Digital Equipment Corporation
Maynard, Massachusetts

PDP-8 Family
Programmer's Reference Manual

# 8K SABR ASSEMBLER

DEC-08-ARXB-D

# PDP-8
# 8K SABR ASSEMBLER
# PROGRAMMER'S REFERENCE MANUAL

For additional copies of this document, order No. DEC-08-ARXB-D from Program Library, Digital Equipment Corporation, Maynard, Mass. 01754 Price $2.00

DIGITAL EQUIPMENT CORPORATION □ MAYNARD, MASSACHUSETTS

1st Printing April 1969
2nd Printing (Rev) March 1970

Your attention is invited to the last two pages of this manual. The Reader's Comments page, when filled in and returned, is beneficial to both you and DEC. All comments received are considered when documenting subsequent manuals, and when assistance is required, a knowledgeable DEC representative will contact you. The Software Information page offers you a means of keeping up-to-date with DEC's software.

Copyright © 1969, 1970 by Digital Equipment Corporation

Documents Referenced (available from DEC's Program Library):

Introduction to Programming, C-18
8K FORTRAN Programmer's Reference Manual, DEC-08-KFXB-D
Paper Tape System User's Guide, DEC-08-NGCC-D
PDP-8/I Disk Monitor System, DEC-08-SDAB-D

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

| | |
|---|---|
| PDP | DEC |
| FLIP CHIP | FOCAL |
| DIGITAL | COMPUTER LAB |

# CONTENTS

Page

## CHAPTER 1
## THE SABR LANGUAGE



## CHAPTER 2
## PSEUDO-OPERATORS

CONTENTS (Cont)

CONTENTS (Cont)

Page

# PREFACE

This manual contains a detailed description of the 8K SABR Symbolic Assembly System. The SABR (Symbolic Assembler for Binary Relocatable programs) programming language is similar to that of PAL III with many additional features. It is an advanced one-pass assembler for use with a PDP-8/I, -8/L, -8, -8/S, or -5 computer with at least 8K (up to 32K) words of core memory and an ASR-33 Teletype; a high-speed photoelectric paper tape reader and punch is not required, although it is highly recommended.

DEC offers four symbolic assemblers for use on PDP-8 family computers, as follows:

a. PAL III Symbolic Assembler, the basic 4K assembly system. It is a two-pass assembler with an optional third pass which produces an octal/symbolic assembly program listing, and is highly recommended for the computer with 4K words of core memory. It is an excellent assembly language for the less experienced programmer yet powerful enough to satisfy the needs of the advanced programmer.

b. MACRO-8 Symbolic Assembler, essentially PAL III with the following additional features: user-defined macros, double precision integers, floating-point constants, arithmetic and Boolean operators, literals, text facilities, and automatic off-page linkage generation. It is recommended for the computer with 4K words of core memory when any of the additional features listed above are desired.

c. PAL-D Symbolic Assembler, essentially MACRO-8 excluding macros. It is used only in the PDP-8/I Disk Monitor System and requires 4K words of core memory.

d. 8K SABR Symbolic Assembler, primarily for experienced programmers to use with a computer that has 8K to 32K words of core memory. It differs from the preceding assemblers in its operating procedures, character set, pseudo-ops, execution of the assembled program, and especially in its assembled output (relocatable binary code).

It is assumed that the reader is familiar with assembly language programming. For an elementary approach to this type of programming, we recommend DEC's publication, Introduction To Programming (specify Order No. C-18), available from the Program Library (address on Title page).

# CHAPTER 1
# THE SABR LANGUAGE

## 1.1 INTRODUCTION

SABR (Symbolic Assembler for Binary Relocatable programs) is an advanced one-pass symbolic assembler. It translates symbolic programs written in the SABR language into binary relocatable code acceptable to the computer. SABR programs are core page independent. Therefore, programs may be written without regard to the 128-word core page of the computer. SABR automatically generates off-page and off-field references for direct or indirect statements. It also automatically connects instructions on one page to those that overflow onto the next. The list of available pseudo-ops is extensive, including external subroutine calling, argument passing, and conditional assembly. SABR offers an optional second pass to produce a side-by-side octal/symbolic listing of the assembled program.

The relocatable binary tapes produced by SABR are loaded into any field of core memory using the 8K Linking Loader, as are the library of subprograms. These subprograms may be called by any SABR program.

In addition to being a stand-alone symbolic assembler, SABR also acts as the second pass of the 8K FORTRAN compiler (see 8K FORTRAN Programmer's Reference Manual, DEC-08-KFXB-D).

SABR requires a PDP-8/I, -8/L, -8, -8/S, or -5* computer with at least 8K words of core memory and an ASR-33 Teletype. A high-speed photoelectric paper tape reader and punch is not necessary, although it is highly recommended.

The assembler system is furnished on four appropriately identified paper tapes. The SABR Assembler and 8K Linking Loader tapes are punched in binary coded format and are loaded into core memory using the Binary Loader (see PDP-8/I System User's Guide, DEC-08-NGCB-D). The two library of subprograms tapes are punched in relocatable binary coded format and are loaded into core memory using the 8K Linking Loader as explained in this manual.

With the exception of a few minor differences (and an entirely different list of pseudo-ops), the symbolic programming language for SABR is similar tothe PAL III language. However, the binary output from SABR is in relocatable binary code and is quite different from the PAL III binary output. The rest of this chapter describes the SABR language in full. For a more elementary approach to assembly language programming, we recommend DEC's new Introduction To Programming (specify Order No. C-18), available from the Program Library (address on Title page).

*The PDP-5 computer requires a PDP-8 extended memory control modification.

## 1.2 THE CHARACTER SET

a. Alphabetic:

Besides the normal alphabetic characters A, B, C, ..., X, Y, Z, the following characters are considered to be alphabetic by SABR:

| | |
|---|---|
| [ | left bracket, |
| ] | right bracket, |
| \ | back slash, |
| ↑ | up arrow. |

b. Numeric:

0, 1, 2, ..., 8, 9

c. Special:

| | | |
|---|---|---|
| , | Comma | delimits a symbolic address label |
| / | Slash | indicates start of a comment |
| ( | Left parenthesis | indicates a literal<br>(D indicates numeric literal is decimal;<br>(K indicates numeric literal is octal |
| " | Quote | precedes an ASCII constant |
| - | Minus sign | negates a constant |
| # | Number sign | increases value of preceding symbol by one |
| ↵ | RETURN<br>(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.

## 1.3 STATEMENTS

SABR symbolic programs are written as a sequence of statements, and are usually prepared on a teletype with the aid of the Symbolic Editor program. Each statement is written on a single line and is terminated by typing the RETURN key (carriage return/line feed sequence, abbreviated CR/LF).

SABR statements are virtually format free, because elements of a statement are not placed in numbered columns with rigidly controlled spacing between elements, as in punched-card oriented assemblers.

A statement line is composed of one or all of the following elements: label, operator, operand, comment, and/or format effectors. 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
```

SABR interprets and processes the statements, generating one or more machine (binary) instructions or data words during assembly.

An input line may be up to $72_{10}$ characters long, including spaces and tabs. Any characters beyond this limit are ignored.

### 1.3.1 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 is terminated by a comma.

Examples:

```
0200    0000    SAVE,   0
0201    1200    ABC,    TAD     SAVE
```

Where SAVE and ABC are the labels, the statements are in location 0200 and 0201, and generate the instructions 0000 and 1200.

### 1.3.2 Operators

An operator may be any one of the following items.

a. A mnemonic memory reference instruction followed by an operand.

b. A mnemonic memory reference instruction followed by an I followed by an operand. This creates an indirect memory reference instruction.

c. A single mnemonic microinstruction (operate or IOT instruction) or a string of such instructions separated by spaces or tabs. Combinations of microinstructions are formed by inclusive ORing the octal values of the instructions. Group 1 operate instructions can be combined with Group 1 instructions only, and Group 2 with Group 2 only, except for the CLA instruction which may be combined with either group. IOT instructions may not be combined with operate instructions. (Refer to Appendix B for a summary of all microinstructions.)

d. A pseudo-operator (Refer to Chapter 2, Pseudo-Operators).

Operators are terminated with a space or tab if an operand follows, otherwise they may be terminated with either a semicolon, slash, or carriage return.

Examples:

```
0200    1320        TAD       SAVE
0201    1550        TAD I     POINTR
0202    7004        RAL
0203    7620        SNL  SMA  CLA
                    :
                    PAGE
```

All SABR operators are listed in Appendix B.

### 1.3.3 Operands

Operands may occur in three ways:

a. Following a memory reference instruction, and separated from it by a space or tab; the operand is the address of the data to be accessed by the instruction. This address may be a user-defined address symbol or a numeric constant. If a symbol is used as the operand, it must be defined somewhere in the program. Constant addresses must be used with great care because the assembled program will be relocatable. If the memory reference instruction is indirect (followed by I) the operand is the address of the address of the data to be accessed. An operand following a direct memory reference instruction may also be a literal.

b. As the argument of a pseudo-operator.

c. On a line with no operator. In this case, the operand is called a parameter. A parameter may be a numeric constant, a literal, or a user-defined address symbol.

Examples:

```
0200    0200    ABC,        200;-320; "M
0201    7460
0202    0315
0203    0176    POINTR,     PG0ADR
                            REORG    1000
1000    1576    START,      TAD I    POINTR
1001    1375                TAD      (3
```

### 1.3.4 Comments

A programmer may add notes to a statement following a slash mark. Such comments do not affect assembly processing or program execution, but they are useful in the program listing for later analysis and debugging. Entire lines of comments may be present in the program.

NOTE

None of the special characters or symbols have significance when they appear in a comment.

Examples:

```
/THIS IS A COMMENT LINE.
/THIS TOO.  TAD;  CALL;  # "-2 (+ = !
A,    TAD      SAVE    /COMMENT
```

### 1.3.5 Format Effectors

Spaces and tabs are the formatting characters, usually used in the body of a symbolic program to provide a neat page. They can separate elements of a statement, as between an instruction and a comment. For example, the lines

```
GO, TAD TOTAL/MAIN LOOP
DCA I SAVE
TAD BUFPTR
SZA CLA/CHECK FOR END LOOP
JMP GO
```

are much easier to read when written as:

```
GO,   TAD       TOTAL      /MAIN LOOP
      DCA I     SAVE
      TAD       BUFPTR
      SZA CLA              /CHECK FOR END LOOP
      JMP       GO
```

The RETURN key (CR/LF) is both a statement and a line terminator. The semicolon may be used to terminate an instruction without terminating a statement line. This allows the programmer to place several lines of coding on a single line. If, for example, he 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

```
...
RTR
RTR
RTR
...
```

But, with the semicolon, the programmer may place all three RTR's on a single line, separating each RTR with a semicolon and terminating the line with the RETURN key. The above sequence of instructions could then be written

```
RTR;  RTR;  RTR  (terminated with the RETURN key)
```

This format is particularly useful when creating a list of data.

Example:

```
0200   0020      LIST,      20;  50;  -30;  62
0201   0050
0202   7750
0203   0062
```

Null lines may also be used as format effectors. A null line is a line containing only a carriage return, and possibly spaces or tabs. Such lines appear in the listing simply as blank lines.

## 1.4 SYMBOLS

Symbols are composed of legal alphanumeric characters. There are two major types of symbols, permanent symbols and user-defined symbols, and there are variations within each major type. A symbol is delimited by a nonalphanumeric character.

### 1.4.1 Permanent Symbols

Permanent symbols are predefined and maintained in SABR's permanent symbol table. They include all of the basic instructions and pseudo-ops listed in Appendix B. These symbols may be used without prior definition by the user. The OPDEF and SKPDF pseudo-operators are used to define instruction operators not included in the permanent symbol table.

### 1.4.2 User-Defined Symbols

A user-defined symbol is a string of from one to six legal alphanumeric characters delimited by a nonalphanumeric character. User-defined symbols are composed according to the following rules.

a. The characters must be legal alphanumerics, which are:

ABCD...XYZ[\]↑ and 0123456789.

b. The first character must be alphabetic.

c. The symbol should not contain more than six characters. Only the first six characters of any symbol are meaningful, the remainder, if any, are ignored. Therefore, a symbol such as INTEGER would be interpreted as INTEGE since the seventh character is ignored, and because the two symbols GEORGE1 and GEORGE2 differ only in the seventh character, they would be treated as the same symbol, GEORGE.

d. A user-defined symbol cannot be the same as any of the predefined permanent symbols, and,

e. A user-defined symbol must be defined only once. Subsequent definitions of the same symbol will be ignored and cause SABR to type the error message M (Multiple Definition).

A symbol is defined by appearing as a symbolic address label (Refer to Section 1.3.1) or by appearing in an ABSYM, COMMN, OPDEF or SKPDF statement (Refer to Chapter 2, Pseudo-Operators). No more than 64 different user-defined symbols may occur on any one core page.

### 1.4.3 Equivalent Symbols

When an address label appears alone on a line, i.e., with no instruction or parameter, the label is assigned the value of the next address assembled.

For example,

```
TAG1,
TAG2,     30
TAG3,
```

TAG1 and TAG2 are equivalent 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.

### 1.4.4 Incrementing Operands

Because SABR is a one-pass assembler and also sometimes generates more than one machine instruction for a single user instruction, operand arithmetic is impossible; i.e., statements of the form

```
TAD     TAG + 3
TAD     LIST1 - LIST2
JMP     . +6
```

are illegal.

However, in one special case such references are possible. By appending a number sign (#) to an address operand, the user will reference a location exactly one (1) greater than the location of the address operand. Thus TAD LOC# is equivalent to the PAL language statement TAD LOC+1.

Example:

```
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#
```

NOTE

In assembling # - references, SABR does not attempt to determine if multiple machine code words are generated at the symbolic address referenced.

Example:

```
START,  TAD I   LOC     /LOC IS OFF-PAGE
        NOP             /USER HOPES TO MODIFY
        :
        TAD     (7500   /SMA
        DCA     START#
```

The user hopes to change the NOP instruction to an SMA. However, this is not possible because the 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 execute a program with # - references.

In the previous example, the proper sequence is:

```
START,  TAD I   LOC
VAR,    NOP
        :
        TAD     (7500
        DCA     VAR
```

The # - sign feature is intended primarily for use in manipulating DUMMY variables, in picking up subroutine arguments in external subroutines, and returning from external subroutines. Refer to Section 2.4.4 for a full explanation of how this is done.

### 1.4.5 The Symbol Table Listing

Symbols are listed in alphabetic order at the end of the assembly pass (Pass 1) with their relative addresses beside them.

The following flags are added to special types of symbols.

| | |
|---|---|
| ABS | The address is absolute. |
| COM | The address is in COMMON. |
| OP | The symbol is an operator. |
| EXT | The symbol is an external and may or may not be defined. If not defined, there is no difficulty; it is 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.<br>A location is reserved for the instruction containing the undefined symbol, but nothing is placed in it. |

## 1.5 CONSTANTS

There are two types of constants: numeric and ASCII. These are discussed individually below. ASCII constants are used only as parameters. Numeric constants may be used as parameters or as operand addresses.

Example:

```
0200   1412        TAD I      12
```

Constant operand addresses are treated as absolute addresses, just as a symbol defined by an ABSYM statement. References to them are not generally relocatable. Therefore, they should be used only with great care. The primary use of constant operand addresses is to reference locations in page 0. (See Appendix D for a list of free locations in 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 or subtracted to/from each other or to/from symbols.

### 1.5.1 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-op. Octal mode is assumed at the beginning of assembly. The digits 8 and 9 must not appear in an octal string.

Examples:

```
0200  5020    A,    5020
0201  7575          -203
                    DECIM
0202  0120          80
```

### 1.5.2 ASCII Constants

Eight-bit ASCII values may be created as constants by typing the ASCII character immediately following a double quotation mark ("). A minus may be used to negate an alpha constant. The minus sign must precede the quotation mark.

Examples:

```
0200  0273    A,    ";
0201  7477    -"A   / -301
0202  0207    "     / BELL FOLLOWS "
```

The following characters are illegal as alpha constants: carriage return, line feed, form feed, rubout.

## 1.6 LITERALS

The use of literals is a special and convenient way of generating constant data in a program. Literals are normally used by TAD and AND instructions, as in the following examples:

```
0200    0376      A,        AND       (777
0201    1375                TAD       (-50
0202    1374                TAD       ("C
   .
   .
0374  . 0303
0375    7730
0376    0777
```

A literal is always a numeric or ASCII constant and must be preceded by a left parenthesis. The value of the literal will be assembled in a table near the end of the core page on which the instruction referencing it is assembled. The instruction itself will be assembled as an appropriate reference to the location where the numeric value of the literal is assembled. Literals may not be referenced indirectly.

The current numeric conversion mode can be changed on a purely local basis for a literal by inserting a D for decimal or a K for octal between the left parenthesis and the constant.

Examples:

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

This usage does not alter the prevailing permanent conversion mode.

A literal may also be used as a parameter (i.e., with no operator). In such a 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.

Example:

```
0200    0376 01   A,      (20
  .
  .
0376    0020
```

This feature is intended primarily for use in passing external subroutine arguments with the ARG pseudo-op (see Section 2.4.1).

# CHAPTER 2

# PSEUDO-OPERATORS

## 2.1 ASSEMBLY CONTROL

END — Every program or subprogram to be assembled must contain the END pseudo-op as its last line. If this requirement is not met, an error message (E) is given.

PAUSE — The PAUSE pseudo-op causes assembly to halt. It is designed to allow the user to break up large source tapes into several smaller ones. To do this, the user need only place a PAUSE statement at the end of each section of this source except the last. Then when assembly halts at a PAUSE, he may remove the source tape just read from the reader and insert the next one. Assembly may then be continued by pressing the console CONTinue switch.

WARNING

The PAUSE pseudo-op is designed specifically for use at the end of partial tapes and should not be used otherwise.

The reason for this is that the reader routine may have read data from the paper tape into its buffer that is actually beyond the PAUSE statement. Consequently, when CONTinue is pressed after the PAUSE is found by the line interpreting routine, the entire content of the reader buffer following the PAUSE is destroyed, and the next tape begins reading into a fresh buffer. Thus, if there is any meaningful data on the tape beyond the PAUSE statement, it will be lost.

DECIM — Initially the numeric conversion mode is set for octal conversion. However, if the user wishes, he may change it to decimal by use of the DECIM pseudo-op.

OCTAL — If the numeric conversion mode has been set to decimal, it may be changed back to octal by use of the OCTAL pseudo-op.

No matter which conversion mode has been permanently set, it may always be changed locally for literals by use of the (D or (K syntax described earlier.

Examples:

```
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.

Examples:

| | | | | |
|---|---|---|---|---|
| 0200 | 7200 | START, | CLA | |
| | | | PAGE | |
| 0400 | 7040 | | CMA | |
| | | | REORG | 1000 |
| 1000 | 7041 | | CIA | |

CPAGE The CPAGE pseudo-op followed by a numerical argument N specifies that the following N words of code* must be kept together in a single unit and not be split up by page escapes and literal tables. If the N words of code will not fit on the current page of code, the current page is assembled as if a PAGE pseudo-op had been encountered. The N words of code will then be assembled as a unit on the next core page.

NOTE

N must be less than or equal 200 (octal) in nonautomatic paging mode or less than or equal 176 octal in automatic paging mode.

*Normally data. However, if these N words are instructions (for example, a JMS with arguments), it is the user's responsibility to count extra machine instructions which must be inserted by SABR.

Example:

```
     START,   CLA
              LAP                          /INHIBIT PAGE ESCAPE
              CPAGE 200                    /CLOSES THE
              NAME 1                       /CURRENT PAGE
              NAME 2                       /& ASSEMBLES THE
                                           /NAMES ON THE
                                           /NEXT PAGE.
```

IF The conditional pseudo-op, IF, is used with the following syntax:

```
     IF       NAME, 7
```

The action of the pseudo-op, so given, is to first determine whether the symbol NAME has been previously defined. If NAME is defined, the pseudo-op has no effect. If NAME is not defined, the next seven symbolic instructions (not counting null lines and comment lines) will be treated as comments and not assembled.

Example:

```
     /ABSYM   NAME     176
     IF NAME,  2                           /THE NEXT LINE TO BE
              CLL  RTL                     /ASSEMBLED WILL BE
              RAL                          /"DCA LOC".
     /IF THE SLASH BEFORE "ABSYM NAME 176" IS
     /REMOVED, THE "CLL RTL" AND "RAL" WILL
     /BE ASSEMBLED.
              DCA      LOC
```

Normally the symbol referenced by an IF statement should be either an undefined symbol or a symbol defined by an ABSYM statement. If this is done, the situation mentioned below cannot occur.

WARNING

In a situation such as the following, a special restriction applies.

```
     /EXAMPLE:
              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.

## 2.2 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 in Appendix D.

OPDEF
SKPDF — Operation codes not already included in the symbol table may be defined by use of the OPDEF or SKPDF pseudo-ops. Non-skip instructions must be defined with the OPDEF pseudo-op and skip-type instructions must be defined with the SKPDF pseudo-op.

Examples of ABSYM, OPDEF and SKPDF syntax:

```
ABSYM    TEM      177          /PAGE ZERO ADDRESSES
ABSYM    AX       10
OPDEF    DTRA     6761         /A NON-SKIP INSTR.
SKPDF    DTSF     6771         /SKIP-TYPE INSTRUCTIONS
SKPDF    SMZ      7540
```

NOTE

ABSYM, OPDEF and SKPDF definitions must be made before they are used in the program.

COMMN — The COMMN pseudo-op is used to name locations in field 1 as externals so that they may be referenced by any program. If any COMMN statements are used, they must occur at the beginning of the source, before everything else including the ENTRY statement. COMMON storage is always in field 1 and is allocated from location 0200 upwards. Since the top page of field 1 is reserved, no more than $3840_{10}$ words of COMMON storage may be defined.

A COMMN statement normally takes a symbolic address label, since storage is being allocated. However, COMMON storage may be allocated without an address label.

A COMMN statement always takes a numerical argument which specifies how many words of COMMON storage to be allocated; however, a 0 argument is allowed. A COMMN statement with 0 argument allocates no COMMON storage; it merely defines the given location symbol at the next free COMMON location.

The syntax of the COMMN statement is shown below.

Example:

```
A,      COMMN     20
B,      COMMN     10
        COMMN     300
C,      COMMN     0
D,      COMMN     10
        ENTRY     SUBRUT
```

In this example 20 words of COMMON storage are allocated from 0200 to 0217, and A is defined at location 0200. Then, 10 words are allocated from 0220 to 0227, and B is defined at 0220. Notice that if A is actually a 30 word array, this example equates B(1) with A(21).

The example continues by allocating COMMON storage from 0230 to 0527 with no name being assigned to this block. Then 10 words are allocated from 0530 to 0537 with both C and D being defined at 0530.

## 2.3 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
/WITHIN A SUBPROGRAM AREA MAY
/BE ACHIEVED:

    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.)

Examples:

```
LIST,     BLOCK 3          /ASSEMBLES AS
                           /3 ZEROS WITH
                           /"LIST" DEFINED
                           /AT THE 1ST LOCATION
NAME1,    BLOCK            /DEFINES NAME1 =
NAME2,    BLOCK 0          /NAME2 = NAME3 =
NAME3,                     /NAME4
NAME4,    BLOCK 2
```

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 six-bit for the letter I. In general, characters outside the ASCII range of 240-337 should not be used.

Example:

```
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
```

## 2.4 EXTERNAL SUBROUTINE

SABR and the Linking Loader possess extensive capabilities for calling external subprograms and for passing arguments between them. In addition to the facilities mentioned in this section, COMMON storage is also available (refer to Section 2.2).

For example, a user wishes to write a long main program, MAIN, which uses two major subroutines, S1 and S2. S1 requires two arguments and S2 requires one argument. The user would then write MAIN, S1 and S2 as three separate programs in the following fashion:

```
MAIN,    CLA              /START OF MAIN
         .
         .
         END

ENTRY    S1
S1,      BLOCK 2
         .
         .
         RETRN  S1
         END
ENTRY    S2
S2,      BLOCK 2
         .
         .
         RETRN  S2
         END
```

He would then assemble each of these subprograms with SABR and load all of them with the Linking Loader.

MAIN would contain statements in the form

```
CALL     2, S1
ARG      X
ARG      Y
CALL     1, S2
ARG      Z
```

Also S1 could contain CALLs to S2 or S2 CALLs to S1.

In addition, any of the subprograms could make use of DUMMY variables.

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 Section 3.3.4), 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.

The particular pseudo-operators required to make use of these facilities are described next.

### 2.4.1 The CALL and ARG Statements

The CALL and ARG statements are the usual means of calling an external subroutine. For example, a subroutine named SUBR with two arguments can be called by another program with the instruction sequence:

```
TAG,     CALL     2, SUBR
N1,      ARG      (50
N2,      ARG      LOCATN
ETC,     ...
```

A CALL statement must contain both the number of arguments and the ENTRY point of the subprogram being called in that order and separated by a comma. Arguments may or may not have address labels. Constant arguments may be specified as literals for the reason explained below. However, true constant arguments may also be specified.

The above instructions are assembled as follows:

```
         CPAGE 6               /Make sure the following
                               /2N + 2 words will
                               /fit on the current core page.
TAG,     JMS LINK              /Call the CALL Linkage Routine,
         020X (06)             /where 2 = the number of
                               /arguments and X =
                               /the local number of the
                               /subprogram being called
                               /viz., SUBR.
N1,      CDF CUR (05)          /Field address of argument
                               /in form of a CDF instruction.
         POINTER               /Address in the literal
                               /table where the 50 is
                               /assembled.
N2,      CDF CUR or CDF 10     /Field of the argument
                               /depending on whether it is
                               /or is not in COMMON.
         LOCATN                /Address of argument.
```

When a subprogram is referenced in a CALL statement, the Run-Time Linkage Routine, LINK, always executes the transfer to the subprogram as follows.

First, it assumes that the ENTRY point to the subprogram is a two-word block. Into the first word of this block it places the number of the field where the CALL to the subprogram occurred. In the second word, it places the address where the CALL occurred, plus 2. In the example above SUBR would receive a 62M1 where TAG is in field M, and SUBR# would receive the address of N1. If there were no arguments, SUBR# would receive the address of ETC. Thus, the two-word block at the ENTRY point serves as storage for the 15-bit address vector for picking up arguments and also for returning from the subprogram.

Execution of the subprogram begins at the first location following the two-word ENTRY block.

The number of arguments in a CALL sequence must be less than $64_{10}$. The ARG statement may be used only in conjunction with a CALL statement.

When the ARG pseudo-op is used with a literal, as in the above examples, the actual literal (50 in this case) will be generated in the literal table, and in the location following the CDF CUR, there will be generated a relocatable pointer to the literal. This is the same as using a literal as a parameter.

If the ARG statement is used with a true constant argument, the constant itself is assembled in the location following the CDF instruction. In this case, the CDF is useless and is always just a meaningless space filler.

The advantage of using the ARG - literal method is that it allows a subroutine to pick up an argument which is sometimes a variable and sometimes a constant.

### 2.4.2 The ENTRY and DUMMY Statements

ENTRY — The ENTRY pseudo-op is used at the beginning of a subprogram to name its entry point, and define this symbol as an external for the Linking Loader.

The ENTRY statement must occur before the symbolic name of the entry point appears as a symbolic address label. The actual entry location must be a two-word reserved space so that both the return address and field can be saved when the routine is called.

Example:

```
        ENTRY       SUBROU
        SUBROU,     BLOCK 2
                    CLA
```

For convenience of picking up subprogram arguments following a CALL statement, an ENTRY acquires all the properties of a DUMMY variable.

DUMMY — A DUMMY variable is a special type of variable in the FORTRAN/SABR system. It must be so defined in the subprogram which references it. When referenced directly a DUMMY variable is treated the same as any other local symbol. However, when referenced indirectly it causes a call to the DUMMY Variable Run-Time Linkage Routine. This Linkage Routine assumes that the DUMMY variable is a two-word vector such that the first word is a 62N1 (where N = the field of the address to be referenced) and the second word contains the actual 12-bit address to be referenced.

DUMMY variables are used for passing arguments to and from subroutines. (See Section 2.4.4.)

Example:

```
ENTRY        AI
DUMMY        X
DUMMY        Y
AI,          BLOCK 2
 .
 .
 .
X,           M
Y,           N
```

### 2.4.3 The RETRN Statement

The RETRN statement is used to return from a subprogram to the calling program. The name of the subprogram being returned from must be specified so that the Return Linkage Routine can determine the action required, and because a subprogram may have differently named entry points. It is possible for the careful user to return to the location following the last call of any subprogram merely by specifying it in a RETRN statement.

Example:

```
TAG,      RETRN      SUBROU
```

Before the RETRN statement is used, the user must be sure to increment the pointer in the second word of the subprogram entry to the proper point beyond all the arguments following the CALL statement. An example of how this is done is given below.

### 2.4.4 Picking up Subprogram Arguments

An advanced technique for picking up subprogram arguments is provided because:

a. Subprogram arguments are two-word addresses and a subprogram CALL is executed by the Run-Time Linkage Routine.

b. The calling program and subprogram may reside in different fields.

A subprogram entry point is assumed to have been defined as a two-word reserved block and defined as an ENTRY. The appearance of the subprogram name in an ENTRY statement gives the two-word block the properties of a DUMMY variable. This means that when the subprogram name is referenced indirectly this generates a call to the DUMMY Variable Run-Time Linkage Routine where the details of locating and picking up the argument address words are worked out. Thus, the user, need only use the number sign feature to increment the argument pointer in the second word of the entry point.

The following example shows how SUBR would pick up the arguments 50 and LOCATN in the example and deposit them in LOC1 and LOC2.

Example:

```
        /MAIN       PROGRAM
        MAIN,       CLA
                    :
        TAG,        CALL         2,SUBR
        N1,         ARG          (50
        N2,         ARG          LOCATN
        ETC,        ...
                    END

        /SUBROUTINE
              ENTRY    SUBR
              DUMMY    TEM
SUBR,         BLOCK    2

              TAD I    SUBR         /THIS GIVES YOU THE FIELD ADDRESS
                                    /(CDF CUR) OF THE (50
                                    /I.E., THE CONTENTS OF N1
              DCA      TEM          /TO FIRST WORD OF DUMMY
              INC      SUBR#        /MOVE ARG PTR TO N1#
              TAD I    SUBR         /GET ADDRESS OF (50
                                    /I.E., CONTENTS OF N1#
              DCA      TEM#         /TO 2ND WORD OF DUMMY
              TAD I    TEM          /PICK UP THE 50
              DCA      LOC1
              INC      SUBR#        /MOVE ARG PTR TO N2
               :
              Similar method to pick up contents of LOCATN.
               :
              INC      SUBR#        /MOVE PTR FOR RETURN AT ETC
              RETRN    SUBR
TEM,          BLOCK    2
```

Constant arguments are specified as literals because the subprogram may not know that a constant argument is being used. Hence, specifying constant arguments as literals will ensure that the second word of every assembled argument is actually the address of the argument.

The ARG statement may be used with a constant (e.g., if a constant address is intended).

The following technique may be used if SUBR can assume that the first argument is always a constant:

Example:

```
    /MAIN PROGRAM
    TAG,        CALL                2, SUBR
    N1,         ARG                 50
    N2,         ARG                 LOCATN
    ETC,
                 .
                 .
                 .
                END

    /SUBROUTINE

            ENTRY       SUBR
            DUMMY       TEM
SUBR,       BLOCK       2
            INC         SUBR#       /MOVE ARG PTR TO N1#
            TAD I       SUBR        /THIS GETS THE 50
                                    /IMMEDIATELY
            DCA         LOC1
            INC         SUBR#       /GET C(LOCATN) IN
                                    /THE USUAL WAY
```

# CHAPTER 3

# THE ASSEMBLED BINARY CODE

Because SABR is not a one-for-one assembler, it is necessary to give a general description of the type of code which it produces. The ordinary user needs only a general understanding of this topic; a more detailed discussion is included for the advanced user.

## 3.1 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 twelve bits contain the data word itself.

| | | |
|---|---|---|
| Relocation Code | High Order of Data Word | first frame |
| Low Order of Data Word | | second frame |

The assembled binary tape is preceded and followed by leader/trailer code 200. The check-sum is contained in the last two frames of tape before the trailer code. It appears as a normal 16-bit word as shown below.

| | | |
|---|---|---|
| 1 0 0 0 | High Order of Checksum | first frame |
| Low Order of Checksum | | second frame |

All assembled programs have a relative origin of 0200.

## 3.2 THE LOADER RELOCATION CODES

The four-bit relocation codes issued by SABR for use by the Linking Loader are all explained below. The codes are given in octal.

00 Absolute — Load the data word at the current loading address. No change is required.

Example:

| | | | |
|---|---|---|---|
| 0205 | 5277 | JMP LOC where LOC is at 0277 (on page) | |
| 0242 | 7500 | SMA | |
| 0356 | 0020 | 20 | (a constant) |

| | | |
|---|---|---|
| 01 | Simple Relocation | Add the relocation constant to the word before loading it. (The relocation constant is 200 less than the actual address where the first word of the program is loaded.) Items with this code are always program addresses. |

Example:

```
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 ASCII code for the symbol.

Example:

| 03 | address |
|---|---|
| address low order | |
| L | |
| O | |
| C | |
| 2 | |
| space | |
| space | |

| | | |
|---|---|---|
| 04 | Reorgin* | 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. |

Example:

```
0300    6201  05   A,    TAD  LOC2
0301    1776             where LOC2 is off page so that
  :                      the TAD instruction 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. They are generated automatically by SABR when a direct reference instruction must be assembled as an indirect, and there is the possibility that the current data field setting is different from the field where the indirect reference occurs.

*Does not appear in assembly listings.

| | | |
|---|---|---|
| 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, Section 2.4) 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.

Example:

```
0200    4033          CALL   3, SUB
0201    0307   06
                      ARG  X
                      ARG  Y
                      ARG  Z
```

Here, SUB has been assigned the local number 07 during assembly. At loading time this number will be changed to the global number (for example, 23), which is assigned to SUB. In this example, 0323 would actually be loaded at relative address 0201.

| | | |
|---|---|---|
| 10 | Leader/Trailer* and Checksum | This code represents normal leader/trailer. At the first occurrence of this code following the assembled program, the computer word contains the checksum. |
| 12 | High Common* | The data word is the highest location in Field 1 assigned to COMMON storage by the program. This item will occur exactly once in every binary tape and it must be the first word after the leader. If no COMMON storage has been allocated in the program, the data 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. |

## 3.3 PAGE ASSEMBLY

SABR assembles page-by-page rather than one instruction at a time. This is accomplished by building various tables as instructions are read. When a full page of instructions has been collected

*Does not appear in assembly listings.

(counting literals, off-page pointers and multiple word instructions) the page is assembled and punched. Several pseudo-ops also cause a page to be assembled.

### 3.3.1 Page Format

A normal assembled page of code has a format as shown below.

| | |
|---|---|
| X000 | Assembled Instructions |
| | Page Escape |
| | Literals and Off-Page Pointers |
| X377 | Page Escape |

Literals and off-page pointers are intermingled in the table at the end of the page.

### 3.3.2 Page Escapes

Under normal circumstances SABR is in Automatic Paging Mode. This mode causes SABR to connect each assembled core page of code to the next page 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. Also, a pseudo-op is available to turn off this Automatic Paging Mode.

There are two types of Page Escapes, depending on whether or not the last instruction is a skip instruction. If the last instruction was not a skip instruction, 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 was a skip instruction, 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
```

### 3.3.3 Multiple Word Instructions

Certain instructions in the source program require SABR to assemble more than one instruction (e.g., off-page indirect references and indirect references where a data field re-setting may be required). In the listing, the source instruction will appear beside the first of the assembled binary words.

A difficulty arises when a multiple word instruction follows a skip instruction. In such a case, extra instructions must be assembled to enable the skip to be effected exactly as desired by the programmer.

### 3.3.4 Run-Time Linkage Routines

The routines described in this section are entirely automatic. The user needs to know nothing about them except to better understand the program assembly listing.

Many of the multiple word instructions involve use of special linking routines called the Run-Time Linking Routines. These routines make up a special portion of the 8K FORTRAN/SABR System. They are used at execution time by all user programs to carry out the linkage for calls to external subroutines and for all the various forms of off-field and off-page indirect memory references.

Since the Linkage Routines are needed by all user programs, their use is entirely automatic. The user need not consider them either at programming time or at loading time. SABR determines when calls to the Linkage Routines are required in the user's program and automatically generates such calls. The Linking Loader always automatically loads the Linkage Routines.

The residence of the Linkage Routines is described in Appendix D.

There are seven Linkage Routines:

| | | |
|---|---|---|
| a. | change Data Field to current and skip | CDFSKP |
| b. | change Data Field to one (COMMON) and skip | CDZSKP |
| c. | off page indirect reference linkage | OPISUB |
| d. | off bank (COMMON) indirect reference linkage | OBISUB |
| e. | DUMMY variable indirect reference linkage | DUMSUB |
| f. | subroutine CALL linkage | LINK |
| g. | subroutine RETURN linkage | RTN |

The following is a description of the individual Linkage Routines.

a. CDFSKP is called when a direct off-page memory reference, requiring that the data field be reset to the current field, follows a skip-type instruction.

Example:

| 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. |

b. CDZSKP is called when a direct memory reference is made to a location in COMMON (which is always in Field 1), the action of CDZSKP is the same as that of CDFSKP except that it always executes a CDF 10 instead of a CDF current.

Example:

| 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. |

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

Example:

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

d. OBISUB is called when there is an indirect reference to a location in COMMON. In such a case it is assumed that the location in COMMON which is being indirectly referenced points to some location that is also in COMMON.

Example:

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

e. DUMSUB is called when there is an indirect reference to a DUMMY variable. In such a case, DUMSUB assumes that the DUMMY variable is a two-word vector in which the first word is a 62N1, where N = the field of the address to be referenced, and the second word is the actual address to be referenced.

Example:

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

f. LINK is called to execute the linkage required by a CALL statement in the user's program. When a CALL statement is used, it is assumed that the entry point of the subprogram is named in the CALL and that this entry point is a two-word, free block followed by the executable code of the subprogram. LINK leaves the return address for the CALL in these two words in the same format as a DUMMY variable.

Example:

| 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 |

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

Example:

| Program | Assembled Code | Meaning |
|---|---|---|
| RETRN SUBR | 4040 | call RTN |
| | 0005 06 | number of the subprogram being returned from (SUBR) |

### 3.3.5 Skip Instructions

In page escapes and in multiple word instructions, skip-type instructions must be distinguished from non-skipping instructions. For this reason, a special pseudo-op, SKPDF, must be used to define skip instructions not in the permanent symbol table.

This also explains why both ISZ and INC are included in the permanent symbol table. ISZ is considered to be a skip instruction and INC is not. INC should be used to conserve space when the programmer desires only to increment a memory word with no possibility of a skip resulting.

Example 1 shows the code which is assembled for an indirect reference instruction to an off page location following an INC instruction and Example 2 shows the same instruction following an ISZ instruction. In Example 1, it is assumed there is no possibility of the INC instruction actually causing a skip.

Example 1

| | | | | | | |
|---|---|---|---|---|---|---|
| INC | | POINTR | 0220 | 2376 | | |
| TAD | I | LOC2 | 0221 | 4062 | | off page indirect execution |
| | | | 0222 | 0520 | 01 | |
| | | | 0223 | 1407 | | |

Example 2

| | | | | | |
|---|---|---|---|---|---|
| ISZ | COUNTR | 0220 | 2376 | | |
| TAD I | LOC2 | 0221 | 7410 | | skip to execution |
| | | 0222 | 5226 | | jump over execution |
| | | 0223 | 4062 | | off page indirect execution |
| | | 0224 | 0520 | 01 | |
| | | 0225 | 1407 | | |

## 3.4 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, in the assembled program, and if A is loaded at 2000, then B will be loaded at 1000 + 1600 or 2600.

All programs to be assembled by SABR must be arranged to fit into one field of memory not counting page 0 of the field or the top page (7600 - 7777) of the field. 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 properly assembled by SABR.

## 3.5 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 wherever possible.

The symbol table, not counting permanent symbols, contains $2644_{10}$ words of storage. However, this space must be shared with the table when unresolved forward and external references are 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 in the program for $616_{10}$ symbols.

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

# CHAPTER 4

# SABR OPERATING PROCEDURES

This chapter describes how to assemble a program source using SABR. The procedure for loading a binary tape of an assembled program is described in Chapter 6.

## 4.1 LOADING SABR IN A BASIC PDP-8 SYSTEM

| Step | Procedure |
|---|---|
| 1 | Make sure the Binary Loader is in memory, say in field n. |
| 2 | Set the console switches as follows:<br>Instruction Field = n, Switch Register = 7777. |
| 3 | Press LOAD ADDress. |
| 4 | Insert the SABR binary tape into the reader. |
| 5 | If using the high-speed reader, depress Switch Register Bit 0. |
| 6 | Press START. |
| 7 | SABR will now be loaded into memory by the Binary Loader; portions of SABR will load into field 0 and field 1. |

## 4.2 LOADING SABR IN A DISK MONITOR SYSTEM

| Step | Procedure |
|---|---|
| 1 | Make sure the Disk Monitor is in memory. (Type CTRL/C† or START at 07600.) |
| 2 | When the Monitor responds with a dot, call the system Loader as follows:<br>.LOAD↲ (↲ represents typing the RETURN key) |
| 3 | Insert the SABR binary tape in the reader. |
| 4 | Answer the loading command dialogue as follows:<br>* IN-R:↲ for high speed reader or *IN-T:↲ for ASR reader<br>*<br>* ST = ↲<br>↑ <CTRL/P> ↑ <CTRL/P> |

After typing the second CTRL/P† it is necessary to reposition the tape in the reader for Pass 2.

---

†CTRL/C and CTRL/P are typed by holding down the CTRL key while typing the C or P key.

| Step | Procedure |
|---|---|
| 5 | SABR is now loaded into memory, partly in field 0 and partly in field 1. It may be saved on the user's system device by responding to the monitor's dot as follows: |

```
. SAVE SABR! 0-7177;200 ↲
. SAVE SAB1!  700,1700-12427; ↲
.
```

| | |
|---|---|
| 6 | SABR is now saved on the user's system device and may be called as follows: |

```
.SAB1 ↲
.SABR ↲
```

The field 1 portion must be called first.

## 4.3 OPERATING SABR

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 this listing is to be typed or punched. Pass 2 is the listing pass. The assembly is carried out as follows.

NOTE

If SABR has been saved on the System I/O device, as in Section 4.2, it will start automatically at Step 3 below when called into memory. The source tape (first section) should be inserted in the reader before operation begins.

| Step | Procedure |
|---|---|
| 1 | Set the console switches as follows: |
| | Data field = 0, Instruction Field = 0, Switch Register = 0200. |
| 2 | Press LOAD ADDress and START. |
| 3 | SABR now types a sequence of two or three questions; |

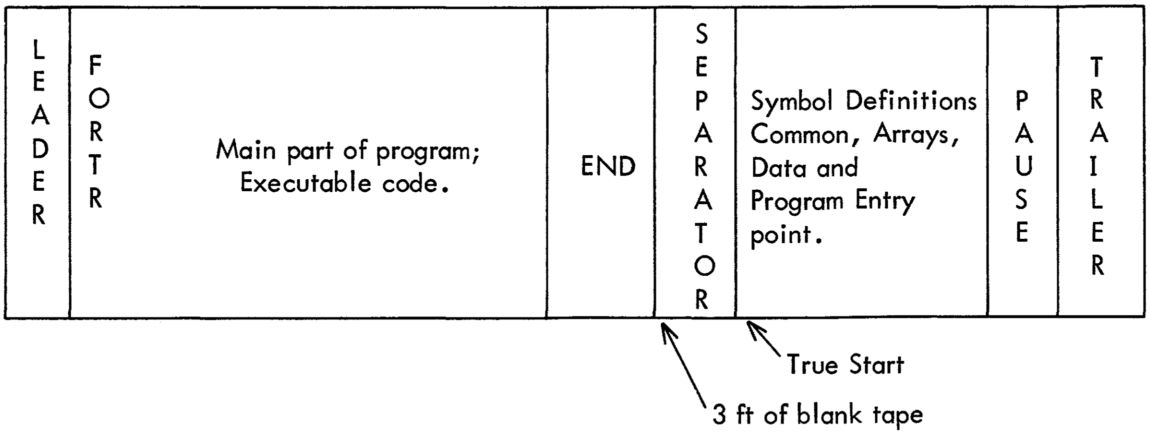
```
HIGH SPEED READER?
HIGH SPEED PUNCH?
LISTING ON HIGH SPEED PUNCH?
```

| Step | Procedure |
|---|---|
| | These questions must be answered with Y if the answer is yes. Any other answer is assumed to be no. The third question is typed only if the second is answered Y. If the third is answered Y, both the symbol table and the listing are punched on the high-speed paper tape punch. Otherwise, they are typed on the teletypewriter. The user need not wait for the full question to be typed before responding. |
| 4 | As soon as SABR has echoed the user's response to the last question, turn on the punch device and, if it is being used, the ASR reader. If the low-speed reader is used, the error message E indicates that the user has waited too long before turning the reader on. The user must begin again. |
| 5 | At this point, Pass 1 begins. SABR reads the source tape and punches the binary tape. After the binary tape has been completed, SABR types or punches the program symbol table. |
| 6 | If the source tape is in several sections (separate tapes with PAUSEs at the end of all except the last), SABR halts at the end of each section. At this point, insert the next section in the reader and then press CONTinue. |
| 7 | At the end of Pass 1, SABR halts. |
| 8 | If an assembly listing is desired, reposition the beginning of the source tape in the reader and if using the ASR reader, set it to START, and then press CONTinue. |
| 9 | At the end of Pass 2, SABR again halts. To restart SABR for assembling another program, press CONTinue. |
| 10 | To restart SABR at any time, press STOP, set the Switch Register = 0200, press LOAD ADDress and START. However, the first pass must always be repeated. |
| 11 | After assembling in a Disk Monitor environment, control may be returned to Monitor by restarting at location 7600. |

## 4.4 OPERATING PROCEDURE FOR USE AS FORTRAN PASS 2

In addition to being a stand-alone assembler, SABR also serves as Pass 2 of 8K FORTRAN compilation. For this purpose, the use of SABR is slightly different from that described in Section 4.3. However, SABR must still be loaded into memory as described in Section 4.1 or 4.2. This difference in the operation of SABR is due only to the unusual format of the FORTRAN Compiler Pass 1.

The Compiler, in one pass, converts the user's FORTRAN source into a symbolic machine language program tape. SABR then converts the symbolic tape into relocatable binary. However, the symbolic tape produced by the Compiler is not a standard format SABR language tape. It is arranged as shown below.




The tape is arranged this way because the data at the end of the tape cannot be inserted in the midst of the executable code, and some data which should be at the beginning of the tape is not known until later. Thus, the true start of the symbolic program is near the end of the symbolic tape, preceded by a segment of leader/trailer code and followed by a PAUSE statement.

To assemble such a tape with SABR, one of three methods must be followed. Actually, the general procedure is the same as that described in Section 4.3, but it differs in special details. The differences are all covered by the three methods explained below.

### 4.4.1 Method 1

The simplest method is to cut the symbolic tape into two parts. The cut should be made at the middle of the blank tape which separates the executable code from the symbol definitions. The latter section of the tape should then be marked "Section 1" and the former section (the executable code) should be marked "Section 2." Assembly then proceeds with the two-part symbolic tape exactly as described in Section 4.3.

### 4.4.2 Method 2

The user may avoid actually cutting the symbolic tape by manipulating the tape as if it were in two parts as explained above. The tape should initially be inserted in the reader with the separator blank tape over the read-head. When SABR halts at the PAUSE statement at the physical end of the tape, the user should reposition the tape, putting the physical beginning of the tape in the reader. Then press CONTinue. The assembly pass will end at the separator blank tape code. The assembly listing can be produced in a similar manner, pressing CONTinue to start the listing pass.

### 4.4.3 Method 3

The third method requires SABR to pass over the symbolic tape two times for each pass of the assembly. However, it allows the tape to be inserted at its physical beginning. It is based on the fact that a symbolic tape output by the FORTRAN Compiler has as its physical first line the special pseudo-op, FORTR. This pseudo-op has no effect except when a symbolic tape output by the Compiler is assembled using this third method.

| Step | Procedure |
|---|---|
| 1 | Insert the symbolic tape in the reader at its physical beginning. |
| 2 | Start SABR as usual. |
| 3 | Sensing the FORTR statement as the first line, SABR ignores all further data until after it passes over the END statement. SABR then begins the actual assembly by processing the symbol definitions, etc., which are at the latter end of the tape. |
| 4 | Then, SABR halts at the PAUSE statement which is at the physical end of the tape. At this time the user should reposition the symbolic tape in the reader at the physical beginning of the tape, and then press CONTinue. SABR now assembles the executable code portion of the tape in the normal way. |
| 5 | If an assembly listing is desired, proceed as in Method 2 after SABR finishes the assembly pass. |

# CHAPTER 5
# THE LINKING LOADER

## 5.1 INTRODUCTION

Relocatable binary program tapes produced by SABR assembly are loaded into memory by using the 8K System Linking Loader. The Linking Loader is capable of loading and linking a user's program and subprograms in any fields of memory. It is even capable, in a special way, of loading programs over itself. The Linking Loader also has options which give storage maps and core availability.

The Linking Loader requires a PDP-8/I, -8/L, -8, -8/S or -5 Computer with at least 8K words of core memory. Either high-speed or ASR paper tape input is acceptable, however, a high-speed reader is highly recommended.

The software requirements are:

a. Binary paper tape copy of the Linking Loader

b. Relocatable binary paper tape copies of both Part 1 and Part 2 of the 8K System Library

c. The relocatable binary paper tapes of the user's own program and subprograms which have been produced by assembling his programs with SABR.

## 5.2 LOADING WITH THE LINKING LOADER

Generally speaking, the Linking Loader is capable of loading any number of user and Library programs into any field of PDP-8 memory. These programs are loaded consecutively via the high-speed reader (or the ASR reader). The choice of which field to load each program into is a Switch Register option. Usually, several programs may be loaded into each field. Because of the space reserved for the Linkage Routines the available space in field 0 is three pages smaller than in all other fields.

Any COMMON storage reserved by the programs being loaded is allocated in field 1 from location 0200 upwards. The space reserved for COMMON is obviously subtracted from the available loading area in field 1. The program reserving the largest amount of COMMON storage must be loaded first.

The Linking Loader uses the following special method to enable loading data over itself. When the Linking Loader encounters data which must be loaded over itself, it punches this data onto paper tape in RIM format. Then, after the user has finished loading all his relocatable binary program tapes, he simply loads the RIM format tape using the standard RIM loader.

The Run-Time Linkage Routines which are necessary to execute SABR programs (see Section 3.3.4) 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 (see Appendix D).

The 8K System Library subprograms (See Appendix E), which may be used by any SABR program, are loaded in the same way as any other relocatable binary programs. Only those library programs which the user's programs actually call need to be loaded.

## 5.3 LOADING INFORMATION 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**

If the ASR punch is turned on, it must be turned off 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 |
| IREM | 00000 U |
| FSB | 04000 |
| FLOAT | 05046 |
| FIX | 04513 |
| IFIX | 04561 |
| CHS | 05231 |
| 0011 | |
| 0033 | |

## 5.4 HOW TO LOAD THE LINKING LOADER

The Linking Loader must be loaded into the highest available field of memory.

| Step | Procedure |
|---|---|
| 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:<br>Data Field = h, Instruction Field = m, Switch Register = 7777. |
| 4 | Press LOAD ADDress. |
| 5 | Place the binary paper tape of the Linking Loader in the reader. |
| 6 | If using a high-speed reader, depress Switch Register Bit 0. |
| 7 | Press START. The Linking Loader will now be loaded into memory. |

## 5.5 OPERATION OF THE LINKING LOADER

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.)

| Step | Procedure |
|---|---|
| 1 | After the Linking Loader has been loaded into the highest memory field, h, the user should set the console switches as follows: Data Field = h, Instruction Field = h, Switch Register = 0200. |
| 2 | Press LOAD ADDress. |
| 3 | Place the relocatable binary tape for the first program to be loaded in the reader. Position the tape with leader code in the reader. |
| 4 | Set Switch Register to 0000. Then, if loading via the ASR reader is required, raise Switch Register Bit 6. If the user <u>does not</u> have a high-speed punch, he should raise Switch Register Bit 7. Finally, set Switch Register Bits 9-11 to the number of the field into which the first program or subprogram is to be loaded. |

Switch Register*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | | | | 1 | 1 | | 0 | 1 | 1 |
| core pages | storage map | | | | | ASR reader | ASR punch | | Number of Loading Field | | |

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.

| Step | Procedure |
|---|---|
| 5 | Press START. |
| 6 | The user's relocatable binary program will now be loaded. When loading is completed, the Linking Loader halts. |
| 7 | The user may now either load another program or select one of the options in steps 9 and 10. |
| 8 | To load another program, insert the program relocatable binary tape in the reader, set Switch Register Bits 9-11 to the number of the field the program is to be loaded into, and then press CONTinue. |
| 9 | To select the Core Availability option, set Switch Register Bit 0 = 1, and press CONTinue. |
| 10 | To select the Storage Map option, set Switch Register Bit 1 = 1, and press CONTinue. |

If the ASR punch is turned on for possible RIM format data punching, as explained in Section 5.2, ensure that it is turned off before selecting either of the options. Turn it on again after the typing of the option is completed.

*All other Switch Register bits are irrelevant.

| Step | Procedure |
|---|---|
| 11 | The user may continue loading more programs as in step 8 after using either of the options. |
| | Any time the Linking Loader halts, the user may access memory directly via the DEPosit and EXAMine console 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). |

# CHAPTER 6
# DEMONSTRATION PROGRAM

The following demonstration program is a SABR program showing the use of the library routines. The program is written to add two integer numbers, convert the result into floating-point, and type the result in both integer and floating-point format. The source program was written and listed using the Symbolic Editor; the Disk Monitor System was used during assembly; and the assembled program was then loaded and run using the 8K Linking Loader.

The system configuration consisted of a PDP-8/I with 8K words of core, DF32 Disk, ASR33 Teletype, and high-speed reader and punch. The Disk Monitor System, Symbolic Editor, and SABR Assembler were available on the disk. The ASR33 paper tape reader was used during assembly for demonstration (printout) purposes.

## Demonstration Program

**Program** / **Comment**

After writing the source program it was printed and punched using the Symbolic Editor.

```
        ENTRY START

START,  CALL    Ø,OPEN  /INITIALIZE IO DEVICES
        TAD     A       /COMPUTE C = A + B
        TAD     B
        DCA     C
        CALL    1,FLOAT /CONVERT TO FLOATING POINT
        ARG     C
        CALL    1,STO
        ARG     D
        CALL    2,WRITE /INITIALIZE THE IO HANDLER
        ARG     N       /DEVICE NUMBER 1 = TELETYPE
        ARG     FORMT   /FORMAT SPECIFICATION
        CALL    1,IOH   /TYPE THE INTEGER NUMBER
        ARG     C
        CALL    1,IOH   /TYPE THE FLOATING POINT NR
        ARG     D
        CALL    1,IOH   /COMPLETE THE IO
        ARG     Ø
        HLT

FORMT,  TEXT "('THE ANSWERS ARE',I5,F7.2)"
N,      1
A,      2
B,      2
C,      Ø
D,      BLOCK   3
        END
```

Demonstration Program

| Program | Comment |
|---|---|
| * | CTRL/C was typed after the asterisk to return control to the Disk Monitor. |
| .SAB1<br>.SABR | SABR was transferred from the disk into core. |
| | The source program tape was placed in the teletype reader. |
| | When started, SABR printed its identification and initial dialogue questions which were answered. |
| PDP-8 SABR DEC-08-A2B2-12<br>HIGH SPEED READER? N<br>HIGH SPEED PUNCH? Y<br>LISTING ON HIGH SPEED PUNCH? N<br><br>E AT +0000 | The TTY reader must be set to START within 3 seconds after typing N to the last question. Otherwise, as was the case here, the error message will appear, and SABR must be restarted at location 0200, as was done here. |
| HIGH SPEED READER? N<br>HIGH SPEED PUNCH? Y<br>LISTING ON HIGH SPEED PUNCH? N | The initial dialogue questions are repeated and again answered. |
| | After typing the N to the last question, the TTY reader was immediately set to START and assembly commenced. |
| A 0257<br>B 0260<br>C 0261<br>D 0262<br>FLOAT 0000EXT<br>FORMT 0240<br>IOH 0000EXT<br>N 0256<br>OPEN 0000EXT<br>START 0200EXT<br>STO 0000EXT<br>WRITE 0000EXT | The Symbol Table concluded the assembly. |
| | Here the source program tape was again placed in the TTY reader and the CONTinue switch was depressed. The program listing was printed. |

```
                                ENTRY START

Ø2ØØ   4Ø33          START,   CALL     Ø,OPEN   /INITIALIZE IO DEVICES
Ø2Ø1   ØØØ2 Ø6
Ø2Ø2   1257                   TAD      A        /COMPUTE C = A + B
Ø2Ø3   126Ø                   TAD      B
Ø2Ø4   3261                   DCA      C
Ø2Ø5   4Ø33                   CALL     1,FLOAT  /CONVERT TO FLOATING POINT
Ø2Ø6   Ø1Ø3 Ø6
Ø2Ø7   62Ø1 Ø5                ARG      C
Ø21Ø   Ø261 Ø1
Ø211   4Ø33                   CALL     1,STO
Ø212   Ø1Ø4 Ø6
Ø213   62Ø1 Ø5                ARG      D
Ø214   Ø262 Ø1
Ø215   4Ø33                   CALL     2,WRITE  /INITIALIZE THE IO HANDLER
Ø216   Ø2Ø5 Ø6
Ø217   62Ø1 Ø5                ARG      N        /DEVICE NUMBER 1 = TELETYPE
Ø22Ø   Ø256 Ø1
Ø221   62Ø1 Ø5                ARG      FORMT    /FORMAT SPECIFICATION
Ø222   Ø24Ø Ø1
Ø223   4Ø33                   CALL     1,IOH    /TYPE THE INTEGER NUMBER
Ø224   Ø1Ø6 Ø6
Ø225   62Ø1 Ø5                ARG      C
Ø226   Ø261 Ø1
Ø227   4Ø33                   CALL     1,IOH    /TYPE THE FLOATING POINT NR
Ø23Ø   Ø1Ø6 Ø6
Ø231   62Ø1 Ø5                ARG      D
Ø232   Ø262 Ø1
Ø233   4Ø33                   CALL     1,IOH    /COMPLETE THE IO
Ø234   Ø1Ø6 Ø6
Ø235   6211                   ARG      Ø
Ø236   ØØØØ
Ø237   74Ø2                   HLT

Ø24Ø   5Ø47          FORMT,   TEXT "('THE ANSWERS ARE',I5,F7.2)"
Ø241   241Ø
Ø242   Ø54Ø
Ø243   Ø116
Ø244   2327
Ø245   Ø522
Ø246   234Ø
Ø247   Ø122
Ø25Ø   Ø547
Ø251   5411
Ø252   6554
Ø253   Ø667
Ø254   5662
Ø255   51ØØ
Ø256   ØØØ1          N,       1
Ø257   ØØØ2          A,       2
Ø26Ø   ØØØ2          B,       2
Ø261   ØØØØ          C,       Ø
Ø262   ØØØØ          D,       BLOCK    3
Ø263   ØØØØ
Ø264   ØØØØ
                              END
```

Demonstration Program

| Program | Comment |
|---|---|

The 8K Linking Loader was loaded into core using the Binary Loader, and started at location 0200 of field 1.

When started, the Linking Loader printed its identification.

```
PDP-8 LINKING LOADER DEC-08-A2B3-06
```

Library Tape Part 1 was loaded into core by placing the tape in the TTY reader, setting the reader to START, and pressing CONTinue.

After loading the library subprograms, Switch Register bit 1 was set to 1, and CONTinue was pressed to get the storage page of the programs and subprograms loaded into core.

```
START   01000
OPEN    06125
FLOAT   05034
STO     04444
WRITE   01302
IOH     03142
READ    01271
SETERR  06200
ERROR   06303
TTYOUT  06027
HSOUT   06055
TTYIN   06000
HSIN    06045
FDV     04711
CLEAR   05227
IFAD    05116
FMP     04623
ISTO    05061
FLOT    05153
FAD     04010
DIV     05445
IREM    05616
FSB     04000
FIX     04510
IFIX    04556
CHS     05211
ABS     05636
IABS    05670
MPY     05400
IRDSW   05713
CKIO    06121
EXIT    06142
CLRERR  06231
0004
0036
```

The last two numbers represent the number of free (available) pages in each core field -- 0004 free pages in field 0, and 0036 free pages in field 1.

| Program | Comment |
| --- | --- |
| | To execute the compiled program, the Switch Register was set to 01000, the starting address of the main program (determined from the Storage Map). |
| THE ANSWERS ARE 4 4.00 | The LOAD ADDress switch was pressed and then START switch was pressed. |
| | The program ran as planned, producing the desired results. |

## APPENDIX A
## ASCII* CHARACTER SET

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

* An abbreviation for U.S.A. Standard Code for Information Interchange.

# APPENDIX B

# PERMANENT SYMBOL TABLE

## Memory Reference Instructions

| Mnemonic | Code | Operation | Event Time |
|---|---|---|---|
| AND | 0000 | combine C(AC) and C(MEM) by logical AND and store result in AC | |
| TAD | 1000 | combine C(AC) and C(MEM) by two's complement addition and store result in AC with carry added to the LINK | |
| ISZ | 2000 | increment C(MEM) and skip if result is 0 | |
| INC | 2000 | same as ISZ except should be used only when it is known that an actual skip cannot occur (see Section 3.3.5) | |
| DCA | 3000 | deposit C(AC) into MEM and clear the AC | |
| JMS | 4000 | jump to subroutine (actually deposit the current value of the PC into MEM and jump to MEM + 1) | |
| JMP | 5000 | jump to MEM location | |
| I | 0400 | indirect memory reference | |

## Operate Microinstructions: Group 1

| Mnemonic | Code | Operation | Event Time |
|---|---|---|---|
| CLA | 7200 | clear AC | 1 |
| CLL | 7100 | clear LINK | 1 |
| CMA | 7040 | complement AC | 2 |
| CML | 7020 | complement LINK | 2 |
| RAR | 7010 | rotate AC and LINK 1 bit right | 4* |
| RTR | 7012 | rotate AC and LINK 2 bits right | 4* |
| RAL | 7004 | rotate AC and LINK 1 bit left | 4* |
| RTL | 7006 | rotate AC and LINK 2 bits left | 4* |
| IAC | 7001 | increment AC | 3 |
| CIA | 7041 | negate AC (CMA IAC combined) | 1,3 |
| STA | 7240 | set AC (CLA CMA combined) | 1,2 |
| STL | 7120 | set LINK (CLL CML combined) | 1,2 |
| NOP | 7000 | no operation | 1 |

## Operate Microinstructions: Group 2

| Mnemonic | Code | Operation | Event Time |
|---|---|---|---|
| CLA | 7600 | clear AC | 2 |
| SMA | 7500 | skip if AC negative | 1 |
| SZA | 7440 | skip if AC zero | 1 |
| SPA | 7510 | skip if AC positive or zero | 1 |
| SNA | 7450 | skip if AC non-zero | 1 |
| SNL | 7420 | skip if LINK non-zero | 1 |
| SZL | 7430 | skip if LINK zero | 1 |

* 3 for PDP-8

## Operate Microinstructions: Group 2 (Cont)

| Mnemonic | Code | Operation | Event Time |
|---|---|---|---|
| SKP | 7410 | skip unconditionally | 1 |
| SPC | 7710 | (SPA CLA combined) | 1 |
| OSR | 7404 | inclusive OR switch register with C(AC); result to AC | 3 |
| HLT | 7402 | halt | 4 |

## IOT Microinstructions

| | | |
|---|---|---|
| Program Interrupt | | |
| ION | 6001 | turn interrupt on |
| IOF | 6002 | turn interrupt off |
| Keyboard/Reader | | |
| KSF | 6031 | skip if keyboard/reader flag =1 |
| KRB | 6036 | clear AC & read keyboard buffer, and clear keyboard flag |
| Teleprinter/Punch | | |
| TSF | 6041 | skip if teleprinter/punch flag = 1 |
| TLS | 6046 | load teleprinter/punch buffer, select and print, and clear teleprinter/punch flag |
| High-Speed Reader (Type PC02) | | |
| RSF | 6011 | skip if reader flag = 1 |
| RRB | 6012 | read reader buffer and clear flag |
| RFC | 6014 | clear flag and buffer and fetch character |
| High-Speed Punch (Type PC03) | | |
| PSF | 6021 | skip if punch flag =1 |
| PLS | 6026 | clear flag and buffer, load and punch |

## Pseudo-Operators

| | |
|---|---|
| ABYSM | Direct Absolute Symbol Definition |
| ARG | Argument for Subroutine Call |
| BLOCK | Reserve Storage Block |
| CALL | Call External Subroutine |
| COMMN | Common Storage Definition |
| CPAGE | Check if Page Will Hold Data |
| DECIM | Decimal Conversion |
| DUMMY | Dummy Argument Definition |
| EAP | Enter Automatic Paging Mode |
| END | End of Program |
| ENTRY | Define Program Entry Point |
| FORTR | Assemble FORTRAN Tape |
| IF | Conditional Assembly |
| LAP | Leave Automatic Paging |
| OCTAL | Octal Conversion |
| OPDEF | Define Non-Skip Operator |
| PAGE | Terminate the Page |

Pseudo-Operators (Cont)

| Mnemonic | Code | Operation |
|---|---|---|
| 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.

# APPENDIX C
# ERROR MESSAGES

## C.1 SABR

Because SABR is a one-pass automatic paging assembler for binary relocatable programs, object errors are difficult to correct. If there are errors in the source, the assembled binary code will be virtually useless. Both errors E and S are fatal; assembly halts when they are encountered. The other types of errors are not fatal, but they cause the line in which they occur to be treated as a comment and thus essentially ignored. An address label on such a line will remain undefined and no space is reserved in the binary output for the erroneous data.

During the assembly pass error messages are typed on the teletype as they occur.

Example:

```
C        AT        LOC        +0004
```

This means that an error of type C has occurred at the fourth instruction after the location tag LOC. This line count includes comment lines and blank lines.

During the listing pass, the error is typed in the address field of the instruction line.

The following error messages may occur.

A Too many or too few ARGs follow a CALL statement.

C An illegal character appears on the line. This could possibly be an 8 or 9 in an octal digit string or an alphabetic character in a digit string.

M A symbol is multiply defined (occurs only during Pass 1). It is impossible to resolve multiple definitions during Pass 2; therefore, listings of programs which contain multiple definitions will have unmarked errors.

I An illegal syntax has been used. Below are listed the types of illegal syntax that may occur.

a. A pseudo-op with improper arguments.
b. A quote mark with no argument.
c. A non-terminated text-string.
d. A memory reference instruction with improper address.
e. An illegal combination of micro-instructions.

E There is no END statement.

S one of the following:

a. The symbol table has overflowed. This can be corrected by using fewer symbols, using shorter symbols, or by breaking the program into smaller parts.

b. Common storage has been exhausted.

c. More than 64 different user-defined symbols have occurred in a core page.

d. More than 64 external symbols have been declared.

One further type of error may occur. This is an undefined symbol. Because SABR is a one-pass assembler, an undefined symbol cannot be determined until the end of the assembly pass, so the error diagnostic UNDF is given in the symbol table listing. (Refer to the discussion of the Symbol Table at the end of Appendix F.)

## C.2 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.

| 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 nonexistent memory. |
| 0003 | The current subprogram has too large a COMMON storage assignment. (Subprogram with largest common storage declaration must be loaded first.) This is a semi-fatal error. Re-initialize the Linking Loader as explained below and reload the programs in the proper order. |
| 0004 | Checksum error in input tape. If the error persists, re-assembly is necessary. |
| 0005 | Illegal Relocation Code has been encountered. This can occur only if the relocatable binary tape is bad or if the user is using it improperly (e.g., not starting at the beginning of the tape, or reader error, or punch error). If the error persists, re-assembly is necessary. |

Recovery from errors 2, 4, and 5 is accomplished by repositioning the tape in the reader to the leader code at the beginning of the subprogram and then pressing CONTinue. When attempting to recover from one of these errors, no other program should be loaded before reloading the program which caused the error. Obviously, on Error 2 a different field should be selected before pressing CONtinue.

The entire loading process may be restarted via the console switches, at any time by re-initializing the Linking Loader. To do this, set the console switches as follows: Data Field = h (the field where the Linking Loader resides), Instruction Field = h, Switch Register = 6200; then press LOAD ADDress and START.

## C.3 LIBRARY PROGRAM

During execution, the Library programs check for certain errors and type out the appropriate error messages in the form

"XXXX" ERROR AT LOC NNNN

where XXXX specifies the type of error, and NNNN is the location of the error. When an error is encountered, execution stops, and the error must be corrected.

When multiple error messages are typed, the location of the last error message is relevant to the user program. The other error messages are to subprograms called by the statement at the relevant location.

| Error Code | 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 |
| "FLPW" | Negative number raised to floating power |
| "FPNT" | Floating - point error may be caused by: Division by zero; floating - point overflow; attempting to fix too large a number. |
| "SQRT" | Attempt to square root a negative number |

To pinpoint the location of a Library execution error:

| Step | Procedure |
| --- | --- |
| 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. |

# APPENDIX D

## FREE PAGE 0 LOCATIONS

Because the Library Linkage Routines must be in core when SABR assembled programs are run, certain core locations are not available as follows:

| | |
|---|---|
| Field 0 | Locations 0400 - 0777 |
| Field 0, 1, 2,... | Locations 0007 and 0033 - 0073 |

Thus in every field of memory the following page 0 locations are available to the user:

| | |
|---|---|
| 0000 - 0006 | for interrupts, debugging, etc. |
| 0010 - 0017 | auto-index registers |
| 0023 - 0032 | arbitrary |
| 0074 - 0177 | arbitrary |

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.

# APPENDIX E

# THE LIBRARY SUBPROGRAMS

The Library is a set of subprograms which may be CALLed by any FORTRAN/SABR program. The relocatable binary versions of these subprograms are arranged in two paper tapes for the convenience of the user. Part 1 contains those subprograms which are used by almost every FORTRAN/SABR program. All the Library subprograms are described below.

Many of the subprograms reference the Floating-Point Accumulator located at ACH, ACM, ACL (20, 21, 22 of field 1).

## E.1 INPUT/OUTPUT

READ is called to initialize the I/O handler before reading data. WRITE is called to initialize the I/O handler before writing data. IOH is called for each item to be read or written. IOH must also be called with a zero argument to terminate an input-output sequence. (Refer to Chapter 6.)

All of these programs require that the Floating-Point Accumulator be set to zero before they are called.

Examples:

```
CALL     2, READ
ARG      (n          /n =DEVICE NUMBER
ARG      fa          /fa =ADDR OF FORMAT
...
CALL     1,IOH
ARG      data 1      /data 1 =ADDR OF HIGH
                     /ORDER WORD OF
                     /FLOATING POINT NUMBER
CALL     1,IOH
ARG      data 2
...
...
CALL     1,IOH
ARG      0
...
CALL     2,WRITE
ARG      (n
ARG      fa
```

The following device numbers are currently implemented:

1. Teletype keyboard/printer
2. High-speed reader/punch

## E.2 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 Accumulator in the argument address

```
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.

```
CALL    0,CLEAR
```

FLOT is called to convert the integer contained in the AC (processor accumulator) to a floating point number and store it in the Floating-Point Accumulator.

```
CALL        0,FLOT
```

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

```
CALL        0,FIX
```

ABS leaves the absolute value of the floating point number at "addr" in the Floating-Point Accumulator.

```
CALL        1,ABS
ARG         addr
```

## E.3 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
```

## E.4 SUBSCRIPTING

SUBSC is called to compute the address of a subscripted variable. The address is left in the AC. When SUBSC is called, it assumes that the AC contains the first dimension of the array. This dimension should be positive if the subscripted variable is an integer, and negative if the subscripted variable is a floating point number.

Example:

Assume S is a $20_8$ X $20_8$ floating-point array.

```
TAD       (20
CIA
CALL      3,SUBSC
ARG       i1          /i1=ADDRESS OF 2ND
                      /SUBSCRIPT
ARG       i2          /i2=ADDRESS OF 1ST
                      /SUBSCRIPT
ARG       base        /BASE ADDRESS
                      /OF ARRAY
```

## E.5 FUNCTIONS

SQRT leaves the square root of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL      1,SQRT
ARG       addr
```

SIN, COS, TAN leave the specified function of the floating-point argument at "addr" in the Floating-Point Accumulator.

```
CALL      1,SIN
ARG       addr
```

ATAN leaves the arctangent of the floating-point number at "addr" in the Floating-Point Accumulator.

```
CALL      1,ATAN
ARG       addr
```

ALOG leaves the natural logarithm of the floating-point number of "addr" in the Floating-Point Accumulator.

```
CALL      1,ALOG
ARG       addr
```

EXP raises "e" to the power specified by the floating-point number at "addr" and leaves the result in the Floating-Point Accumulator.

```
CALL    1,EXP
ARG     addr
```

All of these subprograms require that the Floating-Point Accumulator be set to zero before they are called.

## E.6 POWERS (IIPOW, IFPOW, FIPOW, FFPOW)

These routines are called by FORTRAN to implement exponentiation. The address of the first operand is in the AC (floating-point or processor depending on mode), and the address of the second is an argument. The address of the result is in the appropriate AC upon return.

| Function Name | Mode of Operand 1 (Base) | Mode of Operand 2 (Exponent) | Mode of Result |
|---|---|---|---|
| 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
```

## E.7 LIBRARY ORGANIZATION

| | | | |
|---|---|---|---|
| Part 1. | "IOH" | contains | IOH, READ, WRITE |
| | "FLOAT" | contains | FAD, FSB, FMP, FDV, STO, FLOT, FLOAT, FIX, IFIX, IFAD, ISTO, CHS, CLEAR |
| | "INTEGER" | contains | 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 |

## E.8 DECTAPE I/O ROUTINES

RTAPE and WTAPE (read tape and write tape) are the DECtape read and write subprograms for the 8K FORTRAN and 8K SABR systems. The subprograms are furnished on one relocatable binary-coded paper tape which must be loaded into field 0 by the 8K Linking Loader, where they occupy one page of core.

RTAPE and WTAPE allow the user to read and write any amount of core-image data onto DECtape in absolute, non-file-structured data blocks. Many such data blocks may be stored on a single tape, and a block may be from 1 to 4096 words in length.

RTAPE and WTAPE are subprograms which may be called with standard, explicit CALL statements in any 8K FORTRAN or SABR program. Each subprogram requires four arguments separated by commas. The arguments are the same for both subprograms and are formatted in the same manner. They specify the following:

a. DECtape unit number (from 0 to 7).

b. Number of the DECtape block at which transfer is to start. The user may direct the DECtape service routine to begin searching for the specified block in the forward direction rather than the usual backward direction by making this argument the two's complement of the block number.

c. Number of words to be transferred ($1 \leq N \leq 4096$).

d. Core address at which the transfer is to start.

In 8K FORTRAN, the CALL statements to RTAPE and WTAPE are written in the following format (arguments are taken as decimal numbers):

```
CALL RTAPE (6,128,388,LOCA)
```

In 8K SABR, they are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4, WTAPE        /WOULD BE SAME FOR RTAPE
ARG (6               /DATA UNIT NUMBER
ARG (200             /STARTING BLOCK NUMBER IN OCTAL
ARG (604             /WORDS TO BE TRANSFERRED IN OCTAL
ARG LOCB             /CORE ADDRESS, START OF TRANSFER
```

In these examples, LOCA and LOCB may or may not be in COMMON.

As a typical example of the use of RTAPE and WTAPE, assume that the user wants to store the four arrays A, B, C, and D on a tape with word lengths of 2000, 400, 400, and 20 respectively. Since PDP-8 DECtape is formatted with 1612 blocks (numbered 1-2700 octal) of 129 words each (for a total of 207,948 words), A, B,C, and D will require 16, 4, 4, and 1 blocks respectively. Each array must be stored beginning at the start of some DECtape block. The user may write these arrays on tape as follows:

```
CALL WTAPE (0,1,2000,A)
CALL WTAPE (0,17,400,B)
CALL WTAPE (0,21,400,C)
CALL WTAPE (0,25,20,D)
```

The user may also read or write a large array in sections by specifying only one DECtape block (129 words) at a time. For example, B could be read back into core as follows:

```
CALL RTAPE (0,17,258,B(1))
CALL RTAPE (0,19,129,B(259))
CALL RTAPE (0,20,13,B(388))
```

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

A DECtape read or write is normally initiated with a backward search for the desired block number. To save searching time, the user may request RTAPE or WTAPE to start the block number search in the forward direction. This is done by specifying the negative of the block number. This should be used only if the number of the next block to be referenced is at least fourteen block numbers greater than the last block number used. For example, if the user has just read array A and now wants array D, he may write:

```
CALL RTAPE (0,1,2000,A)
CALL RTAPE (0,-25,20,D)
```

## E.9 DISK I/O ROUTINES (preliminary)

ODISK and CDISK (open disk and close disk) and RDISK and WDISK (read disk and write disk) are the four DECdisk (DF32/DS32) input and output subprograms for the 8K FORTRAN and 8K SABR systems. They are furnished on one relocatable binary-coded paper tape which is loaded into core using the Linking Loader, where they occupy eight pages of core.

### E.9.1 ODISK and CDISK

ODISK is used to open (activate) a file (named using the Linking Loader D function) so that the file can be read or written using RDISK or WDISK. CDISK will close (deactivate) a file which was opened with ODISK so that the contents of the file cannot be altered.

The ODISK and CDISK subprograms may be called with standard, explicit CALL statements, in any 8K FORTRAN or 8K SABR program. ODISK requires one argument when opening a file. However, it requires two arguments when specifying or changing the size (in blocks) of a file. CDISK always requires only one argument.

The first argument of both ODISK and CDISK is the logical number (from 1 through 10 inclusive) of the file as it was named using the Linking Loader. (Refer to Section H.3.1 for a discussion of logical file numbers.) The second argument to ODISK is the number of blocks (from 1 through 128) to be saved for the file.

In 8K FORTRAN, the CALL statements to ODISK and CDISK are written in the following format (arguments must be decimal integer numbers):

```
CALL ODISK (1)
```

when opening a file, or

```
CALL ODISK (1,5)
```

when specifying or changing the size of a file, and

```
CALL CDISK (1)
```

when closing an opened file.

In 8K SABR, the CALL statements to ODISK and CDISK are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 1, ODISK
ARG (1                  /LOGICAL FILE NUMBER
ARG (5                  /NUMBER OF BLOCKS, OCTAL
```

when specifying or changing the size of a file, and

```
CALL 1,CDISK
ARG (1                  /LOGICAL FILE NUMBER
```

when closing an opened file.

ODISK prepares the file named for data transfer. When running the user program using the Disk Monitor System, ODISK uses Disk Monitor I/O and the three scratch blocks on disk zero for a window whenever a file is opened.

All open files should be closed before terminating program execution, thus preserving the contents of the files.

### E.9.2 RDISK and WDISK

The RDISK and WDISK subprograms may be called with standard, explicit CALL statements in any 8K FORTRAN or 8K SABR program. The ODISK subprogram must be used to open the file concerned before using the RDISK or WDISK subprograms.

Each of these subprograms requires four arguments, arranged as listed below:

1. Logical file number (determined using the Linking Loader D function).

2. Logical block of the file number (block number of the file where data transfer is to begin),

3. Number of words to be transferred (from 1 through 4096)

4. Core address where data transfer is to start (field 0).

Both RDISK and WDISK require the arguments above.

In 8K FORTRAN, the CALL statements to RDISK and WDISK are written in the following format (arguments are taken as decimal numbers):

```
CALL RDISK (4,2,55,LOCA)
```

when reading file 4, beginning with block 2, transferring 55 words, starting at the location of tag LOCA, which may be the name of an array defined in a DIMENSION statement. WDISK would be formatted in the same fashion.

In 8K SABR, the CALL statements to RDISK and WDISK are written in the following format (arguments may be either octal or decimal numbers):

```
CALL 4, RDISK        /SAME FOR WDISK
ARG (4               /LOGICAL FILE NUMBER
ARG (2               /BLOCK OF FILE
ARG (55              /WORDS TO TRANSFER, OCTAL
ARG LOCA             /CORE ADDRESS OF START, FIELD 0
```

WDISK would be formatted in the same fashion.

A variable number of words may be transferred. It is not necessary to transfer in 200-word blocks, as with the Disk Monitor System.

## APPENDIX F

## SAMPLE OF AN ASSEMBLY LISTING

This program is offered only to illustrate many of the features and formats of a SABR program. The program cannot be run.

```
PDP-8 SABR DEC-08-A2B2
High Speed Reader?  Y
High Speed Punch?  Y
Listing on High Speed Punch?  N

          DTCA      67620P
          DTSF      67710P
          LOC       0000UNDF
          MUL       0000EXT
          NAME      1000COM
          POINTR    1013
          SUB       0200EXT
          S↑        0202
          S↑2       0214
          S2        0214
          S3        0227
          S4        0233
          TAG       0177ABS
          X         0400
          Y         0401
          Z         0402

                                        /SAMPLE OF SABR CODE

                    6762          OPDEF      DTCA        6762
                    6771          SKPDF      DTSF        6771
                                  /ABSYM     LOC         176
                    0177          ABSYM      TAG         177
                                             DECIM
                    200           NAME,      COMMN       8
                                             ENTRY       SUB
                                             DUMMY       X
                                             LAP
          0200      0000          SUB,       BLOCK       2
          0201      0000
                                             EAP
                                             OCTAL
          0202      0000          S↑,        0
          0203      4067                     TAD I       SUB
          0204      0200  01
          0205      1407
          0206      7106                     CLL RTL;    RTL
          0207      7006
          0210      6211                     DCA         NAME#
```

```
0211      3776
0212      6201  05                  INC       POINTR
0213      2775
                       S↑2,
0214      4033         S2,          CALL      3,MUL
0215      0302  06
0216      6201  05                  ARG       X
0217      0400  01
0220      6201  05                  ARG       (20
0221      0374  01
0222      6201                      ARG       -1
0223      7777
0224      1373                      TAD       (D-49
                                    IF        LOC,1
                                    PAUSE
0225      1372                      TAD       (-"?
0226      5200                      JMP       SUB
                                    CPAGE     4
0227      4233         S3,          JMS       S4
0230      0004                      4
0231      0200                      NAME
0232      0371  01                  (37
0233      6762         S4,          DTCA
0234      5377
0371      0037
0372      7501
0373      7717
0374      0020
0375      1013  01
0376      1001
0377      7000
                                    PAGE
0400      0000         X,           0
0401      0214  01     Y,           S↑2
0402      2301         Z,           TEXT      "SAMP @ = */?456"
0403      1520
0404      0075
0405      4052
0406      5777
0407      6465
0410      6600
0411      6771                      DTSF
0412      5376
0413      5377
0576      7410
0577      7410
                                    REORG     1000
1000      7410                      SKP
1001      7410                      TAD I     S↑2
1002      5206
1003      4062
1004      0214  01
```

```
1005        1407
1006        1377                    TAD         (333
1007        6211                    DCA         NAME
1010        3776
1011        4040                    RETRN       SUB
1012        0001  06
1013        0000        POINTR,     0
1176        1000
1177        0333
                                    END
```

For a multiple word instruction the actual instruction line is typed beside the first instruction.

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

For an erroneous instruction, the error flag appears in the address field. The instruction is not assembled.

```
0700        7200        N2,         CLA
    I                               CLL SKP
0701        7402                    HLT
```

The page escape and literal and off-page pointer table are typed with nothing except the correct address, value and loader code.

```
0770        7006        N3,         RTL
0771        7500                    SMA
0772        5376
0773        5377
0774        0200  01
0775        0020
0776        7410
0777        7410
```

## F.1 THE SYMBOL TABLE

Symbols are listed in alphabetic order at the end of the assembly pass (Pass 1) with their relative addresses beside them.

The following flags are added to special types of symbols.

| | |
|---|---|
| ABS | The address is absolute. |
| COM | The address is in COMMON. |
| OP | The symbol is an operator. |

| | |
|---|---|
| EXT | The symbol is an external and thus, may or may not be defined. If not defined there is no difficulty; it is 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 until the end of Pass 1 that the symbol is undefined. A location is reserved for the instruction containing the undefined symbol, but nothing is placed in it. |

# APPENDIX G
# OPERATING PROCEDURES

This appendix is a condensation of Chapter 4. The figures referenced (in parentheses) are found in the PDP-8/I System User's Guide, DEC-08-NGCC-D.

## G.1 LOADING THE SABR ASSEMBLER

| Step | Procedure |
|---|---|
| 1 | Load the SABR Assembler using BIN (See Figure B-2); IF=1 SR=7777. When loaded, parts of the Assembler will be in field 0 and field 1.<br><br>To load the Assembler on the disk, proceed in step sequence, otherwise, begin at step 4, below. |
| 2 | With the Disk Monitor in memory, call the Disk System Loader by typing:<br><br>.LOAD<br><br>and load the SABR assembler onto the disk (see PDP-8/I Disk/DECtape Monitor System, DEC-D8-SDAB-D). |
| 3 | Save the Assembler by typing:<br><br>.SAVE SABR!0-7177; 200<br>.SAVE SAB1!~~10600~~, 1700 - 12427; 2 [handwritten: 700] |

errata sheet May 30, 69

## G.2 ASSEMBLING (Pass 1)

See section 4.4 for alternate methods of assembling.

| Step | Procedure |
|---|---|
| 4 | Insert source program tape into the tape reader. |
| 5 | Set DF=0, IF=0, SR=0200, press LOAD ADD, START, and answer SABR's initial dialogue. |
| 6 | Turn the appropriate punch and reader ON; the tape reads in and the binary tape is punched. |
| 7 | If the program is in sections, when a PAUSE is encountered, insert the next section of tape into the tape reader and press CONT; assembly is completed when SABR halts after producing the relocatable binary tape.<br><br>SABR may be restarted to assemble another program by starting over at step 4 above.<br><br>SABR may be restarted at any time by pressing STOP and starting over at step 4. |

| Step | Procedure |
| --- | --- |
| | To generate an assembly listing, proceed in step sequence, otherwise, begin at step 9. |
| 8 | Insert the source program tape(s) into the reader and press CONT. |

## G.3 LOADING THE LINKING LOADER

| Step | Procedure |
| --- | --- |
| 9 | Set DF=highest field in the configuration, IF=1, SR=7777, and press LOAD ADD. |
| 10 | Insert Linking Loader tape into the appropriate reader: if ASR reader, turn reader ON; if high-speed reader, set SR=3777. |
| 11 | Press START; the Linking Loader will be read into core memory. |

## G.4 LOADING PROGRAMS AND SUBPROGRAMS

| Step | Procedure |
| --- | --- |
| 12 | Set DF and IF=to DF in step 9 above, SR=0200, and press LOAD ADD. |
| 13 | Insert relocatable binary tape (first, program or subprogram with largest amount of COMMON storage) into the reader with leader code over reader head. |
| 14 | Set SR as explained in Section 5.5. |
| 15 | Press START; the relocatable binary program will be loaded into core memory. |

Repeat from step 13 for subsequent program or subprogram tapes or select an option (core availability or storage map) as explained in Section 5.3.

## G.5 EXECUTING THE SABR PROGRAM

| Step | Procedure |
| --- | --- |
| 25 | Set DF and IF=to field of MAIN program, and SR=to starting address of MAIN program (determined from storage map). |
| 26 | Turn punch ON and/or insert data tape in reader, as required. |
| 27 | Press LOAD ADD and START. |

Program execution will begin.

# APPENDIX H
# DISK LINKING LOADER

## H.1 INTRODUCTION

The Disk Linking Loader (LLDR) is used to load and execute 8K FORTRAN compiled and 8K SABR assembled user programs when the system configuration includes one or more DECdisks and the Disk Monitor System. Such user programs exist as a main program with several subprograms (including necessary 8K library subprograms), all of which must be on punched paper tape in relocatable binary format. LLDR loads these multiple-part programs in a page-wise relocatable manner, and links all calls to and returns from external subprograms.

The user communicates with LLDR via the keyboard in a simple, straightforward manner; LLDR types *OPT - and the user responds with a one-letter code which causes LLDR to perform one of seven possible functions (operations).

LLDR, unlike the standard 8K Linking Loader (Chapter 5), is entirely keyboard oriented and makes extensive use of the disk. For example:

a. It allows user programs to be loaded over LLDR itself by utilizing temporary disk storage in the Disk Monitor System environment.

b. It provides two levels of program overlaying so that much larger programs can be run. Up to eight files (programs and subprograms) can be loaded into each overlay area. Overlay files are saved on the disk and called into core as needed at program execution time.

c. It provides several utility and convenience features such as storage map listing, a listing of necessary subprograms not present in core, a listing of available (unoccupied) core, and automatic program starting.

d. It includes load-time monitoring via the keyboard rather than the console switches, and several other minor features.

LLDR accepts paper tape input only, from either the low- or high-speed readers, as do both the 8K FORTRAN and 8K SABR systems. However, the user program (during execution) can use both DECdisk and DECtape for input/output.

The operating system (Run-Time Linkage Routines) necessary for execution of 8K FORTRAN and 8K SABR programs is contained within the LLDR program, and its use is entirely automatic.

Two loading techniques are provided: normal loading and overlay loading. In normal loading, each file is loaded into a separate core area where it remains during execution. In overlay loading, several files are sequentially loaded into the same core area and saved on the disk. At execution time, each file is brought from the disk into core when it is needed. LLDR provides two levels of overlay, and each allows up to eight files per overlay level. A normally loaded program may call a program in either overlay level, and a program in either overlay level may call a program in the other level.

The following main stipulations should be remembered when using LLDR.

a. A program in an overlay level may not call another external program in the same overlay level, except as explained in Section H.4.

b. Common storage (i.e., data storage accessible by all programs and subprograms) is always located in field 1.

c. The program or subprogram which requests the largest amount of common storage must be loaded first.

d. No one program or subprogram may be greater than 4K in length.

e. Programs may not be loaded across field boundaries, although they may be loaded into any available field.

f. Overlay files may not be loaded over LLDR, although normal files may be.

LLDR requires a PDP-8/I, -8/L, -8, or -8/S computer with at least 8K words of core, an ASR-33 Teleprinter, and at least one DECdisk. A high-speed paper tape reader is optional but highly recommended. LLDR can use all available core memory and disk storage.

## H.2 LOADING, SAVING, AND STARTING LLDR

LLDR is furnished on punched paper tape in binary-coded format, and is loaded into field 0 by the standard Binary Loader (refer to PDP-8/I System User's Guide, DEC-08-NGCC-D).

Before using LLDR or saving it as a systems program on the disk, it should be properly initialized for the amount of core available and for the type of paper tape reader to be used. LLDR is initially set for a basic configuration of 8K words of core and a high-speed paper tape reader. With any other configuration, LLDR should be started and initialized as explained in Section H.2.2. Complete loading, saving, and calling procedures are given below for both basic and expanded configurations. The following procedures assume that the user is familiar with the Disk Monitor System, and that the system is available for use.

### H.2.1 Basic Configuration

The user with 8K of core and a high-speed reader should use the following procedures.

a. Determine that the Disk Monitor is in memory. (Type CTRL/C* or START at 07600.)

b. When Monitor responds with a dot, call the system loader by typing

.LOAD ↵ (↵ represents typing the RETURN key)

*CTRL/C is typed by holding down the CTRL key while typing the C key.

c. Insert the LLDR binary tape in the high-speed reader.

d. Answer the loading command dialogue as follows:

```
*IN-R: ↲
*
*ST = ↲
↑ <CTRL/P> ↑ <CTRL/P>
```

Keys shown within angle brackets are not echoed on the teleprinter when typed by the user.

After each up-arrow which is typed by the Monitor, the user types CTRL/P by holding down the CTRL key while typing the P key; this is equivalent to pressing the CONT switch when loading manually.

e. LLDR is now loaded into core; save it on the disk by typing

```
.SAVE LLDR!0-6777;200↲
```

f. LLDR may now be called to load relocatable binary programs by typing

```
.LLDR ↲
```

## H.2.2 Expanded Configuration

The user, with any configuration other than the basic configuration mentioned above, should use the following procedure:

a. Determine that the Disk Monitor is in memory. (Type CTRL/C or START at 07600.)

b. When Monitor responds with a dot, call the system loader by typing

```
.LOAD↲
```

c. Insert the LLDR binary tape in the appropriate reader.

d. Answer the loading command dialogue as follows:

```
*IN-R: ↲          (R:  for high-speed reader
*                  T:  for ASR reader)
*ST=7400↲
↑<CTRL/P>  ↑ <CTRL/P>
```

e. LLDR is now loaded into core. It automatically starts at location 7400, causing it to type out its initialization questions. Answer the questions as shown below.

```
*GIVE SIZE OF MEMORY IN K-12 ↲     (user typed 12)
*HIGH SPEED READER? Y              (user typed Y)
```

When answering the first question, the user should type the amount of available core memory after K-; the user should type Y for yes, or N for no in answer to the second.

f. When the above questions have been properly answered, LLDR may be saved on the disk by typing

```
.SAVE LLDR!0-6777;200↲
```

g. LLDR may now be called to load relocatable binary programs by typing

```
.LLDR↲
```

whereas the LLDR system program will be transferred from disk storage into core memory, and automatically started (executed) so that it types out its version number and *OPT-. It then waits for the user to specify which of the seven optional functions is to be performed. The version number and option request might appear as shown below.

PDP-8 DISK␣ LINKING␣ LOADER␣ DEC-08-A2C7-03
*OPT-

LLDR is now in core, started, and ready for use.

## H.3 LLDR Functions

When LLDR has been initialized and started as described in the preceding section, it types its program version number (also found on the paper tape identification label) and option statement and then waits for the user to specify the desired function to be performed. For example:

PDP-8 DEC-08-A2B4-02
*OPT-

The user's response to *OPT- is in the form of a one-letter code followed by the RETURN key. LLDR's functions and corresponding one-letter function codes are listed below.

| Code | Function |
|---|---|
| C | Core availability listing |
| D | Disk file assignment |
| E | Exit with halt |
| L | Normal loading |
| M | Storage map listing |
| O | Overlay loading |
| S | Start main program |
| U | Unloaded program listing |

Functions may be called whenever needed or desired, except that the M, U, and S functions must not be called first.

Upon completion of a function (except E or S), LLDR will request another by repeating the option statement (*OPT-).

Any error made by the user when responding to an option statement will cause LLDR to type a question mark, ignore the response, and repeat the option statement.

LLDR may be stopped (e.g., to make a program patch) and restarted without altering the state of the computer by using the console STOP switch and restarting at 00600. This method may be used at any time after completion of any function other than D, except during overlay loading or while a tape is actually being read.

At any time during the use of LLDR (except while a tape is being read in), control may be returned to the Disk Monitor. This is done by typing CTRL/C; however, when CTRL/C is typed, all data temporarily stored on the disk is lost.

### H.3.1 Disk File Assignment Function (D)

If the user's programs or subprograms create or use disk data files with the RDISK and WDISK library functions, the D function must be the first function used. The D function performs the preliminary job of entering the names of user files into the disk directory. This prepares the way for using the RDISK and WDISK library functions, which allow the user to read and write data on the disk at execution time.

Use of the D-function proceeds as shown below:

```
PDP-8 DEC-08-A2B4-01

*OPT-D ↵
*FILES-ABC, WXYZ, M1, M2, 5H, R, 3, P↵
*OPT-
```

where a directory entry is assigned to each of the eight file names. File names may be from one to four characters in length, and up to ten files may be specified. All such files must be named in one execution of the D function.

The order in which the data files are named for the D function is especially important. The reason for this is that when the user's program references disk data files using the RDISK and WDISK library functions, he must reference these files not by name but by logical number (1,2, ..., 10). This logical number is determined by the order in which he names the files for the D function. For example, if files have been named in the D function as shown in the previous example, the user's program will reference file M1 by statements of the form

CALL RDISK (3, ...)

because M1 was the third file named.

Before using the D function the user should study thoroughly the operation and use of the RDISK and WDISK library functions in Section E-9.

The disk directory will accommodate ten file names. If the directory is too full to accommodate all files named, a meaningful error message is printed by LLDR. In the example above, if the directory had room for only four files, the error message

DISK WILL NOT HOLD 5H & FOLLOWING FILES

would have been printed. If this happens, the entire D function request is ignored and LLDR prints another *OPT- to allow the user to repeat the D function with fewer files or to specify a different function.

After the D function has been performed, LLDR will again print *OPT- for the user to continue with the process of loading his program. After the D function has been used or when a different function has been called, the D function is no longer available--if called a second time or after a different function, it is treated as an illegal function code.

Again, if the D function is to be used, it must be the first function used. If it is not chosen as the first function, it is not available for use until a fresh image of LLDR is brought into core from the disk.

## H.3.2 Loading Functions (L and O)

The two loading functions, L for normal loading and O for overlay loading, are available for use at any time. These are the principal functions of LLDR--to load relocatable programs for execution. These functions use the standard technique of link-loading as described in Chapter 5, which applies specifically to the relocatable binary code (Chapter 3) produced by the 8K FORTRAN/SABR system.

Programs and subprograms may be loaded in any order and into any field. The only restrictions are listed below.

a. The subprogram which requires the largest amount of common storage must be loaded first.

b. No subprogram may be loaded across a core field boundary; i.e., no subprogram may be longer than 4K in length.

c. A maximum of 64 subprograms may be loaded, including multiple entry points for single programs.

LLDR loads subprograms in the order presented and into the field specified (see below) from the lowest available memory upward. Common storage is allocated in the lower portion of field 1 before loading actually starts. A maximum of 3840 words of common storage fills field 1.

LLDR loads in a page-wise relocatable fashion (each program begins at the start of a new core page), establishing external links so that each subprogram is properly executed.

### H.3.2.1 Normal Loading (L)

In normal loading, the user's program is loaded directly into core memory where it remains available for, throughout, and after execution. The core area occupied by each normally loaded program is the property of that program, and no other program can be loaded into its core area.

To perform normal loading, the user responds to *OPT- with the letter L. When this is done, LLDR types a request for the number of the field in which the user wishes to load. This specified field must exist in the configuration. For example:

```
*OPT-L ↵
*FIELD-2↵
```

Had field 2 been nonexistent, the following would have occurred:

```
*OPT-L ↵
*FIELD-2↵
?
*OPT-
```

where LLDR ignored the user's response, typed the question mark, and repeated the option statement.

When LLDR is satisfied with user response, it then types an up-arrow. At this point LLDR will pause and wait for the user to place his relocatable binary tape in the tape reader, and to type CTRL/P which causes LLDR to load the program into core. When the program has been loaded, LLDR will type another up-arrow and pause for user response. If the user wishes to load another program into the same field, he need only place the tape in the reader and then type another CTRL/P (or press the CONTinue switch and then type CTRL/P is using the low-speed tape reader). When the user no longer wishes to load into the same field, he should respond to the up-arrrow by typing the RETURN key, and LLDR will type another option statement.

The user may respond to an up-arrow with CTRL/N, which causes LLDR to by-pass the next program on a multi-program tape. This situation may, for example, occur with a library subprogram tape.

A typical example of normal loading is shown below, where three programs are loaded into field 0 and two into field 1, with one program being by-passed.

```
*OPT-L ↵
*FIELD-0 ↵
*↑<CTRL/P> ↑ <CTRL/P> ↑ <CTRL/P> ↑ ↵
*OPT-L ↵
*FIELD-1 ↵
*↑<CTRL/P> ↑ <CTRL/N> ↑ <CTRL/P> ↑ ↵
*OPT-
```

If the low-speed reader had been used in the example above, the CONTinue switch would have been pressed just before each CTRL/key combination.

### H.3.2.2 Overlay Loading (O)

Overlay loading allows the user to load as many as 16 subprograms into the same core area. The user may load one or two overlay levels (each O function call constitutes an overlay level) of subprograms (files) with up to eight files per level. Overlay loading is possible only when no two subprograms of the same level need to be in core at the same time, i.e., they do do not call each other.

All subprograms loaded during the operation of an O function are loaded into the same core area (overlay level) and automatically saved in separate files on the disk. At execution time each file is called back into core as needed. No protection is given to the file of this overlay level that was previously in core. It is completely overwritten in core. Overlay files should use common storage for data which must remain in core.

Files in a given level may be loaded in any order, provided they are all loaded during the same execution of O function. Files in a given level need not be the same length; enough core is allocated for the largest file in the level.

Loading with the O function is quite similar to loading a string of programs in the same field using the L function. An example is given below, where three files are loaded into the first level and two files into the second level, with one file being passed over.

```
*OPT-O )
*FIELD-1 )
*↑<CTRL/P>↑<CTRL/P>↑<CTRL/P>↑ )
*OPT-O )
*FIELD-1 )
*↑<CTRL/P>↑<CTRL/N>↑<CTRL/P>↑ )
*OPT-
```

Loading of a single overlay level is terminated with the RETURN key. Loading of an overlay level will automatically be terminated after eight files have been loaded.

As with the L function, if the low-speed reader had been used in the example above, the CONTinue switch would have been pressed just before each CTRL/key combination.

When the main program is removed from core, linkage to its overlay files is broken. Therefore, for subsequent execution, files must be reloaded with the main program.

### H.3.2.3 Error Messages

When LLDR detects an error during loading of a program, it types an error message of the following form:

ERROR 000n

where n is a number from 1 to 6, representing the type of error detected. If the error is fatal, control returns to the Disk Monitor. If it is not fatal, the user may be able to continue loading (see below).

| Error No. | Error | Fatal? |
|---|---|---|
| 1 | Attempt to load more than 64 subprograms | Yes |
| 2 | Field overflow | No |
| 3 | Subprogram with largest common assignment not loaded first | Yes |

| Error No. | Error | Fatal ? |
|---|---|---|
| 4 | Checksum error | No |
| 5 | Improper or damaged tape or reader error | No |
| 6 | Disk overflow | No |

A discussion of each non-fatal error is given below.

Error 2 - During normal loading, loading may be continued in a different memory field. During overlay loading, the entire overlay level must be reloaded into a different memory field.

Errors 4 and 5 - During either type of loading, the user may reposition the faulty tape in the reader and type CTRL/P in response to the new up-arrow. If the error persists, reassembly or hardware maintenance will be necessary.

Error 6 - Occurs during normal loading only when the user is loading into the upper portion of field 0; the program which caused the error must be loaded into a different field. During overlay loading, the current overlay level will be closed with only the files that were loaded successfully. The file which caused the overflow (the last file read) and succeeding files will have to be loaded normally.

## H.3.3 Utility Functions (C, M, and U)

### H.3.3.1 Core Availability (C)

The user may at any time request a list of the number of pages available for loading in each core field. The following example assumes that the user has a 16K computer (4 fields):

```
*OPT-C ↵
0033
0036
0036
0036
*OPT-
```

The numbers listed are the octal number of free pages left in fields 0, 1, 2, and 3, respectively.

### H.3.3.2 Storage Map (M)

During the link-loading process, LLDR builds a list of external symbols; i.e., main program and subprogram entry points and their actual starting addresses. This list forms a complete storage map of all programs loaded, as shown below:

```
*OPT-M
MAIN          10200
READ          01055
WRITE         01066
IOH           03031
SETERR        00000   U
TTYIN         00000   U
  .
  .
  .
FLOAT         05046
FIX           04513
*OPT-
```

Starting addresses are expressed in five octal digits - the first digit represents the memory field and the other four the address in that field. The U means that the stated subprogram has been called but has not been loaded, and therefore must be loaded before successful execution is possible.

Listing of the storage map may be prematurely terminated by typing CTRL/P.

### H.3.3.3 Unloaded Program Listing (U)

This function is used to obtain a list of those subprograms which must still be loaded before successful execution is possible. All symbols flagged with a U in a storage map listing will be listed as shown below:

```
*OPT-U
SETERR
TTYIN
TTYOUT
HSIN
HSOUT
*OPT-
```

This listing may also be prematurely terminated by typing CTRL/P.

### H.3.4 Exit Functions (E and S)

The E function is used to cause a halt after all loading is complete. The S function is used to automatically start execution of the loaded program at the beginning of the main program.

Both of these functions signal LLDR that loading is complete. They each cause any data which has been temporarily saved on the disk (except overlay files) to be read into core.

When the E function is used, LLDR reads in all data temporarily stored on the disk and then halts. The user's entire program (except overlay files) will be in core, ready for patching, execution, or saving on the disk.

When the S function is used, LLDR checks for a subprogram called MAIN (such as a FORTRAN main program). If found, execution will automatically start at the starting address of MAIN. If MAIN is not found, the S function is executed as an E function.

## H.4 TECHNIQUES, OVERLAY LOADING

In general, any group of subprograms which do not call each other (either directly or indirectly) may be loaded into the same overlay level. A typical situation follows:

| | | |
|---|---|---|
| MAIN | contains calls to | A, B, C, D, E |
| A | contains calls to | D, E, F |
| B | contains calls to | D, G |
| C | contains calls to | D, E, H |
| D | contains calls to | E |
| E,F,G,H | contain no external calls | |

The above combination may be loaded as follows:

| Normal | Overlay 0 | Overlay 1 |
|---|---|---|
| MAIN | A | D |
| E | B | F |
| | | G |
| | | H |

If D contains a call to any other than E, it would be better to load D normally and put E in overlay 1. If F were to call B, the above loading situation would not work; A would be calling B indirectly, and these two are in the same overlay level.

It is possible, however, to call another program in the same overlay level only if the called program never attempts to return to the calling program. In this way, simple chaining may be achieved. For example, a very long FORTRAN main program can be split into sections with each section terminated by a call to the next. Such a situation is shown below.

| | |
|---|---|
| MAIN | calls A, B, C and is terminated by a call to MAIN2 |
| MAIN2 | calls A, B, C and is terminated by a call to MAIN3 |
| MAIN3 | calls A, B, C and is terminated by a call to MAIN4 |
| ⋮ | |
| MAIN8 | calls A, B, C and stops |
| A, B,C | contain no external calls |

The above combination may be loaded as follows:

| Overlay 0 | Overlay 1 |
|---|---|
| MAIN | A |
| MAIN2 | B |
| MAIN3 | C |
| ⋮ | |
| MAIN8 | |

When the MAIN program is contained in an overlay area, the E function cannot be used unless MAIN is loaded last into the overlay level. The S function will work with the above combination since it works regardless of the order in which the segments of MAIN are loaded.

With FORTRAN programming alone, a subprogram other than a MAIN program may not be chained. However, this is possible with careful assembly language programming. An example of such programming is shown below, where SUB is split into a two-part chain, SUB and SUB2. MAIN is a standard FORTRAN program containing calls to SUB in the form:

```
CALL SUB (A1, A2, A3)
```

SUB is written as a standard FORTRAN program which does part of the work for the entire subroutine chain, including processing arguments A1 and A2. It is written with two arguments and concludes with

```
CALL SUB2(Z
```

where Z is any dummy argument. After SUB has been compiled and before the intermediate compiler symbolic is assembled, it should be edited to include the insertions enclosed in brackets.

```
[X,     COMMN2]
        ENTRY SUB
SUB,    BLOCK 2
          .
          .
          .
      [ TAD   SUB         /SAVE RETURN FIELD              ]
      [ DCA   X                                           ]
      [ TAD   (-2         /-2* NO. OF ARGS TO BE PASSED   ]
      [ TAD   SUB#        /SAVE ARGUMENT ADDRESS          ]
      [ DCA   X#                                          ]
        CALL  1,SUB2
        ARG   Z
        END
```

SUB2 is also a standard FORTRAN program containing the latter portion of the entire subroutine, including the processing of argument A3. The actual contents of SUB2 is coded in FORTRAN just as if it were a subroutine taking one argument. After SUB2 has been compiled, the compiler symbolic output is edited as shown below:

```
[X,     COMMN 2]
        ENTRY SUB2
SUB2,   BLOCK 2
      [ TAD      X              /REPLACE ARG POINTER      ]
      | DCA      SUB2                                     |
      | TAD      X#                                       ]
      [ DCA      SUB2#
          .                     /CONTINUE WITH NORMAL FORTRAN
          .                     /CODE, CONCLUDING WITH
        RETRN    SUB2
        END
```

## H.5 USER PROGRAM EXECUTION

If the user chooses not to execute his program automatically with the S function, he may determine the exact address for the start (using the storage map or assembly listing), and execute his program, using the console switches or the Disk Monitor.

At execution time, the Run-Time Linkage Routines (see Section 3.3.4) must be in core. These routines accomplish the necessary linkage for all calls to and returns from external subprograms, all off-page indirect references, and all off-field references (including those to common and passing subroutine arguments).

If, during execution of a user program, a call is made to a nonexistent program or subprogram, an unconditional halt will occur and control will return to the Disk Monitor. This error is fatal. All other execution-time errors are covered in Section C-3.

Program execution may be terminated at any time by typing CTRL/C. However, when CTRL/C is typed, all overlay files stored on the disk are lost.

## H.6 STORAGE ALLOCATION

The following core availability map allows the user to plan his loading.

| Field 0 | |
|---|---|
| 0000-0777 | Used by the Run-Time Linkage Routines and not available to the user for loading. |
| 1000-4377 | Available for any loading. |
| 4000-7577 | Residence of LLDR during loading. Available for normal loading (by automatic use of temporary disk storage), but not available for overlay loading. |
| 7600-7777 | Disk Monitor permanent residence. |

INDEX

## HOW TO OBTAIN SOFTWARE INFORMATION

Announcements for new and revised software, as well as programming notes, software problems, and documentation corrections are published by Software Information Service in the following newsletters.

Digital Software News for the PDP-8 Family
Digital Software News for the PDP-9/15 Family
PDP-6/PDP-10 Software Bulletin

These newsletters contain information applicable to software available from Digital's Program Library.

Please complete the card below to place your name on the newsletter mailing list.

Questions or problems concerning DEC Software should be reported to the Software Specialist at your nearest DEC regional or district sales office. In cases where no Software Specialist is available, please send a Software Trouble Report form with details of the problem to:

Software Information Service
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

These forms, which are available without charge from the Program Library, should be fully filled out and accompanied by teletype output as well as listings or tapes of the user program to facilitate a complete investigation. An answer will be sent to the individual and appropriate topics of general interest will be printed in the newsletter.

New and revised software and manuals, Software Trouble Report forms, and cumulative Software Manual Updates are available from the Program Library. When ordering, include the document number and a brief description of the program or manual requested. Revisions of programs and documents will be announced in the newsletters and a price list will be included twice yearly. Direct all inquiries and requests to:

Program Library
Digital Equipment Corporation
146 Main Street, Bldg. 3-5
Maynard, Massachusetts 01754

Digital Equipment Computer Users Society (DECUS) maintains a user Library and publishes a catalog of programs as well as the DECUSCOPE magazine for its members and non-members who request it. For further information please write to:

DECUS
Digital Equipment Corporation
146 Main Street
Maynard, Massachusetts 01754

---

Send Digital's software newsletters to:

Name ______________________________
Company Name ________________________
Address ____________________________
____________________________
(zip code)

My computer is a

| | | |
|---|---|---|
| PDP-8/I ☐ | PDP-8/L ☐ | |
| LINC-8 ☐ | PDP-12 ☐ | |
| PDP-9 ☐ | PDP-15 ☐ | Please specify |
| PDP-10 ☐ | OTHER ☐ | ____________ |

My system serial number is ____________________ (if known)

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

**Digital Equipment Corporation**
**Software Quality Control**
**Building 12**
**146 Main Street**
**Maynard, Mass. 01754**

8K SABR

DEC-08-ARXB-D

## READER'S COMMENTS

Digital Equipment Corporation maintains a continuous effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback – your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability.

Did you find errors in this manual?

How can this manual be improved?

DEC also strives to keep its customers informed of current DEC software and publications. Thus, the following periodically distributed publications are available upon request. Please check the appropriate boxes for a current issue of the publication(s) desired.

- ☐ Software Manual Update, a quarterly collection of revisions to current software manuals.
- ☐ User's Bookshelf, a bibliography of current software manuals.
- ☐ Program Library Price List, a list of currently available software programs and manuals.

Please describe your position.

Name ______ Organization ______

Street ______ Department ______

City ______ State ______ Zip or Country ______

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

**Digital Equipment Corporation**
**Software Quality Control**
**Building 12**
**146 Main Street**
**Maynard, Mass. 01754**

Digital Equipment Corporation
Maynard, Massachusetts

printed in U.S.A.