WRITER'S MASTER COPY

SDS

SCIENTIFIC DATA SYSTEMS

Reference Manual

DO NOT REMOVE

# SDS 940 TAP

# TAP DIRECTIVES

| | | | |
|---|---|---|---|
| [[$]label] | ASC | {'text'<br>expression, text} | [comment] (RET) |
| [[$]symbol] | BES | expression | [comment] (RET) |
| [[$]symbol] | BSS | expression | [comment] (RET) |
| [[$]label] | COPY | symbol$_1$[, . . . , symbol$_n$] | [comment] (RET) |
| [label] | CRPT | expression[, (s=e$_1$, [e$_2$,] e$_3$)[. . .]] | [comment] (RET) |
| [[$]label] | DATA | expression$_1$[, . . . , expression$_n$] | [comment] (RET) |
| | DEC | | [comment] (RET) |
| | DELSYM | | [comment] (RET) |
| | ELSE | expression | [comment] (RET) |
| | ELSF | expression | [comment] (RET) |
| | END | [expression] | [comment] (RET) |
| | ENDF | | [comment] (RET) |
| | ENDM | | [comment] (RET) |
| | ENDR | | [comment] (RET) |
| [$]symbol | EQU | expression | [comment] (RET) |
| symbol | EXT | [expression] | [comment] (RET) |
| | FREEZE | | [comment] (RET) |
| | FRGT | symbol$_1$[, . . . , symbol$_n$] | [comment] (RET) |
| symbol | IDENT | | [comment] (RET) |
| [label] | IF | expression | [comment] (RET) |
| | LIST | symbol$_1$[, . . . , symbol$_n$] | [comment] (RET) |
| name | MACRO | [P$_1$, P$_2$, P$_3$] | [comment] (RET) |
| [$]symbol | NARG | | [comment] (RET) |
| [$]symbol | NCHR | character string | [comment] (RET) |
| | NOEXT | | [comment] (RET) |
| | NOLIST | symbol$_1$[, . . . , symbol$_n$] | [comment] (RET) |
| | OCT | | [comment] (RET) |
| symbol | OPD | expression, class[, ar] [, type] [, sb] | [comment] (RET) |
| | ORG | expression | [comment] (RET) |
| | PAGE | | [comment] (RET) |
| | POPD | expression, class[, ar] [, type] [, sb] | [comment] (RET) |
| | RAD | expression | [comment] (RET) |
| | RELORG | expression | [comment] (RET) |
| | REM | remark | |
| | RETREL | | [comment] (RET) |
| [label] | RPT | {expression<br>(s=e$_1$, [e$_2$] e$_3$) . . . (s=e$_1$[, e$_2$])} | [comment] (RET) |
| [[$] label] | TEXT | {'text'<br>expression, text} | [comment] (RET) |

Price: $1.50

ERRORS
ii
iii
iv
5
15
17
29

# TAP REFERENCE MANUAL

for

SDS 940 TIME-SHARING COMPUTER SYSTEM

Preliminary Edition

90 11 17B

November 1968

SDS

SCIENTIFIC DATA SYSTEMS/701 South Aviation Blvd./El Segundo, California 90245

## REVISION

This publication, SDS 90 11 17B, is a revision of the SDS 940 TAP Reference Manual, 90 11 17A (dated October 1967). A change in the text from that of the previous manual is indicated by a vertical line in the margin of the page.

## RELATED PUBLICATIONS

| Title | Publication No. |
|---|---|
| SDS 940 Computer Reference Manual | 90 06 40 |
| SDS 940 FORTRAN II Reference Manual | 90 11 10 |
| SDS 940 BASIC Reference Manual | 90 11 11 |
| SDS 940 QED Reference Manual | 90 11 12 |
| SDS 940 DDT Reference Manual | 90 11 13 |
| SDS 940 CAL Reference Manual | 90 11 14 |
| SDS 940 FORTRAN IV Reference Manual | 90 11 15 |
| SDS 940 Time-Sharing System Technical Manual | 90 11 16 |
| SDS 940 Terminal User's Guide | 90 11 18 |

[illegible]

NOTICE

# CONTENTS

## APPENDIXES

## ILLUSTRATIONS

## TABLES

# 1. INTRODUCTION

The SDS 940 Time-Sharing Assembly Program (TAP) includes most features found in other assemblers. Input lines are divided into label, operation, operand, and comments fields. Instructions and addresses can be represented as symbols and used in expressions to form values at assembly time. In addition, other more powerful features help to alleviate the tedious housekeeping chores usually considered a part of machine language programming.

While some assemblers have a fixed-field format, TAP allows a free-form format in which the fields are delimited by the appearance of one or more blanks. In the examples to follow, blanks are represented by the symbol ƀ. For example, the following lines are identical:

```
LOC1ƀƀƀƀLDAƀƀƀƀVALUE2  ALMOST ANY (RET)
+THING GOES(RET)(+specifies continuation)(RET)

LOC1ƀLDAƀVALUE2ƀALMOST ANYTHING GOES (RET)
```

There must be at least one leading blank if no label appears. For example, the following lines are identical:

```
ƀƀƀƀƀSTAƀƀƀTEM4 (RET)

ƀSTAƀTEMP4 (RET)
```

Symbols in most assemblers are used to represent memory addresses or to identify constants and are usually placed in a symbol table as partial word entries. However, in TAP, the values of symbols are held in the computer's full word length (24 bits) as positive or negative values. The assembly-time calculations that can be performed using symbolic expressions are, therefore, more general and much more useful. As examples, one can write:

```
ABC     EQU  -17 (RET)
        DATA  2*ABC+15 (RET) (The integer -19 will
                              be formed)
```

New operation codes can be defined, and these, as well as existing operation codes, can be redefined during assembly. It is possible to specify whether (1) an operation code requires an operand, (2) the operand is truncated to 9 or 14 bits, or (3) the instruction is a SYSPOP (system programmed operator), requiring bit 0 to be set.

Expressions can be written as follows:

```
AB-BC+3, 2 (The ", 2" is the tag for indexing)

A+C-C*37B/10 (Octal numbers are terminated with "B")

A(AND)(NOT)B(OR)CDEF(EOR)777000B
```

The following lines generate the value 0 or 1:

```
DATA    XYZ=ABC (RET)

DATA    AB*6<CD+2 (RET)
```

or alternatively (if keypunch characters are used):

```
DATA    XYZ(EQU)ABC (RET)
DATA    AB*6(LSS)CD+2 (RET)
```

Literals are provided so that constants can be referred to in programs by value, rather than by location. This feature makes a program easier to read and relieves the programmer from remembering to include all the data.

```
LDA     =5 (RET)
LDB     =ABC+77770000B(AND)CD(RET)
LDX     =AB=@(RET)(The value of the literal is 1 or 0)
BRU*    =DEF (RET)
```

Note that the literal can be any general expression.

The assembler's most powerful feature is its macro facility. On the simplest level, a macro can be thought of as an abbreviation or shorthand notation for a body of code that is used repetitively in a program. For example, the 940 has an SKE (Skip if Equal) instruction but not an SKNE (Skip if Not Equal) instruction. Frequently it is convenient to do SKNE, but one has to write

```
START   SKE  ='ABC'(RET)(Skip if equal to 'ABC')
        BRU  *+2(RET)(This is necessary to invert SKE)
        BRU  EQULOC(RET)(In case (A)='ABC')
```

Accordingly, one can define the macro SKNE as follows:

```
SKNE    MACRO   DUMMY(RET)
        SKE     DUMMY (1)(RET)
        BRU     *+2 (RET)
        ENDM (RET)
```

Then, by using the lines

```
        SKNE    ='ABC' (RET)
        BRU     EQULOC (RET)
```

the code shown at location START above will be generated.

A macro reference in a body of a program is a directive to the assembler to insert a predefined block of in-put language while replacing dummy substrings with argument substrings. The block can contain other macros.

The real power of the macro lies in its ability to substitute character strings for values. Further, the macro offers a genuine facility for doing calculations at assembly time, thus providing for program parameterization. Data areas can not only be redimensioned by changing parameter values, but different programs can also be produced. The IF, ELSF, RPT (repeat), and CRPT (conditional repeat) directives provide these calculation facilities. These directives function independently of macros, but their use with the macros enables programmers to easily perform extremely complex operations.

TAP statements can be prepared in several ways. Cards may be punched at any facility for entry into the system at the computer site. Alternatively, the 940 text editor, QED[†], provides another and much easier way of preparing the symbolic input for the assembler on-line at the user's teletype. The text editor is also used to modify existing source programs stored in the system.

When a source error appears during assembly or execution, the user can revert to the text editor in a matter of seconds. Laborious input/output operations are avoided because both the symbolic and object code are kept in secondary, random-access storage. Large programs can be edited on-line and reassembled in a fraction of the time required at a key punch, because they remain within the system on secondary storage.

The on-line debugging subsystem, DDT[††], allows the user to insert breakpoint statements, execute single statements, and examine and change variables symbolically. Significantly, only a minute fraction of the computer's resources is employed while the user is thinking on-line. The Executive dismisses programs awaiting input and moves them to secondary storage. When the user makes his next move, the program resumes operation within a second or two.

To TAP users, the 940 time-sharing software system has the structure shown in Figure 1. All symbolic files accessible to the user can be accessed with the Executive system, the command modes of the text editor (QED), the debugging subsystem (DDT), and the assembler (TAP).

A user can have access to one subsystem (QED, TAP, DDT, etc.) at a time; calling a second one releases the first. In particular, the contents of the QED text buffers are lost when TAP is called. Therefore, files are used to communicate data between subsystems. For a complete description of the 940 files and their use, refer to the SDS 940 Terminal User's Guide.

[†]The text editor is described in the SDS 940 QED Reference Manual.

[††]The on-line debugging system is described in the SDS 940 DDT Reference Manual.

## TYPOGRAPHIC CONVENTIONS

For clarity, several typographic conventions have been used throughout this manual. These are explained below.

1. Underscored copy in an example represents copy produced by the subsystem in control of the computer. Unless otherwise indicated, copy not underscored in an example must be typed by the user.

2. The Ⓡ notation appearing after some lines in the examples indicates a carriage return. The carriage return key is labeled RETURN on the teletype keyboard. The user must depress the carriage return key after each command to inform the computer that the current command is terminated and a new one will begin. The computer then upspaces the paper automatically.

3. The Ⓛ notation indicates the LINE FEED key.

4. Non-printing control characters are represented in this manual by an alphabetic character and a superscript c (e.g., $D^c$). The user depresses the alphabetic key and the Control (CTRL) key simultaneously to obtain a non-printing character. For editing purposes some control characters will cause a symbol to be printed, but this symbol does not appear in the final version of an edited line.

## OPERATING PROCEDURES

The standard procedure for gaining access to an SDS 940 time-sharing computer, from a teletype terminal, is described in the SDS 940 Terminal User's Guide. The publication also includes information concerning the Executive system and the calling of various subsystems available to the terminal user. The following paragraphs summarize the standard procedures as they apply to TAP users.

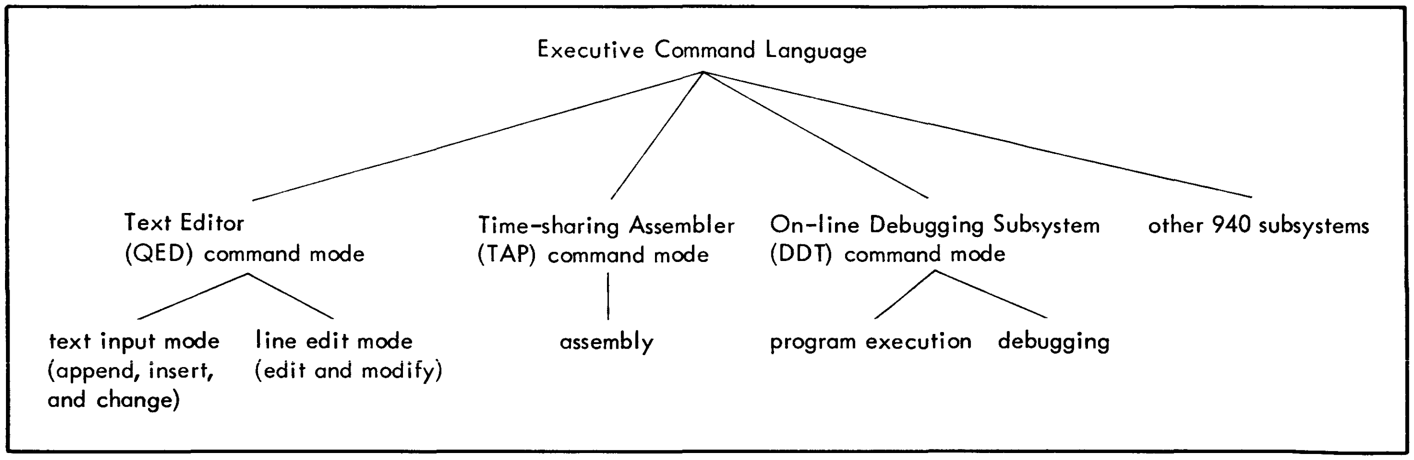

Executive Command Language
Text Editor (QED) command mode
Time-sharing Assembler (TAP) command mode
On-line Debugging Subsystem (DDT) command mode
other 940 subsystems
text input mode (append, insert, and change)
line edit mode (edit and modify)
assembly
program execution
debugging

Figure 1. SDS 940 Time-Sharing Assembly Structure

## LOG-IN

To gain access to the computer, the following operating sequence is performed:

1. If the FD-HD (Full Duplex-Half Duplex) switch is present, turn the switch to FD. When the teletype is not connected to the computer (sometimes called the Local Mode), this switch must be in the HD position.
2. Press the ORIG (originate) key, which is located at the lower right corner of the console directly under the telephone dial. This key is depressed to obtain a dial tone before dialing the computer center.
3. Dial the computer center number. When the computer accepts the call, the ringing will change to a high-pitched tone. A request that the user log in will appear on the teletype:

   ```
   PLEASE LOG IN:
   ```

4. The user must then type his account number, password, name and, optionally a project code, in the following format:

   ```
   number password;name;project code (RET)
   ```

   Only persons who know all three elements (the account number, password, and name) may log in under that particular combination. The following examples all illustrate acceptable practice.

   ```
   PLEASE LOG IN:    C2PASS;JONES;REPUB (RET)
   PLEASE LOG IN:    D1WORD;BROWN;DEMO (RET)
   PLEASE LOG IN:    E1PW;PSEUDO; (RET)
   ```

   The optional 1-12 character project code is provided for installations that have several programmers using the same account number and user name. The project code is not checked for validity.

   If the user does not correctly type his account number, password, and name within a minute and a half, a message is transmitted instructing him to call the computer center for assistance. The computer will then disconnect the user and the dial and log-in procedure will have to be repeated.

5. If the account number, password (nonprinting), and name are accepted by the computer, it will print READY, the date, and the time, on the next line.

   ```
   READY date, time
   -
   ```

   The dash indicates that the 940 Executive is ready to accept a command.[†]

6. In response to the dash, the user types

   ```
   QED (RET)
   ```

   to call the text editor, or

   ```
   TAP (RET)
   ```

   to call the assembler, or

   ```
   DDT (RET)
   ```

   to call the on-line debugging subsystem.

[†] In some 940 time-sharing systems the commercial at sign, @, is used to indicate that the 940 Executive is ready to accept a command.

## ESCAPE

The ESCAPE (ESC) key[†] may be used at any time. It causes the subsystem in control to abort the current operation and ask for a new command. Striking the (ESC) key before terminating a command with (RET) aborts the command.

## EXIT AND CONTINUE

Striking the (ESC) key several times in succession causes computer control to return to the Executive. If the user wants to re-enter a subsystem without losing his program and if he has not subsequently called any other subsystem, he may type CONTINUE in response to the dash. This will return him to the previous subsystem so that he may resume his work.

## LOG-OUT

When the user wishes to be disconnected from the computer, he types several (ESC)'s (to return control to the Executive) and then types

```
-LOGOUT (RET)
TIME USED:    hours:minutes:seconds
```

The computer will respond by printing the amount of hook-up (line) time charged to the user's account since the previous log-in procedure was completed.

## SAMPLE ON-LINE SESSION

The following is typical of a session at a teletype terminal. The subsystem the user is communicating with usually identifies itself by typing a special character at the beginning of the current line. The characters and the systems they identify are:

```
- Executive
*QED
```

1. Log in, as described above.
2. Enter the text editor by typing its name following the - symbol.

   ```
   -QED (RET)
   ```

[†] In some 940 time-sharing system configurations the RUBOUT or ALT MODE key is used instead of the ESCAPE key. Where (ESC) appears in this manual, RUBOUT or ALT MODE may be substituted.

You are now in QED, which types an asterisk, "*", to indicate readiness for commands.

If you have a file of TAP statements from an earlier session, read it into the main QED text buffer; otherwise, create a new program. The QED manual is explicit about the commands that are available.

3. When the program looks ready, write out the main QED text buffer onto a symbolic file, where the assembler can read it. For example:

```
*WRITE/SOURCE1/(RET)

NEW FILE (RET)
```

4. Two (ESC)'s will return you to the Executive, where the assembler is called with the command

```
-TAP (RET)
```

The assembler responds on the next line with

```
INPUT:
```

5. Specify a source language file, followed by a binary output file, and (optionally) a listing file, error message file, and listing mode. For example, if you wish to assemble from file SOURCE1, place the binary output in file BIN1, and output the listing at the teletype:

```
*INPUT: /SOURCE/(RET)

+BINARY: /BIN1/(RET)

    NEW FILE (RET)(RET)

+TEXT OUTPUT: TELETYPE

+ASSEMBLE
```

Immediately after the carriage return is typed, assembly takes place. At the conclusion of the assembly, TAP prints

```
number CELLS USED BY PROGRAM
```

and other information, depending on the list options specified by the LIST and NOLIST directives.

6. Return to the Executive with two (ESC)'s and call the on-line debugging subsystem with the command

```
-DDT (RET)
```

7. Specify the file where the object code can be found. For example:

```
;T/BIN1/(RET)
```

Loading begins with location $240_8$ in this sample. After loading is completed, DDT responds on the next line with the octal address of the first location following the loaded program.

8. Your program is now ready for execution, which is started with the command

   240;G (RET) (begin execution at location $240_8$)

9. You may return to the Executive at any time by typing several successive (ESC)'s, and then come back to your program with the command

```
-CONTINUE (RET)
```

Then, continue execution with the command

   ;P (RET) (resume execution with the next instruction in line for execution)

10. When finished with this program, you may want to call QED or TAP to release the debugging subsystem.

# 2. ASSEMBLER CODING RULES

## LANGUAGE ELEMENTS

Input to the assembler consists of a sequence of characters that are combined to form assembly language elements. These language elements, which include symbols, constants, expressions, and literals, comprise program statements which in turn comprise a source program.

### CHARACTER SET

The TAP character set is shown in Figure 2.

| | |
|---|---|
| Alphabetic: | A through Z |
| Numeric: | 0 through 9 |
| Special Characters: | blank |
| | + plus sign |
| | - minus sign |
| | * asterisk |
| | / slash |
| | , comma |
| | ' single quotation mark |
| | ( left parenthesis |
| | ) right parenthesis |
| | = equals sign |
| | . period or radix point |
| | < less than symbol |
| | > greater than symbol |
| | $ currency symbol |
| | ← left arrow |
| | : colon |
| | ; semicolon |
| | ? question mark |
| | [ left bracket |
| | ] right bracket |
| | " double quotation mark |

Figure 2. TAP Character Set

The following characters, normally on standard teletype keyboards, are not recognized by the assembler:

! # % & @ \ ↑

These non-recognized characters will be replaced by blank characters whenever they appear in a TAP statement.

### SYMBOLS

Numbers may be symbolically represented in assembly language by <u>symbols</u>.

A symbol is any string of alphanumeric characters not forming a constant. In particular, it is not necessary that a symbol begin with an alphabetic character. Although symbols may be of arbitrary length, only the first six characters of a symbol are used to distinguish it from other symbols. When a symbol is used to represent a memory address, it is called a <u>label</u>. Examples of symbols are:

START S1C 3D12 CALCULATE 217AB

Special characters must not be used in forming symbols.

### CONSTANTS

A constant is a self-defining language element. Its value is inherent in the constant itself, and it is assembled as part of the statement in which it appears.

Three types of constants are permitted in TAP:

1. decimal integers: one or more decimal digits optionally terminated with the letter D.

   2129, 600D, -217

2. octal integers: one or more octal digits optionally terminated with the letter B and, optionally, a single-digit octal scaling factor.

   217, 32B, 4B3 (which is the same as $4000_8$)

3. string: '1-4 characters (except ')'

All constants are absolute; i.e., their relocation value is 0.

The assembler normally expects integers to be decimal. This can be changed, however, by using a directive (OCT). In any case, integers terminated with B or D override the normal interpretation of integers. String constants are normally not useful in the direct computation of memory addresses, but exist basically to be used in literals (literals are described later in this chapter).

### EXPRESSIONS

An expression is an assembly language element that represents a value. It consists of a single constant or symbol or a combination of constants and symbols separated by binary operators. Examples of expressions are:

100-2*ABE(OR)DEF/27B

22

C12>D19

#### OPERATORS AND EXPRESSION EVALUATION

The operators recognized by the assembler (and their precedence) are given below. Operators of highest precedence are applied first in the evaluation of expressions.

| Operator | Function[†] | Precedence |
|---|---|---|
| + | unary plus | 4 |
| - | unary negation | 4 |
| (NOT) | unary logical inverse | 4 |
| (R) | unary relocation | 4 |
| (LSS) or < | less than | 3 |
| (GTR) or > | greater than | 3 |
| (EQU) or = | equal to | 3 |
| * | multiplication | 2 |
| / | division | 2 |
| (AND) | logical product | 2 |
| + | addition | 1 |
| - | subtraction | 1 |
| (OR) | logical inclusive sum | 1 |
| (EOR) | logical exclusive sum | 1 |

Note that some operators are more than one character long. These are enclosed in parentheses to avoid conflict with symbols that would otherwise look the same. Parentheses are therefore not allowed in expressions to delineate terms or to modify the order of evaluation.

The relational operators (< > =) produce a value 1 if the relation is true, or 0 if false. There can be only one relational operator in an expression.

The assembler evaluates expressions as 24-bit, signed integers. Expressions are evaluated from left to right, using operators of decreasing precedence. For example, if

A = 100, B = 200, and C = -1, then

A+B*C/A = 98

Again, A = $54321_8$, B = $44444_8$, and C = $00077_8$,

then

A(OR)B(AND)C = $54365_8$

As an expression is evaluated, a parallel calculation of its relocation value (R) is made. Only absolute expressions (R = 0) and relocatable expressions (R = 1) are legal.

## CONSTRAINTS ON RELOCATABILITY OF EXPRESSIONS

The assembler forces the following constraints on the use of expressions:

1. No relocatable term (R = 1) may occur in conjunction with the arithmetic operators * or/. In other words, no relocatable symbol may multiply, be multiplied by, divide, or be divided by anything.

2. In the absence of the special relocation operator (see below), the final relocation value of an expression may be only 0 or 1.

3. If the special relocation operator (R) appears in an expression, the relocation value of the expression may be either 0 or some relocation (such as the value K, where K is the special relocation radix). DDT is informed by the assembler that special relocation is being used in this case. DDT will then multiply the base address by K before adding it to the value of the expression.

## SPECIAL RELOCATION

The special relocation feature permits the programmer limited use of expressions that are not absolute or singly relocatable. For example, consider the process of assembling and loading a relocatable program. Let the symbol A have value 3. If one writes

```
LDA     A (RET)
```

the assembler produces the computer instruction

```
07600003B
```

and marks the instruction's address as being relocatable. Later, when told to load the program beginning at base address 10000B, DDT will form

```
07610003B
```

Thus, no matter where the program is loaded, the memory reference will be to the third word from the base address.

Now, if the user writes

```
LDA     2*A (RET)
```

The assembler, of course, can form

```
07600006B
```

and presumably what DDT should form is

```
07620006B
```

To do this, DDT must be told that 10000B is to be multiplied specifically by 2. However, only one bit is reserved for such information in the assembler's binary output; this fact accounts for the restriction that expressions may have only the relocation values 0 and 1. This restriction can be circumvented by the use of the special relocation operator (R).

Programs may make use of the string-handling System Programmed Operators (SYSPOPs). These SYSPOPs use string pointers, which contain character addresses (characters are packed three per word). A character address consists of a memory address multiplied by 3 (plus 0, 1, or 2, depending on the position of the character in the word). Thus, if a character address is divided by 3, the quotient is the word address and the remainder designates the character's position in the word.

To form a character address at assembly time, one must be able to multiply a word address (a relocatable item) by a

[†]All operators are binary (i.e., require a preceding and a following term) except for the four specifically designated as unary.

constant (in this case, 3). Thus, if A = 3, the statement

```
    DATA    (R)A+1 (RET)
```

will produce the value

```
    00000012B    (3*A+1)
```

together with a notation to DDT that special relocation applies to that value. Later, when told to load the program beginning at base (B) address 10000B, DDT will form the value

```
    00030012B  (3*A+1)+3*B = 3*(A+B)+1
```

In this way, a symbol, representing a relocatable word address, may be used to form character addresses in string pointers.

It should be noted that 3 was the multiplicative constant associated with (R) in the example above because of the nature of string pointers. This constant is called the special relocation radix. It need not always be 3. In fact, it may be changed to any value by the RAD directive. Because of the relative importance of string pointers, however, this assembler is initialized with the value of (R) set to 3. Therefore, it is unnecessary to use RAD to set (R) to 3 (unless it has been changed to some other value).

### LITERALS

Often, data is placed in a program at assembly time. It is frequently convenient to refer to constants by value rather than label. A literal is a symbolic reference to a datum by value. The assembler allows any expression to be used as a literal, which has the form

```
    = expression
```

Some examples of literals are:

```
    =5  =3*XYZ-2  ='END'  =EXTERN
```

Programmers frequently write such items as

```
    LDA     FIVE (RET)
```

where FIVE is the name of the location containing the value 5. The programmer must remember to define the symbol FIVE somewhere in his program. This can be avoided by the use of a literal. For example,

```
    LDA     =5 (RET)
```

will automatically produce a location containing the correct constant in the program.

When a literal is encountered, the assembler first evaluates the expression and looks up its value in a table of literals (constructed for each subprogram). If the value is not found in the table, it is placed there. In either case, the literal is replaced by a reference to the location of its value in the literal table. At the end of assembly the literal table is included as a part of the object module for the program.

The following are examples of literals:

```
    =10   =4B6   =ABC*20-DEF/12   ='HELP'
```

=2=AB   (This is a conditional literal. Its value will be 1 or 0 depending on whether 2=AB is true or false at assembly time.)

It is important to note that the literal table immediately follows the program when the program is loaded.

## SYNTAX

Assembly language elements may be combined with machine instructions and assembler directives to form statements.

### STATEMENTS

Character input to the assembler is arranged into a sequence of statements called instructions, directives, or comments.

Instructions are symbolic representations of computer instructions that are translated by the assembler into the computer's internal language. Directives, by contrast, are messages that serve to control the assembly process or to create data. Comments are ignored by the assembler but are included in the program listing, serving only to document the meaning of a program.

Statements are logical units of input. They may be delimited either by being placed on separate lines (i.e., by being separated with carriage returns) or by being separated with semicolons.[†]

Examples of statements are:

```
    START   LDA   DAT21 (RET)
            MUL   21B (RET)
            STA   ANS (RET)
```

or

```
    START  LDA  DAT21;  MUL  21B;  STA  ANS(RET)
```

If a statement requires more than one line, it can be continued on the next line by typing a "+" as the first character of the next line, as follows:

```
    START LDA DAT21; MUL 21B; STA ANS(RET)
    +THE COMMENT ON THIS LINE REQUIRES(RET)
    +CONTINUATION(RET)
```

Consecutive continuation may occur for about five lines (320 characters).

[†]Semicolons do not serve as statement delimiters when used between single quotes (as in the TEXT directive) or inside of matched parentheses (as in arguments of macro calls).

FIELDS

Directive and instruction statements contain four functional fields. The fields are, from left to right, the label field, the operation field, the operand field, and the comments field. The assembler accepts a free-form statement format; the various fields of a statement are delimited by blanks rather than by restricting them to fixed places in a line. This is explained in more detail below.

The label field is used mainly for symbol definitions. It begins with the first character in the statement and ends on the first blank. Thus, in the following statements, the symbol XYZ appears in two label fields.

```
    XYZ     LDA = 10 (RET)
   ↑STA DEF;XYZ LDA = 10 (RET)   ↑LDB* LMN
    |                            |
    label is omitted             label is omitted
```

The operation field contains a symbolic operation code, a directive name, or a macro call. It begins with the first nonblank character after the termination of the label field. In the statements above, each operation field begins in a different position, and it is terminated with a blank, asterisk, semicolon, or carriage return.

The operand and comments fields each begin with the first nonblank character after the termination of the preceding field. The operand field terminates on the first blank or semicolon that is not between matched single quotes or parentheses. The carriage return always terminates the field (and the statement). The comments field terminates on a semicolon or carriage return. Like the comments statement, the comments field is not used by the assembler. It may contain any sequence of characters.

## COMMENT STATEMENTS

An entire statement may be used as a comment by writing an asterisk as the first character. Any character, except a semicolon, may be used in a comment.

The assembler reproduces the comments on the assembly listing and counts comment lines in making line number assignments.

## PROCESSING SYMBOLS

Symbols are used in the label field of a machine instruction to represent the location of the instruction in the program. In the operand field, a symbol identifies a data value or the location of an instruction.

The treatment of symbols that appear in the label or operand field of a directive varies.

## DEFINING SYMBOLS

A symbol becomes "defined" by its appearance as a label field entry. "Defined" means that it is assigned a value. The definition depends on assembly conditions when the symbol is encountered, the contents of the operand field, and the current contents of the location counter.

Any instruction statement may be labeled; the label is assigned the current value of the location counter; a word within the assembler that contains the relative address of the instruction.

The assembler recognizes the following types of symbols:

LOCAL SYMBOLS

Local symbols are defined by their use in the label field of instructions and in some directives. Their value is that of the location counter at their definition. They are, therefore, symbolic addresses of memory locations. These symbols are relocatable (R = 1) if the assembly is relocatable; if the assembly is absolute, they are absolute. Once defined, a local symbol cannot be redefined. Attempts to do so are considered errors, and result in the appropriate diagnostic in the assembly listing.

EQUATED SYMBOLS

Equated symbols may be defined by equating them to an expression (using directives EQU, NARG, or NCHR). Their relocation value will be the same as the relocation value of the expression. Unlike local symbols, equated symbols may be given new values at any point in the program.

CURRENT LOCATION COUNTER SYMBOL

The character * (if used in the operand field) is defined to mean the current value of the location counter. This value is relocatable or absolute, depending on the nature of the assembly.

EXTERNAL SYMBOLS

External symbols are those that are used but not defined in a given subprogram. No value can be assigned to them, and it is not reasonable to regard them as either absolute or relocatable. An external symbol may be used only as a single term and must not be used in an expression having any other terms.

## PROGRAMS

A program consists of a sequence of statements terminated by an END directive. Normally, programs are assembled in relocatable form. A program is assembled in absolute self-loading form if it begins with an ORG directive. It is possible (by using RELORG) to make an absolute assembly to be loaded by DDT.

A relocatable program is one in which all memory addresses have been computed relative to the first word (or origin) of the program. A loader (for this assembler, DDT) can then place the assembled program into memory, beginning at whatever location may be specified at load time. Placement of the program involves a small calculation. For example, if a memory reference is to the nth word of a program and if the program is loaded beginning at location k, the loader must transform the reference n into absolute address n+k.

This calculation is not performed for each word of a program since some computer instructions (shifts, for example) do not refer to memory locations. Therefore, it is necessary to inform the loader whether or not to relocate the value of address field for each word of the program. Relocation information is determined automatically by the assembler and transmitted to the loader as a binary quantity called the relocation (R) bit. If R = 1 the address field of the word is to be relocated; if R = 0 the address field of the word is unchanged.

An expression value may similarly require relocation, the difference being that the relocation calculation applies to all 24 bits of the word, not just to the address field. The assembler accounts for this difference automatically.

It is possible to disable relocation in the assembler and to do absolute assembly. In this event TAP produces a paper tape that can be loaded into core memory using the 940 FILL switch.

### SUBPROGRAMS

Before executing a program that has been assembled as a series of subprograms, it is necessary to load the subprogram into memory and link them. The symbols used in a given subprogram are generally local to that subprogram. Subprograms do, however, need to refer to symbols that are defined in other subprograms. Such symbols are called external symbols. The loader's linking process takes care of such cross references.

## BASIC STATEMENT ASSEMBLY PROCEDURE

During pass 1 of the 2-pass process, the operands of instructions and some directives are scanned for the presence of symbols. If a symbol is present, a table of symbols is searched. If the symbol is absent from the table, it is added but marked as being undefined (i.e., as having no value).

Labels are placed into the symbol table during pass 1 in similar fashion, except that they are assigned the current value of the location counter. If a label has been previously defined, it is marked as a duplicate (this is an error).

At the end of pass 1 the symbol table is examined. All undefined symbols are assumed to be external. These symbols are then output by the assembler (as part of the object module for the program) for later use by the loader. During pass 2 the labels are not computed; rather, the operand file of instructions and directives are evaluated, using the defined symbol values.

In absolute assemblies, the scan for symbols during pass 1 is disabled.

# 3. INSTRUCTIONS

All SDS 940 instructions may be represented symbolically and combined with other assembly language elements to form instruction statements. This allows the programmer to write symbolic addresses, literals, mnemonic operation codes, and asterisks to specify indirect addressing, expressions to represent references to data, and so on.

There are two classes of 940 instructions. Class 1 instructions include all instructions that may invoke a memory access for an operand. Class 2 instructions, on the other hand, include all instructions that normally do not invoke a memory access for an operand. Appendix B contains a complete list of 940 instructions.

## CLASS 1 INSTRUCTIONS

Class 1 instructions generally use the statement operand field; the absence thereof implies the value zero. It is possible to specify, for each Class 1 instruction, whether or not the operand field must be present. It is also possible to specify that bit 0 of the instruction word is to be set to one (as in SYSPOPs). There are two types of Class 1 instructions:

Type 0: The address is formed modulo $2^{14}$. All instructions making memory references are of this type.

Type 1: The operand is formed modulo $2^9$. This type is used for shift instructions. If indirect addressing is used with this type, the address is formed modulo $2^{14}$.

Class 1 instructions have the following form:

```
[[$] label] mnemonic [*] [operand[, tag]] [comment]
```

Indirect addressing is signified by an asterisk immediately following the mnemonic. The use of the dollar sign is explained later in this chapter. The tag is used to specify bits 0, 1, and 2 of the 940 instruction word.

## CLASS 2 INSTRUCTIONS

Class 2 instructions have no operand field. Indirect addressing is signified by an asterisk immediately following the mnemonic.

Class 2 instructions have the following form:

```
[[$] label]  mnemonic [*]  [comment]
```

## INSTRUCTION FIELD PROCESSING

### LABEL FIELD

A label identifies the instruction or data word being generated. The symbol used in the label field is given the current value of the location counter. Generally, instructions will have labels if they are referred to elsewhere in the program, although it is not necessary that symbols defined in this way be used in references. Symbols defined but not referenced are called nulls; they are marked as such in the assembly listing.

If the same symbol appears in the label field of more than one instruction statement, it is marked as a duplicate and given the newer value.

A $ preceding a label defines an external symbol. (See the description of the EXT directive in Chapter 4.)

### OPERATION FIELD

The operation field generally contains a mnemonic operation code (such as LDA, STA, etc.). However, instruction operation codes may also be specified with decimal or octal numbers, as for example:

```
[[$ label]  76B [*]  [operand[, tag]]  [comment]
```

The assembler shifts the numeric operation code (modulo $177_8$) left to the correct position in the computer instruction word. In such cases, the instruction is assumed to be Class 1, type 0, no operand required, and with bit 0 not set.

### OPERAND FIELD

The operand field contains, at most, two arithmetic expressions (or a literal and one expression) used to determine the address and tag fields of the computer instruction word. The tag, if present, is evaluated modulo $2^3$ and must be absolute (i.e., non-relocatable).

### COMMENTS FIELD

The comments field is not processed by the assembler, but is copied to the assembly listing.

# 4. DIRECTIVES

Commands to the assembler are called "directives". Directives may be combined with other language elements to form directive statements. Directive statements, like instruction statements, have four fields: label, operation, operand, and comments.

A label field entry is required for eight directives: EQU, EXT, IDENT, MACRO, NARG, NCHR, OPD, and POPD.

If any of the directives ASC, BES, BSS, COPY, CRPT, DATA, IF, RPT, or TEXT are labeled, the label is defined as the current value of the location counter and identifies the first word of the area generated or specified by the directive.

For other directives a label field entry is ignored; i.e., it is not defined, entered in the symbol table, or assigned memory locations. As the format of each directive is explained, a label field entry is shown for each that requires or permits a label. For all other directives the label field is blank.

The operation field entry is the directive itself. If this field consists of more than one subfield, the directive must be in the first subfield, followed by the other required entries.

Operand field entries vary for the different directives. These entries are defined in the discussion of each directive. A directive format with a blank operand field implies that arguments are ignored for that directive.

Comments field entries are always optional.

Although many of the directives are similar, each has a specific syntax. Note the summary given below.

| Class | Directive | Use |
|---|---|---|
| Data | COPY | Generate RCH instruction |
| Generation | DATA | Generate data |
| | TEXT | Generate text |
| | ASC | Generate text |
| Value | EQU | Set or change symbol value |
| Declaration | EXT | Define external symbol |

| Class | Directive | Use |
|---|---|---|
| Value Declaration | NARG | Equate symbol to number of arguments in macro call |
| | NCHR | Equate symbol to number of characters in operand |
| | OPD | Define operation code |
| | POPD | Define programmed operator |
| Assembler Control | BES | Block ending symbol |
| | BSS | Block starting symbol |
| | ORG | Program origin: absolute assembly |
| | END | End of assembly |
| | DEC | Interpret integers as decimal |
| | OCT | Interpret integers as octal |
| | RAD | Set special relocation radix |
| | FRGT | Forget name of symbol |
| | IDENT | Identify name of program |
| | DELSYM | Do not transmit symbols to loader |
| | RELORG | Assemble relative with absolute origin |
| | RETREL | Return to relocatable assembly |
| | FREEZE | Preserve symbols, operation codes, and macros |
| | NOEXT | Do not create external symbols |
| Output and Listing Control | LIST | Turn on specified listing controls |
| | NOLIST | Turn off specified listing controls |
| | PAGE | Begin new page of assembly listing |
| | REM | Type out remarks in pass 2 |
| Macro Generation | MACRO | Begin macro definition |
| | ENDM | End macro definition |
| Conditional Assembly | RPT/ENDR | Begin/end repeat block |
| | CRPT/ENDR | Begin/end conditional repeat block |
| | IF/ENDF | Begin/end IF body |
| | ELSF | Alternative IF body |
| | ELSE | Alternative IF body |

In the individual directive descriptions, the name of the directive is followed by its syntactical format and an explanation of its purpose and usage.

## DATA GENERATION DIRECTIVES

**COPY** Generate RCH (REGISTER CHANGE) Instruction

```
[[$] label]     COPY  s1[,...,sn] [comment] (RET)
```

where

$s_i$ are special symbols for the functions of the various bits. Moreover, these symbols have this special meaning only when used with this directive; there is no restriction on their use either as symbols or as operation codes elsewhere in a program. The symbols are:

| Symbol | Address bit set | Function |
|---|---|---|
| A | 23 | Clear A |
| B | 22 | Clear B |
| AB | 21 | Copy (A) $\longrightarrow$ B |
| BA | 20 | Copy (B) $\longrightarrow$ A |
| BX | 19 | Copy (B) $\longrightarrow$ X |
| XB | 18 | Copy (X) $\longrightarrow$ B |
| E | 17 | Bits 15-23 (exponent part) only |
| XA | 16 | Copy (X) $\longrightarrow$ A |
| AX | 15 | Copy (A) $\longrightarrow$ X |
| N | 14 | Copy -(A) $\longrightarrow$ A (negate A) |
| X | 2 | Clear X |

The COPY directive produces an RCH instruction. It takes in its operand field a series of the special symbols written in any sequence, with each symbol standing for a bit in the address field of the instruction. The bits selected by a given choice of symbols are merged together to form the address. For example, instead of using the instruction CAB (04600004), one could write COPY AB. The special symbol AB has the value 00000004.

The advantage of the directive is that unusual combinations of bits in the address field — those for which no operation codes normally exist — may be created quite naturally.

To exchange the contents of the B and X registers and negate A (only for bits 15-23 of all registers) write

```
COPY   BX, XB, N, E (RET)
```

This directive facilitates some special RCH functions that might not otherwise be attempted. For example,

```
COPY   AX, BX (RET)
```

has the effect of loading into X the logical OR (merging) of the A and B registers (refer to the SDS 940 Computer Reference Manual for more details of the RCH instruction).

**DATA** Generate Data

```
[[$] label]  DATA  exp1[,...,expn]    [comment] (RET)
```

The DATA directive is used to produce data in programs. Each expression ($exp_i$) in the operand field is evaluated and the resulting 24-bit values are assigned to ascending memory locations. One or more expressions may be present. The label is assigned to the location of the first value. The effect of this directive is to create a list of data, the first word of which may be labeled.

Since the expressions are not restricted in any way, any type of data can be created with this directive. For example:

```
DATA  100,-217B, START, AB*2/DEF, 'NOTE', 5 (RET)
```

**TEXT** Generate Text

```
[[$] label]  TEXT  { 'text'            }  [comment] (RET)
                   { expression, text  }
```

The TEXT directive is used to create a string of 6-bit trimmed ASCII characters, packed four to a word and assigned to ascending memory locations. The first word of the string can be labeled. The string to be packed can be delineated by enclosing it in quotes (as in the first form). The second form of the directive must be used if the string contains one or more quotes.

Note: If a statement contains a single quote (or any odd number of them), it will not terminate with a semicolon; a carriage return must be used. For example:

```
    TEXT 4, THIS WON'T WORK; TEXT (RET)
    +4, DISASTER AHEAD (RET)
```

In the line above, the semicolon will be part of the text, and the second statement will be interpreted as being in the comments field. Legal examples are:

```
    TEXT    4, THIS WILL ' (RET)
    TEXT    1, A-OK (RET)
```

In the first form of the directive, characters in the last word are left-justified and remaining positions filled in by blanks (octal 00). In the second form, sufficient characters are packed to satisfy the word count.

**ASC** Generate Text with Three Characters per Word

```
[[$] label]  ASC  { 'text'             }  [comment] (RET)
                  { expression, text   }
```

This directive is identical in use to TEXT, except that 8-bit characters are packed three per word. The 940 string processing system normally deals with such text.

## VALUE DECLARATION DIRECTIVES

**EQU** Set or Change Symbol Value

```
[$] symbol  EQU  expression   [comment] (RET)
```

The EQU directive causes the symbol in its label field to be given the value of the expression. The expression must have a value when EQU is first encountered; i.e., symbols present in the operand field must have been previously defined. It is permissible to redefine by EQU any symbol previously defined by EQU (or NARG or NCHR, as described below). This facility is particularly useful in macros and conditional assemblies.

**EXT** Define External Symbol

| |
|---|
| $symbol {directive operand / opcode} [comment] (RET) |
| symbol EXT (comment not permitted) (RET) |
| $symbol EQU expression [comment] (RET) |
| symbol EXT expression [comment] (RET) |

There are four ways to define external symbols. In method 1 the $ preceding the symbol in the label field causes the symbol to be defined externally at the same time it is defined locally.

In method 2 the symbol in the label field is defined externally. This symbol must have been defined previously in the program. The operand and comment fields must be absent.

Methods 1 and 2 have the same effect; the name and value of a local symbol is given to the loader for external purposes. Occasionally it is desirable to define an external symbol whose name is different from that of a local symbol. An external symbol may be defined in terms of an expression involving local symbols. This is performed by utilizing methods 3 and 4.

In method 3, the symbol is defined both locally and externally, at the same time.

Method 4 differs from method 3 in that the symbol in the label field is defined externally only; its name and value are completely unknown to the local program. Method 4 is particularly useful in situations in which two or more subprograms, loaded together, have name conflicts.

For example, assume programs A and B both make use of the symbol START, and A not only refers to its own START but B's as well. The latter references to START can be changed to BEGIN. Then the line

```
    BEGIN  EXT  START (RET)
```

can be inserted into program B.

No other changes need be made either to A or B.

In summary, methods 1-3 define a symbol as both local and external. Method 4 defines a symbol as external but not local.

Occasionally, after having written a program, one would like to make a list of local symbols to be externally defined. A built-in TAP macro, ENTRY, serves the function. For example:

```
    ENTRY    A, B, C, D, . . . (RET)
```

is precisely equivalent to

```
    A   EXT (RET)
    B   EXT (RET)
    C   EXT (RET)
    D   EXT (RET)
    .   .
    .   .
    .   .
```

**NARG** Equate Symbol to Number of Arguments in Macro Call

```
[$] symbol  NARG  [comment] (RET)
```

This directive may be used only in macro definitions. It is mentioned here only for completeness. It operates exactly as EQU except that, in place of an expression in the operand field, the value of the symbol is set to the number of arguments used in calling the macro currently being expanded (see "Macro Generation Directives").

**NCHR** Equate Symbol to Number of Characters in Operand

```
[$]symbol  NCHR  operand  [comment] (RET)
```

This directive is primarily intended for use in macro definitions, but it may be used elsewhere. It operates exactly as EQU except that, in place of an expression in the operand field, the value of the symbol is set to the number of characters in the operand field (see "Macro Generation Directives" for a further explanation of the utility of this directive).

**OPD** Define Operation Code

```
symbol  OPD  expression, class,  [ar]  [, type]  [, sb] (RET)
+ [comment] (RET)
```

where

| | | |
|---|---|---|
| class must be 1 or 2 | | |
| ar (address required) may be 0 or 1 | } | optional |
| type may be 0 or 1 | | |
| sb (sign bit) may be 0 or 1 | | |

The OPD directive gives the programmer the facility to add new codes to the existing table of operation codes kept in the assembler or to change the current ones.

Bits governed by the optional items are set to zero if the items are missing. As examples of how the directive is used, some standard 940 instructions are defined as follows:

```
CLA    OPD    04600001B, 2 (RET)
LDA    OPD    76B5, 1, 1 (RET)
RCY    OPD    662B4, 1, 1, 1 (RET)(Type 1 = SHIFT)
```

A hypothetical SYSPOP LLA might be defined by

```
LLA    OPD    110B5, 1,1,0,1(RET)(class 1, address
  required, type 0, sign bit set).
```

In operation, the assembler simply adds new operation codes defined by OPD to its operation code table. This table is always searched backward, so the new codes are seen first. At the beginning of the second pass the original table boundary is reset; thus, if an operation code is redefined somewhere during assembly, it is treated identically in both passes.

**POPD** Define Programmed Operator

```
symbol  POPD  expression, class,  [ar]  [, type]  [, sb](RET)
+ [comment] (RET)
```

In programs containing POPs it is desirable to provide the POPD directive.

This directive is similar to the OPD and is used in the same way. It differs from OPD in that it automatically places a branch instruction to the body of the POP routine in the POP transfer vector ($100_8$ - $177_8$).

In order to do this, the assembler must know two things:

1. the location for the branch instruction in the transfer vector.

2. the location of the POP routine (i.e., the address of the branch instruction).

Item 1 is given by the POP code itself. Item 2 is provided by the convention that the POPD must immediately precede the body of the POP routine. The address of the branch instruction placed in the transfer vector is the current value of the location counter.

## ASSEMBLER CONTROL DIRECTIVES

**BES** Block Ending Symbol

```
[[$]symbol]   BES   expression   [comment] (RET)
```

BES reserves a block of storage for which the first location after the block can be labeled (if a symbol is presen in the label field). The block size is determined by the value of the expression; therefore, the expression must be absolute and it must have a value when BES is first encountered (symbols present in the operand field must have been previously defined). BES is most useful for labeling a block that is to be referred to by indexing with the BRX instruction (where the contents of X are usually negative). For example, to form the sum of the contents of an array, one might write

```
       LDX  =-100     ARRAY HAS 100 ENTRIES (RET)
       CLA (RET)
LOOP   ADD  ARRAY,2 NEGATIVE INDEXING HERE(RET)
       BRX  *-1 (RET)
       STA  RESULT (RET)
       HLT (RET)
ARRAY  BES  100 (RET)
```

**BSS** Block Starting Symbol

```
[[$]symbol]  BSS  expression  [comment](RET)
```

BSS reserves a block of storage for which the first word may be labeled (if a symbol is present in the label field). The block size is determined by the value of the expression; therefore, the expression must be absolute and it must have a value when BSS is first encountered. The difference between BSS and BES is that in the case of BSS, the first word of the block is labeled; for BES, the first word after the block is labeled by the associated symbol. BSS is most useful for labeling a block that is referred to by positive indexing.

**ORG** Absolute Program Origin

ORG expression [comment] (RET)

The use of ORG forces an absolute assembly. The location counter is initialized to the value of the expression. The expression must therefore be absolute, and it must have a value when ORG is first encountered. An ORG must precede the first instruction or data item in an absolute program, although it does not have to be the first statement. The output of the assembler will include a bootstrap loader that is capable of loading the program after initiation by the 940 FILL switch.

**END** End of Assembly

END expression [comment] (RET)

The END directive terminates the assembly. For relocatable assemblies, no expression is used. For absolute assemblies, the expression gives the starting location for the program. When assembling in the absolute mode (i.e., under control of the ORG directive), the assembler produces a paper tape that allows for loading with the FILL switch (out of the time-sharing mode). If the expression is not included with the END directive, the bootstrap loader on this paper tape will cause the computer to halt after the tape has been read in. Otherwise, control will automatically transfer to the location designated in the expression.

**DEC** Interpret Integers as Decimal

DEC [comment] (RET)

Integers terminated with a B or D are always interpreted, respectively, as having either an octal or decimal base. On the other hand, integers not terminated with these letters may be interpreted either as decimal or octal, depending on the setting of a mode switch within the assembler. The mode switch is set to decimal by the DEC directive.

When a new assembly begins, the mode switch is initialized to decimal. Thus, the DEC directive is not really necessary unless the mode switch has been changed to octal (with the OCT directive) and a return to decimal is desired.

**OCT** Interpret Integers as Octal

OCT [comment] (RET)

This directive sets a mode switch within the assembler to interpret unterminated integers as octal. When a new assembly begins, the mode switch is initialized to decimal. Thus, the OCT directives must be used before unterminated octal integers can be written.

**RAD** Set Special Relocation Radix

RAD expression [comment] (RET)

As explained in Chapter 2 (see "Special Relocation"), it is possible in a limited way to have multiple relocation of symbols. This action is performed when the special relocation operator, (R), is used. The value of a symbol preceded by (R) is multiplied by a constant (called the radix of the special relocation). The loader is informed of this situation so that it can multiply the base address by this same constant before performing the relocation.

Because the special relocation was developed specifically to facilitate the assembly of string pointers, this constant is initialized to 3. If it is desired to change its value, however, the RAD directive must be used. The value of the expression in the operand field sets the new value of the radix. It must be absolute, and the expression must have a value when it is first encountered.

**FRGT** Forget Name of Symbol

FRGT $s_1,\ldots,s_n$ [comment] (RET)

where

$s_i$ are previously defined symbols.

The use of FRGT prevents the symbols named in its operand field from being listed or delivered to DDT. FRGT is especially useful in situations in which symbols have been used in macro expansions or conditional assemblies. Frequently such symbols have meaning only at assembly time; they have no connection whatever with the program being assembled. Later, when DDT is used, however, memory locations sometimes are printed out in terms of these meaningless symbols. It is desirable to keep these symbols from being delivered to DDT.

**IDENT** Identify Program

symbol IDENT [comment] (RET)

IDENT causes the symbol in the label field to be delivered to DDT as a special identification record. DDT uses the IDENT name in conjunction with its treatment of local symbols. In the event of a name conflict between local symbols in two different subprograms, DDT resolves the ambiguity by linking the preceding IDENT name to the symbol in question.

IDENT statements are otherwise useful for editing purposes. They are always listed on pass 2, usually on the teletype.

**DELSYM** Delete Output of Symbol Table and Defined Operation Codes

DELSYM [comment] (RET)

DELSYM inhibits the symbol table and operation codes defined in the course of assembly from being output for later use by DDT. Its main purpose is to shorten the object code output from the assembler. This might be especially desirable for an absolute assembly (that produces a paper tape binary output).

**RELORG** Assemble Relative with Absolute Origin

```
RELORG expression    [comment] (RET)
```

It is occasionally desirable to assemble a section of code (as in the midst of an otherwise standard program) that will be loaded into core in some position, but is destined to run from another position in memory (it will first have to be moved there in a block). This is particularly useful when preparing program overlays.

RELORG, like ORG, takes an absolute expression denoting some origin in memory. It has the following effects:

1. The current value of the location counter is saved and the value of the expression is inserted in its place. This fact is not revealed to DDT; however, during loading, the next instruction assembled will be placed in the next memory cell available as if nothing had happened.
2. The mode of assembly is switched to absolute without changing the object code format; it still resembles a relocatable binary program to DDT. All symbols defined in terms of the location counter will be absolute.

It is possible to restore normal relocatable assembly with the RETREL directive (see below).

Some examples of the use of RELORG follow.

1. A program begins with RELORG 300B and ends with END. The assembler's output represents an absolute program whose origin is $00300_8$ but it can be loaded anywhere, using DDT in the usual fashion. However, it is necessary to move the program to location $00300_8$ before execution.
2. The program starts and continues normally as a relocatable program. Then there is a series of RELORGs and some RETRELs. The effect is as shown below:

```
    ________
    ________  }  Relocatable program
    ________

    RELORG  100 (RET)
    ________
    ________  }  Absolute program origined to 100
    ________

    RELORG  200 (RET)
    ________
    ________  }  Absolute program origined to 200
    ________

    RETREL (RET)
    ________
    ________  }  Relocatable program
    ________

    RELORG  300 (RET)
    ________
    ________  }  Absolute program origined to 300
    ________
    END
```

**RETREL** Return to Relocatable Assembly

```
RETREL    [comment] (RET)
```

This directive is used when it is desired to return to relocatable assembly after a RELORG directive. The effects of RETREL are:

1. To restore the location counter to what it would have been had the RELORG(s) never been used.
2. To return the assembly to the relocatable mode.

**FREEZE** Preserve Symbols, Operation Codes, and Macros

```
FREEZE    [comment] (RET)
```

Subprograms occasionally share definitions of symbols, operation codes, and macros. It is possible to cause the assembler to take note of the current contents of its symbol and operation code tables and the currently defined macros, and to include them in future assemblies. This eliminates the need for including copies of this information in every subprogram's source code. This feature greatly facilitates the editing of source code.

When the FREEZE directive is used, the current table boundaries for symbols and operation codes and the storage area for macros is noted and saved for later use. These tables may then continue to expand during the current assembly (a separate subprogram may be used to make these definitions; it will end with FREEZE; END). An assembly will use the symbols and macros that were defined in a previous assembly that used the FREEZE directive if the FREEZE command (see TAP Commands, Chapter 5) is issued before the ASSEMBLE command

```
*FREEZE
```

If the FREEZE command is not issued, the symbol table will be lost and thus unavailable to this or future assemblies.

Note: When the assembler has been pre-loaded with symbols, operation codes, and macros, it cannot be released (i.e., one cannot call another subsystem such as DDT, QED, etc.) without the loss of this information.

**NOEXT** Do Not Create External Symbols

```
NOEXT    [comment] (RET)
```

Because of its subprogram capability, the assembler automatically assumes that symbols not defined in a given program are external, and will be defined in another subprogram. Thus, it does not list the use of such symbols as errors.

If a program is a free-standing program (i.e., it is complete), undefined symbols are errors and should be so noted during assembly. The NOEXT directive prevents external symbols from being established; undefined symbols are noted as errors. The directive must be used at the beginning of a program (i.e., before instructions or data have been assembled). Its use affects the entire program.

## OUTPUT AND LISTING CONTROL DIRECTIVES

The assembler provides a means of listing a program during assembly, by printing out such items as the location counter, binary code being assembled, source program statements, etc. The association of these items on one page is frequently of great help to programmers. Two directives, LIST and NOLIST, control this process.

**LIST** and **NOLIST** Turn Specified Listing Controls On or Off

```
     LIST    [1 or 2],[ME]  [comment] (RET)
   NOLIST    s,[,...,sn]  [comment]  (RET)
```

where

$s_i$ are from a set of special symbols. A list of the special mnemonic symbols used in conjunction with these two directives is given below. The symbols have special meaning only when used with LIST and NOLIST. They may be used at any other time for any purpose.

| Symbol | Meaning |
|---|---|
| 1 | List during pass 1. (Listing format is controlled by other parameters.) The pass 1 listing is followed by a pass 2 listing. |
| 2. | List during pass 2. The listing includes the location counter (LCT), the binary object code (BIN), the source language (SRC), the comments (COM), and the macro calls (MC). The user can suppress any of these options by using the NOLIST directive. The external symbols, duplicate symbols, and null (non-referenced) symbols will be listed at the end of the assembly. |
| LCT | List location counter value |
| BIN | List binary object code or values |
| SRC | List source language |
| COM | List comments |
| MC | List macro calls |
| ME | List certain directives during macro expansions (EQU, NCHR, NARG, RPT, CRPT, ENDR, IF, ELSF, ELSE, ENDF, ENDM) |

As an example of the meanings of various symbols above, the line of code

```
    A21    STB    OUTCHR    SAVE POINTER
```

might be listed as

$$\underbrace{\text{02157}}_{\text{LCT}}\ \underbrace{\text{0 36 00217}}_{\text{BIN}}\ \underbrace{\text{A21 STB OUTCHR}}_{\text{SRC}}\ \underbrace{\text{SAVE POINTER}}_{\text{COM}}$$

It is not necessary to include each possible symbol, but rather only those parameters for which changes are desired. It is, in fact, not necessary to give any symbols. For example:

```
    LIST    is equivalent to    LIST 2
```

At the beginning of an assembly, the assembler initializes itself to the following directives:

```
    LIST 2
```

In addition, the complete symbol table (including null and duplicate symbols) and the external symbols are listed at the end of the assembly.

Following is an example of the interaction of the LIST and NOLIST directives:

| Directive | Effect |
|---|---|
| LIST 2,ME | Pass 2 listing including the macro call and code generated by the macro expansion |
| LIST<br>NOLIST MC,ME | Pass 2 listing, but only the code generated by a macro call is listed. |
| LIST<br>NOLIST LCT,SRC | Pass 2 listing but not the location counter or the source language. |

**PAGE** Begin New Page Of Assembly Listing

```
    PAGE    [comment] (RET)
```

This directive causes a page eject for the assembly listing medium (unless an automatic page eject has just been given). It is used to improve the appearance of the assembly listing.

**REM** Type Out Remark During Pass 2

```
    REM    remark to be typed (RET)
```

This directive, when encountered during pass 2, causes the remark to be typed out either on the teletype or whatever file has been designated as the output message device. This typeout occurs regardless of specified listing parameters. The directive may be used for a variety of purposes: it may inform the user of the progress of assembly; it may give him instructions on what to do next (this might be especially useful for complicated assemblies); it might announce the last date the source language was updated; or, it might be used within complex macros to show which argument substrings have been created during expansion of a highly nested macro (this is useful for debugging purposes).

## MACRO GENERATION DIRECTIVES

On the simplest level, a macro name may be thought of as an abbreviation or shorthand notation for one or more assembly language statements. In this respect it is like an operation code. The operation code is the name of a machine instruction and the macro name is the name of a sequence of assembly language statements.

The SDS 940 computer has an instruction for skipping if the contents of a specified location are negative, but has none for testing the accumulator. SKA (skip if memory and accumulator do not compare ones) will serve when used with a cell whose contents are 40000000B. The meaning of SKA used in this way is "skip if A positive". Thus, a programmer would write

```
SKA  =4B7 (RET)
BRU  NEGCASE          NEGATIVE CASE (RET)
  .
  .
  .
```

However, a program may have a logical need for skipping if the accumulator is negative. In such a situation, the programmer must write

```
SKA  =4B7 (RET)
BRU  *+2 (RET)
BRU  POSCASE          POSITIVE CASE (RET)
  .
  .
  .
```

Both of the above situations are awkward in terms of assembly language programming. The macro facility allows for performing the operations SKAP and SKAN (skip if accumulator positive or negative) in a simple manner. For example, if the programmer defines these operations as macros, with

```
SKAP    MACRO (RET)
        SKA      =4B7 (RET)
        ENDM (RET)
SKAN    MACRO (RET)
        SKA      =4B7 (RET)
        BRU      *+2 (RET)
        ENDM (RET)
```

he may now write

```
A22     SKAN (RET)
        BRU      POSCASE (RET)
          .
          .
          .
```

The ability to use SKAP or SKAN reduces the amount of code written in the course of a program. This in itself tends to reduce errors. A greater advantage is that SKAP and SKAN are more indicative of the action that the programmer has in mind. Programs written in this way tend to be easier to read. Also, a label may be used with a macro call. Labels used in this way are usually treated like the labels of instructions, they are assigned the current value of the location counter (this will be discussed in more detail later).

## MACRO DEFINITION

Before discussing more complicated use of macros, some additional terminology should be established. A macro is an arbitrary sequence of assembly language statements associated with a symbolic name. During assembly, the macro is held in an area of memory called text storage. Macros may be created or defined. To do this one must give the macro a name and list the sequence of statements comprising the macro. The name and the beginning of the sequence of statements in a macro is designated by the use of the MACRO directive. The end of the sequence of statements in a macro is signaled by the ENDM directive.

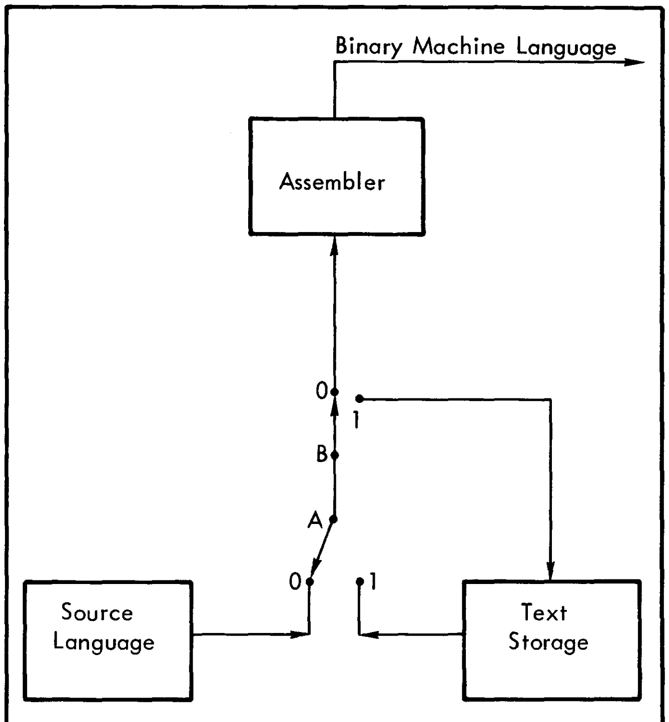
**MACRO/ENDM** Begin/End Macro Definition

```
name   MACRO   [p1, p2, p3]  [comment] (RET)
       ENDM    [comment] (RET)
```

where

| | |
|---|---|
| p(i) | are parameters defined later in this section. |

When the assembler encounters a MACRO directive, switch B (see Figure 3) is set to position 1. The programmer's source language is then copied into text storage. The assembler does not do any processing during this operation. When ENDM is encountered, switch B is put back to position 0.



| A | B | Effect |
|---|---|---|
| 0 | 0 | normal assembly |
| 0 | 1 | macro definition |
| 1 | 0 | macro expansion |
| 1 | 1 | macro definition during macro expansion (to be explained in more detail later) |

Figure 3. Information Flow During Macro Processing

It is possible that other macro definitions may be embedded within a given macro definition. The macro-defining process counts the occurrences of the MACRO directive and matches them against the occurrences of ENDM. Switch B is placed back in position 0 only when the number of ENDM directives equals the number of MACRO directives. Thus, MACRO and ENDM constitute opening and closing brackets around a segment of source language. "Nested" macro structures like the following are possible.

```
NAME1   MACRO(RET)
          :
NAME2   MACRO(RET)
          :
NAME3   MACRO(RET)
          :
        ENDM(RET)
          :
NAME4   MACRO(RET)
          :
        ENDM(RET)
          :
        ENDM(RET)
          :
NAME5   MACRO(RET)
          :
        ENDM(RET)
          :
        ENDM(RET)
```

Use of embedded definitions should be kept to a minimum, since large amounts of text storage are required. What is important, however, is an understanding of when the various macros are defined. In particular, when NAME1 is defined, NAME2, NAME3, etc., are not defined; they are copied into text storage. NAME2 is not defined until NAME1 is used in the operation field of a statement.

## MACRO EXPANSION

The use of a macro name in the operation field of a statement is referred to as a macro call. The assembler, upon recognizing a macro call, moves switch A to position 1 (again see Figure 3). Input to the assembler from the original source language ceases temporarily and, instead, input is called from text storage. During this period, the macro is said to be undergoing expansion. (A macro must first be defined before it can be called.)

A macro expansion may include other macro calls, and these, in turn, may call others. Macros may also call themselves, a process called recursion. When a new macro expansion begins within a current macro expansion, information about the progress of the current expansion is preserved in the assembler's working storage. Successive macro calls cause similar information to be preserved. At the end of each nested macro expansion, the most recently initiated expansion is resumed. When the initial expansion finally terminates, switch A is placed back in position 0. Input then resumes from the source language program.

## MACRO ARGUMENTS

It might be useful to write macros BRAP and BRAN (branch to specified location if contents of the accumulator are positive or negative, respectively), rather than SKAP and SKAN. In such cases, the branch location is not known when the macro is defined; different locations will be used for each call.

The macro processor, therefore, allows the programmer to provide some of the information for macro expansion at the time the macro is called. This is done (in macro definitions) by permitting dummy arguments to be replaced by arguments supplied when the macro is called. Each dummy argument is referred to in the macro definition by a subscripted symbol. This symbol (dummy name) is defined in the operand field of the MACRO directive.

For example, the macro BRAP could be defined as

```
BRAP   MACRO    DUM(RET)
       SKAN(RET)
       BRU      DUM(1)(RET)
       ENDM(RET)
```

When called by the statement

```
       BRAP     POSCASE(RET)
```

the macro will expand to the statements

```
       SKA      =4B7(RET)
       BRU      *+2(RET)
       BRU      POSCASE(RET)
```

Note that BRAP is defined in terms of another macro SKAN (a matter of choice in this example). Also, note that BRAP is intended to take only one argument. Other macros may use more than one argument; e.g., the macro CBE (compare and branch if equal) takes two arguments. The first argument is the location of a cell to be compared for equality with the accumulator; the second is a branch location in case of equality. The definition of CBE is

```
CBE    MACRO    D(RET)
       SKE      D(1)(RET)
       BRU      *+2(RET)
       BRU      D(2)(RET)
       ENDM(RET)
```

When called by the statement

```
       CBE      =21B, EQLOC(RET)
```

the statements generated will be

```
       SKE      =21B(RET)
       BRU      *+2(RET)
       BRU      EQLOC(RET)
         :
```

Arguments in a macro call are separated by commas. It is possible to include both commas and spaces in a list of arguments by enclosing some of them in parentheses; the macro processor strips off the outermost parentheses of any substring used in a call. For example, in the call for the macro MUM

```
MUM    A, (B, C), (D  E)(RET)
```

the dummy arguments would be

```
D(1) = A
D(2) = B, C
D(3) = D E
```

## DUMMY ARGUMENTS IN MACRO DEFINITIONS

Before giving further examples of the use of macros, the various ways that dummy arguments may be used in macro definitions will be discussed. In general, a dummy argument may be referred to thusly:

```
dummy(expression)
```

The only restriction on the expression is that it can not contain other dummies or generated symbols (generated symbols are discussed later in this chapter). Furthermore, the expression must have a known value when the macro is called. It should be noted that a macro call may deliver more arguments than are referred to in its definition. However, the situation wherein a dummy argument is missing from an argument list when the macro call occurs is considered to be an error condition.

More than one dummy may be referred to, by the notation

```
dummy(expression, expression)
```

as in the case of the call

```
MUM     A, B, C, D, E RET
```

where

```
D(3,5)=C, D, E
```

This situation may lead to ambiguity, as in the case of the call

```
MUM     A, B, C, (D, E), F RET
```

where

```
D(3,5)=    C, D, E, F
```

In this case, it is not clear which arguments correspond to D(3), D(4), and D(5). To resolve this ambiguity, the assembler produces the string

```
(C), (D, E), (F)
```

The notation

```
dummy()
```

produces all of the arguments supplied in a macro call, and each argument is surrounded by parentheses, as in the example above.

The symbolism

```
dummy(0)
```

refers to the label field of the macro call. Normally, the current value of the location counter is assigned to the label used with a macro call (as with any instruction). However, explicit use of

```
dummy(0)
```

causes the label field to be used to transmit another argument. This situation is possible in three cases:

1. The macro contains no references to dummy(0). The label field is treated normally in this case; i.e., assigned the current value of the execution location counter.

2. The macro contains at least one reference to dummy(0). In this case, the label field merely transmits an argument that replaces dummy(0) in the expansion.

3. The macro contains no references to dummy(0) explicitly, but does contain

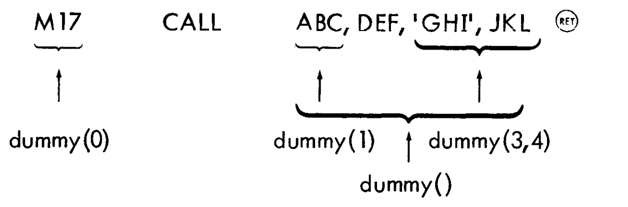
   ```
   dummy(expression)
   ```

   where the value of the expression is zero when the macro call occurs. In this case the label field is handled as in case 1 above and is also used to transmit the argument referred to by

   ```
   dummy(expression)
   ```

   as in case 2.

Thus, in a typical call, we have the following relationships:



Sometimes in a macro definition, it is desirable to refer to only a portion of an argument (perhaps to a character or a few characters). In the case of a single character this may be done by writing

```
dummy(expression$expression)
```

The first expression designates which argument is being referenced; the second expression determines which character of that argument is being referenced. If reference to a substring of an argument is desired, write

```
dummy(expression$expression,expression)
```

The second and third expressions determine the first and last characters of the substring, respectively. For example, the call

```
MUM  A,BCDE,'FGHIJ' RET
```

results in

```
    D(2$3)     =  D

    D(3$4,7)   =  HIJ'
```

Beginning with the ith character of a specified substring, the remaining characters of an argument can be obtained by specifying a terminal bound that is larger than the number of characters remaining in the argument. Thus the specification

```
    D(3$4,1000)  =  HIJ'
```

would obtain the substring beginning with the fourth character (of argument 3) and ending with the last character of the argument.

## CONCATENATION

It is frequently useful to compose statements from macro arguments (or parts of arguments) and other information given in the macro definition. This is done by concatenating the various objects; i.e., by having the assembler place them next to each other in the link.

To avoid ambiguity, use the dot or period character as a concatenation operator. The assembler uses the dot to delineate the terms it must deal with; in producing output, the macro-expansion processor ignores the dots after it recognizes the associated terms. Therefore, the dot character cannot be used in a macro definition for any other purpose.

For example, the macro STORE stores information into three storage cells that begin with the letters A, B, or X, depending on which 940 register is used as the source of information. The definition is

```
    STORE    MACRO     P (RET)
             ST.D(1)   D(2) (RET)
             ENDM
```

If the macro is called with

```
             STORE     B,DUM (RET)
or
             STORE     A,ZAP (RET)
```

the macro will generate

```
    STB    DUM    or    STA    ZAP
```

## GENERATED SYMBOLS

Sometimes it is convenient to put a label on an instruction within a macro. There are two methods (at least) of doing this. The first method involves transmitting the label as a macro argument when the macro is called. This allows the programmer to control the label being defined and refer to it elsewhere in the program.

However, there are situations in which the label is used purely for reasons local to the macro and it will not be referenced elsewhere. In cases like this it is desirable to allow for the automatic creation of labels. This may be done by means of the generated symbol.

A generated symbol name may be declared when a macro is defined. To do this requires both the name and the maximum number of generated symbols that will be encountered during an expansion. These two items may follow the dummy symbol name given in the MACRO directive. The actual format used is

```
name MACRO dummy name, generated name, expression
```

For example:

```
    MUM     MACRO  D,G,4 (RET)
                   :
            ENDM (RET)
```

In the definition of this macro there might be references to G(1), G(2), G(3), and G(4), these being individual generated symbols.

Regarding generated symbols, the macro expansion operates in the following fashion. For each macro, a generated symbol base value is initialized to zero at the beginning of assembly. As each generated symbol is encountered, the expression constituting its subscript is evaluated. This value is added to the base value and the sum is produced as a string of digits concatenated to the generated symbol name. Enough digits are produced to make a resultant symbol of six characters. Thus, the first time MUM is called, for example, G(2) will be transformed into G00002, G(4) into G00004, etc.

At the end of a macro expansion, the generated symbol base value is incremented. The increment is designated by the expression following the generated symbol name in the MACRO directive (this was 4 in the definition of MUM above). Thus, the second call of MUM will produce G00006 in place of G(2), the third call will produce G00010, etc. A generated symbol name should be kept as short as possible; it cannot be longer than 5 characters.

## CONVERSION OF A VALUE TO A DIGIT STRING

As an adjunct to the automatic generation of symbols (or for any other purposes for which it may be suitable), a facility is provided in the assembler's macro expansion process for conversion of the value of an expression at call time, to a string of decimal digits. The construct

```
    ($expression)
```

will be replaced by a string of digits equal in value to the expression. For example, if X = 5, then

```
    AB.($2*X-1)
```

will be transformed into

```
    AB9
```

Further examples of the use of this facility appear below.

### NARG AND NCHR DIRECTIVES

Macros can be more useful if the number of arguments supplied at call time is not fixed. The precise meaning of a macro (and the results of its expansion) may depend on the number or the arrangement of its arguments. In order to permit this, the macro undergoing expansion must be able to determine, at call time, the number of arguments supplied. The NARG directive makes this possible.

NARG functions basically like EQU, except that it is used without an expression. The basic form is

| |
|---|
| [$] symbol NARG [comment] (RET) |

The function of the NARG directive is to equate the value of the symbol to the number of arguments supplied to the macro currently being expanded. The symbol can then be used by itself or in expressions, for any required purpose. Examples of the use of NARG appear later.

It is also useful to be able to determine at call time the number of characters in an argument. The NCHR directive equates the symbol in its label field to the number of characters in its operand field. Its form is

| |
|---|
| [$] symbol NCHR character string [comment] (RET) |

The operand field of a statement is normally terminated by the first blank after the beginning of the field. This rule is rescinded if a macro argument containing blanks appears in the operand field. For example, in the statement

```
    XYZ    LDA    VECTOR,2    THIS IS A COMMENT (RET)
                  ↑       ↑
```

the arrows delineate the operand field. Alternatively, if a statement like

```
            TEXT X,D(1).ERROR
```

is placed in a macro definition and the macro is called by

```
            MUM     (NON-FATAL )
```

then the TEXT statement will turn out to be

```
            TEXT X,NON-FATAL ERROR
           ↑                 ↑
```

(Notice how the operand field terminates in this case.)

In the same example notice that the message produced by the TEXT directive is of unspecified length at definition time. Clearly, X must depend on the number of characters in D(1). Accordingly, MUM might be defined as

```
    MUM     MACRO    D (RET)
    X       NCHR     D(1) (RET)
    X       EQU      X+9  5 FOR 'ERROR',4 TO (RET)
    +ROUND UP (RET)
            TEXT     X/4,D(1).ERROR (RET)
            ENDM (RET)
```

## CONDITIONAL ASSEMBLY DIRECTIVES

The programming power of the assembler's macro capability is considerably multiplied when it is combined with the features explained in this section. These features — basically the if and repeat capabilities — are called conditional assembly capabilities because they permit assembly-time calculations to determine which elements of the source language are actually assembled. They are, however, not strictly a part of the macro capability and may be used quite apart from macros.

**RPT/ENDR** Begin/End Repeat Block

The RPT directive is, like the MACRO directive, an opening bracket for a segment of program called a repeat block. The end of the sequence of statements is signaled by the ENDR directive.

| |
|---|
| [label] RPT expression [comment] (RET) |
| [label] RPT(s=$e_1$, [$e_2$,] $e_3$)[...(s=$e_1$[, $e_2$])] [comment] (RET) |
| ENDR [comment] (RET) |

where

s specifies symbol and e specifies expression.

Form 1 directs the assembler to repeat the following sequence of statements down to the matching ENDR (end repeat) as many times as given by the value of the expression. The operations performed by form 2 are as follows:

1. Set the symbol s to the value of $e_1$.
2. Issue the sequence of statements down to the matching ENDR.
3. Increment s by the value of $e_2$ or by one (if $e_2$ is not present). If the new value of s has not passed the limit ($e_3$), reissue the sequence of statements to the matching ENDR (and increment s) until the value of s passes the limit.

The first parenthesized group determines the number of times the repeat is executed and controls the initial value and increment of a symbol. Subsequent groups (there may be up to ten of them) merely control the initial value and increments of other symbols in the repeat operation.

For example, assume that it is desired to create an area of storage that is cleared to zeros. The BSS directive cannot be used for this purpose since its function (that of reserving storage) is basically to advance the assembler's location counter. The problem is readily solved by

```
ABC   RPT    100 (RET)
      DATA   0 (RET)
      ENDR (RET)
```

which is equivalent to

```
ABC   DATA   0 (RET)  }
      DATA   0 (RET)  }
      DATA   0 (RET)  }
      DATA   0 (RET)  } 100 statements
      :               }
      DATA   0 (RET)  }
```

Note that the label is applied effectively only to the first statement.

As another example, consider the situation wherein it is desired to fill an area of storage with data starting with 0 and increasing by 5 for each cell. To illustrate:

```
X     EQU    0 (RET)
      RPT    20 (RET)
      DATA   X (RET)
X     EQU    X+5 (RET)
      ENDR (RET)
```

Alternatively (and more simply) one can write

```
      RPT      (X=0, 5, 100) (RET)
      DATA     X (RET)
      ENDR (RET)
```

Note that in the latter form, the terminal value (i.e., $e_3$) does not have to be positive or greater than the initial value of the symbol being incremented. Thus, both of the following two sequences are permissible.

```
      RPT      (X=100, -5, 20) (RET)
      :
      ENDR (RET)

      RPT      (X=INIT, -5, -30) (RET)
      :
      ENDR (RET)
```

A repeat block may be nested within other repeat blocks. This is similar to the nesting of macro definitions within other macro definitions; therefore, repeat structures similar to that described under "Macro Definition" may be constructed.

It may be desirable to create a pair of macros (SAVE and RESTOR) for the purpose of saving active registers at the beginning of a subroutine, and restoring the active registers at the completion of the subroutine. The macros should take a variable number of arguments so that, for example, one can write

```
   SAVE     A, SUBRS (RET)
```

to generate the instruction

```
   STA      SUBRSA
```

and also write the macro call

```
   RESTOR   A, B, X, SUBRS (RET)
```

to generate the instruction sequence

```
   LDA      SUBRSA
   LDB      SUBRSB
   LDX      SUBRSX
```

First define a generalized macro (MOVE) that is called by the same arguments delivered to SAVE and RESTOR, plus the strings 'ST' and 'LD' which determine whether to store or load.

```
MOVE     MACRO        D (RET)
X        NARG (RET)
         RPT          (Y=2, X-1) (RET)
         D(1). D(Y)   D(X). D(Y) (RET)
         ENDR (RET)
         ENDM (RET)
```

Then (in terms of MOVE) SAVE and RESTOR are readily defined as

```
SAVE     MACRO        D (RET)
         MOVE         ST, D() (RET)
         ENDM (RET)

RESTOR   MACRO        D (RET)
         MOVE         LD, D() (RET)
         ENDM (RET)
```

Many programs make use of <u>flags</u> (memory cells used as binary indicators). The SKN (skip if memory negative) instruction makes it easy to test these flags. Assume the convention that a flag is set if it contains the value -1 and reset if it contains the value zero; it is necessary to develop the macros SET and RESET to manipulate flags. Additionally, the name of the active register that will be used for the action, together with a list of flag locations must be delivered at call time. Calls for these macros might have the form

```
         SET          A, FLG1, FLG2, FLG3 (RET)
   or
         RESET        X, FLG37, FLG12 (RET)
```

As in the previous example, an intermediate macro (STORE) is used that is called with the same arguments delivered to SET and RESET.

```
STORE    MACRO        D (RET)
X        NARG (RET)
         RPT          (Y=2, X) (RET)
         ST. D(1)     D(Y) (RET)
         ENDR (RET)
         ENDM (RET)
```

Now SET and RESET can be defined as

```
SET       MACRO      D (RET)
          LD.D(1)    =-1 (RET)
          STORE      D() (RET)
          ENDM (RET)
RESET     MACRO      D (RET)
          CL.D(1) (RET)
          STORE      D() (RET)
          ENDM (RET)
```

**CRPT/ENDR** Begin/End Conditional Repeat Block

Occasionally it is necessary to perform an indefinite number of repeats, with termination of the repeat block being determined during the course of the repeat operation. The conditional repeat directive, CRPT, serves this function. Its effect is like that of RPT and its repeat block, like RPT, is terminated by a matching ENDR. However, instead of specifying a set number of repeats in the directive itself, the associated expression (exp$_1$) is evaluated (in a Boolean sense) to determine whether the repeat block should be issued. Its form is

```
[label]  CRPT exp. [, (s=e1, [e2,] e3[... (RET)
+(s=e1, [e2,] e3)]]  [comment] (RET)
```

For example:

```
CRPT      X>Y (RET)
 .
 .
ENDR (RET)
```

or

```
CRPT      STOP, (X=1, 2), (Y=-3, 5) (RET)
 .
 .
ENDR (RET)
```

Note that the statement

```
CRPT   10 (RET)
```

will cause an infinite number of repeats.

The termination of a CRPT operation is governed by the value of exp$_1$. Zero or negative values of exp$_1$ signify that the repeat operation is to occur. Values of one or greater for exp$_1$ signify that the repeat operation is not to occur.

## IF CAPABILITY

It is frequently desirable to permit the assembler either to assemble or merely skip blocks of statements, depending on the value of an expression at assembly time. This is primarily what is meant by the term conditional assembly. Conditional assembly can be done with CRPT by letting the condition be given by an expression. For example:

```
C      EQU (RET)  condition
       CRPT (RET) C
       .  }                   arbitrary block of statements
       .  }
C      EQU (RET)  0
       ENDR (RET)
```

Note that the line before ENDR is required to terminate the CRPT. By using the structure above, conditional assembly can be done; the arbitrary block of statements enclosed in the repeat body can be assembled on condition.

**IF/ENDF** Assemble if Expression is True

The same function, shown in the above example, can be performed much more conveniently by the IF directive. Its form is

```
[label]  IF      expression     [comment] (RET)
         ENDF                   [comment] (RET)
```

As with RPT and CRPT, the IF directive defines the beginning of a block of statements (called the IF body) that is terminated by a matching ENDF directive. The IF body may contain other nested IF bodies in the manner described in "Macro Definition".

For conditional assembly, there are often alternative IF bodies to be assembled in case a certain IF body is not assembled. This situation is most easily dealt with by the use of the ELSF and ELSE directives.

**ELSE/ELSF** Alternative IF Bodies

```
         ELSF    expression     [comment] (RET)
         ELSE    expression     [comment] (RET)
```

These provide a termination for the IF body and also begin another body to be assembled (again possibly on condition) in case the first body is not assembled. For example, consider the following structure.

```
IF      e1   (RET)
 .  }   body1
 .  }
ELSF    e2   (RET)
 .  }   body2
 .  }
ELSF    e3   (RET)
 .  }   body3
 .  }
ELSE  (RET)
 .  }   body4
 .  }
ENDF  (RET)
```

If $e_1>0$, body$_1$ is assembled and bodies 2, 3, and 4 are skipped (regardless of the values $e_2$ and $e_3$).

If $e_1<0$ and $e_2>0$, body 2 is assembled and bodies 1, 3 and 4 are skipped.

If $e_1<0$, $e_2<0$, and $e_3>0$, body 3 is assembled and bodies 1, 2, and 4 are skipped.

Finally, if $e_1<0$, $e_2<0$, and $e_3<0$, only body$_4$ is assembled.

An example of the use of IF (and other features) follows. This example illustrates several of the preceding features as well as the power of macros when used recursively.

The macro MOVE is intended to take any number of pairs of arguments. The first argument of each pair is to be moved to the second. Each argument, however, may itself be a pair of arguments, which may themselves be pairs, etc. So, basically, MOVE extracts pairs of argument structures and transmits such a pair to another macro, MOVE1.

```
MOVE    MACRO     D (RET)
X       NARG (RET)
        RPT       (Y=1, 2, X)(Z=2, 2) (RET)
        MOVE1     D(Y), D(Z) (RET)
        ENDR (RET)
        ENDM (RET)
```

MOVE1 calls itself recursively until it comes up with a single pair of arguments. It then generates code.

```
MOVE1   MACRO     D, G, 2
G(1)    NARG
G(2)    EQU       0
        IF        G(1)=2
        LDA       D(1)
        STA       D(2)
        ELSE
        RPT       G(1)/2
G(2)    EQU       G(2)+1
U       EQU       G(2)
        MOVE1     D(V), D(V+U/2)
        ENDR
        ENDF
        ENDM
```

When called by the line

```
        MOVE      A, B (RET)
```

the code generated will be

```
        LDA       A
        STA       B
```

When called by

```
        MOVE      A, B, C, D (RET)
```

the code generated is

```
        LDA       A
        STA       B
        LDA       C
        STA       D
```

When called by

```
        MOVE      (A, B), (C, D) (RET)
```

the code generated is

```
        LDA       A
        STA       C
        LDA       B
        STA       D
```

Finally, when called by

```
        MOVE      ((A, B),(C,D)),((E,F),(G,H)) (RET)
```

the code generated is

```
        LDA       A
        STA       E
        LDA       B
        STA       F
        LDA       C
        STA       G
        LDA       D
        STA       H
```

In this case the main call results in the call

```
        MOVE1     (A, B), (C, D), (E, F), (G, H) (RET)
```

MOVE1 calls itself by

```
        MOVE1     A, B, E, F
```

and (again)

```
        MOVE1     A, E
```

where the first code is generated. The result is

```
        MOVE1     B, F
```

Recursion then reverts to the call

```
        MOVE1     C, D, G, H
```

and so on.

Another macro example is given in Appendix C.

# 5. TAP COMMANDS

To call the assembler the user types

```
-TAP (RET)
```

All TAP commands are entered by typing the first character of the command. The remaining characters of the command are typed by the system and, except in the case of ASSEMBLE, the system then waits for a confirming carriage return preceded by any optional parameters.

The start of a new sequence of control commands is queued by an asterisk. Subsequent commands for the same assembly are queued by a plus sign. Commands for any assembly terminate with the ASSEMBLE command.

Figure 4 summarizes TAP commands and possible user responses. Standard assignments are given in Table 1.

| TAP Command | User Response | Comments |
|---|---|---|
| FREEZE | (RET) | Preserves any symbol and macro definitions that were defined by using the FREEZE directive in a previous assembly. If the FREEZE command is not issued, the symbols will not be available for this assembly or any subsequent assemblies. |
| INPUT: | Source file, ... (RET) | User types the input file name(s). Up to ten file names may be specified for each assembly. Assembly terminates on an END statement. |
| BINARY:<br>OLD FILE<br>or<br>NEW FILE | Object file (RET)<br><br>(RET) | User supplies the binary file name. The file name must be followed by the OLD/NEW carriage return confirmation. The user may type "escape" if he has provided an erroneous file name. If no binary output is required, the NOTHING file must be specified for binary output. If more than one binary file name is provided, the last file indicated will be used. |
| CREF | (RET) | A cross reference dictionary listing will be provided at the end of the assembly. |
| TEXT OUTPUT:<br>OLD FILE<br>or<br>NEW FILE | Listing file (RET)<br><br>(RET) | Specifies the file for text output. The file name must be followed by the OLD/NEW carriage return confirmation. If TEXT OUTPUT is not specified, but the LIST directive or command is used, the teletype is used as the output file. If TEXT OUTPUT is specified, but the LIST directive or command is not used, the default options of the LIST directive apply. If TEXT OUTPUT is not specified and the LIST command or directive is not used, errors, nulls, and externals are listed on the teletype. |
| LIST | (See Table 1) (RET) | The user may override standard listing parameters or override those he had previously specified with the NOLIST directive. |
| NOLIST | (See Table 1) (RET) | The user may override standard listing parameters or those he had previously specified with the LIST directive. |
| ASSEMBLE | | All input files specified since the initiation of the current control sequence will be assembled. |

Figure 4. TAP Commands and User Responses

The following examples illustrate typical uses of the TAP commands.

Example 1. Using Standard Listing Assignments

```
-TAP                  RET
*INPUT: /SOURCE/      RET   Source language resides on
                            file /SOURCE/.

+BINARY: /BIN/        RET   An object code file must be
 NEW FILE             RET   specified.

+TEXT OUTPUT: TEL     RET   The text output will be listed
                            on the teletype subject to
                            the specified conditions of
                            the LIST directive or the
                            default options of LIST.

+ASSEMBLE                   The assembly process is
                            initiated.
```

The assembly process can be terminated by the user at any time by typing two or more ESC's in succession.

Example 2. Overriding Standard Listing Parameters

```
*INPUT: /DECK/        RET
+BINARY: /BDK/        RET
 OLD FILE             RET

+LIST 2               RET   A pass 2 listing that does
+NOLIST MC,COM        RET   not include macro calls or
+ASSEMBLE                   comments will be output to
                            the teletype.
```

Regardless of the listing control parameters that have been given to the assembler, it can be made to begin listing at any time, in either pass, by typing a single ESC. A listing started in this way can be stopped by typing the letter "S".

Table 1. Standard Assignments for TAP Commands

| Command | | Standard Assignment |
|---|---|---|
| INPUT: file name | | None; must be specified by user. |
| BINARY: file name | | None; must be specified by user. |
| TEXT OUTPUT: file name | | Teletype |
| LIST / NOLIST } list of following parameters | | |
| | 1 | Not listed |
| | 2 | Listed |
| | LCT | Listed |
| | BIN | Listed |
| | SRC | Listed |
| | COM | Listed |
| | MC | Listed |
| | ME | Not listed |

See the LIST and NOLIST directives in Chapter 4 for an explanation of the parameters.

# 6. ASSEMBLER ERROR MESSAGES

Upon discovering an error in the syntax of a program being assembled, the assembler will list the statement in question and information about the character of the error. The listing of errors will occur regardless of whether or not regular listing is being done.

## ERROR MESSAGES

Error messages and their interpretations are given below. The first group deals with difficulties found in a single statement.

| Error | Meaning |
|---|---|
| D | Duplicate symbol |
| L | Error in label field; usually an invalid symbol |
| M | Missing field in statement |
| O | Invalid or undefined operation code |
| R | Relocation error in expression |
| S | General syntax error |
| U | Undefined symbol |

If when calling a macro the user fails to deliver an argument required during expansion, the assembler will replace the argument with the character "↑" and will issue an undefined-symbol message at that point.

The second group of error messages, given in Table 2, deal with more complicated difficulties.

Table 2. Error Messages

| Error Message | Meaning |
|---|---|
| SYMBOL TABLE FULL. ERROR CHECK CONTINUES. | Too many symbols and/or operation codes have been defined. Assembly will continue, but no new symbols or operation codes will be recognized. Break the program into subprograms or otherwise reduce the number of symbols present. |
| LITERAL TABLE FULL. FURTHER LITERALS IGNORED. | Similar to the case above. Reduce the number of literals present. |
| MUST ASSEMBLE ABSPGM ON PAPER TAPE. | The bootstrap loader for self-filling absolute assemblies is intended for paper tape only. Designating any other form of output file (except NOTHING and TELETYPE, another form of paper tape) results in this message. It is possible to assemble an absolute program for loading by DDT (see the RELORG directive). |
| INPUT STACK OVERFLOW | There are too many nested macro calls, repeats, and ifs in combination. The stack provided for storing the previous source of input is full. This is a disaster. The program must be reorganized. |
| EOF— END CARD ASSUMED | No END statement was found at the end of the program. The assembler (except for typing his message) takes the same action as it would if it found the END statement. |
| ILLEGAL COMMAND | The assembler does not recognize a command typed in by a user when entering TAP. A new command is required. |
| INPUT FILE NOT TEXT | The input file described to the assembler is not a type 3 file (i.e., symbolic). |
| BAD CHAR | An unrecognizable character (or one otherwise out of place) is found in the text. The character is typed out in octal following the message, replaced by a blank in the text, and assembly continues. |
| EOF IN MACRO DEFINITION | The end of the program is reached, but the assembler is still defining a macro. Look for a missing ENDM. |

Table 2. Error Messages (cont.)

| Error Message | Meaning |
|---|---|
| INPUT STACK UNDERFLOW. | The opposite problem to the one above. Not too serious. Look for the presence of an extra ENDM, ENDR, or ENDF in the program. |
| INPUT BUFFER FULL. | An input statement must be less than 320 characters long. This message occurs when the rule is violated. It usually happens when macros run wild. Look carefully at the program near where the error occurred. |
| TOO MUCH MACRO RECURSION. | Too many nested macro calls have occurred, resulting in filling available pushdown storage. Reorganize program. |
| TOO MUCH RPT RECURSION. | Similar to above. |
| TOO MANY ARGS IN MACRO. | The macro is being called with more arguments than there is space for. Reduce the number of arguments in the call. |
| TOO MANY REPEAT ARGS. | In beginning a repeat block, too many requests for automatic incrementing of symbols have been made. Reorganize the block. |
| STRING STORE EXCEEDED. | No space remains to store new macro definitions or to do repeats. Caution: Old macro definitions are not thrown away. Do not redefine macros indiscriminately. Reorganize program. |
| EOF IN TEXT. | The end of the input file has occurred in the middle of a statement. |

## INTERPRETATION OF THE ERROR LISTING

When an error is listed on any file other than TELETYPE, the single-letter error message (first group above) is listed in the line below at the point where the error was detected.

In the following line, in Figure 5, there are errors in both the label and operand fields.

Along with each error, the value of the location counter is printed out relative to the symbol most recently defined. If

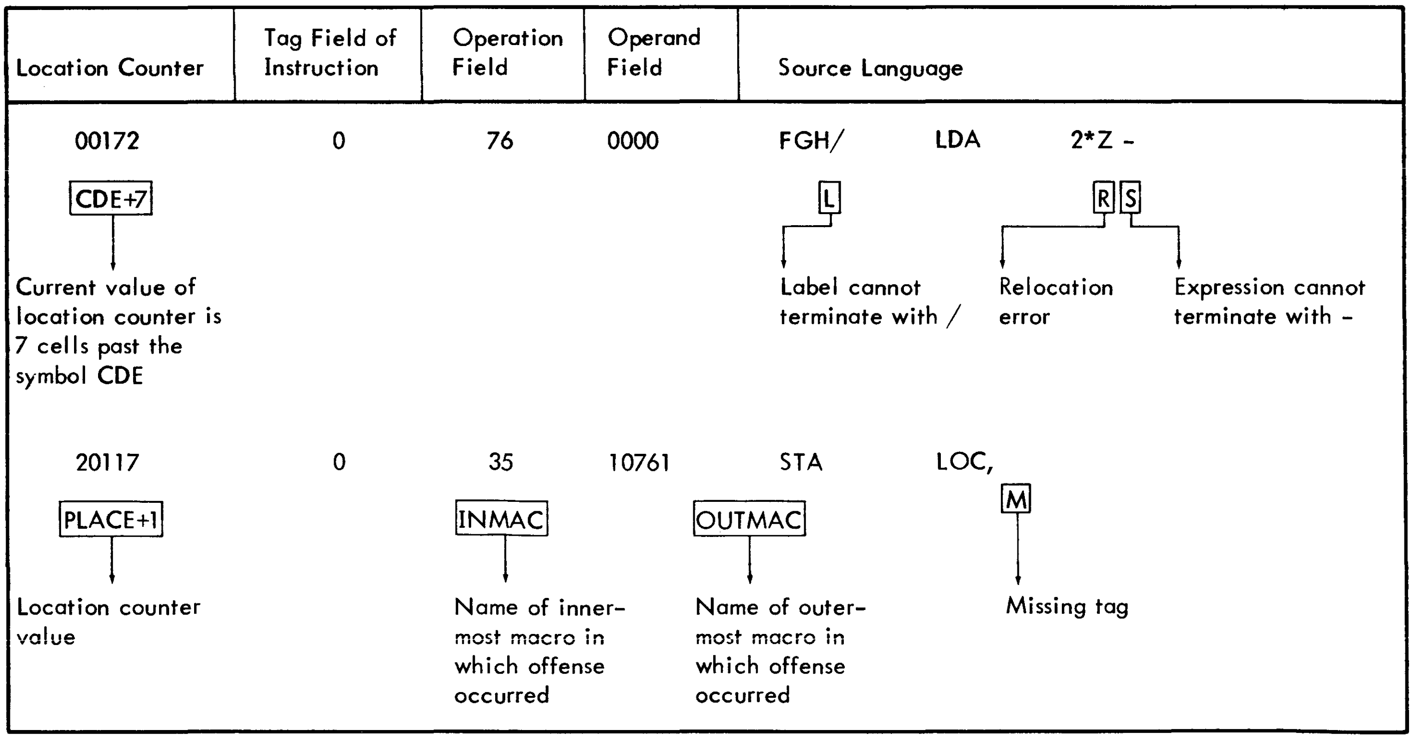

Location Counter
Tag Field of Instruction
Operation Field
Operand Field
Source Language
00172
0
76
0000
FGH/
LDA
2*Z -
CDE+7
L
R
S
Current value of location counter is 7 cells past the symbol CDE
Label cannot terminate with /
Relocation error
Expression cannot terminate with -
20117
0
35
10761
STA
LOC,
PLACE+1
INMAC
OUTMAC
M
Location counter value
Name of inner-most macro in which offense occurred
Name of outer-most macro in which offense occurred
Missing tag

Figure 5. Error Listing Line

the error occurs during macro expansion, the names of the innermost and outermost macros are printed, to indicate where to look for the error. If only one level of macro expansion is involved, then only that name is listed.

In order to save time when error listings are made on the teletype, the single-letter error messages are typed out at the left margin.

# 7. ASSEMBLER BINARY OUTPUT

There are two basic formats for assembler output. The selection depends on whether an assembly is relocatable or absolute.

## RELOCATABLE BINARY OUTPUT

Information in this type of output is divided into variable-length logical records. Each record begins with a control word that defines its type. The first nine bits (bits 0-8) of each control word distinguish it from the others; the remaining bits are used in various ways. The control words are shown in Table 3.

Table 3. Control Words for Relocatable Binary Output

| Control Word | Meaning and Use |
|---|---|
| 1. 000 XXXXX | Binary program follows. Update location counter by amount given in address field of control word. |
| 2. 1XX 00000 | Programmed operator follows. Place branch instruction in location 1XX with address given by current location counter. |
| 3. 200 00000 | End of program. Final record of binary format. |
| 4. 201 XXXXX | Origin of literal table. The origin of the literal table is given in the address field. |
| 5. 202 XXXXX | Change special relocation radix. The new value is given in the address field. |
| 6. 300 00000 | OPD follows. Revert to triplet format (see below). |
| 7. 400 00000 | External symbol definition(s) follows. Revert to triplet format. |
| 8. 500 00000 | Identification record follows. Revert to triplet format. |
| 9. 600 00000 | External symbol usage table follows. Revert to triplet format. |
| 10. 700 00000 | Symbol table follows. Revert to triplet format. |

Control words 2, 3, 4, and 5 cause DDT to take various actions. No additional information is required for these controls; each is complete in itself. This is to be contrasted to control words 6 through 10. Each of the latter prepares DDT to accept a variable-length list of symbols or operation codes. These lists are in so-called "triplet format" because the various symbols and codes are handled as three-word objects. Each list is terminated with a word of all one's.

The contents of the address field of control word 1 are added to DDT's location counter. This control word signals that a binary program (i.e., information to be loaded) follows. The format of a binary program consists of blocks of eight words. Words in each block are either loadable information or control words of types 1 through 5. Controls 6 through 10 also appear in binary programs; when this happens, however, the format immediately reverts to the triplet mode. When the list of triplets is terminated, a new block of eight words is begun. The first word in this block is always a control word of some type.

There are eight different ways in which DDT treats information being loaded. Therefore, it is necessary that a three-bit byte be associated with each word. Each eight-word block of binary program format is preceded, then, by a word of eight three-bit bytes. The association of bytes to words is shown in Figure 6.

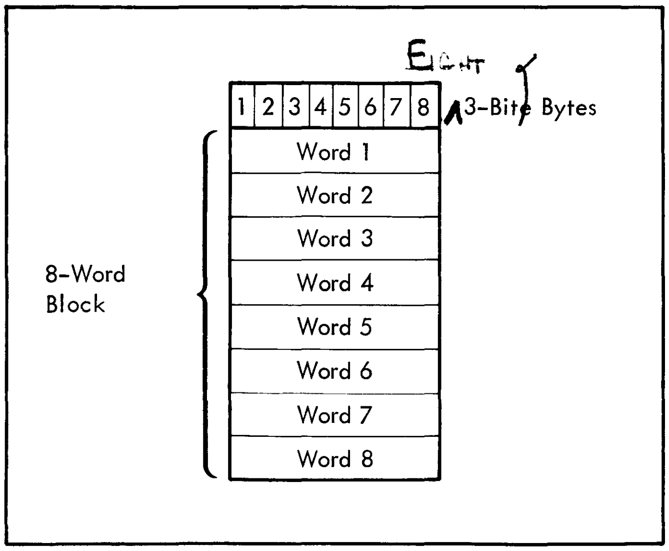
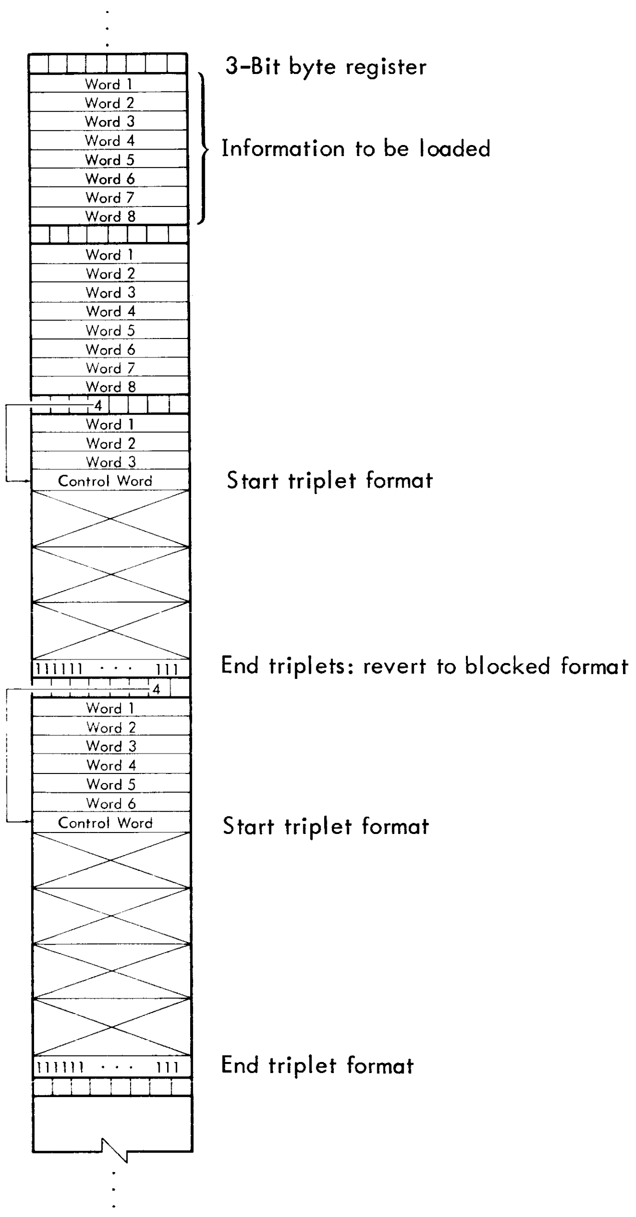


Figure 6. Relocatable Binary Output Format

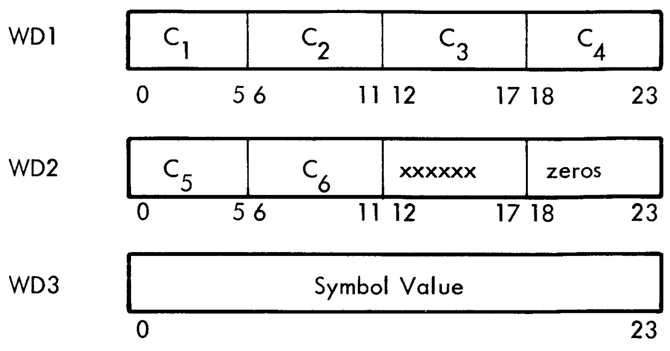
Each 3-bit byte has one of the following values:

| Byte Value | Meaning |
|---|---|
| 0 | Absolute address: load as is |
| 1 | Evaluate address (mod $2^{14}$) from external symbol usage table |
| 2 | Relocate address (mod $2^{14}$) |
| 3 | Special relocation applies |
| 4 | Do not load: interpret word as a control |
| 5 | Derive entire word from external symbol usage table |
| 6 | Relocate entire word (mod $2^{24}$) |
| 7 | Literal reference in address field |

For example, a portion of binary output might have the following appearance.



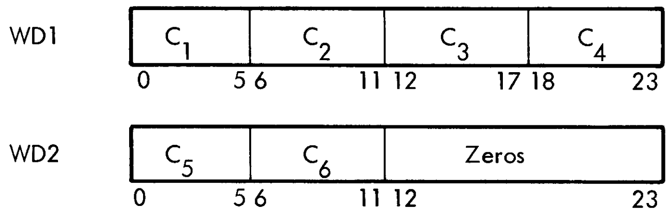
The format of a triplet depends on whether it represents a symbol or a user-defined operation code. For symbols, the following format is used:



$C_1$ through $C_6$ are the six significant characters of the symbol, left-justified, with trailing blanks. Bits 12 through 17 of WD2 are flags having the following meanings.

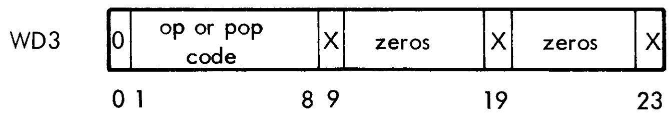
| Bit | Meaning |
|---|---|
| 12 | Relocatable symbol |
| 13 | Duplicate symbol |
| 14 | External symbol |
| 15 | Null symbol |
| 16 | Generated symbol |
| 17 | Equated symbol |

User-defined operation codes have the format



The format of WD3 depends on the type of operation code. The various possibilities are shown below:

1. Class 1 instructions



| Bit | Meaning |
|---|---|
| 9 | Set sign bit of instruction |
| 19 | Operand required |
| 23 | Type number (0 or 1) |

# APPENDIX B. 940 INSTRUCTIONS

| Mnemonic | Operation Code | Function |
|---|---|---|
| **LOAD/STORE** | | |
| LDA | 76 | Load A |
| STA | 35 | Store A |
| LDB | 75 | Load B |
| STB | 36 | Store B |
| LDX | 71 | Load X |
| STX | 37 | Store X |
| EAX | 77 | Copy effective address into X |
| XMA | 62 | Exchange M and A |
| **ARITHMETIC** | | |
| ADD | 55 | Add M to A |
| ADC | 57 | Add with carry |
| ADM | 63 | Add A to M |
| MIN | 61 | Memory increment |
| SUB | 54 | Subtract M from A |
| SUC | 56 | Subtract with carry |
| MUL | 64 | Multiply |
| DIV | 65 | Divide |
| **LOGICAL** | | |
| ETR | 14 | Extract (AND) |
| MRG | 16 | Merge (OR) |
| EOR | 17 | Exclusive OR |
| **REGISTER CHANGE** | | |
| RCH | 46 | Register change |
| CLA | 0 46 00001 | Clear A |
| CLB | 0 46 00002 | Clear B |
| CLAB | 0 46 00003 | Clear AB |
| CLX | 2 46 00000 | Clear X |
| CLEAR | 2 46 00003 | Clear A, B and X |
| CAB | 0 46 00004 | Copy A into B |
| CBA | 0 46 00010 | Copy B into A |
| XAB | 0 46 00014 | Exchange A and B |
| BAC | 0 46 00012 | Copy B into A, clearing B |
| ABC | 0 46 00005 | Copy A into B, clearing A |
| CXA | 0 46 00200 | Copy X into A |
| CAX | 0 46 00400 | Copy A into X |
| XXA | 0 46 00600 | Exchange X and A |
| CBX | 0 46 00020 | Copy B into X |
| CXB | 0 46 00040 | Copy X into B |
| XXB | 0 46 00060 | Exchange X and B |
| STE | 0 46 00122 | Store exponent |
| LDE | 0 46 00140 | Load exponent |
| XEE | 0 46 00160 | Exchange exponents |
| CNA | 0 46 01000 | Copy negative into A |
| AXC | 0 46 00401 | Copy A to X, clear A |

| Mnemonic | Operation Code | Function |
|---|---|---|
| **BRANCH** | | |
| BRU | 01 | Branch unconditionally |
| BRX | 41 | Increment X and branch |
| BRM | 43 | Mark place and branch |
| BRR | 51 | Return branch |
| BRI | 11 | Branch and return from interrupt routine |
| **TEST/SKIP** | | |
| SKS | 40 | Skip if signal not set |
| SKE | 50 | Skip if A equals M |
| SKG | 73 | Skip if A greater than M |
| SKR | 60 | Reduce M, skip if negative |
| SKM | 70 | Skip if A=M on B mask |
| SKN | 53 | Skip if M negative |
| SKA | 72 | Skip if M and A do not compare one's |
| SKB | 52 | Skip if M and B do not compare one's |
| SKD | 74 | Difference exponents and skip |
| **SHIFT** | | |
| RSH | 0 66 00xxx | Right shift AB |
| RCY | 0 66 20xxx | Right cycle AB |
| LRSH | 0 66 24xxx | Logical right shift |
| LSH | 0 67 00xxx | Left shift AB |
| LCY | 0 67 20xxx | Left cycle AB |
| NOD | 0 67 10xxx | Normalize and decrement X |
| **CONTROL** | | |
| HLT, ZRO | 00 | Halt |
| NOP | 20 | No operation |
| EXU | 23 | Execute |
| **BREAKPOINT TESTS** | | |
| BPTx | 0 40 20xx0 | Breakpoint test |
| **OVERFLOW** | | |
| ROV | 0 22 00001 | Reset overflow |
| REO | 0 22 00010 | Record exponent overflow |
| OVT | 0 22 00101 | Overflow test and reset |
| OTO | 0 22 00100 | Overflow test only |

**INTERRUPT**

| Mnemonic | Operation Code | Function |
|---|---|---|
| EIR | 0 02 20002 | Enable interrupts |
| DIR | 0 02 20004 | Disable interrupts |
| AIR | 0 02 20020 | Arm/disarm interrupts |
| IET | 0 40 20002 | Interrupt enabled test |
| IDT | 0 40 20004 | Interrupt disabled test |

**CHANNEL TESTS**

| Mnemonic | Operation Code | Function |
|---|---|---|
| CATW | 0 40 14000 | Channel W active test |
| CETW | 0 40 11000 | Channel W error test |
| CZTW | 0 40 12000 | Channel W zero count test |
| CITW | 0 40 12000 | Channel W inter-record test |

**INPUT/OUTPUT**

| Mnemonic | Operation Code | Function |
|---|---|---|
| EOD | 06 | Energize output to direct access channel |
| MIW | 12 | M into W buffer when empty |
| WIM | 32 | W buffer into M when full |
| PIN | 33 | Parallel input |
| POT | 13 | Parallel output |
| EOM | 02 | Energize output M |
| BETW | 0 40 20010 | W buffer error test |
| BRTW | 0 40 21000 | W buffer ready test |

**SYSPOPS**

| Mnemonic | Operation Code | Function |
|---|---|---|
| BIO | 576 | Block I/O |
| BRS | 573 | Branch to system |
| CIO | 561 | Character I/O |
| CTRL | 572 | Control |
| DBI | 542 | Drum block input |
| DBO | 543 | Drum block output |
| DBI | 544 | Drum word input |
| DWO | 545 | Drum word output |
| EXS | 552 | Execute instruction in system mode |
| FAD | 556 | Floating add |
| FDV | 553 | Floating divide |
| FFAD | 510 | FORTRAN floating add |
| FFDV | 513 | FORTRAN floating divide |
| FFMP | 512 | FORTRAN floating multiply |
| FFSB | 511 | FORTRAN floating subtract |
| FLDF | 514 | FORTRAN floating load |
| FSTF | 515 | FORTRAN floating store |
| GCD | 537 | Get character and decrement |
| GCI | 565 | Get character and increment |
| ISC | 541 | Internal to string conversion (floating output) |
| IST | 550 | Input from specified teletype |
| LAS | 546 | Load from secondary memory |
| LDP | 566 | Load pointer (AB) |
| OST | 551 | Output to specified teletype |
| OUTF | 517 | Skip if no floating overflow |
| QFAD | 500 | Quick floating add |
| QFDI | 505 | Quick floating inverted divide |
| QFDV | 504 | Quick floating divide |
| QFMP | 503 | Quick floating multiply |
| QFNA | 520 | Quick floating negate |
| QFSB | 501 | Quick floating subtract |
| QFSI | 502 | Quick floating inverted subtract |
| QLDF | 506 | Quick floating load |
| QSTF | 507 | Quick floating store |
| SAS | 547 | Store in secondary memory |
| SBRM | 570 | System BRM |
| SBRR | 571 | System BRR |
| SIC | 540 | String to internal conversion (floating input) |
| SKNF | 516 | Skip if floating accumulation negative |
| SKSE | 563 | Skip on string equal |
| SKSG | 562 | Skip on string greater |
| STI | 536 | Simulate teletype input |
| STP | 567 | Store pointer |
| TCI | 574 | Teletype character input |
| TCO | 575 | Teletype character output |
| WCD | 535 | Write character and decrement |
| WCH | 564 | Write character |
| WCI | 557 | Write character and increment |
| WIO | 560 | Word I/O |

# APPENDIX C. MACRO EXAMPLE

The following example makes use of virtually every feature in the macro and conditional assembly processes. It is presented as a demonstration of the power inherent in the use of macros.

The macro COMPILE, when called with an arithmetic expression for its argument, produces assembly language code that computes the value of the expression in a minimum number of steps (subject to the left-to-right scan technique used). COMPILE, in turn, calls a number of other macros. Their functions are explained by comments in the text below.

The COMPILE macro initializes several variables and calls EXPAND (where the more difficult work is done). J is the total number of characters in the expression. K is used to keep track of the recursion level on which the work is being done (EXPAND calls itself recursively when it encounters an opening bracket [ ). AVAIL is the counter for available temporary storage. NPTR and PPTR are stack pointers for the operand and operator stacks, respectively.

```
COMPILE MACRO D;J NCHR D(1);K EQU 0(RET)
 ;AVAIL EQU 1;NPTR EQU -1;PPTR EQU -1(RET)
 EXPAND D(1); ENDM(RET)
```

EXPAND first initializes I, the current character pointer. It then places the value zero on the operator stack (marking its beginning on the current level) and fetches the first operand. Then it sets a switch (G(1)) and goes into a cycle of fetching operators (GETP) and operands (GETN). If the precedence of new operators is less than or equal to that of the previous operators, code is generated. Otherwise the information is stacked and the scan continued.

```
EXPAND MACRO D,G,1;I EQU 1;K EQU K+1(RET)
   STACK 0,P; GETN D(1); SET G(1)(RET)
   CRPT G(1)(RET)
      IF I<J; GETP D(1$I)(RET)
      ELSE;OPTOR EQU 11; RESET G(1)(RET)
      ENDF(RET)
      ;PSTAK EQU PST.($PPTR)(RET)
      CRPT OPTOR/10<PSTAK/10+1; GEN D(1)(RET)
      ENDR(RET)
      IF OPTOR=11;PPTR EQU PPTR-1; RESET G(1)(RET)
      ;K EQU K-1;I EQU I.($K)+I-1(RET)
      ELSE; STACK OPTOR,P(RET)
          IF NPTR>0(RET)
              IF NST.($NPTR-1)<0(RET)
                  IF NST.($NPTR-1)=-1(RET)
                  STA TEMP.($AVAIL)(RET)
                  ELSE; RSH 1; STB TEMP.($AVAIL)(RET)
                  ENDF(RET)
                  ;NST.($NPTR-1) EQU AVAIL(RET)
                  ;AVAIL EQU AVAIL+1(RET)
              ENDF(RET)
          ENDF(RET)
          GETN D(1$I,J)(RET)
      ENDF(RET)
   ENDR(RET)
 ENDM(RET)
```

SET and RESET change the setting of flags. STACK is used to put values and pointers on "stacks" (these are not physical stacks in memory but rather conceptual ones existing in the assembler's symbol table). STACK functions by creating an ordered progression of names and assigning values to the names by means of the EQU directive.

```
SET MACRO D;D(1) EQU 1; ENDM(RET)
RESET MACRO D;D(1) EQU 0; ENDM(RET)
STACK MACRO D;TS EQU D(2).PTR+1(RET)
D(2).PTR EQU TS;D(2).ST.($TS) EQU D(1)(RET)
 ENDM(RET)
```

GETN fetches the next operand. Its complexity is due to the fact that it must recognize symbols (in this example, using the assembler's symbol rules) and numbers. When this recognition is complete it puts in the operand stack a pair of pointers to the head and tail of the operand (i.e., character numbers in the string and a flag bit which denotes whether the object is a symbol or a number). Note that if an opening bracket is encountered, GETN calls EXPAND recursively.

```
GETN MACRO D;TO EQU I; RESET ERROR(RET)
  GETC D(1$I-TO+1)(RET)
  IF CHAR='[';I.($K) EQU I; EXPAND D(1$2,J)(RET)
  ELSE(RET)
     IF LETTER; RESET NUMBER(RET)
     ELSE; SET NUMBER(RET)
     ENDF(RET)
     IF DIGIT; SET SWITCH(RET)
         CRPT SWITCH; GETC D(1$I-TO+1)(RET)
             IF DIGIT(RET)
             ELSF LETTER; RESET SWITCH(RET)
                 IF CHAR='B'; GETC D(1$I-TO+1)(RET)
                     IF LETTER; RESET NUMBER(RET)
                     ELSF DIGIT; RESET NUMBER(RET)
                     ENDF(RET)
                 ELSE; RESET NUMBER(RET)
                 ENDF(RET)
             ELSE; RESET SWITCH(RET)
             ENDF(RET)
         ENDR(RET)
     ELSF LETTER(RET)
     ELSE; SET ERROR(RET)
     ENDF(RET)
     IF NUMbER(RET)
     ELSE; SET SWITCH(RET)
         CRPT SWITCH; GETC D(1$I-TO+1)(RET)
             IF LETTER(RET)
             ELSF DIGIT(RET)
             ELSE; RESET SWITCH(RET)
             ENDF(RET)
         ENDR(RET)
     ENDF(RET)
         IF ERROR; ERROR; STACK 0,N(RET)
         ELSE; STACK TO*1B4+I-2+4B3*NUMBER,N(RET)
         ENDF(RET)
     ;I  EQU I-1(RET)
  ENDF(RET)
  ENDM(RET)
```

GETC's main function is to determine whether a given character is a letter, digit, or other type of character. GETP fetches the next operator. It checks the results, and if valid, sets OPTOR to a value carrying both operator and precedence information.

```
GETC MACRO D;CHAR EQU 'D(1)'(RET)
    ;I EQU I+1;A EQU CHAR>'Z';B EQU CHAR<'A'(RET)
    IF A(OR)B;A EQU CHAR>'9';B EQU CHAR<'0'(RET)
        IF A(OR)B; RESET LETTER; RESET DIGIT(RET)
        ELSE; SET DIGIT; RESET LETTER(RET)
        ENDF(RET)
    ELSE; SET LETTER; RESET DIGIT(RET)
    ENDF(RET)
 ENDM(RET)
```

```
GETP MACRO D; GETC D(1)(RET)
    IF LETTER(OR)DIGIT; ERROR (RET)
    ELSE;A EQU CHAR>11B6;B EQU CHAR<20B6(RET)
        IF A(AND)B;OPTOR EQU OPS.($CHAR/1B6) (RET)
        ELSF CHAR=']';OPTOR EQU 11(RET)
        ELSE;OPTOR EQU -1(RET)
        ENDF (RET)
        IF OPTOR=-1; ERROR;OPTOR EQU 40 (RET)
        ENDF (RET)
    ENDF(RET)
 ENDM (RET)
```

GEN and GENA serve to reconstruct the operands from the string pointers and call generators that actually produce code.

```
GEN MACRO D;R EQU -1;PP2 EQU PST.($PPTR)(RET)
 ;PP3 EQU NST.($NPTR-1)(RET)
 ;PP4 EQU PP3/1B4;PP5 EQU PP3-PP4*1B4(RET)
      IF PP5>4B3;PP5 EQU PP5-4B3; SET LIT1(RET)
      RESET LIT2(RET)
      ELSE; RESET LIT1; RESET LIT2(RET)
      ENDF(RET)
      IF PP3>1B4; GENA D(1),D(1$PP4,PP5)(RET)
   ELSF PP3>0; GENA D(1),TEMP.($PP3)(RET)
      ;AVAIL EQU PP3(RET)
          ELSF PP3=-1; GENA D(1),AREG(RET)
          ELSF PP3=-2; GENA D(1),BREG(RET)
          ENDF(RET)
      ;NPTR EQU NPTR-2; STACK R,N(RET)
   ;PPTR EQU PPTR-1;PSTAK EQU PST.($PPTR)(RET)
 ENDM(RET)
```

```
GENA MACRO D;PP5 EQU NST.($NPTR)(RET)
 ;PP6 EQU PP5/1B4(RET)
 ;PP7 EQU PP5-PP6*1B4(RET)
 IF PP7>4B3;PP7 EQU PP7-4B3; SET LIT2(RET)
 ENDF(RET)
    IF PP5>1B4; GEN.($PP2) D(2),D(1$PP6,PP7)(RET)
    ELSF PP5>0; GEN.($PP2) D(2),TEMP.($PP5)(RET)
 ;AVAIL EQU PP5 (RET)
    ELSF PP5=-1; GEN.($PP2) D(2),AREG (RET)
    ELSF PP5=-2; GEN.($PP2) D(2),BREG(RET)
    ENDF(RET)
 ENDM(RET)
```

GEN20, 21, 30, 31, and 40 are the code producing macros. They reference LIT1 and LIT2 (flags set by GEN and GENA) and call macros TEST, LA, LB, and ST. The purpose of the latter macros is to interpret contents of the A and B registers to prevent superfluous code.

```
GEN20 MACRO D; TEST D(1),D(2),X(RET)
   LA D(X),LIT.($X)(RET)
   IF X=1(RET)
       IF LIT2; ADD =.D(2)(RET)
       ELSE; ADD D(2)(RET)
       ENDF(RET)
   ELSE (RET)
       IF LIT1; ADD =.D(1)(RET)
       ELSE; ADD D(1)(RET)
       ENDF (RET)
   ENDF (RET)
 ENDM (RET)
```

```
GEN21 MACRO D; TEST D(2),X (RET)
  IF X; LA D(2),LIT2(RET)
      IF LIT1; CNA; ADD =.D(1)(RET)
      ELSE; CNA; ADD D(1)(RET)
      ENDF(RET)
  ELSE; LA D(1),LIT1(RET)
      IF LIT2; SUB =.D(2)(RET)
      ELSE; SUB D(2)(RET)
      ENDF(RET)
  ENDF(RET)
 ENDM(RET)
```

```
GEN30 MACRO D; TEST D(1),D(2),X(RET)
   LA D(X),LIT.($X)(RET)
   IF X=1(RET)
       IF LIT2; MUL =.D(2)(RET)
       ELSE; MUL D(2)(RET)
       ENDF(RET)
   ELSE(RET)
       IF LIT1; MUL =.D(1)(RET)
       ELSE; MUL D(1)(RET)
       ENDF (RET)
   ENDF(RET)
    ;R EQU -2
 ENDM
```

```
GEN31 MACRO D; TEST D(2),X(RET)
   IF X; ST D(2$1); LB D(1),LIT1(RET)
   DIV TEMP.($AVAIL) (RET)
   ELSE; LB D(1),LIT1(RET)
       IF LIT2; DIV =.D(2)(RET)
       ELSE; DIV D(2)(RET)
       ENDF(RET)
   ENDF (RET)
 ENDM(RET)
```

```
GEN40 MACRO D; NOP D(1); NOP D(2)(RET)
 ENDM(RET)
```

```
LA MACRO D (RET)
   IF 'D(1)'='AREG' (RET)
   ELSF 'D(1)'='BREG'; LSH 23(RET)
   ELSE (RET)
       IF D(2); LDA =.D(1)(RET)
       ELSE; LDA D(1)(RET)
       ENDF(RET)
   ENDF(RET)
 ENDM(RET)
```

```
LB MACRO D (RET)
   IF 'D(1)'='BREG'(RET)
   ELSE (RET)
       IF 'D(1)'='AREG'(RET)
       ELSE (RET)
           IF D(2); LDA =.D(1)(RET)
           ELSE; LDA D(1)(RET)
           ENDF (RET)
       ENDF (RET)
   RSH 23 (RET)
   ENDF(RET)
 ENDM (RET)
```

```
ST MACRO D(RET)
   IF 'D(1)'='BREG'; RSH 1(RET)
   ENDF(RET)
 ST.D(1$1) TEMP.($AVAIL)(RET)
 ENDM (RET)
```

```
TEST MACRO D;Y NARG;D(Y) EQU 0(RET)
   RPT (Z=1,Y-1)(RET)
       IF 'D(Z$1,4)'='AREG';D(Y) EQU Z(RET)
       ELSF 'D(Z$1,4)'='BREG';D(Y) EQU Z(RET)
       ENDF (RET)
   ENDR (RET)
   IF Y>2 (RET)
       IF D(Y)=0;D(Y) EQU 1 (RET)
       ENDF (RET)
   ENDF (RET)
 ENDM (RET)
```

The following lines establish precedence information for the arithmetic operators.

```
OPS10 EQU 30;OPS11 EQU 20;OPS12 EQU -1
OPS13 EQU 21;OPS14 EQU -1;OPS15 EQU 31
```

When called by the following lines, the macro generates code as shown

```
Call:     COMPILE      X+200*Y (RET)

Result:   LDA          =200
          MUL          Y
          ADD          X

Call:     COMPILE      AB-[C+D]/[E+F] (RET)

Result    LDA          C
          ADD          D
          STA          TEMP1
          LDA          E
          ADD          F
          STA          TEMP2
          LDA          TEMP1
          RSH          23
          DIV          TEMP2
          CNA
          ADD          AB

Call:     COMPILE      A+200*34C21-[DEF/34B-HI*(RET)
                       +[J+20*K]/LM33B -N]/OPQ-22(RET)

Result    LDA          =200
          MUL          34C21
          LSH          23
          ADD          A
          STA          TEMP1
          LDA          DEF
          RSH          23
          DIV          =34B
          STA          TEMP2
          LDA          =20
          MUL          K
          LSH          23
          ADD          J
          MUL          HI
          DIV          LM33B
          CNA
          ADD          TEMP2
          SUB          N
          RSH          23
          DIV          OPQ
          CNA
          ADD          TEMP1
          SUB          =22
```