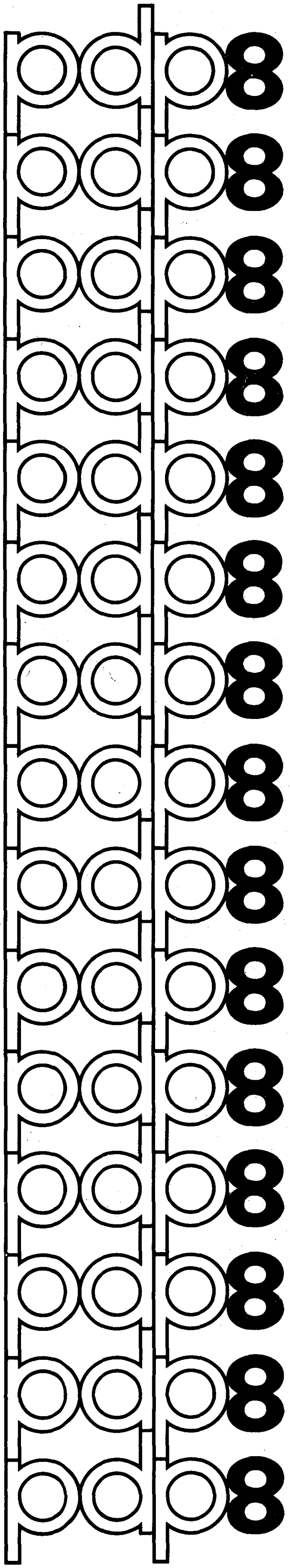
digital

8K fortran
sabr assembler



digital equipment corporation

DEC-08-LFTNA-A-D

8K FORTRAN
SABR ASSEMBLER

For additional copies, order No. DEC-08-LFTNA-A-D
from Software Distribution Center, Digital Equipment
Corporation, Maynard, Mass.

digital equipment corporation · maynard. massachusetts

First Printing, July 1973

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

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

PREFACE

The "HOW TO OBTAIN SOFTWARE INFORMATION" page, located at the back of this document, explains the various services available to DIGITAL software users.

The postage prepaid "READER'S COMMENTS" form on the last page of this document requests the user's critical evaluation. All comments received are acknowledged and will be considered when subsequent documents are prepared.

contents

CHAPTER 1 8K FORTRAN

CHAPTER 2 SABR ASSEMBLER

chapter 1
8K fortran

INTRODUCTION

This chapter presents a version of FORTRAN II specifically designed for the PDP-8/I, -8/L, -8, and -8/E computers, with at least 8K words of core memory, a Teletype, and a high-speed reader and punch. Although the information contained in this chapter deals particularly with the 8K FORTRAN available under the OS/8 Operating System, it is applicable as well to 8K paper tape FORTRAN. In cases where there are inconsistencies between the two, they are clearly noted.

"8K FORTRAN" 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 FORTRAN source program may be initially prepared off-line or by using the appropriate Editor program. 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 8K paper tape FORTRAN system consists of a one-pass FORTRAN compiler, the SABR Assembler, the Linking Loader, and a library of subprograms. Methods of loading and operating 8K paper tape FORTRAN are discussed toward the end of this chapter.

OS/8 8K FORTRAN is an expanded version of 8K paper tape FORTRAN which is designed to run under the OS/8 Operating System. It includes features not found in the paper tape version such as Hollerith constants, implied DO loops, chaining, mixing of SABR and FORTRAN statements, and device independent I/O. It is called from the OS/8 Keyboard Monitor. Complete operating instructions for the OS/8 FORTRAN system are found in the OS/8 chapter of *Introduction to Programming.*

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, and *A Guide to FORTRAN Programming* by Daniel D. McCracken, published by John Wiley and Sons, New York) if review is needed.

Character Set

The following characters are used in the FORTRAN language.[1]

1. The alphabetic characters, A through Z.
2. The numeric characters, 0 through 9.
3. The special characters:[2]

!
“
$
%
&
*
=
#
;
:
?
’
(
)
+
—
/
.
,
<
>
(space)
↑
[
]
\
←

FORTRAN Constants

Constants are self-defining numeric values appearing in source statements and are of three types: integer, real, and Hollerith.[3]

INTEGER CONSTANTS

An integer (fixed point) constant is represented by a digit string of from one to four decimal digits, written with an optional sign,

---

[1] Appendix B lists the octal and decimal representations of the FORTRAN character set.

[2] Of these, the characters ” ! $ % & # : ? < > ↑ [ ] \ ← may only appear inside FORMAT statements or Hollerith constants.

[3] Hollerith constants are available only in OS/8 FORTRAN.

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	(zero)

REAL CONSTANTS

A real constant is 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 of $\pm 1.7\times10^{38}$. (8K paper tape FORTRAN allows a range of $.14\times10^{-38}$ to 1.7×10^{38} for real constants.) For example:

+4.50	(+ is optional)
4.50	
−23.09	
−3.0E14	(same as -3.0×10^{14})

HOLLERITH CONSTANTS

A Hollerith constant is a string of up to 6 characters (including blanks) enclosed in single quotes. A Hollerith constant is treated like a real constant, except that it cannot be used in arithmetic expressions other than for simple equivalence (A=B). Any character except the quote character itself can be used in a Hollerith constant. For example:

```
'MOM'
'A+B=C'
'5 & 10'
```

FORTRAN 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. Although any number of characters may be used to make up the variable name, only the first five characters are interpreted as defining the name;

the rest are ignored. For example, DELTAX, DELTAY, and DELTA all represent the same variable name.

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

The name of an integer variable must begin with an I, J, K, L, M, or N. An integer variable undergoes 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 (see Arithmetic Statements). Integer variables must fall within the range −2047 to +2047.

2

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* or any text on binary arithmetic.

REAL VARIABLES

A real variable name begins with any alphabetic 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 may be either integer or real and represents a single quantity. For example:

7

```
LM
A
G2
TOTAL
J
```

ARRAY VARIABLES

An array (subscripted) 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).

Examples of one-dimensional arrays are:

```
Y(1)
PORT(K)
```

while a two-dimensional array appears as follows:

```
A(3*K+2,1)
```

Any array must appear in a DIMENSION statement prior to its 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:

column major

B(1, 1), B(2, 1), . . . , B(J, 1), B(1, 2), B(2, 2), . . . , B(J, 2),
. . . , B(J, K)

For representation of arrays of more than two dimensions, refer to the section entitled Representation of N-Dimensional Arrays toward the end of this chapter.

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 arithmetic 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 priority, 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 arithmetic 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 (see Function Calls):

```
A*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 yields a result of 1. Zero raised to any other power yields 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 does not cause an error message but uses the absolute value. Thus, the expression (−3.0)**3.0 yields a result of +27.

Any arithmetic expression may be enclosed in parentheses and be considered a basic element.

```
IFIX(X+Y)/2
(ZETA)
(COS(SIN(PI*EM)+X))
```

An arithmetic expression may consist of a single element (constant, variable, or function call). For example:

```
2.71828
Z(N)
TAN(THETA)
```

Compound arithmetic expressions may be formed using arithmetic operators to combine basic elements. For example:

```
X+3.
TOTAL/A
TAN(PI*EM)
```

Expressions preceded by a + or a − sign are also arithmetic expressions. For example:

```
+X
-(ALPHA*BETA)
-SQRT(-GAMMA)
```

As an example of a typical arithmetic expression using arithmetic operators and a function call, the expression for the largest root of the general quadratic equation:

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

is coded as:

```
(-B+SQRT(B**2-4.*A*C))/2.*A)
```

FORTRAN STATEMENTS

A FORTRAN source program consists of a series of statements, each of which must start on a separate line. Any FORTRAN statement may appear in the statement field (columns 7 through 72) and may be preceded by a positive number, called a statement number, of from 1 to 4 digits which serves as an address label and is used when referencing the statement. When used, statement numbers are coded in columns 1 through 5 of the 72 column line. Statement numbers need not appear in sequential order, but no two statements should have the same number. Statement numbers are limited to a value of 2047 or less.

When using the Symbolic Editor to create the source program, typing a CTRL/TAB (generated by holding down the CTRL key and depressing TAB) causes a jump over the statement number columns and into the statement field. Except for data within a Hollerith field (see Input/Output Statements), spaces are ignored by the compiler. The programmer may use spaces freely, however, to make the program listing more readable and to organize data into columns.

Line Continuation Designator

Statements too long for the statement field of a single Teletype line may be continued on the next line. The continued portion must not be given a line number, but must have an alphanumeric character other than 0 in column 6. If the Symbolic Editor is used, the programmer may type a CTRL/TAB followed by a digit from 1 to 9 before continuing the line. The continuation character is not treated as part of the statement.

For example, using spaces, a continued statement would look as follows:

```
      WRITE (3,30)
30    FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
     1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

Using tabs, the same statement would be typed:

```
        WRITE (3,30)
30      FORMAT (1X,'THE FOLLOWING DATA IS GROUPED INTO THREE
        1 PARTS UNDER THE HEADINGS X, Y, AND Z.')
```

There is no limit to the number of continuation lines which may appear. However, one restriction is that an implied DO loop must not be broken, but must be on one line. For ease in program correction, it is recommended that continuation lines be minimized.

Comments

The letter C in column 1 of a line designates that line as a comment line. A comment appears in a program listing but has no effect on program compilation. Any number of comment lines may appear in a given program, and comments that are too long for one line may be continued by placing a C in the first column of the next line. A comment line may not appear between another line and its continuation.

FORTRAN statements are of five types:

1. Arithmetic, defining calculations to be performed;
2. Input/Output, directing communication between the program and input/output devices;
3. Control, governing the sequence of execution of statements within a program;
4. Specification, describing the form and content of data within the program; Declare
5. Subprogram, defining the form and occurrence of subprograms and subroutines. Subr

Each of these five types is explained in the following paragraphs.

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. In the statement:

```
META=W*(ABETA+E)
```

since META is an integer and the expression is real, the expression value is truncated to an integer before assignment to META.

INPUT/OUTPUT STATEMENTS

Input/Output (I/O) statements are used to control the transfer of data beween computer memory and peripheral devices and to specify the format of the output data. I/O statements may be divided into two categories:

1. Data transmission statements, READ and WRITE, specify transmission of data between computer memory and I/O devices.
2. Nonexecutable FORMAT statements enable conversion between internal data (within core memory) and external data.

Data Transmission Statements

The two data transmission statements, READ and WRITE, accomplish input/output transfer of data listed in a FORMAT statement. The two statements are of the form:

READ (unit, format) I/O list
WRITE (unit, format) I/O list

where *unit* is a device designation which can be an integer constant or an integer variable, *format* is a FORMAT statement line number, and the *I/O* list is a list specifying the order of transmission of the 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.

An element in an I/O list can take one of the following forms:

1. Arithmetic expression: expressions more complicated than a single variable (which can be subscripted) are meaningless in an input operation.
2. The name of an array (1 or 2 dimensional)[4]: this indicates that every element of the array is to be transmitted. Elements are transmitted in the order in which they are stored in core.

   For example:

   ```
   DIMENSION A(2,2)
   READ (1,100) A
   ```

---

[4] Arrays in I/O lists are allowed only in OS/8 FORTRAN.

reads:

```
A(1,1),A(2,1),A(1,2),A(2,2)
```

3. Implied DO Loops[5] of the form:

$$(s_1,s_2,\ldots,s_n,i=m_1,m_2,m_3)$$

repeat the list elements (s_n) with the value of i being equal to m_1 through m_2 having an optional step value of m_3. The m's are integer constants or variables, i is an integer variable, and s_1-s_n are the I/O list elements (possibly including an implied DO loop). For example:

```
DIMENSION A(3,6)
WRITE (1,100) I,(A(J,I)J=1,3)
```

will output the values:

```
I,A(1,I),A(2,I),A(3,I)
```

It is important to remember that when using implied DO loops, the entire implied DO loop must be on the same input line or card. An implied DO loop cannot be continued onto the next line with a continuation character.

If no I/O list is specified for a WRITE statement, then information is read directly from the specified FORMAT statement and written on the device designated.

Data appears on the external device in the form of records.[6] 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 ASCII record is specified by the FORMAT statement and the I/O list.

---

[5] Implied DO loops are not allowed in 8K paper tape FORTRAN. Refer to Implementation Notes at the end of this chapter for a way of circumventing this restriction.

[6] This should not be confused with the OS/8 record, which is equal to 256_{10} words (2 DECtape blocks with the 129th word of each block ignored.)

Each execution of an I/O 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
```

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 I/O statement requests less than a full record of information, the unrequested part of the record is lost and cannot be recovered by another I/O statement without repositioning the record.

If an I/O list requires more than one ASCII 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 statement line number, and list is a list of variables whose values are to be input.

The READ statement causes ASCII 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 specifies transfer of 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 statement line number, and list is a list of variables to be output.

The WRITE statement followed by a list causes the values of the variables in the list to be read from memory and written on the designated device in ASCII form. The data is converted to external form as specified by the designated FORMAT statement.

The WRITE statement without a list causes information (generally Hollerith type) to be read directly from the specified format and written on the designated device in ASCII form.

DEVICE DESIGNATIONS

The I/O device designations used in the READ and WRITE statements are described in Table 1-1.

Table 1-1 Device Designations

Device Code	Input Designation	Output Designation
1	Teletype keyboard or low-speed reader	Teleprinter
2	High-speed reader	High-speed punch
3[7]	Card reader (CR8/I)	Line printer (LP08)
4[7]	Assignable device	Assignable device
	(see Device Independent I/O and Chaining)	

Device code 3 is assigned to the card reader (for all READ statements), and the line printer (for all WRITE statements). The card reader uses a two-page device handler, which is too large to

---

[7] Device designations 3 and 4 are available only in OS/8 FORTRAN.

be used with the device independent I/O feature (device code 4). Therefore, the card reader has its own device code.

The line printer is a separate output device because it can require special formatting, such as inserting a Form Feed to skip to the top of a page. The contents of the first column of any line is a control character. These control characters are never printed. They are as follows:

Character in Column 1	Resulting Spacing
space	single space
0	double space
1	skip to top of next page (Form Feed)
all others	single space

FORMAT Statement

The nonexecutable FORMAT statement specifies the form and arrangement of data on the selected external device. FORMAT statements are of the form:

m FORMAT $(S_1, S_2, \ldots S_n)$

where m is a statement number and each S is a data field specification. 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.

FORMAT statements may be placed anywhere in the source program. Unless the FORMAT statement contains only alphanumeric data for direct I/O 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, 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 reverts to the rightmost left parenthesis in the FORMAT statement. If an input list requires more characters than the input device supplies for a given record, blanks are inserted.

I/O information which is relevant to the 8K paper tape FORTRAN is contained in Implementation Notes at the end of this chapter.

NUMERIC FIELDS

Numeric field specification codes and the corresponding internal and external forms of the numbers are listed in Table 1-2.

Table 1-2 Numeric Field Codes

Conversion Code	Internal Form	External Form
E	Binary floating point	Decimal floating point[8] with E exponents: 0.324E+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
rIw

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:

---

[8] When using E format, or with numbers less than 1.0 when using F format in a WRITE statement, a zero will be typed to the left of the decimal point. This is not true in 8K paper tape FORTRAN, in which case the example given would by output as: .324E+10

```
FORMAT (I5,F10.2,E16.8)
```

could be used to output the line

```
   32     -17.60  0.59624575E+03
```

on the output listing.

The field width should always be large enough to include the decimal point, sign, and exponent (plus a leading zero in OS/8 FORTRAN). In all numeric field conversions, if the field width is not large enough to accommodate the converted number, asterisks will be printed; the number is always right-justified in the field.

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 which field digits fall. Thus, numbers are treated as if they are right-justified within a field. In an F5.2 format, the following:

   bbb12
   12bbb
   00012

   are read as the number 0.12 (where 'b' represents a blank space).

2. A null line delimited by two carriage return/line feed (CR/LF) combinations is treated as a line of blanks, and blanks are appended to the right of a line (if necessary) to fill out a FORMAT statement. Thus:

   12 (CR/LF)
   12bbb
   bbb12

   are identical under an F5.2 format. If an entire line is blank, numeric data from that line is read as zeros.

BAD

3. No distinction is made between E and F format on input. Thus:

   100.
   100E2
   1.E2
   10000

   are all read identically under either an F5.2 or E5.2 format.

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 I/O 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. The characters are stored in stripped ASCII format. If not enough data is supplied as input to the variables, the data is padded with blanks on the right. For example:

```
        READ (1,20) M1,M2,M3,M4,M5,M6,M7,M8
20      FORMAT (8A1)
```

if the user types at this point:

```
123ABC
```

followed by a carriage return, the following are the values of the variables:

Variable	Decimal	Octal	ASCII
M1	−928	6140	1
M2	−864	6240	2
M3	−800	6340	3
M4	96	0140	A
M5	160	0240	B
M6	224	0340	C
M7	−2016	4040	blank
M8	−2016	4040	blank

?

If the above had been read in 4A2 format, the values would be as follows:

Variable	Decimal	Octal	ASCII
M1	−910	6162	1 2
M2	−831	6301	3 A
M3	131	0203	B C
M4	−2016	4040	blanks
	....	....	
M8	−2016	4040	blanks

As a second example:

```
        READ (1,20) ALPHA
20      FORMAT (A6)
```

the user types:

```
123AB
```

and a carriage return, and the octal value of ALPHA is:

```
6162   6301   0240
```

NOTE

The numeric value of alphanumeric characters stored in floating point variables is generally not meaningful.

? Appendix B lists the octal and decimal (in A1 format) representations of the FORTRAN character set. The decimal representation applies only to 8K paper tape and OS/8 FORTRAN. ?

HOLLERITH CONVERSION

Alphanumeric data may be transmitted directly from the FORMAT statement by using Hollerith (H) conversion. H-conversion format is normally referenced by WRITE statements only.

In H-conversion, the alphanumeric string is specified by the form

$$nH\ h_1, h_2, \ . \ . \ . \ , \ h_n$$

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.

An attempt to use H format specifications with a READ statement will cause characters from the format field to be either printed or punched. This can be a useful feature since it provides a simple way of identifying data that is to be read from the Teletype keyboard. For example, the following instructions:

```
        READ (1,30)A,B
30      FORMAT (4HA = ,F7.2/4HB = ,F7.2)
```

cause A = and B = to be printed out before the data is read.

By merely enclosing the alphanumeric data in single quotes, the same result is achieved as in H-conversion; on input, the characters between the single quotes are typed as output 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')
```

causes PROGRAM COMPLETE to be printed. This method eliminates the need to count characters.

BLANK OR SKIP FIELDS

Blanks can be introduced into an output record or characters skipped on an input record by use of the nX specification. The number n indicates the number of blanks or characters skipped and must be greater than zero. For example:

```
FORMAT(5H STEPI5,10X2HY=F7.3)
```

can be used to output the line:

```
 STEP   28          Y=  3.872
```

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(218,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 CR/LF 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 parenthesis at the end of the format indicate the termination of a record. If the list of an I/O 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))
```

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
```

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:

$$\text{GO TO } (n_1, n_2, \ldots, n_k), J$$

where $n_1, n_2, \ldots, n_k$ are statement numbers and J is a nonsubscripted integer variable. This statement transfers control to the statement numbered $n_1, n_2 \ldots, n_k$ if J has the value 1, 2, . . . , 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:

$$\text{IF (expression) } n_1, n_2, n_3$$

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:

$$\text{DO } n\ i = m_1, m_2, m_3$$

where n is a statement number, i is a scalar 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 J=1,N
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. After the last execution of the range, control passes to the statement immediately following the terminal statement in what is called a *normal exit.* An exit may also occur by a transfer out of the range taking place before the loop has been executed the total number of times specified in the DO statement.

DO loops may be nested, or contained within one another, provided the range of each contained loop is entirely within the range of the containing DO statement. Nested DO loops may contain the same terminal statement, however. A transfer into a DO loop 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.[9] 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.

---

[9] After a normal exit from a DO loop, the index of the DO statement has the value of the index the final time through the loop plus whatever increment was assigned. For example:

```
DO 10 I=1,5
```

after a normal exit the value of the index is 6. However, it is good programming practice to avoid using the index as a variable following a normal exit until it has been redefined, as according to *ANSI FORTRAN Standards* the value is undefined.

The last statement of a DO loop must be executable, and must not be an IF, GO TO or DO statement.

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:

```
           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 affect FORTRAN object program operation; the END statement affects 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 button. For example:

```
PAUSE
CALL OPEN
```

NOTE

The CALL OPEN statement in OS/8 FORTRAN also resets all I/O on unit 4, the assignable channel. Any further READs or WRITEs on unit 4 without an intervening IOPEN or OOPEN will print an error message and abort.

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 either directed to a STOP statement or transferred around it.

END STATEMENTS

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. (In OS/8 FORTRAN, the END statement generates a STOP statement as well.)

SPECIFICATION STATEMENTS

Specification statements allocate storage and furnish information about variables and constants to the compiler. The specification statements are COMMON, DIMENSION, 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. Varibles 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:

$$\text{COMMON } v_1, v_2, \ldots, v_n$$

where v is a variable name. See the section entitled Common Storage Allocation for greater detail.

DIMENSION Statement

The DIMENSION statement is used to declare 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:

$$\text{DIMENSION } s_1, s_2, \ldots, s_n$$

where s is an array specification. For example:

```
DIMENSION A(100)
DIMENSION Y(10),PORT(25),B(10,10),J(32)
```

Dimension statements are used for the purpose of reserving sufficient storage space for anticipated data; it is the user's responsibility to see that his subscripting does not conflict with the DIMENSION statement declarations. For example:

```
DIMENSION I(10),J(10),K(10)
I(2,4)=2
J(12)=3
```

The above statements would assemble without error; at run time I(8) would be set equal to 2 and K(2) would be set equal to 3.

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. This is useful when the programmer desires to conserve storage space. The form of the statement is:

$$\text{EQUIVALENCE } (v_1, v_2 \ldots) , \ldots$$

where v represents a variable name. The inclusion of two or more variables within the parenthetical list indicates that these variables are to share the same memory location and thus have the same value. For example:

```
EQUIVALENCE(RED,BLUE)
```

The variables RED and BLUE are now of equal value. The subscripts of array variables must be integer constants. For example:

```
EQUIVALENCE(X,A(3),Y(2,1)),(BETA(2,2),ALPHA)
```

Because of core memory restrictions within the compiler, variables cannot appear in EQUIVALENCE statements more than once.

```
EQUIVALENCE(A,B,C)
```

is valid, but the statement:

```
EQUIVALENCE(A,B),(B,C)
```

would not compile correctly.

Variables may not appear in both EQUIVALENCE and COMMON statements.

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. Functions and subroutines must be compiled independently of the main program and then loaded together with the main program by the Linking Loader.

NOTE

Care should be exercised when naming a subprogram or subroutine. It must not have the same name as any of the FORTRAN library functions or subroutines, or assembler mnemonics or pseudo-ops, as errors are likely to result. The Library Functions are listed in this chapter, and the symbol table for the SABR Assembler is listed in Appendix C.

Subprogram definition statements may optionally contain dummy arguments representing the arguments of the subprogram. They are used as ordinary identifiers within the subprogram and are replaced by the actual arguments when the subprogram is executed.

Function Subprograms

A function subprogram is a subprogram which is called from an arithmetic expression within the main program and returns a single numeric value. 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 (a_1, a_2 . . . , a_n)

where FUNCTION (or FUNC) declares that the program which follows is a function subprogram, and identifier is the name of the function being defined. The identifier must appear as a scalar variable and be assigned a value during execution of the subprogram. This value is the function's value.

Arguments appearing in the list enclosed in parentheses are dummy arguments representing the function arguments. A function must have at least one dummy argument. The arguments must agree in number, order and type with the actual arguments used in the calling program. Function subprograms may be called with expressions and array names as arguments. The corresponding dummy arguments in the FUNCTION statement would then be scalar and array identifiers, respectively. Those 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. Dummy arguments to FUNCTION cannot appear in COMMON or EQUIVALENCE statements within the function subprogram.

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 name is SUM.

```
      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. Assume that the main program is set up 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 then calculates his gross pay with a statement such as the following:

```
GROSS = SUM(TIME,RATE)
```

and goes on to calculate withholdings, etc.

Subroutine Subprograms

A subroutine subprogram is a subprogram which is called by the main program via a CALL statement, and may return several or no values. The 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:

$$\text{SUBROUTINE identifier } (a_1, a_2 \ldots a_n)$$

where SUBROUTINE declares the program which follows to be a subroutine subprogram and the 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, if any, 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 are as follows:

```
SUBROUTINE FACTO (COEFF,N,ROOTS)
SUBROUTINE RESID (NUM,N,DEN,M,RES)
SUBROUTINE SERIE
```

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 ICHGE (I,J)
ITEM=I
I=J
J=ITEM
RETURN
END
```

The calling statement for this subroutine might look as follows:

```
CALL ICHGE (M,N)
```

where the values for the variables M and N are to be exchanged.

CALL STATEMENT

The CALL statement assumes one of two forms:

CALL identifier
or CALL identifier (a_1, a_2 . . . , a_n)

The CALL statement is used to transfer control to a subroutine subprogram. The identifier is the subroutine name.

The arguments (indicated by a_1, 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.

Function Calls

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:

IDENT (ARG, ARG, . . . , 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 1-3 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.

NOTE

A FORTRAN compiler and its corresponding Library constitute an interlocking set of programs. No user should attempt to compile a program under OS/8 and load it with the paper tape FORTRAN, or vice versa. Similarly, programs developed with the current FORTRAN compiler should not be run under an old FORTRAN system.

Table 1-3 Function Library

Function	Definition	Type of Argument(s)
ABS(x)	the absolute value of x	real
IABS(x)	the absolute value of x	integer
FLOAT(x)	convert x from integer to real format	integer
IFIX(x)	convert x from real to integer format	real
IREM(0)	remainder of last integer divide is returned[10]	integer
IREM(x/y)	remainder of x/y is returned[10]	integer
EXP(x)	exponential of x, e^x	real
ALOG(x)	natural logarithm of x, $\log_e x$	real
SIN(x)	sine of x, where x is given in radians	real
COS(x)	cosine of x, where x is given in radians	real
TAN(x)	tangent of x, where x is given in radians	real
ATAN(x)	arctangent of x, where x is given in radians	real
SQRT(x)	square root of x is returned	real

---

[10] If IREM is called as IREM (x/y), the remainder of x/y will be returned. If the argument of IREM does not contain a division, the remainder of the last integer division will be returned. Subsequent calls to IREM without a division being performed will return the value 0. If a READ or WRITE is executed after a division but before calling IREM, the value 0 will be returned.

Table 1-3 (Cont.) Function Library

Function	Definition	Type of Argument(s)
IRDSW(0)	read the console switch register, returning the decimal equivalence of the octal integer in the switch register. The switch register can be set before executing the FORTRAN program, or during execution using the PAUSE statement.	integer

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. With the exception of the arctangent function, which is accurate to seven places over the entire range, results of function operations are accurate to six decimal places.[11]

The floating point arithmetic routines check for both overflow and underflow. Overflow will cause the OVFL error message (or FPNT if using 8K paper tape FORTRAN) 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.

---

[11] The arctangent function in 8K paper tape FORTRAN is accurate to six decimal places for arguments whose absolute value is greater than .01.

DEVICE INDEPENDENT I/O AND CHAINING[12]

OS/8 FORTRAN provides for device independent, file-oriented, formatted I/O through use of the device number 4 in the READ and WRITE statements and several utility subroutines. These are described below.

The IOPEN Subroutine

The subroutine IOPEN prepares the system to accept input from a specified device when device code 4 is used in a READ statement. IOPEN takes two arguments which are interpreted as Hollerith strings. After a

```
CALL IOPEN(A,B)
```

any READ statement reading from device 4 will read from the file specified by B (which must have the extension .DA) on the device specified by A. For example:

```
CALL IOPEN('DTA5','INPUT')
```

will prepare for input from the file DTA5:INPUT.DA

```
CALL IOPEN('F1',0)
```

will prepare for input from the device F1, which, in this case, is a non-file-structured device.

If the file and device names are input via READ statements which use A format in their FORMAT statements, then A6 format must be used. @ signs rather than spaces should be used to fill in

---

[12] The information described in this section is available only in OS/8 FORTRAN.

empty characters. For example, the following statements are contained in a program:

```
        WRITE (1,20)
20      FORMAT ('ENTER FILE NAME')
        READ (1,22)FNAME
22      FORMAT (A6)
        CALL IOPEN('DSK',FNAME)
        .
        .
        .
```

The Teletype prints:

```
ENTER FILE NAME
```

and the user responds:

```
ABC@@@
```

The OOPEN Subroutine

The subroutine OOPEN prepares the system to send output to a specified device when device code 4 is used in a WRITE statement. The arguments of OOPEN are treated like those of IOPEN. Future WRITE statements using device 4 write on the device and file specified in the call to OOPEN. An error message is printed if the program has previously issued a CALL OOPEN without issuing a subsequent CALL OCLOSE. For example:

```
CALL OOPEN('PTP',0)
```

prepares device 4 to output on device PTP.

```
CALL OOPEN('SYS','LADE')
```

prepares device 4 to output to the file SYS:LADE.DA.

The OCLOSE Subroutine

The subroutine OCLOSE is called with no arguments. Its function is to terminate output on the output file opened by OOPEN. If OCLOSE is not called after a file has been written, that output file will never exist on the specified device.

The CHAIN Subroutine

A call to the subroutine CHAIN terminates execution of the calling program and starts execution of the core image on the system device as specified by the argument to CHAIN. Variables in common storage are not disturbed. For example:

```
CALL CHAIN('PROG2')
```

causes the file SYS:PROG2.SV to be loaded and started. Notice that PROG2 *must* be compiled and stored on the system device as a core image (.SV) file in order to be successfully accessed.

The EXIT Subroutine

To return to the Keyboard Monitor from a FORTRAN program, the EXIT subroutine is used, as follows:

```
CALL EXIT
```

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. For the paper tape FORTRAN system, these 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* (DEC-08-SUCO-D).

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 (n_1, n_2, n_3, n_4)

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 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words)[13], A, B, C, and D will require 16, 4, 4, and 1 blocks respectively.

---

[13] The block numbers used by RTAPE and WTAPE should not be confused with the record numbers used by OS/8. An OS/8 record is 256 words—roughly twice the size of a DECtape block. An RTAPE or WTAPE record number is exactly twice the corresponding OS/8 record number. For example, to read the first segment of the OS/8 directory on DECtape #5, the statements:

```
DIMENSION IDIR(258)
CALL RTAPE(5,2,258,IDIR)
```

would read Block 2 (OS/8 Block 1) of DECtape #5.

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 ten 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,-27,20,D)
```

The following section of a program demonstrates 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
```

OS/8 FORTRAN LIBRARY SUBROUTINES[14]

Table 1-4 contains a summary of the OS/8 FORTRAN library subroutines. This list describes the routines available under OS/8 FORTRAN, their functions, and other routines which must also be present in order for them to be used. The Subroutine Names listed are the files which comprise OS/8 Source DECtape #3 (available from the Software Distribution Center upon request).

Table 1-4 OS/8 FORTRAN Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
IOH	'READ' 'WRITE' 'IOH'	FLOAT UTILTY INTEGR	11	Handles Input and Output Conversion

---

[14] This table does not apply to 8K paper tape FORTRAN. The Subprogram Library for the paper tape version is available on two relocatable binary paper tapes. Part 1 contains those subprograms used by almost every FORTRAN/SABR program. The organization of the programs is described in the SABR chapter of this manual.

Table 1-4 (Cont.) OS/8 FORTRAN Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
FLOAT	'FAD' 'FSB' 'FMP' 'FDV' 'STO' 'FLOT' 'FLOAT' 'FIX' 'IFIX' 'IFAD' 'ISTO' 'ABS' 'CHS'	UTILTY	5	Floating Point Arithmetic Package
UTILTY	'OPEN' 'GENIO' 'EXIT' 'ERROR' 'CKIO'	INTEGR	3	FORTRAN Device Routines, Error Exit, Normal Exit
POWERS	'IFPOW' 'FFPOW' 'EXP' 'ALOG'	FLOAT UTILTY IPOWRS INTEGR	3	Handles Numbers to Floating Powers
INTEGR	'IREM' 'IABS' 'DIV' 'MPY' 'IRDSW' 'CLEAR' 'SUBSC'	UTILTY	2	Integer Math Package

Table 1-4 (Cont.) OS/8 FORTRAN Library Subroutines

Subroutine Name	Entry Points, or Defined External Symbols	Routines That are Pre-requisites	Core Require-ments (Pages)	Function the Routine Performs
TRIG	'SIN' 'COS' 'TAN'	FLOAT	2	Handles Sine, Cosine, and Tangent
ATAN	'ATAN'	FLOAT	2	Handles Arc-tangents
SQRT	'SQRT'	FLOAT UTILTY	1	Handles Square Roots
IPOWRS	'IIPOW' 'FIPOW'	FLOAT INTEGR	1	Handles Numbers to Integer Powers
IOPEN	'IOPEN' 'OOPEN' 'OCLOS' 'CHAIN'	UTILTY	1	OS/8 Device-Independent I/O, and Chaining Routines
RWTAPE	'RTAPE' 'WTAPE'	UTILTY	1	OS/ Independent DECtape I/O Routines

MIXING SABR AND FORTRAN STATEMENTS[15]

An S in column 1 of an input line identifies that line as containing SABR code. This feature is very useful for performing instructions which are undefined in the FORTRAN language. For example:

---

[15] Available only in OS/8 FORTRAN.

```
        DIMENSION M(10)
        .
        .
        J=M(1)
        DO 55 K=2,10
        L=M(K)
S       TAD      \L
S       AND      \J
S       DCA      \J
55      CONTINUE
```

This section of code will form the logical AND of M(1) through M(10) in the variable J.

Notice that whenever a FORTRAN variable is used in a SABR statement, the variable name is preceded by a backslash (\). FORTRAN line numbers referenced in SABR statements are also preceded by a backslash for identification purposes. (A backslash is produced by typing a SHIFT/L.)

Information on calling subroutines which are written in SABR assembly language from a FORTRAN program may be found in the SABR chapter of this manual.

SIZE OF A FORTRAN PROGRAM

The maximum size of any FORTRAN program is 36 octal or 30 decimal pages of code.

OS/8 can run FORTRAN programs in 8 to 32K of core. No one program or subprogram can be longer than 4K, however.

The user can estimate the size of his program as follows: Take the amount of core available on the system (at least 8K) and from it subtract 4K for the linkage subroutines, external symbol table, and I/O, math, error, and utility subroutines. From the remainder subtract the amount of storage required for data. The remaining space can be used to hold FORTRAN coding, at the rate of 50-70 FORTRAN statements per 1K of core.

One way to have a longer FORTRAN program in core than is usually possible is to divide a FORTRAN program into three chained segments:

Segment 1—inputs data into common storage
Segment 2—FORTRAN program for data processing
Segment 3—does output to desired device(s)

This gives two space advantages:

1. The entire program does not have to fit into available core, only the largest segment.
2. If no I/O statements are used in the middle (computational) segment, the I/O conversion routines will not be loaded with that segment. Since these routines occupy over 1100_{10} words, this technique allows the computational segment to be from 50 to 80 statements longer than a similar program containing I/O statements.

When chaining to a subroutine, the user must be sure he has compiled, loaded, and saved a complete runnable main program on the *system device*. This program is brought into core by the FORTRAN CHAIN subroutine.

Information concerning using FORTRAN or SABR with the interrupt on, or using PAL8 with SABR or FORTRAN can be found in the OS/8 chapter of *Introduction to Programming*.

OPERATING INSTRUCTIONS

The Compiler, SABR Assembler, and Linking Loader are used (in that order) to compile, assemble, and execute FORTRAN programs. Throughout 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.

Loading and Operating the Compiler

The following instructions for loading and operating the compiler apply only to 8K paper tape FORTRAN. (OS/8 operating instructions are found in *Introduction to Programming*.)

1. Make sure the Binary Loader is in memory, assume field 1.
2. Place the FORTRAN Compiler binary tape in the reader.
3. Set Switches 6-8 = 001.
4. Press EXTD ADDRess LOAD.
5. Set Switch Register = 7777.
6. Press ADDress LOAD.
7. If using a high-speed reader, depress Switch Register bit 0.
8. Press CLEAR and CONTinue.
9. 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.

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 ASCII 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.

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: switches 6-8 = 001, and switches 9-11 = 000.
2. Press EXTD ADDRess LOAD.
3. Place the FORTRAN program source tape in the reader, and press the punch ON.
4. Set Switch Register = 1000 (the compiler may also be started at location 5364 in field 0).
5. Press ADDress LOAD and CLEAR and CONTinue.
6. 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.

8K FORTRAN 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.

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 up arrow (↑) 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.

The compiler does not print messages for certain errors. This usually occurs due to one of three reasons:

1. Erroneous FORMAT statements or unbalanced DO statements—at compile time the processing of the FORMAT statements is superficial and errors will not be detected until execution.
2. No DIMENSION statement for subscripted variables—the variable is treated as a function name and will not be detected unless referenced. This can be checked by producing a loader map or list of undefined external symbols. (OS/8 provides a U option for producing a loader map, while this is available in 8K paper tape FORTRAN as a switch option. See the appropriate operating instructions.)
3. Undefined statement number—the compiler does not detect undefined statement numbers. These will be caught during assembly. Therefore, it is important to examine the assembly symbol table for undefined symbols and statement numbers before loading and executing the program. In OS/8

FORTRAN, if no symbol table printout is requested, the message U AT \ 10+0000 will occur where there is no statement numbered 10.

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)

When the paper tape FORTRAN compiler has finished punching both sections of tape it will halt. It may be restarted to compile additional programs by pressing CONTinue.

The paper tape FORTRAN compiler may be restarted at any time by pressing HALT and resetting the console switches.

OS/8 FORTRAN contains the following error messages in addition to those listed above:

Message	Explanation
I/O ERROR	A device handler has signalled an I/O error.
NO ROOM FOR OUTPUT	The file FORTRN.TM cannot fit on the system device.
SABR.SV NOT FOUND	The SABR Assembler is not present on the system device.
NO END STATEMENT	The input to the compiler has been exhausted.
COMPILER MALFUNCTION	The meaning of this message has been extended to cover

	various unlikely monitor errors.
SUBR. OR FUNCT. STMT. NOT FIRST	FORTRAN detected a SUBROUTINE or FUNCTION statement in the middle of a computation.

OS/8 FORTRAN LIBRARY ERROR MESSAGES

During execution, the various library programs check for certain errors and print error messages in the form:

XXXX ERROR AT LOC NNNNN

where XXXX is the error code and NNNNN is the location of the error. Table 1-5 summarizes the Library Error Messages.

Table 1-5 FORTRAN Library Error Messages

Error Code	Meaning
	The following errors are fatal and cause a return to the Keyboard Monitor.
ALOG	Attempt to compute log of negative number.
IOER	One of the following has occurred: 1. Device independent input or output attempted without /I or /O options, Refer to Chapter 9 of *Introduction to Programming* 2. Bad arguments to IOPEN or OOPEN, or 3. Transmission error while doing I/O.
CHER	File specified as argument to CHAIN not found on system device.
FMT1	Invalid Format Statement
	The following input errors are fatal unless input is coming from the Teletype, in which case the entire READ statement is tried again.
FMT2	Illegal character in I format.
FMT3	Illegal character in F or E format.
	The following errors do not terminate execution of the user's program.

Table 1-5 (Cont.) FORTRAN Library Error Messages

Error Code	Meaning
DIVZ	Division by zero—very large number is returned.
EXP	Argument to EXP too large—very large number is returned.
OVFL	Floating point overflow—very large number is returned.
FLPW	Negative number raised to floating point power—absolute value taken.
SQRT	Attempt to take square root of negative number—absolute value used.
FIX	Attempt to fix a number >2047; 2047 is returned.

In addition, the error message:

```
USER ERROR 1 AT 00537
```

means that the user tried to reference an entry point of a program which was not loaded.

To pinpoint the location of a library program execution error:

1. Determine, from the storage map, 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 subprogram containing the error from the location of the error indicated 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.

Loading the SABR Assembler

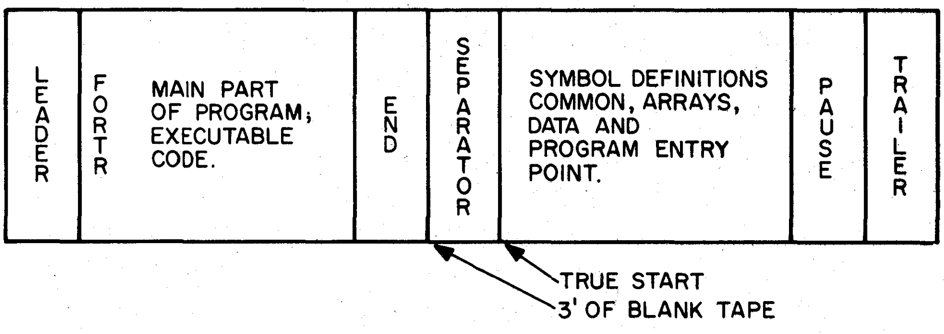
Procedures for loading SABR and assembling a source program are given below. See Appendix A for instructions on use of the Binary Loader. OS/8 SABR instructions are included in the OS/8 chapter of *Introduction to Programming.*)

1. Make sure the Binary Loader is in memory, assume field 1.
2. Set Switches 6-8 = 001.
3. Press EXTD ADDRess LOAD.
4. Set Switch Register = 7777.
5. Press ADDress LOAD.
6. Insert the SABR binary tape into the reader.
7. If using the high-speed reader, depress Switch Register Bit 0.
8. Press CLEAR and CONTinue.
9. SABR will now be loaded into memory by the Binary Loader. Portions of SABR will be loaded into both Field 0 and 1.

Operating the SABR Assembler[16]

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 the SABR chapter of this manual. 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 containing standard PDP-8 mnemonics. However, the symbolic tape produced by the compiler is not a standard format SABR language tape. It is arranged as shown in the following figure:



---

[16] Applies to 8K paper tape FORTRAN only.

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 the pass is completed. 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 and convert it into relocatable binary, one of three methods must be followed. The general procedure is the same as that described in the SABR chapter but in particular details it differs. The differences are covered by the three methods explained below.

METHOD 1

Cut the symbolic tape produced by the compiler into two parts. The cut should be made at the middle of the blank portion of tape which separates the executable code from the symbol definitions. The section containing the symbol definitions (the latter part of the tape) should be marked "Section 1," and the section containing the executable code marked "Section 2."

The first pass through SABR creates the relocatable binary version of the user's program; at the end of this pass, the symbol table may be typed and/or punched. Pass 2 creates the listing. Section 1 should be inserted in the reader before assembly is begun.

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 switches 6-8 = 0, and switches 9-11 = 0.
2. Press EXTD ADDRess LOAD.
3. Set the Switch Register = 0200.
4. Press ADDRess LOAD, CLEAR and CONTinue.
5. 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.

6. As soon as SABR has echoed the user's response to the last question, the punch device and, if it is being used, the Teletype 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 and will have to start over.
7. At this point, pass 1 begins. 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.
8. 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.
9. At the end of Pass 1 SABR halts.
10. 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 type) before beginning Pass 2.
11. At the end of Pass 2 SABR will again halt. It may be restarted for assembling another program by pressing CONTinue.
12. SABR may be restarted at any time by pressing HALT, setting the switch register =0200, pressing ADDress LOAD and CLEAR and CONTinue. 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. 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.

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 the user desires an assembly listing, he should proceed as in Step 10 of Method 1 after SABR finishes the assembly pass.

The Linking Loader

(The OS/8 Linking Loader is described in *Introduction to Programming*. For additional details concerning the 8K System Linking Loader, the reader is referred to Chapter 2 of this manual.)

Relocatable binary program tapes produced by SABR assemblies and the FORTRAN/SABR Library programs are loaded into memory by 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, and has options which give storage maps and core availability.

LOADING THE LINKING LOADER[17]

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:
   Switches 6-8 = m, and switches 9-11 = h.
4. Press EXTD ADDRess LOAD.
5. Set the Switch Register = 7777.
6. Press ADDress LOAD.
7. Place the binary paper tape of the Linking Loader in the reader.
8. If using a high-speed reader, depress switch register Bit 0.
9. Press CLEAR and CONTinue. 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

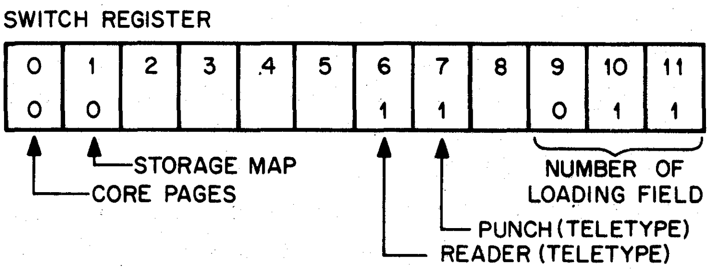
> The program or subprogram which uses the largest amount of common storage should be loaded first.

1. After the Linking Loader has been loaded into the highest memory field, h, the user should set the console switches as follows: Switches 6-8 = h and switches 9-11 = h.
2. Press EXTD ADDRess LOAD.
3. Set the Switch Register = 0200.
4. Press ADDress LOAD.
5. Place the relocatable binary tape for the first program to be loaded in the reader. Position the tape with leader code in the reader.
6. 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

---

[17] Applies to 8K paper tape FORTRAN only.

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.

7. Press CLEAR and CONTinue.
8. The user's relocatable binary program will now be loaded. When loading is completed, the Linking Loader halts.
9. The user may now either load another program or select one of the options in steps 11 and 12.
10. 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.
11. To select the Core Availability option, set switch register bit 0 = 1, and press CONTinue.
12. To select the Storage Map option, set switch register bit 1 = 1, and press CONTinue.
13. The user may continue loading more programs as in step 10 after using either of the options.
14. The Linking Loader may be restarted via the console switches at location 7200 (in the highest field, where the Linking Loader resides).

Executing the FORTRAN Program[18]

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

---

[18] Applies to 8K paper tape FORTRAN only.

1. Set switches 6-8 = d, and switches 9-11 = d.
2. Press the EXTD ADDRess LOAD.
3. Set Switch Register = nnnn.
4. Turn on paper tape punch and/or put data tape in reader as required.
5. Press ADDRess LOAD, CLEAR, and CONTinue. Program execution will begin.

DEMONSTRATION PROGRAM

This program computes the factorials of the even integers from 2 through 34. The MAIN program calls the subprogram to perform the computation. The source programs were created using the Symbolic Editor, listed on the Teletype for inclusion here, and punched on the high-speed punch. They were then compiled using the 8K paper tape FORTRAN Compiler on a PDP-8/I with 8K words of core memory, and a high-speed reader/punch.

This demonstration may also be run under the OS/8 Operating System. The only differences the user will note are that under OS/8 the operating process is considerably shorter, and the output contains leading zeros before the decimal point. A demonstration program is also contained in the OS/8 chapter of *Introduction to Programming.*

```
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,'! = ',E14.7)
        END
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,'! EXCEEDS CAPACITY OF PROGRAM.')
        END
```

The FORTRAN Compiler is loaded, and the Compiler types out an identification label such as the following:

```
PDP-8 FORTRAN DEC-08-A2B1-4
```

The source programs are compiled and tapes of the compiled programs are punched on the high-speed punch.

The SABR Assembler is loaded next. The tapes prepared by the Compiler are assembled, and the symbol tables listed on the Teletype:

```
PDP-8 SABR DEC-08-A2D2-16
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       0510
\A       0200
\N       0351
\10      0425
\60      0477
↑A       0361
↑B       0471
↑C       0410
↑D       0447
↑E       0462
↑F       0474
↑G       0510
```

```
HIGH SPEED READER?  Y
HIGH SPEED PUNCH? Y
LISTING ON  HIGH SPEED PUNCH? N


FACT     0220EXT
FAD      0000EXT
FLOT     0000EXT
FMP      0000EXT
IOH      0000EXT
OPEN     0000EXT
STO      0000EXT
WRITE    0000EXT
[0       0474
\C       0205
\FACT    0201
\K       0204
\M       0200
\N       0472
\1       0254
\2       0264
\3       0334
\4       0357
\5       0406
\6       0447
]3       0213
]6       0210
↑A       0310
↑B       0351
↑C       0422
↑D       0472
```

The Linking Loader is loaded. The FORTRAN/SABR Library programs and the binaries created by the SABR Assembler are loaded into core in fields 0 and 1; the switch register is set appropriately, and a memory map is typed. (In this case all the Library programs have been loaded—this is not necessary; if the user wishes to determine which Library programs his program will use, and how much core must be available, he may do so by using the memory map option and loading the appropriate programs into any fields available.)

```
PDP-8 LINKING LOADER DEC-08-A2C3-07

READ     10271
WRITE    10302
IOH      12142
SETERR   15200
ERROR    15303
TTYOUT   15027
HSOUT    15055
TTYIN    15000
HSIN     15045
FDV      13711
CLEAR    14227
IFAD     14116
FMP      13623
ISTO     14061
STO      13444
FLOT     14153
FAD      13010
DIV      14445
IREM     14616
FSB      13000
FLOAT    14034
FIX      13510
IFIX     13556
CHS      14211
ABS      14636
IABS     14670
MPY      14400
IRDSW    14713
OPEN     15125
CKIO     15121
EXIT     15142
CLRERR   15231
SUBSC    01000
IIPOW    01600
IFPOW    01662
FIPOW    01676
FFPOW    02050
EXP      01452
ALOG     01347
SQRT     02211
SIN      02673
COS      02663
TAN      02461
ATAN     03057
MAIN     03552
FACT     04020
0015
0010
```

Finally the starting address of the program is determined from the memory map (MAIN 03552), and execution is started at this location. The output is typed:

```
 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 program output.

STATEMENT AND FORMAT SPECIFICATIONS

Tables 1-6 and 1-7 summarize the statements and format specifications available in 8K FORTRAN.

Table 1-6 Statement Specifications

STATEMENT	FORM[19]		WHERE
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 $(n_1, \ldots, n_m)$, i	$1 \leqslant i \leqslant m$ and control goes to statement n_i . i is a nonsubscripted integer variable.

---

[19] R or P indicates a required or prohibited statement number. N indicates a nonexecutable statement.

Table 1-6 (Cont.) Statement Specifications

STATEMENT		FORM	WHERE
IF		IF (E) n_1, n_2, n_3	control goes to $\begin{matrix}n_1\\n_2\\n_3\end{matrix}$ if expression E $\begin{matrix}\leqslant\\=\\\geqslant\end{matrix}$ 0.
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) 1 WRITE (d, f) 1	d is device number, f is a FORMAT statement number and 1 is list of variable names separated by commas.
FORMAT	NR	FORMAT $(k_1, \ldots, k_n)$	k's are format specifications
COMMON	NP	COMMON a, b, . . . , n	a , . . . , n are nonsubscripted variable names
DIMENSION	NP	DIMENSION $a_1 (k_1), \ldots, a_n (k_n)$	a's are array names and k's are maximum subscripts.

Table 1-6 (Cont.) Statement Specifications

STATEMENT		FORM	WHERE
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.
CALL		CALL name (a_1 , . . . , 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 , . . . , v_n) , . . . , (v_m, . . . , v_n)	v's are variables or subscripted array names.

Table 1-7 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, (a leading zero in OS/8 FORTRAN), decimal point, and d is the number of characters in exponent.
Alphanumeric	rAw	r is the repetition count, w is field width.

Table 1-7 (Cont.) FORMAT Specifications

KIND	FORM	WHERE
H (Hollerith or Literal)	nHcharacters 'characters'	n is total number of characters including spaces 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.
Blank or Skip Fields	nx	n blanks (spaces are introduced into an output record or n characters skipped in an output record).

STORAGE ALLOCATION

Representation of Constants and Variables

INTEGERS

Integers are each allocated one machine word. They are represented in two's complement binary .

$\boxed{0\ 1 \qquad\qquad\qquad 11}$

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 $\boxed{1\ 11\ 111\ 111\ 110} + 1 = \boxed{1\ 11\ 111\ 111\ 111} = 7777_8$

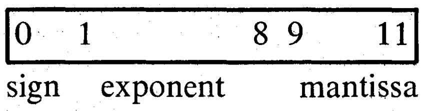
The largest negative number is -2048 which is represented by 4000_8 or

$\boxed{1\ 00\ 000\ 000\ 000}$

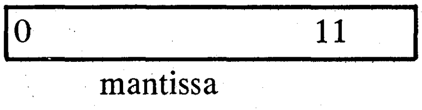
REAL NUMBERS

Real numbers are each allocated three machine words. They are represented as a binary mantissa multiplied by 2 raised to a binary exponent:

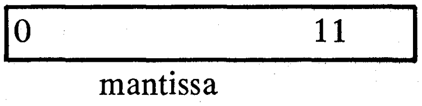
Word 1



Word 2

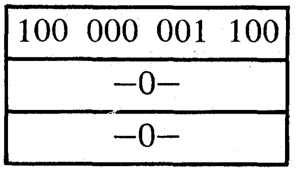


Word 3



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



Sign: 1_2
Exponent: $10\,000\,001_2$
Mantissa: $.100_2$
Exponent $= 201_8 - 200_8 = 1_8$
Mantissa $= .4_8$
No. $= -.4_8 \times 2_8{}^1$
$= -\frac{1}{2} \times 2 = -1$

Example 2

010 000 101 100
–0–
–0–

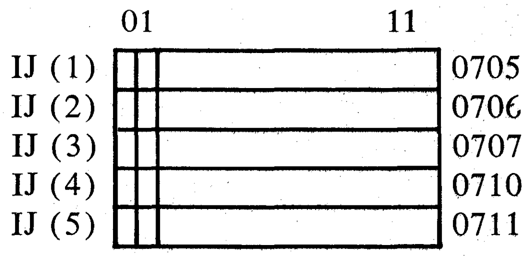
Sign: 0_2
Exponent: $10\ 000\ 101_2$
Mantissa: $.1_2$
Mantissa $= .4_8$
Exponent $= 205_8 - 200_8 = 5_8$
No. $= .4_8 \times 2_8{}^5$
$= \frac{1}{2} \times 32 = 16$

Storage of Arrays

Array variables are stored in core according to *ANSI FORTRAN Standards,* in columns and from top to bottom. For example, the array IJ:

```
DIMENSION IJ(5)
```

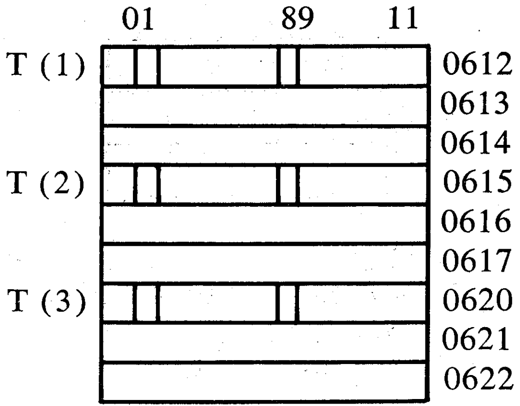
if started at location 0705 would be stored:



The real array, T:

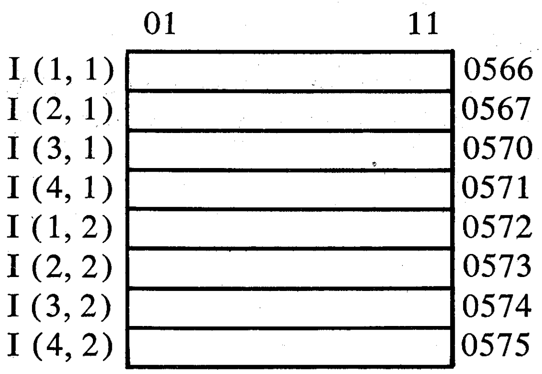
```
DIMENSION T(3)
```

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 in the preceding illustration. 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:

$$i_1+D_1*(i_2-1)+D_1*D_2(i_3-1)+...D_1*D_2...D_n(i_n-1)$$

where the subscript values are i_1, $i_2...i_n$ in an array whose dimensions are D_1, $D_2...D_n$.

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
```

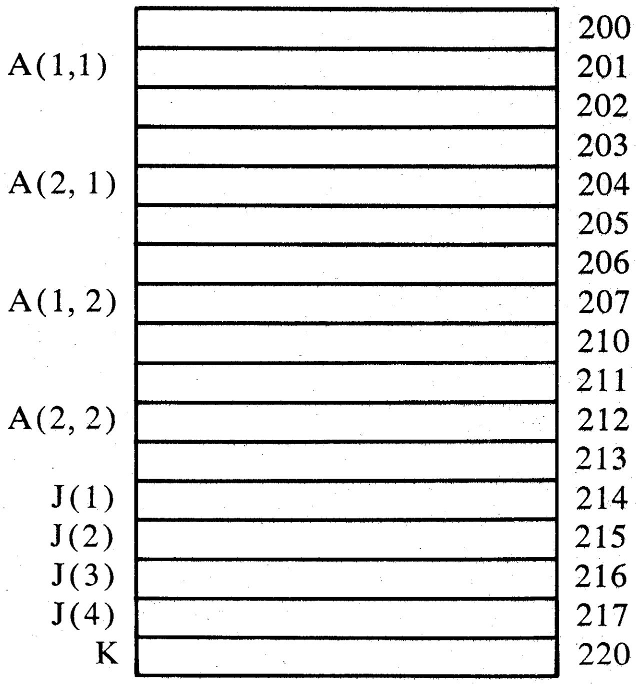
Common Storage Allocation

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:



NOTE

K does not appear in the DIMENSION statement.

If another subprogram defines a variable in common, such as the following:

```
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

8K FORTRAN (paper tape version) does not have implied DO loops in READ and WRITE statements. However, a simple way to circumvent this restriction has been implemented. Normally a 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 allowed in 8K paper tape 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.

The preceding comments apply as well to READ statements.

These methods are also useable in OS/8 FORTRAN, although the implied DO loop is preferred.

FORMAT Handling

For more complicated FORMAT handling the following technique can be used. For example:

```
        WRITE (1,20) (A(K),K=1,N)
20      FORMAT (F7.2,2E15.6)
```

which is not legal in 8K paper tape 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 [0
```

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 paper tape 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)
```

If desired, these methods may also be used with OS/8 FORTRAN.

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 UTILTY 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 2 for more information. (This applies only to 8K paper tape FORTRAN).
2. If it is desired to input or output on a special device but not in ASCII format write a subroutine to handle the particular device in the SABR assembly language. For more information refer to Chapter 2.
3. If it is desired to add devices which can be used in addition to those allowed 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 Software Distribution Center. 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. (This applies only to 8K paper tape FORTRAN.)
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.

Extra devices may be added to OS/8 FORTRAN by modifying the OS/8 FORTRAN Library routine GENIO. Only three more devices may be added, however, and these must have device numbers of 10, 11, and 12 respectively.

chapter 2
sabr assembler

SABR (Symbolic Assembler for Binary Relocatable programs) is an advanced, one-pass assembler producing relocatable binary code with automatically generated page and field linkages. It supports an extensive list of pseudo-operations which provide, among other facilities, external subroutine calling with argument passing and conditional assembly.

A SABR program may call routines from a large library of subroutines and functions; these are loaded together with the SABR program by the Linking Loader. In an optional second pass, SABR produces an octal/symbolic listing of assembled programs.

The relocatable binary tape produced by a SABR assembly is loaded into core for execution with the 8K Linking Loader. SABR and the Linking Loader are also incorporated in the OS/8 Operating System (see Chapter 9 of *Introduction to Programming)* and the 8K FORTRAN Operating System.

With the exception of their pseudo-operators, SABR and the PAL assembly languages share a common subset of instructions; the information contained in Chapters 1-5 of *Introduction to Programming* is prerequisite to the use of SABR.

In particular, SABR features include:

SABR produces relocatable binary code;

SABR is page and field independent—field settings and links are automatically generated, thus alleviating the programmer's need to consider page boundaries, and simplifying the development of programs greater than 4K;

SABR programs are loaded with the 8K Linking Loader and use run-time linkage routines provided by the Loader.

In general, a programmer might use SABR if he wants to write a program quickly without regard to page boundaries, and if he is not primarily concerned with program size. The programmer must also use SABR if he wants to write subroutines that can be called from a FORTRAN program.

SABR can be run on any PDP-8 series computer with at least 8K of core storage and a Teletype. A high-speed paper tape reader/punch is recommended.

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

SABR recognizes the numbers:

0–9

SPECIAL CHARACTERS

The following printing and non-printing characters are legal:

,	Comma	delimits a symbolic address label
/	Slash	indicates start of a comment
(	Left parenthesis	indicates a literal
"	Quote	precedes an ASCII 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 ASCII 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.

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 72_{10} characters long, including spaces and tabs. Any characters beyond this limit are ignored.

The RETURN key (CR/LF) is both an instruction 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
•
•
```

Using 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 preceding 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 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.

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 is a symbol or code which indicates an action or operation to be performed, and may be one of the following:

1. A direct or indirect memory reference instruction
2. An operate or IOT microinstruction
3. A pseudo-operator

All SABR operators, microinstructions and memory reference instructions are summarized in Appendix C.

OPERANDS

An operand represents that part of the statement which is manipulated or operated upon, and may be a numeric constant, a literal or a user-defined address symbol.

In the example last given, SAVE represents an operand.

Constants

Constants are data used but not changed by a program and are of two types: numeric and ASCII. ASCII constants are used only as parameters. Numeric constants may be used as parameters or as operand addresses, for example:

```
0200   1412                 TAD I    12
```

Constant operand addresses are treated as absolute addresses, just as a symbol defined by an ABSYM statement (see Symbol Definition). References to them are not generally relocatable, therefore, they should be used only with great care. The primary use of constant operand addresses is to reference locations on page 0 (see Linkage Routine Locations 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 (explained under Assembly Control). 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             -203
                        DECIM
0202   0120             80
```

ASCII Constants

Eight-bit ASCII values may be created as constants by typing the ASCII character immediately following a double quotation marks ("). 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 are illegal as alphabetic constants: carriage return, line feed, form feed and rubout.

Literals

A literal is a numeric or ASCII constant preceded by a left parenthesis. The use of literals provides a special and convenient way of generating constant data in a program. 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 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
```

The numeric conversion mode is initially set to octal, but is controllable with the DECIM and OCTAL pseudo-operators. This mode can be changed on a local basis by inserting a D (decimal) or a K (octal) between the left parenthesis and the constant. For example:

(D32 becomes 0040 (octal)
(K-32 becomes 7746 (octal)

This usage is confined only to the statement in which it is found and does not alter the prevailing conversion mode.

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, which is explained in greater detail later in the chapter.

Parameters

A parameter is generally either a numeric constant, a literal or a user-defined address symbol, which is intended to represent data rather than serve as an instruction. It appears as an operand in a statement line containing no operator. (An exception to this is a parameter used in conjunction with the ARG pseudo-operator, explained in Subroutines.) In the following example, 200 and −320, M, and PGOADR all represent parameters.

```
0200   0200       ABC,     200;-320;"M
0201   7460
0202   0315
0203   0176       POINTR, PGOADR
```

Symbols

Symbols are composed of legal alphanumeric characters and are delimited by a non-alphanumeric character. There are two major types of symbols: permanent, and user-defined.

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 C. These symbols may be used without prior definition by the user.

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:

1. The characters must be legal alphanumerics—
   ABCD . . . XYZ, [] \ ↑ and 0123456789.
2. The first character must be alphabetic.
3. 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.
4. A user-defined symbol cannot be the same as any of the pre-defined permanent symbols.
5. 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.

COMMENTS

A programmer may add notes to a statement by preceding them with 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 ALSO. 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              30
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 preceding example 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:

```
0202  4067       START,  TAD I LOC
0203  0200 01
0204  1407
0205  7000       VAR,    NOP
0206  1377               TAD (7500
0207  3205               DCA VAR
0377  7500
```

The #-sign feature is intended primarily for manipulating DUMMY variables when picking up arguments from external subroutines and returning from external subroutines (see Passing Subroutine Arguments).

Pseudo-Operators

Table 2-1 lists all the Pseudo-operators available in SABR, whether used as a free-standing assembler, or in conjunction with the Fortran compiler. The pseudo-operators are categorized and explained in the following paragraphs.

Table 2-1 SABR Pseudo-Operators

Mnemonic	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

Table 2-1 (Cont.) SABR Pseudo-Operators

Mnemonic		Operation
END		End of Program
ENTRY		Define Program Entry Point
FORTR		Assemble FORTRAN Tape
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 and is designed to allow the programmer to break up a large source tape into several smaller segments. To do this, the programmer need only place a PAUSE statement at the end of each section of his source program except the last. Each of these sections of the program is then output as an individual tape. When assembly halts at a PAUSE, the user removes the source tape just read from the reader and inserts the next one. Assembly may then be continued by pressing the 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. For example:

```
0200  0320       START,  320
                         DECIM
0201  0500               320
0202  0377 01            (K320
0203  1000               512
                         OCTAL
0204  0512               512
0205  0376 01            (D512
0206  0320               320
  .
  .                      END
  .
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 subprogram 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[1] 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

---

[1] 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.

PAGE pseudo-op had been encountered. The N words of code will then be assembled as a unit on the next core page. An example follows.

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.

```
0200  7200      START,  CLA
                        LAP        /INHIBIT PAGE ESCAPE
                        CPAGE 200  /CLOSES THE
0400  0000              NAME1      /CURRENT PAGE
0401  0000              NAME2      /AND 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 in this case 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
                        CLL RTL    /TO BE ASSEMBLED
                        RAL        /WILL BE "DCA LOC"
                /IF THE SLASH BEFORE "ABSYM NAME 176"
                /IS REMOVED, THE "CLL RTL" AND "RAL"
                /WILL BE ASSEMBLED.

0200  3201              DCA LOC
0201  0000      LOC,    0
                        .
                        .
                        .
```

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 under Linkage Routine Locations.
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:

```
0177     ABSYM TEM        177    /PAGE 0 ADDRESSES
0010     ABSYM AX         10
6761     OPDEF DTRA       6761   /NON-SKIP INSTR.
6771     SKPDF DTSF       6771   /SKIP-TYPE INSTR.
7540     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 are 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 as follows.

```
0200        A,       COMMN 20
0220        B,       COMMN 10
0230                 COMMN 300
0530        C,       COMMN 0
0530        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 allocated 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.)

```
0200  0000      LIST,    BLOCK 3   /ASSEMBLES AS
0201  0000
0202  0000

                                   /THREE ZEROS
                                   /WITH "LIST"
                                   /DEFINED AT THE
                                   /FIRST LOCATION
                NAME1,   BLOCK     /DEFINES NAME1=
                NAME2,   BLOCK 0   /NAME2=NAME3=
                NAME3,             /NAME4
0203  0000      NAME4,   BLOCK 2
0204  0000
```

TEXT

The TEXT pseudo-op is used to obtain packed six-bit ASCII 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 (ASCII 211) will be stored as an 11, which is equivalent to the coding for the letter I. In general, characters outside the ASCII 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
```

Subroutines

A subroutine is a subprogram which performs a specific operation and is generally designed so that it can be used more than once or by more than one program. Direction of flow goes from the main, or calling, program to the subroutine, where the action is performed, followed by a return back to the address following the subroutine call in the main program.

Internal subroutines are those subroutines which can only be called from within a program. This type of subroutine is used extensively in nearly all PDP-8 programs, and is handled through the use of the JMS, JMS I, and JMP I instructions. An example of an internal subroutine call follows:

```
0200  7300        START,  CLA CLL
0201  1204                TAD N      /GET NUMBER IN AC
0202  4206                JMS TWO    /TRANSFER TO SUB-
                                     /ROUTINE
0203  3205                DCA RESLT  /STORE NUMBER
                                     /(CONTROL RETURNS
                                     /HERE)
0204  0001        N,      1
0205  0000        RESLT,  0

                  /SUBROUTINE
0206  0000        TWO,    0
0207  7104                CLL RAL    /ROTATE LEFT AND
                                     /MULTIPLY BY 2
0210  7430                SZL        /CHECK FOR OVERFLOW
0211  7402                HLT        /STOP IF OVERFLOW
0212  6201 05             JMP I TWO  /RETURN TO MAIN
0213  5606
                                     /PROGRAM
                          END
```

The main program picks up a number (N) and jumps to the subroutine (TWO) where N is multiplied by two. A check is made, and if there is no overflow, control returns to the main program through the address stored at the location TWO.

External subroutines are distinguished from internal subroutines by the fact that they may be called by a program which has been compiled, or assembled, without any knowledge of where the subroutine will be located in core memory. Thus, external subroutines must be loaded with a relocatable linking loader. This makes it possible for a programmer to build a library of frequently used programs and subroutines which can be combined in various configurations, and eliminates the need to reassemble, or recompile, each individual program when a minor change is made in the system.

A call to an external subroutine can be illustrated using the following FORTRAN programs:

```
        IPARM=5                                (Calling Program)
        CALL TWO(IPARM)
        WRITE (1,100) IPARM
100     FORMAT (I5)
        END

        SUBROUTINE TWO(IARG)                   (Subroutine)
        IARG=IARG+IARG
        RETURN
        END
```

NOTE

> Care should be exercised when naming a function or subroutine. It must not have the same name as any of the assembler mnemonics or pseudo-ops or FORTRAN/SABR library functions or subroutines, as errors are likely to result. The symbol table for SABR Assembler is listed in Appendix C, and the library functions are described in the section The Subprogram Library.

Any time a subroutine is called, it must have data to process. This data is contained in parameters in the calling program which are then passed to the subroutine. The data is picked up by the subroutine where it is referred to as arguments. (The subroutine actually picks up the arguments by a series of TAD I's, and one final TAD I for an integer argument, or by a call to the IFAD subroutine if a floating point argument. This is illustrated in the section entitled SABR Programming Notes.) SABR has special pseudo-operators which facilitate the passing/handling of arguments, and each will be explained in turn.

CALL AND ARG

The CALL pseudo-op is used by the main program to transfer control to the subroutine and is of the form:

CALL n,NAME

where n represents a one or two-digit number (62_{10} maximum) indicating the number of parameters to be passed to the subroutine, and NAME (separated from n by a comma) represents the symbolic name of the subroutine entry point.

The Assembler must know the number of parameters which follow the call so that enough room on the current page can be allowed. The CALL pseudo-op and its corresponding parameters must always be coded on the same memory page; that is, there must be no intervening page escapes. (Page format and page escapes are discussed later in the chapter.)

The ARG pseudo-op is used only in conjunction with CALL and consists of the symbol ARG followed by one of the parameters (referred to as arguments in the subroutine) to be passed. One ARG statement must be coded for each parameter.

In the previous FORTRAN example, the main program (or it may have been a subroutine) called a subroutine named TWO, and supplied one argument:

```
CALL 1,TWO
ARG IPARM
•
•
•
```

SABR actually assembles the above instructions as follows (the user may wish to consult the section concerning the Loader Relocation Codes):

```
0200  0000        IPARM,  BLOCK 1
•
•
•
0206  4033                CALL 1,TWO
0207  0103 06
0210  6201 05             ARG IPARM
0211  0200 01
•
•
•
                  END
```

ENTRY AND RETRN

In the subroutine, the ENTRY statement must occur before the 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. Execution of the subroutine begins at the first location following the two-word ENTRY block. For example, the TWO subroutine mentioned in the previous example would begin as follows:

```
                          ENTRY TWO
0200  0000        TWO,    BLOCK 2
0201  0000
•
•
•
0227  4040                RETRN TWO
0230  0001 06
                          END
```

When a subroutine is referenced in a CALL statement, the Run-Time Linkage Routine LINK executes the transfer to the subroutine. It assumes that the entry point to the routine is a two-word block. Into the first word of this block it places a CDF instruction which specifies the field of the calling program. In the second word it places the address from which the CALL occurred. (This is analogous to the operation of the JMS instruction.) In the previous example, if the MAIN program had been in field 0, a 6201 would have been deposited in the location at TWO, and a 0210 at TWO #.

The RETRN statement allows the user to return to the calling program from the subroutine. The name of the subroutine being returned from must be specified in the RETRN statement so that the Return Linkage Routine can determine the action required, and also because a subroutine may have differently named ENTRY points. (This is analagous to the operation of a JMP I instruction.)

When a subroutine is entered, the second word of the entry name block contains the address of the argument or next instruction immediately following the subroutine call in the calling program, and it is to this address that control returns.

EXAMPLE

A user wishes to write a long main program, MAIN[2], which uses two major subroutines, S1 and S2. S1 requires two arguments and S2 one argument. The user writes MAIN, S1, and S2 as three separate programs in the following manner:

```
        ENTRY MAIN
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
```

---

[2] A useful procedure in SABR programming is to provide an ENTRY point named MAIN in the main program at the address where execution is to begin. This assures that the starting address of the program will appear in the Linking Loader's symbol print-out where it may be easily referenced. If using OS/8, execution will begin at this address automatically, eliminating the need to specify a 5-digit starting address.

S1 could also contain calls to S2, or S2 calls to S1. Each of these programs is independently assembled with SABR and loaded 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 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

DUMMY

A DUMMY pseudo-op is used in SABR to define a two word block which contains an argument address. Indirect instructions are used to pass arguments to and from subroutines through these DUMMY variables. If a DUMMY variable is referenced indirectly, it causes a CALL to the DUMMY Variable Run-Time Linkage Routine (see Run-Time Linkage Routines) which assumes that the DUMMY variable is a two-word reserved space where the first word is a 62N1 (CDF N), with N representing the field of the address to be referenced, and that the second word contains a 12-bit address.

As an example, consider the FORTRAN subroutine TWO shown earlier. This could be written in SABR as follows (the user may wish to refer to the section concerning the Subprogram Library):

```
                       /CALLED BY:  CALL TWO (IARG)

                                   ENTRY TWO      /DEFINE THE
                                                  /ENTRY PT. USED
                                   DUMMY IARG     /TO PICK UP ARG.
0200  0000            IARG,        BLOCK 2
0201  0000
0202  0000            TWO,         BLOCK 2        /ENTRY POINT
0203  0000
0204  4067                         TAD I TWO
0205  0202 01
0206  1407
0207  2203                         INC TWO#       /GET ARG ADDRESS
0210  3200                         DCA IARG
0211  4067                         TAD I TWO
0212  0202 01
0213  1407
0214  2203                         INC TWO#
0215  3201                         DCA IARG#
0216  4067                         TAD I IARG     /GET ARGUMENT
0217  0200 01
0220  1407
                                                  /INTO AC
0221  4067                         TAD I IARG     /ADD IT AGAIN
0222  0200 01
0223  1407
0224  4067                         DCA I IARG     /RETURN ARG. TO
0225  0200 01
0226  3407
                                                  /CALLING PROGRAM
0227  4040                         RETRN TWO
0230  0001 06
                                   END
```

A second example may be one in which a user has written a FORTRAN program which contains a call to a SABR subroutine ADD:

```
      A=2
      N=3
      CALL ADD(A,N,C)
      WRITE (1,20)C
20    FORMAT (' THE SUM IS',F6.1)
      STOP
      END
```

The FORTRAN program is compiled and the resulting SABR code translates the subroutine call as follows:

```
0223  4033       CALL 3,ADD
0224  0305 06
0225  6201 05    ARG A
0226  0200 01
0227  6201 05    ARG N
0230  0203 01
0231  6201 05    ARG C
0232  0204 01
```

The CALL statement defines 3 parameters—A, N, and C, and the subroutine name ADD. The subroutine itself would appear as follows (the DUMMY variables X, K, and Z facilitate the passing of the arguments to and from the subroutine):

```
                 /CALLED BY:  CALL ADD (X,K,Z)
                        ENTRY ADD
                        DUMMY X
                        DUMMY K
                        DUMMY Z
0200  0000      X,      BLOCK 2
0201  0000
0202  0000      K,      BLOCK 2
0203  0000
0204  0000      Z,      BLOCK 2
0205  0000
0206  0200 01   XPNT,   X
0207  0000      PNTR,   0
0210  0000      CNTR,   0
0211  0000      ADD,    BLOCK 2      /ENTRY POINT
0212  0000
0213  1206              TAD XPNT
0214  3207              DCA PNTR
0215  1377              TAD (-6
0216  3210              DCA CNTR
0217  4067      A1,     TAD I ADD
0220  0211 01
0221  1407
0222  2212              INC ADD#
0223  6201 05           DCA I PNTR
0224  3607
0225  2207              INC PNTR
0226  2210              ISZ CNTR
0227  5217              JMP A1
0230  4067              TAD I K      /GET 2ND ARG
0231  0202 01
0232  1407
0233  4033              CALL 0,FLOT  /CONVERT TO
0234  0002 06
                                     /FLOATING PT.
0235  4033              CALL 1,IFAD  /ADD 1ST ARG
0236  0103 06
0237  6201 05           ARG X
0240  0200 01
0241  4033              CALL 1,ISTO  /RETURN RESULT
0242  0104 06
0243  6201 05           ARG Z
0244  0204 01
0245  4040              RETRN ADD
0246  0001 06
0377  7772
                        END
```

The COMMN pseudo-op may be used to specify variables as externals so that they may be referenced by any program. This pseudo-op has been explained under Symbol Definition; an example of its usage is included here.

```
      0200       C,      COMMN 3          /RESERVES COMMON
                                          /STORAGE
                         ENTRY CSQR       /DEFINES ENTRY PT.
0200  0000       CSQR,   BLOCK 2          /ACTUAL ENTRY POINT
0201  0000
0202  4033               CALL 1,FAD       /GET THE ARGUMENT
0203  0102 06
0204  6211               ARG C
0205  0200
0206  4033               CALL 1,FMP       /MULTIPLY IT
0207  0103 06
0210  6211               ARG C
0211  0200
0212  4033               CALL 1,STO       /REPLACE WITH RESULT
0213  0104 06
0214  6211               ARG C
0215  0200
0216  4040               RETRN CSQR       /RETURN TO CALLING
0217  0001 06
                                          /PROGRAM
                         END
```

This subroutine computes the square of a variable C. C resides in field 1 in common storage where it can be referenced by any calling program through argument passing. The above is equivalent to the FORTRAN subroutine:

```
SUBROUTINE CSQR
COMMON C
C=C*C
RETURN
END
```

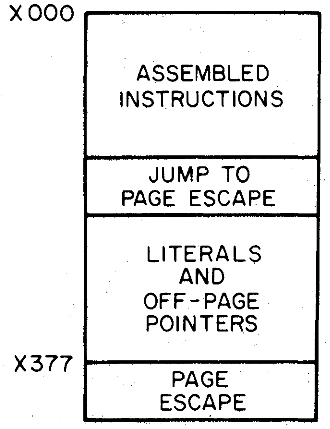
SABR Operating Characteristics

PAGE-BY-PAGE 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 are available 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. The user need be aware that extra instructions are automatically 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. (The user may wish to refer to the section entitled Loader Relocation Codes.)

There are seven linkage routines:

1. Change data field to current and skip	CDFSKP
2. Change data field to 1 (common) and skip	CDZSKP
3. Off-page indirect reference linkage	OPISUB
4. Off-bank (common) indirect reference linkage	OBISUB
5. Dummy variable indirect reference linkage	DUMSUB
6. Subroutine call linkage	LINK
7. Subroutine return linkage	RTN

The individual linkage routines function as follows:

1. 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.

Program	Assembled Code	Meaning
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.

2. 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 (see Loader Relocation Codes).

Program	Assembled Code	Meaning
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.

3. OPISUB is called when there is an indirect reference to an off-page location.

Program	Assembled Code	Meaning
DCA I PTR	4062	call OPISUB
	0300 01	relative address of PTR
	3407	execute the DCA I via 0007

4. OBISUB is called when there is an indirect reference to a location in common storage. 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.

Program	Assembled Code	Meaning
DCA I CPTR	4055	call OBISUB
	1000	address of CPTR in Field 1
	3407	execute the DCA I via 0007

5. 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.

Program	Assembled Code	Meaning
DCA I DMVR	4067	call DUMSUB
	0300 01	relative address of DMVR
	3407	execute DCA I via pointer in location 0007

6. 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-bit 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.

Program	Assembled Code	Meaning
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

7. RTN is called to execute the linkage by a RETRN statement in the user's program.

Program	Assembled Code	Meaning
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 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  0220  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
```

A special pseudo-operator, SKPDF, must be used to define skip instructions used in source programs but not included in the permanent symbol table. For example:

```
SKPDF DTSF 6771
```

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, then B will be loaded at 2000 + (1000-0200) 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.

The OS/8 version of SABR has a smaller space for symbol tables, leaving 1364_{10} words of storage, or 1620_{10} if used as the second pass of 8K FORTRAN.

Symbol table overflow is a fatal condition which generates the error message S.

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.

The Subprogram Library

The Library is a set of subprograms which may be CALLed by any FORTRAN/SABR program. These subprograms are automatically loaded with the OS/8 FORTRAN/SABR system; in the paper tape system they are provided on two relocatable binary paper tapes with part 1 containing those subprograms used by almost every FORTRAN/SABR program. This allows the user to load only those routines which his program makes use of, thus conserving symbol space.

Many of the subprograms reference the Floating-Point Accumulator located at ACH, ACM, ACL (20,21,22 of field 1). The OS/8 Subprogram Library is summarized in the 8K FORTRAN chapter. The organization of the library programs, as they are provided in the paper tape system, is described in the following pages.

Part 1.	"IOH"	contains	IOH, READ, WRITE
	"FLOAT"	contains	FAD, FSB, FMP, FDV, STO, FLOT, FLOAT, FIX, IFIX, IFAD, ISTO, CHS, CLEAR
	"INTEGER"	contains	IREM, 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 IIPOW, IFPOW, FIPOW, FFPOW, EXP, ALOG
"SQRT" contains SQRT
"TRIG" contains SIN, COS, TAN
"ATAN" contains ATAN

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 the 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          /TERMINATES READ
ARG     0
●●●
CALL    2, WRITE        /INITIALIZES WRITE
ARG     (n
ARG     fa
```

The following device numbers are currently implemented:

1 (Teletype keyboard/printer)
2 (High-speed reader/punch)
3[3] (Card reader/line printer)
4[3] (Assignable device)

FLOATING-POINT ARITHMETIC

FAD is called to add the argument to the Floating-Point Accumulator.

```
CALL    1, FAD
ARG     addres
```

FSB is called to subtract the argument from the Floating-Point Accumulator.

```
CALL    1, FSB
ARG     addres
```

FMP is called to multiply the Floating-Point Accumulator by the argument.

```
CALL    1, FMP
ARG     addres
```

FDV is called to divide the Floating-Point Accumulator by the argument.

```
CALL    1, FDV
ARG     addres
```

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 "addres".

STO is called to store the contents of the Floating-Point

---

[3] Device numbers 3 and 4 are available only under the OS/8 Operating System.

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
```

FLOAT and FLOT are 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  1, FLOAT
CALL    0, FLOT     or    ARG   addr
```

IFIX and FIX are called to convert the number in the Floating-Point Accumulator to a 12-bit signed integer and leave the result in the AC.

```
                        CALL  1, IFIX
CALL    0, FIX     or   ARG   addr
```

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, and can be used for doubly or singly subscripted arrays. On entry, the AC should be negative for floating-point variables—any negative number for singly subscripted variables, and 1's complement of the first dimension for doubly subscripted variables. For doubly subscripted integer variables, the AC must be the first dimension.

The general calling sequence for SUBSC is as follows:

---

*Applies to OS/8 only.

```
*TAD (M         /1ST DIMENSION (USED ONLY
                /IF 2 DIMENSIONS)
*CMA            /USED ONLY IF ARRAY IS
                /FLOATING POINT
    ┌2,SUBSC┐   /SINGLE SUBSCRIPT
 CALL│      │
    └3,SUBSC┘   /DOUBLE SUBSCRIPT
*ARG J          /2ND DIMENSION
 ARG I          /1ST DIMENSION
 ARG BASE       /BASE ADDRESS OF ARRAY
 LOCA           /ADDRESS OF TWO WORD DUMMY
                /ADDRESS LOCATION
```

* Optional Statements.

For example, to load the I,J th element of a floating-point array whose dimensions are 5 by 7:

```
TAD (5
CMA                /DIMENSIONS ARE 5 BY 7
CALL 3,SUBSC
ARG J              /ADDRESS OF 2ND SUBSCRIPT
ARG I              /ADDRESS OF 1ST SUBSCRIPT
ARG ARRAY          /BASE ADDRESS OF ARRAY
LOC                /MUST BE A DUMMY VARIABLE
CALL 1,IFAD
ARG LOC
```

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 at "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.

The POWER routines (IIPOW, IFPOW, FIPOW, FFPOW) are called by FORTRAN to implement exponentiation. 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
IIPOW	INTEGER	INTEGER	INTEGER
IFPOW	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
```

UTILITY ROUTINES

OPEN is called at the beginning of every FORTRAN program to start the high-speed reader/punch and teleprinter, and to initialize the I/O routines for device code 4 if using the OS/8 FORTRAN/SABR system. The form is:

```
CALL 0,OPEN
```

When an error is encountered in a program, the ERROR routine is called. The program passes to the ERROR routine the address of the error message to be printed. The format of the error message is 4 characters in stripped ASCII and packed into 2 words:

```
                                ENTRY ABC
2343   0102        XYZ,         0102;0304
2344   0304
2345   0000        ABC,         BLOCK 2
2346   0000        .
                   .
                   .
                                CALL 1,ERROR
                                ARG XYZ
```

When control passes to the ERROR routine, the parameters passed are picked up. In the case above, the parameters are as follows:

```
62N1                 ARG XYZ
2343
```

where N is the field that XYZ is in, and 2343 is the address of XYZ. The ERROR routine then prints the message at location 2343 plus a 5-digit address which is 2 greater than 2343.

```
ABCD ERROR AT N2345
```

Since XYZ is 2 locations before ABC, the address printed will be the address of ABC.

The error message is usually placed just before the entry point of the routine in which the error was detected—thus the address printed by ERROR will be the address of the entry point. This provides a convenience to the programmer since the entry point will appear in the Loader Map.

CKIO is a subroutine which waits for the TTY flag to be set. It is called by the OS/8 EXIT subroutine to eliminate the possibility of a garbled TTY output. It may be used in FORTRAN for possible expansion with interrupts, and is of the form:

```
CALL 0,CKIO
```

The following subroutines—IOPEN, OOPEN, OCLOSE, CHAIN, EXIT, and GENIO—are used by the OS/8 FORTRAN/SABR Operating System for device independent I/O and chaining. They are discussed in detail in Chapter 1 of this manual.

DECAPE 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:

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.
3. Number of words to be transferred ($1 \leqslant N \leqslant 4096$)
4. Core address at which the transfer is to start.

DECtape I/O Routines for the 8K FORTRAN system are explained in Chapter 1. In 8K SABR, the CALL statements to RTAPE and WTAPE 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 1474 blocks (numbered 0-2701 octal) of 129 words each (for a total of 190,146 words), A, B, C, and D will require 16, 4, 4, and 1 blocks respectively. (The block numbers used by RTAPE and WTAPE should not be confused with the record numbers used by OS/8. A OS/8 record is 256 words—roughly twice the size of a DECtape block.)

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))
```

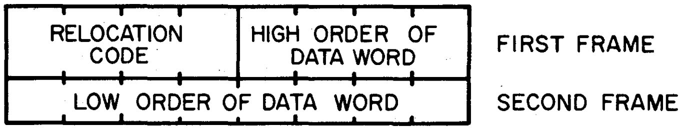
As shown above, it is possible to read or write less than 129 words by starting at the beginning of a DECtape block. It is impossible, however, to read or write starting in the middle of a block. For example, the last 10 words of a DECtape block may not be read without reading the first 119 words as 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 ten 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:

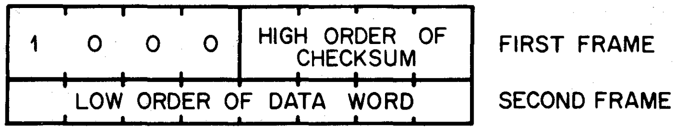
```
CALL RTAPE (0,1,2000,A)
CALL RTAPE (0,-27,20,D)
```

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
                                /AT 0077 (OF
                                /CURRENT PAGE)
```

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 stripped ASCII code for the symbol.

---

* Does not appear in assembly listings.

<table>
<tr><td>03</td><td>ADDRESS</td></tr>
<tr><td colspan="2">ADDRESS LOW ORDER</td></tr>
<tr><td colspan="2">L</td></tr>
<tr><td colspan="2">O</td></tr>
<tr><td colspan="2">C</td></tr>
<tr><td colspan="2">2</td></tr>
<tr><td colspan="2">SPACE</td></tr>
<tr><td colspan="2">SPACE</td></tr>
</table>

04	Re-origin*	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
                                  /INSTR. MUST BE
                                  /INDIRECT
0376   0520 01
```

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. It is generated automatically by

---

* Does not appear in assembly listings.

		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.
06	Subroutine Linkage Code	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.

BITS 0-5	BITS 6-11
NUMBER OF ARGUMENTS FOLLOWING THE CALL	LOCAL PROGRAM NUMBER ASSIGNED TO THE EXTERNAL SUBROUTINE BEING CALLED

Before the 12-bit, two-part code word is loaded into memory, a global external number will be substituted for the local external symbol 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 06 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. The checksum is contained in the last two frames of paper tape preceding the trailer code.
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 word 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 ASCII code for the symbol. 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 Listings

The following examples are offered to illustrate many of the features and formats of the SABR Assembler. Loading and operating instructions immediately follow this section.

When a multiple word instruction occurs, the actual instruction line is typed beside the first instruction.

---

* Does not appear in assembly listings.

```
0650  6201 05   LOC2,    JMP NAME  /OFF PAGE
0651  5774
0652  7106               CLL RTL;RTL;RTL
0653  7006
0654  7006
```

When there is 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                         /SKP TO 1ST LOC.-
                                   /NEXT PAGE (AC IS
                                   /NOT MINUS)
0777  7410                         /SKP TO 2ND LOC.-
                                   /NEXT PAGE (AC IS
                                   /MINUS)
```

Locations 0772, 0773, 0776 and 0777 make up the page escape since the last instruction is a skip instruction (SMA). Refer to the section concerning Page Escapes.

The following program has been assembled and listed. It cannot be run without first debugging and editing it.

During the first pass, SABR outputs the binary tape and prints error messages as they occur. In this case, none of the errors are fatal, and assembly continues. The symbol table is printed, and undefined symbols, external symbols, or any other special types of symbols which cannot be determined until the end of the pass are flagged in the symbol table.

The optional second pass of the Assembler produces a listing. The 4-digit first column contains the octal address, while the second column contains the octal code for each line of instructions. Errors are also printed during the listing pass at the line in which they occur. Meanings of error codes are described later in the chapter.

The reader is also referred to Demonstration Program Using Library Routines.

```
C AT  PUNCH +0003

COUNT   0302
DECIMA  0000UNDF
LT      0264
MAIN    0000EXT
MESG    0243
ORG     0303
PTAPE   0201EXT
PUNCH   0274
REF     0177ABS
RPT     0267
START   0205
TYPE    0000EXT


                /PROGRAM TO PUNCH RIM FORMAT PAPER TAPES

      6026              OPDEF PLS 6026   /DEFINE HI SPEED
      6021              SKPDF PSF 6021   /IOTS
      0177              ABSYM REF 177
                        ENTRY MAIN
0200  0000              DECIMAL
                        LAP
0201  0000      PTAPE,  BLOCK 2          /PUNCH LEADER
0202  0000
                                         /TAPE (200 CODE)
0203  1377              TAD (-32         /32 LOCATIONS
0204  3302              DCA COUNT
                        OCTAL
0205  1303      START,  TAD ORG
0206  7132              CLL CML RTR;RTR;RTR
0207  7012
0210  7012
```

```
0211  0376                AND (177
0212  4274                JMS PUNCH       /PUNCH LEADING
0213  1303                TAD ORG         /DIGITS OF ADDRESS
0214  0376                AND (177        /PUNCH SECOND
0215  4274                JMS PUNCH       /DIGITS OF ADDRESS
0216  1703                TAD I ORG       /NOW PUNCH CONTENTS
0217  7112                CLL RTR;RTR;RTR /OF THAT LOCATION
0220  7012
0221  7012
0222  0375                AND (77
0223  4274                JMS PUNCH
0224  1703                TAD I ORG       /GET SECOND DIGITS
0225  0375                AND (77         /OF THAT LOCATION
0226  4274                JMS PUNCH
0227  2303                INC ORG         /POINT TO NEXT
                                          /CORE LOCATION
0230  2302                ISZ COUNT       /DONE YET?
0231  5205                JMP START       /NO
0232  4033                CALL 1,TYPE     /YES, TYPE MESSAGE
0233  0102 06
0234  6201 05             ARG MESG
0235  0243 01
0236  4264                JMS LT          /ENDING 200 CODE
0237  7404                OSR             /GET NEW ADDRESS
0240  3303                DCA ORG         /FROM SWITCH REGISTER
                                          /PUT IT IN ORG
0241  7402                HLT             /PAUSE
0242  5774                JMP MAIN        /PUNCH NEW TAPE

0243  2401       MESG,    TEXT "TAPE PUNCHED. ENTER ORIGIN & CONT."
0244  2005
0245  4020
0246  2516
0247  0310
0250  0504
0251  5640
0252  0516
0253  2405
0254  2240
0255  1722
0256  1107
0257  1116
0260  4046
0261  4003
0262  1716
0263  2456

0264  0000       LT,      0
                          OCTAL
0265  1373                TAD (-40
0266  3302                DCA COUNT       /32 FRAMES OF
0267  1372       RPT,     TAD (200        /LEADER/TRAILER
0270  4274                JMS PUNCH       /PUNCH IT
0271  2302                ISZ COUNT       /DONE?
0272  5267                JMP RPT         /NO
0273  5664                JMP I LT        /RETURN
```

```
0274  0000      PUNCH,  0
0275  6026              PLS             /PUNCH
0276  6021              PSF             /WAIT FOR FLAG
   C                    JMP .-1
0277  4045              JMP I PUNCH     /EXIT
0300  7410
0301  5674

0302  0000      COUNT,  0
0303  7300      ORG,    7300
0304  4040              RETRN PTAPE
0305  0003 06
0372  0200
0373  7740
0375  0077
0376  0177
0377  7746
                        END
```

SABR Programming Notes

OPTIMIZING SABR CODE

There are generally two types of programmers who will use the SABR Assembler—those who like the convenience of a page-boundary-independent code and need not be concerned with program size, and those who need a relocatable assembler, but are still very location conscious. These optimizing hints are directed to the latter user.

One way to circumvent the cost of non-paged code is to make use of the LAP (Leave Automatic Paging) pseudo-op and the PAGE pseudo-op to force paging where needed. This saves 2 to 4 instructions per page by elimination of the page escape. In addition, the fact that the program must be properly segmented may save a considerable amount.

Extra core may be reduced by eliminating the CDF instructions which SABR inserts into a program. This is done by using "fake indirects". Define the following op codes:

```
OPDEF ANDI 0400
OPDEF TADI 1400
OPDEF ISZI 2400
OPDEF DCAI 3400
```

These codes correspond to the PDP-8 memory reference instructions but they include an indirect bit. The difference can best be illustrated by an example:

If X is off-page, the sequence:

```
LABEL,  SZA
        DCA X
```

is assembled by SABR into:

```
LABEL,  SZA
        JMS 45
        SKP
        DCA I (X)
```

or four instructions and one literal.

The sequence:

```
PX,     X
        .
        .
        .
LABEL,  SZA
        DCAI PX
```

assembles into three instructions for a saving of 40 percent. Note, however, that the user *must* be sure that the data field will be correct when the code at LABEL is encountered. Also note that SABR assumes that the Data Field is equal to the Instruction Field after a JMS instruction, so subroutine returns should not use the JMP I op code.

The standard method to fetch a scalar integer argument of a subroutine in SABR is:

```
                                  DUMMY X
0200   0000          IARG,        0
0201   0000          X,           BLOCK 2
0202   0000
0203   0000          SUBR,        BLOCK 2
0204   0000
0205   4067                       TAD I SUBR
0206   0203 01
0207   1407
0210   3201                       DCA X
0211   2204                       INC SUBR#
0212   4067                       TAD I SUBR
0213   0203 01
0214   1407
0215   3202                       DCA X#
0216   2204                       INC SUBR#
0217   4067                       TAD I X
0220   0201 01
0221   1407
0222   3200                       DCA IARG
                                  .
                                  .
                                  .
```

This is the method the FORTRAN compiler uses, and although it is standard, it is also the slowest. This code requires 19 words of core and takes several hundred microseconds to execute.

The fastest way to pick up arguments within a SABR coded external subroutine is as follows (this takes approximately one fifth of the time of the previous method and four less locations):

```
0200   0000          IARG,        0
0201   0000          SUBR,        BLOCK 2
0202   0000
0203   1201                       TAD SUBR
0204   3205                       DCA X1
0205   7402          X1,          HLT                 /REPLACED
                                                      /BY CDF
0206   1602                       TADI SUBR#
0207   3214                       DCA X2
0210   2202                       INC SUBR#
0211   1602                       TADI SUBR#
0212   3200                       DCA IARG
0213   2202                       INC SUBR#
0214   7402          X2,          HLT                 /REPLACED
                                                      /BY CDF
0215   1600                       TADI IARG
0216   3200                       DCA IARG
                                  .
                                  .
```

To pick up multiple arguments, the locations from X1 to X2+1 inclusive can be made into a subroutine.

CALLING THE OS/8 USR AND DEVICE HANDLERS

One important point to remember is that any code which calls the USR must not reside in locations 10000 to 11777. Therefore, any SABR routine which calls the USR must be loaded into a field other than field 1 or above location 2000 in field 1. To call the USR from SABR use the sequence:

```
CPAGE N             /N=7+(# OF ARGUMENTS)
6212                /CIF 10
JMS 7700            /OR 200 IF USR IN CORE
REQUEST
ARGUMENTS           /OPTIONAL DEPENDING ON REQUEST
ERROR RETURN        /OPTIONAL DEPENDING ON REQUEST
```

To call a device handler from SABR use the sequence:

```
        CPAGE 12        /10 IF "HAND" IN PAGE 0
        6202            /CIF 0
        JMS I HAND      /DO NOT USE JMSI
        FUNCT
        ADDR
        BLOCK
        ERROR RETURN
        SKP
HAND,   0               /"HAND" MUST BE ON SAME PAGE
                        /AS CALL, OR IN PAGE 0
```

Loading and Operating SABR

Procedures for loading SABR and assembling a source program are given below. See Appendix A for instructions on the use of the Binary Loader. Loading and operating instructions for OS/8 SABR are contained in Chapter 9 of *Introduction to Programming*.

1. Make sure the Binary Loader is in memory, in field n.
2. Set switches 6–8=n (Instruction field), and switches 9-11=0 (Data field).
3. Press EXTD ADDRess LOAD.
4. Set the Switch Register=7777.
5. Press ADDRess LOAD.
6. Insert the SABR binary tape into the reader.

7. If using the high-speed reader, depress Switch Register Bit 0.
8. Press CLEAR and CONTinue.
9. SABR will now be loaded into memory by the Binary Loader; portions of SABR will load into field 0 and field 1.

ASSEMBLY PROCEDURE

It is assumed that the programmer has written his program in SABR language and punched this source program on paper tape in ASCII 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 the user has specified that his listing is to be typed or punched. Pass 2 is the listing pass. The assembly is carried out as follows:

1. Set switches 6-8=0 (Instruction field), and switches 9-11=0 (Data field).
2. Press EXTD ADDRess LOAD.
3. Set the Switch Register=0200.
4. Press ADDRess LOAD, CLEAR, and CONTinue.
5. SABR now types an identification label and a sequence of two or three questions:

   ```
   PDP-8 SABR DEC-08-A2D2-16
   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.
6. 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 Teletype 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 and must begin again. If using the high-speed reader, the tape must be positioned in the reader before answering the last question.

7. 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.
8. 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.
9. At the end of Pass 1, SABR halts.
10. If an assembly listing is desired, reposition the beginning of the source tape in the reader, and if using the Teletype reader, set it to START, and then press CONTinue.
11. At the end of Pass 2, SABR again halts. To restart SABR for assembling another program, press CONTinue.
12. To restart SABR at any time, press HALT, set the Switch Register=0200, press ADDRess LOAD, CLEAR, and CONTinue. The first pass must always be repeated.

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. Methods for assembling FORTRAN source tapes with SABR are contained in Chapter 1.

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, and is even capable, in a special way, of loading programs over itself. It also has options which give storage maps and core availability.

The Linking Loader requires a PDP-8 series computer with at least 8K words of core memory. Either high-speed or Teletype paper tape input is acceptable; however, a high-speed reader is highly recommended.

The software requirements are:

1. Binary paper tape copy of the Linking Loader (DEC-08-A2C3-PB) (The Linking Loader is pre-built into the OS/8 Operating System.)
2. Relocatable binary paper tape copies of both Part 1 and Part 2 of the 8K System Library
3. 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 Teletype 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 program 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 0200-0777
Field 0, 1, 2, . . .	Locations 0007 and 0033-0073
	Locations 0007 and 0033-0124 if using the device independent I/O options or the CHAIN subroutine in OS/8 FORTRAN.

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
0125-0177	if using the device independent I/O option available in OS/8 FORTRAN.

*Location 10 is not available in OS/8

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 field of memory. Loading instructions for the OS/8 Linking Loader are contained in Chapter 9 of *Introduction to Programming.*

1. Make sure the Binary Loader is in memory, for example, in field m, and let h represent the number of the highest field in the user's configuration.
2. Set switches 6-8=m (Instruction field) and switches 9-11=h (Data field).
3. Press EXTD ADDRess LOAD.
4. Set the Switch Register=7777.
5. Press ADDRess LOAD.
6. Place the binary paper tape of the Linking Loader in the reader.
7. If using a high-speed reader, depress Switch Register Bit 0.
8. Press CLEAR and CONTinue.

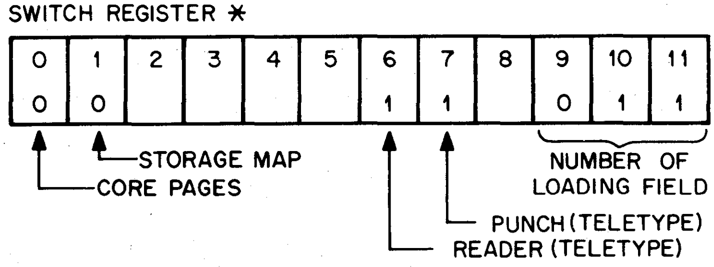
LOADING RELOCATABLE PROGRAMS

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, set the switches as follows: switches 6-8=h (Instruction field) and 9-11=h (Data field).
2. Press EXTD ADDRess LOAD.
3. Set the Switch Register=0200.
4. Press ADDRess LOAD.
5. Place the relocatable binary tape for the first program to be loaded in the reader. Position the tape with leader code in the reader.
6. Set Switch Register to 0000. Then, if loading via the Teletype reader is required, raise Switch Register bit 6. If the configuration does not include a high-speed punch, 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.

7. Press CLEAR and CONTinue.
8. The Linking Loader types out an identification label, and the user's relocatable binary program will be loaded:
   PDP-8 LINKING LOADER DEC-08-A2C3-07
   When loading is completed, the Linking Loader halts.
9. The user may now either load another program or select one of the options in steps 11 and 12.
10. 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.
11. To select the Core Availability option, set Switch Register bit 0 = 1, and press CONTinue.
12. To select the Storage Map option, set Switch Register bit 1 = 1, and press CONTinue.*
    If the Teletype punch is turned on for possible RIM format data punching, as explained earlier, ensure that it is turned off before selecting either of the options. Turn it on again after the typing of the options is completed.
13. The user may continue loading more programs as in step 10 after using either of the options.

Any time the Linking Loader halts, the user may access memory directly via the DEPosit and EXAMine console 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).

---

* All other Switch Register bits are irrelevant.

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, and 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 multiple 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. 1. A pseudo-op with improper arguments. 2. A quote mark with no argument. 3. A non-terminated text-string. 4. A memory reference instruction with improper address. 5. An illegal combination of micro-instructions.

E There is no END statement.

S This error message means either one of four things:

1. The symbol table has overflowed. This can be corrected by using fewer symbols, using shorter symbols, or by breaking the program into smaller parts.
2. Common storage has been exhausted.
3. More than 64 different user-defined symbols have occurred in a core page.
4. 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.

Codes flagged beside symbols in the symbol table are explained in the section concerning the symbol table.

In addition to the SABR error messages already described, OS/8 SABR contains the following:

D A device handler has returned a fatal error condition.

L /L or /G option was indicated, but the LOADER.SV file does not exist on the system device.

U No symbol table is being produced, but there is at least one undefined symbol in the program. The name of the first undefined symbol found appears in the error message.

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.

Table 2-2 Linking Loader Error Codes

Error Code	Explanation
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 on page 15-87 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.

The OS/8 Linking Loader includes these additional error codes:

Error Code	Explanation
0000	/I or /O specified too late. Refer to Chapter 9 of *Introduction to Programming.*
0006	An output error has occurred while reading a binary file.
0007	An input error has occurred (either a physical device error or an attempt to read from a write-only device).
0010	No starting address has been specified and there is no entry point named MAIN.
0011	An error occurred while the Loader was trying to load a device handler, or no /H was specified (see Chapter 9).

0012	I/O error on system device.

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 at any time by reinitializing the Linking Loader via the console switches. To do this, set switches 6-8=h (the field where the Linking Loader resides), switches 9-11=h, and press EXTD ADDRess LOAD. Then set the Switch Register=6200, and press ADDRess LOAD, CLEAR, and CONTinue.

LIBRARY PROGRAMS

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 relevant to subprograms called by the statement at the relevant location.

Table 2-3 Library Error Messages

Error Message	Explanation
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
FMT3	Illegal character in I, E, or F field
FMT4	Multiple minus signs
FMT5	Invalid FORMAT statement

Table 2-3 (Con't) Library Error Messages

Error Message	Explanation
FLPW	Negative number raised to floating power
FPNT	Floating-point error; may be caused by division by zero; floating-point overflow; attempt to fix too large a number.
SQRT	Attempt to take root of a negative number

OS/8 includes, in addition, the error message:

```
USER ERROR 1 AT 00537
```

which means that the user tried to reference an entry point of a program which was not loaded.

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.

Demonstration Program Using Library Routines

The following demonstration program is a SABR program showing the use of the library routines. The program was 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 using the Symbolic Editor, assembled with SABR, and loaded with the Linking Loader, under the OS/8 Operating System. (An example of a FORTRAN program compiled and then assembled with the paper tape SABR system is contained in the section, Demonstration Program, in Chapter 1.)

```
A        0257
B        0260
C        0261
D        0262
FLOAT    0000EXT
FORMT    0240
IOH      0000EXT
N        0256
OPEN     0000EXT
START    0200EXT
STO      0000EXT
WRITE    0000EXT


                                ENTRY START

0200  4033        START,   CALL 0,OPEN      /INITIALIZE
0201  0002 06
                                            /I/O DEVICES
0202  1257                 TAD A            /COMPUTE C=A+B
0203  1260                 TAD B
0204  3261                 DCA C
0205  4033                 CALL 1,FLOAT     /CONVERT TO
0206  0103 06
                                            /FLOATING POINT
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     /INITIALIZE
0216  0205 06
                                            /I/O HANDLER
0217  6201 05              ARG N            /DEVICE NUMBER
0220  0256 01
                                            /1=TELETYPE
0221  6201 05              ARG FORMT        /FORMAT SPECI-
0222  0240 01
                                            /FICATION
0223  4033                 CALL 1,IOH       /TYPE INTEGER
0224  0106 06
                                            /NUMBER
0225  6201 05              ARG C
0226  0261 01
0227  4033                 CALL 1,IOH       /TYPE FLOATING
0230  0106 06
                                            /POINT NUMBER
0231  6201 05              ARG D
0232  0262 01
0233  4033                 CALL 1,IOH       /COMPLETE THE I/O
0234  0106 06
0235  6211                 ARG 0
0236  0000
0237  7402                 HLT
```

```
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
                        END
```

The binary tape produced by the assembly was then run using OS/8 with the following results:

```
THE ANSWERS ARE    4   4.00
```

appendix a
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 0.
8. SING STEP is not set.
9. High-speed punch is OFF.
10. 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 series 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 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 (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 A-1.

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

Table A-1. RIM Loader Programs

	Instruction	
Location	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 A-2, which also shows how to correct an incorrectly stored instruction.

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

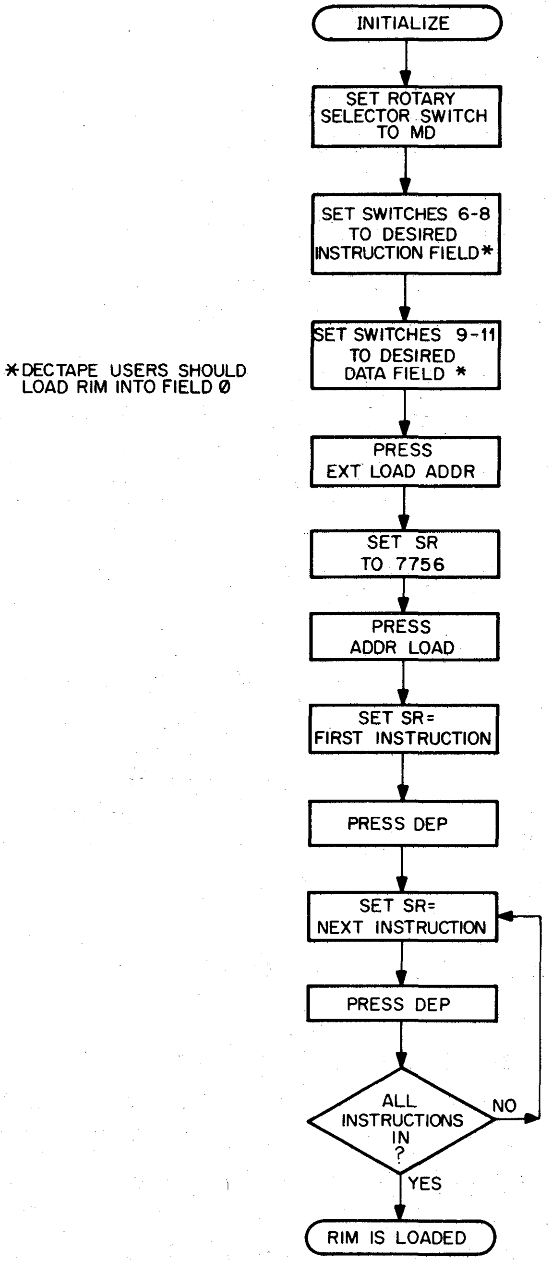


Figure A-1. Loading the RIM Loader

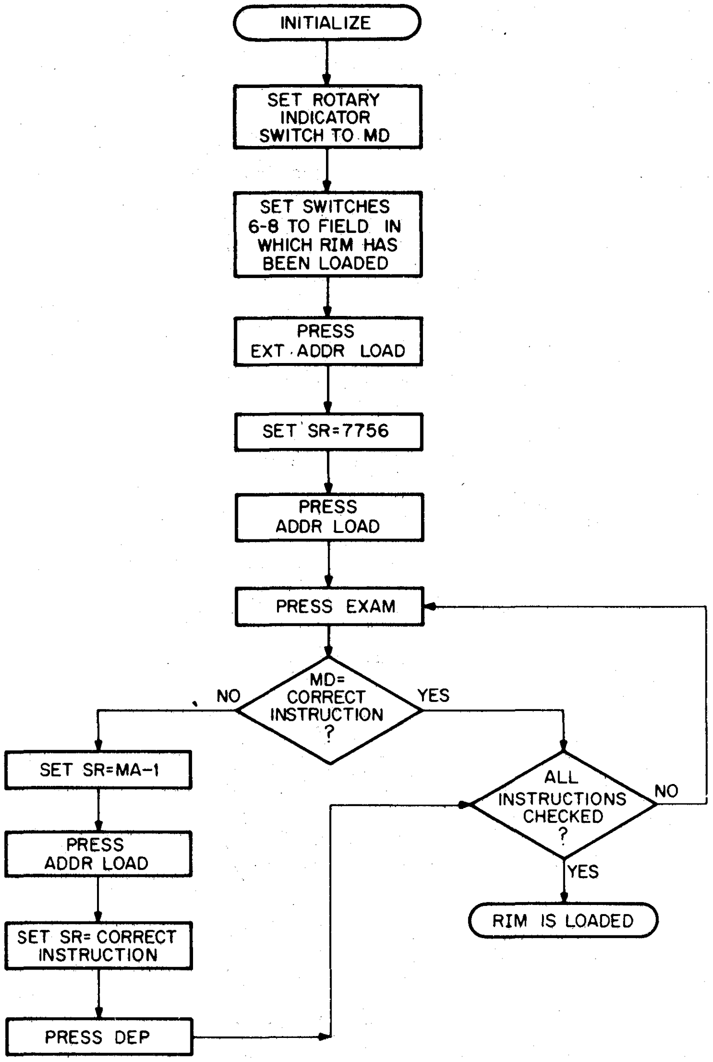


Figure A-2. Checking the RIM Loader

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 A-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.

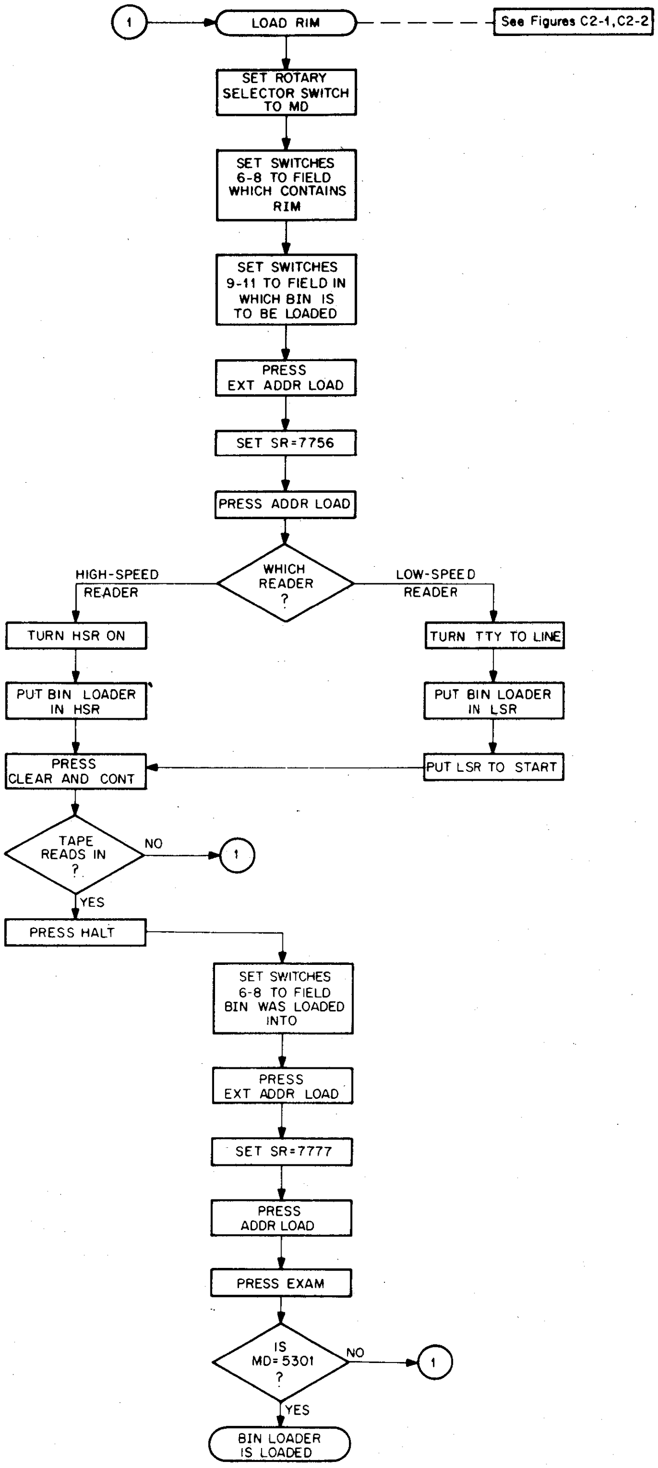


Figure A-3 Loading the BIN Loader

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). 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.

Binary tapes to be loaded should be started on the leader-trailer code (Code 200), otherwise zeros may be loaded into core, destroying previous instructions.

Figure A-4 lilustrates the procedure for loading binary tapes into core.

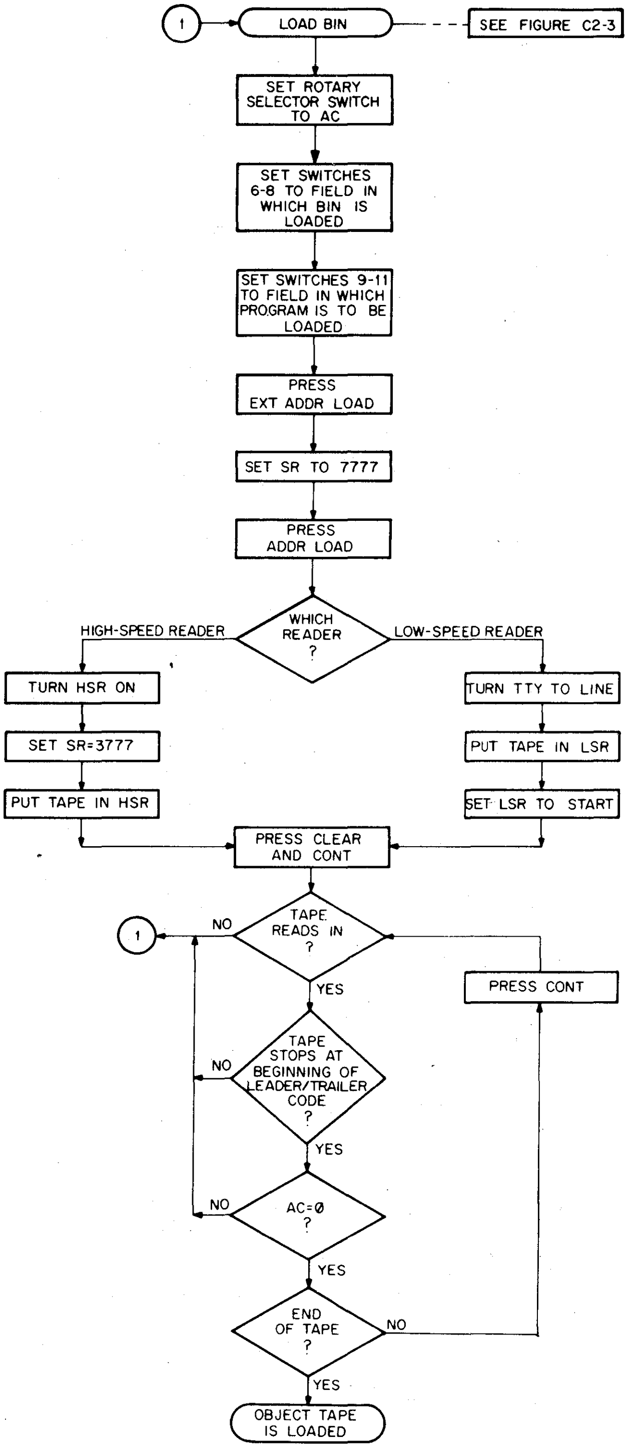


Figure A-4. Loading A Binary Tape Using BIN

appendix b
character codes

ASCII-1[1] Character Set

Character	8-Bit Octal	6-Bit Octal	Decimal Equivalent (A1 Format)	Character	8-Bit Octal	6-Bit Octal	Decimal Equivalent (A1 Format)
A	301	01	96	!	241	41	−1952
B	302	02	160	"	242	42	−1888
C	303	03	224	#	243	43	−1824
D	304	04	288	$	244	44	−1760
E	305	05	352	%	245	45	−1696
F	306	06	416	&	246	46	−1632
G	307	07	480	'	247	47	−1568
H	310	10	544	(	250	50	−1504
I	311	11	608	)	251	51	−1440
J	312	12	672	*	252	52	−1376
K	313	13	736	+	253	53	−1312
L	314	14	800	,	254	54	−1248
M	315	15	864	-	255	55	−1184
N	316	16	928	.	256	56	−1120
O	317	17	992	/	257	57	−1056
P	320	20	1056	:	272	72	−352
Q	321	21	1120	;	273	73	−288
R	322	22	1184	<	274	74	−224
S	323	23	1248	=	275	75	−160
T	324	24	1312	>	276	76	−96
U	325	25	1376	?	277	77	−32
V	326	26	1440	@	300		32
W	327	27	1504	[	333	33	1760
X	330	30	1568	\	334	34	1824
Y	331	31	1632	]	335	35	1888
Z	332	32	1696	↑(∧)[2]	336	36	1952
0	260	60	−992	←(—)[2]	337	37	2016
1	261	61	−928	Leader/Trailer	200		
2	262	62	−864	LINE FEED	212		
3	263	63	−800	Carriage RETURN	215		
4	264	64	−736	SPACE	240	40	−2016
5	265	65	−672	RUBOUT	377		
6	266	66	−608	Blank	000		
7	267	67	−544	BELL	207		
8	270	70	−480	TAB	211		
9	271	71	−416	FORM	214		

---

[1] **An abbreviation for American Standard Code for Information Interchange.**

[2] The character in parentheses is printed on some Teletypes.

appendix c

permanent symbol table

The following are the elements of the PDP-8 instruction set found in the SABR permanent symbol table. 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 *Small Computer Handbook,* available from the DEC Software Distribution Center.

INSTRUCTION CODES

Mnemonic	Code	Operation	Time (μsec.)[1]
Memory Reference Instructions			
AND	0000	Logical AND	2.6
TAD	1000	Two's complement add	2.6
ISZ	2000	Increment and skip if zero	2.6
INC	2000	Nonskip ISZ	2.6
DCA	3000	Deposit and clear AC	2.6
JMS	4000	Jump to subroutine	2.6
JMP	5000	Jump	1.2
			Sequence
Group 1 Operate Microinstructions (1 cycle[2])			
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	Complemented link	2
CMA	7040	Complement AC	2
CLL	7100	Clear link	1
CLA	7200	Clear AC	1

---

[1] Times are representative of the PDP-8/E.

[2] 1 cycle is equal to 1.2 microseconds.

Mnemonic	Code	Operation	Sequence
Group 2 Operate Microinstructions (1 cycle)			
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
Combined Operate Microinstructions			
CIA	7041	Complement and increment AC	2, 3
STL	7120	Sent link to 1	1, 2
STA	7240	Set AC to − 1	2
Internal IOT Microinstructions			
ION	6001	Turn interrupt processor on	
IOF	6002	Disable interrupt processor	
Keyboard/Reader (1 cycle)			
KSF	6031	Skip on keyboard/reader flag	
KRB	6036	Clear AC, read keyboard buffer (dynamic), clear keyboard flags	
Teleprinter/Punch (1 cycle)			
TSF	6041	Skip on teleprinter/punch flag	
TLS	6046	Load teleprinter/punch, print, and clear teleprinter/punch flag	
High Speed Reader—Type PR8/E (1 cycle)			
RSF	6011	Skip on reader flag	
RRB	6012	Read reader buffer and clear reader flag	
RFC	6014	Clear flag and buffer and fetch character	
High Speed Punch—Type PP8/E (1 cycle)			
PSF	6021	Skip on punch flag	
PLS	6026	Clear flag and buffer, load buffer and punch character	

PSEUDO-OPERATORS

The following is a list of the SABR assembler pseudo-operators.

ABSYM
ACH
ACM
ACL
ARG
BLOCK
CALL
COMMN
CPAGE
DECIM
DUMMY
EAP
END
ENTRY
FORTR
I
IF
LAP
OCTAL
OPDEF
PAGE
PAUSE
REORG
RETRN
SKPDF
TEXT

INDEX

HOW TO OBTAIN SOFTWARE INFORMATION

SOFTWARE NEWSLETTERS, MAILING LIST

The Software Communications Group, located at corporate headquarters in Maynard, publishes newsletters and Software Performance Summaries (SPS) for the various Digital products. Newsletters are published monthly, and contain announcements of new and revised software, programming notes, software problems and solutions, and documentation corrections. Software Performance Summaries are a collection of existing problems and solutions for a given software system, and are published periodically. For information on the distribution of these documents and how to get on the software newsletter mailing list, write to:

Software Communications
P. O. Box F
Maynard, Massachusetts 01754

SOFTWARE PROBLEMS

Questions or problems relating to Digital's software should be reported to a Software Support Specialist. A specialist is located in each Digital Sales Office in the United States. In Europe, software problem reporting centers are in the following cities.

Reading, England
Paris, France
The Hague, Holland
Tel Aviv, Israel
Milan, Italy
Solna, Sweden
Geneva, Switzerland
Munich, West Germany

Software Problem Report (SPR) forms are available from the specialists or from the Software Distribution Centers cited below.

PROGRAMS AND MANUALS

Software and manuals should be ordered by title and order number. In the United States, send orders to the nearest distribution center.

Digital Equipment Corporation
Software Distribution Center
146 Main Street
Maynard, Massachusetts 01754

Digital Equipment Corporation
Software Distribution Center
1400 Terra Bella
Mountain View, California 94043

Outside of the United States, orders should be directed to the nearest Digital Field Sales Office or representative.

USERS SOCIETY

DECUS, Digital Equipment Computer Users Society, maintains a user exchange center for user-written programs and technical application information. A catalog of existing programs is available. The society publishes a periodical, DECUSCOPE, and holds technical seminars in the United States, Canada, Europe, and Australia. For information on the society and membership application forms, write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

DECUS
Digital Equipment, S.A.
81 Route de l'Aire
1211 Geneva 26
Switzerland

READER'S COMMENTS

NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form (see the HOW TO OBTAIN SOFTWARE INFORMATION page).

Did you find errors in this manual? If so, specify by page.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

Please indicate the type of user/reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Non-programmer interested in computer concepts and capabilities

Name ______________________ Date ______________

Organization ______________________

Street ______________________

City ______________ State ______________ Zip Code or Country ______________

If you do not require a written reply, please check here. ☐

Fold Here

Do Not Tear - Fold Here and Staple

FIRST CLASS
PERMIT NO. 33
MAYNARD, MASS.

BUSINESS REPLY MAIL
NO POSTAGE STAMP NECESSARY IF MAILED IN THE UNITED STATES

Postage will be paid by:

digital

Software Communications
P. O. Box F
Maynard, Massachusetts 01754