User's Manual

EXTENDED ASSEMBLER

093-000040-01

ABSTRACT

The DGC Extended Assembler is upward compatible with the DGC Absolute Assembler, providing the additional features of relocation, interprogram communication, conditional assembly, and more powerful number definition. The Extended Assembler may be used under the Real Time Disk Operating System (RDOS), the Real Time Operating System (RTOS), the Stand-alone Operating System (SOS), or in stand-alone operation.

Ordering No. 093-000040-01

Printed in the United States of America
Rev. 01, May 1974

NOTICE

Original Release	October, 1969
First Revision	May, 1974

This manual, 093-000040-01, supersedes 093-000040-00 and is a complete revision of the manual.

NOTATION CONVENTIONS USED IN THIS MANUAL

The formats shown in this manual contain notations that are not part of the assembly language itself but are of the formal language used to describe the Extended Assembler. The notation conventions used in this manual are:

)	A curved arrow represents a carriage return.
UPPER CASE LETTERS	Parts of the format in upper case letters are literal parts of the assembly language and must appear in context exactly as shown in the format.
lower case letters	Parts of the format in lower case letters are variables indicating that the programmer must substitute an appropriate item within a class, such as a symbol, a digit, or an expression. Standard notation variables used in this manual are: usr-sym — a programmer-defined symbol exp — an expression n — a number
{ }	Broken square brackets enclose optional parts of a format. (Parts of a format not in square brackets are required.)
Δ	A delta represents a terminator or break character; it can be any number and combination of spaces, tabulations, and commas.
...	Ellipsis indicates that the preceding field can be repeated.
C( )	A parenthesized quantity preceded by C means contents of . For example, C(AC2) means the contents of the accumulator AC2.

As an example of notation conventions, the source line format for the .END pseudo-op is:

```
.END {Δexp} )
```

The format shows that the programmer must include the characters .END in the source line and may optionally include an expression. If an expression is included in the line, it must be preceded by a terminator such as a space, tab, or comma or series of spaces, tabs and commas. A carriage return must terminate the entire line.

TABLE OF CONTENTS

CHAPTER 5 - INSTRUCTIONS

CHAPTER 6 - PSEUDO-OPS

CHAPTER 1

INTRODUCTION TO THE EXTENDED ASSEMBLER

The Extended Assembler allows the user to write source programs using such familiar characters as letters and numbers. With these characters, symbols can be created that are meaningful to the programmer, but not meaningful to the computer. It is the responsibility of the assembler to process source programs to produce object programs in machine language, meaningful to the computer. To do this, the assembler simply substitutes a numeric code for each symbolic instruction code and a numeric address for each symbolic address. Each line of symbolic instruction is translated into one line of numeric instruction by the assembler.

SPECIAL EXTENDED ASSEMBLER FACILITIES

In addition to providing basic assembly functions, the Extended Assembler includes facilities for relocatable programs, interprogram communication, and conditional assembly.

Relocatable Programs

Relocation is the process of moving a program from one portion of core to another and adjusting the necessary address references so that the program, in its new location, can be executed. This means that the address at which a relocatable program is loaded (its loaded origin) need not be the address at which it is assembled (its assembled origin). A relocatable program can be placed in any suitable area of core. The Relocatable Loader computes the algebraic difference between the loaded and assembled origins of a program and uses it to adjust all addresses that are dependent on the assembled origin so that the program can execute properly from its loaded origin.

The Extended Assembler offers two pseudo-ops that specify that program code is to be relocatable. Normally, the Extended Assembler assembles each source statement at an absolute address until it encounters these pseudo-ops. Refer to "Location Counter Pseudo-ops" in Chapter 6 for more information on these pseudo-ops and relocatable programs.

Interprogram Communication

With the interprogram communication facilities of the Extended Assembler, data, addresses, and constants can be defined in one program and referenced in another. By using the interprogram communication pseudo-ops, explained in Chapter 6, programmers can write related subprograms without concern for the absolute

locations of data and addresses shared by these programs at run time.

Conditional Assembly

The conditional assembly facilities of the Extended Assembler provide the programmer with the capability to include or not to include portions of source code in the assembly process. Depending on the evaluation of an absolute expression, the Extended Assembler will either assemble or bypass a section of source code. The pseudo-ops necessary for controlling the conditional assembly are explained in Chapter 6.

INPUT AND OUTPUT

The Extended Assembler accepts one or more source files written in assembly language as input. It makes two passes (or reads) through the source files to produce output which includes, minimally, a listing of source program errors and, optionally, a source program listing and relocatable binary file. Figure 1-1 illustrates the input to and possible output from the Extended Assembler. Each type of I/O is explained separately below.

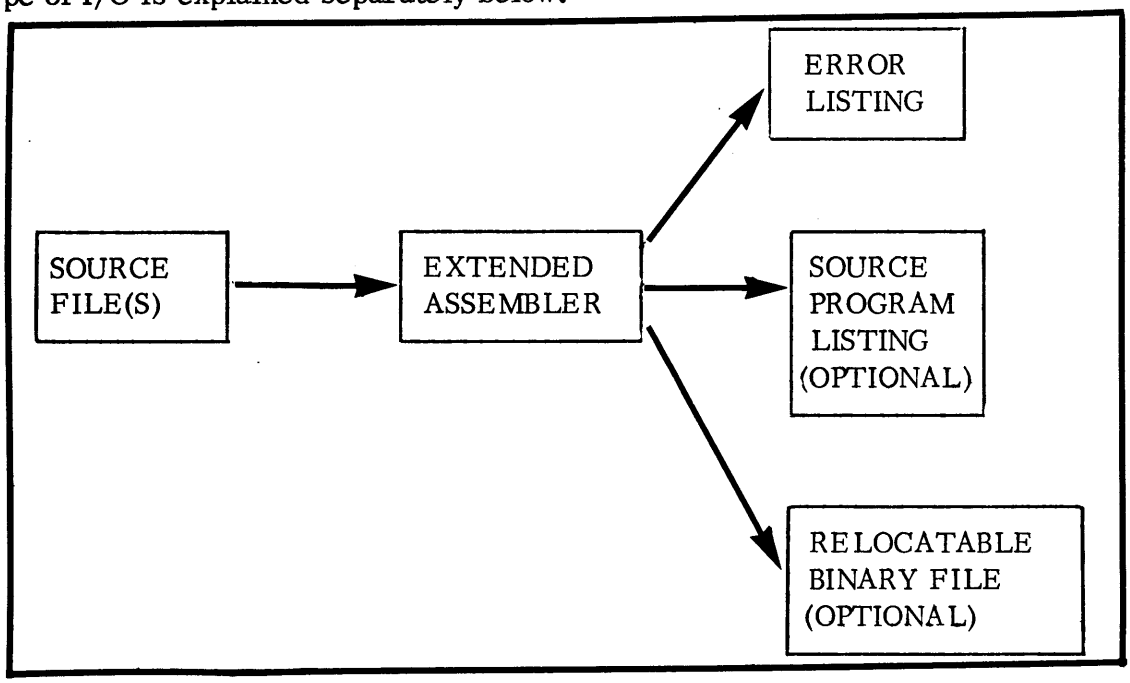


Figure 1-1. Extended Assembler Input and Output

Source Files

The source file input to the Extended Assembler consists of characters that are a subset of the ASCII character set, written as a series of lines. The assembler scans this input line by line and translates it to binary machine language code.

Error Listing

On the first pass over the source input, the assembler outputs a small subset of errors on the teletype. If a program listing is output, the error listing is optional and during the second pass, all errors in the source program are included as part of the program listing. (If a program listing is not output, a separate error listing is produced automatically.) The error listing output on the second pass is a list of only those lines containing errors. The format of these lines is the same as the format of the source program listing.

Source Program Listing

The optional program listing allows the programmer to compare source input to the assembler output. (A sample source program listing is shown in Figure 1-2.) Each line of the program listing contains the following information:

Column(s)	Contents
1-3	Up to three error codes. Each error in input generates a single-letter error code. The first error generates a letter in column 3; the second, in column 2; and the third, in column 3. Additional errors cannot be flagged. The meaning of each error code is given in Appendix B. If no error is detected in the line, these columns contain a two-digit line number, followed by a space.
4-8	Location counter, if relevant; otherwise, these columns are blank.
9	Relocation flag pertaining to the location counter. It can be one of the following:
10-15	Data, if relevant; otherwise, these columns are blank.

Flag	Meaning
blank	absolute
-	page zero relocatable
'	normal relocatable

Column(s)	Contents
16	Relocation flag pertaining to the data field. It can be one of the following:

Flag	Meaning
blank	absolute
-	page zero relocatable
=	page zero byte relocatable
'	normal relocatable
"	normal byte relocatable
$	displacement field is externally defined

17 on	Source line as input.

As part of the program listing, the Extended Assembler produces a cross reference listing of the symbol table, which may include user symbols or both user symbols and semi-permanent symbols. A sample cross reference listing follows.

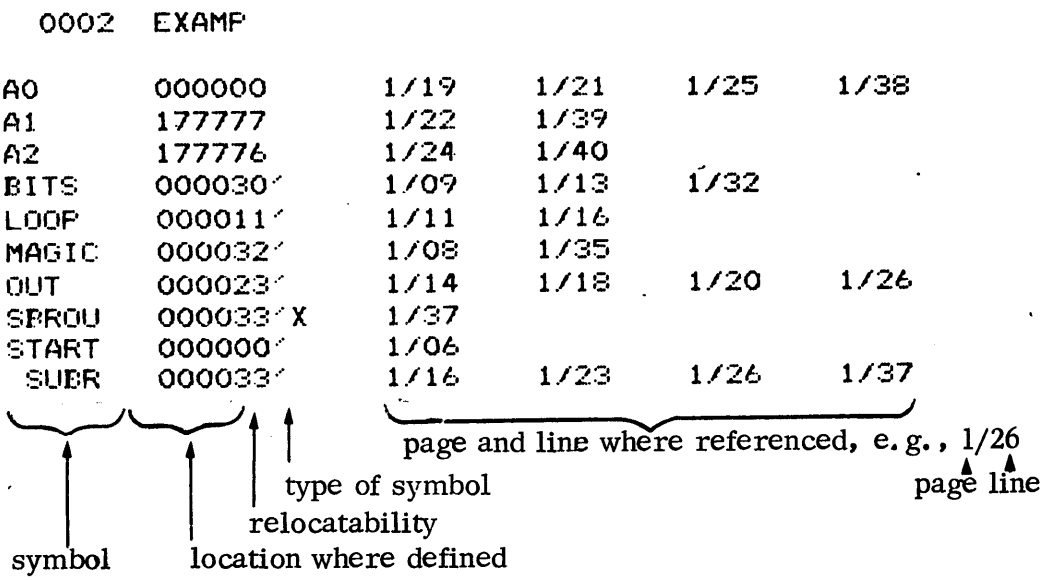


Figure 1-3. Sample Cross Reference Listing

The meaning of the relocatability values given in the cross reference listing is identical to those given for the program listing.

Relocatable Binary File

The optional relocatable binary file is an object file that can be loaded by the relocatable loader and executed. It is a translation of the lines of source program into a special blocked binary code. Most lines of source input translate into a single 16-bit (one-word) binary number for storage in core by the loader. Associated with each number is an address, although it is not necessarily the computer address at which the number will be stored by the relocatable loader.

```
!0002 FTASK
01
02                ; INITIATE A TASK
03
04
05
06
07
08
09
10
                            .TITLE  FTASK
12                          .ENT    FTASK
13                          .EXTN   CTASK
14                          .EXTN   FRET
15                          .EXTD   .CPYL
16                          .NREL
17 00000'000004             4
18 00001'102400 FTASK:      SUB     0,0
19 00002'041614             STA     0,IASM,3
20 00003'006001$            JSR     @.CPYL
21 00004'023613             LDA     0,@PRI,3
22 00005'025611             LDA     1,NAME,3
23 00006'125120             MOVZL   1,1
24 00007'031614             LDA     2,IASM,3
25 00010'151015             MOV#    2,2,SNR
26 00011'000404             JMP     FTAS1
27 00012'033614             LDA     2,@IASM,3
28 00013'151014             MOV#    2,2,SZR
29 00014'125241             MOVOR   1,1,SKP
30 00015'125220 FTAS1:      MOVZR   1,1
31 00016'006407             JSR     @.CTASK
32 00017'000402             JMP     ERR
33 00020'177777             FRET
34 00021'031601 ERR:        LDA     2,FOSP,3
35 00022'021612             LDA     0,ERTN,3
36 00023'041210             STA     0,FRTN,2
37 00024'000020'            FRET
38 00025'177777 .CTASK:     CTASK
39       177611 NAME = FTSTR
40       177612 ERTN = NAME+1
41       177613 PRI = ERTN+1
42       177614 IASM = PRI+1
43
44                          .END
```

Figure 1-2. Sample Source Program Listing

CHAPTER 2

SOURCE PROGRAMS

Assembly language source programs are composed of a series of lines. A line is all characters scanned by the assembler up to a carriage return or form feed. The assembler recognizes several types of lines; each source line must conform to a given structure, depending on its type. In addition, each line must contain only characters in the Extended Assembler character set.

CHARACTER SET

The Extended Assembler accepts the following characters in a source program:

1. Alphabetics A through Z
2. Numerals 0 through 9
3. Special Characters:
   ! " # & * + , -
   . / : ; < = > @
4. Format control and line terminators:
   carriage return, form feed, space, tab

Appendix A contains a table of the Extended Assembler character set and their octal equivalents. As shown in that table, the assembler also accepts lower-case alphabetics, but automatically translates them to their upper-case equivalent.

Three characters are unconditionally ignored by the assembler:

Character	Octal Value
null	000
line feed	012
rubout	177

Any character not in the Extended Assembler character set is flagged with a B (bad character) on the assembly listing.

Source program characters having an incorrect parity are replaced by the assembler with the ASCII character "\". This character is ignored by higher level processing; that is, L\A is processed as LA.

SOURCE LINES

Members of the Extended Assembler character set are combined to form source lines. The majority of source lines affect the generation of a 16-bit value (with relocation properties) that is to occupy a memory location at execution time. Any line of this type is said to produce a storage word. The storage word has a value, usually defined by an expression or instruction, and an address. At assembly time, the address assigned is the contents of the current location counter (LC). The generation of each 16-bit storage word causes the contents of the location counter to be incremented by one. Thus, in general, storage words are assigned to consecutive increasing LC values.

Several types of source lines produce storage words. Others are used to define symbols, control the assembly process, and provide instructions to the assembler. Assembly source lines must be one of the following types:

1. Data
2. Instruction
3. Pseudo-op
4. Equivalence

Data Lines

A data line is one of the simplest in the assembly language. It consists of a single numeric expression.

A data line generates either a 16-bit storage word (if the expression is a single precision integer) or a 32-bit storage word (if the expression is a double precision integer or a floating point number). In fact, data lines provide the only means for storage of double precision and floating point values.

The special character @ (explained in the next chapter) can be used anywhere in a data line to generate a full word indirect address. After the expression is evaluated, the assembler places a 1 in bit 0 (the indirect addressing bit) of the storage word. Thus, for example, all of the following data lines have the same value.

```
1Ø2644
1Ø2644@
2644@
@1322*2
```

Instruction Lines

An instruction line is an instruction mnemonic, or op code, and any required or optional argument fields. All instruction lines generate a 16-bit storage word, which provides an instruction to the assembler, such as load an accumulator, add two accumulators, or increment an accumulator.

Instructions are described in Chapter 5.

Pseudo-op Lines

A pseudo-op line must begin with a permanent symbol (except the symbol .) and may be followed by one or more required or optional arguments. Some pseudo-op lines (such as .NREL and .ZREL) are merely commands to the assembler and do not generate either a storage word or 16-bit value. Others (such as .RDX) generate a 16-bit value, but do not increment the location counter.

Pseudo-ops and pseudo-op line syntax are described in Chapter 6.

Equivalence Lines

One means of assigning a symbolic name to a numeric value is by equivalence. An equivalence line associates a value with a symbol; that symbol can then be used any time the value is required. An equivalence line has the form:

```
usr-sym=exp
```

where usr-sym is a user symbol (conforming to the rules for symbols given in Chapter 3) and exp is an expression or instruction. The symbol to the left (usr-sym) must be previously undefined in pass 1, and the expression at the right must be evaluable in pass 1. Examples of equivalence lines follow.

```
A = 342
B = A/2
INS = ADD# Ø, 1, SKP
```

An equivalence line assembles as a 16-bit value, but does not affect the current location counter.

Labels

Any source program line can contain a label. A label allows the programmer to name a storage word symbolically. Using the label, a programmer can then reference the storage word without regard for its numeric address.

A label is simply a user symbol; it must appear at the beginning of a source line and must be followed by a colon (:). A label must conform to rules established for any symbol, as described in Chapter 3. Like other symbols, a label has a value: its value is that of the current location counter; that is, it is the address of the next storage word assembled. Since some source lines do not generate storage words, this definition is not necessarily associated with the statement that it appears in. The following source line is given the label LOOP:

```
LOOP:  ADD# Ø, 1, SKP )
```

A source line can consist solely of a label. For example:

```
LAB:  )
```

Any source line can have one or more labels, provided all symbols are defined at the beginning of the line. For example:

```
LOOP:LAB1:LAB:  ADD# Ø,1, SKP )
```

Comments

An assembly language program can include comments to facilitate program checkout, maintenance, and documentation. A comment is not interpreted in any way by the assembler and cannot affect the generation of the object program. All comments must be preceded by a semicolon (;). Upon encountering a semicolon, the assembler ignores all subsequent characters up to a carriage return. The following source program lines illustrate the use of comments.

```
; THIS SUBROUTINE CALCULATES THE ABSOLUTE VALUE OF A NUMBER
; IN ACØ
        .TITL    .ABSL
        .ENT     .ABS
        .NREL
.ABS:   MOVL# Ø, Ø, SZC        ;TEST SIGN
        NEG Ø, Ø               ;NEGATE IF NEGATIVE
        JMP Ø, 3
        .END                   ;END OF ABSOLUTE VALUE SUBR.
```

Source Line Formatting

Within broad limits, the programmer is free to determine the format of the source lines for a program. For example, all of the following lines are identical in meaning to the assembler; they differ only in format.

```
LAB:  ADD#   2,3,  SZR              ;SKIP IF SUM = ZERO
LAB:ADD,2,3,SZR#;SKIP IF SUM = ZERO
        LAB:  ADD 2 3 SZR #   ;   SKIP IF SUM = ZERO
```

(The special character # can appear anywhere in a source line.)

A common practice in writing source programs is to divide each line into four columns by means of three tab settings, using the leftmost column for labels, the second column for the beginning of the source line, the third for arguments, and the rightmost for comments. The first example above is in this format. If the listing device is not equipped with automatic tabbing (such as the ASR 33), the Extended Assembler simulates tabs by spacing to the nearest assembler-defined tab position (and always leaving at least one space between fields). Assembler-defined tab positions are at every eight columns; that is, at columns 9, 17, 25 etc.

CHAPTER 3

ATOMS

An atom, the basic unit of the assembly language, is a character or group of characters having special meaning to the assembler. All characters, except those in comments or text strings, are interpreted by the Extended Assembler as an atom or part of an atom. The general classes of atoms,

1. terminators,
2. numbers,
3. symbols, and
4. special atoms

are described on the following pages.

TERMINATORS

Terminators separate numbers and symbols from other numbers and symbols. They can be used as either operators or breaks.

Operators

Operators are a set of terminators used with single precision integers and symbols to form expressions. The operators are:

Arithmetic	B	bit alignment (shift)
	+	addition
	-	subtraction
	*	multiplication
	/	division
Logical	&	AND
	!	inclusive OR

Breaks

Break characters are terminators that are used primarily as separators. These characters are:

space
, (comma)
; (semicolon)
: (colon)
= (equals sign)
horizontal tab
form feed

Space, comma and tab are interchangeable in source code; the assembler handles them identically. These characters are represented by a Δ in this manual. Whenever a Δ is shown, any number or combination of spaces, commas, and tabs can be used.

Colon terminates a label. Equals sign terminates an equivalenced symbol. A semicolon is used to indicate the beginning of a comment and may, optionally, terminate a line of source code. Carriage return and form feed terminate a line of source code.

NUMBERS

The Extended Assembler recognizes three types of numbers:

1. Single precision integer
2. Double precision integer
3. Single precision floating point

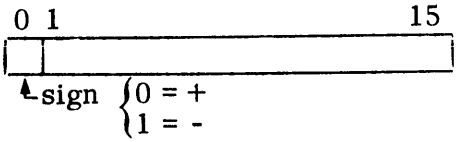
Single Precision Integers

A single precision integer is a string of one or more digits in the current radix. It can be preceded by a minus sign (-) if it is negative or an optional plus sign (+) if it is positive. (An unsigned integer is considered positive.) If the integer is decimal it may be followed by a decimal point. All integers must be terminated by an operator or break atom. Thus, the source code format for a single precision integer is:

```
[ ± ] d[ d ...d ] [ . ] term
```

where d is any digit within the current radix and term is a terminator (operator or break atom). If a decimal point precedes the terminator, the integer is evaluated as decimal. If there is no decimal point, the integer is evaluated in the current radix. (The normal radix, eight, can be changed by the .RDX pseudo-op. Refer to "Radix Pseudo-op" in Chapter 6.)

The Extended Assembler translates all single precision integers to a single word of 16 bits. The integer can be interpreted as signed using twos complement arithmetic in which bit 0 is the sign bit. Bit 0 is 0 if the integer is positive; 1 if it is negative. A single precision integer is represented in core as:



The range of a single precision integer must be 0 through 65535 (decimal) or 0 through 177777 (octal).

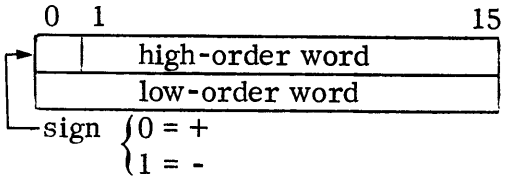
Double Precision Integers

A double precision integer is one or more digits followed by the letter D. It can be preceded by a minus sign or optional plus sign. If the integer is decimal, it may be followed by a decimal point. All double precision integers must be terminated by a break atom. Thus, the source code for a double precision integer is:

```
[ ±] d[ d...d][ . ] D break
```

where d is any digit within the current radix and break is a break atom, typically a space, semicolon, or carriage return. (If a double precision integer is followed by an operator, a format error results.) If a decimal point precedes the D, the integer is evaluated as decimal. If there is no decimal point, the integer is evaluated in the current radix.

The Extended Assembler translates all double precision integers to two contiguous words, the first word of which is the high-order word (the first 16 bits of the integer). Bit 0 of the high-order word contains the sign and bits 1 through 31 contain the magnitude in twos complement notation. A double precision integer is represented in core as:



Double precision integers cannot be combined in expressions; they can be used only in data lines.

Some examples of assembled values of data lines containing double precision integers are:

```
000001  200000D
000000
000003  262147.D
000003
```

Floating Point Numbers

A floating point number is one or more digits followed by (1) a decimal point and at least one more digit and/or (2) the letter E and at least one digit. It can be preceded by a minus sign or optional plus sign. All floating point numbers must be terminated by a break atom. Thus, the source code for a floating point number is:

```
{ ± } d{ d...d } .d { d...d } break
```

or

```
{ ± } d{d...d } .d {d...d} E { ± } d{ d } break
```

or

```
{ ± } d{ d...d} E { + } d{ d } break
```

where d is any digit within the current radix and break is a break atom, typically a space, semicolon, , or carriage return. (If a floating point number is followed by an operator, a format error results.) The number following the E is a decimal power of ten used to evaluate the number. For example, the number .5 can be represented as:

```
+0.5
```

or

```
0.5
```

or

```
5.0E-1
```

The Extended Assembler translates all floating point numbers to two words, using the binary fraction representation described in Appendix C of How to Use the Nova Computers. If a number is specified that is too large or too small to be represented, it is regarded as an error and flagged with an N on the assembly listing.

Floating point numbers cannot be combined in expressions; they can be used only in data lines.

Some examples of floating point numbers in data lines with their assembled values are:

```
Ø4Ø42Ø  1.Ø
ØØØØØØ
Ø4Ø426  3.1415926
Ø41766
 14Ø42Ø  -1EØ
ØØØØØØ
Ø4Ø2ØØ  +5.ØE-1
ØØØØØØ
```

SYMBOLS

A primary function of the Extended Assembler is the recognition and interpretation of symbols. Symbols are used both to direct the action of the assembler and to represent numeric values. A symbol can be written as a series of letters, numbers, or periods. Any other character in a symbol is interpreted as an error and is given a bad character (B) flag in the assembly listing. The following rules also apply to symbols.

1. The first character in a symbol must be an alphabetic (A through Z) or a period (.)
2. A symbol must be unique within the first five characters. Although symbols can have any number of characters, the assembler uses only the first five to differentiate among them. All symbols whose first five characters are the same are indistinguishable to the assembler.
3. A symbol must be terminated by an operator or break atom.

Thus, the character strings

```
A12  .SYB    EXIT  Z
```

are all legitimate symbols while the strings

```
1.27  1BC  LA$B
```

are not: the first two do not begin with a letter or . and the last contains an illegal character. Also, the character strings

```
BITMASK  BITMA.7  BITMA1
```

are treated as the same symbol (BITMA) by the assembler.

Symbols recognized by the Extended Assembler are classified as:

1. Permanent

2. Semi-permanent

3. User

An understanding of the differences among these classes is essential to the use of the assembly language.

Permanent Symbols

Permanent symbols are defined by the assembler and cannot be altered in any way. These symbols are used to direct the assembly process and to represent numeric values of internal assembler variables.

Symbols used to direct the assembly process are called pseudo-ops. Pseudo-ops are used for such purposes as specifying the radix for numeric conversions, setting the location counter, and assembling ASCII text. Pseudo-ops are described in Chapter 6.

The permanent symbol period (.), when used alone, is a special symbol whose value is equal to the current contents of the location counter. Thus, the instruction

```
LDA 3,.+6
```

is equivalent to the instruction

```
LDA 3,6,1
```

Semi-Permanent Symbols

Semi-permanent symbols form a very important class usually thought of as instruction mnemonics or op codes. With appropriate pseudo-ops, symbols can be defined as semi-permanent; their future use implies further syntax analysis. For example, a symbol can be defined as "requiring an accumulator". Use of this symbol causes the assembler to scan for an expression following the symbol. If not found, a format (F) error results. If found, the value of the expression determines the value of the accumulator field bit positions to give a 16-bit value. The assembler instruction set is described in Chapter 5.

Semi-permanent symbols can be saved and used, without redefinition, for all subsequent assemblies. The Extended Assembler contains a number of semi-

Semi-Permanent Symbols (Continued)

permanent symbols defined specifically to conform to the Nova family instruction set. The user can eliminate these symbols and define his own set or, more commonly, can add to the given set. In addition to instruction mnemonics, several semi-permanent symbols are provided by Data General that can be used as operands within expressions. These include the skip mnemonics (such as SKP, SZR, SNR, and SZC) used in arithmetic and logical instructions and device codes (such as TTI, TTO, PTR, and PTP) used in I/O instructions.

User Symbols

The user can define any symbol that does not conflict with permanent or semi-permanent symbols. Symbolic definitions are used for many reasons: to name a location symbolically, to assign a numeric parameter to a symbol, to name external values, to define global values, etc. These user symbols are maintained for the duration of an assembly in a symbol table that is printed after the assembly source listing.

User symbols can be further classified as local or global. Local symbols have a value known only for the duration of the single assembly in which they are defined. Global symbols have a value known at load time; these symbols are used for inter-program communication. During assembly, the user can specify whether or not local symbols are to be included with the binary output.

SPECIAL ATOMS

Three special one-character atoms are available to the Extended Assembler user for:

1. Converting an ASCII character to its 7-bit octal equivalent (").
2. Performing indirect addressing (@), and
3. Setting the no-load bit of an arithmetic or logical instruction (#).

ASCII Character Conversion

A single ASCII character (except null, line feed, and rubout) can be converted to its 7-bit octal equivalent if it is preceded by a double-quote (") character; thus an ASCII character can be represented as a single precision integer. For example:

```
000053          "+          ; ASCII "+"
000055          "-          ; ASCII "-"
```

ASCII Character Conversion (Continued)

The " atom can be used in expressions, for example:

```
000141          "A+40
000172          "Z+40
000112          "A+9.
```

Note that ") assembles as octal 15 and also terminates the line.

Indirect Addressing

Indirect addressing can be specified for a memory reference instruction or a data word if one or more "at" signs (@) are included anywhere in a source program line. If the Extended Assembler encounters an @, it evaluates the memory reference instruction or data word, then sets the indirect addressing bit (bit 5 for a memory reference instruction, bit 0 for a data word) to 1. For example:

```
000004          JMP  RLO
002004          JMP @RLO
```

Setting No-load bit

The programmer can set the no-load bit (bit 12) of an arithmetic or logical instruction by including one or more "pound" signs (#) in the source program line. If the assembler encounters a #, it evaluates the arithmetic or logical instruction, then sets bit 12 to 1, thus preventing the loading of the shifter output.

```
101102          MOVL 0,0,SZC
101112          MOVL #0,0,SZC
```

CHAPTER 4

EXPRESSIONS

An expression is

1. A symbol or single precision integer or
2. A series of symbols and/or single precision integers separated by operators.

Thus, the format of an expression in source code is:

```
{ opn1} opr opn2
```

where $\underline{opn_1}$ and $\underline{opn_2}$ are operands: a symbol, a single precision integer, or another expression evaluating to a single precision integer. opr is one of the following operators:

Operator	Meaning
B	bit alignment (See below.)
+	addition or plus
-	subtraction or minus
*	multiplication
/	division
&	logical AND. The result in a given bit position is 1 if both $\underline{opn_1}$ and $\underline{opn_2}$ are 1 in the corresponding bit position.
!	logical inclusive OR. The result in a given bit position is 1 if $\underline{opn_1}$ and/or $\underline{opn_2}$ is 1 in the corresponding bit position.

Except for the unary operators + and - , an operand must precede every operator. Either unary operator can follow an operator or precede an expression.

If an expression contains an illegal operand (such as an external symbol, instruction mnemonic, double precision number, or floating point number), the source line is given a Z error flag on the assembly listing.

EVALUATION OF EXPRESSIONS

Like a symbol or single precision integer, an expression has a 16-bit value which the assembler computes by performing the indicated operations. This computation, or evaluation, generally proceeds from left to right, one operator at a time. However, bit alignment is always performed before any other operation in an expression.

If two operators are contiguous, the assembler assumes a zero operand between them. Hence, the expression:

```
A+-B
```

is equivalent to

```
A+0-B
```

or simply A-B; while the expression

```
A*-B
```

is equivalent to

```
A*0-B
```

or -B. (However, the expression -B*A correctly multiplies A by -B.)

During expression evaluation, no check is made for overflow.

BIT ALIGNMENT OPERATOR

The Extended Assembler bit alignment operator provides a facility for right justification of a single precision integer on a bit boundary. The bit alignment operator is used in source code as follows:

```
n B d
```

where n is a number in the current radix to be aligned and d is a decimal number specifying the rightmost bit at which n is to be aligned. The aligned number is given the value:

$$(n)_r * 2^{(15-d)}$$

where r is the current radix.

100000	1B0
000002	1B14
002400	12B8

RADIX 8

RELOCATION PROPERTIES OF EXPRESSIONS

Associated with each operand of an expression is its relocation property. The relocation property of the operands in an expression, in turn, determines the relocation property of the expression. Expressions described thus far have had absolute operands and the result of their evaluation has been absolute.

An operand can have one of several relocation properties:

absolute

page zero relocatable

normal relocatable

page zero byte relocatable

normal byte relocatable

An absolute operand is one whose address is fixed; its loaded address is the same as its assembled address.

Page zero relocatable operands are relocatable, yet must reside in page zero; normal relocatable operands can be relocated anywhere in core except page zero. These relocatable operands are converted to absolute during the loading process by the addition of a relocation constant. The relocatable loader maintains two relocation constants, a zero relocation (C_z) and a normal relocation (C_n) constant, to "fix" the addresses of relocatable values.

Byte relocatable operands are page zero or normal relocatable storage words that act as byte pointers: bits 0-14 of the pointer contain an address and bit 15 specifies the byte to be operated on. (Refer to page 2-21 in How to Use the Nova Computers.) A byte pointer of this kind can be formed simply by doubling an address, and can be retrieved and regenerated by a shifting operation. The loader adds either $2C_z$ or $2C_n$ to each byte relocatable value to convert it to absolute.

During loading, the relocatable loader can add one and only one of five possible constants to a word: 0, C_z, C_n, $2C_z$, or $2C_n$. This has two implications when relocatable operands are combined in expressions:

1. Although the Extended Assembler permits the combining of page zero and normal relocatable operands in expressions, the operands must be such that either the page zero or the normal relocatable operands "cancel out". For example, the expression

   $$Z_1 + N_1 - Z_2 + N_2 - N_3$$

   (where Z_i represent page zero relocatable operands; N_i represent normal relocatable operands) is legal, but

   $$Z_1 + Z_2 + N_1$$

   is not.

2. Loader modification of an address by more than twice a relocation factor is illegal.

The following list indicates legal combinations of absolute and relocatable operands in expressions. Any expression not in that list, or not reducing to a form in that l list, is flagged as a relocation (R) error in the assembly listing. (In the following list, a is an absolute operand, r is a relocatable operand.)

Expression	Relocation Property
a + a	absolute
a - a	absolute
r + a	relocatable
r - a	relocatable
r + r	byte relocatable
r - r	absolute
a * a	absolute
a / a	absolute

Expression	Relocation Property
a & a	absolute
a ! a	absolute
2 * r	byte relocatable

In addition, the following expressions are unconditionaly illegal:

r*r	a&r
a/r	r!r
r&r	a!r

In the example following, A is defined as an absolute value and R as a relocatable value.

```
        000002  A=2
                .NREL
  00000 ' 000020' .+20
  00001 ' 000000  R: 0
        000002' S= R+1      ;RELOCATABLE + ABSOLUTE = RELOCATABLE
  00002 ' 000001  A/A       ;ABSOLUTE / ABSOLUTE = ABSOLUTE
  00003 ' 000002" R+R       ;RELOC + RELOC = BYTE RECOCATABLE
  00004 ' 177777' R-A       ;RELOCATABLE - ABSOLUTE = RELOCATABLE
  00005 ' 000001  S-R       ;RELOCATABLE - RELOCATABLE = ABSOLUTE
R00006 ' 000000' A!R        ;ILLEGAL. RESULTS IN RELOCATION ERROR
```

CHAPTER 5

INSTRUCTIONS

An instruction is the assembly of one or more fields, initiated by the occurrence of a semi-permanent symbol(called the "instruction mnemonic") to form a 16-bit value. Fields in an instruction can be separated by a space, comma, or tab and must conform in number and type to the field requirements for the type of semi-permanent symbol.

The DGC family of computers recognizes seven basic types of instructions. Each type has a corresponding pseudo-op enabling definition of semi-permanent symbols within the type. (Refer to "Symbol Table Pseudo-ops" in Chapter 6 for additional information on these pseudo-ops.) Instructions fall into the following seven types:

1.	Arithmetic and logical	(Instruction mnemonic defined by .DALC)
2.	Memory reference without accumulator	(Instruction mnemonic defined by .DMR)
3.	Memory reference with accumulator	(Instruction mnemonic defined by .DMRA)
4.	Input/output without accumulator	(Instruction mnemonic defined by .DIO)
5.	Input/output with accumulator	(Instruction mnemonic defined by .DIOA)
6.	Input/output without device code	(Instruction mnemonic defined by .DIAC)
7.	Input/output without argument fields	(Instruction mnemonic defined by .DUSR)

The instructions corresponding to these instruction types are described fully in How to Use the Nova Computers. The syntax required for each instruction type is given on the following pages. The semi-permanent symbols listed for each type are those defined by Data General.

ARITHMETIC AND LOGICAL (ALC) INSTRUCTIONS

An arithmetic and logical (ALC) instruction is implied when the instruction mnemonic is one of the following:

COM	MOV	ADC	ADD
NEG	INC	SUB	AND

The format of the source program instruction is:

```
alc-mnemonic [carry] [shift]  Δ source-ac Δ destination-ac [Δskip]
```

where:

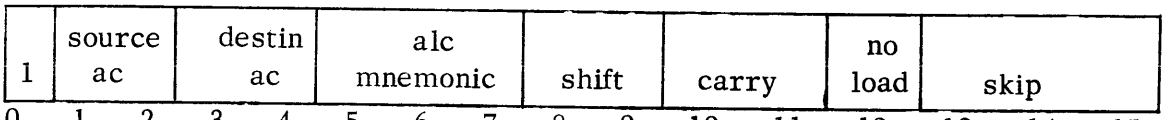
alc-mnemonic	is one of the eight semi-permanent symbols listed above.
carry	is an optional Carry bit mnemonic.
shift	is an optional shift mnemonic.
source-ac	is a 0, 1, 2, or 3, indicating the accumulator to be used as the source accumulator.
destination-ac	is a 0, 1, 2, or 3, indicating the accumulator to be used as the destination accumulator.
skip	is an optional skip mnemonic.

In addition, the atom # can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 12, the no-load bit, thus inhibiting loading of shifter output.

Figure 5-1 shows the assembled ALC instruction as well as the bit pattern and effect of the instruction, shift, Carry, and skip mnemonics.

Examples of ALC instructions follow

```
151112          MOVL# 2,2,SZC
137000          ADD 1,3
```



ALC Mne.	Bits 5 6 7	Effect *
COM	0 0 0	Places logical complement of C (source-ac) in destination-ac
NEG	0 0 1	Places negative of C (source-ac) in destination-ac
MOV	0 1 0	Moves C (source-ac) to destination-ac
INC	0 1 1	Places C (source-ac)+1 in destination-ac
ADC	1 0 0	Adds logical complement of C (source-ac) to C (destination-ac)
SUB	1 0 1	Subtracts C (source-ac) from C (destination-ac)
ADD	1 1 0	Places sum of C (source-ac) and C (destination-ac) in destination-ac
AND	1 1 1	Places logical AND of C (source-ac) with C (destination-ac) in destination-ac

Shift Mne.	Bits 8 9	Effect	Carry Mne.	Bits 10 11	Effect
L	0 1	Shifts word left one bit	Z	0 1	Sets Carry to zero
R	1 0	Shifts word right one bit	O	1 0	Sets Carry to one
S	1 1	Swaps bytes of word	C	1 1	Complements current state of Carry

Skip Mne.	Bits 13 14 15	Effect
SKP	0 0 1	Skips next sequential word (NSW) unconditionally
SZC	0 1 0	Skips NSW on zero Carry
SNC	0 1 1	Skips NSW on nonzero Carry
SZR	1 0 0	Skips NSW on zero result
SNR	1 0 1	Skips NSW on nonzero result
SEZ	1 1 0	Skips NSW on zero Carry or result
SBN	1 1 1	Skips NSW on zero Carry and result

*Refer to Chapter 2 of How to Use the Nova Computers for the effect of these instructions on the Carry bit.

Figure 5-1. Assembly of ALC Instruction

MEMORY REFERENCE (MR) INSTRUCTIONS WITHOUT ACCUMULATOR

A memory reference (MR) instruction without an accumulator field is implied when the instruction mnemonic is one of the following:

JMP ISZ

JSR DSZ

The format of the source program instruction is:

```
mr-mnemonic Δ displacement Δ mode

                or
mr-mnemonic Δ address
```

where:

mr-mnemonic — is one of the four semi-permanent symbols listed above.

displacement — is any legal expression evaluating to an 8-bit integer in the range of -200_8 through $+177_8$.

mode — is a 1, 2, or 3, indicating an explicit mode for forming an effective address (E), as follows:

mode	Formation of Effective Address (E)
1	Addressing is based on the contents of location counter: E=C(LC) + displacement and, therefore $C(LC) - 200_8 \leq E \leq C(LC)+177_8$
2	Addressing is based on the contents of AC2: E = C(AC2) + displacement and, therefore $C(AC2)-200_8 \leq E \leq C(AC2)+177_8$
3	Addressing is based on the contents of AC3: E = C(AC3) + displacement and, therefore $C(AC3)-200_8 \leq E \leq C(AC3)+177_8$

address — is any legal expression evaluating to an 8-bit integer in one of the following ranges:

1. 0 through $+377_8$ (for page zero addressing) : addressing is direct and E = address.

2. C(LC)-200_8 through C(LC)+177_8 (for LC-relative addressing): addressing is based on the contents of the location counter and E=C(LC)+address.

In addition, the atom @ can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 5, the indirect addressing bit. Thus, the effective address in the instruction is a pointer to another location, which may, in turn, contain an indirect address.

If only address is specified, the assembler determines if this address is in page zero (0 through 377_8) or within 177_8 words of the location counter. If the address is in page zero, bits 6 and 7 of the instruction word are set to 00 and the displacement field is set as follows:

1. If the address is absolute, the displacement field is set to address.

2. If the address is page zero relocatable (that is, assembled with the .ZREL pseudo-op), the displacement field is set to address with page zero relocation and the line is flagged with a - in column 16 of the source program listing.

3. If the address is an external displacement (that is, assembled with the .EXTD pseudo-op), the displacement is set to the external's value and the line is flagged with a $ in column 16 of the source program listing. (The value of an external is given in the cross reference listing following the program listing.)

If address is within 177_8 words of the contents of the location counter, bits 6 and 7 are set to 01 and addressing is based on the current contents of the location counter (as in addressing mode 1). The displacement field of the instruction word is set to: address - C(LC).

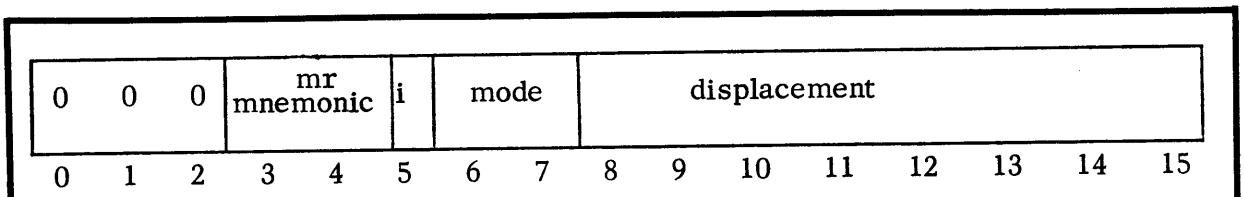
If address or the evaluation of displacement to an address does not produce an effective address within the appropriate range, an addressing (A) error is reported.

Figure 5-2 shows the assembled MR instruction as well as the bit pattern and effect of the instruction mnemonics. Figure 5-3 illustrates how effective addresses are formed.

Examples of MR instructions and their assembled address and value follow.

```
00003 010456 SORT1:  ISZ     STAK
                     :
00046 004735         JSP     SORT1
                     :
00061 000000 STAK:   0
```

```
00002 001400         JMP 0,3
```



MR Mnemonic	Bits 3 4	Effect
JMP	0 0	Jumps to effective address (loads effective address into LC)
JSR	0 1	Jumps to subroutine at effective address; loads C(LC)+1 into AC3
ISZ	1 0	Increments contents of effective address; skips next sequential word (NSW) if result is zero
DSZ	1 1	Decrements contents of effective address; skips NSW if result is zero

Figure 5-2. Assembly of MR Instruction without Accumulator

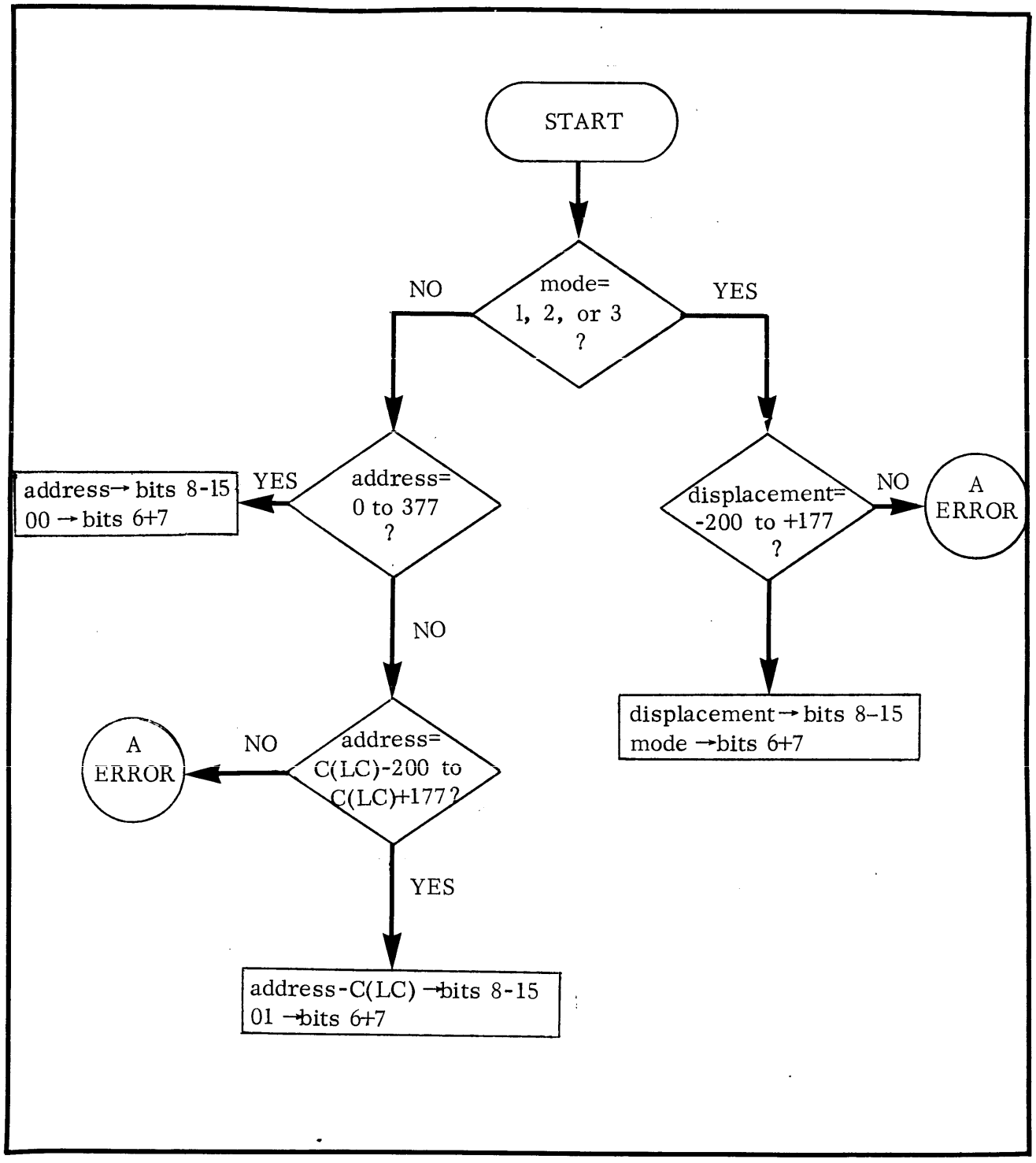


Figure 5-3. Formation of Effective Address for MR Instruction

MEMORY REFERENCE (MR) INSTRUCTIONS WITH ACCUMULATOR

A memory reference (MR) instruction with an accumulator field is implied when the instruction mnemonic is one of the following:

LDA STA

The format of the source program instruction is:

mra-mnemonic Δ accumulator Δ displacement Δ mode
or
mra-mnemonic Δ accumulator Δ address

where:

mra-mnemonic — is a semi-permanent symbol: LDA or STA.

accumulator — is a 0, 1, 2, or 3, indicating the accumulator to receive or supply the data.

displacement, mode, and address are the same as for MR instructions without an accumulator field.

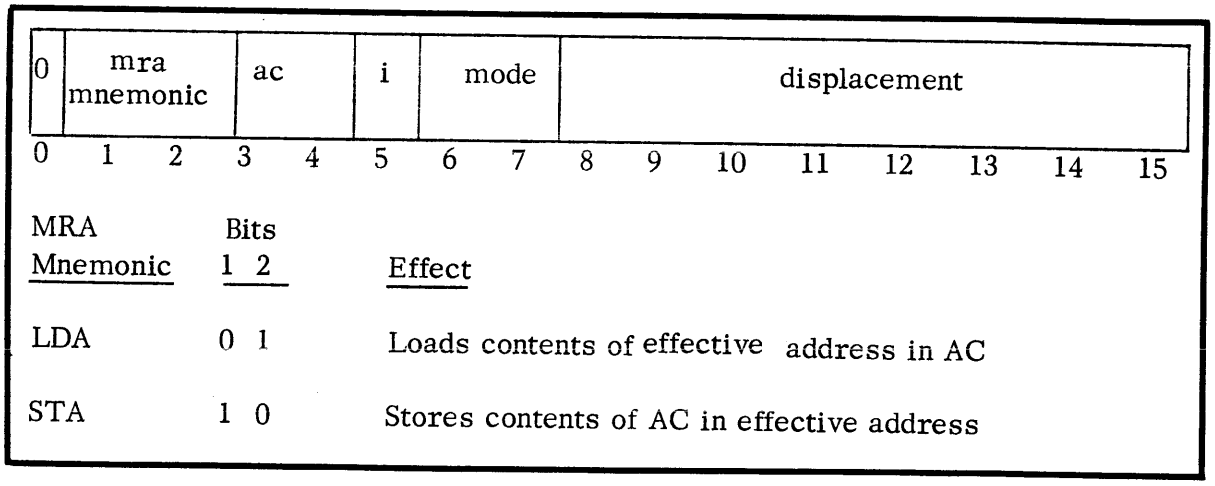
In addition, the atom @ can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 5, the indirect addressing bit.

Figure 5-4 shows the assembled MR instruction as well as the bit pattern and effect of the instruction mnemonics.

Examples of MR instructions follow.

```
00014'040434            STA 0, FB11
00015'024432            LDA 1, FB10
                  :
00047'000000  FB10:  0
00050'000000  FB11:  0
```

```
00014'025001            LDA 1,1,2
00015'035003            LDA 3,3,2
00016'031002            LDA 2,2,2
```



MRA Mnemonic	Bits 1 2	Effect
LDA	0 1	Loads contents of effective address in AC
STA	1 0	Stores contents of AC in effective address

Figure 5-4. Assembly of MR Instruction with Accumulator

I/O INSTRUCTIONS WITHOUT ACCUMULATOR

An input/output instruction without an accumulator field is implied when the instruction mnemonic is one of the following:

NIO	SKPBN	SKPDN
	SKPBZ	SKPDZ

The format of the source program instruction is:

io-mnemonic [busy/done] Δ device-code

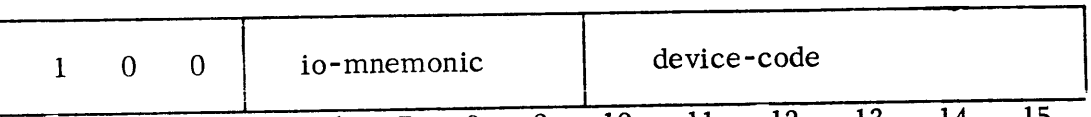
where:

io-mnemonic	is one of the five semi-permanent symbols listed above.
busy/done	is an optional Busy/Done bit mnemonic (NIO instruction only).
device-code	is any legal expression evaluating to an integer that specifies a device. (Refer to Appendix E of How to Use the Nova Computers for legal device codes.)

Figure 5-5 shows the assembled I/O instruction as well as the bit pattern and effect of the instruction and Busy/Done mnemonics.

Examples of I/O Instructions without an accumulator follow:

```
000112          NIOS 12
000112          NIOS PTR
000177          NIOS CPU
000177          NIOS 77
```



I/O Mnemonic	Bits 5 6 7 8 9	Effect
NIO	0 0 0 0 0	No operation
SKPBN	1 1 1 0 0	Skips next sequential word (NSW) if Busy is 1
SKPBZ	1 1 1 0 1	Skips NSW if Busy is zero
SKPDN	1 1 1 1 0	Skips NSW if Done is 1
SKPDZ	1 1 1 1 1	Skips NSW if Done is zero

The following one-character mnemonics can be appended to the NIO instruction only.

Busy/Done Mnemonic	Bits 8 9	Effect
S	0 1	Clears Done and sets Busy, starting device (If device=77 or CPU, sets Interrupt On flag)
C	1 0	Clears Done and Busy, idling device (If device=77 or CPU, clears Interrupt On flag)
P	1 1	Sets Done and Busy, pulsing I/O bus control line (If device=77 or CPU, has no effect)

Figure 5-5. Assembly of I/O Instruction without Accumulator

I/O INSTRUCTIONS WITH ACCUMULATOR

An input/output instruction with an accumulator field is implied when the instruction mnemonic field is one of the following:

DIA DIB DIC

DOA DOB DOC

The format of the source program instruction is:

```
ioa-mnemonic [ busy/done ] Δ accumulator Δ device-code
```

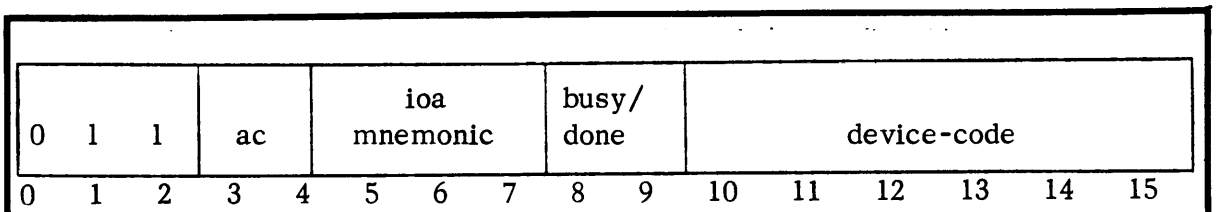
where:

ioa-mnemonic	is one of the six semi-permanent symbols listed above.
busy/done	is an optional Busy/Done bit mnemonic.
accumulator	is a 0, 1, 2, or 3, indicating the accumulator to receive or supply the data.
device-code	is any legal expression evaluating to an integer that specifies a device. (Refer to Appendix E of How To Use the Nova Computers for legal device codes.)

Figure 5-6 shows the assembled I/O instruction as well as the bit pattern and effect of the instruction and Busy/Done mnemonics.

Examples of I/O instructions with an accumulator field follow.

```
074177      DIA 3,CPU
071514      DIAS 2,PTR
063077      DOC 0,77
```



IOA Mnemonic	Bits 5 6 7	Effect
DIA	0 0 1	Inputs data in A buffer of device to AC
DOA	0 1 0	Outputs data in AC to A buffer of device
DIB	0 1 1	Inputs data in B buffer of device to AC
DOB	1 0 0	Outputs data in AC to B buffer of device
DIC	1 0 1	Inputs data in C buffer of device to AC
DOC	1 1 0	Outputs data in AC to C buffer of device

Busy/Done Mnemonic	Bits 8 9	Effect
S	0 1	Clears Done and sets Busy, starting device (If device=77 or CPU, sets Interrupt On flag)
C	1 0	Clears Done and Busy, idling device (If device=77 or CPU, clears Interrupt On flag)
P	1 1	Sets Done and Busy, pulsing I/O bus control line (If device=77 or CPU, has no effect)

Figure 5-6. Assembly of I/O Instruction with Accumulator

I/O INSTRUCTIONS WITHOUT DEVICE CODE

Certain commonly used I/O instructions have been defined with a device code of CPU. These instructions require an accumulator field, but no device code field. An I/O instruction without a device code field is implied when the instruction mnemonic is one of the following:

READS INTA MSKO

The format of the source program instruction is:

```
iac-mnemonic Δ accumulator
```

where:

iac-mnemonic — is one of the three semi-permanent symbols listed above.

accumulator — is a 0, 1, 2, or 3, indicating the accumulator to receive or supply the data.

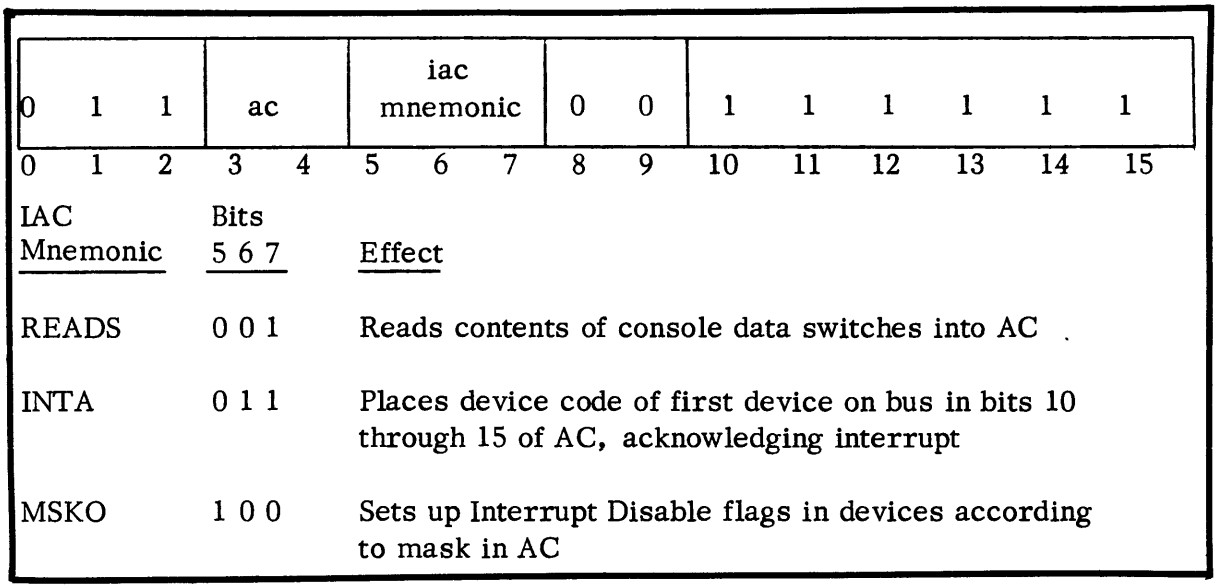
These instructions are equivalent to the following I/O instructions.

I/O Instruction Without Device Code	Equivalent Instruction
READS accumulator	DIA accumulator, CPU
INTA accumulator	DIB accumulator, CPU
MSKO accumulator	DOB accumulator, CPU

Figure 5-7 shows the assembled I/O instruction as well as the bit pattern and effect of the instruction mnemonics.

Examples of I/O instructions without a device code follow.

```
074177          READS 3
061477          INTA 0
```



IAC Mnemonic	Bits 5 6 7	Effect
READS	0 0 1	Reads contents of console data switches into AC
INTA	0 1 1	Places device code of first device on bus in bits 10 through 15 of AC, acknowledging interrupt
MSKO	1 0 0	Sets up Interrupt Disable flags in devices according to mask in AC

Figure 5-7. Assembly of I/O Instruction without Device Code

I/O INSTRUCTIONS WITHOUT ARGUMENT FIELDS

Four commonly used I/O instructions have been defined as semi-permanent symbols that do not require any argument field:

IORST INTDS

INTEN HALT

The equivalent I/O instruction, assembled octal value, and effect of these instructions are given below.

I/O Instruction Without Argument	Equivalent Instruction	Octal Value	Effect
IORST	DICC Ø, CPU	062677	Clears all I/O devices and Interrupt On flag; resets clock to line frequency
INTEN	NIOS CPU	060177	Sets Interrupt On flag, enabling interrupts
INTDS	NIOC CPU	060277	Clears Interrupt On flag, disabling interrupts
HALT	DOC Ø, CPU	063077	Halts the processor

I/O INSTRUCTIONS WITHOUT ARGUMENT FIELDS (contd.)

Examples of these instructions follow.

```
062677          IORST
063077          HALT
060177          INTEN
```

CHAPTER 6

PSEUDO-OPS

Pseudo-ops are permanent symbols that direct the assembly process. Extended Assembler pseudo-ops are grouped into the following eight categories, according to the functions they perform:

1. Title pseudo-op
2. Radix pseudo-op
3. Symbol table pseudo-ops
4. Location counter pseudo-ops
5. Interprogram communication pseudo-ops
6. Text pseudo-ops
7. Conditional assembly pseudo-ops
8. File terminating pseudo-ops

Each category and pseudo-op is explained in this chapter in the above order. Appendix C gives a summary of all pseudo-ops, including their function, syntax, and use.

In general, pseudo-op lines can appear anywhere within a source program; however, the title and entry pseudo-ops must be declared before any statement generating object data.

TITLE PSEUDO-OP

The debugger, relocatable loader, and library file editor all use titles to identify relocatable binary output. For this reason, the Extended Assemlber includes the .TITL pseudo-op, which allows the user to specify a title for a program. The syntax for the .TITL pseudo-op is:

```
.TITL Δusr-sym
```

TITLE PSEUDO-OP (Contd.)

The symbol specified with the pseudo-op becomes the program title and is printed at the top of every listing page. The symbol need not be unique from other symbols defined by the program.

The .TITL source line must appear before any statement generating object data in a source program. If this pseudo-op is omitted, the program assumes the default title of .MAIN. If more than one .TITL pseudo-op is specified in a program, the last one before a line generating data is used as the program title.

The following pseudo-op line names the program SQRT.

```
.TITL    SQRT
```

RADIX PSEUDO-OP

At the beginning of each assembly pass, the Extended Assembler interprets integers not containing a decimal point as octal. The user can change the radix of a number by including the .RDX pseudo-op in the source program. The format of the pseudo-op is:

```
.RDX Δ exp
```

where exp is evaluated in decimal. Its range must be between 2 and 10 (decimal); that is, the range of exp must be:

$$2 \leq \underline{exp} \leq 10$$

When the assembler encounters a .RDX pseudo-op, it converts every integer thereafter to the specified radix. Note that regardless of the radix used, any number containing a decimal point is interpreted as decimal. This feature allows the programmer to combine decimal numbers in expressions with numbers of other radices.

Examples of the use of the .RDX pseudo-op follow.

```
          .RDX 2
000005    101
000152    101+101.
          .RDX 8
000101    101
000246    101+101.
          .RDX 10
000145    101
000312    101+101.
```

SYMBOL TABLE PSEUDO-OPS

The symbol table, maintained by the Extended Assembler, is a list of all semi-permanent symbols defined by Data General (such as the instruction mnemonics) and permanent symbols (such as pseudo-ops). In addition, the symbol table can contain semi-permanent symbols defined by the user. The Extended Assembler includes eight pseudo-ops that allow users to add or delete semi-permanent symbols in the symbol table.

The symbol table pseudo-ops define a user symbol as a semi-permanent one and assign a value to it. The pseudo-ops have the general form:

```
pseudo-op Δ symbol = expression
```

where pseudo-op is one of the following:

.DALC .DMRA .DIOA .DUSR

.DMR .DIO .DIAC

(An eighth pseudo-op, .XPNG, has a unique form and is described later.) The pseudo-op defines symbol as a semi-permanent symbol; its value is the value of expression.

Each symbol table pseudo-op, except .DUSR, implies a certain type of instruction. Thus, once defined, the semi-permanent symbol must be used with expressions appropriate to the format required. For example, the pseudo-op .DALC defines a symbol that is an implied arithmetic and logical instruction mnemonic and which requires expressions following the symbol that are entered into those bit fields that would represent in an ALC the source and destination accumulations and the optional skip field. The symbol table pseudo-ops can be used to define semi-permanent symbols of the following instruction types.

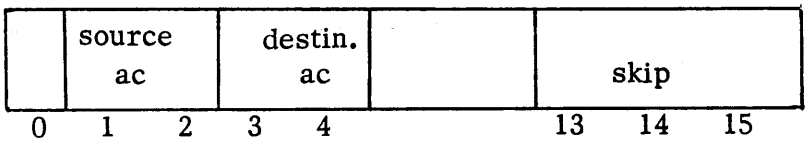
SYMBOL TABLE PSEUDO-OPS (contd.)

Pseudo-op	Instruction Type
.DALC	Arithmetic and logical
.DMR	Memory reference without accumulator
.DMRA	Memory reference with accumulator
.DIO	I/O without accumulator
.DIOA	I/O with accumulator
.DIAC	I/O without device code

For example, the pseudo-op .DALC defines a symbol that is an implied arithmetic and logical instruction mnemonic. When this symbol is used in a source program, the fields following the symbol are entered into the bit positions that represent source and destination accumulators and an optional skip mnemonic. The format for .DALC symbol definition and the source line with the symbol as it would later be used follow.

```
.DALC Δ symbol = expression
        .
        .
        .
symbol Δ source-ac Δ destination-ac [ Δ skip ]
```

The fields are assembled as shown below.



The following is an example of .DALC symbol definition.

```
103120 .DALC MULT4=103120

127120 MULT4 1,1
```

If an expression is to be added to a field that cannot accommodate it, an overflow (O) error is reported and the field remains unaltered. The following example illustrates a field overflow.

```
        103120 .DALC MULT4=103120

000000  107120 MULT4 4,1
```

If an expression is to be added to a nonzero field, the expression must evaluate to zero; otherwise, an overflow (O) error occurs. For example:

```
        123120 .DALC MULT4=123120  ;BITS 1-2 NOT ZEROED

000001  127120 MULT4 1,1           ;OVERFLOW ON  EXP1

 00002  123120 MULT4 0,0           ;ACCEPTABLE SINCE EXP1 =0
```

If an expression following a semi-permanent symbol does not fit the implied format, a format (F) error results. For example:

```
        103120 .DALC MULT4=103120  ;TWO, OPTIONALLY THREE,
                                   ;EXPRESSIONS REQUIRED FOR
                                   ;MULT4.
F       123120 MULT4 1
 00001  127121 MULT4,1,1,1
F00002  127121 MULT4 1,1,1,1
```

In summary, the expressions following a semi-permanent symbol must meet the following criteria.

1. As many expressions must follow the semi-permanent symbol as are required by the implied format. Some formats permit optional as well as required fields.

2. The expression must be able to fit the field in the semi-permanent symbol; otherwise, if

$$\underline{\text{expression}} > (2^{\text{field-width}} - 1)$$

the field is unaltered and an overflow (O) error is reported.

SYMBOL TABLE PSEUDO-OPS (contd.)

3. If the expression is to be stored in a nonzero field, the expression must evaluate to zero; otherwise, the field is unaltered and an overflow (O) error results.

4. The expression must meet the requirements of the implied format; otherwise, a format (F) error results.

Although a user symbol defined in one pseudo-op can be redefined in another symbol table pseudo-op, only the last definition is assigned to the user symbol.

.DALC Pseudo-op

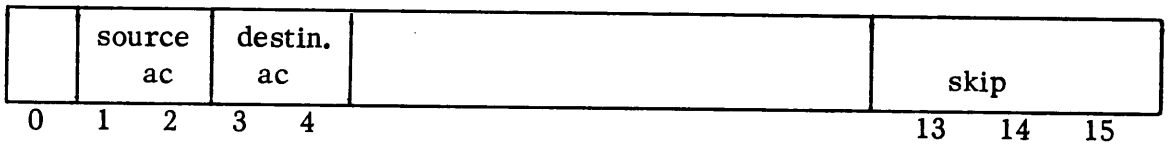
The .DALC pseudo-op

```
.DALC Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of an arithmetic and logical instruction. At least two fields, and optionally three, are required with the symbol. The format in which the semi-permanent symbol is later used in a source program is:

```
symbol Δ source-ac Δ destination-ac [ Δ skip ]
```

The fields are assembled as shown below.



The atom # can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 12, the no-load bit, thus inhibiting loading of shifter output.

The shift and Carry bits can be set by appending the following letters to a three-character symbol during .DALC definition:

```
.DALC Δ symbol [Z O C] [L R S] = expression
```

These letters cause the shift and Carry bits to be set as described in Chapter 5 for ALC instructions.

An example of the .DALC pseudo-op follows.

```
       103400 .DALC ADD=103400
00000  103400 ADD 0,0
00001  103402 ADD 0,0,SZC
00002  133401 ADD 1,2,SKP
```

.DMR Pseudo-op

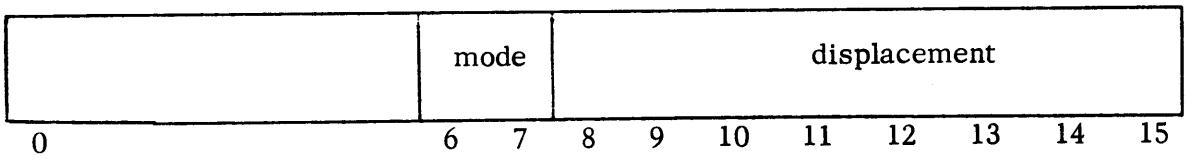
The .DMR pseudo-op

```
.DMR Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of a memory reference (MR) instruction without an accumulator field. The format in which the semi-permanent symbol is later used in the source program is:

```
symbol Δ displacement Δ mode
```

The fields are assembled as shown below.



The atom @ can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 5, the indirect addressing bit.

An example of the .DMR pseudo-op is given below.

```
       000000   .DMR   JMP = 000000
                         :
00205  001400   JMP    0,3
```

.DMRA Pseudo-op

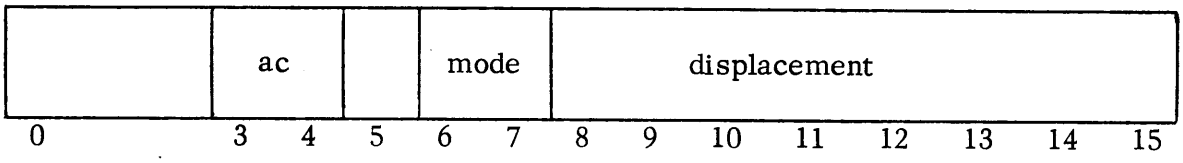
The .DMRA pseudo-op

```
.DMRA Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of a memory reference instruction requiring an accumulator field. The format in which the semi-permanent symbol is later used in the source program is:

```
symbol Δ accumulator Δ displacement [ Δ mode ]
```

The fields are assembled as shown below.



The atom @ can be specified anywhere in the source line as a break character. If it is used, a 1 is assembled at bit 5, the indirect addressing bit.

An example of the .DMRA pseudo-op is given below.

```
        020000    .DMRA   LDA = 20000
                    :
00011   023400    LDA     0,@0,3
```

.DIO Pseudo-op

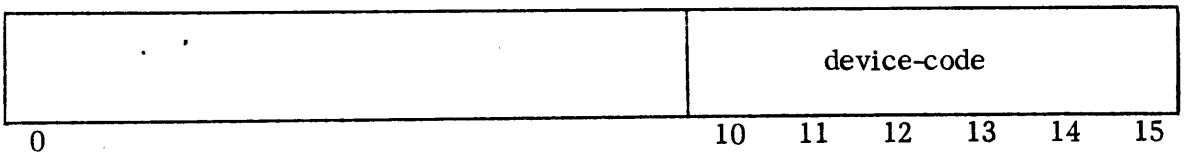
The .DIO pseudo-op

```
.DIO Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of an I/O instruction without an accumulator field. Only one field is required with the instruction. The format in which the semi-permanent symbol is later used in the source program is:

```
symbol Δ device-code
```

The field is assembled as shown below.



The Busy/Done bits can be set by appending one of the following letters to a three-character symbol during .DIO definition.

```
.DIO Δ symbol [S|C|P] = expression
```

These letters cause the Busy/Done bits to be set as described in Chapter 5 for I/O instructions without an accumulator field.

An example of the .DIO pseudo-op is shown below.

```
        063400  .DIO  SKPDN = 063400
                      :
                      :
00017   063413  SKPDN  PTP
```

.DIOA Pseudo-op

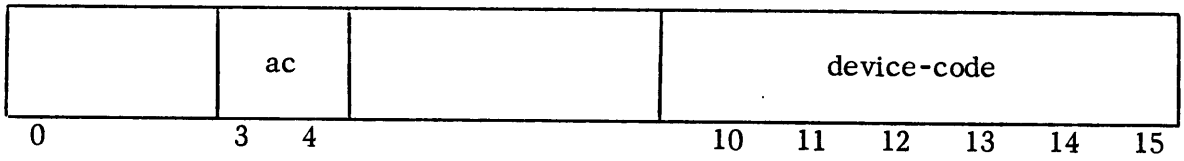
The .DIOA pseudo-op

```
.DIOA Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of an I/O instruction requiring two fields. The format in which the semi-permanent symbol is later used in a source program is:

```
symbol Δ accumulator Δ device-code
```

The fields are assembled as shown below.



The Busy/Done bits can be set by appending one of the following letters to a three-character symbol during .DIOA definition.

```
.DIOA Δ symbol [S]
               [C] = expression
               [P]
```

These letters cause the Busy/Done bits to be set as described in Chapter 5 for I/O instructions with an accumulator field.

An example of the .DIOA pseudo-op is shown below.

```
        060500  .DIOA   DIA = 060500
                         :
00110   060545  DIA      0,45
```

.DIAC Pseudo-op

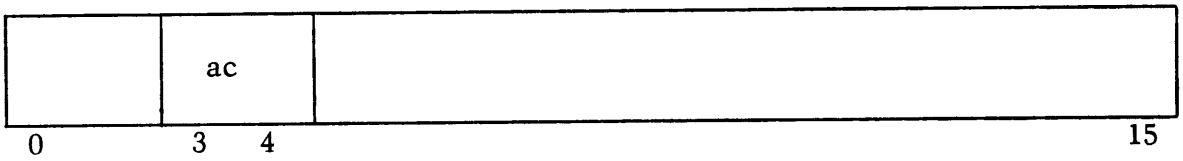
The .DIAC pseudo-op

```
.DIAC Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. In addition, the use of this symbol implies the formatting of an I/O instruction without a device code field. The format in which the semi-permanent symbol is later used in the source program is:

```
symbol Δ accumulator
```

The field is assembled as shown below.



The following is an example of the .DIAC pseudo-op.

```
        000430    .DIAC   RPT = 000430
                          :
00150   010430    RPT     2
```

.DUSR Pseudo-op

The .DUSR pseudo-op

```
.DUSR Δ symbol = expression
```

defines symbol as a semi-permanent symbol with the value of expression. Unlike other semi-permanent symbols, a symbol defined by a .DUSR pseudo-op is merely given a value and has no implied formatting. The semi-permanent symbol can be used in the source program anywhere a single precision operand can be used. The following illustrates the .DUSR pseudo-op.

```
000025    .DUSR    B = 25

000250    .DUSR    C = B*10

000223    C - B
```

.XPNG Pseudo-op

All symbol definitions except permanent symbols can be deleted from the assembler symbol table by the pseudo-op:

```
.XPNG
```

After expunging the symbol definitions from the table, the programmer can define instruction mnemonics, such as ADD and JMP, in any way desired.

```
        .XPNG

020000  .DMRA    LDA = 20000

040000  .DMRA    STA = 40000

        .END
```

LOCATION COUNTER PSEUDO-OPS

The extended Assembler provides four pseudo-ops directly affecting the location counter that allow the user to:

1. Allocate a block storage (.BLK),
2. Change the value and relocation property of the current location counter (.LOC), and
3. Specify the relocation property of source lines (.ZREL and .NREL).

.BLK Pseudo-op

The .BLK pseudo-op

```
.BLK Δ exp      .
```

allocates a block storage, exp words in length. The current location counter is incremented by exp. If the expression cannot be evaluated in pass 1 or the value of the current location counter exceeds 2^{15}-1, a location (L) error results.

The following example shows the effect of the .BLK pseudo-op on the contents of the location counter.

```
00125'000000  BY03:  0

       000004  BY10:  .BLK 4

       000002  BY11:  .BLK 2
00134'000000  BY13:  0
```

.LOC Pseudo-op

The .LOC pseudo-op

```
.LOC Δ exp
```

changes both the value and relocation property of the current location counter to that of exp. For example, if the expression evaluates to a zero relocatable value, the current location counter is set to that value and subsequent code is assembled as zero relocatable. Similarly, when the expression evaluates as either normal relocatable or absolute, the location counter is set to the normal relocatable or absolute address.

The following pseudo-op sets the absolute location counter to octal 1400:

```
00000'000010 10                    ;NORMAL RELOCATABLE
       001400 .LOC 1400            ;RETURN TO ABSOLUTE LOCATION 1400
01400  103400 ADD 0  0
```

The current location symbol (.) can be used with the .LOC pseudo-op. Depending on the relocation property of the code it is used with, it can mean: "current absolute address," "current normal relocatable address," or "current page zero relocatable address."

```
                .NREL


00002'000003 3
       000005'.LOC  .+2
00005'020010 LDA  0,10
```

.ZREL and .NREL Pseudo-ops

In addition to the assembly of absolute code, the programmer can use the Extended Assembler to produce two types of relocatable code: zero relocatable and normal relocatable.

Storage words that can be relocated but must reside in page zero should be assembled using zero relocatable mode. The user informs the assembler that a body of code is to be zero relocatable by preceding it with the pseudo-op:

.ZREL and .NREL Pseudo-ops (Contd.)

.ZREL

Storage words that can be relocated anywhere except page zero must be assembled using normal relocatable mode. The user informs the assembler that a portion of code is to be normally relocatable by preceding it with the pseudo-op:

.NREL

The Extended Assembler initially assumes the assembly mode to be absolute and assembles in this mode until it encounters either a .ZREL or .NREL pseudo-op. Thereafter, the assembly continues at the next available zero relocatable or normal relocatable address.

```
00000 000000 AL: 0          ;ABSOLUTE
             .ZREL
00000-000000 Z: 0           ;ZERO RELOCATABLE
00001-000000 ZL: 0
       000100 .LOC 100      ;RETURN TO ABS.  TO REMAIN IN ZREL,
00100 000003 M              ;USE THE .LOC .+number
             .NREL
00000'000003 .A: M          ;NORMAL RELOCATABLE
00001'002777 JMP @.A
```

INTERPROGRAM COMMUNICATION PSEUDO-OPS

The Extended Assembler provides facilities for separately assembled programs to share data, addresses, and symbols. Using the interprogram communication pseudo-ops, the programmer can:

1. Reserve a labeled or unlabeled program area to be shared by several programs (.COMM and .CSIZ),

2. Define entry symbols that can be referenced by other programs (.ENT), and

3. Name a program that is to become an overlay segment (.ENTO).

Then using the pseudo-ops .EXTD, .EXTN, .GADD, and .GLOC, programs can reference these program areas, entry symbols, and overlays. Because not all symbols will be defined at the end of pass 1 if these pseudo-ops are used, the Extended Assembler also includes the .EXTU pseudo-op which directs the assembler to treated undefined symbols as displacement externals.

.COMM Pseudo-op

The .COMM pseudo-op specifies the name and size of a program area to be used for interprogram communication by programs loaded together. The format of the pseudo-op is:

```
.COMM Δ usr-sym Δ exp
```

where usr-sym is the name of the area for interprogram communication and exp is its size, in words. This area is reserved by the first loaded routine declaring the name usr-sym. The area is reserved at NMAX (the first location available to load normal relocatable data), immediately preceding any normal relocatable code loaded. Other loaded routines declaring usr-sym share this original area, provided the sizes specified (exp) are the same.

Since usr-sym is an entry in the program, it cannot be redefined elsewhere in the program.

The symbol usr-sym can be referenced by other programs loaded together using any of the following pseudo-ops:

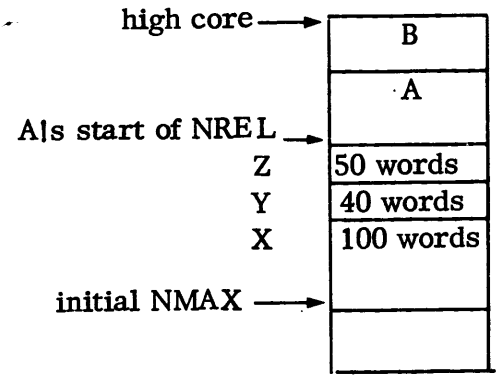
.EXTD .EXTN .GADD .GLOC

. COMM Pseudo-op (Contd.)

```
        .TITL A
000100  .COMM X,100
000040  .COMM Y,40
000050  .COMM Z,50
          .
          .
          .
        .END

        .TITL B
000100  .COMM X,100
          .
          .
          .
        .END
```

Loading would take place as follows:



. CSIZ Pseudo-op

The . CSIZ pseudo-op specifies the size of an unlabeled program area to be used for interprogram communication. Its format is:

```
. CSIZ Δ exp
```

where exp is the size, in words of the program area. The Extended Assembler evaluates this expression and passes the value to the loader. The loader stores this value in the User Status Table, at location 410 (USTCS). If more than one . CSIZ pseudo-op appears in a program to be loaded, at the termination of the load USTCS is set to the largest value specified by all . CSIZ pseudo-ops.

```
        .TITL X
000050  .CSIZ 50
          .
          .
          .
        .END


RLDR    X

    X
      NMAX 001037
      ZMAX 000050
      CSZE 000050      ;50 WORDS ALLOCATED BY LOADER
       EST
       SST
```

.ENT Pseudo-op

A program can define a symbol that can be referenced by separately assembled programs by including it in the .ENT pseudo-op line. The .ENT pseudo-op has the format:

```
.ENT Δ usr-sym1 [Δ usr-sym2...Δusr-symn]
```

where each usr-sym is a symbol defined in the declaring program and available to other programs. Each usr-sym must be defined with the program in which it is declared. Each symbol must be unique from entries defined in other programs loaded together to form a save file. If not, the loader issues an error message indicating multiply defined entries. Entry symbols can be referenced in separately assembled programs using any of the following pseudo-ops:

.EXTD .EXTN .GADD .GLOC

```
                                .TITLE  OVKIL
                                .ENT    OVKIL,OVKIX
                                .EXTD   .CPYL,.RTER
                                .EXTN   FRET,TOVRL,KILL

                        .NREL
00000'000001            1
00001'006002$OVKIL:     JSR     @.CPYL
00002'023611            LDA     0,@OVNUM,3
                          :
00007'000001            1
00010'031210 OVKIX:     LDA     2,FRTN,2
00011'021001            LDA     0,1,2
```

.ENTO Pseudo-op

If a program is to become an overlay within an overlay segment, the .ENTO pseudo-op can be used to name the overlay. The overlay can then be referenced from other programs using this name. The format of the .ENTO pseudo-op is:

```
.ENTO Δ usr-sym
```

where usr-sym becomes associated with the node number and overlay number assigned to the overlay at load time. Programs referencing this overlay must include the symbol usr-sym in an .EXTN or .GADD pseudo-op line.

Because the order in which overlays are loaded determines the overlay and node numbers, usr-sym is assigned a value at load time. For this reason, the symbol usr-sym cannot be used elsewhere in the program.

Overlay

```
.TITL  TIME
.ENTO  METER
   :
   :
```

Referencing Program

```
        .TITL  ROOT
        .EXTN  METER
        .NREL
           :
           :
ONO:    METER
           :
           :
        LDA 0, ONO
        ADC 1, 1
        .TOVLD      ;RDOS MULTITASKING LOAD
                    ;COMMAND
```

.EXTD Pseudo-op

A symbol can be defined in one program and referenced in a memory reference instruction or data line in another program if the referencing program contains the .EXTD pseudo-op. The format of the pseudo-op is:

```
.EXTD Δusr-sym1 [ Δusr-sym2 ... Δusr-symn ]
```

where each usr-sym$_i$ is a symbol (externally) defined in another program. This pseudo-op declares the symbols as displacement externals (or external displacements). Once a symbol has been declared a displacement external, it can be used in either of two ways: (1) as an address or displacement in a memory reference instruction or (2) to specify the contents of a 16-bit storage word.

When evaluated by the Extended Assembler, displacement externals must resolve to a value representable in eight binary digits. This means that:

1. If used as a page zero address in a memory reference instruction with index = 00, or in a data line, the displacement must resolve to a value in the range of 0 through 377 (octal), inclusive.

2. When used in a memory reference instruction with index ≠ 00 (addressing relative to the location counter or a base address contained in AC2 or AC3), the displacement must resolve to a value in the range of -200 through 177 (octal), inclusive.

The program defining the symbol usr-sym must declare it an entry symbol in a .ENT or .COMM pseudo-op.

.EXTD Pseudo-op (Contd.)

```
                                .TITLE  HOLD
                                .ENT    HOLD
                                .EXTD   .CPYL

                                .NREL

00000'000002            2
00001'006001$HOLD:      JSR     @.CPYL
```

EXTN Pseudo-op

With the .EXTN pseudo-op a program can reference a symbol in a data line that is defined in another program. The format of the pseudo-op is:

```
.EXTN Δusr-sym1 [ Δusr-sym2... Δusr-symn ]
```

where each usr-sym$_i$ is a symbol (externally) defined in another program. This pseudo-op declares the symbol(s) as normal external symbols. Normal externals can be used only in data line because an entire storage word must be reserved for each symbol.

The usr-sym must be defined in a separately assembled program and must appear in that program in an .COMM, .ENT, or .ENTO pseudo-op.

```
                        .TITLE  STTSK
                        .EXTN   FRET,TIDST
                          :
                          :
00007'177777            FRET
00010'177777 .IDST:     TIDST

                                .END
```

.EXTU Pseudo-op

This pseudo-op causes the assembler to treat all symbols that are undefined after pass 1 as if they had appeared in an .EXTD pseudo-op. The format of the pseudo-op is:

```
.EXTU
```

```
        .TITL FRED
        .EXTU
          :
          :
00006'024001$  LDA 1,UNDEF          ;UNDEF IS TREATED AS EXTERNAL
```

.GADD Pseudo-op

The .GADD pseudo-op generates a storage word whose contents is resolved at load time. The format of the pseudo-op is:

```
.GADD Δ usr-sym Δ exp
```

where the value of usr-sym is searched for at load time and, if found, is added to the value of exp to form the contents of the storage word. If usr-sym is not found, a loader error results and the storage word contains just the value of exp.

The usr-sym must be defined in a separately assembled program and must appear in that program in an .ENT, .ENTO, or .COMM pseudo-op. .GADD can thus be used in a similar manner to .EXTN with the following differences:

1. A user symbol in a .EXTN pseudo-op and used as a storage word is resolved regardless of whether the defining program is loaded before or after the program containing the .EXTN.

2. A user symbol in a .GADD block is resolved only if the defining program is loaded before the program containing the .GADD block.

.GADD Pseudo-op (Contd.)

```
                .NREL
                  .
                  .
                  .
00027'000032' . +3
      000202  .GADD  CL1, 2    ;CL1 MUST HAVE BEEN LOADED AND HAVE A
00031'000020  20               ;VALUE OF 200
                  .
                  .
                  .
```

.GLOC Pseudo-op

The .GLOC pseudo-op defines a global location block containing absolute data. Its format is:

```
.GLOC Δusr-sym
```

which begins the block of absolute data origined at the value of usr-sym at load time. The block is terminated by the next occurrence of a .LOC, .NREL, .ZREL or .END pseudo-op.

The usr-sym must be defined by an .ENT or .COMM in a program loaded prior to the global location block or a fatal load error results.

The global location block cannot contain external references, label definitions, or label references.

```
                .TITL B
                .GLOC MYAREA
  00000'000001  1
  00001'000002  2
  00002'000003  3
                .NREL
                 .
                 .
                 .
                .END
```

TEXT PSEUDO-OPS

.TXT, .TXTE, .TXTF, and .TXTO Pseudo-ops

These text pseudo-ops store ASCII octal equivalents of characters of a string. The pseudo-ops have the forms:

```
.TXT  Δ /string/
.TXTE Δ/string/
.TXTF Δ/string/
.TXTO Δ/string/
```

where string is a text string of ASCII characters and / represents a character that is used to delimit string.

Any character may be used to delimit string except:

1. A character that also appears within the string.
2. Carriage return, space, tab, comma, null, line feed, form feed, or rubout.

Upon encountering a text pseudo-op, the assembler takes the next significant character as a delimiter. Thereafter, pairs of characters are assigned to consecutive words of memory until the terminating delimiter is encountered. Every two characters generate a storage word. If the string contains an odd number of characters, the final character is paired with a null. If the string contains an even number of characters, a null word is assigned to the location immediately following the string (unless the .TXTN pseudo-op, described later, is used).

Storage of ASCII octal codes requires seven bits of an eight-bit byte. The leftmost bit is used to indicate parity, as indicated by the text pseudo-op used:

.TXT sets the leftmost bit of the byte to 0 unconditionally.
.TXTE sets the leftmost bit of the byte for even parity on the byte.
.TXTF sets the leftmost bit of the byte to 1 unconditionally.
.TXTO sets the leftmost bit of the byte for odd parity on the byte.

Within the string, any character may be used with the following limitations:

1. A carriage return or form feed appearing in the string will continue the string to the next line or page respectively and willnnot be stored as part of the string.

.TXT, .TXTE, .TXTF, and .TXTO Pseudo-ops

2. The characters null, line feed, and rubout are not interpreted and thus will not be stored if they appear in the string.

3. The characters < and > are used to delimit characters that would otherwise not be stored and, as delimiters, are not stored as part of the string.

The characters listed above that are not normally stored can be introduced as part of a text string by enclosing either the octal equivalent of the character or the special integer form of the character in angle brackets:

```
.TXT @LINE 1 <15>@          ←stores carriage return
.TXT @PENS SELLING <"@> $20/DOZEN@   ←delimiter stored
.TXT @LESS THAN IS WRITTEN AS <"< >.@  ←angle bracket stored
```

By default, text bytes are packed from right to left; the programmer can change the packing mode by using the .TXTM pseudo-op described later.

```
00000 041101  .TXT #AB CD#
      041440
      000104

00003 041101  .TXTE #AB CD#
      141640
      000104

00006 141301  .TXTF #AB CD#
      141640
      000304

00011 141301  .TXTO #AB CD#
      041440
      000304
```

.TXTN Pseudo-op

The user can suppress the addition of a null word to a character string containing an even number of characters using the pseudo-op .TXTN. This pseudo-op has the form:

.TXTN Pseudo-op

```
.TXTN Δ exp
```

where exp is any legal expression. The assembler evaluates exp and performs the following action.

Value of exp	Assembler Action
zero	All subsequent text strings containing an even number of bytes are followed by a full word consisting of two zero bytes.
nonzero	All subsequent text strings containing an even number of bytes terminate with the last two bytes of the string.

The .TXTN pseudo-op has no effect on text string containing an odd number of bytes; the last character is always paired with a zero byte.

```
       000000 .TXTN 0
00000  031061 .TXT /1234/
       032063
       000000
       000001 .TXTN 1
00003  031061 .TXT /1234/
       032063
```

.TXTM Pseudo-op

Normally, bytes are packed from right to left. This packing mode can be changed by the pseudo-op .TXTM, which has the form:

```
.TXTM Δ exp
```

where if the value of exp is 0, bytes are packed right to left and if the value of exp is nonzero, bytes are packed left to right.

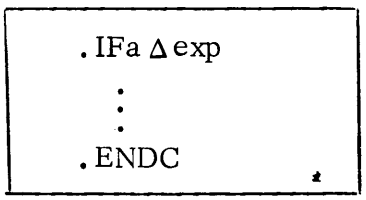
.TXTM Pseudo-op (contd)

```
       000000 .TXTM 0
00000  041101 .TXT #AB CD#
       041440
       000104

       000001 .TXTM 1
00003  040502 .TXT #AB CD#
       020103
       042000
```

CONDITIONAL PSEUDO-OPS

The Extended Assembler provides a conditional assembly feature which allows portions of a program to be assembled or to be by-passed by assembly on the basis of evaluation of absolute expressions. The conditional portion of the program has the general form:



where a is one of the letters E, G, L, or N and exp is an absolute expression. Together a and exp determine the conditions under which assembly will take place. The lines following, indicated by the elipsis, are assembled conditionally up to the the .ENDC pseudo-op, which terminates conditional assembly. The meaning of the letters E, G, L, and N is:

.IFE exp	Assemble if exp is equal to 0.
.IFG exp	Assemble if exp is greater than 0.
.IFL exp	Assemble if exp is less than 0.
.IFN exp	Assemble if exp is not equal to 0.

The expression in the .IFE, .IFG, .IFL, or .IFN pseudo-op must be evaluable in pass 1 of the assembly process, that is, all symbols used in the expression must be absolute and defined previous to the occurence of the .IFE, .IFG, .IFL, or .IFN.

Conditional assemblies should not be nested; if a conditional pseudo-op is encountered after a .IFa but before an .ENDC, the second conditional pseudo-op is ignored and is flagged with a K in the assembly listing.

If an .END or .EOT pseudo-op is included in conditionally assembled code, it will not be ignored. Regardless of the value of the expression with the .IFa, the .END or .EOT pseudo-op always terminates the assembly process. For example, in the conditionally assembled code:

```
.IFE 1
.END
.ENDC
```

the .END is not bypassed but causes the assembler to cease assembly.

```
       000000 A=0
       000000 B=A
              .NREL

       000000 .IFE B-2
              LDA 0,A
              .ENDC

       000000 .IFG B-2
              LDA 0,A
              .ENDC

       000001 .IFL B-2
00000'020000  LDA 0,A
              .ENDC

       000001 .IFN B-2
00001'020000  LDA 0,A
              .ENDC
```

The expression evaluates to false in these cases, so the load instruction is not assembled.

The expression evaluates to true in these cases, so the load instruction is assembled.

FILE TERMINATING PSEUDO-OPS

It is sometimes necessary to continue a program on more than one input source tape. The following pseudo-op indicates the end of the input file, but not the end of the input source:

```
.EOT
```

Upon encountering the .EOT pseudo-op, the Extended Assembler stops the source input device and halts with a 00006 in the address lights on the programmer's console. The assembly can be continued by loading the input device and pressing the console CONTINUE switch.

The .END pseudo-op terminates a source program, providing an end-of-program indicator for the relocatable loader. The syntax of the .END pseudo-op is:

```
.END{ Δexp}
```

where exp is an optional argument specifying a starting address for execution. The loader initializes TCBPC of the active TCB to the last address, if any, specified by a relocatable binary at load time. Execution of the loaded save file begins at this address. If the loader finds no starting address among programs loaded, an error message is given.

The following example illustrates the use of the .END pseudo-op.

```
                        .TITLE  FP
                        .ENT    PRI
                        .EXTN   TPR
                        .EXTN   FR
                        .EXTD   .CP
                        .NREL

00000'000001            1
00001'000001$PRI:       JSR     @.CP
00002'023611            LDA     0,@PRIO,3
00003'006402            JSR     @.TPRI
00004'177777            FR
      177611 PRIO = FTSTR

00005'177777 .TPRI:     TPR
                        .END
```

APPENDIX A

EXTENDED ASSEMBLER CHARACTER SET

7-Bit Octal Code	Character	7-Bit Octal Code	Character	7-Bit Octal Code	Character	7-Bit Octal Code	Character
000	Null*	072	:	125	U	165	u
011	Tab	073	;	126	V	166	v
012	LF*	074	<	127	W	167	w
014	FF	075	=	130	X	170	x
015	CR	076	>	131	Y	171	y
040	SP	100	@	132	Z	172	z
041	!	101	A	141	a	177	Rubout*
042	"	102	B	142	b		
043	#	103	C	143	c		
046	&	104	D	144	d		
052	*	105	E	145	e		
053	+	106	F	146	f		
054	,	107	G	147	g		
055	-	110	H	150	h		
056	.	111	I	151	i		
057	/	112	J	152	j		
060	0	113	K	153	k		
061	1	114	L	154	l		
062	2	115	M	155	m		
063	3	116	N	156	n		
064	4	117	O	157	o		
065	5	120	P	160	p		
066	6	121	Q	161	q		
067	7	122	R	162	r		
070	8	123	S	163	s		
071	9	124	T	164	t		

*These characters are ignored by the Extended Assembler.

APPENDIX B

ASSEMBLY LISTING ERROR CODES

ERROR CODE	MEANING
A	An address error has occurred: (1) an expression evaluates to other than an absolute, normal relocatable, or page zero relocatable address; (2) a page zero relocatable instruction references an address outside page zero; or (3) a normally relocatable instruction references an address outside the range of the location counter relative addressing.
B	An illegal character (not in the Extended Assembler character set) has been encountered.
C	A colon error has occurred; a label was defined improperly.
D	A radix error has occurred. Possible sources of error include an expression that is outside the range (2 through 10) of the .RDX pseudo-op.
E	An equals sign has been used incorrectly.
F	A format error has occurred. Possible sources of error include an incorrect number of arguments for an instruction or an operator follows a double precision or floating point number.
G	An error has occurred in defining an internal or external symbol.
I	A parity error has occurred on input.
K	A conditional assembly error has occurred: (1) an expression used with a .IFE, .IFG, .IFL, or .IFN pseudo-op cannot be evaluated in pass 1, or (2) a .IFE, .IFG, .IFL, or .IFN pseudo-op is nested within a conditional assembly.
L	A location counter error has occurred. An attempt was made to set the location counter to an illegal value.
M	A symbol has been defined more than once.
N	A number error has occurred. Possible sources of error include (1) a letter directly follows a number, (2) a digit has been used that is greater than the current radix, or (3) a number is too large or too small to be represented as a floating point number.

ERROR CODE	MEANING
O	A field overflow has occurred.
P	A phase error has occurred; the value of a symbol in pass 2 differs from that of pass 1.
Q	A source statement is questionable. The Extended Assembler is not able to evaluate a source line.
R	An expression error has occurred: (1) an expression does not evaluate to be absolute, relocatable, or byte relocatable; or (2) any expression combines normal and page zero relocatable symbols incorrectly.
S	The symbol table capacity has been reached.
T	A symbol table pseudo-op has been specified incorrectly.
U	A symbol is undefined.
X	A text error has occurred.
Z	An expression contains an illegal operand such as an external symbol, instruction mnemonic, double precision number, or floating point number.

APPENDIX C

EXTENDED ASSEMBLER PSEUDO-OPS

FUNCTION AND MNEMONIC	SYNTAX	USE
Title		
.TITL	.TITL△ usr-sym	Names a program.
Number Radix		
.RDX	.RDX △exp	Defines the radix exp to be used for numeric conversion: $2 \leq exp \leq 10$ (decimal)
Symbol Table		
.DALC	.DALC △ sym = exp	Defines a symbol as an arithmetic or logical instruction.
.DIAC	.DIAC △ sym = exp	Defines a symbol as an instruction requiring an accumulator.
.DIO	.DIO △ sym = exp	Defines a symbol as an input/output instruction without an accumulator field.
.DIOA	.DIOA △ sym = exp	Defines a symbol as an input/output instruction requiring an accumulator.
.DMR	.DMR △ sym = exp	Defines a symbol as a memory reference instruction without an accumulator field.
.DMRA	.DMRA △ sym = exp	Defines a symbol as a memory reference instruction requiring an accumulator.
.DUSR	.DUSR △ sym = exp	Defines a user symbol.

EXTENDED ASSEMBLER PSEUDO-OPS (cont.)

FUNCTION AND MNEMONIC	SYNTAX	USE
Symbol Table (cont.)		
.XPNG	.XPNG	Removes all symbols, except permanent symbols, from the symbol table.
Location Counter		
.BLK	.BLK Δ exp	Allocates a block of storage exp words in length.
.LOC	.LOC Δ exp	Changes the value and location property of the current location counter to that of exp.
.NREL	.NREL	Specifies that subsequent source lines are to be assembled using normally relocatable addresses.
.ZREL	.ZREL	Specifies that subsequent source lines are to be assembled using page zero relocatable addresses.
Interprogram Communication		
.COMM	.COMM Δ usr-sym Δ exp	Reserves an area exp words in length with the name usr-sym for interprogram communication.
.CSIZ	.CSIZ Δ exp	Reserves exp words for interprogram communication.
.ENT	.ENT Δ usr-sym$_1$ ...	Specifies that each usr-sym$_i$ is an entry point that can be referenced by other programs.

EXTENDED ASSEMBLER PSEUDO-OPS (cont.)

FUNCTION AND MNEMONIC	SYNTAX	USE
Interprogram Communication (cont.)		
.ENTO	.ENTO Δ usr-sym	Associates usr-sym with the node and overlay numbers assigned to the overlay.
.EXTD	.EXTD Δ usr-sym$_1$...	Specifies that each usr-sym$_i$ is a displacement external that is defined in another program.
.EXTN	.EXTN Δ usr-sym$_1$...	Specifies that each usr-sym$_i$ is a normal external that is defined in another program.
.EXTU	.EXTU	Specifies that all symbols undefined after pass 1 are to be treated as if they were in an .EXTD pseudo-op.
.GADD	.GADD Δ usr-sym Δ exp	Generates a storage word whose contents is, at load time, usr-sym + exp. usr-sym must be defined in a separately assembled program.
.GLOC	.GLOC Δ usr-sym	Begins a block of absolute data starting at the value of usr-sym up to the next .LOC, .NREL, .ZREL, or .END.
Text		
.TXT	.TXT Δ #string#	Stores the ASCII code for each byte in the text string. Left-most bit of each byte is set to zero.
.TXTE	.TXTE Δ #string#	Stores the ASCII code for each byte in the text string. Left-most bit of each byte is set for even parity.

EXTENDED ASSEMBLER PSEUDO-OPS (cont.)

FUNCTION AND MNEMONIC	SYNTAX	USE
Text (cont.)		
.TXTF	.TXTF Δ #string#	Stores the ASCII code for each byte in the text string. Left-most bit of each byte is set to one.
.TXTM	.TXTM Δ exp	Defines the packing of bytes for other text pseudo-ops: exp = Ø bytes are packed right to left. exp ≠ Ø bytes are packed left to right.
.TXTN	.TXTN Δ exp	Suppresses a terminating null word, normally appended to an even number of text bytes.
.TXTO	.TXTO Δ #string#	Stores the ASCII code for each byte in the text string. Leftmost bit of each byte is set for odd paritv.
Conditional Assembly		
.ENDC	.ENDC	Terminates a group of lines of conditional assembly.
.IFE	.IFE Δ exp	Specifies that subsequent lines up to .ENDC are assembled if exp = Ø.
.IFG	.IFG Δ exp	Specifies that subsequent lines up to .ENDC are assembled if exp > Ø.
.IFL	.IFL Δ exp	Specifies that subsequent lines up to .ENDC are assembled if exp < Ø .
.IFN	.IFN Δ exp	Specifies that subsequent lines up to .ENDC are assembled if exp ≠ Ø.

EXTENDED ASSEMBLER PSEUDO-OPS (contd.)

FUNCTION AND MNEMONIC	SYNTAX	USE
File Terminating		
.END	.END{ exp}	Terminates a source program and optionally provides a starting address for execution.
.EOT	.EOT	Indicates the end of an input tape but not the end of the source file.

APPENDIX D

OPERATING PROCEDURES

This appendix describes the procedures for operating the Extended Assembler under control of the Real Time Disk Operating System (RDOS) and the Stand-Alone Operating System (SOS) as well as in stand-alone mode, without operating system intervention.

The input to and output from the assembler are described in Chapter 1. Appendix E describes the format of the relocatable binary output from the assembler that must be input to the extended relocatable loader. Operation of the loader is described in the Extended Relocatable Loaders User's Manual, document number 093-000080.

RDOS OPERATING PROCEDURES

One or more source files can be assembled by the RDOS Extended Assembler using the CLI command:

```
ASM filename1 . . . filenamen )
```

Input to the assembler must be ASCII source files. Output can be a relocatable finary file, a listing file, or both.

Switches

Global: The following switches can be appended to the ASM command name.

- /L Produce a listing file.
- /N Do not produce a relocatable binary file.
- /U Include local (user) symbols in the relocatable binary output.
- /E Suppress error messages.
- /S Skip pass two. A BREAK is signaled after pass one, permitting the user to save a version of the assembler containing user semi-permanent symbols.
- /T Do not produce a symbol table list as part of the listing. (If a listing is requested, a symbol table is produced by default; this switch must be used to suppress the symbol table.)
- /X Produce a cross reference list of the symbol table.

Local: The following switches can be appended to a file name.

- /B Output the relocatable binary to the specified file name.
- /E Output error messages to the specified file name.
- /L Output the listing to the specified file name, overriding the global /L.

Switches (Continued)

/S Skip this file on pass two. (This switch should be used only if the file does not assemble any storage words.)

/N Do not list the specified file. (This switch is used when a listing is requested and only a selected number of files are to be assembled.)

File Name Searches

For input files, a search is performed first for filename.SR. If it is not found and filename has no extension, a search is then made for filename. On output, a relocatable binary file, filename.RB and a listing file (if the global /L switch is specified), filename.LS, are produced; filename is the first source file specified without a /S, /L, or /B local switch.

Caution

The following command would cause the loss of the first relocatable binary disk image:

```
ASM ($PTR,$PTR) $LPT/L
```

Although two distinct source files are read by the high-speed reader, each relocatable binary produced is labeled $PTR.RB. Thus, the first relocatable binary is overwritten by the second.

Error Messages

The following error messages may be produced during assembly.

Message	Meaning
NO SOURCE FILE SPECIFIED.	No input source file was specified in the command line.
ILLEGAL FILE NAME.	A file name is illegal.
FILE DOES NOT EXIST.	An input source file does not exist.
FILE ALREADY EXISTS.	An output file already exists.

Error Messages (Continued)

Message	Meaning
FILE WRITE PROTECTED.	An attempt has been made to write to a write-protected output file.
FILE READ PROTECTED.	An attempt has been made to read from a read-protected input file.
SWITCH ERRORS	The same file has been specified for both the listing and binary files.

Examples

1. The following command line assembles source file Z, producing the relocatable binary file Z.RB.

```
ASM Z )
```

2. This command line assembles file A, producing a listing file A.LS.

```
ASM/N/L A )
```

3. The following RDOS command assembles files A, B, C, and D from the default directory, of file E on fixed head disk unit 0 and paper tape mounted on the high-speed reader. (The source program mounted on the reader must be reloaded since the assembler requires two passes.) Binary relocatable files for each source file are output on the high-speed punch. Separate assembly listings are produced on the line printer.

```
ASM (A,B,C,D,DK0:E,$PTR) $PTP/B $LPT/L
```

SOS OPERATING PROCEDURES

The SOS Extended Assembler assembles one or more ASCII source files and produces a relocatable binary file with an optional listing file. The assembler can be loaded from paper tape; once loaded the assembler prints the prompt:

ASM

The user must respond with a command line of the form:

$$\left\{\begin{matrix}0\\1\\2\end{matrix}\right\} \text{ filename}_1 \ldots \text{filename}_n \;)$$

where: 0, 1, and 2 are keys describing the number of passes to be performed:

0 Perform one pass only on the specified input source file(s), then halt with the highest symbol table address in AC0. Normally, if this key is used, the input source is a Command Definition tape. The core image writer can then be called to preserve a copy of this assembly.

1 Perform two passes on the specified input source file(s), producing the specified binary and listing files. At the completion of pass two, the assembler outputs a new prompt and awaits a new command line.

2 Perfrom pass two only on the specified input source file(s), producing the specified binary and listing files. The symbol table used for this pass is that produced by the most recently executed pass one assembly. At the completion of this pass, the assembler outputs a new prompt and awaits a new command line.

$\text{filename}_1 \ldots \text{filename}_n$ are file or device names specifying input or output files. Input to the assembler must be ASCII source files. Input files are assembled in the order specified in the command line, from left to right. A cassette or magnetic tape unit cannot be used for both input and output or for more than one output file. However, a cassette or magnetic tape unit can be used for more than one input file.

Switches

Global: The following switches can be appended to the numeric keys at the beginning of the command line.

/E Suppress assembly error messages normally output to the $TTO.

/T Suppress symbol table listing.

/U Include local (user) symbols in the binary output file.

Local: The following switches can be appended to a file name.

/B Output the relocatable binary file on the specified device.

/L Output the listing file on the specified device.

/N Do not list the specified file on pass two.

/P Pause before accepting a file from the specified device. The following message is output by the assembler:

PAUSE - NEXT FILE, devicename

Assembly does not continue until the user depresses any key on the teletypewriter console.

/S Skip the specified source file on pass two.

/n Repeat the specified source file n times, where n is a digit from 2 through 9.

Error Messages

The following error messages may be produced during assembly.

Message	Meaning
I/O ERROR 1	A filename is illegal.
I/O ERROR 7	An attempt was made to read from a read-protected input file.

Error Messages (Continued)

Message	Meaning
I/O ERROR 10	An attempt was made to write to a write-protected output file.
I/O ERROR 12	File does not exist.
NO .END	No .END statement was found in any source file.

Examples

1. The following command line executes a two-pass assembly using cassette files 0, 1, and 2 on unit 0 as input. A binary file is produced on unit 1, file 0; a listing file is printed on the line printer. On pass 2, input file CT0:0 is skipped and input file CT0:2 is not listed.

```
1 CT1:0/B $LPT/L CT0:0/S CT0:1 CT0:2/N )
```

2. The following command executes the second pass of an assembly using input files 13, 14, 18, and 8 (in that order) on cassette unit 0. The binary, containing user symbols, is produced on file 1 of cassette unit 1; the listing is produced on file 0 of cassette unit 2.

```
2/U CT0:13 CT0:14 CT0:18 CT0:8 CT1:1/B CT2:0/L )
```

3. This command line executes a two-pass assembly on input files CT0:16, CT0:17, CT1:0, and CT1:1 with a listing printed on the line printer. Error messages normally output to the $TTO are suppressed; no binary file is produced.

```
1/E CT0:16 CT0:17 CT1:0 CT1:1 $LPT/L )
```

4. This command executes a two-pass assembly on two files, both read from unit 0. The first file is a parameter list to be read during the first pass only. After this parameter list is read, the pause message is output, and a new file must be placed in cassette unit 0. The first file of this new reel is scanned for both passes to complete the assembly. File 1 of unit 1 receives the binary output; no listing is produced.

```
1 CT0:0/S CT0:0/P CT1:1/B )
```

STAND-ALONE OPERATION

The assembler can be loaded from paper tape using the binary loader. Once loaded, the assembler requests information of the user concerning input/output devices and assembly mode. The user must respond to these queries with a single digit. The assembler queries and appropriate responses are given below.

Query	User Response and Meaning
IN:	Input device is one of the following:
	1 Teletypewriter reader without parity checking
	2 Teletypewriter reader with parity checking
	3 High-speed paper tape reader without parity checking
	4 High-speed paper tape reader with parity checking
	5 Teletypewriter keyboard without parity checking
LIST:	Listing device is one of the following:
	1 Teletypewriter Model 33 printer
	2 Teletypewriter Model 35 printer
	3 Line printer
	4 Teletypewriter Model 33 paper tape punch
	5 Teletypewriter Model 35 paper tape punch
BIN:	Output device for relocatable binary (object tape) is one of the following:
	1 Teletypewriter punch without local symbols
	2 High-speed paper tape punch without local symbols
	3 Teletypewriter punch with local symbols
	4 High-speed paper tape punch with local symbols
MODE:	Assembly mode is one of the following:
	1 Perform pass 1 only
	2 Perform pass 2 only and output an object tape
	3 Perform pass 2 only and output a listing
	4 Perform pass 2 only and output an object tape and listing. (In this case, the same device cannot be used for both the object tape and listing.)

RTOS OPERATION

To assemble one or more relocatable binaries, either the RDOS Extended Assembler or the SOS Extended Assembler may be used. Procedures for each have been given previously in this appendix.

To load and execute relocatable binaries under RTOS, follow the procedures given in Appendix B of the RTOS Manual, 093-000056. As described there, the relocatable binaries may be loaded using the (1) SOS relocatable loader, (2) RDOS relocatable loader, or (3) stand-alone relocatable binary loader.

APPENDIX E

RELOCATABLE BINARY BLOCK TYPES

The relocatable binary output of the Extended Assembler is divided into a series of blocks. The order of the blocks, if each type is present, is shown in Figure E-1.

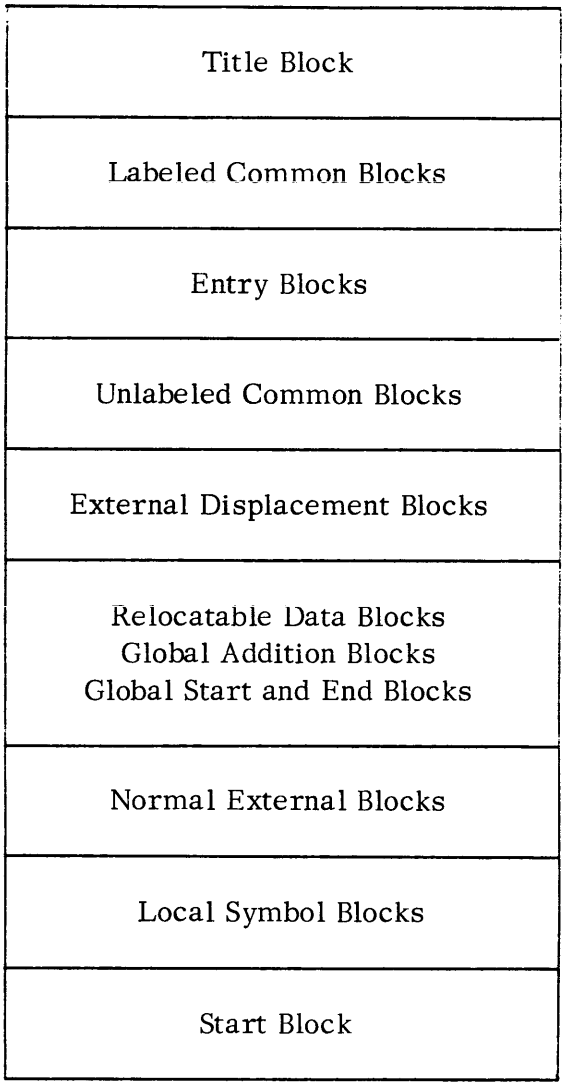


Figure E-1. Order of Relocatable Binary Blocks

The relocatable binary output must contain at least a Title and Start Block. Presence of one or more of the other types of blocks depends upon source input.

Every block contains the following information:

Word	Contents
1	Type of block, indicated by the following octal values:

Value	Block Type
2	Relocatable Data Block
3	Entry (.ENT) Block
4	External Displacement (.EXTD) Block
5	Normal External (.EXTN) Block
6	Start Block
7	Title (.TITL) Block
10	Local Symbol Block
11	Library Start Block
12	Library End Block
13	Labeled Common (.COMM) Block
14	Global Addition (.GADD) Block
15	Unlabeled Common Size (.CSIZ) Block
16	Global Location Start Block
17	Global Location End (.GLOC) Block

2	Word count, in two's complement format, never exceeding 15. If the word count is a constant for every block of the type, the word count is shown in parentheses in the format of the block.
3-5	Relocation flags or zero. The relocation property of each address, datum, or symbol is defined in three bits. For example, for a Relocatable Data Block, bits 0 through 2 of word 3 apply to the address, bits 3 through 5 apply to the first first word of data, bits 6 through 8 apply to the second word of data, etc. The meaning of the settings of three bits are as follows:

Bits	Meaning
000	Illegal
001	Absolute
010	Normal Relocatable
011	Normal Byte Relocatable
100	Page zero Relocatable
101	Page zero Byte Relocatable
110	Data Reference External Displacement
111	Illegal

If a block does not use relocation flags, these words are set to zero.

6 Checksum, such that the sum of all words in the block is zero.

Additional words in the block vary with block type.

For blocks containing user symbols, each symbol entry is three words in length. The first 27 bits of the three-word entry contain the user symbol name in radix 50 form. (Appendix F contains an explanation of radix 50 notation.) The last five bits of the second word are a symbol type flag. The meaning of the bit settings are as follows:

Bits	Meaning
00000	Entry Symbol
00001	Normal External Symbol
00010	Labeled Common
00011	External Displacement Symbol
00100	Title Symbol
00101	Overlay Symbol
01000	Local Symbol

The setting of the third word for each symbol entry varies with the block type.

The following pages show the format of each block type, arranged numerically by block type.

RELOCATABLE DATA BLOCK

	Word
2	1
word count	2
relocation flags 1	3
relocation flags 2	4
relocation flags 3	5
checksum	6
address	7
data	8
data	9
.	.
.	.
.	.
data	word count +6

Contents of the relocation flag words (words 3 through 5) are as described previously.

ENTRY BLOCK (.ENT)

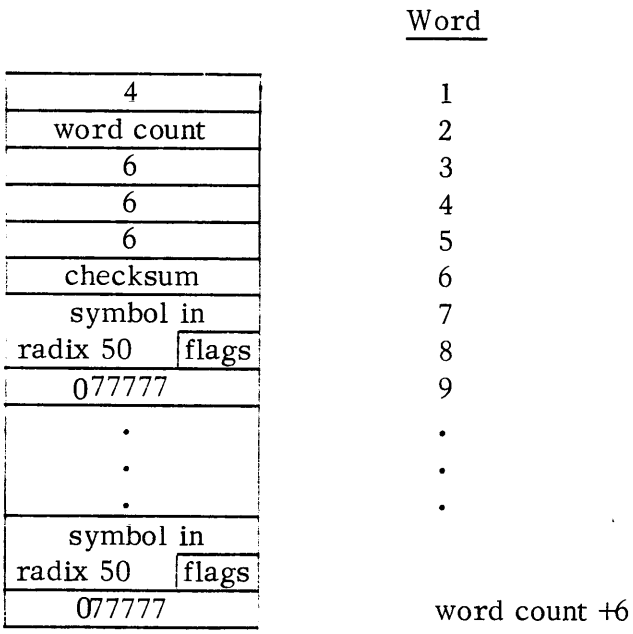
	Word
3	1
word count	2
relocation flags 1	3
relocation flags 2	4
relocation flags 3	5
checksum	6
symbol in	7
radix 50 | flags	8
equivalence	9
.	.
.	.
.	.
symbol in	
radix 50 | flags	
equivalence	word count +6

ENTRY BLOCK (.ENT) (Continued)

Note that the relocation flags for the Entry Block are as previously described, except that they apply to the third word of every user symbol entry. For Entry Block user symbols, the third word of the user symbol entry is used for the equivalence of entry symbol.

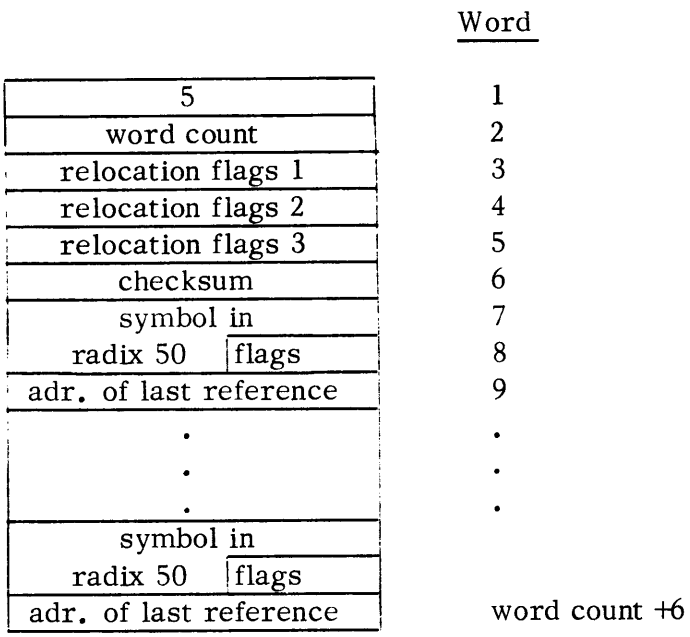
The overlay .ENTO is the same as the .ENT except for different flag values in word S1 (word 8, etc.).

EXTERNAL DISPLACEMENT BLOCK (.EXTD)



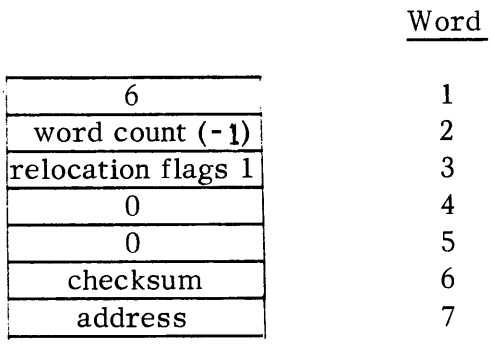
The third word of each user symbol entry in the External Displacement Block is set to 077777.

NORMAL EXTERNAL BLOCK (.EXTN)



The third word of each user symbol entry in the Normal External Block contains the address of the last reference. Relocation flags are used as in .ENT blocks.

START BLOCK



TITLE BLOCK (.TITL)

	Word
7	1
word count (-3)	2
0	3
0	4
0	5
checksum	6
title in radix 50 / flags	7
	8
0	9

The third word of the user symbol entry for a title is set to 0.

LOCAL SYMBOL BLOCK

	Word
10	1
word count	2
relocation flags 1	3
relocation flags 2	4
relocation flags 3	5
checksum	6
symbol in radix 50 / flags	7
	8
equivalence	9
.	.
.	.
.	.
symbol in radix 50 / flags	
equivalence	word count +6

The third word of every symbol entry is used for the equivalence of local symbols. Relocation flags are used as in .ENT blocks.

LIBRARY START AND END BLOCKS

The format of the Library Start and Library End Blocks differs from the format of other relocatable binary blocks, since the blocks are not generated by the assembler and are thus not internal to the binary output program but mark the beginning and termination of a file of binary output programs that constitutes a library file.

Library Start Block	Word	Library End Block
11	1	12
0	2	0
0	3	0
0	4	0
0	5	0
-11	6	-12

LABELED COMMON BLOCK (.COMM)

	Word
13	1
word count (-4)	2
relocation flags 1	3
0	4
0	5
checksum	6
symbol in radix 50 | flags	7
0	8
expression value	9
	10

Bits 0 - 2 of the relocation flags (word 3) apply to the expression (exp following .COMM). All other bits of the word are zeroed.

GLOBAL ADDITION BLOCK (.GADD)

	Word
14	1
word count (-5)	2
relocation flags 1	3
0	4
0	5
checksum	6
address	7
symbol in radix 50 (00000)	8
	9
0	10
expression value	11

Bits 0 - 2 of the relocation flags (word 3) apply to the address and bits 3 - 5 apply to the expression. All other bits of the word are zeroed.

UNLABELED COMMON SIZE BLOCK (.CSIZ)

	Word
15	1
word count (-1)	2
relocation flags 1	3
0	4
0	5
checksum	6
expression value	7

Bits 0 - 2 of the relocation flags (word 3) apply to the expression (exp) following the .CSIZ pseudo-op. All other bits of the word are zeroed.

GLOBAL LOCATION START AND END BLOCKS (.GLOC)

Start Block	Word	End Block
16	1	17
-3	2	-1
0	3	0
0	4	0
0	5	0
checksum	6	17
symbol in radix 50 00000	7	
	8	
0	9	

APPENDIX F

RADIX 50 REPRESENTATION

Radix 50 representation condenses symbols of up to five characters into two words of storage using only 27 bits. Each symbol, consisting of from one through five characters, is representable as:

$$a_4a_3a_2a_1a_0$$

where each a_i can be one of the following characters:

A through Z

0 through 9

If necessary, all symbols are padded with nulls to make a five-character symbol. Each character is then translated into an octal representation, as follows:

Character a_i	Octal Translation b_i
null	0
0 through 9	1 through 12
A through Z	13 through 44
.	45

Then, if a_i is translated to b_i, the bits required to represent the symbol can be computed as follows:

$$N_1 = ((b_4*50 + b_3) *50) + b_2$$

$N_{1maximum} = 50^3 - 1 = 174777$, which can be represented in 16 bits (one word).

$$N_2 = (b_1*50) + b_0$$

$N_{2maximum} = 50^2 - 1 = 3077$, which can be represented in 11 bits.

Thus, the symbol can be represented in 27 bits of storage, as shown in Appendix B in the binary output block formats.

Where there are a large number of page references for a given topic, the primary page reference will be indicated by an asterisk (*) following the reference.

cut along dotted line

DATA GENERAL CORPORATION
PROGRAMMING DOCUMENTATION
REMARKS FORM

DOCUMENT TITLE _____

DOCUMENT NUMBER (lower righthand corner of title page) _____

TAPE NUMBER (if applicable) _____

Specific Comments. List specific comments. Reference page numbers when applicable. Label each comment as an addition, deletion, change or error if applicable.

General Comments and Suggestions for Improvement of the Publication.

FROM: Name: _____ Date: _____
Title: _____
Company: _____
Address: _____
_____

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary If Mailed In The United States

Postage will be paid by:

Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: Programming Documentation

FOLD UP SECOND FOLD UP

STAPLE