Teletype. A high speed paper tape reader/punch is recommended.

Statements

SABR symbolic programs are written as a sequence of statements and are usually prepared on the Teletype, on line, with the aid of the Symbolic Editor program. SABR statements are virtually format free. Each statement is terminated by typing the RETURN key. (Editor automatically provides a line feed). Two or more statements can be typed on the same line using the semicolon as a separator.

A statement line is composed of one or all of the following elements: label, operator, operand and comment, separated by spaces or tabs (labels require a following comma). The types of elements in a statement are identified by the order of appearance in the line

and by the separating or delimiting character which follows or precedes the element.

Statements are written in the general form

label, operator operand /comment (preceded by slash)

SABR generates, one or possibly more, machine (binary) instructions or data words for each source statement.

An input line may be up to 72₁₀ characters long, including spaces and tabs. Any characters beyond this limit are ignored.

The RETURN key (CR/LF) is both a statement and a line terminator. The semicolon may be used to terminate an instruction without terminating a line. If, for example, the programmer wishes to write a sequence of instructions to rotate the contents of the accumulator (AC) and link (L) six places to the right, it might look like this:

```
...  
RTR  
RTR  
RTR  
...
```

But, with the semicolon, the programmer may place all three RTR's on a single line, separating each RTR with a semicolon and terminating the line with the RETURN key. The above sequence of instructions could then be written

RTR; RTR; RTR (terminated with the RETURN key)

This format is particularly useful when creating a list of data:

```
0200    0020    LIST,20;50;-30;62  
0201    0050  
0202    7750  
0203    0062
```

Null lines may also be used to format program listings. A null line is a line containing only a carriage return and possibly spaces or tabs. Such lines appear as blank lines in the program listing.

The Character Set

ALPHABETIC:

In addition to the letters A through Z, the following are considered by SABR to be alphabetic:

[left bracket
]	right bracket
\	back slash
↑	up arrow

NUMERIC:

0-9

SPECIAL CHARACTERS:

,	Comma	delimits a symbolic address label
/	Slash	indicates start of a comment
(Left parenthesis	indicates a literal (D indicates numeric literal is decimal; (K indicates numeric literal is octal
“	Quote	precedes an ANSCII constant
-	Minus sign	negates a constant
#	Number sign	increases value of preceding symbol by one
	RETURN (carriage return)	terminates a statement
;	Semicolon	terminates an instruction
↓	LINE FEED	ignored
	FORM FEED	ignored
	SPACE	separates and delimits items on the statement line
	TAB	same as space
	RUBOUT	ignored

All other characters are illegal except when used as ANSCII constants following a quote (“), or in comments or text strings.

Legal characters used in ways different from the above, and all illegal characters, cause the error message C (Illegal Character) to be printed by SABR.

STATEMENT ELEMENTS

Labels

A label is a symbolic name or location tag created by the programmer to identify the address of a statement in the program. Subsequent references to the statement can be made merely by referencing the label. If present, the label is written first in a statement and terminated with a comma.

```
0200      0000      SAVE,   0
0201      1200      ABC,    TAD     SAVE
```

SAVE and ABC are labels referencing the statements in location 0200 and 0201, respectively.

Operators

An operator may be one of the following:

- a. A direct or indirect memory reference instruction
- b. An operate or IOT microinstruction (Appendix B2 gives a summary of microinstructions)
- c. A pseudo-operator

All potential SABR operators are listed in Appendix A2.

Operands

An operand can be a user-defined address symbol, a literal or a numeric constant.

SYMBOLS

Symbols are composed of legal alphanumeric characters. There are two major type of symbols, permanent, and user-defined, and there are variations within each major type. A symbol is delimited by a non-alphanumeric character.

PERMANENT SYMBOLS. Permanent symbols are predefined and maintained in SABR's permanent symbol table. They include all of the basic instructions and pseudo-operators in Appendix A2. These symbols may be used without prior definition by the user. The OPDEF and SKPDF pseudo-operators are used to define instruction operators not included in the permanent symbol table.

USER-DEFINED SYMBOLS. A user-defined symbol is a string of from one to six legal alphanumeric characters delimited by a

non-alphanumeric character. User-defined symbols must conform to the following rules:

- a. The characters must be legal alphanumerics—
 ABCD . . . XYZ and 0123456789.
- b. The first character must be alphabetic.
- c. Only the first six characters are meaningful. A symbol such as INTEGER would be interpreted as INTEGE. Since the symbols GEORGE1 and GEORGE2 differ only in the seventh character, they would be treated as the same symbol: GEORGE.
- d. A user-defined symbol cannot be the same as any of the predefined permanent symbols.
- c. A user-defined symbol must be defined only once. Subsequent definitions will be ineffective and will cause SABR to type the error message M (Multiple Definition).

A symbol is defined when it appears as a symbolic address label or when it appears in an ABSYM, COMMN, OPDEF or SKPDF statement (see Pseudo-Operators). No more than 64 different user-defined symbols may occur on any one core page.

EQUIVALENT SYMBOLS. When an address label appears alone on a line—with no instruction or parameter—the label is assigned the value of the next address assembled.

```
TAG1,  
TAG2,   30  
TAG3,
```

TAG1 and TAG2 are equivalent symbols, in that they are assigned the same value. Therefore, a TAD TAG1 will reference the data at TAG2. TAG3, however, is not equivalent to TAG2. TAG3 would be defined as 1 greater than TAG2.

SYMBOL TABLE FLAGS

Symbols are listed in alphabetic order at the end of the assembly pass 1 with their relative addresses beside them. The following flags are added to denote special types of symbols:

- | | |
|-----|--|
| ABS | The address referenced by this symbol is absolute. |
| COM | The address is in COMMON. |

- OP The symbol is an operator.
- EXT The symbol is an external one and may or may not be defined within this program. If not defined there is no difficulty: it is defined in another program.
- UNDF The symbol is not an external symbol and has not been defined in the program. This is a programmer error. No earlier diagnostic can be given because it is not known that the symbol is undefined until the end of pass 1. A location is reserved for the undefined symbol, but nothing is placed in it.

LITERALS

The use of literals is a special and convenient way of generating constant data in a program. Literals are normally used by TAD and AND instructions, as in the following examples:

```

0200       0376     A,       AND       (777
0201       1375               TAD       (-50
0202       1374               TAD       ('C
.
.
.
0374       0303
0375       7730
0376       0777

```

A literal is always a numeric or ANSCII constant and must be preceded by a left parenthesis. The value of the literal will be assembled in a table near the end of the core page on which the instruction referencing it is assembled. The instruction itself will be assembled as an appropriate reference to the location where the numeric value of the literal is assembled. Literals may not be referenced indirectly.

The current numeric conversion mode can be changed on a purely local basis for a literal by inserting a D for decimal or a K for octal between the left parenthesis and the constant.

(D32 becomes 0040 (octal))
(K-32 becomes 7745 (octal))

This usage does not alter the prevailing permanent conversion

mode (initially octal but controllable with the DECIM and OCTAL pseudo-operators).

A literal may also be used as a parameter (i.e., with no operator). In this case the numeric value of the literal is assembled as usual in the literal table near the end of the core page currently being assembled, and a relocatable pointer to the address of the literal is assembled in the location where the literal parameter appeared.

```
0200      0376      01 A,      (20
      .
      .
0376      0020
```

This feature is intended primarily for use in passing external subroutine arguments with the ARG pseudo-operator.

CONSTANTS

Constants are of two types: numeric and ANSCII. ANSCII constants are used only as parameters. Numeric constants may be used as parameters or as operand addresses.

```
0200      1412      TAD I      12
```

Constant operand addresses are treated as absolute addresses, just as a symbol defined by an ABSYM statement. References to them are not generally relocatable. Therefore, they should be used only with great care. The primary use of constant operand addresses to reference locations on page 0 (See page 14-52 for free locations on page 0 of each field). All constant operand addresses are assumed to be in the field into which the program is loaded by the Linking Loader.

Constants may not be added to or subtracted from each other or from symbols.

NUMERIC CONSTANTS

A numeric constant consists of a single string of from one to four digits. It may be preceded by a minus sign (-) to negate the constant. The digit string will be interpreted as either octal or decimal according to the latest permanent mode setting by an OCTAL or DECIM pseudo-operator. Octal mode is assumed at

the beginning of assembly. The digits 8 and 9 must not appear in an octal string.

```

0200      5020      A,      5020
0201      7575
                                DECIM
0202      0120      80

```

ANSII CONSTANTS

Eight-bit ANSCII values may be created as constants by typing the ANSCII character immediately following a double quotation mark ("). A minus sign may be used to negate an alphabetic constant. The minus sign must precede the quotation mark.

```

0200      0273      A,      ";
0201      7477      -"A      /-301
0202      0207      "      /BELL FOLLOWS"

```

The following characters are illegal as alphabetic constants: carriage return, line feed, form feed and rubout.

PARAMETERS

An operand on a line with no operator is treated as a parameter. A parameter may be a numeric constant, a literal, or a user-defined address symbol.

```

0200      0200      ABC,      200;-320;      "M
0201      7460
0202      0315
0203      0176      POINTR,      PGOADR
                                REORC      1000
1000      0576      START      TAD I      POINTR
1001      1375      TAD      C3

```

Comments

A programmer may add notes to a statement following a slash mark. Such comments do not affect assembly or program execution but are useful in interpreting the program listing for later analysis and debugging. Entire lines of comments may be present in the program.

None of the special characters or symbols have significance when they appear in a comment.

```

/THIS IS A COMMENT LINE.
/THIS TOO. TAD;CALL;#"-2C+=!
A,      TAD      SAVE      /SLASH STARTS COMMENT

```

INCREMENTING OPERANDS

Because SABR is a one-pass assembler and also because it sometimes generates more than one machine instruction for a single user instruction, operand arithmetic is impossible. Statements of the form,

```

TAD      TAG+3
TAD      LIST-LIST2
JMP      .+6

```

are illegal. However, by appending a number sign to an operand the user can reference a location exactly one greater than the location of the operand (the next sequential location): TAD LOC# is equivalent to the PAL language statement TAD LOC+1.

```

0200      0020      LOC,      20
0201      0030
0202      1200      START,    TAD      LOC      /GET 20
0203      1201      TAD      LOC#     /GET 30
                                PAGE
0400      0200      A,      LOC
0401      0201      B,      LOC#

```

In assembling #-type references SABR does not attempt to determine if multiple machine code words are generated at the symbolic address referenced.

```

START,    TAD I    LOC      /LOC IS OFF-PAGE
          NOP      /USER HOPES TO MODIFY
          .
          .
          .
          TAD      (7500    /SMA
          DCA      START#

```

In the example above the user wishes to change the NOP instruction to an SMA. However, this is not possible because TAD I LOC will be assembled as three machine code words; if START

is at 0200, the NOP will be at 0203. The SMA will be inserted at 0201, thus destroying the second word of the TAD I LOC execution.

To avoid this error, the user should carefully examine the assembly listing before attempting to modify a program with #-type references. In the previous example the proper sequence is:

```
START,  TAD I   LOC
VAR,    NOP
      .
      .
      .
      TAD      (7500
      DCA      VAR
```

The #-sign feature is intended primarily for manipulating DUMMY variables when picking up arguments from external subroutines and returning from external subroutines (see Picking up Subroutine Arguments).

PSEUDO-OPERATORS

Table 14-1 lists all the pseudo-operators available in SABR, whether used as a free-standing assembler, or in conjunction with the Fortran compiler (see 8K Fortran, Chapter 15). The pseudo-operators are categorized and explained in the following paragraphs.

Table 14-1.
SABR Pseudo-Operators

Mnemonic	Code	Operation
ABYSM		Direct Absolute Symbol Definition
ARG		Argument for Subroutine Call
BLOCK		Reserve Storage Block
CALL		Call External Subroutine
COMMN		Common Storage Definition
CPAGE		Check if Page Will Hold Data
DECIM		Decimal Conversion
DUMMY		Dummy Argument Definition
EAP		Enter Automatic Paging Mode
END		End of Program
ENTRY		Define Program Entry Point
FORTR		Assemble FORTRAN Tape

Table 14-1. (Cont.)

Mnemonic	Code	Operation
IF		Conditional Assembly
LAP		Leave Automatic Paging
OCTAL		Octal Conversion
OPDEF		Define Non-Skip Operator
PAGE		Terminate the Page
PAUSE		Pause for Next Tape
REORG		Terminate Page and Reset Origin
RETRN		Return from External Subroutine
SKPDF		Define Skip-Type Operator
TEXT		Text String
		Floating-Point Accumulator
ACH	20*	high-order word
ACM	21*	middle word
ACL	22*	low-order word

* The floating point accumulator is in field 1.

Assembly Control

END Every program or subprogram to be assembled must contain the END pseudo-op as its last line. If this requirement is not met, an error message (E) is given.

PAUSE The PAUSE pseudo-op causes assembly to halt. It is designed to allow the user to break up large source tapes into several smaller segments. To do this, the user need only place a PAUSE statement at the end of each section of this source except the last. Then when assembly halts at a PAUSE, he may remove the source tape just read from the reader and insert the next one. Assembly may then be continued by pressing the console CONTInue switch.

WARNING

The PAUSE pseudo-op is designed specifically for use at the end of partial tapes and should not be used otherwise.

The reason for this is that the reader routine may have read data from the paper tape into its buffer that is actually beyond the PAUSE statement. Consequently, when CONTInue is pressed after the PAUSE is found by the line interpreting routine, the entire content of the reader buffer following the PAUSE is destroyed, and the next tape begins reading into a fresh buffer. Thus, if there is any meaningful data on the tape beyond the PAUSE statement, it will be lost.

DECIM

Initially the numeric conversion mode is set for octal conversion. However, if the user wishes, he may change it to decimal by use of the DECIM pseudo-op.

OCTAL

If the numeric conversion mode has been set to decimal, it may be changed back to octal by use of the OCTAL pseudo-op.

No matter which conversion mode has been permanently set, it may always be changed locally for literals by use of the (D or (K syntax described earlier.

```

0200      0320      START                320
                                                DECIM
0201      0500
0202      0377      01                    (K320
0203      1000
                                                512
                                                OCTAL
0204      0512
0205      0376      01                    (D512
0206      0320
                                                320
      .
      .
      .
0376      1000
0377      0320

```

LAP

The assembler is initially set for automatic generation of jumps to the next core page when the page being assembled fills up (Page Escapes), or when PAGE or REORG pseudo-ops are encountered. This feature may be suppressed by use of the LAP (Leave Automatic Paging) pseudo-op.

EAP If the user has previously suppressed the automatic paging feature, it may be restored to operation by use of the **EAP** (Enter Automatic Paging) pseudo-op.

PAGE The **PAGE** pseudo-op causes the current core page to be assembled as is. Assembly of succeeding instructions will begin on the next core page. No argument is required.

REORG The **REORG** pseudo-op is similar to the **PAGE** pseudo-op, except that a numerical argument specifying the relative location within the sub-program where assembly of succeeding instructions is to begin must be given. A **REORG** below 200 may not be given. A **REORG** should always be to the first address of a core page. If a **REORG** address is not the first address of a page, it will be converted to the first address of the page it is on.

0200	7200	START,	CLA	
			PAGE	
0400	7040		CMA	
			REORG	1000
1000	7041		CIA	

CPAGE The **CPAGE** pseudo-op followed by a numerical argument **N** specifies that the following **N** words of code* must be kept together in a single unit and not be split up by page escapes and literal tables. If the **N** words of code will not fit on the current page of code, the current page is assembled as if a **PAGE** pseudo-op had been encountered. The **N** words of code will then be assembled as a unit on the next core page.

NOTE

N must be less than or equal to 200 (octal) in nonautomatic paging mode or less than or equal to 176 octal in automatic paging mode.

* Normally data. However, if these **N** words are instructions, for example a **CALL** with arguments, it is the user's responsibility to count extra machine instructions which must be inserted by **SABR**.

```

START,  CLA
        LAP           /INHIBIT PAGE ESCAPE
        CPAGE 200     /CLOSES THE
        NAME1        /CURRENT PAGE
        NAME2        /& ASSEMBLES THE
                   /NEXT PAGE.

```

IF

The conditional pseudo-op, IF, is used with the following syntax:

```
IF      NAME,      7
```

The action of the pseudo-op, so given, is to first determine whether the symbol NAME has been previously defined. If NAME is defined, the pseudo-op has no effect. If NAME is not defined, the next seven symbolic instructions (not counting null lines and comment lines) will be treated as comments and not assembled.

```

/ABSYM NAME      176
IF NAME,        2           /THE NEXT LINE TO BE
                   CLL RTL   /ASSEMBLED WILL BE
                   RAL       /"DCA LOC"
/IF THE SLASH BEFORE "ABSYM NAME 176" IS
/REMOVED, THE "CLL RTL" AND "RAL" WILL
/BE ASSEMBLED.
                   DCA      LOC

```

Normally the symbol referenced by an IF statement should be either an undefined symbol or a symbol defined by an ABSYM statement. If this is done, the situation mentioned below cannot occur.

WARNING

In a situation such as the following, a special restriction applies.

```

NAME,      0
.
.
.
IF NAME,      3

```


The restriction is that if the line NAME, 0 happens to occur on the same core page of instructions as the IF statement, then, even though it is before the IF statement, NAME will not have been previously defined when the IF statement is encountered, and on the first pass (though not in the listing pass) the three lines after the IF statement will not be assembled. The reason for this is that location tags cannot be defined until the page on which they occur is assembled as a unit.

Symbol Definition

ABSYM An absolute core address may be named using the ABSYM pseudo-op. This address must be in the same core field as the subprogram in which it is defined. The most common use of this pseudo-op is to name page zero addresses not used by the operating system. These addresses are listed on page 14-52.

OPDEF
SKPDF Operation codes not already included in the symbol table may be defined by use of the OPDEF or SKPDF pseudo-ops. Non-skip instructions must be defined with the OPDEF pseudo-op and skip-type instructions must be defined with the SKPDF pseudo-op.

Examples of ABSYM, OPDEF and SKPDF syntax:

ABSYM	TEM	177	/PAGE ZERO ADDRESSES
ABSYM	AX	10	
OPDEF	DTRA	6761	/A NON-SKIP INSTR.
SKPDF	DTSF	6771	/SKIP-TYPE INSTRUCTIONS
SKPDF	SMZ	7540	

NOTE

ABSYM, OPDEF and SKPDF definitions must be made before they are used in the program.

COMMN

The COMMN pseudo-op is used to name locations in field 1 as externals so that they may be referenced by any program. If any COMMN statements are used, they must occur at the beginning of the source, before everything else including the ENTRY statement. COMMON storage is always in field 1 and is allocated from location 0200 upwards. Since the top page of field 1 is reserved, no more than 3840_{10} words of COMMON storage may be defined.

A COMMN statement normally takes a symbolic address label, since storage is being allocated. However, COMMON storage may be allocated without an address label.

A COMMN statement always takes a numerical argument which specifies how many words of COMMON storage to be allocated; however, a 0 argument is allowed. A COMMN statement with 0 argument allocates no COMMON storage; it merely defines the given location symbol at the next free COMMON location.

The syntax of the COMMN statement is shown below.

```
A,      COMMN    20
B,      COMMN    10
        COMMN    300
C,      COMMN     0
D,      COMMN    10
        ENTRY    SUBRUT
```

In this example 20 words of COMMON storage are allocated from 0200 to 0217, and A is defined at location 0200. Then, 10 words are allocated from 0220 to 0227, and B is defined at 0220. Notice that if A is actually a 30 word array, this example equates B(1) with A(21).

The example continues by allocating COMMON storage from 0230 to 0527 with no name being assigned to this block. Then 10 words are al-

located from 0530 to 0537 with both C and D being defined at 0530.

Data Generating

BLOCK The **BLOCK** pseudo-op given with a numerical argument N will reserve N words of core by placing zeros in them. This pseudo-op creates binary output, and thus may have a symbolic address label.

Before the N locations are reserved, a check is made to see if enough space is available for them on the current core page. If not, this page is assembled and the N locations are reserved on the next core page. The action here is similar to that of the **CPAGE** pseudo-op. Similar restrictions on the argument apply.

```
/EXAMPLE OF HOW LARGE BLOCK STORAGE  
/MAY BE ACHIEVED WITHIN A SUBPROGRAM AREA
```

```
LAP                /INHIBIT PAGE ESCAPES  
BLOCK 200         /RESERVE 500  
BLOCK 200         /((OCTAL) LOCATIONS  
BLOCK 100  
EAP              /RESUME NORMAL CODING
```

As a special use, if the **BLOCK** pseudo-op is used with a location tag (but with no argument or a zero argument), no code zeros are assembled; instead the symbolic address label is made equivalent to the next relative core location assembled. (This is equivalent to using a symbolic address label with no instruction on the same line.)

```
LIST,  BLOCK 3    /ASSEMBLES AS  
                /3 ZEROS WITH  
                /"LIST" DEFINED  
                /AT THE 1ST LOCATION  
NAME1,  BLOCK    /DEFINES NAME1=  
NAME2,  BLOCK 0  /NAME2=NAME3=  
NAME3,  BLOCK    /NAME4  
NAME4,  BLOCK 2
```

TEXT

The TEXT pseudo-op is used to obtain packed six-bit ANSCII text strings. Its function and use are almost exactly the same as for the BLOCK pseudo-op except that instead of a numerical argument, the argument is a text string. In particular, a check is made to be sure that the text string will fit on the current page without being interrupted by literals, etc.

The text string argument must be contained on the same line as the TEXT pseudo-op. Any printing character may be used to delineate the text string. This character must appear at both the beginning and the end of the string. Carriage return, line feed and form feed are illegal characters within a text string (or as delineators). All characters in the string are stored in simple stripped six-bit form. Thus, a tab character (ANSSCII 211) will be stored as an 11, which is equivalent to the coding for the letter I. In general, characters outside the ANSCII range of 240-337 should not be used.

0200	2405	TAG,	TEXT	/TEXT EXAMPLE 123*??/
0201	3024			
0202	4005			
0203	3001			
0204	1520			
0205	1405			
0206	4061			
0207	6263			
0210	5273			
0211	7700			

EXTERNAL SUBROUTINE PSEUDO-OPERATORS

SABR and the linking Loader utilize CALL, ARG, ENTRY, DUMMY and RETURN pseudo-ops for calling external subprograms, passing arguments among them and returning from them. COMMON storage may also be utilized for this purpose.

CALL and ARG

A CALL statement consists of the CALL pseudo-operator followed by two indicators: the first, a one or two-digit number (64_{10}

maximum) indicating the number of arguments to be passed to the subroutine; the second, separated from the first by a comma, is the symbolic name of the subroutine entry point.

ARG statements consist of an ARG pseudo-operator symbol followed by one of the arguments to be passed. One ARG statement must be coded for each argument. ARG is used only in conjunction with CALL.

In the following example the main program (or a subroutine) calls a subroutine named SUBR. and supplies two arguments.

```

.
.
TAG,    CALL    2, SUBR
N1,    ARG     (50
N2,    ARG     LOCATN
ETC,    ...

```

The above instructions are assembled as follows:

```

CPAGE 6                                /MAKE SURE THE FOLLOWING
                                        /2N+2 WORDS WILL FIT
                                        /ON THE CURRENT CORE PAGE.
TAG,    JMS     LINK                                /CALL THE CALL LINKAGE ROUT
        020X   (06)                                /WHERE 2=THE NUMBER OF
                                        /ARGUMENTS AND X=THE
                                        /LOCAL NUMBER OF THE
                                        /SUBPROGRAM BEING CALLED
                                        /VIZ., SUBR.
N1,    CDF CUR(05)                                /FIELD ADDRESS OF ARGUMENT
        POINTER                                /IN FORM OF CDF INSTRUCTION
                                        /ADDRESS IN THE LITERAL
                                        /TABLE WHERE THE 50 IS
                                        /ASSEMBLED.
N2,    CDF CUR OR CDF10 /FIELD OF THE ARGUMENT
        LOCATN                                /DEPENDING ON WHETHER IT
                                        /IS OR IS NOT IN COMMON.
                                        /ADDRESS OF ARGUMENT

```

When a subprogram is referenced in a CALL statement, the Run-Time Linkage Routine LINK executes the transfer to the subprogram. It assumes that the entry point to the program is a two-word block. Into the first word of this block it places the number of the field where the CALL to the subprogram occurred.

In the second word it places the address where the call occurred, plus 2.

In the example above, SUBR would receive a 62M1 where TAG is in field M, and SUBR# would receive the address of N1. If there were no arguments, SUBR# would receive the address of ETC. Thus, the two-word block at the ENTRY point serves as storage for the 15-bit address vector for picking up arguments and also for returning from the subprogram.

Execution of the subprogram begins at the first location following the two-word entry block.

When the ARG pseudo-op is used with a literal, as in the above examples, the actual literal (50 in this case) is generated in the literal table and a relocatable pointer to the literal is generated in the location following the CDF CUR (see Loader Relocation Codes). This is the same as using the literal as a parameter.

If the ARG statement contains a true constant parameter, the constant itself is assembled in the location following the CDF instruction: in this case the CDF is useless.

The advantage of using the ARG-literal method is that it allows a subroutine to pick up an argument which is sometimes a variable and sometimes a constant.

ENTRY and DUMMY

The ENTRY pseudo-operator is used at the beginning of a subprogram to establish the name of its entry point and define it as external for the Linking Loader.

The ENTRY statement must occur before the symbolic name of the entry point appears as a symbolic address label. The actual entry location must be a two-word reserved space so that both the return address and field can be saved when the routine is called.

```
ENTRY   SUBROU
SUBROU, BLOCK 2
        CLA
```

An entry point name in an ENTRY statement acquires all of the properties of a DUMMY variable.

A DUMMY variable is a special type in the SABR/FORTRAN system. It must be defined in the subprogram which references it. When referenced directly, a DUMMY variable is treated as a local

symbol. However, when referenced indirectly it causes a call to the DUMMY Variable run-time linkage routine (see Run-Time Linkage Routines). This routine assumes that the DUMMY variable is a two-word reserved space where the first word is a 62N1 (with N the field of the address to be referenced) and the second word contains the 12-bit address. DUMMY variables are used in passing arguments to and from subroutines.

```
ENTRY    A1
DUMMY    X
DUMMY    Y
A1,      BLOCK 2
.
.
.
X,       M
Y,       N
```

RETRN

The RETRN statement is used to return from a subprogram to the calling program. The name of the subprogram being returned from must be specified so that the Return Linkage Routine can determine the action required, and because a subprogram may have differently-named entry points. It is possible for the careful user to return to the location following the last call of any subprogram merely by specifying it in a RETRN statement.

```
TAG,     RETRN  SUBROU
```

Before the return statement is used, however, the pointer in the second word of the subprogram entry must be incremented to the point following all arguments in the CALL statement.

EXAMPLE:

A user wishes to write a long main program, MAIN, which uses two major subroutines, S1 and S2. S1 requires two arguments and S2 one argument. The user would then write MAIN, S1 and S2 as three separate programs in the following manner:

```

MAIN,   CLA      /START OF MAIN
      .
      .
      CALL      2,S1
      ARG       X
      ARG       Y
      CALL      1,S2
      ARG       Z
      .
      .
      END
ENTRY  S1
S1,    BLOCK 2
      .
      .
      .
      RETRN S1
      END
ENTRY  S2
S2,    BLOCK 2

      RETRN S2
      END

```

S1 could contain calls to S2, or S2 calls to S1. In addition, the subprograms could make use of dummy variables.

The user then assembles each of these subprograms with SABR and loads all of them with the Linking Loader. During the loading process, all of the proper addresses will be saved in tables so that when the user begins execution of MAIN, the Run-Time Linkage Routines (see under SABR Operating Characteristics) which were automatically loaded, will be able to execute the proper reference. Thus, MAIN will be able to fully use S1 and S2 and be able to pass data to and receive it from them.

Passing Subroutine Arguments

The following example shows how SUBR would pick up the arguments 50 and LOCATN and deposit them in LOC1 and LOC2.


```

/MAIN      PROGRAM
MAIN,     CLA
          .
          .
          .
TAG,      CALL      2, SUBR
N1,       ARG       (50
N2,       ARG       LOCATN
ETC,      ...
          END

/SUBROUTINE
ENTRY    SUBR
DUMMY    TEM
SUBR,    BLOCK      2
TAD I    SUBR      /THIS GIVES THE FIELD ADD-
          /RESS (CDF CUR) OF THE (50
          /I.E. THE CONTENTS OF N1
DCA      TEM      /TO FIRST WORD OF DUMMY
INC      SUBR#    /MOVE ARG POINTER TO N1#
TAD I    SUBR      /GET ADDRESS OF (50
          /I.E. CONTENTS OF N1#
DCA      TEM#     /TO 2ND WORD OF DUMMY
TAD I    TEM      /PICK UP THE (50
DCA      LOC1
INC      SUBR#    /MOVE ARG POINTER TO N2
          .
          .
          .
/SIMILAR METHOD TO PICK UP CONTENTS
/OF LOCATN
          .
          .
          .
INC      SUBR#    /MOVE PTR FOR RETURN
          /AT ETC
CLA
RETRN    SUBR
TEM,     BLOCK      2

```

Constant arguments are specified as literals because the sub-program may not know that a constant argument is being used. Hence, specifying constant arguments as literals will ensure that the second word of every assembled argument is actually the address of the argument.

The ARG statement may be used with a constant (e.g., if a constant address is intended). The following technique may be used if SUBR can assume that the first argument is always a constant:

```

/MMAIN PROGRAM
TAG,      CALL      2, SUBR
N1,       ARG       50
N2,       ARG       LOCATN
ETC,
.
.
.
      END

/SUBROUTINE
      ENTRY      SUBR
      DUMMY     TEM
SUBR,     BLOCK 2
      INC       SUBR#  /MOVE ARG PTR TON1#
      TAD I     SUBR   /THIS GETS THE 50
                          /IMMEDIATELY
      DCA       LOC1
      INC       SUBR#  /GET C(LOCATN) IN
                          /THE USUAL WAY

```

To summarize: an advanced technique for picking up subprogram arguments is provided because subroutine arguments are two-word addresses and the calling program and the subprogram may reside in different fields.

A subprogram entry point is assumed to have been defined as a two-word reserved block and defined in an ENTRY statement. The appearance of the subprogram name in an ENTRY statement gives the two-word block the properties of a dummy variable. This means that when the subprogram name is referenced indirectly a call is generated to the Dummy Variable Run-time Linkage Routine where the details of locating and picking up the argument address words are worked out. Thus, the user, need only use the number sign (#) feature to increment the argument pointer in the second word of the entry point.

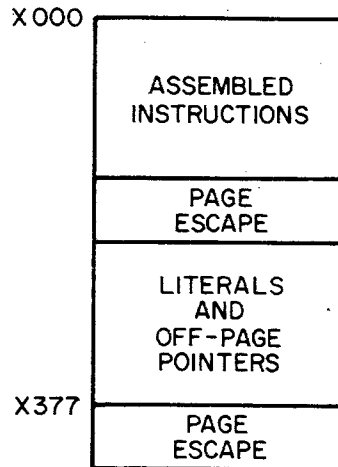
SABR OPERATING CHARACTERISTICS

Pagewise Assembly

SABR assembles page-by-page rather than one instruction at a time. To accomplish this it builds various tables as instructions are read. When a full page of instructions has been collected (counting literals, off-page pointers and multiple word instructions) the page is assembled and punched. Several pseudo-operators can be used to control page assembly.

PAGE FORMAT

A normal assembled page of code is formatted as below:



Literals and off-page pointers are intermingled in the table at the end of the page.

PAGE ESCAPES

SABR is normally in automatic paging mode: it connects each assembled core page to the next by an appropriate jump. This is called a page escape. For the last page of code, SABR leaves the Automatic Paging Mode and issues no page escape. The LAP (Leave Automatic Paging) pseudo-operator turns off the automatic paging mode. EAP (Enter Automatic Paging) turns it back on if it has been turned off.

Two types of page escape may be generated depending on whether or not the last instruction is a skip. If the last instruction is not a skip, the page escape is as follows:

last instruction (non-skip)
5377 (JMP to x177)
literals
and
off-page
pointers
x177/NOP

If the last instruction on the page is a skip type, the page escape takes four words, as follows:

last instruction (a skip)
5376 (JMP to x176)
5377 (JMP to x177)
literals
etc.
x176/SKP
x177/SKP

Multiple Word Instructions

Certain instructions in the source program require SABR to assemble more than one machine language instruction (e.g., off-page indirect references and indirect references where a data field re-setting may be required). In the listing, the source instruction will appear beside the first of the assembled binary words.

A difficulty arises when a multiple word instruction follows a skip instruction. In such a case, extra instructions must be assembled to enable the skip to be effected correctly.

Run-time Linkage Routines

These routines are loaded by the Linking Loader and perform their tasks automatically when certain pseudo-ops or coding sequences are encountered in the user program. The user needs knowledge of them only to better understand the program listing.

There are seven linkage routines:

- | | |
|---|--------|
| a. Change data field to current and skip | CDFSKP |
| b. Change data field to 1 (COMMON) and skip | CDZSKP |
| c. Off page indirect reference linkage | OPISUB |
| d. Off bank (COMMON) indirect reference linkage | OBISUB |
| e. Dummy variable indirect reference linkage | DUMSUB |
| f. Subroutine call linkage | LINK |
| g. Subroutine return linkage | RTN |

The individual linkage routines function as follows:

- CDFSKP is called when a direct off-page memory reference follows a skip-type instruction requiring the data field to be reset to the current field.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
SZA	7440	
DCA LOC	4045	call CDFSKP
	7410	SKP in case AC = 0 at .-2
	3776	execute the DCA via a pointer near the end of the page.

b. CDZSKP is called when a direct memory reference is made to a location in COMMON (which is always in Field 1). The action of CDZSKP is the same as that of CDFSKP except that it always executes a CDF 10 instead of a CDF current.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
SZA	7440	
DCA CLOC	4051	call CDZSKP
	7410	SKP in case AC = 0 at .-2
	3776	execute the DCA via a pointer near the end of the page.

c. OPISUB is called when there is an indirect reference to an off page location.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
DCA I PTR	4062	call OPISUB
	0300 01	relative address of PTR
	3407	execute the DCA 1 via 0007

d. OBISUB is called when there is an indirect reference to a location in COMMON. In such a case it is assumed that the location in COMMON which is being indirectly referenced points to some location that is also in COMMON.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
DCA I CPTR	4055	call OBISUB
	1000	address of CPTR in Field 1
	3407	execute the DCA 1 via 0007

e. DUMSUB is called when there is an indirect reference to a DUMMY variable. In such a case, DUMSUB assumes that the DUMMY variable is a two-word vector in which the first word is a 62N1, where N = the field of the address to be referenced, and the second word is the actual address to be referenced.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
DCA 1 DMVR	4067	call DUMSUB
	0300 01	relative address of DMVR
	3407	execute DCA 1 via pointer in location 0007

f. LINK is called to execute the linkage required by a CALL statement in the user's program. When a CALL statement is used, it is assumed that the entry point of the subprogram is named in the CALL and that this entry point is a two-word, free block followed by the executable code of the subprogram. LINK leaves the return address for the CALL in these two words in the same format as a DUMMY variable.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
CALL 2, SUBR	4033	call LINK
	0205 06	code word
ARG X	62M1	X resides in field M
	0300 01	relative address of X
ARG C	6211	C is in COMMON
	1007	absolute address of C

g. RTN is called to execute the linkage by a RETRN statement in the user's program.

<u>Program</u>	<u>Assembled Code</u>	<u>Meaning</u>
RETRN SUBR	4040	call RTN
	0005 06	number of the subprogram being returned from (SUBR)

Skip Instructions

In page escapes and multiple word instructions, skip-type instructions must be distinguished from non-skipping instructions. For this reason, a special pseudo-operator, SKPDF, must be used to define skip instructions not in the permanent symbol table, should these be used in source programs.

This also explains why both ISZ and INC are included in the permanent symbol table. ISZ is considered to be a skip instruction and INC is not. INC should be used to conserve space when the programmer desires to increment a memory word without the possibility of a skip.

The first example below shows the code which is assembled for an indirect reference to an off-page location following an INC instruction. The second example shows the same code following an ISZ instruction.

EXAMPLE 1:

```
INC     POINTR  0020  2376
TAD I   LOC2   0221  4062
          0222  0520 01 } /OFF PAGE INDIRECT EXECUTION
          0223  1407
```

EXAMPLE 2:

```
ISZ     COUNTR 0220  2376
TAD I   LOC2   0221  7410    /SKIP TO EXECUTION
          0222  5226    /JUMP OVER EXECUTION
          0223  4062
          0224  0520 01 } /OFF PAGE INDIRECT EXECUTION
          0225  1407
```

Program Addresses

Since each assembly is relocatable, the addresses specified by SABR always begin at 0200, and all other addresses are relative to this address. At loading time, the Linking Loader will properly adjust all addresses. For example, if 0200 and 1000 are the relative addresses of A and B, respectively, and if A is loaded at 2000, the B will be loaded at 1000 + 1600, or 2600.

All programs to be assembled by SABR must be arranged to fit into one field of memory, not counting page 0 of the field, or the

top page (7600 – 7777). If a program is too large to fit into one field, it should be split into several subprograms.

Explicit CDF or CIF instructions are not needed by SABR programs because of the availability of external subroutine calling and common storage. Explicit CDF or CIF instructions cannot be assembled properly.

The Symbol Table

Entries in the symbol table are variable in length. A one or two-character symbol requires three symbol table words. A three- or four-character symbol requires four words, and a five- or six-character symbol, five words. Thus, for long programs it may be to the user's advantage to use short symbols whenever possible.

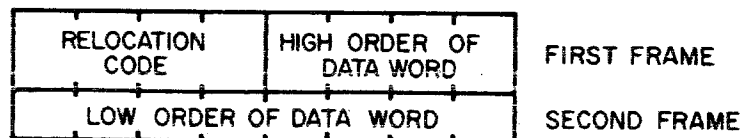
The symbol table, not counting permanent symbols, contains 2644_{10} words of storage. However, this space must be shared when there are unresolved forward and external references temporarily stored as two-word entries.

If we may assume that a program being assembled never has more than 100_{10} of these unresolved references at any one time, this leaves 2464_{10} words of storage for symbols. Using an average of four words per symbol, this allows room for 616_{10} symbols.

Symbol table overflow is a fatal condition which generates the error message S.

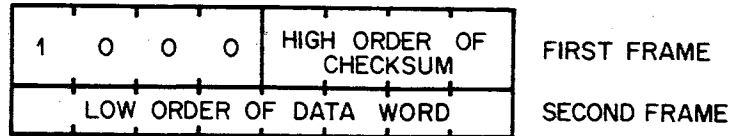
THE BINARY OUTPUT TAPE

SABR outputs each machine instruction on binary output tape as a 16-bit word contained in two 8-bit frames of paper tape. The first four bits contain the relocation code used by the Linking Loader to determine how to load the data word. The last 12 bits contain the data word itself.



The assembled binary tape is preceded and followed by leader/trailer code (code 200). The checksum is contained in the last two

frames of tape before the trailer code. It appears as a normal 16-bit word, as shown below.



All assembled programs have a relative origin of 0200.

Loader Relocation Codes

The four-bit relocation codes issued by SABR for use by the Linking Loader are explained below. The codes are given in octal.

00 Absolute Load the data word at the current loading address. No change is required.

```
0205    5277    JMP LOC /WHERE LOC IS
0242    7500    SMA     /AT 0277 (ON
0356    0020    20     /PAGE)
                       /A CONSTANT
```

01 Simple Relocation Add the relocation constant to the word before loading it. (The relocation constant is 200 less than the actual address where the first word of the program is loaded.) Items with this code are always program addresses.

```
0376    0520    01    A,    LOC2
```

In the above example, LOC2 is at relative address 0520. If the first word of the program (relative address 0200) is loaded at 1000, then the actual address of A is 1176 and location 1176 will be loaded with the value 1320, which will

be the actual address of LOC2 when loaded.

03 External
Symbol
Definition*

The data word is the relative address of an entry point. Before entering this definition in the Linkage Tables so that the symbol may be referenced by other programs at run-time, the Linking Loader must add the relocation constant to it. The six frames of paper tape following the two-frame definition are the ANSCII code for the symbol.

03	ADDRESS
ADDRESS LOW ORDER	
L	
O	
C	
2	
SPACE	
SPACE	

04 Reorgan*

Change the current loading address to the value specified by the data word plus the relocation constant.

05 CDF
Current

The data word is always a 6201 (CDF) instruction which has been generated automatically by SABR. The code 05 indicates to the Linking Loader that the number of the field currently being loaded into must be inserted in bits 6-8 before loading.

```

0300      6201 05 A, TAD LOC2
0301      1776          /WHERE LOC2 IS OFF PAGE SO THAT
                        /THE TAD INSTRUCTION MUST BE
                        /INDIRECT.

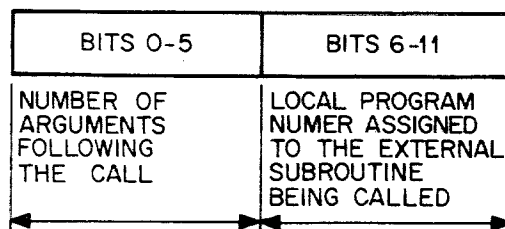
0376      0520      01
  
```

* Does not appear in assembly listings.

06 Subroutine
Linkage
Code

If the program containing this code is being loaded into field 4, relative location 0300 will be loaded with 6241.

Such an instruction is referred to in this document as CDF Current. They are generated automatically by SABR when a direct reference instruction must be assembled as an indirect, and there is the possibility that the current data field setting is different from the field where the indirect reference occurs. The data word is a special constant enabling the Linking Loader to perform the necessary linking for an external subroutine call. (c.f., CALL Pseudo-op). The structure of the data word is shown below.



Before the 12-bit, two-part code word is loaded into memory, a global external number will be substituted for the local external sym-

bol number in the right half of the data word.

```
0200      4033      CALL 3, SUB
0201      0307 06
           ARG X
           ARG Y
           ARG Z
```

Here, SUB has been assigned the local number 07 during assembly. At loading time this number will be changed to the global number (for example, 23), which is assigned to SUB. In this example, 0323 would actually be loaded at relative address 0201.

- | | | |
|----|------------------------------------|---|
| 10 | Leader/Trailer*
and
Checksum | This code represents normal leader/trailer. At the first occurrence of this code following the assembled program, the computer word contains the checksum. |
| 12 | High Common* | The data word is the highest location in Field 1 assigned to COMMON storage by the program. This item will occur exactly once in every binary tape and it must be the first word after the leader. If no COMMON storage has been allocated in the program, the data will be 0177. |
| 17 | Transfer*
Vector | Signifies that reference to an external symbol occurs in the assembled program. The 12-bit data word is meaningless. The next six frames contain the ANSCII code for the symbol. |

* Does not appear in assembly listings.

The Linking Loader uses this definition to create a transfer table, whereby local external symbol numbers assigned during assembly of this particular program can be changed to the global external symbol number when several programs are being loaded.

SAMPLE ASSEMBLY LISTING

This program is offered only to illustrate many of the features and formats of a SABR program. The program cannot be run (see also, Demonstration Program Using Library Routines).

```
PDP-8 SABR DEC-08-A2B2
HIGH SPEED READER? Y
HIGH SPEED PUNCH? Y
LISTING ON HIGH SPEED PUNCH? N
```

```
.DTCA      67620P
DTSF      67710P
LOC       0000UNDF
MUL       0000EXT
NAME      1000COM
POINTR    1013
SUB       0200EXT
S1        0202
S12       0214
S2        0214
S3        0227
S4        0233
TAG       0177ABS
X         0400
Y         0401
Z         0402
```

/SAMPLE OF SABR CODE

```
6762      OPDEF   DTCA      6762
6771      SKPDF   DTSF      6771
          /ABSYM  LOC       176
0177      ABSYM   TAG       177
          DECIM
200       NAME,  COMMN     8
          ENTRY  SUB
          DUMMY  X
          LAP
```

0200	0000	SUB,	BLOCK	2	
0201	0000		EAP		
			OCTAL		
0202	0000	S+,	0		
0203	4067		TAD I	SUB	
0204	0200 01				
0205	1407				
0206	7106		CLL RTL;		RTL
0207	7006				
0210	6211		DCA	NAME#	
0211	3776				
0212	6201 05		INC	POINTR	
0213	2775				
		S:2,			
0214	4033	S2,	CALL	3,MUL	
0215	0302 06				
0216	6201 05		ARG	X	
0217	0400 01				
0220	6201 05		ARG	(20	
0221	0374 01				
0222	6201		ARG	-1	
0223	7777				
0224	1373		TAD	(D-49	
			IF	LOC,1	
			PAUSE		
0225	1372		TAD	(-"?	
0226	5200		JMP	SUB	
			CPAGE	4	
0227	4233	S3,	JMS	S4	
0230	0004		4		
0231	0200		NAME		
0232	0371 01		(37		
0233	6762	S4,	DTCA		
0234	5377				
0371	0037				
0372	7501				
0373	7717				
0374	0020				
0375	1013 01				
0376	1001				
0377	7000				
			PAGE		
0400	0000	X,	0		
0401	0214 01	Y,	S+2		
0402	2301	Z,	TEXT	"SAMP@=*/?456"	
0403	1520				
0404	0075				
0405	4052				
0406	5777				
0407	6465				
0410	6600				
0411	6771		DTSF		

0412	5376		
0413	5377		
0576	7410		
0577	7410		
1000	7410	REORG	1000
1001	7410	SKP	
1002	5206	TAD I	S+2
1003	4062		
1004	0214 01		
1005	1407		
1006	1377	TAD	(333
1007	6211	DCA	NAME
1010	3776		
1011	4040	RETRN	SUB
1012	0001 06		
1013	0000	POINTR, 0	
1176	1000		
1177	0333		
		END	

For a multiple word instruction the actual instruction line is typed beside the first instruction.

0650	6201 05 LOC2,	JMP	NAME	/OFF PAGE
0651	5774			
0652	7106	CLL	RTL;RTL;RTL	
0653	7006			
0654	7006			

For an erroneous instruction, the error flag appears in the address field. The instruction is not assembled.

0700	7200	N2,	CLA
	I		CLL SKP
0701	7402		HLT

The page escape and literal and off-page pointer table are typed with nothing except the correct address, value and loader code.

0770	7006	N3,	RTL
0771	7500		SMA
0772	5376		
0773	5377		
0774	0200 01		
0775	0020		
0776	7410		
0777	7410		

LOADING AND OPERATING SABR

Procedures for loading SABR and assembling a source program are given below. See Appendix C2 for instructions for use of the Binary Loader.

Loading in a Paper Tape System

1. Make sure the Binary Loader is in memory, say in field n.
2. Set the console switches as follows:
Instruction Field = n, Switch Register = 7777.
3. Press LOAD ADDRESS.
4. Insert the SABR binary tape into the reader.
5. If using the high-speed reader, depress Switch Register Bit 0.
6. Press START.
7. SABR will now be loaded into memory by the Binary Loader; portions of SABR will load into field 0 and field 1.

Loading in a Disk Monitor System

1. Make sure the Disk Monitor is in memory. (Type CTRL/C† or START at 07600.)
2. When the Monitor responds with a dot, call the system Loader as follows:

.LOAD (followed by the RETURN key)

† CTRL/C and CTRL/P are typed by holding down the CTRL key while typing the C or P key.

3. Insert the SABR binary tape in the reader.
4. Answer the loading command dialogue as follows:
 *IN-R: for high speed reader or *IN-T: for Teletype.
 *
 *ST =
 ↑ < CTRL/P > ↑ < CTRL/P >
5. SABR is now loaded into memory, partly in field 0 and partly in field 1. It may be saved on the user's system device by responding to the monitor's dot as follows:
 - SAVE SABR! 0-7177;200
 - SAVE SAB1! 700,1700-12427
 -
6. SABR is now saved on the user's system device and may be called as follows:
 - SAB1
 - SABR
 The field 1 portion must be called first.

Assembly Procedure

It is assumed that the programmer has written his program in SABR language and punched this source program on paper tape in ANSCII code. The source tape may have been split into several separate tapes by placing a PAUSE statement at the end of each section except the last. The last tape must have an END statement at the end.

After SABR has been loaded into memory, it is used to assemble the source program. In Pass 1 the relocatable binary version of the user's program is created and, at the end of this pass, the symbol table is either typed or punched, according to whether this listing is to be typed or punched. Pass 2 is the listing pass. The assembly is carried out as follows.

NOTE

If SABR has been saved on the System I/O device, it will start automatically at Step 3 below when called into memory. The source tape (first section) should be inserted in the reader before operation begins.

1. Set the console switches as follows:

Data field = 0, Instruction Field = 0,
Switch Register = 0200.

2. Press LOAD ADDRESS and START.
3. SABR now types a sequence of two or three questions;

HIGH SPEED READER?
HIGH SPEED PUNCH?
LISTING ON HIGH SPEED PUNCH?

These questions must be answered with Y if the answer is yes. Any other answer is assumed to be no. The third question is typed only if the second is answered Y. If the third is answered Y, both the symbol table and the listing are punched on the high-speed paper tape punch. Otherwise, they are typed on the teletypewriter. The user need not wait for the full question to be typed before responding.

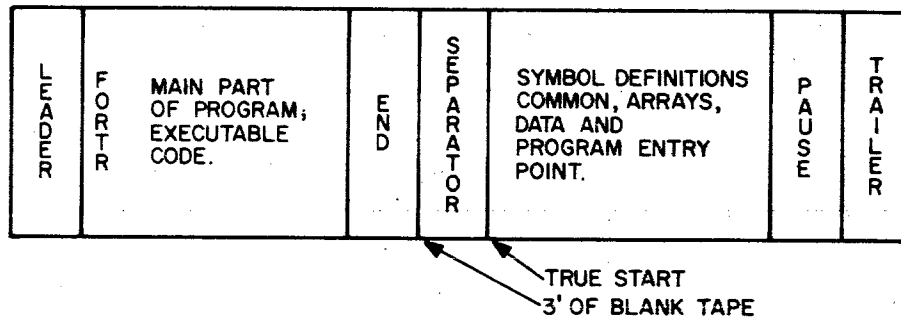
4. As soon as SABR has echoed the user's response to the last question, turn on the punch device and, if it is being used, the ASR reader. If the low-speed reader is used, the error message E indicates that the user has waited too long before turning the reader on. The user must begin again.
5. At this point, Pass 1 begins. SABR reads the source tape and punches the binary tape. After the binary tape has been completed, SABR types or punches the program symbol table.
6. If the source tape is in several sections (separate tapes with PAUSES at the end of all except the last), SABR halts at the end of each section. At this point, insert the next section in the reader and then press CONTINUE.
7. At the end of Pass 1, SABR halts.
8. If an assembly listing is desired, reposition the beginning of the source tape in the reader and if using the ASR reader, set it to START, and then press CONTINUE.
9. At the end of Pass 2, SABR again halts. To restart SABR for assembling another program, press CONTINUE.
10. To restart SABR at any time, press STOP, set the Switch Register = 0200, press LOAD ADDRESS and START. However, the first pass must always be repeated.

11. After assembling in a Disk Monitor environment, control may be returned to Monitor by restarting at location 7600.

Procedure For Use as Fortran Pass 2

In addition to its status as a stand-alone assembler, SABR serves as pass 2 of the 8K Fortran compiler. For this purpose, SABR procedures differ slightly. The Fortran compiler, in one pass, converts the user's Fortran source program into a symbolic source program containing standard PDP-8 mnemonics. SABR then converts the symbolic tape into a relocatable, binary-coded program.

The symbolic tape produced by the Fortran compiler is not in standard Format; it is arranged as shown below.



The tape is arranged this way because the data at the end of the tape cannot be inserted in the midst of the executable code, and some data which should be at the beginning of the tape is not known until later. Thus, the true start of the symbolic program is near the end of the symbolic tape, preceded by a segment of blank tape code and followed by a PAUSE statement.

To assemble such a tape with SABR, one of three methods must be followed. Actually, the general procedure is the same as that described in Assembly Procedure, differing in special details. The differences are covered by the three methods below.

METHOD 1

The simplest method is to cut the symbolic tape into two parts. The cut should be made at the middle of the blank tape which separates the executable code from the symbol definitions. The latter section of the tape should then be marked "Section 1" and the former section (the executable code) should be marked "Section

2." Assembly then proceeds with the two-part symbolic tape as previously described.

METHOD 2

The user may avoid actually cutting the symbolic tape by manipulating the tape as if it were in two parts as explained above. The tape should initially be inserted in the reader with the separator blank tape over the read-head. When SABR halts at the PAUSE statement at the physical end of the tape, the user should reposition the tape, putting the physical beginning of the tape in the reader. Then press CONTInue. The assembly pass will end at the separator blank tape code. The assembly listing can be produced in a similar manner, pressing CONTInue to start the listing pass.

METHOD 3

The third method requires SABR to pass over the symbolic tape two times for each pass of the assembly. However, it allows the tape to be inserted at its physical beginning. It is based on the fact that a symbolic tape output by the FORTRAN Compiler has as its physical first line the special pseudo-op, FORTR. This pseudo-op has no effect except when a symbolic tape output by the Compiler is assembled using this third method.

1. Insert the symbolic tape in the reader at its physical beginning.
2. Start SABR as usual.
3. Sensing the FORTR statement as the first line, SABR ignores all further data until after it passes over the END statement. SABR then begins the actual assembly by processing the symbol definitions, etc., which are at the latter end of the tape.
4. Then, SABR halts at the PAUSE statement which is at the physical end of the tape. At this time the user should reposition the symbolic tape in the reader at the physical beginning of the tape, and then press CONTInue. SABR now assembles the executable code portion of the tape in the normal way.
5. If an assembly listing is desired, proceed as in Method 2 after SABR finishes the assembly pass.

THE LINKING LOADER

Relocatable binary program tapes produced by SABR assembly are loaded into memory by using the 8K System Linking Loader.

The Linking Loader is capable of loading and linking a user's program and subprograms in any fields of memory. It is even capable, in a special way, of loading programs over itself. The Linking Loader also has options which give storage maps and core availability.

The Linking Loader requires a PDP-8/I, -8/L, -8, -8/S or -5 Computer with a least 8K words of core memory. Either high-speed or ASR paper tape input is acceptable, however, a high-speed reader is highly recommended.

The software requirements are:

- a. Binary paper tape copy of the Linking Loader
- b. Relocatable binary paper tape copies of both Part 1 and Part 2 of the 8K System Library
- c. The relocatable binary paper tapes of the user's own program and subprograms which have been produced by assembling his programs with SABR.

Operation

Generally speaking, the Linking Loader is capable of loading any number of user and Library programs into any field of PDP-8 memory. These programs are loaded consecutively via the high-speed reader (or the ASR reader). The choice of which field to load each program into is a switch register option. Usually, several programs may be loaded into each field. Because of the space reserved for the Linkage Routines the available space in field 0 is three pages smaller than in all other fields.

Any COMMON storage reserved by the programs being loaded is allocated in field 1 from location 0200 upwards. The space reserved for COMMON is obviously subtracted from the available loading area in field 1. The program reserving the largest amount of COMMON storage must be loaded first.

The Linking Loader uses the following special method to enable loading data over itself. When the Linking Loader encounters data which must be loaded over itself, it punches this data onto paper tape in RIM format. Then, after the user has finished loading all his relocatable binary program tapes, he simply loads the RIM format tape using the standard RIM loader.

The Run-Time Linkage Routines which are necessary to execute SABR programs are automatically loaded into the required areas of every field by the Linking Loader as a part of its initialization.

For the user, the only required knowledge of these routines is the particular areas of core they occupy.

Linkage Routine Locations

Because the Library Linkage Routines must be in core when SABR assembled programs are run, certain core locations are not available as follows:

Field 0	Locations 0400-0777
Field 0, 1, 2, . . .	Locations 0007 and 0033-0073

Thus in every field of memory the following page 0 locations are available to the user:

0000-0006	for interrupts, debugging, etc.
0010-0017	auto-index registers
0023-0032	arbitrary
0074-0177	arbitrary

RESERVED LOCATIONS. Locations 20, 21, 22 in field 1 are used for the Floating-Point Accumulator. The user should use these locations with great care. When using the Library routines, locations 20-32 in the field where the routines reside, are used for temporary storage by the routines. Locations 176 and 177 in the field where the I/O handler routines (IOH) reside are used for temporary storage by the I/O handler.

The 8K System Library subprograms, which may be used by any SABR program, are loaded in the same way as other relocatable binary programs. Only those library programs which the user's programs actually call need to be loaded.

Switch Register Options

During the loading operation with the Linking Loader, two user options are available to obtain information about what has already been loaded. The switch register is used to select these options. Either option may be selected after any program has finished loading.

WARNING

The Teletype punch must be at OFF or FREE before selecting these options.

The switch register bits used are as follows:

BIT 0 = 1 selects the Core Availability option;
BIT 1 = 1 selects the Storage Map option.

The Core Availability option causes the number of free pages of memory in every field of memory to be typed in a list on the Teletype. For example, if the user has a 16K configuration, a list like the following might be typed.

```

0002      (number of free pages in field 0)
0010      (number of free pages in field 1)
0030      (number of free pages in field 2)
0036      (number of free pages in field 3)

```

The number of pages initially available in field 0 is 0033 and in all other fields is 0036.

The Storage Map option causes a list of all program entry points to be typed, along with the actual address at which they have been loaded. The entry points of programs which have been called but which have not been loaded are also listed along with a U flag for undefined. Such flagged programs must be loaded before execution of the user's programs is possible. The Core Availability list is automatically appended to the Storage Map. A sample is shown below.

```

MAIN      10200
READ      01055
WRITE     01066
IOH       03031
SETERR    00000 U
ERROR     00000 U
TTYOUT    00000 U
HSOUT     00000 U
TTYIN     00000 U
HSIN      00000 U
FDV       04722
CLEAR     05247
IFAD      05131
FMP       04632
ISTO      05074
STO       04447
FLOT      05210
FAD       04010
DIV       00000 U

```

Loading the Linking Loader

The Linking Loader must be loaded into the highest available field of memory.

1. Make sure the Binary Loader is in memory, for example, in field m.
2. Let h represent the number of the highest field in the user's configuration.
3. Set the console switches as follows:
Data Field = h, Instruction Field = m, Switch Register = 7777.
4. Press LOAD ADDRESS.
5. Place the binary paper tape of the Linking Loader in the reader.
6. If using a high-speed reader, depress switch register Bit 0.
7. Press START. The Linking Loader will now be loaded into memory.

Loading Relocatable Programs

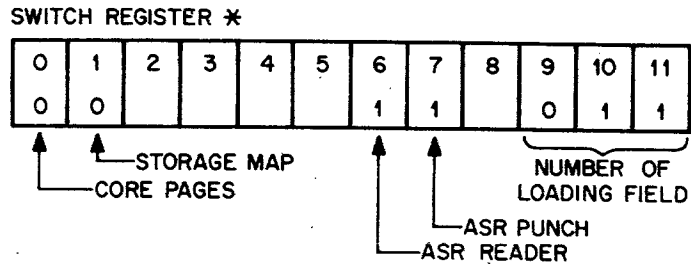
The Linking Loader is used to load the user's relocatable programs and 8K Library subprograms as outlined below.

NOTE

The program or subprogram which uses the largest amount of COMMON storage should be loaded first. (The Library subprograms do not use COMMON.)

1. After the Linking Loader has been loaded into the highest memory field, h, the user should set the console switches as follows: Data Field = h, Instruction Field = h, Switch Register = 0200.
2. Press LOAD ADDRESS.
3. Place the relocatable binary tape for the first program to be loaded in the reader. Position the tape with leader code in the reader.
4. Set switch register to 0000. Then, if loading via the Teletype reader is required, raise switch register bit 6. If the user does not have a high-speed punch, he should raise switch register bit 7. Finally, set switch register bits 9-11.

to the number of the field into which the first program or subprogram is to be loaded.



Example:

If the user wishes to load his first program into field 3, and if he has no high-speed I/O device, then he should set the switch register to 0063 before the next step.

5. Press START.
6. The user's relocatable binary program will now be loaded. When loading is completed, the Linking Loader halts.
7. The user may now either load another program or select one of the options in steps 9 and 10.
8. To load another program, insert the program relocatable binary tape in the reader, set switch register bits 9-11 to the number of the field the program is to be loaded into, and then press CONTInue.
9. To select the Core Availability option, set switch register bit 0 = 1, and press CONTInue.
10. To select the Storage Map option, set switch register bit 1 = 1, and press CONTInue.*

If the ASR punch is turned on for possible RIM format data punching, as explained on page 14-51, ensure that it is turned off before selecting either of the options. Turn it on again after the typing of the options is completed.

11. The user may continue loading more programs as in step 8 after using either of the options.

Any time the Linking Loader halts, the user may access memory directly via the DEPosit and EXAMine console

* All other switch register bits are irrelevant.

switches. After this is done the Linking Loader may be restarted via the console switches at location 7200 (in the highest field, where the Linking Loader resides).

THE DISK LINKING LOADER

The Disk Linking Loader (LLDR) is used to load and execute 8K FORTRAN compiled and 8K SABR assembled user programs when the system configuration includes one or more disks and the Disk Monitor System. Such user programs exist as a main program with several subprograms (including necessary 8K library subprograms), all of which must be on punched paper tape in relocatable binary format. LLDR loads these multiple-part programs in a page-wise relocatable manner, and links all calls to and returns from external subprograms.

The user communicates with LLDR via the keyboard in a simple, straightforward manner; LLDR types *OPT- and the user responds with a one-letter code which causes LLDR to perform one of seven possible functions (operations).

LLDR, unlike the standard 8K Linking Loader, is entirely keyboard oriented and makes extensive use of the disk. For example:

- a. It allows user programs to be loaded over LLDR itself by utilizing temporary disk storage in the Disk Monitor System environment.
- b. It provides two levels of program overlaying so that much larger programs can be run. Up to eight files (programs and subprograms) can be loaded into each overlay area. Overlay files are saved on the disk and called into core as needed at program execution time.
- c. It provides several utility and convenience features such as storage map listing, a listing of necessary subprograms not present in core, a listing of available (unoccupied) core, and automatic program starting.
- d. It includes load-time monitoring via the keyboard rather than the console switches, and several other minor features.

LLDR accepts paper tape input only, from either the low- or high-speed readers, as do both the 8K FORTRAN and 8K SABR systems. However, the user program (during execution) can use both disk and DECTape for input/output.

The operating system (Run-Time Linkage Routines) necessary

for execution of 8K FORTRAN and 8K SABR programs is contained within the LLDR program, and its use is entirely automatic.

Two loading techniques are provided: normal loading and overlay loading. In normal loading, each file is loaded into a separate core area where it remains during execution. In overlay loading, several files are sequentially loaded into the same core area and saved on the disk. At execution time, each file is brought from the disk into core when it is needed. LLDR provides two levels of overlay, and each allows up to eight files per overlay level. A normally loaded program may call a program in either overlay level, and a program in either overlay level may call a program in the other level.

The following main stipulations should be remembered when using LLDR.

a. A program in an overlay level may not call another external program in the same overlay level, except as explained in Overlay Loading.

b. Common storage (i.e., data storage accessible by all programs and subprograms) is always located in field 1.

c. The program or subprogram which requests the largest amount of common storage must be loaded first.

d. No one program or subprogram may be greater than 4K in length.

e. Programs may not be loaded across field boundaries, although they may be loaded into any available field.

f. Overlay files may not be loaded over LLDR, although normal files may be.

LLDR requires a PDP-8/I, -8/L, -8, or -8/S computer with at least 8K words of core, a Teletype and at least one disk. A high-speed paper tape reader is optional but highly recommended. LLDR can use all available core memory and disk storage.

Loading, Saving and Starting LLDR

LLDR is furnished on punched paper tape in binary-coded format, and is loaded into field 0 by the standard Binary Loader (refer to PDP-8/I System User's Guide, DEC-08-NGCB-D).

Before using LLDR or saving it as a systems program on the disk, it should be properly initialized for the amount of core avail-

able and for the type of paper tape reader to be used. LLDR is initially set for a basic configuration of 8K words of core and a high-speed paper tape reader. With any other configuration, LLDR should be started and initialized as explained under Expanded Configuration. Complete loading, saving, and calling procedures are given below for both basic and expanded configurations. The following procedures assume that the user is familiar with the Disk Monitor System, and that the system is available for use.

BASIC CONFIGURATION. The user with 8K of core and a high-speed reader should use the following procedures.

a. Determine that the Disk Monitor is in memory. (Type CTRL/C* or START at 07600.)

b. When Monitor responds with a dot, call the system loader by typing

.LOAD (followed by the RETURN key)

c. Insert the LLDR binary tape in the high-speed reader.

d. Answer the loading command dialogue as follows:

*IN-R:	Keys shown within angle brackets
*	are not echoed on the teleprinter
*OPT-1	when typed by the user.
*ST=	
↑ <CTRL/P> ↑ <CTRL/P>	

After each up-arrow which is typed by Monitor, the user types CTRL/P by holding down the CTRL key while typing the P key; this is equivalent to pressing the CONT switch when loading manually.

e. LLDR is now loaded into core; save it on the disk by typing

.SAVE LLDR!0-6777;200

f. LLDR may now be called to load relocatable binary programs by typing

.LLDR

* CTRL/C is typed by holding down the CTRL key while typing the C key.

EXPANDED CONFIGURATION

The user, with any configuration other than the basic configuration mentioned above, should use the following procedure:

a. Determine that the Disk Monitor is in memory. (Type CTRL/C or START at 07600.)

b. When Monitor responds with a dot, call the system loader by typing

.LOAD

c. Insert the LLDR binary tape in the appropriate reader.

d. Answer the loading command dialogue as follows:

*IN-R: (R: for high-speed reader
* T: for ASR reader)

*OPT-1

*ST = 7400

↑ <CTRL/P> ↑ <CTRL/P>

e. LLDR is now loaded into core. It automatically starts at location 7400, causing it to type out its initialization questions. Answer the questions as shown below.

*GIVE SIZE OF MEMORY IN K-12 (user typed 12)

*HIGH SPEED READER? Y (user typed Y)

When answering the first question, the user should type the amount of available core memory after K-; the user should type Y for yes, or N for no in answer to the second.

f. When the above questions have been properly answered, LLDR may be saved on the disk by typing

.SAVE LLDR!0-6777;200

g. LLDR may now be called to load relocatable binary programs by typing

.LLDR

LLDR Functions

When LLDR has been initialized and started as described in the preceding section, it types its program version number (also found on the paper tape identification label) and option statement and then waits for the user to specify the desired function to be performed. For example:

PDP-8 DEC-08-A2B4-02

*OPT-

The user's response to *OPT- is in the form of a one-letter code followed by the RETURN key. LLDR's functions and corresponding one-letter function codes are listed below.

Code	Function
C	Core availability listing
D	Disk file assignment
E	Exit with halt
L	Normal loading
M	Storage map listing
O	Overlay loading
S	Start main program
U	Unloaded program listing

Functions may be called whenever needed or desired, except that the M, U, and S functions must not be called first. Upon completion of a function (except E or S), LLDR will request another by repeating the option statement (*OPT-). Any error made by the user when responding to an option statement will cause LLDR to type a question mark, ignore the response, and repeat the option statement.

LLDR may be stopped (e.g., to make a program patch) and restarted without altering the state of the computer by using the console STOP switch and restarting at 6000 FIELD 0. This method may be used at any time after completion of any function other than D, except during overlay loading or while a tape is actually being read.

At any time during the use of LLDR (except while a tape is being read in), control may be returned to the Disk Monitor. This is done by typing CTRL/C; however, when CTRL/C is typed, all data temporarily stored on the disk is lost.

Disk File Assignment Function (D)

If the user's programs or subprograms create or use disk data files with the RDISK and WDISK library functions, the D function must be the first function used. The D function performs the preliminary job of entering the names of user files into the disk directory. This prepares the way for using the RDISK and WDISK library functions, which allow the user to read and write data on the disk at execution time.

Use of the D-function proceeds as shown below:

```
PDP-8 DEC-08-A2B4-01
*OPT-D
*FILES-ABC, WXYZ, M1, M2, 5H, R, 3, P
*OPT-
```

where a directory entry is assigned to each of the eight file names. File names may be from one to four characters in length, and up to ten files may be specified. All such files must be named in one execution of the D function.

The order in which the data files are named for the D function is especially important. The reason for this is that when the user's program references disk data files using the RDISK and WDISK library functions, he must reference these files not by name but by logical number (1, 2, . . . , 10). This logical number is determined by the order in which he names the files for the D function. For example, if files have been named in the D function as shown in the previous example, the user's program will reference file M1 by statements of the form

```
CALL RDISK (3, . . .)
```

because M1 was the third file named.

Before using the D function the user should study thoroughly the operation and use of the RDISK and WDISK library functions under Disk I/O Routines.

The disk directory will accommodate ten file names. If the directory is too full to accommodate all files named, a meaningful error message is printed by LLDR. In the example above, if the directory had room for only four files, the error message

```
DISK WILL NOT HOLD 5H & FOLLOWING FILES
```

would have been printed. If this happens, the entire D function request is ignored and LLDR prints another *OPT- to allow the user to repeat the D function with fewer files or to specify a different function.

After the D function has been performed, LLDR will again print *OPT- for the user to continue with the process of loading his program. After the D function has been used or when a different function has been called, the D function is no longer available

—if called a second time or after a different function, it is treated as an illegal function code.

Again, if the D function is to be used, it must be the first function used. If it is not chosen as the first function, it is not available for use until a fresh image of LLDR is brought into core from the disk.

Loading Functions (L and O)

The two loading functions, L for normal loading and O for overlay loading, are available for use at any time. These are the principal functions of LLDR—to load relocatable programs for execution. Programs and subprograms may be loaded in any order and into any field. The only restrictions are listed below.

a. The subprogram which requires the largest amount of common storage must be loaded first.

b. No subprogram may be loaded across a core field boundary; i.e., no subprogram may be longer than 4K in length.

c. A maximum of 64 subprograms may be loaded, including multiple entry points for single programs.

LLDR loads subprograms in the order presented and into the field specified (see below) from the lowest available memory upward. Common storage is allocated in the lower portion of field 1 before loading actually starts. A maximum of 3840 words of common storage fills field 1.

LLDR loads in a page-wise relocatable fashion (each program begins at the start of a new core page), establishing external links so that each subprogram is properly executed.

NORMAL LOADING (L)

In normal loading, the user's program is loaded directly into core memory where it remains available for, throughout, and after execution. The core area occupied by each normally loaded program is the property of that program, and no other program can be loaded into its core area.

To perform normal loading, the user responds to *OPT- with the letter L. When this is done, LLDR types a request for the number of the field in which the user wishes to load. This specified field must exist in the configuration. For example:

*OPT-L

*FIELD-2

Had field 2 been nonexistent, the following would have occurred:

```
*OPT-L
*FIELD-2
?
*OPT-
```

where LLDR ignored the user's response, typed the question mark, and repeated the option statement.

When LLDR is satisfied with user response, it then types an up-arrow. At this point LLDR will pause and wait for the user to place his relocatable binary tape in the tape reader, and to type CTRL/P which causes LLDR to load the program into core. When the program has been loaded, LLDR will type another up-arrow and pause for user response. If the user wishes to load another program into the same field, he need only place the tape in the reader and then type another CTRL/P (or press the CONTInue switch and then type CTRL/P if using the low-speed tape reader). When the user no longer wishes to load into the same field, he should respond to the up-arrow by typing the RETURN key, and LLDR will type another option statement.

The user may respond to an up-arrow with CTRL/N, which causes LLDR to by-pass the next program on a multi-program tape. This situation may, for example, occur with a library sub-program tape.

A typical example of normal loading is shown below, where three programs are loaded into field 0 and two into field 1, with one program being by-passed.

```
*OPT-L
*FIELD-0
* ↑ <CTRL/P> ↑ <CTRL/P> ↑ <CTRL/P> ↑
*OPT-L
*FIELD-1
* ↑ <CTRL/P> ↑ <CTRL/N> ↑ <CTRL/P> ↑
*OPT-
```

If the low-speed reader had been used in the example above, the CONTInue switch would have been pressed just before each CTRL/key combination.

OVERLAY LOADING (O)

Overlay loading allows the user to load as many as 16 subprograms into the same core area. The user may load one or two overlay levels (each O function call constitutes an overlay level) of subprograms (files) with up to eight files per level. Overlay loading is possible only when no two subprograms of the same level need to be in core at the same time; i.e., they do not call each other.

All subprograms loaded during the operation of an O function are loaded into the same core area (overlay level) and automatically saved in separate files on the disk. At execution time each file is called back into core as needed. No protection is given to the file of this overlay level that was previously in core. It is completely overwritten in core. Overlay files should use common storage for data which must remain in core.

Files in a given level may be loaded in any order, provided they are all loaded during the same execution of O function. Files in a given level need not be the same length; enough core is allocated for the largest file in the level.

Loading with the O function is quite similar to loading a string of programs in the same field using the L function. An example is given below, where three files are loaded into the first level and two files into the second level, with one file being passed over.

```
*OPT-O
*FIELD-1
*↑<CTRL/P>↑<CTRL/P>↑<CTRL/P>↑
*OPT-O
*FIELD-1
*↑<CTRL/P>↑<CTRL/N>↑<CTRL/P>↑
*OPT-
```

Loading of a single overlay level is terminated with the RETURN key. Loading of an overlay level will automatically be terminated after eight files have been loaded.

As with the L function, if the low-speed reader had been used in the example above, the CONTINUE switch would have been pressed just before each CTRL/key combination.

When the main program is removed from core, linkage to its overlay files is broken. Therefore, for subsequent execution, files must be reloaded with the main program.

Loading Errors

When LLDR detects an error during loading of a program, it types an error message of the following form:

ERROR 000n

where n is a digit from 1 to 6, representing the type of error detected. If the error is fatal, control returns to the Disk Monitor. If it is not fatal, the user may be able to continue loading.

<u>Error No.</u>	<u>Error</u>	<u>Fatal?</u>
1	Attempt to load more than 64 subprograms	Yes
2	Field overflow	No
3	Subprogram with largest common assignment not loaded first	Yes
4	Checksum error	No
5	Improper or damaged tape or reader error	No
6	Disk overflow	No

NON-FATAL ERRORS

Error 2—During normal loading, loading may be continued in a different memory field. During overlay loading, the entire overlay level must be reloaded into a different memory field.

Errors 4 and 5—During either type of loading, the user may reposition the faulty tape in the reader and type CTRL/P in response to the new up-arrow. If the error persists, reassembly or hardware maintenance will be necessary.

Error 6—Occurs during normal loading only when the user is loading into the upper portion of field 0; the program which caused the error must be loaded into a different field. During overlay loading, the current overlay level will be closed with only the files that were loaded successfully. The file which caused the overflow (the last file read) and succeeding files will have to be loaded normally.

Utility Functions (C, M, and U)

CORE AVAILABILITY (C)

The user may at any time request a list of the number of pages

available for loading in each core field. The following example assumes that the user has a 16K computer (4 fields):

```
*OPT-C
0033
0036
0036
0036
*OPT-
```

The numbers listed are the octal number of free pages left in fields 0, 1, 2, and 3, respectively.

STORAGE MAP (M)

During the link-loading process, LLDR builds a list of external symbols; i.e., main program and subprogram entry points and their actual starting addresses. This list forms a complete storage map of all programs loaded, as shown below:

```
*OPT-M
MAIN      10200
READ      01055
WRITE     01066
IOH       03031
SETERR    00000  U
TTYIN     00000  U
.
.
.
FLOAT     05046
FIX       04513
*OPT-
```

Starting addresses are expressed in five octal digits—the first digit represents the memory field and the other four the address in that field. The U means that the stated subprogram has been called but has not been loaded, and therefore must be loaded before successful execution is possible.

Listing of the storage map may be prematurely terminated by typing CTRL/P.

UNLOADED PROGRAM LISTING (U)

This function is used to obtain a list of those subprograms which must still be loaded before successful execution is possible. All symbols flagged with a U in a storage map listing will be listed as shown below:

*OPT-U
SETERR
TTYIN
TTYOUT
HSIN
HSOUT
*OPT-

This listing may also be prematurely terminated by typing CTRL/P.

Exit Functions (E and S)

The E function is used to cause a halt after all loading is complete. The S function is used to automatically start execution of the loaded program at the beginning of the main program.

Both of these functions signal LLDR that loading is complete. They each cause any data which has been temporarily saved on the disk (except overlay files) to be read into core.

When the E function is used, LLDR reads in all data temporarily stored on the disk and then halts. The users' entire program (except overlay files) will be in core, ready for patching, execution, or saving on the disk.

When the S function is used, LLDR checks for a subprogram called MAIN (such as a FORTRAN main program). If found, execution will automatically start at the starting address of MAIN. If MAIN is not found, the S function is executed as an E function.

Overlay Loading

In general, any group of subprograms which do not call each other (either directly or indirectly) may be loaded into the same overlay level. A typical situation follows:

MAIN	contains calls to	A, B, C, D, E
A	contains calls to	D, E, F
B	contains calls to	D, G
C	contains calls to	D, E, H
D	contains calls to	E
E, F, G, H,	contain no external calls	

The above combination may be loaded as follows:

Normal	Overlay 0	Overlay 1
MAIN	A	D
E	B	F
	C	G
		H

If D contains a call to any other than E, it would be better to load D normally and put E in overlay 1. If F were to call B, the above loading situation would not work; A would be calling B indirectly, and these two are in the same overlay level.

It is possible, however, to call another program in the same overlay level only if the called program never attempts to return to the calling program. In this way, simple chaining may be achieved. For example, a very long FORTRAN main program can be split into sections with each section terminated by a call to the next. Such a situation is shown below.

MAIN	calls A, B, C and is terminated by a call to MAIN2
MAIN2	calls A, B, C and is terminated by a call to MAIN3
MAIN3	calls A, B, C and is terminated by a call to MAIN4
•	
•	
•	
MAIN8	calls A, B, C and stops
A, B, C	contain no external calls

The above combination may be loaded as follows:

Overlay 0	Overlay 1
MAIN	A
MAIN2	B
MAIN3	C
•	
•	
•	
MAIN8	

When the MAIN program is contained in an overlay area, the E function cannot be used unless MAIN is loaded last into the overlay level. The S function will work with the above combination since it works regardless of the order in which the segments of MAIN are loaded.

With FORTRAN programming alone, a subprogram other than a MAIN program may not be chained. However, this is possible with careful assembly language programming. An example of such programming is shown below, where SUB is split into a two-part chain, SUB and SUB2. MAIN is a standard FORTRAN program containing calls to SUB in the form:

```
CALL SUB (A1, A2, A3)
```

SUB is written as a standard FORTRAN program which does part of the work for the entire subroutine chain, including processing arguments A1 and A2. It is written with two arguments and concludes with Z, which is any dummy argument. After SUB has been compiled and before the intermediate compiler symbolic is assembled, it should be edited to include the insertions enclosed in brackets.

```
[X,      COMMN2]
ENTRY SUB
SUB,     BLOCK 2
        .
        .
        .
        [TAD     SUB           /SAVE RETURN FIELD
          DCA     X
          TAD     (-2          /-2*NO. OF ARGS TO PASS
          TAD     SUB#        /SAVE ARGUMENT ADDRESS
          DCA     X#
          CALL    1, SUB2
          ARG     Z
          END
```

SUB2 is also a standard FORTRAN program containing the latter portion of the entire subroutine, including the processing of argument A3. The actual contents of SUB2 is coded in FORTRAN just as if it were a subroutine taking one argument. After SUB2 has been compiled, the compiler symbolic output is edited as shown below:

```

[X,      COMMN 2]
        ENTRY SUB2
SUB2,    BLOCK 2
        TAD      X          /REPLACE ARG POINTER
        DCA      SUB2
        TAD      X#
        DCA      SUB#
        .
        .
        .
        RETRN   SUB2
        END

```

User Program Execution

If the user chooses not to execute his program automatically with the S function, he may determine the exact address for the start (using the storage map or assembly listing), and execute his program, using the console switches or the Disk Monitor.

At execution time, the Run-Time Linkage Routines must be in core. These routines accomplish the necessary linkage for all calls to and returns from external subprograms, all off-page indirect references, and all off-field references (including those to common and passing subroutine arguments).

If, during execution of a user program, a call is made to a non-existent program or subprogram, an unconditional halt will occur and control will return to the Disk Monitor. This error is fatal.

Program execution may be terminated at any time by typing CTRL/C. However, when CTRL/C is typed, all overlay files stored on the disk are lost.

Storage Allocation

The following core availability map allows the user to plan his loading.

Field 0	
0000-0777	Used by the Run-Time Linkage Routines and not available to the user for loading.
1000-4377	Available for any loading.
4000-7577	Residence of LLDR during loading. Available for normal loading (by automatic use of temporary disk storage), but not available for overlay loading.
7600-7777	Disk Monitor permanent residence.

ERROR MESSAGES

SABR

Because SABR is a one-pass automatic paging assembler object errors are difficult to correct. If there are errors in the source, the assembled binary code will be virtually useless. Both errors E and S are fatal, assembly halts when they are encountered. The other types of errors are not fatal, but they cause the line in which they occur to be treated as a comment and thus essentially ignored. An address label on such a line will remain undefined and no space is reserved in the binary output for the erroneous data.

During the assembly pass error messages are typed on the teletype as they occur.

```
C      AT      LOC      +0004
```

This means that an error of type C has occurred at the fourth instruction after the location tag LOC. This line count includes comment lines and blank lines.

During the listing pass, the error is typed in the address field of the instruction line.

The following error messages may occur.

- A Too many or too few ARGs follow a CALL statement.
- C An illegal character appears on the line. This could possibly be an 8 or 9 in an octal digit string or an alphabetic character in a digit string.
- M A symbol is multiply defined (occurs only during Pass 1). It is impossible to resolve multiple definitions during Pass 2; therefore, listings of programs which contain multiple definitions will have unmarked errors.
- I An illegal syntax has been used. Below are listed the types of illegal syntax that may occur.
 - a. A pseudo-op with improper arguments.
 - b. A quote mark with no argument.
 - c. A non-terminated text-string.
 - d. A memory reference instruction with improper address.
 - e. An illegal combination of micro-instructions.
- E There is no END statement.
- S either one of three things:
 - a. The symbol table has overflowed. This can be cor-

- ected by using fewer symbols, using shorter symbols, or by breaking the program into smaller parts.
- b. Common storage has been exhausted.
- c. More than 64 different user-defined symbols have occurred in a core page.
- d. More than 64 external symbols have been declared.

One further type of error may occur. This is an undefined symbol. Because SABR is a one-pass assembler, an undefined symbol cannot be determined until the end of the assembly pass, so the error diagnostic UNDF is given in the symbol table listing.

LINKING LOADER

If during the process of loading a program or subprogram the Linking Loader encounters an error, the user is notified by an error message; the partially loaded program or subprogram is ignored, removed from the field, and core is freed. The error messages are typed out in the form

ERROR XXXX

where XXXX is the error code number.

<u>Error Code</u>	<u>Explanation</u>
0001	More than 64 ₁₀ subprogram names have been seen by the Loader (64 ₁₀ subprogram names is the capacity of the Loader's symbol table).
0002	The current field is full, or load was to non-existent memory.
0003	The current subprogram has too large a COMMON storage assignment. (Subprogram with largest common storage declaration must be loaded first.) This is a semi-fatal error. Re-initialize the Linking Loader as explained below and reload the programs in the proper order.
0004	Checksum error in input tape. If the error persists, re-assembly is necessary.
0005	Illegal Relocation Code has been encountered. This can occur only if the relocatable binary tape is bad or if the user is using it

improperly (e.g., not starting at the beginning of the tape, or reader error, or punch error). If the error persists, reassembly is necessary.

Recovery from errors 2, 4, and 5 is accomplished by repositioning the tape in the reader to the leader code at the beginning of the subprogram and then pressing CONTInue. When attempting to recover from one of these errors, no other program should be loaded before reloading the program which caused the error. Obviously, on Error 2 a different field should be selected before pressing CONTInue.

The entire loading process may be restarted via the console switches, at any time by reinitializing the Linking Loader. To do this, set the console switches as follows: Data Field = h (the field where the Linking Loader resides), Instruction Field = h, Switch Register = 6200; then press LOAD ADDRESS and START.

DISK LINKING LOADER (See section entitled, DISK LINKING LOADER)

LIBRARY PROGRAM

During execution, the Library programs check for certain errors and type out the appropriate error messages in the form

“XXXX” ERROR AT LOC NNNN

where XXXX specifies the type of error, and NNNN is the location of the error. When an error is encountered, execution stops, and the error must be corrected.

When multiple error messages are typed, the location of the last error message is relevant to the user program. The other error messages are to subprograms called by the statement at the relevant location.

<u>Error Code</u>	<u>Explanation</u>
“ALOG”	Attempt to compute log of negative number
“ATAN”	Result exceeds capacity of computer
“DIVZ”	Attempt to divide by 0
“EXP”	Result exceeds capacity of computer
“FIPW”	Error in raising a number to a power
“FMT1”	Multiple decimal points
“FMT2”	E or . in integer

<u>Error Code</u>	<u>Explanation</u>
"FMT3"	Illegal character in I, E, or F field
"FMT4"	Multiple minus signs
"FMT5"	Invalid FORMAT statement
"FLPW"	Negative number raised to floating power
"FPNT"	Floating point error: may be caused by Division by zero; floating point overflow; at- tempt to fix too large a number.
"SQRT"	Attempt to take root of a negative number

To pinpoint the location of a Library execution error:

1. From the Storage Map, determine the next lowest numbered location (external symbol) which is the entry point of the program or subprogram containing the error.
2. Subtract in octal the entry point location of the program or subroutine containing the error from the LOC of the error in the error message.
3. From the assembly symbol table, determine the relative address of the external symbol found in step 1 and add that relative address to the result of step 2.
4. The sum of step 3 is the relative address of the error, which can then be compared with the relative addresses of the numbered statements in the program.

THE SUBPROGRAM LIBRARY

The Library is set of subprograms which may be CALLED by any FORTRAN/SABR program. The relocatable binary versions of these subprograms are arranged in two paper tapes for the convenience of the user. Part 1 contains those subprograms which are used by almost every FORTRAN/SABR program. All the Library subprograms are described below.

Many of the subprograms reference the Floating-Point Accumulator located at ACH, ACM, ACL (20, 21, 22 of field 1).

Input/Output

READ is called to initialize the I/O handler before reading data. WRITE is called to initialize the I/O handler before writing data. IOH is called for each item to be read or written. IOH must also be called with a zero argument to terminate an input-output sequence.

All of these programs require that the Floating-Point Accumulator be set to zero before they are called.

```
CALL      2, READ
ARG      (n          /n=DEVICE NUMBER
ARG      fa          /fa=ADDR OF FORMAT
...
CALL      1, IOH
ARG      data 1      /data 1=ADDR OF HIGH
                   /ORDER WORD OF
                   /FLOATING POINT
                   NUMBER

CALL      1, IOH
ARG      data 2
...
...
CALL      1, IOH
ARG      0
...
CALL      2, WRITE
ARG      (n
ARG      fa
```

The following device numbers are currently implemented:

- 1 (Teletype keyboard/printer)
- 2 (High-speed reader/punch)

Floating Point Arithmetic

FAD is called to add the argument to the Floating-Point Accumulator.

```
CALL      1, FAD
ARG      adres
```

FSB is called to subtract the argument from the Floating-Point Accumulator.

```
CALL      1, FSB
ARG      adres
```

FMP is called to multiply the Floating-Point Accumulator by the argument.

```
CALL      1, FMP
ARG       adres
```

FDV is called to divide the Floating-Point Accumulator by the argument.

```
CALL      1, FDV
ARG       adres
```

CHS is called to change the sign of the Floating-Point Accumulator.

```
CALL      0, CHS
```

All of the above programs leave the result in the Floating-Point Accumulator. The address of the high-order word of the floating-point number is "adres".

STO is called to store the contents of the Floating-Point Accumulator in the argument address. The floating-point accumulator is cleared.

```
CALL      1, STO
ARG       storag  /storag=ADDRESS WHERE
                          /RESULT IS TO BE PUT
```

IFAD is called to execute an indirect floating point add to the Floating-Point Accumulator.

```
CALL      1, IFAD
ARG       ptr      /ptr=2-word POINTER
                          /TO HIGH ORDER
                          /ADDRESS OF FLOATING
                          /POINT ARGUMENT
```

ISTO is called to execute an indirect floating point store.

```
CALL      1, ISTO
ARG       ptr
```

CLEAR is called to clear the Floating-Point Accumulator. The AC is unchanged.

```
CALL      0, CLEAR
```

FLOT is called to convert the integer contained in the AC (processor accumulator) to a floating point number and store it in the Floating-Point Accumulator.

```
CALL      0, FLOT
```

FIX is called to convert the number in the Floating-Point Accumulator to a 12-bit signed integer and leave the result in the AC.

```
CALL      0, FIX
```

ABS leaves the absolute value of the floating point number at "addr" in the Floating-Point Accumulator.

```
CALL      1, ABS  
ARG      addr
```

Integer Arithmetic

MPY is called to multiply the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
CALL      1, MPY  
ARG      addr
```

DIV is called to divide the integer contained in the AC by the integer contained in "addr." The result is left in the AC.

```
CALL      1, DIV  
ARG      addr
```

IREM leaves the remainder from the last executed integer divide in the AC.

```
CALL      1, IREM  
ARG      0
```

(The argument is ignored.)

IABS leaves the absolute value of the integer contained in "addr" in the AC.

```
CALL      1, IABS  
ARG      addr
```

IRDSW reads the value set in the console switch register into the AC.

```
CALL      0, IRDSW
```

Subscripting

SUBSC is called to compute the address of a subscripted variable. The address is left in the AC. When SUBSC is called, it assumes that the AC contains the first dimension of the array. This dimension should be positive if the subscripted variable is an integer, and negative if the subscripted variable is a floating point number.

Assume S is a $20_8 \times 20_8$ floating-point array.

```
TAD      (20
CIA
CALL     3, SUBSC
ARG      i1      /i1=ADDRESS OF 2ND
                /SUBSCRIPT
ARG      i2      /i2=ADDRESS OF 1ST
                /SUBSCRIPT
ARG      base    /BASE ADDRESS
                /OF ARRAY
```

Functions

SQRT leaves the square root of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL     1, SQRT
ARG      addr
```

SIN, COS, TAN leave the specified function of the floating-point argument at "addr" in the Floating-Point Accumulator.

```
CALL     1, SIN
ARG      addr
```

ATAN leaves the arctangent of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL     1, ATAN
ARG      addr
```

ALOG leaves the natural logarithm of the floating-point number of "addr" in the Floating-Point Accumulator.

```
CALL     1, ALOG
ARG      addr
```


EXP raises "e" to the power specified by the floating-point number at "addr" and leaves the result in the floating-point accumulator.

```
CALL      1, EXP
ARG      addr
```

All of these subprograms require that the floating-point accumulator be set to zero before they are called.

POWER (IIPW, IFPW, FIPOW, FFPOW)

These routines are called by FORTRAN to implement exponentiation. The address of the first operand is in the AC (floating-point or processor depending on mode), and the address of the second is an argument. The address of the result is in the appropriate AC upon return.

FUNCTION NAME	MODE OF OPERAND 1 (BASE)	MODE OF OPERAND 2 (EXPONENT)	MODE OF RESULT
IIPW	INTEGER	INTEGER	INTEGER
IFPW	INTEGER	FLOATING POINT	FLOATING POINT
FIPOW	FLOATING POINT	INTEGER	FLOATING POINT
FFPOW	FLOATING POINT	FLOATING POINT	FLOATING POINT

```
CALL      2, FFPOW
ARG      addr 2      /ADDRESS OF OPERAND 2
```

LIBRARY ORGANIZATION

Part 1.	"IOH"	contains	IOH, READ, WRITE
	"FLOAT"	contains	FAD, FSB, FMP, FDV, STO, FLOT, FLOAT, FIX, IFIX, IFAD, ISTO, CHS, CLEAR
	"INTEGER"	contains	LREM, ABS, IABS, DIV, MPY, IRDSW
	"UTILITY"	contains	TTYIN, TTYOUT, HSIN, HSOUT, OPEN, CKIO
	"ERROR"	contains	SETERR, CLRERR, ERROR
Part 2.	"SUBSC"	contains	SUBSC
	"POWERS"	contains	IIPW, IFPW, FIPOW, FFPOW, EXP, ALOG
	"SQRT"	contains	SQRT
	"TRIG"	contains	SIN, COS, TAN
	"ATAN"	contains	ATAN

DECTAPE I/O ROUTINES

RTAPE and WTAPE (read and write tape) are the DECTape read and write subprograms for the 8K FORTRAN and 8K SABR systems. The subprograms are furnished on one relocatable binary-coded paper tape which must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECTape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms which may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

- a. DECTape unit number (from 0 to 7)
- b. Number of the DECTape block at which transfer is to start. The user may direct the DECTape service routine to begin searching for the specified block in the forward direction rather than the usual backward direction by making this argument the two's complement of the block number.
- c. Number of words to be transferred ($1 \leq N \leq 4096$)
- d. Core address at which the transfer is to start.

In 8K FORTRAN, the CALL statements to RTAPE and WTAPE are written in the following format (arguments are taken as decimal numbers):

```
CALL RTAPE (6, 128, 388, LOCA)
```

In 8K SABR, they are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4, WTAPE /WOULD BE SAME FOR RTAPE
ARG (6 /DATA UNIT NUMBER
ARG (200 /STARTING BLOCK NUMBER IN
OCTAL
ARG (604 /WORDS TO BE TRANSFERRED
IN OCTAL
ARG LOCB /CORE ADDRESS, START OF
TRANSFER
```

In these examples, LOCA and LOCB may or may not be in COMMON.

As a typical example of the use of RTAPE and WTAPE, assume that the user wants to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively. Since PDP-8 DECTape is formatted with 1612 blocks (numbered 1-2700 octal) of 129 words each (for a total of 207,948 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively. Each array must be stored beginning at the start of some DECTape block. The user may write these arrays on tape as follows:

```
CALL WTAPE (0, 1, 2000, A)
CALL WTAPE (0, 17, 400, B)
CALL WTAPE (0, 21, 400, C)
CALL WTAPE (0, 25, 20, D)
```

The user may also read or write a large array in sections by specifying only one DECTape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE (0, 17, 258, B(1))
CALL RTAPE (0, 19, 129, B(259))
CALL RTAPE (0, 20, 13, B(388))
```

As shown above, it is possible to read or write less than 129 words by starting at the beginning of a DECTape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECTape block may not be read without reading the first 119 words as well.

A DECTape read or write is normally initiated with a backward search for the desired block number. To save searching time, the user may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least fourteen block numbers greater than the last block number used. For example, if the user has just read array A and now wants array D, he may write:

```
CALL RTAPE (0, 1, 2000, A)
CALL RTAPE (0, -25, 20, D)
```

DISK I/O ROUTINES

ODISK and CDISK (open disk and close disk) and RDISK and WDISK (read disk and write disk) are the four DECdisk (DF32/DS32) input and output subprograms for the 8K FORTRAN and 8K SABR systems. They are furnished on one relocatable binary-coded paper tape which is loaded into core using the Linking Loader, where they occupy eight pages of core.

ODISK and CDISK

ODISK is used to open (activate) a file (named using the Linking Loader D function) so that the file can be read or written using RDISK or WDISK. CDISK will close (deactivate) a file which was opened with ODISK so that the contents of the file cannot be altered.

The ODISK and CDISK subprograms may be called with standard, explicit CALL statements, in any 8K FORTRAN or 8K SABR program. ODISK requires one argument when opening a file. However, it requires two arguments when specifying or changing the size (in blocks) of a file. CDISK always requires only one argument.

The first argument of both ODISK and CDISK is the logical number (from 1 thru 10 inclusive) of the file as it was named using the Linking Loader. The second argument to ODISK is the number of blocks (from 1 thru 128) to be saved for the file.

In 8K FORTRAN, the CALL statements to ODISK and CDISK are written in the following format (arguments must be decimal integer numbers):

```
CALL ODISK (1)
```

when opening a file, or

```
CALL ODISK (1, 5)
```

when specifying or changing the size of a file, and

```
CALL CDISK (1)
```

when closing an opened file.

In 8K SABR, the CALL statements to ODISK and CDISK are

written in the following format (arguments may be either octal or decimal numbers):

```
CALL 1, ODISK  
ARG (1          /LOGICAL FILE NUMBER
```

when opening a file, or

```
CALL 2, ODISK  
ARG (1          /LOGICAL FILE NUMBER  
ARG (5          /NUMBER OF BLOCKS, OCTAL
```

when specifying or changing the size of a file, and

```
CALL 1, CDISK  
ARG (1          /LOGICAL FILE NUMBER
```

when closing an opened file.

ODISK prepares the file named for data transfer. When running the user program using the Disk Monitor System, ODISK uses Disk Monitor I/O and the three scratch blocks on disk zero for a window whenever a file is opened.

All open files should be closed before terminating program execution, thus preserving the contents of the files.

RDISK and WDISK

The RDISK and WDISK subprograms may be called with standard, explicit CALL statements in any 8K FORTRAN or 8K SABR program. The ODISK subprogram must be used to open the file concerned before using the RDISK or WDISK subprograms.

Each of these subprograms requires four arguments, arranged as listed below:

1. Logical file number (determined using the Linking Loader D function),
2. Logical block of the file number (block number of the file where data transfer is to begin),
3. Number of words to be transferred (from 1 thru 4096)
4. Core address where data transfer is to start (field 0).

Both RDISK and WDISK require the arguments above.

In 8K FORTRAN, the CALL statements to RDISK and WDISK are written in the following format (arguments are taken as decimal numbers):

```
CALL RDISK (4, 2, 55, LOCA)
```

when reading file 4, beginning with block 2, transferring 55 words, starting at the location of tag LOCA, which may be the name of an array defined in a DIMENSION statement. WDISK would be formatted in the same fashion.

In 8K SABR, the CALL statements to RDISK and WDISK are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4, RDISK  /SAME FOR WDISK
ARG (4        /LOGICAL FILE NUMBER
ARG (2        /BLOCK OF FILE
ARG (55       /WORDS TO TRANSFER, OCTAL
ARG LOCA     /CORE ADDRESS OF START, FIELD 0
```

WDISK would be formatted in the same fashion.

A variable number of words may be transferred. It is not necessary to transfer in 200-word blocks, as with the Disk Monitor System.

DEMONSTRATION PROGRAM USING LIBRARY ROUTINES

The following demonstration program is a SABR program showing the use of the library routines. The program is written to add two integer numbers, convert the result into floating-point, and type the result in both integer and floating-point format. The source program was written and listed using the Symbolic Editor; the Disk Monitor System was used during assembly; and the assembled program was then loaded and run using the 8K Linking Loader.

The system configuration consisted of a PDP-8/I with 8K words of core, DF32 Disk, Teletype, and high-speed reader and punch. The Disk Monitor System, Symbolic Editor, and SABR Assembler were available on the disk. The Teletype paper tape reader was used during assembly for demonstration (printout) purposes.

Comments are interspersed throughout the listing to provide a step-by-step analysis.

After writing the source program it was printed and punched using the Symbolic Editor.

CTRL/C was typed after the asterisk to return control to the Disk Monitor.

SABR was transferred from the disk into core.

The source program tape was placed in the teletype reader.

When started, SABR printed its identification and initial dialogue questions which were answered.

The TTY reader must be set to START within 3 seconds after typing N to the last question. Otherwise, as was the case here, the error message will appear, and SABR must be restarted at location 0200, as was done here.

The initial dialogue questions are repeated and again answered.

After typing the N to the last question, the TTY reader was immediately set to START and assembly commenced.

The Symbol Table concluded the assembly.

Here the source program tape was again placed in the TTY reader and the CONTINUE switch was depressed. The program listing was printed.

The 8K Linking Loader was loaded into core using the Binary Loader, and started at location 0200 of field 1.

When started, the Linking Loader printed its identification.

Library Tape Part 1 was loaded into core by placing the tape in the TTY reader, setting the reader to START, and pressing CONTINUE.

After loading the library subprograms, switch register bit 1 was set to 1, and CONTINUE was pressed to get the storage map of the programs and subprograms loaded into core.

The last two numbers represent the number of free (available) pages in each core field—0004 free pages in field 0, and 0036 free pages in field 1.

To execute the compiled program, the switch register was set to

01000, the starting address of the main program (determined from the Storage Map).

The LOAD ADDRESS switch was pressed and then START switch was pressed.

The program ran as planned, producing the desired results.

```
ENTRY START

START, CALL    0,OPEN  /INITIALIZE IO DEVICES
      TAD     A      /COMPUTE C = A + B
      TAD     B
      DCA     C
      CALL    1,FLOAT /CONVERT TO FLOATING POINT
      ARG     C
      CALL    1,STO
      ARG     D
      CALL    2,WRITE /INITIALIZE THE IO HANDLER
      ARG     N      /DEVICE NUMBER 1 = TELETYPE
      ARG     FORMT  /FORMAT SPECIFICATION
      CALL    1,IOH  /TYPE THE INTECER NUMBER
      ARG     C
      CALL    1,IOH  /TYPE THE FLOATING POINT NR
      ARG     D
      CALL    1,IOH  /COMPLETE THE IO
      ARG     0
      HLT

FORMT, TEXT " ('THE ANSWERS ARE',I5,F7.2)"
N,      1
A,      2
B,      2
C,      0
D,      BLOCK  3
      END
```


*

.SAB1
.SABR

PDP-8 SABR DEC-08-A2B2-12
HIGH SPEED READER? N
HIGH SPEED PUNCH? Y
LISTING ON HIGH SPEED PUNCH? N

E AT +0000

HIGH SPEED READER? N
HIGH SPEED PUNCH? Y
LISTING ON HIGH SPEED PUNCH? N

0240	5047	FORMT, TEXT	"('THE ANSWERS ARE',I5,F7.2)"
0241	2410		
0242	0540		
0243	0116		
0244	2327		
0245	0522		
0246	2340		
0247	0122		
0250	0547		
0251	5411		
0252	6554		
0253	0667		
0254	5662		
0255	5100		
0256	0001	N,	1
0257	0002	A,	2
0260	0002	B,	2
0261	0000	C,	0
0262	0000	D,	BLOCK 3
0263	0000		
0264	0000		

A	0257
B	0260
C	0261
D	0262
FLOAT	0000EXT
FORMT	0240
IOH	0000EXT
N	0256
OPEN	0000EXT
START	0200EXT
STO	0000EXT
WRITE	0000EXT

. Demonstration Program

		ENTRY START	
0200	4033	START,	CALL 0,OPEN /INIT IO DEVS
0201	0002 06		
0202	1257	TAD	A /C = A + B
0203	1260	TAD	B
0204	3261	DCA	C
0205	4033	CALL	1,FLOAT /CONVT TO FP
0206	0103 06		
0207	6201 05	ARG	C
0210	0261 01		
0211	4033	CALL	1,STO
0212	0104 06		
0213	6201 05	ARG	D
0214	0262 01		
0215	4033	CALL	2,WRITE /INIT IO HNDLR
0216	0205 06		
0217	6201 05	ARG	N /DEV NR 1 = TELE
0220	0256 01		
0221	6201 05	ARG	FORMT /FORMAT SPEC
0222	0240 01		
0223	4033	CALL	1,IOH /TYPE INTGER NR
0224	0106 06		
0225	6201 05	ARG	C
0226	0261 01		
0227	4033	CALL	1,IOH /TYPE FP NR
0230	0106 06		
0231	6201 05	ARG	D
0232	0262 01		
0233	4033	CALL	1,IOH /COMPLETE IO
0234	0106 06		
0235	6211	ARG	0
0236	0000		
0237	7402	HLT	

PDP-8 LINKING LOADER DEC-08-A2B3-06

START	01000
OPEN	06125
FLOAT	05034
STO	04444
WRITE	01302
IOH	03142
READ	01271
SETERR	06200
ERROR	06303
TTYOUT	06027
HSOUT	06055
TTYIN	06000
HSIN	06045
FDV	04711
CLEAR	05227

Demonstration Program

14-88

IFAD	05116
FMP	04623
ISTO	05061
FLCT	05153
FAD	04010
DIV	05445
IREM	05616
FSB	04000
FIX	04510
IFIX	04556
CHS	05211
ABS	05636
IABS	05670
MPY	05400
IRDSW	05713
CKIO	06121
EXIT	06142
CLRERR	06231
0004	
0036	

THE ANSWERS ARE 4 4.00



Chapter 15

8K FORTRAN

CONTENTS

Introduction	15-5
FORTRAN Statements	15-6
Statement Numbers	15-7
Line Continuation Designator	15-7
Comments	15-7
Arithmetic Statements	15-8
Character Set	15-8
Constants	15-9
Variables	15-10
Expressions	15-12
Function Calls	15-14
Library Subprograms	15-14
Floating Point Arithmetic	15-16
Control Statements	15-16
GO TO Statement	15-16
IF Statement	15-17
DO Statement	15-17
CONTINUE Statement	15-18
PAUSE, STOP and END Statements	15-19
Input/Output Statements	15-20
FORMAT Statement	15-20
Data Transmission Statements	15-26
READ Statement	15-27
WRITE Statement	15-27
DECTape I/O Routines	15-28
Disk I/O Routines	15-30
Specification Statements	15-33
COMMON Statement	15-33
DIMENSION Statement	15-33
EQUIVALENCE Statement	15-34
Subprogram Statements	15-35

Function Subprograms	15-35
Subroutine Subprograms	15-37
Operating Instructions	15-39
Loading the Compiler	15-39
Operating the Compiler	15-40
Errors	15-40
Compiler Error Messages	15-41
Loading the SABR Assembler	15-42
Operating the SABR Assembler	15-42
Executing the FORTRAN Program	15-46
Demonstration Program	15-46
Statement and Format Specification	15-50
Storage Allocation	15-53
Representation of Constants and Variables	15-53
Storage of Arrays	15-54
Common Storage Allocation	15-56
Implementation Notes	15-57
Implied DO LOOPS	15-57
FORMAT Handling	15-58
Numeric Input Conversion	15-59
Alphanumeric Data Within FORMAT Statements	15-60
Special I/O Devices	15-61

INTRODUCTION

This chapter presents a version of FORTRAN II specifically designed for the PDP-8/I, -8/L, -8, -8S and -5 computers with at least 8K words of core memory, and a Teletype, with an optional high-speed reader and punch.

It is assumed that the reader is familiar with the basic concepts of FORTRAN programming. Several excellent elementary texts are available, such as, "*FORTRAN Programming*", by Frederic Stuart, published by John Wiley and Sons, New York, 1969.

8K FORTRAN (an acronym for FORMula TRANslation) is used interchangeably to designate both the 8K FORTRAN language and the translator or compiler.

The language enables the programmer to express his problem using English words and mathematical statements similar to the language of mathematics and yet acceptable to the computer. The compiler translates the programmer's source program into symbolic language (SABR). The symbolic version of the program is then assembled into (relocatable) binary code, which is the language of the computer. The binary code is output on paper tape and loaded into the computer for execution.

The 8K FORTRAN system has the following features:

1. Subroutines
2. Two levels of subscripting
3. Function subprograms
4. Input/output supervisors
5. Relocatable output adapted to the Linking Loader
6. COMMON statements
7. I, F, E, A, X, and H format specifications
8. Arithmetic and trigonometric library subroutines

The 8K FORTRAN system consists of a one-pass FORTRAN Compiler, the SABR Assembler, the Linking Loader, and a library of subprograms. If the equipment configuration includes a disk and the Disk Monitor System, the Disk Linking Loader (LLDR) should be used to load and execute 8K FORTRAN programs. LLDR is described in Chapter 15.

FORTRAN STATEMENTS

Any FORTRAN statement may appear in the statement field (columns 7 through 72). Each statement must begin on a separate line.

FORTRAN statements are of five types:

1. *Arithmetic*, which define calculations to be performed;
2. *Control*, governing the sequence of execution of statements within a program;
3. *Input/Output*, directing communication between the program and input/output devices;
4. *Specification*, which describe the form and content of data within the program;
5. *Subprogram*, defining the form and occurrence of subprograms and subroutines.

When programs are prepared on-line with the Symbolic Editor, statements are coded following a TAB character (generated by holding down the CTRL key and depressing TAB) or space, unless a statement number is used.

Except for data within a Hollerith field, (see Input/Output Statements), spaces are ignored and may be used freely to organize data into columns for easier reading.

Each statement must start on a separate line. Any statement may be preceded by a positive number of from one to four digits which serves subsequently as an address label.

```

C      THIS PROGRAM WILL SORT AN ARRAY OF NUMBERS
C      INTO ASCENDING ORDER.
C      FIRST READ THE NUMBERS, THEN SORT THE
C      NUMBERS, LAST, WRITE THE NUMBERS IN ORDER.
C
      DIMENSION A(100)
      ND=2
      N=100
      DO 10 I=1,N,2
10     READ (ND,12) A(I),A(I+1)
12     FORMAT (2E12.0)
      DO 30 K=2,N
      J=K-1
20     IF (A(J)-A(J+1)) 30,30,22
22     TEM=A(J)
      A(J)=A(J+1)
      A(J+1)=TEM
      J=J-1
      IF (J) 30,30,20
30     CONTINUE
40     DO 42 I=1,N,2
42     WRITE (ND,44) A(I),A(I+1)
      STOP
44     FORMAT (2E16.8)
      END

```

Figure 16-1 A Sample FORTRAN Program

Statement Numbers

Statement numbers, when used, are coded in columns 2 through 5 of the 72 column line, they are typed preceding the statement, and separated from it by a space. When using the Symbolic Editor, CTRL/TAB causes a jump over the statement number column.

Statement numbers may be assigned non-sequentially, but no two statements can have the same number. Statement numbers are limited to a value of 2047 or less.

Line Continuation Designator

Statements too long for the statement field or a single Teletype line, may be continued. Continued portions of statements may not be given line numbers, and must have an alphanumeric character (other than 0) in column 6. In Teletype input with the Symbolic Editor a digit from 1 to 9 must be used following the CTRL/TAB.

Comments

The letter C in column 1 of a line designates that line as a com-

ment line. A comment has no effect on program compilation, but appears in the program listing. There is no limitation on the number of comment lines which may appear in a given program.

ARITHMETIC STATEMENTS

Constants and *variables*, identified as to type and connected by logical and arithmetic operators form expressions: one or more expressions form an *arithmetic statement*. Arithmetic statements are of the general form

$$V=E$$

where *V* is a variable name (subscripted or nonsubscripted), *E* is an expression, and *=* is a replacement operator. The arithmetic statement causes the FORTRAN object program to evaluate the expression *E* and assign the resultant value to the variable *V*. Note that *=* signifies replacement, not equality. Thus, expressions of the form:

$$A=A+B$$

$$A=A*B$$

are quite meaningful and indicate that the value of the variable *A* is to be changed.

For example:

$$Y=1.1*Y$$

$$P=X**2+3.*X+2.0$$

$$X(N)=EN*ZETA*(ALPHA+EM/PI)$$

The expression value is made to agree in type with the variable before replacement occurs. For example, in the statement:

$$META=W*(ABETA+E)$$

if *META* is an integer and the expression is real, the expression value is truncated to an integer before assignment to *META*.

Character Set

The following characters are used in the FORTRAN language.²

² Appendix B2 lists the octal and decimal representations of the FORTRAN character set.

1. The alphabetic characters, A through Z.
2. The numeric characters, 0 through 9.
3. The special characters:

!	,
"	(
\$)
%	+
&	-
*	/
=	.
#	,
;	<
:	>
?	(space)

Constants

Constants are self-defining numeric values appearing in source statements. Two types of constants, integer and real, are permitted in a FORTRAN source program.

INTEGER CONSTANTS

Integer (fixed point) constants are represented by a digit string of from one to four decimal digits, written with an optional sign, and without a decimal point. An integer constant must fall within the range -2047 to $+2047$. For example:

47	
+47	(+ sign is optional)
-2	
0434	(leading zeros are ignored)
-0	(same as zero)

REAL CONSTANTS

Real constants are represented by a digit string, an explicit decimal point, an optional sign, and possibly an integer exponent to denote a power of ten (7.2×10^3 is written $7.2E+03$). A real constant may consist of any number of digits but only the leftmost eight digits appear in the compiled program. Real constants must fall within the range $.14 \times 10^{-38}$ to 1.7×10^{38} .

+4.50 (+ is optional)
4.50
-23.09
-3.0E14 (same as -3.0×10^{14})

Variables

A variable is a named quantity whose value may change during execution of a program. Variables are specified by name and type. The name of a variable consists of one or more alphanumeric characters the first of which must be alphabetic. Only the first five characters are interpreted as defining the variable name, the rest are ignored.

The type of variable (integer or real) is determined by the first letter of the variable name. A first letter of I, J, K, L, M, or N indicates an integer variable, and any other first letter indicates a real variable. Variables of either type may be either scalar or array variables. A variable is an array variable if it first appears in a DIMENSION statement.

INTEGER VARIABLES

Integer variables undergo arithmetic calculations with automatic truncation of any fractional part. For example, if the current value of K is 5 and the current value of J is 9, J/K would yield 1 as a result.

Integer variables may be converted to real variables by the function FLOAT (see Function Calls) or by an arithmetic statement. Integer variables must fall within the range -2048 to $+2047$.

Integer arithmetic operations do not check for overflow. For example, the sum $2047+2047$ will yield a result of -2 . For more information refer to Chapter 1 of *Introduction to Programming* (Volume 1 in this set) or any text on binary arithmetic.

REAL VARIABLES

A variable is a real variable when its name begins with any character other than I, J, K, L, M, or N. Real variables may be converted to integer variables by the function IFIX (see Function Calls) or by an arithmetic statement. Real variables undergo no truncation in arithmetic calculations.

SCALAR VARIABLES

A scalar variable, which may be either integer or real, represents a single quantity.

For example:

LM
A
G2
TOTAL
J

ARRAY VARIABLES

An array variable represents a single element of a one- or two-dimensional array of quantities. The array element is denoted by the array name followed by a subscript list enclosed in parentheses. The subscript list may be any integer expression or two-integer expressions separated by a comma. The expressions may be arithmetic combinations of integer variables and integer constants. Each expression represents a subscript, and the values of the expressions determine the referenced array element. For example, the row vector A_i would be represented by the subscripted variable $A(I)$, and the element in the second column of the first row of the matrix A , would be represented by $A(1, 2)$.

For example:

$Y(1)$
 $PORT(K)$
 $A(3*K + 2, 1)$

The arrays above (Y , $PORT$, and A) would have to appear in a **DIMENSION** statement prior to their first appearance in an executable statement. The **DIMENSION** statement specifies the number of elements in the array.

Arrays are stored in increasing storage locations with the first subscript varying most rapidly (see Storage Allocation). The two-dimensional array $B(J, K)$ is stored in the following order:

$B(1, 1), B(2, 1), \dots, B(J, 1), B(1, 2), B(2, 2), \dots, B(J, 2),$
 $\dots, B(J, K)$

SUBSCRIPTING

Since excessive subscripting tends to use core memory inefficiently, it is suggested that subscripted variables be used judiciously. For example, the statement:

$A = ((B(I) + C2) * B(I) + C1) * B(I)$

could be rewritten with a considerable saving of core memory as follows:

```
T=B(I)
A=((T+C2)*T+C1)*T
```

Expressions

An expression is a sequence of constants, variables, and function references separated by numeric operators and parentheses in accordance with mathematical convention and the rules given below.

Without parentheses, algebraic operations are performed in the following descending order:

**	exponentiation
-	unary negation
* and /	multiplication and division
+ and -	addition and subtraction
=	equals or replacement sign

Parentheses are used to change the order of precedence. An operation enclosed in parentheses is performed before its result is used in other operations. In the case of operations of equal precedence, the calculations are performed from left to right.

Integers and real numbers may be raised to either integer or real powers. An expression of the form

$$A^{**}B$$

means A^B and is real unless both A and B are integers. Exponential (e^x) and natural logarithmic ($\log_e(x)$) functions are supplied as subprograms and are explained later.

Excluding ** (exponentiation), no two numeric operators may appear in sequence unless the second is a unary plus or minus.

The mode (or type) of an expression may be either integer or real and is determined by its constituents. Variable modes may not be mixed in an expression with the following exceptions:

1. A real variable may be raised to an integer power:

$$A^{**}2$$

2. Mode may be altered by using the functions IFIX and FLOAT:

$$A*\text{FLOAT}(I)$$

The I in example 2, above, indicates an *integer* variable; it is changed to *real* (in floating point format) by the FLOAT function.

Zero raised to a power of zero will yield a result of 1. Zero raised to any other power will yield a zero result. Numbers are raised to integer powers by repetitive multiplication. Numbers are raised to floating point powers by calling the EXP and ALOG functions. A negative number raised to a floating point power will not cause an error message but will use the absolute value. Thus, the expression $(-3.0)**3.0$ will yield a result of +27.

Any numeric expression may be enclosed in parentheses and be considered a basic element.

```
IFIX(X + Y)/2
(ZETA)
(COS(SIN(PI*EM) + X) )
```

A numeric expression may consist of a single element (constant, variable, or function call). For example:

```
2.71828
Z(N)
TAN(THETA)
```

Compound numeric expressions may be formed using numeric operators to combine basic elements. For example:

```
X + 3.
TOTAL/A
TAN(PI*EM)
```

Alphabetic expressions preceded by a + or a - sign are also numeric expressions. For example:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

As an example of a typical numeric expression using numeric

operators and a function call, the expression for the largest root of the general quadratic equation

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

would be coded as

$$(-B + \text{SQRT}(B**2 - 4.*A*C))/(2.*A)$$

Function Calls

In addition to the basic numeric operators, function calls are provided to facilitate the evaluation of functions such as sine, cosine, and square root. A function is a subprogram which acts upon one or more quantities (arguments) to produce a single quantity called the function value. A function call may be used in place of a variable name in any arithmetic expression.

Function calls are denoted by the identifier which names the function (i.e., SIN, COS, etc.) followed by an argument enclosed in parentheses as shown below:

$$\text{IDENT} (\text{ARG}, \text{AGR}, \dots, \text{ARG})$$

where IDENT is the identifying function name and ARG is an argument which may be any expression. A function call is evaluated before the expression in which it is contained.

Library Subprograms

The standard FORTRAN library contains built-in functions, including user defined functions and subroutine subprograms.

Table 2-1 lists the built-in functions. These are open subroutines; they are incorporated into the compiled program each time the source program names them.

Function and subroutine subprograms are closed routines; their coding appears only once in the compiled program. These routines are entered from various points in a program through jump-type linkages.

Table 2-1 Function Library

Name	Call	Definition	Argument
Absolute Value	ABS or IABS	$ X $	Real Integer
Float	FLOAT	Conversion from integer to real	Integer
Fix	IFIX	Conversion from real to integer	Real
Remainder	IREM	Remainder of last integer divide ³	Integer
Exponential	EXP	e^x	Real
Switch Register	IRDSW	Read console switch reg.	Integer
Natural Logarithm	ALOG	$\log_e(x)$	Real
Trigonometric Sine ⁴	SIN	sine(x)	Real
Trigonometric Cosine ⁴	COS	cos(x)	Real
Tangent ⁴	TAN	tan(x)	Real
Square Root	SQRT	$(x)^{1/2}$	Real
Arctangent ⁴	ATAN	arctan(x)	Real

The IRDSW function call (Switch Register) takes the decimal equivalence of the octal integer in the switch register as its result. For example, if the contents of the switch register is 1234 (668 in decimal) when the statement

```
N=IRDSW(0)
```

is executed, the switch register is read and its contents becomes the value of N:

$$N = 668$$

³ If IREM is called as IREM(I/J), the remainder of I/J will be returned. If the argument of IREM does not contain a division, the remainder of the last integer division will be returned.

⁴ Trigonometric functions use radians rather than degrees.

The switch register can be set at either of two times:

1. Before executing the FORTRAN program, after pressing LOAD ADD and before pressing START.
2. During execution of the FORTRAN program, following a PAUSE statement.

Floating Point Arithmetic

In general, floating point arithmetic calculations are accurate to seven digits with the eighth digit being questionable. Subsequent digits are not significant even though several may be typed to satisfy a field width requirement.

The floating point arithmetic routines check for both overflow and underflow. Overflow will cause the EPNT error message to be typed and program execution will be terminated. Underflow is detected but will not cause an error message. The arithmetic operation involved will yield a zero result. The arctangent function is accurate to six decimal places for arguments whose absolute value is greater than .01.

CONTROL STATEMENTS

The control statements GO TO, IF, DO, PAUSE, STOP, and END alter the sequence of statement execution, temporarily or permanently halt program execution, and stop compilation.

GO TO Statement

The GO TO statement has two forms: unconditional and computed.

UNCONDITIONAL GO TO

Unconditional GO TO statements are of the form:

GO TO n

where n is the number of an executable statement. Control is transferred to the statement numbered n.

COMPUTED GO TO

Computed GO TO statements have the form:

GO TO (n_1, n_2, \dots, n_k), J

where n_1, n_2, \dots, n_k are statement numbers and J is a nonsubscripted integer variable. This statement transfers control to the

statement numbered n_1, n_2, \dots, n_k if J has the value $1, 2, \dots, k$, respectively. The index (J in the above example) of a computed GO TO statement must never be zero or greater than the number of statement numbers in the list (in the example above, not greater than k). For example, in the statement:

```
GO TO(20,10,5),K
```

the variable K acts as a switch, causing a transfer to statement 20 if $K = 1$, to statement 10 if $K = 2$, or to statement 5 if $K = 3$.

IF Statement

Numerical IF statements are of the form:

IF (expression) n_1, n_2, n_3

where n_1, n_2, n_3 are statement numbers. This statement transfers control to the statement numbered n_1, n_2, n_3 if the value of the numeric expression is less than, equal to, or greater than zero, respectively. The expression may be a simple variable or any arithmetic expression.

```
IF (ETA)4,7,12  
IF(KAPPA-L(10))20,14,14
```

DO Statement

The DO statement simplifies the coding of iterative procedures. DO statements are of the form:

DO n i = m_1, m_2, m_3

where n is a statement number, i is a nonsubscripted integer variable, and m_1, m_2, m_3 are integer constants or nonsubscripted integer variables. If m_3 is not specified, it is understood to be 1.

The DO statement causes the statements which follow, up to and including the statement numbered n , to be executed repeatedly. This group of statements is called the range of the DO statement. In the example above, the integer variable i is called the index, the values of m_1, m_2, m_3 are, respectively, the initial, terminal, and increment values of the index.

For example:

```
DO 10 I=1,5,2  
DO 20 I=J,K,5  
DO 30 L=I,J,K
```

The index is incremented and tested before the range of the DO is executed. If the terminal value is less than the initial value, the range of the DO will not be executed.

After the last execution of the range, control passes to the statement immediately following the range. This exit from the range is called the normal exit. Exit may also be accomplished by a transfer from within the range.

The range of a DO statement may include other DO statements, provided the range of each contained DO statement is entirely within the range of the containing DO statement. That is, the ranges of two DO statements must intersect completely or not at all. A transfer into the range of a DO statement from outside the range is not allowed.

Within the range of a DO statement, the index is available for use as an ordinary variable. After a transfer from within the range, the index retains its current value and is available for use as a variable.⁵ The values of the initial, terminal, and increment variables for the index and the index of the DO loop may not be altered within the range of the DO statement.

The last statement of a DO loop must be executable, and must not be an IF, GO TO, or DO statement.

Implied DO Loops

See Implementation Notes.

CONTINUE Statement

This is a dummy statement, used primarily as a target for transfers, particularly as the last statement in the range of a DO statement. For example, in the sequence

⁵ The index of a DO loop should not be used as a variable after a normal exit from that DO loop until it has been redefined.

```

DO 7 K=INIT,LIMIT
.
.
.
IF (X(K)) 22,13,7
.
.
.
7 CONTINUE

```

a positive value of X(K) begins another execution of the range. The CONTINUE provides a target address for the IF statement and ends the range of the DO statement.

PAUSE, STOP and END Statements

The PAUSE and STOP statements effect FORTRAN object program operation; the END statement effects assembler operation only.

PAUSE STATEMENT

The PAUSE statement enables the program to incorporate operator activity into the sequence of automatic events. The PAUSE statement assumes one of two forms:

```

                PAUSE
or             PAUSE n

```

where n is an unsigned decimal number.

Execution of the PAUSE statement causes the octal equivalent of the decimal number n, to be displayed in the accumulator on the user's console. Program execution may be resumed (at the next executable statement) by depressing the CONTINUE key on the console.

In some cases the PAUSE statement may be used to give the operator a chance to change data tapes or to remove a tape from the punch. When this is done it is necessary to follow the PAUSE statement with a call to the OPEN subroutine. This subroutine initializes the I/O devices and sets hardware flags that may have been cleared by pressing the tape feed buttons. For example:

```

PAUSE
CALL OPEN

```

STOP STATEMENT

The STOP statement has the form:

STOP

It terminates program execution. STOP may occur several times within a single program to indicate alternate points at which execution may cease. Program control is directed either to or around STOP statements.

END STATEMENT

The END statement is of the form:

END

and signals the compiler to terminate compilation. The END statement must be the last statement of every program.

INPUT/OUTPUT STATEMENTS

Input/Output (I/O) statements are used to control the transfer of data between computer memory and peripheral devices and to specify the format of the output data. I/O statements may be divided into two categories.

1. Nonexecutable FORMAT statements enable conversion between internal data (within core memory) and external data.
2. Data transmission statements, READ and WRITE, specify transmission of data between computer memory and I/O devices.

FORMAT Statement

The nonexecutable FORMAT statement enables the user to specify the form and arrangement of data on the selected external device. (See Implementation Notes for special uses of the FORMAT statement.)

FORMAT statements are of the form:

n FORMAT (S₁, S₂, . . . , S_n)

where n is a statement number and each S is a data field specification.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct input/output transmission, it will be used in

conjunction with the list of a data transmission statement.

During transmission of data, the object program scans the designated FORMAT statement; if a specification for a numeric field is present (see Data Transmission Statements) and the data transmission statement contains items remaining to be transmitted, transmission takes place according to the specification. This process ceases and execution of the data transmission statement is terminated as soon as all specified items have been transmitted. The FORMAT statement may contain specifications for more items than are indicated by the data transmission statement. The FORMAT statement may also contain specifications for fewer items than are indicated by the data transmission statement, in which case, format control will revert to the rightmost left parenthesis in the FORMAT statement.

Both numeric and alphanumeric field specifications may appear in a FORMAT statement. The FORMAT statement also provides for handling multiple record formats, skipping characters, space insertion, and repetition. If an input list requires more characters than the input device supplies for a given unit record, blanks are inserted.

NUMERIC FIELDS

Numeric field specification codes and the corresponding internal and external forms of the numbers are listed in the following table.

Table 16-2 Numeric Field Codes

Conversion Code	Internal Form	External Form
E	Binary floating point	Decimal floating point ⁶ with E exponents: .324E+10
F	Binary floating point	Decimal floating point with no exponent: 283.75
I	Binary integer	Decimal integer: 79

Conversions are specified by the form:

rEw.d
rFw.d
rlw

⁶ When using the WRITE statement with either E or F format, and numbers less than 1.0, a zero will not be typed to the left of the decimal point.

where r is a repetition count, E, F, and I designate the conversion code, w is an integer specifying the field width, and d is an integer specifying the number of decimal places to the right of the decimal point. For E and F input, the position of the decimal point in the external field takes precedence over the value of d. For example:

```
FORMAT (I5,F10.2,E16.8)
```

could be used to output the line

```
32      -17.60      .59625476E+03
```

on the output listing.

The field width should always be large enough to include the decimal point, sign, and exponent. In all numeric field conversions, if the field width is not large enough to accommodate the converted number, the excess digits on the left are lost; if the number is less than the field width, the number is right-justified in the field.

ALPHANUMERIC FIELDS

Alphanumeric data can be transmitted in a manner similar to numeric data by use of the form

rAw

where r is a repetition count, A is the control character, and w is the number of characters in the field. Alphanumeric characters are transmitted as the value of a variable in an input/output list; the variable may be either integer or real.

Although w may have any value, the number of characters transmitted is limited by the maximum number of characters which can be stored in the space allotted for the variable. This maximum depends upon the variable type; for a real variable the maximum is six characters, for an integer variable the maximum is two characters. If w exceeds the maximum, the leftmost characters are lost on input and replaced with blanks on output. If, on input, w is less than the maximum, blanks are filled in to the right of the given characters until the maximum is reached. If, on output, w is less than the maximum, the leftmost w characters are transmitted to the external device.

HOLLERITH CONVERSION

Alphanumeric data may be transmitted directly from the FORMAT statement by using Hollerith (H) conversion. H-conversion format is referenced by WRITE statements only.

In H-conversion, the alphanumeric string is specified by the form

$$nH h_1, h_2, \dots, h_n$$

where H is the control character and n is the number of characters in the string, including blanks. For example, the statement below can be used to print PROGRAM COMPLETE on the output listing.

```
FORMAT(17H PROGRAM COMPLETE)
```

A Hollerith string may consist of any characters capable of representation in the processor. The space character is a valid and significant character in a Hollerith string. (See Implementation Notes for an alternate method of outputting alphanumeric data.)

MIXED FIELDS

A Hollerith format field may be placed among other fields of the format. The statement

```
FORMAT(I5,7H FORCE=F10.5)
```

can be used to output the line:

```
22 FORCE= 17.68901
```

The separating comma may be omitted after a Hollerith format field, as shown above.

REPETITION OF FIELDS

Repetition of a field specification may be specified by preceding the control character E, F, or I by an unsigned integer giving the number of repetitions desired.

```
FORMAT(2E12.4,3I5)
```

is equivalent to

```
FORMAT(E12.4,E12.4,I5,I5,I5)
```

REPETITION OF GROUPS

A group of field specifications may be repeated by enclosing the group in parentheses and preceding the whole with the repetition number.

For example:

```
FORMAT(2I8,2(E15.5,2F8.3))
```

is equivalent to

```
FORMAT(2I8,E15.5,2F8.3,E15.5,2F8.3)
```

MULTIPLE RECORD FORMATS

To handle a group of output records where different records have different field specifications, a slash is used to indicate a new record. For example, the statement

```
FORMAT(3I8/I5,2F8.4)
```

is equivalent to

```
FORMAT(3I8)
```

for the first record and

```
FORMAT(I5,2F8.4)
```

for the second record.

The separating comma may be omitted when a slash is used. When *n* slashes appear at the end or beginning of a format, *n* blank records may be written on output (producing a carriage return/line feed for each record) or ignored on input. When *n* slashes appear in the middle of a format, *n*-1 blank records are written or *n*-1 records skipped. Both the slash and the closing parentheses at the end of the format indicate the termination of a

record. If the list of an input/output statement dictates that transmission of data is to continue after the closing parenthesis of the format is reached, the format is repeated from the last open parenthesis of level one or zero. Thus, the statement:

```
FORMAT(F7.2,(2(E15.5,E15.4),I7))
```

level 0	level 1
level 1	level 0

causes the format:

```
F7.2,2(E15.5,E15.4),I7
```

to be used on the first record, and the format:

```
2(E15.5,E15.4),I7
```

to be used on succeeding records.

As a further example, consider the statement:

```
FORMAT(F7.2/(2(E15.5,E15.4),I7))
```

The first record has the format:

```
F7.2
```

and successive records have the format:

```
2(E15.5,E15.4),I7
```

BLANK OR SKIP FIELDS

Blanks may be introduced into an output record or characters skipped on an input record by use of the specification nX. The control character is X; n indicates the number of blanks or characters skipped and must be greater than zero. For example, the statement:

```
FORMAT(5H STEP I5,10X2HY=F7.3)
```

may be used to output the line:

```
STEP    28                Y=    3.872
```

DATA TRANSMISSION STATEMENTS

There are two data transmission statements, READ and WRITE. Data transmission statements accomplish input/output transfer of data that may be listed in a FORMAT statement. The data transmission statement contains a list of the quantities to be transmitted. The data appears on the external device in the form of records.

1. *Input/Output Lists*⁷—The list of an input/output statement specifies the order of transmission of variable values. During input, the new values of listed variables may be used in subscript or control expressions for variables appearing later in the list. For example:

```
READ(2,1000)L,A(L),B(L+1)
```

reads a new value of L and uses this value in the subscripts of A and B; where 2 is the device designation code, and 1000 is a FORMAT statement number.

2. *Input/Output Records*—All information appearing on input is grouped into records. On output to the printer a record is one line. The amount of information contained in each ANSCII record is specified by the FORMAT reference and the input/output list.

Each execution of an input or output statement initiates the transmission of a new data record. Thus, the statement:

```
READ(1,100)FIRST,SECOND,THIRD
```

is not necessarily equivalent to the statements below where 100 is the FORMAT statement referenced:

```
READ(1,100)FIRST  
READ(1,100)SECOND  
READ(1,100)THIRD
```

⁷ The implied DO in input/output lists is not implemented.

In the second case, at least three separate records are required, whereas, the single statement

READ (d, f) FIRST, SECOND, THIRD

may require one, two, three, or more records depending upon FORMAT statement f.

If an input/output statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another input/output statement without repositioning the record.

If an input/output list requires more than one ANSCII record of information, successive records are read.

READ Statement

The READ statement specifies transfer of information from a selected input device to internal memory, corresponding to a list of named variables, arrays or array elements. The READ statement assumes the following form:

READ (d, f) list

where d is a device designation which may be an integer constant or an integer variable, f is a format reference, and list is a list of variables.

The READ statement causes ANSCII information to be read from the device designated and stored in memory as values of the variables in the list. The data is converted to internal form as specified by the referenced FORMAT statement.

For example:

```
READ(1,15)ETA,PI
```

WRITE Statement

The WRITE statement is used to transmit information from the computer to a specified output device. The WRITE statement assumes one of the following forms:

WRITE (d, f) list

WRITE (d, f)

where d is a device designation (integer constant or integer variable), f is a format reference, and list is a list of variables.

The first form of the WRITE statement causes the values of the variables in the list to be read from memory and written on the device designated in ANSCII form. The data is converted to external form as specified by the designated FORMAT statement.

The second form of the WRITE statement causes information to be read directly from the specified format and written on the device designated in ANSCII form.

DEVICE DESIGNATIONS

The I/O device designations are used in the READ and WRITE statements. The device codes are:

<u>Device Code</u>	<u>Designating</u>
1	Teletype and low-speed reader and punch
2	High-speed reader and punch

For additional I/O information, see SABR, chapter 15.

DECTape I/O Routines

RTAPE and WTAPE (read tape and write tape) are the DECTape read and write subprograms for the 8K FORTRAN and 8K SABR systems. The subprograms are furnished on one relocatable binary-coded paper tape which must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECTape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms which may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

1. DECTape unit number (from 0 to 7)
2. Number of the DECTape block at which transfer is to start. The user may direct the DECTape service routine to begin searching for the specified block in the forward direction rather than the usual backward direction by making this argument the two's complement of the block number. For

additional information on this and other features the reader is referred to the *DECtape Programmer's Reference Manual*.

3. Number of words to be transferred ($1 < N < 4096$).
4. Core address at which the transfer is to start.

The general form is:

```
CALL RTAPE (n1, n2, n3, n4)
```

where n_1 is the DECTape unit number, n_2 is the block number, n_3 is the number of words to be transferred, and n_4 is the starting address.

In 8K FORTRAN, an example CALL statement to RTAPE could be written in the following format (arguments are taken as decimal numbers):

```
CALL RTAPE(6,128,388,LOCA)
```

In this example, LOCA may or may not be in COMMON.

As a typical example of the use of RTAPE and WTAPE, assume that the user wants to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively. Since PDP-8 DECTape is formatted with 1612 blocks (numbered 1-2700 octal) of 129 words each (for a total of 207,948 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively. Each array must be stored beginning at the start of some DECTape block. The user may write these arrays on tape as follows:

```
CALL WTAPE(0,1,2000,A)
CALL WTAPE(0,17,400,B)
CALL WTAPE(0,21,400,C)
CALL WTAPE(0,25,20,D)
```

The user may also read or write a large array in sections by specifying only one DECTape block (129 words) at a time. For example, B could be read back into core as follows.

```
CALL RTAPE(0,17,258,B(1))
CALL RTAPE(0,19,129,B(259))
CALL RTAPE(0,20,13,B(388))
```

As shown above, it is possible to read or write less than 129 words starting at the beginning of a DECTape block. It is impossible, however, to read or write starting in the middle of a block.

For example, the last 10 words of a DECTape block may not be read without reading the first 119 words as well.

A DECTape read or write is normally initiated with a backward search for the desired block number. To save searching time, the user may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least fourteen block numbers greater than the last block number used. For example, if the user has just read array A and now wants array D, he may write:

```
CALL RTAPE(0,1,2000,A)
CALL RTAPE(0,-25,20,D)
```

The following is a section of a program demonstrating the use of DECTape I/O. Assume that values are already present on the DECTape.

```
      DIMENSION DATA(500)
      .
      .
      .
      NB=0
      SUM=0
      DO 100 N=1,10
      CALL RTAPE(1,-NB,1500,DATA)
      TEM=0
      DO 50 K=1,500
50      TEM=TEM+DATA(K)
      SUM=SUM+TEM
100     NB=NB+24
      AMEAN=SUM/5000.
      WRITE(1,110)SUM,AMEAN
      CALL EXIT
110     FORMAT('SUM=',E15.7' MEAN=',E15.7///)
      END
```

Disk I/O Routines

ODISK AND CDISK (open disk and close disk) and RDISK and WDISK (read disk and write disk) are the four DECdisk (DF32/DS32) input and output subprograms for the 8K FORTRAN system. They are furnished on one relocatable binary-coded paper tape which is loaded into core using the Linking Loader, where they occupy eight pages of core.

ODISK AND CDISK

ODISK is used to open (activate) a file (named using the Linking Loader D function) so that the file can be read or written using RDISK or WDISK. CDISK will close (deactivate) a file which was opened with ODISK so that the contents of the file cannot be altered.

The ODISK and CDISK subprograms may be called with standard, explicit CALL statements in any 8K FORTRAN program. ODISK requires one argument when opening a file. However, it requires two arguments when specifying or changing the size (in blocks) of a file. CDISK always requires only one argument.

The first argument of both ODISK and CDISK is the logical number (from 1 thru 10 inclusive) of the file as it was named using the Linking Loader. (Refer to Chapter 15 for a discussion of logical file numbers.) The second argument to ODISK is the number of blocks (from 1 thru 128) to be saved for the file.

In 8K FORTRAN, the CALL statements to ODISK and CDISK are written in the following format (arguments are taken as decimal integer numbers).

```
CALL ODISK(1)
```

when opening a file, or

```
CALL ODISK(1,5)
```

when specifying or changing the size of a file, and

```
CALL CDISK(1)
```

when closing an opened file.

ODISK prepares the file named for data transfer. When running the user program using the Disk Monitor System, ODISK uses Disk Monitor I/O and the three scratch blocks on disk zero for a window whenever a file is opened.

All open files should be closed before terminating program execution, thus preserving the contents of the files.

RDISK AND WDISK

The RDISK and WDISK (read disk and write disk) subprograms may be called with standard, explicit CALL statements in any 8K FORTRAN or 8K SABR program. The ODISK subprogram must be used to open the file concerned before using the RDISK or WDISK subprograms.

Each of these subprograms requires four arguments, arranged as listed below.

1. Logical file number (determined using the Linking Loader D function),
2. Logical block of file number (block number of the file where data transfer is to begin),
3. Number of words to be transferred (from 1 thru 2047), and
4. Core address where data transfer is to start (field 0).

Both RDISK and WDISK require the arguments above. The general form is:

```
CALL RDISK (n1, n2, n3, n4)
```

In 8K FORTRAN, the CALL statements to RDISK and WDISK are written in the following format (arguments are taken as decimal numbers).

```
CALL RDISK(4,2,55,LOCA)
```

when reading file 4, beginning with block 2, transferring 55 words, starting at location of tag LOCA, which may be the name of an array defined in a DIMENSION statement. WDISK would be formatted in the same fashion.

A variable number of words may be transferred. It is not necessary to transfer in 200-word blocks as with the Disk Monitor System.

The sample code written as an example of DECTape I/O usage has been recorded below to demonstrate disk I/O. Again, assume the data is on the disk.

```

        DIMENSION DATA (500)
        .
        .
        .
        CALL ODISK(1)
        NB=0
        SUM=0
        DO 100 N=1,10
        CALL RDISK(1,NB,1500,DATA)
        TEM=0
        DO 50 K=1,500
50      TEM=TEM+DATA(K)
        SUM=SUM+TEM
100     NB=NB+12
        AMEAN=SUM/5000.
        WRITE (1,110)SUM,AMEAN
        CALL CDISK(1)
        CALL EXIT
110    FORMAT('SUM=',E15.7,' MEAN=',E15.7///)
        END

```

SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. The specification statements are **DIMENSION**, **COMMON**, and **EQUIVALENCE**, and when used, must appear in the program prior to any executable statement.

COMMON Statement

The **COMMON** statement causes specified variables or arrays to be stored in an area available to other programs. By means of **COMMON** statements, the data of a main program and/or the data of its subprograms may share a common storage area. Variables in **COMMON** statements are assigned to locations in ascending order in field 1 beginning at location 200 storage allocation. The **COMMON** statement has the general form:

$$\text{COMMON } v_1, v_2, \dots, v_n$$

where v is a variable name.

DIMENSION Statement

The **DIMENSION** statement is used to declare identifiers to be array identifiers and to specify the number and bounds of the array subscripts. The information supplied in a **DIMENSION** statement is required for the allocation of memory for arrays. Any

number of arrays may be declared in a single DIMENSION statement. The DIMENSION statement has the form:

DIMENSION s_1, s_2, \dots, s_n

where s is an array specification. For example:

```
DIMENSION A(100)
DIMENSION Y(10),PORT(25),B(10,10),J(32)
```

NOTE

When variables in COMMON storage are dimensioned, the COMMON statement must appear before the DIMENSION statement.

EQUIVALENCE Statement

The EQUIVALENCE statement causes more than one variable within a given program to share the same storage location. The EQUIVALENCE statement has the form:

EQUIVALENCE (v_1, v_2, \dots), ...

where v is a variable name.

The inclusion of two or more references in a parenthetical list indicates that the quantities in the list are to share the same memory location. For example:

```
EQUIVALENCE(RED,BLUE)
```

specifies that the variables RED and BLUE are stored in the same place, and therefore they have the same value. The subscripts of array variables must be integer constants. For example:

```
EQUIVALENCE(X,A(3),Y(2,1)),(BETA(2,2),ALPHA)
```

Identifiers may not appear in both EQUIVALENCE and COMMON statements.

Because of core memory restrictions within the compiler, variables cannot appear in EQUIVALENCE statements more than once.

```
EQUIVALENCE(A,B,C)
```

would be valid, but the statement

```
EQUIVALENCE(A,B),(B,C)
```

would not compile correctly.

SUBPROGRAM STATEMENTS

External subprograms are defined separately from the programs that call them, and are complete programs which conform to all the rules of FORTRAN programs. They are compiled as closed subroutines, that is, they appear only once in core memory regardless of the number or times they are used. External subprograms are defined by means of the statements FUNCTION and SUBROUTINE. Function and subroutines must be compiled independent of the main program and then loaded together with the main program by the Linking Loader.

Subprogram definition statements contain dummy identifiers, representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram and indicate the sort of arguments that may appear and how the arguments are used. The dummy identifiers are replaced by the actual arguments when the subprogram is executed.

Function Subprograms

A function subprogram is a single-valued function that may be called by using its name as a function name in an arithmetic expression, such as FUNC(N), where FUNC is the name of the subprogram that evaluates the corresponding function of the argument N. A function subprogram begins with a FUNCTION statement and ends with an END statement. It returns control to the calling program by means of one or more RETURN statements.

The FUNCTION statement has the form:

```
FUNCTION identifier (a1, a2, . . . , an)
```

This statement declares that the program which follows is a function subprogram. The identifier is the name of the function being defined. This identifier must appear as a scalar variable and be assigned a value, which is the function value, during execution of the subprogram.

Arguments appearing in the list enclosed in parentheses are *dummy arguments* representing the function argument. The arguments must agree in number, order and type with the actual arguments used in the calling program. Function subprograms may have expressions and array names as arguments.

Dummy arguments may appear in the subprogram as scalar identifiers or array identifiers. A function must have at least one dummy argument. Dummy arguments representing array names must appear within the subprogram in a DIMENSION statement. Dimensions must be indicated as constants and should be smaller than or equal to, the dimensions of the corresponding arrays in the calling program.

A function should not modify any arguments which appear in the FORTRAN arithmetic expression calling the function. The only FORTRAN statements not allowed in a function subprogram are SUBROUTINE and other FUNCTION statements.

The type of function is determined by the first letter of the identifier used to name the function, in the same way as variable names.

The following short example calculates the gross salary of an individual on the basis of the number of hours he has worked (TIME) and his hourly wage (RATE). The function calculates time and a half for overtime beyond 40 hours. The function is called SUM and would look as follows:

```
FUNCTION SUM(TIME,RATE)
  IF (TIME-40.) 10,10,20
10  SUM = TIME * RATE
   RETURN
20  SUM = (40.*RATE) + (TIME-40.)*1.5*RATE
   RETURN
END
```

Depending upon which path the program takes, control will return to the main program at one of the two RETURN statements with the answer. We would probably have set up the main program with a statement to read the employee's weekly record from a list of information prepared on the high-speed reader:

```
READ(2,5) NAME, NUM, NDEP, TIME, RATE
```


This statement reads the person's name, number, department number, time worked, and hourly wage. The main program would then go on to calculate his gross pay with a statement like the following:

```
GROSS = SUM(TIME,RATE)
```

and go on to calculate withholdings, etc.

Subroutine Subprograms

A subroutine subprogram may be multivalued and can be referred to only by a CALL statement. A subroutine subprogram begins with a SUBROUTINE statement and returns control to the calling program by means of one or more RETURN statements.

The SUBROUTINE statement has the form:

```
SUBROUTINE identifier (a1, a2, . . . . , an)
```

This statement declares the program which follows to be a subroutine subprogram. The first identifier is the subroutine name. The arguments in the list enclosed in parentheses are dummy arguments representing the arguments of the subprogram. The dummy arguments must agree in number, order, and type with the actual arguments used by the calling program.

Subroutine subprograms may have expressions and array names as arguments. The dummy arguments may appear as scalar or array identifiers.

Dummy identifiers which represent array names must be dimensioned within the subprogram by a DIMENSION statement. The dummy arguments must not appear in an EQUIVALENCE or COMMON statement in the subroutine subprogram.

A subroutine subprogram may use one or more of its dummy identifiers to represent results. The subprogram name is not used for the return of results. A subroutine subprogram need not have any arguments, or may use the arguments to return numbers to the calling program. Subroutines are generally used when the result of a subprogram is not a single value.

Example SUBROUTINE statements may look as follows:

```
SUBROUTINE FACTOR (COEFF,N,ROOTS)
SUBROUTINE RESIDU(NUM,N,DEN,M,RES)
SUBROUTINE SERIES
```

The only FORTRAN statements not allowed in a subroutine subprogram are FUNCTION and other SUBROUTINE statements.

The following short subroutine takes two integer numbers from the main program and exchanges their values. If this is to be done at several points in the main program it is a procedure best performed by a subroutine.

```
SUBROUTINE ICHANGE(I,J)
ITEM = I
I = J
J = ITEM
RETURN
END
```

The calling statement for this subroutine might look as follows:

```
CALL ICHANGE(M,N)
```

where the values for the variables M and N would be exchanged.

CALL STATEMENT

The CALL statement assumes one of two forms:

```
CALL identifier
or CALL identifier (a1, a2, . . . , an)
```

The CALL statement is used to transfer control to a subroutine subprogram. The identifier is the subroutine name.

The arguments (indicated by a₁, through a_n) may be expressions or array identifiers. Arguments may be of any type, but must agree in number, order, type, and array size with the corresponding arguments in the SUBROUTINE statement of the called subroutine. Unlike a function, a subroutine may produce more than one value and cannot be referred to as a basic element in an expression.

A subroutine may use one or more of its arguments to return results to the calling program. If no arguments at all are required, the first form is used. For example:

```
CALL EXIT
CALL TEST (VALUE,123,275)
```

The identifier used to name the subroutine is not assigned a type and has no relation to the types of the arguments. Arguments which are constants or formed as expressions must not be modified by the subroutine.

RETURN STATEMENT

The RETURN statement has the form:

RETURN

This statement returns control from a subprogram to the calling program. Each subprogram must contain at least one RETURN statement. Normally, the last statement executed in a subprogram is a RETURN statement; however, any number of RETURN statements may appear in a subprogram. The RETURN statement may not be used in a main program.

OPERATING INSTRUCTIONS

The Compiler, SABR Assembler, and Linking Loader are used (in that order) to compile, assemble, and execute FORTRAN programs. In carrying out the following procedures, the Data Field setting can be ignored since all system tapes, with the exception of Linking Loader, have field settings coded on them. For detailed information on the Linking Loader see Chapter 15, SABR.

Loading the Compiler

PAPER TAPE SYSTEM

1. Make sure the Binary Loader is in memory, say field 1.
2. Place the FORTRAN Compiler binary tape in the reader.
3. Set the console switches as follows: (Data field is ignored)
instruction field = 1, Switch Register = 7777.
4. Press LOAD ADDRESS.
5. Depress Switch Register bit 0.
6. Press START
7. The FORTRAN Compiler has now been loaded into memory by the Binary Loader. Parts of the compiler will load into field 0 and field 1.

DISK MONITOR SYSTEM

1. Make sure the Disk Monitor is in memory. (Type CTRL/C⁴ or START at 7600).
2. When the Monitor responds with a dot, call the system loader by typing
.LOAD (the . denotes typing the RETURN key)
3. Place the Compiler binary tape in the reader.
4. Answer the Loader command dialogue as follows:

*IN-R:

*

*ST =

↑ <CTRL/P> ↑ <CTRL/P>

5. The FORTRAN Compiler has now been loaded into memory, parts into field 0 and field 1. It must now be saved on the system device as follows:

- SAVE FTC0!0-7577;5363
- SAVE FTC1!200,1000-1577,2600,6000-16377;

6. The compiler has now been saved on the user's system device and may be called as follows:

- FTC1
- FTC0

The field 1 part must be called first.

Operating the Compiler

It is assumed that the programmer has written his main program and possibly one or more subprograms, and that these source programs have been punched on paper tape in ANSCII format. Remember that each source tape must have an END statement at the end of the tape.

After the compiler has been loaded into memory, it is used to translate each FORTRAN statement into one or more SABR assembler instructions. The compiler output will be punched in two parts separated by approximately three feet of blank tape. The first part, (executable code) will be punched as the source tape is read. The second part, (variable storage and constants) will be punched after the entire source tape has been read.

⁴ CTRL/C and CTRL/P are typed by holding the CTRL key while typing the C or P key. They do not echo (print) when typed, therefore, their presence are indicated by being enclosed in angle brackets.

If the compiler has been saved on the Disk Monitor System, it will halt after it is loaded into memory. Be sure that the source tape has been placed in the reader and the punch has been turned ON, then simply press CONTInue to initiate compiler output.

It may be desirable to suppress all compiler output the first time a particular program is compiled, simply to check for errors. To do this it is necessary to load the compiler and then deposit 3075 in location 0356 (field 0), prior to starting the compiler.

1. Set the console switches as follows: Data field = 0, Instruction field = 1 Switch Register = 1000. (The compiler may also be started at location 5364 in field 0.)
2. Place the FORTRAN program source tape in the reader, and press the punch ON.
3. Press LOAD ADDRESS and START.
4. As soon as the compiler has typed out an identification number, it will begin compiling the user's program. The compiler output will generally be several times the length of the FORTRAN source program.

Errors

All compile time, assembly time, and execution time errors are fatal (the program will not be further processed). For this reason it is desirable to suppress punched output of the compiler and assembler until the source program is believed to be correct. For specific instructions refer to the appropriate System User's Guide.

Note especially that the compiler will not detect undefined statement numbers. Therefore it is important to examine the assembly symbol table for undefined symbols before loading and executing the program.

Do not attempt to load or run a program which has assembly errors. Do not attempt to proceed after an execution time error by pressing CONTInue. Unpredictable results will be obtained in either case.

Compiler Error Messages

When an error is encountered during compilation of a statement, the incorrect statement and an error message are printed. Further compilation of that statement is terminated, and output is suppressed for the rest of the compilation. The compiler, however, will scan the remaining statements for errors, and will print an

error message for any errors found.

An example of an error message follows:

```
A=B+M(6)+N(1)
      ↑
MIXED MODE EXPRESSION
```

Note that an ↑ was printed directly below the incorrect statement. This indicates that the error occurred somewhere between the point and the beginning of the statement. In some cases the arrow may point directly at the illegal character or word, but this cannot always be assumed.

If an error occurs in the middle of a series of continuation lines, all remaining lines in that statement will be printed with the error message ILLEGAL CONTINUATION.

Compiler error messages are self-explanatory:

- ARITHMETIC EXPRESSION TOO COMPLEX
- EXCESSIVE SUBSCRIPTS
- ILLEGAL ARITHMETIC EXPRESSION
- ILLEGAL CONSTANT
- ILLEGAL CONTINUATION
- ILLEGAL EQUIVALENCING
- ILLEGAL OR EXCESSIVE DO NESTING
- ILLEGAL STATEMENT
- ILLEGAL STATEMENT NUMBER
- ILLEGAL VARIABLE
- MIXED MODE EXPRESSION
- SYMBOL TABLE EXCEEDED
- SYNTAX ERROR (usually illegal punctuation)

If an error is discovered in the user's FORTRAN program, the compiler will type the incorrect line, followed by an error message. Although compiler output will be suppressed, the rest of the user's program will be read, and additional error messages may be typed.

When the compiler has finished punching both sections of tape it will halt. It may be restarted to compile additional programs by pressing CONTInue.

The FORTRAN Compiler may be retarted at any time by pressing STOP and resetting the console switches.

Loading The SABR Assembler

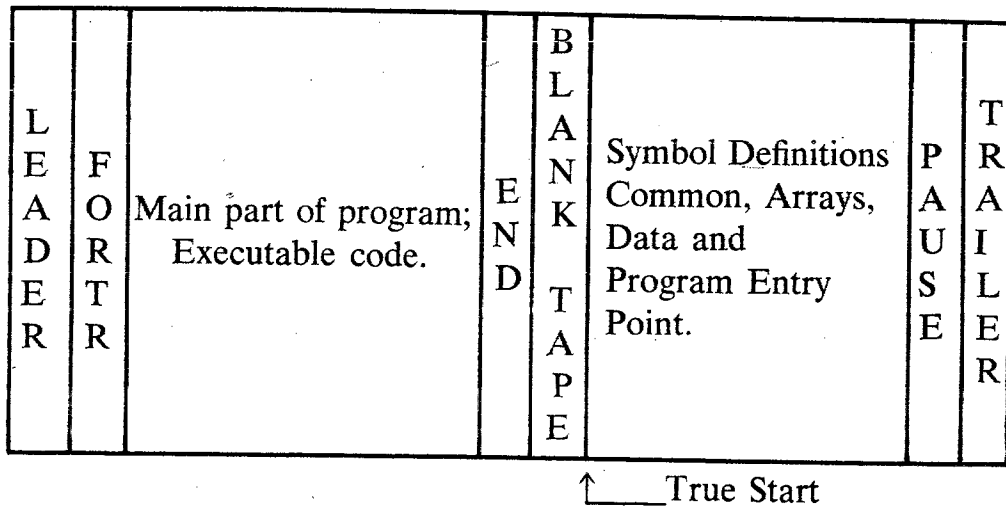
See Chapter 14, or the appropriate System User's Guide for directions on loading the SABR Assembler.

SABR is loaded into memory, partly in Field 0 and partly in Field 1. It may be saved on the user's system device by responding to the monitor's dot as follows:

Operating The SABR Assembler

In addition to being a stand-alone assembler, SABR also serves as the second pass of 8K FORTRAN compilation. For this purpose the use of SABR is slightly different from that described in Chapter 14. This difference in the operation of SABR is due only to the unusual format of the FORTRAN compiler output.

The compiler, in one pass, converts the user's FORTRAN source program into a symbolic machine language program tape. SABR then converts the symbolic tape into relocatable binary. However, the symbolic tape produced by the compiler is not a standard format SABR language tape. It is arranged as shown in the figure on the facing page.



The tape is arranged this way because the data at the end of the tape cannot be inserted in the midst of the executable code, and some of it which should be at the beginning of the tape is not known until later. Thus the true start of the symbolic program is near the end of the symbolic tape preceded by a segment of blank tape and followed by a PAUSE statement.

To assemble such a tape with SABR, one of three methods must be followed. Actually, the general procedure is the same as that described in the SABR manual, but in particular details it differs.

The differences are covered by the three methods explained below.

METHOD 1

The simplest method is to cut the symbolic tape into two parts. The cut should be made at the middle of the blank tape which separates the executable code from the symbol definitions. The latter section of the tape should then be marked "Section 1" and the former section (the executable code) should be marked "Section 2." Assembly then proceeds with the two part symbolic tape exactly as described below.

After SABR has been loaded into memory, it is used to assemble the compiler output. In the first pass through SABR the relocatable binary version of the user's program is created and, at the end of this pass, the symbol table may be typed and/or punched. Pass 2 is the listing pass. The assembly is carried out as follows.

If SABR has been saved on the system I/O device it will start automatically at step (3) on the next page when called into memory. The source tape (first section) should be inserted in the reader before operation begins.

It may be desirable to suppress all assembler output the first time a particular program is assembled, simply to check for errors. To do this it is necessary to load SABR and then deposit 5370 in location 3165 (Field 0) before beginning step (1) below.

1. Set the console switches as follows: Data field = 0, Instruction field = 0, Switch Register = 0200.
2. Press LOAD ADDRESS and START.
3. SABR now types a sequence of two or three questions;

HIGH SPEED READER?
HIGH SPEED PUNCH?
LISTING ON HIGH SPEED PUNCH?

These questions must be answered with "Y" if the answer is "yes." Any other answer is assumed to be "no." The third question is typed only if the second is answered "Y." If the third is answered "Y," both the symbol table and the listing will be punched on the high-speed paper tape punch. Otherwise, they are typed on the teletypewriter. Incidentally, the user need not wait for the full question to be typed before responding.

4. As soon as SABR has echoed the user's response to the last question, the punch device and, if it is being used, the ASR

- reader should be turned on. If using the low-speed reader, the error message E indicates that the user has waited too long before turning the reader on. He will have to start over.
5. At this point, pass 1 begin. SABR reads the source tape and punches the binary tape. After the binary tape has been completed SABR will type or punch the program symbol table.
 6. If the source tape is in several sections (separate tapes with PAUSE at the end of all except the last), SABR will halt at the end of each section. At this point the user should insert the next section in the reader and then press CONTInue.
 7. At the end of Pass 1 SABR will halt.
 8. If the user desires an assembly listing, he should now reposition the beginning of the source tape in the reader and press CONTInue.

If the listing is going to be punched on the high speed punch, the user may want to list the symbol table (at the end of the binary relocatable tape) before beginning Pass 2.

9. At the end of Pass 2 SABR will again halt. It may be restarted for assembling another program by pressing CONTInue.
10. SABR may be restarted at any time by pressing STOP, setting the switch register =0200, pressing LOAD ADDRESS and START. However, Pass 1 must always be repeated.

METHOD 2

The user may avoid actually cutting the symbolic tape by manipulating the tape as if it were two parts, as explained above. The tape should initially be inserted in the reader with the separator blank tape over the read-head. When SABR halts at the PAUSE statement at the physical end of the tape, the user should reposition the tape, putting the physical beginning of the tape in the reader. Then press CONTInue. The assembly pass will end at the separator blank tape code. The assembly listing can be produced in a similar manner, pressing CONTInue to start the listing pass.

METHOD 3

The third method requires SABR to pass the symbolic tape two times for each pass of the assembly. However, it allows the tape to be inserted at its physical beginning. It is based on the fact that

a symbolic tape output by the FORTRAN Compiler has as its physical first line the special pseudo-op, FORTR. This pseudo-op has no effect except when a symbolic tape output by the compiler is assembled using this third method.

The method is this:

1. Insert the symbolic tape in the reader at its physical beginning.
2. Start SABR as usual.
3. Sensing the FORTR statement as the first line, SABR ignores all further data until after it passes over the END statement. SABR then begins the actual assembly by processing the symbol definitions, etc., which are at the latter end of the tape.
4. Then SABR halts at the PAUSE statement which is at the physical end of the tape. At this time the user should reposition the symbolic tape in the reader at the physical beginning of the tape, and then press CONTINUE. SABR will now assemble the executable code portion of the tape in the normal way.
5. If the user desires an assembly listing, he should proceed as in Method 2 after SABR finishes the assembly pass.

One other type of error may occur. This is an undefined symbol. Because SABR is a one-pass assembler, this can not be determined until the end of the assembly pass, so the error diagnostic UNDF is given in the symbol table listing.

The Linking Loader

See Chapter 14.

Executing The FORTRAN Program

Determine the starting address of your main program by using the Linking Loader Storage Map option. The address will be typed in the form:

MAIN dnnnn

1. Set Data Field = d, Instruction Field = d, Switch Register = nnnn.

2. Turn on paper tape punch and/or put data tape in reader as required.
3. Press LOAD ADDRESS, and START.
Program execution will begin.

DEMONSTRATION PROGRAM

This program computes the factorials of the even integers from 1 through 34. The MAIN program calls the subprogram to perform the computation.

This demonstration program was run on a PDP-8/I with 8K words of core memory and high-speed reader/punch. The demonstration, from start to finish, required 15 minutes. Actual Teletype printout is used on the following pages.

Both source programs were written using the Symbolic Editor, listed on the Teletype for inclusion here, and punched on the high-speed punch.

```

C      FORTRAN DEMONSTRATION PROGRAM
      DIMENSION A(35)
      DO 10 N=2,34,2
      A(N)=FACT(N)
10     WRITE (1,60)N,A(N)
      STOP
60     FORMAT (I3,4H! = ,E12.7)
      END

P
L

C      FORTRAN FUNCTION TO COMPUTE FACTORIALS
      FUNCTION FACT(N)
      IF (N-34) 1,5,5
1     IF (N) 2,4,2
2     M=N-2
      FACT=N
      DO 3 K=1,M
      C=N-K
3     FACT=FACT*C
      RETURN
4     FACT=1.
      RETURN
5     WRITE (1,6) N
      FACT=0
      RETURN
6     FORMAT (I5,30H! EXCEEDS CAPACITY OF PROGRAM.)
      END

P

```

This is the system program tape identification. At this point we have loaded the FORTRAN Compiler and compiled both source programs.

PDP-8 FORTRAN DEC-08-A2B1-3

PDP-8 SABR DEC-08-A2B2-10
HIGH SPEED READER? Y
HIGH SPEED PUNCH? Y
LISTING ON HIGH SPEED PUNCH? N

CKIO	0000EXT
FACT	0000EXT
IOH	0000EXT
ISTO	0000EXT
MAIN	0352EXT
OPEN	0000EXT
SUBSC	0000EXT
WRITE	0000EXT
[0	0512
\A	0200
\N	0351
\10	0426
\60	0501
!A	0361
!B	0473
!C	0411
!D	0450
!E	0463
!F	0476
!G	0512

HIGH SPEED READER? Y
HIGH SPEED PUNCH? Y
LISTING ON HIGH SPEED PUNCH? N

FACT	0215EXT
FAD	0000EXT
FLOT	0000EXT
FMP	0000EXT
IOH	0000EXT
OPEN	0000EXT
STO	0000EXT
WRITE	0000EXT
[0	0473
\C	0205
\FACT	0201
\K	0204
\M	0200
\N	0471
\1	0251
\2	0261
\3	0331

\4	0354
\5	0406
\6	0445
13	0210
†A	0305
†B	0346
†C	0422
†D	0471

Loaded the SABR Assembler, responded to the initial dialogue and assembled both compiled programs.

Load the Library programs using the Linking Loader. Set the switch register for the memory map.

DEC-08-A2B3-5

MAIN	01152
OPEN	10325
SUBSC	11000
FACT	01415
ISTC	06062
WRITE	02066
IOH	03744
CKIO	10321
FLOT	06200
STO	05444
FAD	05010
FMP	05623
READ	02055
SETERR	10400
ERROR	10503
TTYOUT	10227
HSOUT	10255
TTYIN	10200
HSIN	10245
FDV	05711
CLEAR	06237
IFAD	06117
DIV	06443
IREM	06616
FSB	05000
FLOAT	06034
FIX	05510
IFIX	05556
CHS	06221
ABS	06636
IABS	06700
MPY	06400
IRDSW	06723
EXIT	10344
CLRERR	10431
0003	
0032	

Load the relocatable binary tapes and start the MAIN program at location 01152 (see memory map).

2! = .2000000E+01
 4! = .2400000E+02
 6! = .7200000E+03
 8! = .4032000E+05
 10! = .3628800E+07
 12! = .4790016E+09
 14! = .8717829E+11
 16! = .2092279E+14
 18! = .6402374E+16
 20! = .2432902E+19
 22! = .1124001E+22
 24! = .6204484E+24
 26! = .4032915E+27
 28! = .3048883E+30
 30! = .2652529E+33
 32! = .2631308E+36
 34! EXCEEDS CAPACITY OF PROGRAM.
 34! = .0000000E+00

End of expected program output.

STATEMENT AND FORMAT SPECIFICATIONS

Table 16-3 Statement Specifications

STATEMENT	FORM		WHERE
	(R or P indicates a required or prohibited statement number, N indicates a nonexecutable statement)		
COMMENT	NP	"C" in column 1	columns 2 through 80 will be ignored.
CONTINUE		CONTINUE	control goes to next statement.
ARITHMETIC		v=e	variable name= expression.
GO TO		GO TO n	n is a statement number.
		GO TO (n ₁ , ..., n _m), i	1 ≤ i ≤ m and control goes to statement n _i . i is a unsubscripted integer variable.
IF		IF (E) n ₁ , n ₂ , n ₃	control goes to n ₁ if $\begin{matrix} n_1 \\ \vee \\ n_2 \\ \vee \\ n_3 \end{matrix}$ expression E = 0.

Table 16-3. Statement Specifications (Cont.)

STATEMENT		FORM	WHERE
DO		DO n i=m ₁ , m ₂ , m ₃	repeated execution through statement n beginning with i=m ₁ , incrementing by m ₃ , while i is less than or equal to m ₂ . m's and i may not be subscripted.
		DO n i=m ₁ , m ₂	m ₃ assumed to be 1.
PAUSE		PAUSE	temporary halt, resumed by CONTInue key.
		PAUSE n	octal equivalent of the integer n displayed.
STOP		STOP	must be used to halt execution of a main program.
		STOP n	octal equivalent of the integer n displayed.
END	NP	END	an END statement at the end of a subprogram tells the compiler there is no more program.
READ WRITE		READ (d, f) l WRITE (d, f) l	d is device number, f is a FORMAT statement number and l is list of variable names separated by commas.
FORMAT	NR	FORMAT (k ₁ , . . . , k _n)	k's are format specifications
COMMON	NP	COMMON a, b, . . . , n	a, . . . , n are nonsubscripted variable names
DIMENSION	NP	DIMENSION a ₁ (k ₁), . . . , a _n (k _n)	a's are array names and k's are maximum subscripts.
FUNCTION	NP	FUNCTION name (a ₁ , . . . , a _n)	a's are dummy arguments and name must be defined as a variable containing the value of the function.
SUBROUTINE	NP	SUBROUTINE name (a ₁ , . . . , a _n)	a's are dummy arguments and name may not appear elsewhere in the subroutine.

Table 16-3 Statement Specifications (Cont.)

STATEMENT	FORM	WHERE
CALL	CALL name (a_1, \dots, a_n)	a's are actual arguments of a subroutine and may be expressions.
RETURN	RETURN	for subroutines, control returned to statement following CALL. For functions, evaluation of expression in calling program is resumed using value of the function.
EQUIVALENCE	NP EQUIVALENCE (v_1, \dots, v_n), ..., (v_m, \dots, v_n)	v's are variables or subscripted array names.

Table 16-4. FORMAT Specifications

KIND	FORM	WHERE
Integer	rIw	r is the repetition count; w is total field width in characters.
Floating Point (Decimal)	rFw.d	r is the repetition count, w is field width including sign and decimal point, and d is number of characters to right of decimal point.
Exponential	rEw.d	r is the repetition count, w is field width including sign, decimal point, and d is the number of characters in exponent.
Alphanumeric	rAw	r is the repetition count, w is field width.
H (Hollerith or Literal)	nHcharacters 'characters'	n is total number of characters following H. Parentheses in each format statement must balance. Characters enclosed within single quotes (SHIFT/7) are also printed.
Parentheses	n (specification)	format specification in parentheses is repeated n times.
Carriage Control		indicates beginning of a new data record.

STORAGE ALLOCATION

Representation of Constants and Variables

INTEGERS

Integers are each allocated one machine word. They are represented in two's complement binary.

$\boxed{01 \quad \quad \quad 11}$

sign . Two's complement magnitude

Positive numbers in two's complement binary are represented as straight binary with the first bit zero.

$\boxed{0 \ 11 \ 111 \ 111 \ 111}$

$3777_8 = +2047_{10}$, the largest positive integer.

Negative numbers are represented by replacing each 0 bit with a 1 and each 1 bit with a 0, then adding 1 to the binary result.

+1 is

$\boxed{0 \ 00 \ 000 \ 000 \ 001}$

-1 is $\boxed{1 \ 11 \ 111 \ 111 \ 110} + 1 = \boxed{1 \ 11 \ 111 \ 111 \ 111} = 7777_8$

The largest negative number is -2048 which is represented by 4000_8 or

$\boxed{1 \ 00 \ 000 \ 000 \ 000}$

REAL NUMBERS

Real numbers are each allocated three machine words. They are represented as a binary mantissa multiplied by 2 raised to a binary exponent:

Word 1

$\boxed{0 \ 1 \quad \quad \quad 8 \ 9 \quad \quad 11}$

sign exponent mantissa

Word 2

$\boxed{0 \quad \quad \quad \quad \quad \quad \quad 11}$

mantissa

Word 3

$\boxed{0 \quad \quad \quad \quad \quad \quad \quad 11}$

mantissa

The sign of the number is bit 0 of word 1 (0=+, 1=-). The value and sign of the exponent are obtained by subtracting $1\ 000\ 000_2$ (or 200_8) from bits 1 through 8 of word 1.

Example 1

110 000 001 100
-0-
-0-

Sign: 1_2
 Exponent: $10\ 000\ 0001_2$
 Mantissa: $.100_2$
 Exponent = $201_8 - 200_8 = 1_8$
 Mantissa = $.4_8$
 No. = $-.4_8 \times 2_8^1$
 $= -1/2 \times 2 = -1$

Example 2

010 000 101 100
-0-
-0-

Sign: 0_2
 Exponent: $10\ 000\ 101_2$
 Mantissa: $.1_2$
 Mantissa = $.4_8$
 Exponent = $205_8 - 200_8 = 5_8$
 No. = $.4_8 \times 2_8^5$
 $= 1/2 \times 32 = 16$

Storage of Arrays

Array variables are stored in core according to USA Standards, in columns and from top to bottom. For example, the array IJ

DIMENSION IJ(5)

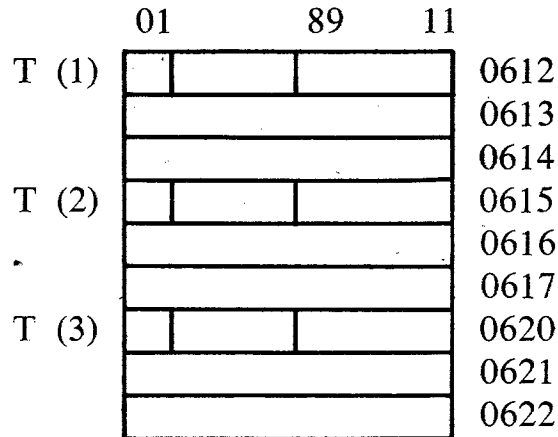
if started at location 0705 would be stored:

	01	11	
IJ (1)			0705
IJ (2)			0706
IJ (3)			0707
IJ (4)			0710
IJ (5)			0711

The real array, T

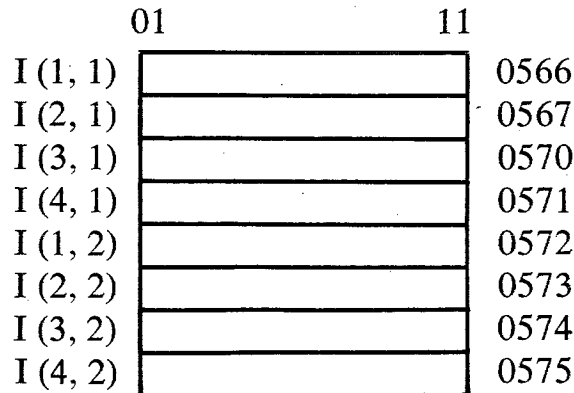
DIMENSION T(3)

starting in location 0612 would appear:



Two-dimensional arrays are stored as shown below.

DIMENSION I(4,2)



In the array $A(M(J, K))$, M is a two-dimensional integer array stored as indicated above. No element of M may be less than 1.

If the element $M(3, 4)$ contains the integer 7, then $A(M(3, 4))$ will be evaluated as $A(7)$. The largest integer stored in M must not exceed the dimensions of A .

REPRESENTATION OF N-DIMENSIONAL ARRAYS

Although arrays of more than two dimensions are illegal, the values of the subscripts of larger arrays may be calculated by using the following algorithm:

$$i_1 + D_1 * (i_2 - 1) + D_1 * D_2 * (i_3 - 1) + \dots + D_1 * D_2 * \dots * D_n * (i_n - 1)$$

where the subscript values are i_1, i_2, \dots, i_n in an array whose dimensions are D_1, D_2, \dots, D_n .

Subprograms may be written to compute and insert subscript values in such illegal arrays. For example, in an array $A(3, 4, 5)$, the following subprogram inserts the value of element $A(N1, N2, N3)$:

```

        DIMENSION ARRAY (60)
        READ (1,5) N1,N2,N3,VALUE
        I=N1+3*(N2-1)+3*4*(N3-1)
        ARRAY(I)=VALUE
5       FORMAT (3I1,F5.3)
        END
    
```

Common Storage Allocation

Common storage begins in absolute location 200 in field 1. Variables are assigned locations in the common storage area in ascending order as they appear in COMMON statements.

For example:

```

COMMON A,J,K
DIMENSION A(2,2),J(4)
    
```

would be stored as follows.

		200
A(1, 1)		201
		202
		203
A(2, 1)		204
		205
		206
A(1, 2)		207
		210
		211
A(2, 2)		212
		213
J(1)		214
J(2)		215
J(3)		216
J(4)		217
K		220

NOTE

K does not appear in a DIMENSION statement.

If the COMMON statement of another subprogram defines

```
COMMON J  
DIMENSION J(5)
```

J(1) through J(5) will be assigned to locations 200 through 204 respectively, thus overlapping the variables A(1, 1) and A(2, 1). The Loader is not aware of this, therefore it is advisable to make COMMON statements identical in all subprograms in which they appear.

However, the statements

```
COMMON DUMMY, J  
DIMENSION DUMMY(2, 2), J(4)
```

would not produce overlapping and could be used in subprograms. In the example above, DUMMY is an arbitrary variable which need not be used in the subprogram.

IMPLEMENTATION NOTES

Implied DO Loops

Because of core memory restrictions, 8K FORTRAN does not have implied DO loops in READ and WRITE statements. However, a simple way to circumvent this restriction has been implemented. Normally a carriage return/line feed (CR/LF) is produced at the end of each WRITE statement. The CR/LF can easily be suppressed by terminating the WRITE statement with a comma. The CR/LF can be generated explicitly in one of two ways:

1. By using a WRITE (d, f) instruction.
2. By using a FINI pseudo instruction.

The second method is more efficient since it generates only four words of code, whereas the first method will generate somewhat more than that. For example, the following statements:

```
DO 10 J=1, M  
10 WRITE (1, 20) (A(J, K), K=1, N)  
20 FORMAT (10F7.3)
```

which are not legal in 8K FORTRAN, could be written as follows:

```
DO 15 J=1,M
DO 10 K=1,N
10 WRITE (1,20) A(J,K),
15 WRITE (1,20)
20 FORMAT (F7.3)
```

or

```
DO 15 J=1,M
DO 10 K=1,N
10 WRITE (1,20) A(J,K), ,
15 FINI
20 FORMAT (F7.3)
```

The second method is preferred for more efficient utilization of core memory. Note that it is not necessary to specify a repetition count in the FORMAT statement since the I/O handler initializes itself to the beginning of the FORMAT statement each time the WRITE statement is executed.

FORMAT Handling

For more complicated FORMAT handling a somewhat different technique can be used. For example:

```
WRITE (1,20) (A(K),K=1,N)
20 FORMAT (F7.2,2E15.6)
```

which again is not legal in 8K FORTRAN, could be written as follows:

(comma suppresses CR/LF)

```
WRITE (1,20),
DO 10 K=1,N
10 CALL IOH(A(K))
FINI
20 FORMAT (F7.2,2E15.6)
```

In the example above, the statement WRITE (1, 20), generates the following assembly code:

```
CALL2,WRITE
ARG <1
ARG \20
```

The statement CALL IOH (A(K)) will generate code to call the subscripting routine SUBSC and will then generate the following code:

```
CALL 1, IOH  
ARG I0
```

where [0 is a temporary location generated by the compiler. Finally the FINI pseudo instruction will generate the following:

```
CALL 1, IOH  
ARG 0
```

which will cause execution of the WRITE statement to be completed.

Although only WRITE statements have been shown in the previous examples, the same techniques apply equally to READ statements. To read in an array of arbitrary size, one might use the following FORTRAN IV statements

```
DO 15 I=1, M  
15 READ (ID, 100) (A(I, J), J=1, N)  
100 FORMAT (F5.2, F5.0, 2F5.2, 2F5.0)
```

This will not work with 8K FORTRAN, but the correct results can be obtained using the following.

```
DO 15 I=1, M  
READ (ID, 100)  
DO 10 J=1, N  
10 CALL IOH(A(I, J))  
15 FINI  
100 FORMAT (F5.2, F5.0, 2F5.2, 2F5.0)
```

Numeric Input Conversion

In general, numeric input conversion is compatible with most other FORTRAN processors. A few exceptions are listed below:

1. Blanks are ignored except to determine in what field digits

fall. Thus numbers are treated as if they were right justified within a field. In an F5.2 format, the following:

```
12
12
.12
00012
```

would be read as the number 0.12.

2. A null line delimited by two CR/LFs will be treated as a line of blanks, and blanks will be appended to the right of a line (if necessary) to fill out a FORMAT statement. Thus:

```
30      READ(1,30)A,B
        FORMAT (4HA = ,F7.2/4HB = ,F7.2)
```

would all be identical under an F5.2 format. If an entire line is blank, numeric data from that line will be read as zeros.

3. No distinction is made between E and F format on input. Thus

```
100.
100E2
1.E2
10000
```

would all be read identically under either an F5.2 or E5.2 format.

Alphanumeric Data Within FORMAT Statements

Alphanumeric data may be transmitted directly from the FORMAT statement by two different methods: H-conversion or the use of single quotes .

Hollerith (H) format is used to output data only. An attempt to use H format specifications with a READ statement will cause characters from the format field to be either typed or punched. This may occasionally be a useful feature since it provides a simple way of identifying data that is to be read from the Teletype. For example, the following instructions:

```
FORMAT ('PROGRAM COMPLETE')
```


would cause A = and B = to be typed out before the data was read.

The same effect is achieved by merely enclosing the alphanumeric data in single quotes. The result is the same as in H-conversion; on input, the characters between the single quotes are replaced by input characters, and, on output, the characters between the single quotes (including blanks) are written as part of the output data. For example, when referred to from a WRITE statement,

```
50 FORMAT ('PROGRAM COMPLETE')
```

would cause PROGRAM COMPLETE to be printed. This method eliminates the need to count characters.

Special I/O Devices

I/O can be performed on devices other than Teletype and high-speed paper tape reader and punch in several different ways:

1. If it is desired to use other devices in place of the high-speed paper tape reader and punch, rewrite the Utility library subroutine defining the entry points for the desired input and output devices as HSIN and HSOUT respectively. The source tape for the utility subroutine is available from the program library and is very short. Refer to Chapter 14 for more information.
2. If it is desired to input or output on a special device but not in ANSCII format write a subroutine to handle the particular device in the SABR assembly language. For more information refer to Chapter 14.
3. If it is desired to add additional devices which can be used with READ and WRITE statements, then edit part I of the Library Subroutine IOH. New entries must be made in the device transfer table at the beginning of IOH. Copies of this source tape and listings of the library subroutines are available from the program library. The service routines for the additional I/O devices must be written in SABR assembly language and can then be assembled along with the revised version of IOH.
4. Programs written in SABR language can call PAL subroutines in various ways:
 - a) A JMS 7000 instruction will call a PAL program which starts at location 7000 in the same memory field.

- b) A CONTINUE (or PAUSE) statement might be inserted in the user's FORTRAN program. Then JMS to the PAL subroutine may be inserted using the switch register.

It is possible to load any size PAL III program linkage with an 8K FORTRAN program by merely dimensioning an integer variable to the proper size for the PAL III program. This offers two advantages, virtually unlimited size programs in PAL III can be linked to 8K FORTRAN main programs, and none of the library routines are disturbed by this linkage.

Chapter 16

Floating Point & Math Routines

TABLE OF CONTENTS

Floating Point System	16-5
Floating Point Representation	16-5
Storage	16-6
Normalization	16-6
Instruction Set	16-7
Interpreter	16-9
Floating Point Package Versions	16-10
Assembly Instructions	16-11
Input Routine	16-11
Overflow	16-13
Output Routine	16-14
Output Controller	16-15
User Subroutines	16-15
Summary of Basic Package	16-16
Floating Point Algorithms	16-17
Fixed to Floating/Floating to Fixed	16-19
Extended Floating Point Package	16-21
Using Extended Functions	16-24
Math Routines	16-25
Implementation Notes	16-27
Tape Format	16-27
Signs	16-28
External Calls	16-28
Program Control	16-28
Errors	16-29

FLOATING POINT SYSTEM

The floating point system maintains a constant number of significant digits throughout programmed computations and performs its own input, arithmetic and output operations.

Floating point notation is particularly useful for computations involving numerous multiply and divide operations where operand magnitudes may vary widely. The system stores very large and very small numbers by saving only the significant digits and computing an exponent to account for leading and trailing zeroes.

Floating Point Representation

A floating point number consists of a *mantissa* and an *exponent* such that the mantissa, multiplied by the base (radix) of the number system in use, raised to a power as supplied in the exponent, gives the value of the number in fixed point notation.

As an example, in fixed point notation the number twelve can be represented as

12

or

12.0

In floating point notation with radix of 10, the number might appear as

$.12 \cdot 10^2$

where the mantissa is .12 and the exponent, 2.

A fraction, such as twelve ten-thousandths, is represented as

.0012

in fixed-point notation. It appears in floating point representation as

$.12 \cdot 10^{-2}$

the minus sign before the exponent indicating that the significant digits of the mantissa are to be shifted right from the decimal point.

BINARY NOTATION

With radix of two, the decimal number twelve is represented as

$$1100$$

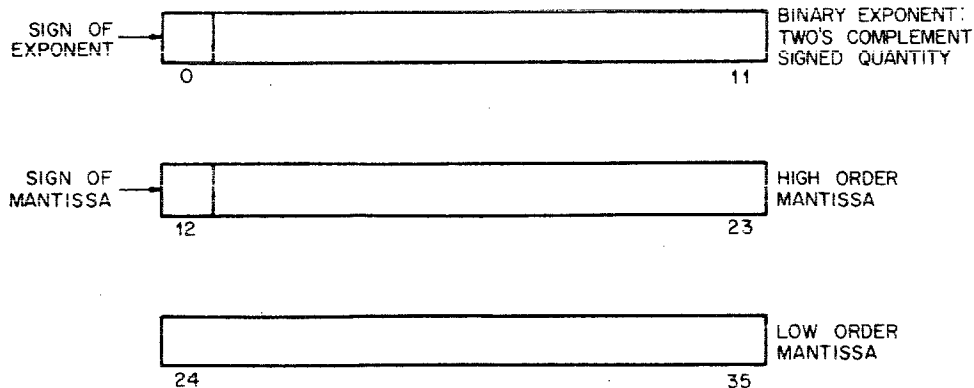
Its representation in floating point format is, logically,

$$.1100 \cdot 2^4$$

In computer operations there is no necessity for keeping track of the base; all operations are in binary. Multiplication and division are accomplished by shift operations: each one-place shift to the left represents multiplication by two; each equivalent shift to the right represents division by two (see Introduction to Programming, Chapter 1). Thus, the binary exponent is used to keep track of the number (and direction) of shift operations necessary to correctly scale the mantissa.

Storage

The floating point system utilizes three consecutive locations, registers 44, 45 and 46, to store the exponent and double precision mantissa.



These registers form the floating pseudo accumulator.

Normalization

In computing a mantissa from decimal input the system uses the convention:

$$\frac{1}{2} \leq |\text{MANTISSA}| < 1.$$

The input value is said to be *normalized* when scaled in this manner. The value of the number is then:

$$\text{MANTISSA} \cdot 2^{\text{EXPONENT}}$$

where the MANTISSA is a signed quantity.

Instruction Set

The basic floating-point operations are:

- Load floating accumulator
- Store floating accumulator
- Add to floating accumulator
- Subtract from floating accumulator
- Multiply by floating accumulator
- Divide into floating accumulator
- Normalize floating accumulator

The floating-point instructions are:

<u>Op Code</u>	<u>Mnemonic</u>	<u>Effect</u>
1	FADD	Floating Addition Add the contents of the effective address to the floating accumulator.
2	FSUB	Floating Subtract Subtract the contents of the effective address from the floating accumulator.
3	FMPY	Floating Multiply Multiply the floating accumulator by the contents of the effective address.
4	FDIV	Floating Divide Divide the floating accumulator by the contents of the effective address.
5	FGET	Floating Get Load the floating accumulator with the contents of the effective address.

<u>Op Code</u>	<u>Mnemonic</u>	<u>Effect</u>
6	FPUT	Floating Put Store the contents of the floating accumulator at the locations specified by the effective address. The contents of the floating accumulator are unchanged.
7	FNOR	Floating Normalize Normalize the contents of the floating accumulator.
0	is decoded as follows:	
	Bits 8-11 = 0000	Floating Exit Return control to following instruction.
	= 0001	Floating Square Square the contents of the floating accumulator.
	= 0010	Floating Square Root Take the root of the absolute value of the floating accumulator.
	= 0011 - 1111	Expandable commands

The PAL III and MACRO assemblers recognize all of these mnemonics except SQUARE and SQROOT, which may be defined as:

SQUARE = 0001
SQROOT = 0002

NOTE

The PAL-D disk assembler does not recognize these Floating point mnemonics. Thus the mnemonics FADD through FPUT must be fixed as memory reference instructions, and the mnemonics for FNOR through SQROOT must be defined as user symbols in any program to be assembled with PAL-D.

Floating point operations are summarized by the following expressions.

FADD	Y; 1000;	$C(FAC) + C(Y) \rightarrow C(FAC)$	} Result is normalized	} C(Y) unchanged
FSUB	Y; 2000;	$C(FAC) - C(Y) \rightarrow C(FAC)$		
FMPY	Y; 3000;	$C(FAC) \times C(Y) \rightarrow C(FAC)$		
FDIV	Y; 4000;	$C(FAC) \div C(Y) \rightarrow C(FAC)$		
FGET	Y; 5000;	$C(Y) \rightarrow C(FAC)$		
FPUT	Y; 6000;	$C(FAC) \rightarrow C(Y)$		
FNOR	; 7000;	$C(FAC) \text{ normalized} \rightarrow C(FAC)$		
FEXT	; 0000;	exit from interpreter to instruction following this command		
SQUARE	; 0001;	$C(FAC)^2 \rightarrow C(FAC)$		
SQROOT	; 0002;	$C(FAC)^{1/2} \rightarrow C(FAC)$		

Interpreter

All arithmetic operations are called through an interpreter. The interpreter contains, at all times, the address of the memory location containing the next pseudo instruction to be executed. This is initially stored when the program enters the interpreter using an effective JMS 5600.

When the interpreter encounters an instruction with an op code of 0 and with bits 8-11 of the pseudo instruction equal to 0, it exits to the next memory location.

Example:

```

SQROOT=0002
*7
5600
*2000
      JMS I    7
      FGET   A
      SQROOT
      FPUT I  B
      FEXT
      HLT
A,    0003
      2000
      0000
B,    300
$

```

When this program is started at 0200, it will halt at location 205. The state of the machine will be:

44/	0002	
45/	2000	floating accumulator contains
46/	0000	$(\frac{1}{2}) \cdot 2^2$ or 2.0
206/	0003	
207/	2000	register A contains 4.0
210/	0000	
300/	0002	
301/	2000	answer stored here
302/	0000	

FLOATING POINT PACKAGE VERSIONS

The four versions of the floating-point package are:

PACKAGE 1—DEC-08-YQ1B-PB

This is the basic floating-point package, consisting of the basic input/output routines and basic arithmetic instructions. Its core limits are:

7; 40-61; 5600-7577

PACKAGE 2—DEC-08-YQ2B-PB

This is the basic package, but with the output modified by the output controller to allow for formatting of output. Its core limits are:

7; 40-62; 5400-7577

PACKAGE 3—DEC-08-YQ3B-PB

This is the basic package plus the extended functions. Its core limits are:

7; 40-61; 5100-7577

PACKAGE 4—DEC-08-YQ4B-PB

This is the basic package plus the output controller and the extended functions. Its core limits are:

7; 40-62; 4700-7577

In all versions of the floating-point package, the input and output routines may be called interpretively (see User Subroutines).

Assembly Instructions

The various versions of the floating point package are assembled from the floating point source tapes by the following methods:

PACKAGE 1—(DEC-08-YQ1B-PB)—Assemble together, using PAL-3, source tapes 1 (Basic I/O) and 4 (Interpreter) in that order.

PACKAGE 2—(DEC-08-YQ2B-PB)—Assemble together, using PAL-3, source tapes 2 (Basic I/O plus Controller) and 4 (Interpreter), in that order.

PACKAGE 3—(DEC-08-YQ3B-PB)—Assemble together, using PAL-3, source tapes 1 (Basic I/O), 3 (Extended Functions) and 4 (Interpreter) in that order.

PACKAGE 4—(DEC-08-YQ4B-PB)—Assemble together, using PAL-3, source tapes 2 (Basic I/O plus Controller) 3 (Extended Functions), and 4 (Interpreter), in that order.

NOTE

Assembly of PACKAGES 3 and 4 yields five PAL-III RD diagnostics—one each for the symbols PSINF, PCOSF, PATANF, PEXPF, and PLOGF. These symbols are used to insert the proper address constants for the extended functions into the interpreter's extended op code calling table—see User Subroutines.

INPUT ROUTINE

The basic floating-point package contains an input routine to read characters from the Teletype keyboard. Input format is floating decimal which is converted to floating-point binary format. The number 726.7 may be typed in any of the following forms:

726.7
.7267E3
.7267E+03
+7267E-1
etc.

Input is terminated when a character is typed that is not a part of a format. The conversion of "12.0." would be terminated on the second "."

The input routine is entered with an effective JMS 7400. It returns control to the instruction following the calling JMS upon receipt of a terminator. The floating accumulator contains the input number in normalized floating-binary. Register 0057 contains the terminating character in ANSCII, and C(0060) indicates whether or not there was a valid input.

Example:

```

*5
7400
*7
5600
*200
      JMS I    5           /INPUT ROUTINE
      JMS I    7           /CALL FLOATING POINT
      FPUT     A
      FEXT
      JMS I    5
      JMS I    7
      FPUT     B
      FEXT
      HLT
A,    0
      0
      0
B,    0
      0
      0
.s

```

When this program is started at 0200 and the following is typed:

X2.0Y

The program will halt at location 0210, and A and B will contain

```

A,  0
    0
    0   and C(57) = 0331, the second TERMINATOR
B,  0002
    2000
    0000

```

The first input was considered a 0 because an "X" terminator was used initially.

This program could be written to ignore the non-numeric information as follows:

```

*5          7400
*7          5600
*200
    JMS I    5          /CALL INPUT ROUTINE
    TAD     60         /ANY VALID INPUT?
    SNA CLA
    JMP     .-3        /NO - IGNORE
    JMS I    7          /YES
    FPUT    A          /STORE IT
    FEXT
    JMS I    5          /GET NEXT
    TAD     60
    SNA CLA          /VALID?
    JMP     .-3        /NO - IGNORE
    JMS I    7          /YES
    FPUT    B          /STORE IT
    FEXT
    HLT
A,          0
           0
           0
B,          0
           0
           0
S

```

Register 57 may be used for integrating control characters into the input. Register 56 is a switch that has the following meaning:

If $C(56) = 0$, do not type a line-feed after a carriage-return is read.
 If $C(56) \neq 0$, type a line-feed when a carriage-return is input.
 This switch is initially set to 7777.

Overflow

The input conversion routine halts on overflow during calculation of the mantissa. Typing RUBOUT and pressing CONTINUE on the console will restart the routine.

Overflow during exponent calculation yields an unpredictable result; the package may halt in which case it may be restarted as above.

Capacity of the input routine is approximately from .999999E-615 to .999999E+615.

The overflow halt may be eliminated by depositing 7000 (NOP) into location 7564. If this is done or if the user continues from the overflow halt without typing RUBOUT, the contents of the floating accumulator will be unpredictable.

RUBOUT

If RUBOUT is struck before an input delimiter, the input routine is restarted and previous numbers ignored.

RUBOUT
276 ↓ 1T

The input routine exits with 1 in the floating accumulator.

OUTPUT ROUTINE

The output routine is entered with an effective JMS 7200. Upon entry the contents of the floating accumulator are converted to floating-point decimal and typed out in the following format:

± 0.XXXXXXXE ± XX

If the floating accumulator contained:

44/ 0002
 2000
 0000

and the output routine were entered,

+ 0.200000E + 01

would be typed. Control returns to the instruction following the calling JMS instruction. The contents of the floating accumulator are lost.

The floating-output routine has a switch in location 0055 on page 0: if not equal to 0, carriage-return/line feed follows each printout (initial value 7777.)

Output scaling errors can cause a floating point number, stored

correctly in core, to be output incorrectly. Users should preserve the core image of data to be used in further calculations in a save area, using the FPUT instruction. Subsequent routines reference this save area for needed data.

The save area can also be punched, via ODT or a binary punch routine, and loaded when needed.

The range of the output conversion routine is approximately the same as that of the input routine.

Output Controller

This is an additional routine that modifies the basic output routine to enable it to format output. It is incorporated in floating point packages 2 and 4.

The controller is called, like the basic output routine, by an effective JMS 7200. It requires two pre-set parameters:

$C(62)$ = total number of digits to be output. If $C(62) = 0$, output in E format. $C(62)$ is initially set to 0.

$C(AC)$ = number of digits to the right of the decimal point. If $C(AC) = 0$, do not type a “.”

If the number in the FAC is larger than the field width allows the sign of the number is typed and the field filled with “X’s”; if smaller, the field is filled with spaces.

User Subroutines

When the floating-point interpreter encounters a 0 op code, it further decodes bits 8-11. If these bits equal 0, the interpreter exits. If the bits are nonzero, they are used in a table look-up to specify the address of a subroutine. The called subroutine may use the floating-point interpreter, the input, or the output, but it may not use bits 8-11 to call another subroutine.

For example, the interpreter has a subroutine to negate the floating accumulator. Its entry point is 6000. If the negate subroutine were to be called by 0010 in the interpreter mode, 6000 would be placed in address 6554 of the calling table. Input and output could be called in the interpreter mode if 7400 were placed in 6555 (0011) and 7200 were placed in 6556 (0012).

```

NEGATE=0010
INPUT=0011
OUTPUT=0012
SQUARE=0001
*7
5600
*200
      KCC
      TLS
      JMS I    7      /ENTER INTERPRETER
      INPUT    /CALL INPUT ROUTINE
      SQUARE   /SQUARE IT
      NEGATE   /CALL OUTPUT ROUTINE
      OUTPUT   /CALL OUTPUT ROUTINE
      FEXT     /EXIT
      HLT      /HALT

```

. S

To avoid timing errors the user must pattern his typeout routines after that in the floating point packages:

```

      TSF
      JMP  .-1
      TLS

```

This routine waits for the Teletype flag to be set before executing the TLS instruction.

Summary of Basic Package

ENTRY POINTS

```

5600   Arithmetic Interpreter
7400   Floating Input
7200   Floating Output

```

NOTE

Both the input and the output routines require that register 7 contain the interpreter entry point: 5600.

FLAGS

```

55     If ≠ 0, type carriage-return/line-feed after output.
56     If ≠ 0, follow each input carriage-return by a line-feed.
57     Contains the input terminating character.
60     Equals 0 if no valid input.
61     Is nonzero if (a) divide by 0 or (b) square root of a
       negative number.

```

COMMANDS

FADD	1000	Floating Add
FSUB	2000	Floating Subtract
FMPY	3000	Floating Multiply
FDIV	4000	Floating Divide
FGET	5000	Floating Get
FPUT	6000	Floating Put
FNOR	7000	Floating Normalize
FEXT	0000	Floating Exit
SQUARE	0001	Square
SQROOT	0002	Square Root
	0003	} Expandable
	
	0017	

STORAGE

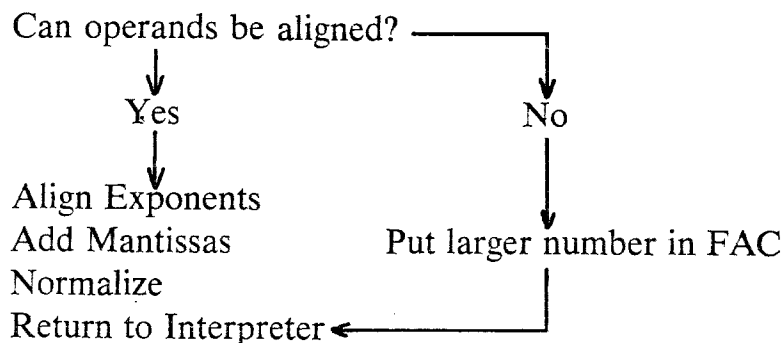
The floating accumulator is in registers:

- 44 – Exponent
- 45 – High Order Mantissa
- 46 – Low Order Mantissa

Floating Point Algorithms

ADDITION

Floating-point addition is carried out by first aligning the binary points of the two numbers. This is accomplished by scaling the smaller number to the right. Then the mantissas are both scaled right once so that overflow will not occur into the sign bit. A 2's complement addition of the mantissas is then made. The result is normalized and control returns to the interpreter. This may be represented as:



SUBTRACTION

Floating-point subtraction is accomplished by negating the operand, and then calling the addition subprogram.

- Negate Operand
- Call Addition

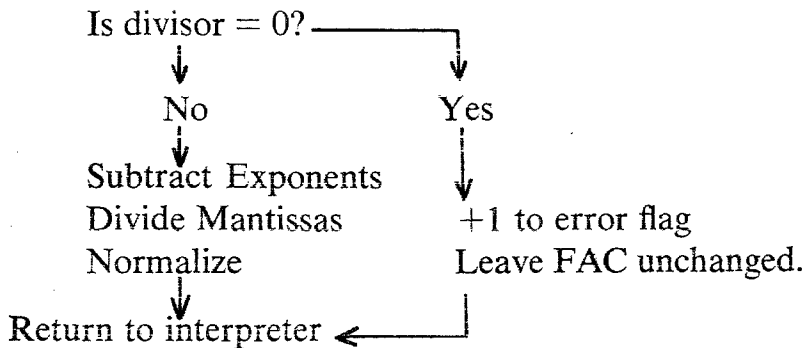
MULTIPLICATION

Floating-point multiplication is accomplished by adding the exponents together and then performing a double-precision multiplication. The result is normalized and control returns to the interpreter.

- Add Exponents
- Multiply Mantissas carrying result to 35 bits
- Normalize
- Return to Interpreter

DIVISION

Floating-point division is accomplished by subtracting the exponent of the divisor from the exponent of the dividend. The mantissa is divided and the result is normalized. Control returns to the interpreter.



SQUARE

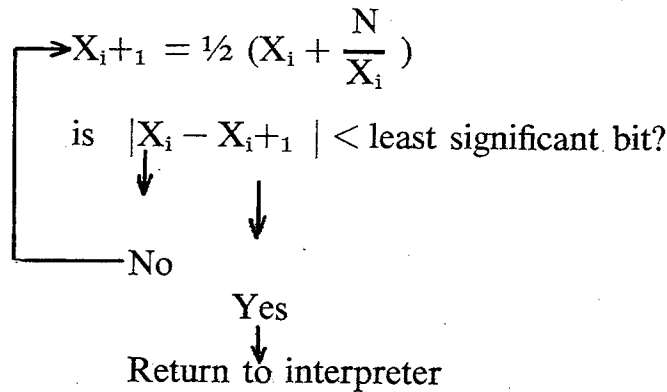
The square routine calls the multiplication routine internally.

SQUARE ROOT

The square root is calculated using Newton's method in which an initial approximation is made and then each succeeding approximation is calculated. The routine exits when two successive approximations are equal to within the least significant bit of the mantissas.

Form first approximation, X_1 , of \sqrt{N} :

then:



ERROR FLAG

Division by 0 causes C(61) to be incremented by 1. The FAC is unchanged.

Attempting to extract the square of a negative number causes C(61) to be incremented by 1. The root of the absolute value is taken.

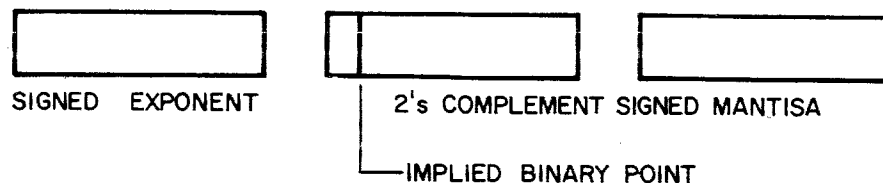
The contents of 61 are set to 0 at the beginning of each square root operation (but not at the beginning of each divide operation).

EXPONENT UNDERFLOW

The FAC is set to zero on the occasion of exponent underflow.

Fixed to Floating/Floating to Fixed

Since the floating-point package stores numbers in the following format:



and since the exponent indicates where the real binary point is, this information may be used to convert a fixed point number to floating point or a floating point number to fixed point.

For example, assume that there is an integer in the accumulator

that is less in magnitude than 2047. To float this number, the following sequence of steps may be employed:

```

...
DCA      45      /PUT INTO HIGH-ORDER MANTISSA
DCA      46      /PUT 0 INTO LOW-ORDER MANTISSA
TAD      C13     /11 (10) INTO
C,       DCA      44      /EXPONENT
        JMS I    7       /CALL INTERPRETER
        FNOR     /NORMALIZE
        FEXT     /LEAVE INTERPRETER
...
C13,     0013    /11 (DECIMAL)

```

At point C, we have set the binary point of the integer to the right end of the high order mantissa word, or eleven (decimal) locations to the right of the implicit binary point.

To float this number, the floating accumulator is scaled right until the exponent contains the location of the desired binary point. To fix the floating accumulator as an 11-bit signed integer, the following sequence of coding may be employed:

```

...
CLA
TAD      44      /FETCH EXPONENT
SZA SMA  /IS THE NUMBER<1?
JMP      .+3     /NO:
CLA      /YES: FIX IT TO 0
JMP      DONE+1
TAD      M13    /NO: SET BINARY POINT AT
SNA      /11(10) PLACES TO RIGHT OF CURRENT POINT
JMP      DONE   /IT IS ALREADY THERE: ALL DONE
SMA      /TEST TO SEE IF IT IS TOO LARGE
JMP      ERROR  /YES: NUMBER >2**11
DCA      44     /NO: SET SCALE COUNT
GO,     CLL     /0 TO C(L)
        TAD      45     /FETCH MANTISSA
        SPA      /IS IT <0?
        CML     /YES: PUT A 1 IN LEFT BIT
        RAR     /SCALE RIGHT
        DCA      45     /RESTORE IT
        ISZ     44     /TEST IF SHIFTED ENOUGH
        JMP     00     /NO: CONTINUE
DONE,   TAD      45     /ANSWER IN C(AC)
...
...
M13,   -13     /-11 (DECIMAL)

```

This may be coded as a subroutine.

EXTENDED FLOATING POINT PACKAGE

The extended floating point package provides the following additional operations:

- 0003 *Sine*: take the trigonometric sine of the floating accumulator (radian measure); return the result in the floating accumulator.
- 0004 *Cosine*: take the trigonometric cosine of the floating accumulator (radian measure); return the result in the floating accumulator.
- 0005 *Arctangent*: compute the angle whose tangent is in the floating accumulator; return the result (radian measure) in the floating accumulator.
- 0006 *Exponential*: compute $e(=2.71828 \dots)$ raised to the power contained in the floating accumulator; return the result in the floating accumulator.
- 0007 *Logarithm*: take the natural logarithm (base= $2.71828 \dots$) of the positive number in the floating accumulator; return the result in the floating accumulator (N.B.—attempts to take the logarithm of a negative or zero argument yield an error halt).

SINE

The sine routine makes use of the identity:

$$\text{SIN}(X) = \text{COS}\left(\frac{\pi}{2} - X\right)$$

and subsequently calls the cosine routine.

COSINE

The argument is first brought within the range:

$$-\frac{\pi}{2} < X \leq \frac{\pi}{2}$$

by use of the identity:

$$\text{COS}(X) = -\text{COS}(X - \pi)$$

The cosine of the resulting angle is computed using a sufficient number of terms of the series:

$$\text{COS}(X) = 1 - \frac{X^2}{2!} + \frac{X^4}{4!} - \frac{X^6}{6!} + \dots$$

ARCTANGENT

The arctangent is computed using a sufficient number of terms of the appropriate series, depending on the sign and magnitude of the argument:

$$\text{ATAN}(X) = X - \frac{X^3}{3} + \frac{X^5}{5} - \frac{X^7}{7} + \dots \quad (X^2 \leq 1)$$

$$\text{ATAN}(X) = \frac{\pi}{2} - \frac{1}{X} + \frac{1}{3X^3} - \frac{1}{5X^5} + \dots \quad (X > 1)$$

$$\text{ATAN}(X) = -\frac{\pi}{2} - \frac{1}{X} + \frac{1}{3X^3} - \frac{1}{5X^5} + \dots \quad (-1 > X)$$

EXPONENTIAL

The exponential is computed using a sufficient number of terms of the series:

$$\text{EXP}(X) = 1 + X + \frac{X^2}{2!} + \frac{X^3}{3!} + \frac{X^4}{4!} + \dots$$

LOGARITHM

The logarithm (natural) of the argument is computed using a sufficient number of terms of the imbedded series:

$$\text{LOG}(X) = 2 \left[\frac{X+1}{X+1} + \frac{1}{3} \frac{X-1}{X+1} + \frac{1}{5} \frac{X-1}{X-1} + \dots \right] \quad (X > 0)$$

SAMPLE PROGRAM

Input is A and B. Output is $Y = \text{LOG}(\text{COS}(A/B) + \sqrt{A \cdot B})$


```

SQROOT=0002          /DEFINITIONS TO ASSEMBLER
COS=0004
LOG=0007
*5
7400
7200
5600
/USES EXTENDED INTERPRETER

```

```

*200
KCC
TLS
BEGIN,  JMS I 5      /INPUT A
        JMS I 7      /ENTER INTERPRETER
        FPUT A      /STORE A
        FEXT        /EXIT
        JMS I 5      /INPUT B
        JMS I 7      /ENTER INTERPRETER
        FPUT B      /STORE B
        FMPY A      /A.B
        SQROOT      /EXTRACT ROOT
        FPUT TEMP   /STORE IT
        FGET A      /LOAD FAC WITH A
        FDIV B      /DIVIDE BY B
        COS         /TAKE COSINE
        FADD TEMP   /ADD
        LOG         /TAKE LOG.
        FEXT
        JMS I 6      /OUTPUT ANSWER
        JMP BEGIN
A,      0
        0
        0
B,      0
        0
        0
TEMP,   0
        0
        0
$

```

Using Extended Functions

Using extended functions the previous sample program can be rewritten as:

```
SQROOT=2
COS=4
LOG=7
INPUT=13
OUTPUT=14
*7
5600
*200
KCC
TLS
BEGIN,   JMS I    7           /CALL INTERPRETER
         INPUT   /READ A
         FPUT    A         /STORE 1T
         INPUT   /READ B
         FPUT    B
         FMPY    A         /A.B
         SQROOT
         FPUT    TEMP     /(A.B)**.5
         FGET    A
         FDIV    B
         FCOS
         FADD    TEMP
         FLOG
         OUTPUT
         FEXT
         JMP     BEGIN
A,       0
         0
         0
B,       0
         0
         0
TEMP,    0
         0
         0
S
```

Extended Function Operating Domains

SINE, COSINE

Trigonometric sine and cosine are designed for:

$$\text{argument} < 10^3$$

No guarantee is made for arguments which lie outside this range; the user is moreover cautioned that as the argument becomes large compared with $\pi/2$, significance will be lost in the result.

ARCTANGENT

Trigonometric sine and cosine are designed to operate for:

$$\text{argument} < 10^{300}$$

No guarantee is made for arguments which lie outside this range.

EXPONENTIAL

Exponential is designed to operate for:

$$\text{argument} < 75 \text{ (approximately)}$$

No guarantee is made for arguments which lie outside this range.

LOGARITHM

Naperian (or natural) logarithm is designed to operate with all positive arguments. Negative or zero arguments result in an error halt, at Loc. 4715 (Package 4) or 5115 (Package 3). Continuing from the halt will yield unpredictable results. Storing 5700 in place of the halt will cause LOG to return the argument when the argument is negative or zero. Because of the methods used, some significance will be lost for arguments very close to $+ 1.0$.

MATH ROUTINES

This chapter outlines the use of the following routines:

Single Precision

Square Root (DEC-08-FMAA-D)

Multiply (DEC-08-FMBA-D)

Divide (DEC-08-FMCB-D)

Double Precision

Multiply (DEC-08-FMDA-D)

Divide (DEC-08-FMEB-D)

Sine (DEC-08-FMFC-D)

Cosine (DEC-08-FMGB-D)

In addition to the above, the following routines are also available:

Four-Word Floating Point Package (DEC-08-FMHA-D)

Logical Subroutines (DEC-08-FMIA-D)

Arithmetic Shift Subroutines (DEC-08-FMJA-D)

Logical Shift Subroutines (DEC-08-FMKA-D)

IMPLEMENTATION NOTES

The following mathematical subroutines may be used, in general, by preparing a brief program identifying the routine symbol and allocating any registers required for inputting data and storing answers. The program is assembled along with the desired subroutine.

Tape Format

Each of the routines is supplied on an ANSCII-coded tape: formats are as follows:

<u>Routine</u>	<u>Origin Setting</u>	<u>Ending</u>
Square Root	none	\$
Multiply (Single Precision)	none	\$
Divide (Single Precision)	none	PAUSE
Multiply (Double Precision)	none (see Note)	\$

<u>Routine</u>	<u>Origin Setting</u>	<u>Ending</u>
Divide (Double Precision)	none	PAUSE
Sine	*400	PAUSE
Cosine	*1000	\$

NOTE

The Sine routine calls the Multiply routine (double precision) and assumes that it has been loaded starting at location 0200. If Multiply is loaded elsewhere the pointers to it on page 1 of the Sine routine must be changed.

If a tape ending with a PAUSE pseudo-operator is to be assembled alone, it must be followed by a tape containing a dollar sign only.

Signs

The multiply and divide routines assign signs to products and dividends algebraically.

In the divide routines, remainders are given the sign of the dividend.

External Calls

The sine routine makes use of the multiply routine (double precision). The multiply routine must be in core when the sine routine is used, or must be assembled with the sine routine in user programs.

The cosine routine makes use of the sine routine. Both the sine and multiply (double precision) routines must be in core when cosine is used, or must be assembled with the cosine routine in user programs.

Program Control

When a routine has been executed program control returns to the next sequential location to the last argument following a JMS or JMS I instruction.

As an example, in the Divide routine (single precision: see Table 16-1) control returns to JMS I + 3.

If no arguments follow the jump instruction, as in the square root routine, control returns to the jump instruction +1.

Errors

The Divide routines, both single and double precision, make certain error checks and return identical codes in the link and accumulator.

Errors for which checks are made are zero divisor, quotient overflow and fractional quotient. The double precision routine checks, additionally, for non-identical signs in the dividend.

Error codes are as follows:

Error	Link	Accumulator
Zero Divisor	1	7777
Quotient Overflow	1	0000
Fractional Quotient	1	7776
Non-identical signs in dividend (double Precision only)	1	5777

Table 16-1. Mathematical Routines

SINGLE PRECISION

Name	Mnemonic	Calling Sequence	Arguments	Limits	Sign	Result	Location & Accuracy	Sign
Square Root Size: 27*	SQRT	TAD addressX JMS I SQRTPT • • • HLT SQRTPT, SQRT X, 0000	X, Number for root extraction	$0 < N < 2^{12}$	(Absolute value)	Closest root Remainder	AC: 6 bits SQR1: 12 bits	Assumed ± Assumed ±
Multiply Size: 54	MULT	TAD addressX JMS MULT argumentY X, 0000	X, Multiplier Y, Multiplicand	$\pm 2047_{10}$ $\pm 2047_{10}$	Bit 0 of AC JMS+1 bit 0	Product	AC: 10 bits MPI: 12 bits	AC bits 0 and 1 each contain sign
Divide Size: 130	DIVIDE	TAD addressX JMS I DIVDP argument (Y) argument (Z) • • • HLT DIVDP, DIVIDE X, 0000	X, Dividend high order Y, Dividend low order Z, Divisor	$\pm 2^{23}-1$ $\pm 2047_{10}$	Bit 0 of high order word Bit 0 of JMS+2	Quotient Remainder	AC: 11 bits HDIVND: 11 bits	AC bit 0 HDIVND bit 0
DOUBLE PRECISION								
Multiply Size: 175	DMUL	JMS I DMULTP addressX addressY • • • HLT X, 0000 0000 Y, 0000 0000 DMULTP, DMUL	X, Multiplier Y, Multiplier	$\pm 2^{32}-1$	Bit 0 of JMS-1 Bit 0 of JMS+2	Product	AC: 10 bits B: 12 bits C: 12 bits D: 12 bits	AC bits 0 and 1 each contain sign

*Core space requirement (octal)

Table 16-1. Mathematical Routines (Cont.)

DOUBLE PRECISION

Name	Mnemonic	Calling Sequence	Arguments	Limits	Sign	Result	Location & Accuracy	Sign
Divide Size: 307*	DUBDIV	TAD addressX JMS I DDIV addressY . . . HLT X, .+1 0000 0000 0000 0000 Y, 0000 0000 DDIV, DUBDIV	X. Dividend Y. Divisor	$\pm 2^{47}-1$	Bit 0 of high order word	Quotient Remainder	AC: 11 bits DIVND+4: 12 bits DIVND+1: 11 bits	AC bit 0 DIVND+1 bit 0
Sine Size: 337	DSIN	JMS I SINP addressX . . . HLT SINP, DSIN X, 1000 0000	X. Radian value of angle	$-4 < X < 4$	Bit 0 of high order word	Sine	AC: 12 bits (implied binary point after bit 0) ARG+1: 12 bits	Assumed + (angle adjusted to first quadrant)
Cosine Size: 100	DCOS	JMS I DCOSP addressX . . . HLT DCOSP, DCOS X, 1000 0000	X. Radian value of angle	$-4 < X < 4$	Bit 0 of high order word	Cosine	(See above. Cosine uses same register as Sine)	(See above)

*Core space requirement (octal)

Appendices

Appendix A2

Permanent Symbol Table for PDP-8 Assemblers

PAL III, 4K PAL-D, 8K PAL-D, and 8K SABR

The following are the most commonly used elements of the PDP-8 instruction set. For that reason they are found in the permanent symbol table within the assemblers. These instructions are already defined within the computer. For additional information on these instructions and for a description of the symbols used when programming other, optional, I/O devices, see the *1970 Small Computer Handbook*, available from the DEC Program Library.

MACRO

MACRO has, at present, 174 symbols in its permanent symbol table. Because many of these instructions are not used by every user and take up valuable space, it is recommended that you delete the present MACRO permanent symbol table with the EXPUNGE pseudo-op and recreate it to correspond with the PAL III and PAL-D permanent symbol table (below) as explained under the FIXTAB pseudo-op, including in the new table any additional instructions that are needed for optional equipment.

INSTRUCTION CODES

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
Memory Reference Instructions			
AND	0000	Logical AND	
TAD	1000	Two's complement add	
ISZ	2000	Increment and skip if zero	
INC ¹	2000	Nonskip ISZ	
DCA	3000	Deposit and clear AC	
JMS	4000	Jump to subroutine	
JMP	5000	Jump	

¹ Not present in PAL III, 4K PAL-D or 8K PAL-D.

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
Floating-Point Instructions ²			
FEXT	0000	Floating exit	
FADD	1000	Floating add	
FSUB	2000	Floating subtraction	
FMPY	3000	Floating multiply	
FDIV	4000	Floating divide	
FGET	5000	Floating get	
EPUT	6000	Floating put	
FNOR	7000	Floating normalize	

Group 1 Operate Microinstructions

OPR ³	7000	Same as NOP	
NOP	7000	No operation	
IAC	7001	Increment AC	3
RAL	7004	Rotate AC and link left one	4
RTL	7006	Rotate AC and link left two	4
RAR	7010	Rotate AC and link right one	4
RTR	7012	Rotate AC and link right two	4
CML	7020	Complement link	2
CMA	7040	Complement AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1

Group 2 Operate Microinstructions

HLT	7402	Halts the computer	3
OSR	7404	Inclusive OR SR with AC	3
SKP	7410	Skip unconditionally	1
SNL	7420	Skip on nonzero link	1
SZL	7430	Skip on zero link	1
SZA	7440	Skip on zero AC	1
SNA	7450	Skip on nonzero AC	1
SMA	7500	Skip on minus AC	1
SPA	7510	Skip on positive AC (zero is positive)	1
CIA			

Combined Operate Microinstructions

	7041	Complement and increment AC	2, 3
STL	7120	Set link to 1	1, 2
GLK ⁴	7204	Get link (put link in AC, bit 11)	1, 4
STA	7240	Set AC to -1	2
LAS	7604	Load AC with SR	1, 3

² Not present in 4K PAL-D or 8K SABR.

³ Not present in PAL III or 8K SABR.

⁴ Not present in 8K SABR.

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
Program Interrupt			
IOT ⁵	6000		
ION	6001	Turn interrupt processor on	
IOF	6002	Disable interrupt processor	
Keyboard/Reader			
KSF	6031	Skip on keyboard flag	
KCC	6032	Clear keyboard flag and AC	
KRS	6034	Read keyboard buffer (static)	
KRB	6036	Read keyboard buffer (dynamic)	
Teleprinter/Punch			
TSF	6041	Skip on teleprinter flag	
TCF	6042	Clear teleprinter flag	
TPC	6044	Load teleprinter and print	
TLS	6046	Load teleprinter sequence	
High Speed Reader			
RSF	6011	Skip on reader flag	
RRB	6012	Read reader buffer and clear reader flag	
RFC	6014	Reader fetch character	
High Speed Punch			
PSF	6021	Skip on punch flag	
PCF	6022	Clear on punch flag	
PPC	6024	Load punch buffer and punch character	
PLS	6026	Load punch buffer sequence	
DECtape Transport Type TU55 and DECtape Control Type TC01 ⁶			
DTRA	6761	Contents of status register is ORed into AC bits 0-9	
DTCA	6762	Clear status register A, all flags undisturbed	

⁵ Not present in PAL III or 8K SABR.

⁶ Not present in 8K SABR.

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
DTXA	6764	Status register A loaded by exclusive OR from AC. If AC bit 10=0, clear error flags; if AC bit 11=0, DEC-tape control flag is cleared	
DTLA ⁷	6766	Combination of DTCA and DTXA	
DTSF	6771	Skip if error flag is 1 or if DECtape control flag is 1	
DTRB	6772	Contents of status register B is ORed into AC	
DTLB	6774	Memory field portion of status register B loaded from AC bits 6-8	
Disk File and Control, Type DF32 ⁸			
DCMA	6601	Clear disk memory request and interrupt flags	1
DMAR	6603	Load disk from AC, clear AC read into core, clear interrupt flag	1, 2
DMAW	6605	Load disk from AC, write onto disk from core, clear interrupt flag.	1, 3
DCEA	6611	Clear disk extended address and memory address extension register	1
DSAC	6612	Skip if address confirmed flag = 1	2
DEAL	6615	Clear disk extended address and memory address extension register and load same from AC	1, 3
DEAC	6616	Clear AC, load AC from disk extended address register, skip if address confirmed flag = 1	2, 3
DFSE	6621	Skip if parity error, data request late, or write lock switch flag = 0 (no error)	1
DFSC	6622	Skip if completion flag = 1 (date transfer completed)	2
DMAC	6626	Clear AC, load AC from disk memory address register	2, 3

⁷ Not present in 4K PAL-D, 8K PAL-D, or 8K SABR.

⁸ Not present in 8K SABR.

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
-----------------	-------------	------------------	-------------------

Memory Extension Control, Type 183⁹

CDF	N0	6201	Change to data field N
CIF	N0	6202	Change to instruction field N
RDF		6214	Read data field
RIF		6224	Read instruction field
RIB		6234	Read interrupt buffer
RMF		6244	Restore memory field

Memory Parity Type MP8/I (MP8/L)¹⁰

SMP		6101	Skip if memory parity error flag = 0
CMP		6104	Clear memory parity error flag

⁹ Not present in 8K SABR.

¹⁰ Not present in 4K PAL-D, 8K PAL-D, or 8K SABR.

PSEUDO-OPERATORS

The following is a list of the 4K and 8K assembler pseudo-ops. The first section consists of those pseudo-ops which have counterparts in the other assemblers. Below the blank space are the various pseudo-ops individual to the particular assembler.

<u>PAL III</u>	<u>MACRO</u>	<u>4K PAL-D</u>	<u>8K PAL-D</u>	<u>8K SABR</u>
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL
OCTAL	OCTAL	OCTAL	OCTAL	OCTAL
FIELD	FIELD	FIELD	FIELD	
PAUSE	PAUSE	PAUSE	PAUSE	PAUSE
I	I	I	I	I
Z	Z	Z	Z	
\$	\$	\$	\$	\$
EXPUNGE	EXPUNGE	EXPUNGE	EXPUNGE	
FIXTAB	FIXTAB	FIXTAB	FIXTAB	
	PAGE	PAGE	PAGE	PAGE
FIXMRI	DEFINE	XLIST	XLIST	ABSYM
	DUBL	TEXT	IFDEF	ARG
	FLTG		IFZERO	BLOCK
	TEXT		ENPUNCH	CALL
			NOPUNCH	COMMN
			ZBLOCK	CPAGE
			EJECT	DUMMY
			TEXT	EAP
				END
				ENTRY
				FORTR
				IF
				LAP
				OPDEF
				REORG
				RETRN
				SKPDEF

NOTE: The symbols ACH, ACM, and ACL are also present in the permanent symbol table for 8K SABR. For details, see Chapter 14.

4K PAL-D IOT MICROINSTRUCTIONS FOR TSS/8 MONITOR

The following instructions are permanent symbols in addition to those already mentioned, and are unique to 4K PAL-D on TSS/8.

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
Keyboard/Reader			
KSB	6400	Set keyboard break	
SBC	6401	Set buffer control flags	
KSR	6030	Read keyboard string	
Teleprinter/Punch			
SAS	6040	Send a string	
High-Speed Reader (Type PC02)			
RRS	6010	Read reader string	
High-Speed Punch (Type PC03)			
PST	6020	Punch string	
Program Control			
URT	6411	User run time	
TOD	6412	Time of day	
RCR	6413	Return clock rate	
DATE	6414	Date	
SYN	6415	Quantum synchronization	
STM	6416	Set timer	
TSS	6420	Skip on TSS/8	
USE	6421	User	
SSW	6430	Set Switch Register	
CKS	6200	Check status	
ASD	6440	Assign device	
REL	6442	Release device	
DUP	6402	Duplex	
CON	6422	Console	

<u>Mnemonic</u>	<u>Code</u>	<u>Operation</u>	<u>Event Time</u>
File Control			
REN	6600	Rename file	
OPEN	6601	Open file	
CLOS	6602	Close file	
RFILE	6603	Read file	
PROT	6604	Protect file	
WFILE	6605	Write file	
CRF	6610	Create file	
EXT	6611	Extend file	
RED	6612	Reduce file	
FINF	6613	File information	
SIZE	6614	Segment size	
SEGS	6406	Segment count	
ACT	6617	Account number	
WHO	6616	Who	

PAL III EXTENDED SYMBOLS TAPE

The PAL III Extended Symbols Tape, available from the Program Library, contains symbols for the following hardware options (which are further described in the *1970 Small Computer Handbook*).

- Extended Arithmetic Element, Type KE8/I, KE8/L
- Power Fail Detection and Restart, Type KP8/I, KP8/L
- Disk File, Type RS08 and Control, Type RF08
- Card Reader and Control, Type CR8/I, CR8/L
- General Purpose Converter and Multiplexor Control, Type AF01A
- Guarded Scanning Digital Voltmeter, Type AF04A
- Real Time Clocks, Type KW8/I, KW8/L
- Storage Tube Display Control, Type KV8/I, KV8/L
- Incremental Magnetic Tape Controller, Type TR02
- Oscilloscope Display, Type VC8/I, VC8/L
- Incremental Plotter and Control, Type VP8/I, VP8/L
- Automatic Magnetic Tape Control, Type TC58
- Time Sharing Hardware Modification, Type KT8/I
- 8 Channel Sample and Hold Control, Type AC01A
- Digital to Analog Converter, Type AA01A, AA0SC/AA07
- Synchronous Modem Interface, Type DP01AA
- Line Printer, Type LP08

Appendix B2

Character Codes

ANSII-1 Character Set

Character	8-Bit Octal	6-Bit Octal	Character	8-Bit Octal	6-Bit Octal
A	301	01	!	241	41
B	302	02	"	242	42
C	303	03	#	243	43
D	304	04	\$	244	44
E	305	05	%	245	45
F	306	06	&	246	46
G	307	07	'	247	47
H	310	10	(250	50
I	311	11)	251	51
J	312	12	*	252	52
K	313	13	+	253	53
L	314	14	,	254	54
M	315	15	-	255	55
N	316	16	.	256	56
O	317	17	/	257	57
P	320	20	:	272	72
Q	321	21	;	273	73
R	322	22	<	274	74
S	323	23	=	275	75
T	324	24	>	276	76
U	325	25	?	277	77
V	326	26	@	300	
W	327	27	[333	33
X	330	30	\	334	34
Y	331	31]	335	35
Z	332	32	↑	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	

¹ An abbreviation for American National Standard Code for Information Interchange.



B2-2

Faint, illegible text or markings at the bottom right of the page, possibly a page number or reference code.

Appendix C2

Loading 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, Vol. 1). (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 C2-1.

The procedure for loading (toggling) the RIM Loader into core is illustrated in Figure C2-1.

Table C2-1. 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 C2-2, which also shows how to correct an incorrectly stored instruction.

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

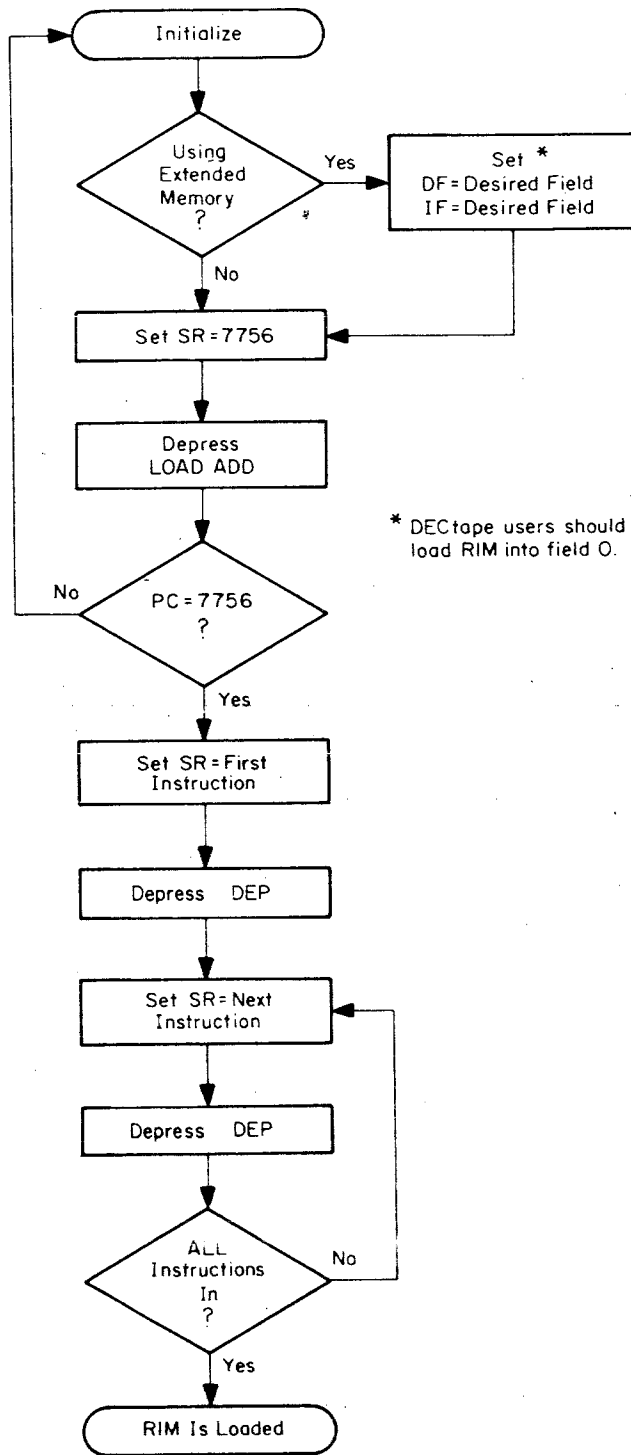


Figure C2-1. Loading the RIM Loader

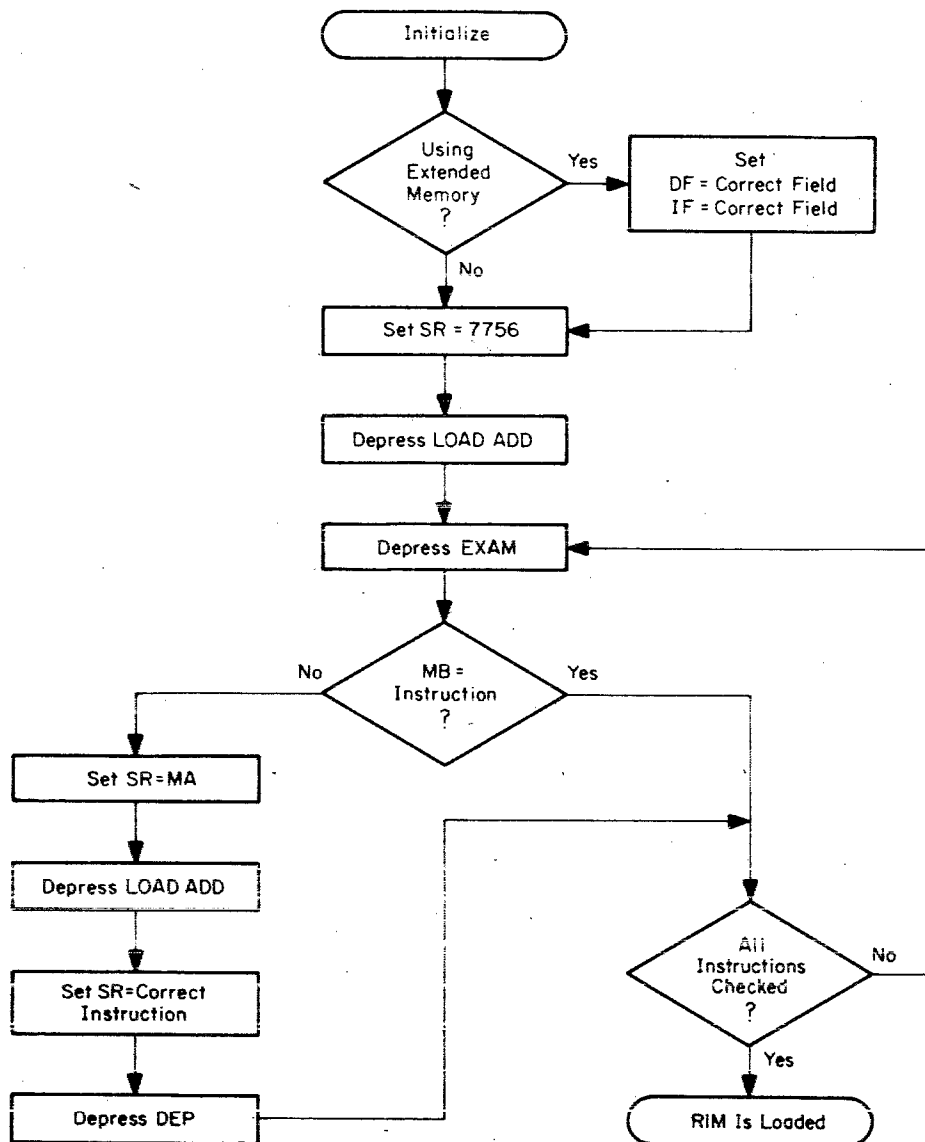


Figure C2-2. Checking the RIM Loader

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 C2-3 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.

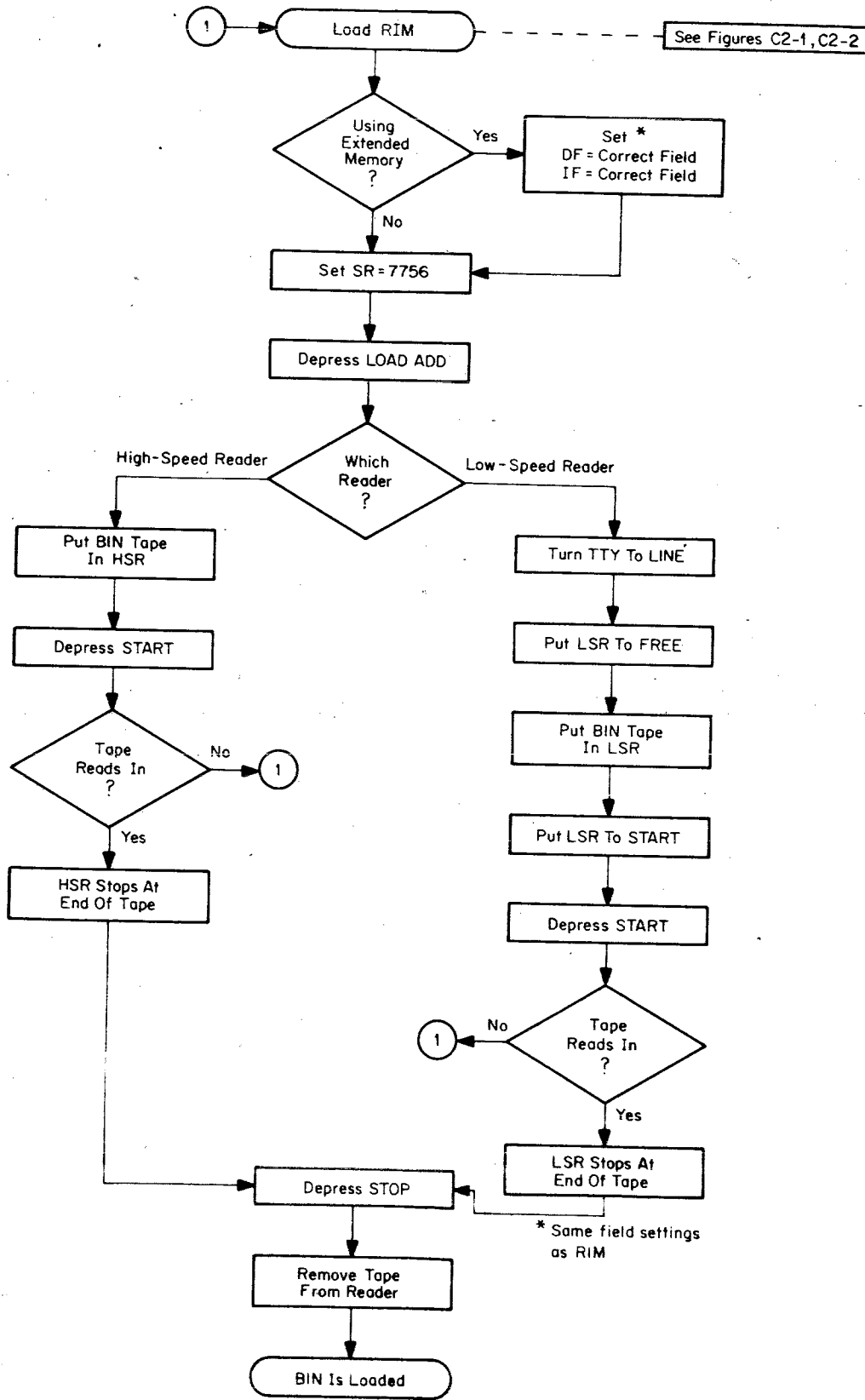


Figure C2-3. Loading the BIN Loader

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 Vol. 1). 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 C2-4 illustrates the procedure for loading binary tapes into core.

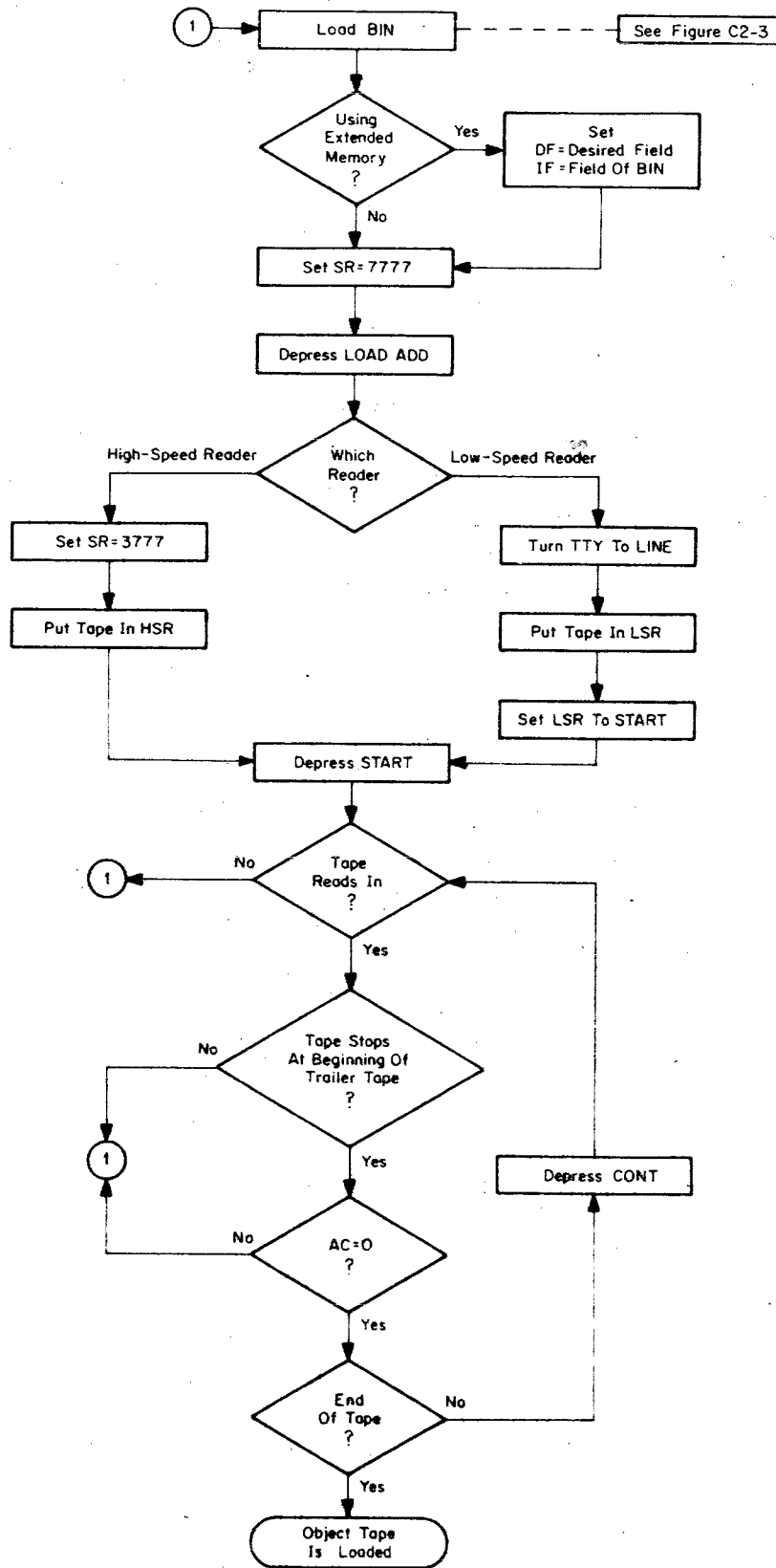


Figure C2-4. Loading A Binary Tape Using BIN



Index/Glossary

A

- Absolute address*: A binary number that is permanently assigned as the address of a storage location.
- Absolute value function, FOCAL, 11-25
- ABSYM Pseudo-op, SABR, 14-23
- Accumulator*: A 12-bit register in which the result of an operation is formed; abbreviation: AC.
- Accumulator, floating, 16-17
- Accuracy, Increased FOCAL, 11-34
- Addition, 16-17
- Additional FOCAL Segments, 11-34
- Address*: A label, name, or number which designates a location where information is stored.
- Address assignments
- 4K Assemblers, 13-21
 - 8K Assemblers, 14-37
- Algorithm*: A prescribed set of well-defined rules or processes for the solution of a problem in a finite number of steps.
- Algorithms, Floating-Point, 16-17
- Alphanumeric*: Pertaining to a character set that contains both letters and numerals, and usually other characters.
- Alphanumeric data, 15-22, 15-60
- Alphanumerics
- SABR, 14-11
 - FOCAL, 11-29
- ALT MODE key
- BASIC, 12-50
 - FOCAL, 11-17
- ANSII character set, B2-1
- Arctangent, 16-21; FOCAL, 11-26
- ARG pseudo-op, SABR, 14-26
- Argument*:
1. A variable or constant which is given in the call of a sub-routine as information to it;
 2. A variable upon whose value the value of a function depends;
 3. The known reference factor necessary to find an item in a table or array (i.e., the index).
- Arguments, passing
- SABR, 14-30
- Arithmetic expressions
- FORTRAN, 15-11
 - FOCAL, 11-6
- Arithmetic operators
- BASIC, 12-12
 - FOCAL, 11-74
- Arithmetic statements
- FORTRAN, 15-6, 15-8
- Arithmetic unit*: The component of a computer where arithmetic and logical operations are performed.
- Array*: A set or list of elements, usually variables or data.
- Array storage
- FORTRAN, 15-54
- Array variables,
- FORTRAN, 15-11, 15-33, 15-53, 15-54
 - FOCAL, 11-14
 - BASIC, 12-11
- ASCII*: An abbreviation for American Standard Code for Information Interchange. See ANSCII, A2-1.
- ASK statement, FOCAL, 11-17
- Assemble*: To translate from a symbolic program to a binary program by substituting binary operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.
- Assembled binary code
- 8K FORTRAN, 14-49
 - SABR, 14-38
- Assembler*: A program which translates symbolic op-codes into machine language and assigns memory locations for variables and constants.
- Assembler output, 4K ASSEMBLERS, 13-33

- SABR, 14-38
- Assemblers, PDP-8
 - 4K, 13-5
 - 8K, 14-5
 - MACRO, 13-41
 - PAL III, 13-5
 - 4K PAL-D, 13-65
 - 8K PAL-D, 14-5
 - 8K SABR, 14-9
- Assembly
 - Floating-Point Package, 16-11
 - SABR, 14-47
- Auto-indexing*: When an absolute location 0010 through 0017 is addressed indirectly, the content of that location is incremented by one, rewritten in that same location, and used as the effective address of the current instruction.
- Auto-indexing, 13-23
- Automatic paging mode,
 - SABR, 14-33
- Auxiliary storage*: Storage that supplements core memory such as disk or DECTape.
- B**
- Background processing*: The automatic execution of a low priority computer program when higher priority programs are not using the system resources.
- Base address*: A given address from which an absolute address is derived by combination with a relative address. Synonymous with address constant.
- BASIC**
 - see Table of Contents, 12-3
 - error messages, 12-60
 - example programs, 12-7, 12-21, 12-25, 12-31, 12-40
 - summary of Edit and Control commands, 12-58
 - summary of functions, 12-58
 - summary of statements, 12-56
- BASIC commands**
 - ALT MODE key, 12-50
 - BYE, 12-50
 - CATALOG, 12-49
 - COMPILE, 12-54
 - DELETE, 12-49
 - EDIT, 12-53
 - LIST, 12-48
 - NEW, 12-49
 - OLD, 12-49
 - REPLACE, 12-48
 - RUN, 12-46
 - SAVE, 12-47
 - UNSAVE, 12-48
- BASIC statements**
 - DATA, 12-23
 - DEF, 12-19
 - DIM, 12-33
 - END, 12-41
 - FOR, 12-34
 - GOSUB, 12-39
 - GOTO, 12-37
 - IF-GOTO, 12-38
 - IF-THEN, 12-38
 - INPUT, 12-25
 - LET, 12-11
 - NEXT, 12-35
 - PRINT, 12-27
 - RANDOMIZE, 12-17
 - READ, 12-22
 - REM, 12-9
 - RESTORE, 12-24
 - RETURN, 12-40
 - STOP, 12-41
 - TAB function, 12-30
- Beginning programming**
 - BASIC, 12-5
 - FOCAL, 11-5
- Binary*: Pertaining to the number system with a radix of two.
- Binary code*: A code that makes use of exactly two distinct characters, 0 and 1. Same as object code.
- Binary Loader, C2-1; FOCAL, 11-42
- Binary output
 - FORTRAN, 14-38, 14-49
- Bit*: A binary digit. In PDP-8 computers each word is composed of 12 bits.
- Blank lines, 8K ASSEMBLERS, 14-10
- Blanks in output
 - FORTRAN, 15-25
- Block*: A set of consecutive machine words, characters, or digits han-

dled as a unit, particularly with reference to I/O.

BLOCK pseudo-op, SABR, 14-25

Bootstrap: A technique or device designed to bring itself into a desired state by means of its own action, e.g., a routine whose first few instructions are sufficient to bring the rest of itself into the computer from an input device.

Branch: A point in a routine where one of two or more choices is made under control of the routine.

Bug: A mistake in the design or implementation of a program resulting in erroneous results.

Byte: A group of binary digits usually operated upon as a unit.

C

Call: To transfer control to a specified routine.

CALL pseudo-op, SABR, 14-26

Calling sequence: A specified set of instructions and data necessary to set up and call a given routine.

CALL statement,

FORTRAN, 15-29, 15-38

Carriage return: The Teletype operation that causes the next character to be printed at the left margin. A non-printing character.

Carriage return/line feed: Two Teletype functions often done together; rolls paper up one line and moves the printing head to the left margin.

CDF current data field, SABR, 14-40

CDFSKP Linkage routine, 14-34

CDISK routine, SABR, 14-82

FORTRAN, 15-31

CDZSKP Linkage routine, 14-35

Central processing unit: The unit of a computing system that includes the circuits controlling the interpretation and execution of instructions—the computer proper, excluding I/O and other peripheral devices.

Chaining, SABR, 14-69

Character: A single letter, numeral,

or symbol used to represent information.

Character set

ANSII, A2-1

FORTRAN, 15-8

MACRO, 13-44

PAL III, 13-7, 13-8

8K PAL-D, 14-5

SABR, 14-11

Clear: To erase the contents of a storage location by replacing the contents, normally with zeros or spaces; to set to zero.

Cline, **FOCAL**, 11-34

Coding: To write instructions for a computer using symbols meaningful to the computer, or to an assembler, compiler, or other language processor.

Coding practices

4K and 8K Assemblers, 13-11

Command: A user order to a computer system. Usually given through a Teletype keyboard.

Command decoder: That part of a computer system which interprets user commands. Also called command-string decoder.

Commands, Floating-Point, 16-8, 16-17

Comment statements

4K Assemblers, 13-11

BASIC (REM), 12-9

FORTRAN, 15-7

SABR, 14-16

FOCAL, 11-13

COMMN pseudo-op, SABR, 14-24

COMMON statement

FORTRAN, 15-33, 15-37

COMMON storage

SABR, 14-24

FORTRAN, 15-56

Computability: The ability of an instruction or source language to be used on more than one computer.

Compile: To produce a binary-coded program from a program written in source (symbolic) language, by selecting appropriate subroutines from a subroutine library, as directed by the instructions or other

- symbols of the source program. The linkage is supplied for combining the subroutines into a workable program, and the subroutines and linkage are translated into binary code.
- COMPILE** command
 BASIC, 12-54
- Compiler*: A program which translates statements and formulas written in a source language into a machine language program, e.g., a FORTRAN Compiler. Usually generates more than one machine instruction for each statement.
- Compiler error messages
 FORTRAN, 15-41
- Compiling a program
 BASIC, 12-54
- Complement (one's)*: To replace all bits in a binary word with 1 bits and vice versa.
- Complement (two's)*: To form the one's complement and add 1.
- Computed GO TO, FORTRAN, 15-16
- Conditional assembly*: Assembly of certain parts of a symbolic source program only if certain conditions have been met.
- Conditional skip*: Depending upon whether a condition within the program is met, control may transfer to another point in the program. See Operate micro instructions.
- Conditional transfer
 BASIC, 12-38
- Console*: Usually the external front side of a device where controls and indicators are available for manual operation of the device.
- Constant*: Numeric data used but not changed by the program.
- Constants
 ANSCII, 14-16
 FORTRAN, 15-8, 15-9, 15-53
 Numeric, 14-15
 SABR, 14-15
- Continuation lines
 FORTRAN, 15-7
- CONTINUE statement
 FORTRAN, 15-18
- Confroller, output, 16-15
- Control statements
 FORTRAN, 15-6, 15-16
- Conversational languages
 BASIC, 12-5
 FOCAL, 11-5
- Conversational program*: A program which interacts dynamically with on-line users, FOCAL, 11-5
- Conversion, decimal-to-binary, 16-11
 Fixed-to-Floating, 16-19
 Floating-to-Fixed, 16-19
- Conversion, Numeric, SABR, 14-77
- Convert*:
 1. To change numerical data from one radix to another.
 2. To transfer data from one recorded format to another.
- Core availability, SABR, fields, 14-53
 Linking Loader option, 14-52
- Core memory*: The main high-speed storage of a computer in which binary data is represented switching polarity of magnetic cores.
- Correction of errors
 BASIC, 12-42
 FOCAL, 11-9
- Cosine, Fl. Pt. Package, 16-21, 16-29; FOCAL, 11-27
- Count*: The successive increase or decrease of a cumulative total of the number of times an event occurs.
- Counter*: A register or storage location (variable) used to represent the number of occurrences of an operation (see Loop).
- CPAGE pseudo-op, SABR, 14-21
- Current address indicator
 4K Assemblers, 13-22
- Current location counter*: A counter kept by an assembler to determine the address assigned an instruction or constant being assembled. 4K Assemblers, 13-19, 13-21.
- Current page or page 0 bit*: Bit 4 of a PDP-8 memory reference instruction.
- Cycle time*: The length of time it takes the computer to reference one word of memory.

D

- Data*: A general term used to denote any or all facts, numbers, letters, and symbols. It connotes basic elements of information which can be processed or produced by a computer.
- Data break*: A facility which permits I/O transfers to occur on a cycle-stealing basis without disturbing program execution.
- Data generating, SABR, 14-25
- DATA statement
BASIC, 12-23
- Data output, FOCAL, 11-7
- Data transmission statements
FORTRAN, 15-26
- Debug*: To detect, locate, and correct mistakes in a program.
- DECIMAL pseudo-op, 4K Assemblers 13-28
SABR, 14-20
- DEctape I/O
FORTRAN, 15-28
SABR, 14-80
- DEctape instructions, A2-3
- DEF statement, BASIC, 12-19
- Deleting a command
BASIC, 12-50
- Deleting a program on Disk
BASIC, 12-48
- Delimiter*: A character that separates, terminates, and organizes elements of a statement or program.
- Demonstration programs, SABR, 14-84 FOCAL, 11-45
- Device designator
FORTRAN, 15-28
- Device selection code*: A 6-bit number which is used to specify the device referred to by an IOT instruction.
- Diagnostic*: Pertaining to the detection and isolation of a malfunction or mistake.
- Diagnostics, see Error Messages.
- Digit*: A character used to represent one of the non-negative integers smaller than the radix, e.g., in binary notation, either 0 or 1.
- Digital computer*: A device that operates on discrete data, performing sequences of arithmetic and logical operations on this data.
- DIM statement, BASIC, 12-33
- DIMENSION statement
FORTRAN, 15-33, 15-37
- Direct address*: An address that specifies the location of an instruction operand.
- Direct Assignment statement, Assemblers, 13-16
- Directory device*: A device (such as a disk) which is partitioned by software into several distinct files. A directory of these files (e.g., an index) is maintained on the device to locate the individual files.
- Disk instructions, A2-4
- DISKIN, FOCAL, 11-45
- Disk I/O
FORTRAN, 15-30
SABR, 14-82
- Disk Linking Loader, 14-56
normal loading, 14-62
overlay loading, 14-64, 14-67
error messages, 14-73
- Disk Monitor System
FORTRAN, 15-40
4K PAL-D, 13-68
FOCAL, 8K, 11-49
- Division, floating-point, 16-18
- DO statement,
FORTRAN, 15-17
FOCAL, 11-19, 11-12
- Dollar sign (\$), 13-30
- Double precision*: Pertaining to the use of two computer words to represent one number. In the PDP-8 a double precision result is stored in 24 bits. See Chapter 16.
- Double precision integers, 13-41
- Downtime*: The time interval during which a device is inoperative.
- Dummy*: Used as an adjective to indicate an artificial address, instruction, or record of information inserted solely to fulfill prescribed conditions, as in a "dummy" variable. E.g., in the BASIC function RND(x), where x has no significance. See also DUMMY pseudo-op in 8K SABR.

Dummy
 arguments, 15-18
 statement, 15-36
 variables, 14-28
 DUMMY pseudo-op, SABR, 14-28
Dump: To copy the contents of all or part of core memory, usually onto an external storage medium.
 DUMSUB Linkage routine, 14-36

E

e, raising to power, 16-21
 EAP pseudo-op, SABR, 14-20
 Page escapes, 14-33
 EDIT command
 BASIC, 12-53
 Editing command
 BASIC, 12-53
 Editing phase
 BASIC, 12-47
 Editor, see Symbolic Editor
Effective address: The address actually used in the execution of a computer instruction.
 8K. FOCAL, 11-34; loading procedures, 11-49
 End of Program, 13-30
 End of Tape, 13-30
 END pseudo-op, SABR, 14-19
 END statement
 FORTRAN, 15-20
 Entry points, floating-point,
 input routine, 16-12
 interpreter, 16-9
 listed, 16-16
 negate subroutine, 16-15
 output routine, 16-14
 ENTRY pseudo-op, SABR, 14-28
 Equal sign, explanation of, 12-11
 Equate statements, 4K Assemblers, 13-16
 Equipment requirements, FOCAL, 11-53
 EQUIVALENCE statement,
 FORTRAN, 15-34, 15-37
 Equivalent symbols, SABR, 14-13
 ERASE command, FOCAL, 11-16
 ERASE ALL command, FOCAL, 11-16, 11-22

Erasing a program in core
 BASIC, 12-43
 Error flag, floating-point, 16-19
 Error correction, FOCAL, 11-15, 11-21
 Error messages
 BASIC, 12-42
 Disk Linking Loader, 14-61, 14-65
 Floating Point, 16-11, 16-19, 16-27
 FORTRAN, 15-41
 Library programs, 14-73
 Linking Loader, 14-72
 MACRO, 13-61
 PAL III, 13-38
 4K PAL-D, 13-70
 SABR, 14-71
 FOCAL, 11-15, 11-5
 Escapes, page, 14-33
 Estimating program length, FOCAL, 11-40
 Evaluation of expressions
 4K Assemblers, 13-14
Execute: To carry out an instruction or run a program on the computer.
 Executing FORTRAN programs, 15-46
 Exit functions
 Disk Linking Loader, 14-67
 Exponent, 16-5, 16-8
 in Floating-Point format, 16-6, 16-19
 routines, SABR, 14-79
 storage, 16-6
 FOCAL, 11-26
 Expressions
 4K Assembler, 13-18, 13-44
 EXPUNGE pseudo-op, 4K Assemblers, 13-31
 Extended Floating-Point package, 16-21
 Extended memory
 4K Assembler, 13-28
 Extended symbols
 tape, A2-8
 External Calls
 Math Routines, 16-26
External storage: A separate facility or device on which data usable

by the computer is stored (such as paper tape, DECTape, or disk).

External subroutines

SABR, 14-26

F

FADD, 16-7

FDIV, 16-7

FGET, 16-7

Field:

1. One or more characters treated as a unit.
2. A specified area of a record used for a single type of data.
3. A division of memory on a PDP-8 computer referring to a 4K section of core.

FIELD pseudo-op, 4K Assemblers, 13-28

File: A collection of related records treated as a unit.

File assignment

Disk Linking Loader, 14-60

Filename: Alphanumeric characters used to identify a particular file.

Filename extension: A short appendage to the filename to identify the type of data in the file. e.g., "BIN" signifying a binary program.

File structured device: A device such as disk or DECTape which contains records organized into files and accessible through file names found in a directory file. See directory device.

Fixed point: The position of the radix point of a number system is constant according to a predetermined convention.

Fixed-to Floating-Point Format, 16-19

FIXMRI pseudo-op, 4K Assemblers, 13-32

FIXTAB pseudo-op, 4K Assemblers, 13-31

Flag: A variable or register used to record the status of a program or device. In the latter case, sometimes called a device flag.

Flags, Floating-Point, 16-16

SABR, 14-13

Flip-flop: A device with two stable states.

Floating-point: A number system in which the position of the radix point is indicated by one part (the exponent part), and another part represents the significant digits (the fractional part).

Floating-Point

algorithms, 16-17

extended package, 16-21

input routine, 16-11

instructions, 16-8

packages, assembly, 16-11

package versions, 16-10

operations, 16-9

output controller, 16-15

output routine, 16-14

representation, 16-5

storage, 16-6

summaries, 16-15, 16-16, 16-21

Floating-Point accumulator, 14-74

Floating-Point arithmetic, Library subprograms, 14-75

Floating-Point constants, 13-42

Floating-Point instructions, A2-2

FORTRAN, 16-16

Floating-Point Package, 16-16

Floating-to Fixed-Point Format, 16-19

Flowchart: A graphical representation of the sequence of operations required to carry out data processing. See Appendix C of *Introduction to Programming*.

FMPY, 16-7

FNOR, 16-7

FOCAL, 11-5

ASK, 11-17

COMMENT, 11-13

Current tapes, 11-39

DO, 11-12, 11-19

ERASE, 11-16, 11-22

FOR, 11-20

GO, 11-11

GOTO, 11-11, 11-19

IF, 11-18

Loading Procedures, 11-42

LOCATIONS, 11-50

MODIFY, 11-21, 11-22
 RETURN, 11-13
 Systems, 11-30
 Techniques, 11-29
 Trace, 11-23
 TYPE, 11-6
 WRITE, 11-13, 11-16
 FOR statement, FOCAL, 11-20;
 BASIC, 12-34
Format: The arrangement of data.
 Also a FORTRAN statement.
 FORMAT fields, FORTRAN, 15-23
 Format handling
 FORTRAN, 15-58
 FORMAT statement
 FORTRAN, 15-20
 Form feed
 PAL III, 13-8
 FORTRAN
 demonstration program, 15-47
 features, 15-5
 Pass, 2,
 assembly methods, 14-49
 operating procedures, 14-51
 specifications, format, 15-52
 statement summary, 15-50
 FORTRAN statements, 15-6
 CALL, 15-29, 15-38
 CDISK, 15-30
 COMMON, 15-33
 CONTINUE, 15-18
 DIMENSION, 15-33
 DO, 15-17
 END, 15-19
 EQUIVALENCE, 15-34
 FORMAT, 15-20
 GO TO, 15-16
 IF, 15-17
 ODISK, 15-30
 PAUSE, 15-19
 RDISK, 15-32
 READ, 15-27
 RTAPE, 15-28
 STOP, 15-19
 WDISK, 15-32
 WRITE, 15-27
 WTAPE, 15-28
 FORTR pseudo-op, SABR, 14-50

Four-user FOCAL, 11-31
 4 WORD, FOCAL, 11-34
 FPUT, 16-7
 Free core, Page (SABR), 14-52
 FSUB, 16-7
Full duplex: Describes a communications channel capable of simultaneous and independent transmission and reception. Same as duplex.
Function subprogram: A subprogram which returns a single value result, usually in the accumulator.
 Functions
 BASIC, 12-14
 FORTRAN, 15-14, 15-35
 Functions, Library subprograms,
 14-74
 FOCAL, 11-24

G

GET (FGET), 16-7
 GO statement, FOCAL, 11-11
 GOSUB statement
 BASIC, 12-39
 GOTO statement
 BASIC, 12-37
 FORTRAN, 15-16
 FOCAL, 11-19
 GRAPH, FOCAL, 11-37, 11-47
 Graphics package, FOCAL, 11-34
 Group 1 operate microinstructions,
 13-25, A2-2
 Group 2 operate microinstructions,
 13-26, A2-2

H

Half duplex: Describes a system permitting communication in only one direction at a given instant.
Hardware: Physical equipment, e.g., mechanical, electrical, or electronic devices. Disk Linking Loader, 14-58; Linking Loader, 14-9; SABR, 14-9.
Head: A component that reads, records, or erases data on a storage device.
 High order (register), 16-17
 High-speed Reader/Punch instructions, A2-3
 Hollerith fields, 15-23

I

I pseudo-op, 4K Assemblers, 13-22
Identities, Floating Point, 16-21, 16-22
IF-GOTO statement
 BASIC, 12-38
IF pseudo-op, 14-22
IF statement
 FORTRAN, 15-17
 FOCAL, 11-18
IF-THEN statement
 BASIC, 12-38
Illegal characters
 PAL III, 13-8
Implementation notes
 BASIC, 12-52
Implementation Notes (Math routines), 17-25
Implied DO loops
 FORTRAN, 15-57
Incrementing operands
 SABR, 14-17
Index
 FORTRAN, 15-18
Indirect address: An address in a computer instruction which indicates a location where the address of the referenced operand is to be found.
Indirect addressing, 13-22
Indirect mode, FOCAL, 11-11
Initial Dialogue
 BASIC, 12-45
 FOCAL, 11-54
Initialize: To set counters, switches, and addresses to zero or other starting values at the beginning of, or at prescribed points in, a computer routine.
Input: The transferring of data from auxiliary or external storage into the core memory of the computer.
Input/Output (see also I/O)
 BASIC, 12-22 to 12-31
 Library subprograms, 14-74
 FORTRAN, 15-6, 15-20
 Floating-point, 16-11, 16-14
Instruction: A command which causes the computer or system to perform an operation. Usually one line of a source program.

Instructions

Assembler, 13-11, 13-24
Floating Point, 16-7
SABR
 IOT, A2-1
 Memory Reference, A2-1
 Micro, A2-1
 Multiple word, 14-34, 14-45
 Skip, 14-23, 14-37
Integer arithmetic; Library subprograms, 14-77
Integer constants
 FORTRAN, 15-9
 FOCAL, 11-25
Integer function
 BASIC, 12-15
Integer variables
 FORTRAN 15-10
Interactive, see conversational
Internal storage: The storage facilities forming an integral physical part of the computer and directly controlled by the computer. Also called main memory and core memory.
Internal symbol representation
 MACRO, 13-41, 13-42, 13-57
 PAL III, 13-12
Interpage references, 13-23
Interpreter: A program that translates and executes source language statements at run time. 16-9
I/O: Abbreviation for input/output
I/O devices, special
 FORTRAN, 15-61
I/O list
 FORTRAN, 15-26
I/O records
 FORTRAN, 15-26
I/O routines, DECTape, 14-80
IOH routines, SABR, 14-74
IOH instructions, SABR, 14-12, A2
Iteration: Repetition of a group of instructions.

J

Job: A unit of code which solves a problem, i.e., a program and all its related subroutines and data.

Jump: A departure from the normal sequence of executing instructions in a computer.

K

K: An abbreviation for the prefix kilo, i.e., 1000 in decimal notation. In the computer field, loosely, two to the tenth power, which is 1024 in decimal notation. Hence, a 4K memory has 4096 words.

8K, FOCAL, see Eight (8) K.

Keyboard/Reader instructions, A2-3

L

Label: One or more characters used to identify a source language statement or line. SABR 14-12; 4K Assemblers, 13-11.

Language, assembly: The machine-oriented programming language belonging to an assembly system, e.g., PAL III, PAL-D, and SABR.

Language, computer: A systematic means of communicating instructions and information to the computer.

Language machine: Information that can be directly processed by the computer, expressed in binary.

Language, source: A computer language, such as PAL III or FOCAL, in which programs are written and which require a translation in order to be executed by the computer.

LAP pseudo-op, SABR, 14-20

Page escapes, 14-33

Leader: The blank section of tape at the beginning of the tape.

Least significant digit: The right-most digit of a number.

Legal characters

8K PAL-D, 14-5

SABR, 14-11

LET statement

BASIC, 12-11

LIBRA, FOCAL, 11-32

commands, 11-32

common storage function, 11-33

limitations, 11-33

loading, 11-44

stop and restart, 11-48

Library subprograms

demonstration program, 14-84

error messages, 14-73

floating-point arithmetic, 14-75

functions, 14-78

input/output, 14-74

integer arithmetic, 14-77

organization, 14-74

powers, 14-79

subscripting, 14-78

Library routines: A collection of standard routines which can be incorporated into larger routines. FORTRAN, 15-15.

Line feed: The Teletype operation which advances the paper by one line.

Line number: In source languages such as FOCAL, BASIC, and FORTRAN, a number which begins a line of the source program for purposes of identification. A numeric label. BASIC, 12-9; FORTRAN, 15-6, 15-7; FOCAL, 11-11

Link:

1. A one bit register in the PDP-8.
2. An address pointer generated automatically by the PAL-D or MACRO-8 Assembler to indirectly address an off page symbol.
3. An address pointer to the next element of a list, or the next block number of a file.

Linkage: In programming, code that connects two separately coded routines.

Linkage routines, runtime (SABR), 14-34, 14-52, 14-56

Link generation

MACRO, 13-46

Linking Loader (Paper-Tape), 14-34, 14-50

error messages, 14-72

executing FORTRAN programs, 15-46

FORTRAN disk I/O, 15-30

information options, 14-52

loading, 14-54

memory map option, 14-52

relocation codes, 14-39

system requirements, 14-9, 14-51
LINK Linkage routine, SABR, 14-36

List:

1. A set of items.
2. To print out a listing on the line printer or Teletype.
3. See pushdown list.

Listing

8K PAL-D, 14-6, 14-7

Pass 2, assembly, 14-47

SABR, 14-43, 14-48

Unloaded Program, 14-67

Listing control

PAL-D, 13-65

Listing Files

BASIC, 12-49

Listing a paper tape

BASIC, 12-51

Listing a program

BASIC, 12-48

Literal: A symbol which defines itself. MACRO, 13-48; SABR, 14-14

Load: To place data into internal storage.

Loader, Binary (BIN), C2-4

Disk Linking, see Disk Linking Loader

Paper tape, see Linking Loader

Read-in Mode (RIM), C2-1

Loading procedures; see Appendix C2

Disk Linking Loader, 14-57

FOCAL, 11-42

FORTRAN, 15-39

Linking Loader, 14-54

programs and subprograms, 14-54
overlay, 14-56

SABR, 14-9, 14-46

Switch register options, 14-52

Location: A place in storage or memory where a unit of data or an instruction may be stored.

Location counter, see current location counter

LOCATIONS command, FOCAL, 11-50

Logarithm, 16-21; FOCAL, 11-26

Lookup table, 16-15

Loop: A sequence of instructions

that is executed repeatedly until a terminal condition prevails.

BASIC, 12-34

Low order (register), 16-17

M

Machine language-programming: In this text, synonymous with assembly language programming (the term is sometimes used to mean the actual binary machine instructions).

Macro instruction: An instruction in a source language that is equivalent to a specified sequence of machine instructions.

MACRO

see Table of Contents, 13-3

Link generation, 13-46

literals, 13-48

switch options, 13-60

symbol table, A2-1

versions, 13-58

Macros, 13-52

calling, 13055

defining, 13-53

macro table, 13-55

restrictions on, 13-54

Mantissa, 16-5

binary point relative position, 16-6, 16-19

storage, 16-6

Manual input: The entry of data by hand into a device at the time of processing.

Manual operation: The processing of data in a system by direct manual techniques.

Mask: A bit pattern which selects those bits from a word of data which are to be used in some subsequent operation.

Mass storage: Pertains to a device such as DECTape or disk which stores large amounts of data readily accessible to the central processing unit.

Math routines, 16-16; FOCAL, 11-24

Matrix: A rectangular array of elements. A table can be considered to be a matrix.

Memory:

1. The alterable storage in the computer.
2. Pertaining to a device in which data can be stored and from which it can be retrieved.

Memory extension control instructions, A2-5

Memory map option, Linking Loader, 14-52

Memory parity instructions, A2-5

Memory protection: A method of preventing the contents of some part of main memory from being destroyed or altered.

Memory reference instructions, 13-14, to 13-21, 13-24, A2-1

Methods, FORTRAN pass 2 assembly, 14-44

Micro instructions

4K Assemblers 13-25

Mnemonics, 16-8

Mode, numeric conversion, 14-77

MODIFY command, FOCAL, 11-21, 11-22

Monitor: The master control program that observes, supervises, controls, or verifies the operations of a system. In TSS/8, controls the sequencing of user programs.

Most significant digit: The leftmost nonzero digit.

Multiple record formats

FORTRAN, 15-24

Multiple word instructions

SABR, 14-34

in listing, 14-45

Multiplication, 16-18

Multiprocessing: Utilization of several computers or processors to logically or functionally divide jobs or processes, and to execute them simultaneously.

Multiprogramming: Pertains to the execution of two or more programs kept in core at the same time. Execution cycles between programs.

Multi-user FOCAL Segments, 11-31

N

Nesting:

1. Including a program loop within another program loop. Note special rules for nesting FORTRAN DO-loops.
2. Algebraic nesting, such as $(A + B * (C + D))$, where execution proceeds from innermost to outermost level.

Nesting literals

MACRO, 13-49

Nesting loops

BASIC, 12-36

NEXT statement

BASIC, 12-35

NOP: An instruction that specifically instructs the computer to do nothing (control proceeds to the next instruction in sequence).

Normalization, 16-6

Normalize: To adjust the exponent and fraction of a floating-point quantity so that the fraction appears in a prescribed format.

Null lines, 14-10

Number base, see Radix.

Number formats

BASIC, 12-10

FOCAL, 11-8

MACRO, 13-41, 13-42

PAL III, 13-8

Numeric conversion mode, SABR, 14-77

Numeric input conversion

FORTRAN, 15-60

O

OBISUB Linkage routine, SABR, 14-35

Object program: The binary coded program which is the output after translation from the source language. The binary program which runs on the computer.

Octal: Pertaining to the number system with a radix of eight.

OCTAL pseudo-op, 13-28, 14-20

ODISK routine, SABR 14-82

FORTRAN, 15-31

Off-line: Pertaining to equipment or devices not under direct control of the computer.

- One's complement*, see Complement, (one's), (two's).
- On-line*: Pertaining to equipment or devices under direct control of the computer; also to programs operating directly and immediately to user commands, e.g., FOCAL and DDT.
- OPDEF pseudo-op, SABR, 14-23
- Operand*: That which is affected, manipulated, or operated upon. The address or symbolic name, portion of a PAL-III instruction. Assembler, 13-11
SABR, 14-12
incrementing, 14-17
- Operate micro instructions, 13-25, A2-2
- Operating instructions
BASIC, 12-45, 12-50
FORTRAN, 15-40, 14-49
MACRO, 13-58
PAL III, 13-36
SABR with FORTRAN, 15-43, 14-49
SABR, 14-47
Linking Loader, 14-59
FOCAL, 11-42
- Operator*: That symbol or code which indicates an action (or operation) to be performed, e.g., + or TAD: SABR, 14-12
- OPISUB linkage routine, 14-35
- Options, switch register See Switch Register Options.
- OR*: (Inclusive) A logical operation such that the result is true if either or both operands are true, and false if both operands are false. (Exclusive) A logical operation such that the result is true if either operand is true, and false if both operands are either true or false when neither is specifically indicated, the default case is inclusive OR.
- Order of operator evaluation
BASIC, 12-13
- Origin*: The absolute address of the beginning of a section of code.
- Origin setting
MACRO, 13-21, 13-45
- Output*: Information transferred from the internal storage of a computer to output devices or external storage.
- Output controller, 16-15
- Output formats, 14-38
BASIC, 12-28
- Overflow*: A condition that occurs when a mathematical operation yields a result whose magnitude is larger than the program is capable of handling. Floating-point, 16-13
- Overlay Combinations, FOCAL, 11-38, 11-39

P

- Page*: A 128-word section of core memory, beginning at an address which is a multiple of 200₈.
- Paging, SABR, 14-32
escapes, 14-33
format, 14-33
- PAGE pseudo-op, 4K Assemblers, 13-45; SABR, 14-21
- PAL III
see Table of Contents, 13-2
extended symbols tape, A2-8
programming, 13-7
symbol table, A2-1
- PAL-D, 4K
see Table of Contents, 13-3
programming, 13-65
symbol table, A2-1
- PAL-D, 8K, 14-5
see Table of Contents, 15-3
symbol table, A2-1
requirements, 14-9
- Paper tape system
FORTRAN, 15-39
- Parameters, SABR, 14-16
- Parentheses
BASIC, 12-13
MACRO, 13-48
FOCAL, 11-6
- Pass*: One complete cycle during which a body of data is processed. An assembler usually requires two passes during which a source program is translated into binary code. SABR, 14-47; 4K Assemblers, 13-35

- Patch*: To modify a routine in a rough or expedient way.
- PAUSE statement
 FORTRAN, 15-19
- PAUSE pseudo-op, 4K Assemblers, 13-30, SABR, 14-19
- Period (.), 4K Assemblers 13-22
- Percent sign, FOCAL, 11-8
- Peripheral equipment*: In a data processing system, any unit of equipment, distinct from the central processing unit, which may provide the system with outside storage or communication.
- Permanent symbols, A2-1
- Permanent Symbol Table, altering, 13-31
 MACRO, 13-56
- PLOTR, FOCAL, 11-37
- Pointer address*: Address of a core memory location containing the actual (effective) address of desired data.
- Priority interrupt*: An interrupt which is given preference over other interrupts within the system.
- Priority of operators
 BASIC, 12-13
- Procedure*: The course of action taken for the solution of a problem; also called an algorithm.
- Program*: The complete sequence of instructions and routines necessary to solve a problem.
- Program addresses
 SABR, 14-37
- Program control math routines, 16-26
- Program execution, SABR, 14-70
- Program interrupt instructions, A2-3
- Program Length, FOCAL, 11-40
- Program Listing, Unloaded, 14-67
- Program preparation, 4K Assemblers, 13-33, 13-66
- Pseudo floating accumulator, 16-6
- Pseudo-operation*: An instruction to the assembler; an operation code that is not part of the computer's operation repertoire as realized by the hardware. Also pseudo-op.
- Pseudo-ops, A2-6
 8K PAL-D, 14-6
 SABR, 14-18
 External subroutine, 14-26
- Pseudo-ops, A2-6
 DECIMAL, 13-28
 DEFINE, 13-53
 DUBL, 13-41
 EXPUNGE, 13-31, 13-56
 FIELD, 13-28, 13-65
 FIXMRI, 13-32
 FIXTAB, 13-31, 13-56
 FLTG, 13-43
 I, 13-22, 13-27
 OCTAL, 13-28
 PAGE, 13-45
 PAUSE, 13-30
 TEXT, 13-51
 XLIST, 13-65
 Z, 13-23, 13-27
- Punched paper tape*: A paper tape on which a pattern of holes is used to represent data.
- Punching paper tape
 BASIC, 12-50
- Pushdown list*: A list constructed and maintained so that the next item to be retrieved is the item most recently stored in the list.
- Q
- QUAD, FOCAL, 11-31
 error procedures, 11-51
 loading procedures, 11-46
 Teletype problems, 11-52
- Queue*: A waiting list. In time-sharing, the Monitor maintains a queue of user programs waiting for processing time.
- QUIT command, FOCAL, 11-13
- Quote ("), 13-51
- R
- Radix*: The base of a number system, the number of digit symbols required by a number system. See Binary, Octal.
- Random access*: A storage device in which the accessibility of data is effectively independent of the location of the data. Synonymous with direct access.
- Random number function
 BASIC, 12-16, 12-17
 FOCAL, 11-26

- Range, FORTRAN 15-18
- RDISK routine, SABR, 14-83
- FORTRAN, 15-32
- Read*: To transfer information from an input device to core memory.
- Reading a paper tape
- BASIC, 12-51
- FOCAL, 11-52, 11-54
- (silent)
- READ statement
- BASIC, 12-22
- FORTRAN, 15-27
- Real constants
- FORTRAN, 15-9
- Real-time*: Pertaining to computation performed while the related physical process is taking place so that results of the computation can be used in guiding the physical process.
- Real variables
- FORTRAN, 15-10
- Record*: A collection of related items of data, treated as a unit.
- **Recursive subroutine*: A subroutine capable of calling itself and returning, at some later point, to the program which initially called it.
- Register*: A device capable of storing a specified amount of data, usually one word.
- Register, 16-12 17.
- Relative address*:
1. The number that specifies the difference between the actual address and a base address.
 2. In the PDP-8 the character period (.) is used to represent the current location counter; addresses can be indicated relative to the current location counter (.+5 indicates five locations from the current location), or relative to an origin assigned by use of the asterisk.
- Relocatable*: Used to describe a routine whose instructions are written so that they can be located and executed in different parts of core memory. SABR, 14-9
- Relocation codes, Loader, 14-39
- REM statement
- BASIC, 12-9
- Removing a line of code
- BASIC, 12-42
- Removing program lines
- BASIC, 12-49
- REORG pseudo-op, 14-21
- Replacement operator, (=), 12-11
- Replacing a program on Disk
- BASIC, 12-47
- Response time*: Time between initiating some operation from a terminal and obtaining results. Includes transmission time to the computer, processing time, access time to file records needed, and transmission time back to the terminal.
- Restart*: To resume the execution of a program.
- FOCAL, 11-43
- RESTORE statement
- BASIC, 12-24
- RETRN pseudo-op, SABR, 14-29
- RETURN statement
- BASIC, 12-40
- FOCAL, 11-13
- Routine*: A set of instructions arranged in proper sequence to cause the computer to perform a desired task.
- RTAPE statement
- FORTRAN, 15-28
- RTN linkage routine, 14-36
- Run*: A single continuous execution of a program.
- RUN command
- BASIC, 12-46
- Run time*: The time in which a program is executed.
- Run-time linkage routines
- Disk Linking Loader, 14-56
- Linking Loader, 14-34, 14-52
- S
- SABR
- error messages, 14-71
- language, 14-9
- operating procedures, 14-46
- statements, 14-9 to 14-16
- symbol table, A2-1

- system requirements, 14-9
- Sample of assembly listing, 14-43
- SAVE command
 - BASIC, 12-47
- Saving a program on Disk
 - BASIC, 12-47
- Saving FOCAL Programs, 11-43
- Scalar variables
 - FORTRAN 15-10
- Segment:*
 1. That part of a long program which may be resident in core at any one time.
 2. To divide a program as in 1. into two or more segments or to store part of a program or routine on an external storage device to be brought into core as needed.
 3. A unit of disk storage on TSS/8, generally 400_q words.
- FOCAL, 11-34
- Serial access:* Pertaining to the sequential or consecutive transmission of data to or from core, for example, paper tape. Contrast with random access.
- SET statement, FOCAL, 11-7
- Seven-user FOCAL, 11-32
- Shift:* A movement of bits to the left or right frequently performed in the accumulator.
- Sign function
 - BASIC, 12-15
 - FOCAL, 11-25
- Signs, math routines, 17-26
- Simple programming languages
 - BASIC, 12-5
 - FOCAL, 11-5
- Simulate:* To represent the functioning of a device, system, or computer program with another system or program.
- Sine, Floating-Point, Package, 16-21, 16-29; FOCAL, 11-27
- Single step:* Operation of the computer in which each instruction is performed by setting the *single-step* or *single instruction* switch and repeatedly depressing CONTinue.
- SKIP instructions
 - SABR, 14-37
- SKPDF pseudo-op, SABR, 14-23
- Software:* The collection of programs and routines associated with the computer.
- Source language:* See language, source.
- Source program:* A computer program written in a source language.
- Spaces
 - BASIC, 12-13
 - FORTRAN, 15-6
- Special characters
 - 8K PAL-D, 14-5
 - SABR, 14-11
- Specification codes
 - FORTRAN, 15-21
- Specification statements
 - FORTRAN, 15-6, 15-33
- Square, Floating-Point, 16-8
- Square brackets
 - MACRO, 13-48
- Square root, Floating-Point, 16-8, 16-28; FOCAL, 11-25
- Statement:* An expression or instruction in a source language.
- STOP statement
 - FORTRAN, 15-20
- Stopping a run
 - BASIC, 12-44
- Storage allocation:* The assignment of blocks of data and instructions to specified blocks of storage.
- Storage allocation
 - FORTRAN, 15-53
- Storage capacity:* The amount of data that can be contained in a storage device.
- Storage, COMMON (SABR), 14-24
- Storage device:* A device in which data can be entered, retained, and retrieved.
- Storage, Floating-Point, 16-6, 16-17
- Storage map option, SABR, 14-52, 14-66
- Store:* To enter data into a storage device.
- String:* A connected sequence of entities, such as characters in a command string.

- Subprogram arguments, picking up, 14-30
 - Subprogram statements
 - FORTRAN, 15-6, 15-35
 - Subroutine, closed*: A subroutine not stored in the main part of a program. Such a subroutine is normally called or entered with a JMS instruction and provision is made to return control to the main routine at the end of the subroutine.
 - Subroutine, open*: A subroutine that must be relocated and inserted into a routine at each place it is used.
 - Subroutines, external, 14-26
 - BASIC, 12-37
 - external, 14-26
 - FORTRAN, 15-37
 - user, 16-15
 - Subscript*: A value used to specify a particular item in an array.
 - Subscripted variables
 - BASIC, 12-31
 - FOCAL, 11-14
 - FORTRAN, 15-11
 - Library subprograms, 14-78
 - Subtraction, Fl. pt., 16-18
 - Swapping*: In a time-sharing environment, the action of either temporarily bringing a user program into core or storing it on the disk or other system device.
 - Switch*: A device or programming technique for making selections.
 - Switch register options
 - Linking Loader
 - core availability, 14-52, 14-55
 - storage map, 14-52, 14-55
 - tape reader, 14-46, 14-54
 - SABR,
 - core availability, 14-52
 - overlay, 14-67
 - storage map, 14-52, 14-55
 - tape reader, 14-46, 14-54
 - Symbol definition, SABR, 14-23
 - Symbolic address*: A set of characters used to specify a memory location within a program. 4K Assembler, 13-13
 - Symbolic code, see Language, Source
 - Symbolic Editor*: A PDP-8 System Library program which helps users in the preparation and modification of source language programs by adding, changing, or deleting lines of text. 15-6
 - Symbolic instructions
 - 4K Assembler, 13-15
 - Symbolic tape format
 - 8K FORTRAN output, 14-48
 - Symbols
 - 4K Assemblers, 13-12
 - SABR, 14-12
 - equivalent, 14-13
 - flags, 14-13
 - permanent, 14-12
 - storage, 14-38
 - user defined, 14-13
 - FOCAL, 11-13, 11-14
 - Symbol table*: A table in which symbols and their corresponding values are recorded.
 - 4K Assemblers: altering, 13-31, Assembler, A2-1, MACRO, 13-56, PAL III, 13-16.
 - 8K Assemblers: 8K PAL-D, 14-7, SABR, 14-38, 14-47
 - System*: A combination of software and hardware which performs specific processing operations.
 - System configuration
 - Disk Linking Loader, 14-51
 - Linking Loader, 14-9
 - 12K PAL-D, 14-7
 - SABR, 14-9
- T
- TAB character
 - 4K Assemblers, 13-7
 - TAB function
 - BASIC, 12-30
 - Table*: A collection of data stored for ease of reference, generally an array.
 - Tabulations
 - 4K Assemblers, 13-8
 - Tag*: See Label
 - Tangent, 16-21
 - Tape Format,
 - FORTRAN, 14-49
 - SABR, 14-38

Teleprinter/punch instructions, A2-3
Temporary storage: Storage locations reserved for intermediate results.
Terminal: A peripheral device in a system through which data can either enter or leave the computer.
 Terminator. 16-11
 Text facility
 MACRO, 13-51
 SABR, 14-27
Time sharing: A method of allocating central processor time and other computer services to multiple users so that the computer, in effect, processes a number of programs simultaneously.
Time quantum: In time-sharing, a unit of time allotted to each user by the Monitor. (quanta, pl.)
Toggle: Using switches to enter data into the computer memory.
 Trace feature, FOCAL, 11-23
 Transfer of control
 BASIC, 12-37
Translate: To convert from one language to another.
 Trig functions, FOCAL, 11-28
Truncation: The reduction of precision by dropping one or more of the least significant digits: e.g., 3.141592 truncated to 4 decimal digits is 3.141.
 TSS/8
 BASIC, 12-5
 LOGIN procedure, 12-45
 PAL-D. 13-67
 Microinstructions, A2-7
 TYPE statement, FOCAL, 11-6
 Typeout subroutine, Floating-Point, 16-14, 16-15 timing, 16-16

U

Unconditional GOTO, FORTRAN, 15-16
 Unconditional transfer
 BASIC, 12-37
 Undefined addresses, 13-15
Underflow: A condition that occurs when a floating point operation

yields a result whose magnitude is smaller than the program is capable of handling.

Unloaded program listing, SABR, 14-67
 Up arrow (↑), BASIC, 12-44
User: Programmers and operators of PDP-8 computer systems.
 User-defined functions
 BASIC, 12-19
 User-defined macros, 13-52
 User-defined symbols, SABR, 14-13
 User program execution, SABR, 14-70
 User subroutines, Floating-Point, 16-15
 Utility Package loading procedures, FOCAL, 11-46

V

Variable: A symbol whose value changes during execution of a program.
 BASIC, 12-11
 Dummy, 14-28
 FORTRAN, 15-8, 15-10, 15-53
 FOCAL, 11-14, 11-13

W

WDISK routine. 14-83
 WDISK statement
 FORTRAN, 15-32
Word: In the PDP-8, a 12-bit unit of data which may be stored in one addressable location.
 4WORD, see Four (4) WORD.
 Write: To transfer information from core memory to a peripheral device or to auxiliary storage.
 WRITE statement
 FORTRAN, 15-27
 FOCAL, 11-13, 11-16
 WTape statement
 FORTRAN, 15-28

X

XLIST pseudo-op, 13-65

Z

Z pseudo-op, 13-23