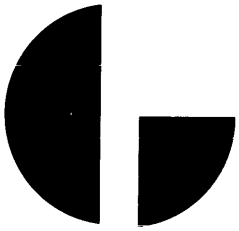


DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100

PROGRAM

Macro Assembler
User's Manual

ABSTRACT

The DGC Macro Assembler is upward compatible with the RDOS Extended Assembler. The Macro Assembler takes full advantage of the RDOS file system capabilities. It executes under RDOS and requires 16K core.

093-000081-02

Original Release - July 1972
First Revision - November 1972
Second Revision - June 1973

This revision, 093-000081-02, of the Macro Assembler Manual constitutes a major manual revision to 093-000081-01. The previous revision contained only those features peculiar to the macro assembler; this revision describes all features of the macro assembler, including those features common to both the macro and extended assemblers.

INTRODUCTION

The DGC macro assembler is upward compatible with the RDOS extended assembler, providing the following added features:

1. Expanded expression evaluation that provides for explicit as well as implicit precedence. The class of operators has been expanded to include relational operators.

2. A class of special symbols having a value (like ".") related to an internal assembler variable. This new class of symbols allows the user to determine useful information such as the number of arguments specified by a current macro call. Further, many pseudo-ops have a value associated with them and, using the proper syntax, may be used within expressions.

3. An assembly repeat feature for producing many lines of source from a simple repeat construct. This facility also encompasses conditional assembly. Conditionals may be nested to any depth.

4. A powerful macro facility which allows complete recursion as well as nested macro calls.

5. Literal references by any memory reference instruction. All literals will be optimally resolved in page zero. Literals are not restricted to absolute numeric quantities and, in fact, may consist of any legitimate expression.

6. The assembler has the facility to generate three-character alphanumerics for each occurrence of the character $ within a label. The facility is implemented in such a way that, for example, unique labels can be generated within nested macros.

NOTATION CONVENTIONS USED IN THIS MANUAL

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

< > Angle brackets enclose a variable definition of a class. The programmer replaces the variable definition with the appropriate atom within the class.

<variable>

_ Two or more words may be required for a variable definition. These are connected by underscores to indicate that the number of words in the variable definition is not significant when replacing the variable definition. for example:

.MACRO <user_sym>

↵ ↵ represents a carriage return.

↓ ↓ represents a form feed.

upper case letters Parts of the format that are written in upper case letters are literal parts of the Macro Assembly Language and must appear in context exactly as shown in the format.

lower case letters Parts of the format that are written in lower case letters are variables indicating that the programmer substitutes an appropriate item within a class. (user symbol, punctuation character, etc.)

⌈ ⌉ Broken square brackets are used to enclose optional parts of a format, e.g.,

LDA <exp> <exp> ⌈ <exp> ⌉ ↵

... Three dots indicate an omission of a word or words that should be obvious from the format.

△ A terminator or break character, defined as any number and combination of spaces, tabulations, and commas (i.e., class of space).

<sp> In certain positions a terminator must consist of a single space, a single tabulation, or a single comma. A terminator of this type is designated <sp>. For example:

<macro_symbol>⌈ △ <arg1><sp>... <arg_n>⌉ ↵

NOTATION CONVENTIONS USED IN THIS MANUAL

= Used to designate a terminator that may be written as a single equals sign (=) or class of space followed by an equal sign (Δ=).

: Used to designate a terminator that may be written as a single colon (:) or class of space followed by a colon (Δ:).

{ } Braces indicate alternate choices of formats one of which must be used. Where the choices are themselves optional, square brackets with hyphens will be used.

Example:

{ .DUSR
.DIAC
.DALC
.DIO
.DIOA
.DMR
.DMRA
.DISD } Δ <user_sym>=<stmt>

NOTATION VARIABLES

Some of the standard notation variables used throughout this manual are listed below.

<user_sym>	Any programmer defined symbol.
<semi_sym>	A symbol previously defined as semi-permanent by a symbol table pseudo-op.
<mac_sym>	A symbol previously defined by the .MACRO pseudo-op.
<exp>	An expression that may consist of symbols, numbers, and operators.
<stmt>	A syntax implicit statement of the form <semi_sym> Δ ... where <semi_sym> is followed by expressions appropriate to the type of semi-permanent symbol.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION TO MACRO ASSEMBLY

THE ASSEMBLY PROCESS

Assembly Language

A language is a set of representations, conventions, and rules used to convey information. A machine language is a language designed to be interpreted by a computer and it consists of numeric codes that can be understood by the computer.

An assembly language substitutes symbols for numeric codes. The purpose of an assembly language is to ease the task of the programmer by permitting him to write instructions in a form that is more meaningful to the programmer and easier for him to learn and remember. For example, if the programmer wishes to load a value that is stored at a given memory address, e.g., 5, into an accumulator, e.g., 0, the numeric code in machine language for this instruction to the Nova family computers would be:

```
20005  in octal or 0010000000000101 in binary
```

It is much easier for the programmer to remember symbolic names rather than numeric codes. For example, the instruction can be input to an assembler as:

```
LDA  0, 5
```

LDA is a symbol, in this case an instruction mnemonic meaning 'load accumulator' . The accumulator to be loaded is 0 and the memory address from which the accumulator is to be loaded is 5. The address might also have been represented by some symbol defined by the programmer. For example, the programmer might have defined the address as TEMP and the instruction would have been written:

```
LDA 0, TEMP
```

Processing of Assembly Language

A symbolic language, which is meaningful to the programmer, is not meaningful to the computer. Therefore, the symbolic assembly language must be translated into machine language. The process of translation from symbolic assembly language into machine language is called assembly, and the program that handles the translation is called an assembler.

Processing of Assembly Language (Continued)

The assembler simply substitutes numeric codes for symbolic instruction codes and numeric addresses (either absolute or relocatable as defined later) for symbolic addresses. There is a one-to-one correspondence between the symbolic instruction format written by the programmer and the numeric instruction format generated by the assembler. In other words, one line of symbolic instruction will be translated into one line of numeric instruction by the assembler.

The machine language output of assembly must then be processed by the computer - executed - to perform the functions desired by the programmer. The assembly process only translates symbolic language, called a source program, into machine language, called an object program.

Macro Facility

When symbolic assembly language and use of assemblers are substituted for machine language programming, the work of the programmer is simplified. Macro assembly is another step in simplifying the writing of a program.

Quite often a programmer needs to use the same set of symbolic instructions many times within his program. The basic function of macro assembly is to permit the programmer to write a set of instructions only once and then cause those instructions to be substituted in his program wherever he wishes.

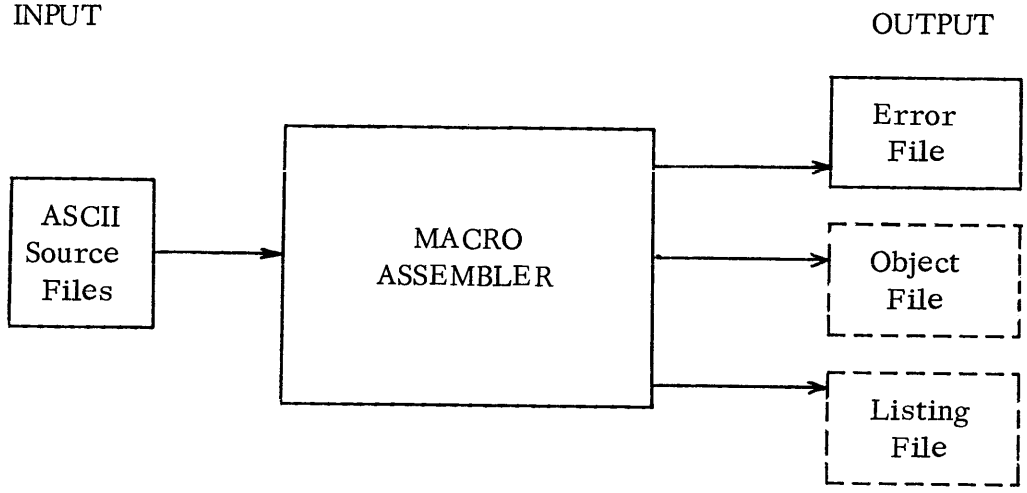
Fundamentally, a macro facility works as follows:

1. The programmer writes a set of symbolic instructions. The set of instructions is called a macro definition. The macro definition is given a name by the programmer.
2. Wherever the programmer wants that set of symbolic instructions in his program, he writes a macro call. At a minimum, the macro call contains the name of the macro definition.
3. The assembler contains a macro processor which substitutes the sequence of instructions (macro definition) for the macro instruction. This substitution is called macro expansion.

The macro facility is in fact much more sophisticated than this. For example, if the programmer has a series of instructions that repeat, except for accumulators and addresses, the macro definition may contain dummy arguments for the accumulators and addresses. The macro call in the program will contain the actual arguments to be substituted for the dummy arguments when the macro processor expands the macro. Thus, a macro definition is usually a skeleton of the actual instruction set that will result from macro expansion.

INPUT AND OUTPUT OF THE ASSEMBLER

The diagram below shows the input to and the possible outputs from the Macro Assembler. Input consists of one or more source files written in a subset of the ASCII character set. Output includes, at a minimum, a listing of any source program errors. Output can consist of a program listing, which includes any errors, a separate error listing, and an object file that can be loaded and executed. The object file is in a form called relocatable binary. The object file can be loaded by the relocatable binary loader for execution. The program and error listings are for programmer information purposes.



Broken boxes in the diagram represent optional output.

Assembler Input

The source program input to the assembler consists of characters that are a subset of the ASCII character set. The elementary scan of input by the assembler is a line-by-line read as follows:

1. Characters scanned up to a carriage return or a form feed character constitute a line of source program.

2. Three characters are unconditionally ignored. These are:

Character	Value
null	000
line feed	012
rubout	177

3. Characters having incorrect parity are replaced by the ASCII character "\". This character is transparent to higher level character processing, i.e., L\A is processed as LA.

Types of Assembler Output

There are three possible outputs from assembly:

1. A relocatable binary object program.
2. A program listing.
3. An error listing.

Relocatable Binary Output

The relocatable binary output 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. The address associated with the number by the assembler is not necessarily the final computer address in which the number will be stored by the relocatable binary loader; it may be a relative address that is relocated by the loader. The assembler produces as part of the object file the information needed by the loader for mapping each address as well as the contents of each address.

The programmer may choose not to output an object file.

Program Listing

The program listing permits the programmer to compare his input against the assembler output. A line of the program listing contains the following information:

Columns 1-3	If there are no errors detected in the input, these columns contain a two-digit line number followed by a blank space. If there are any input errors, each error generates a single letter code. The first error generates a letter code in column 3, the next in column 2, and a third in column 1. Only three error codes can be listed per line. If any error code is generated, there will be no line number given the line.
Columns 4-8	The location counter, if relevant. Otherwise the columns are left blank.
Column 9	The relocation flag pertaining to the location counter.
Columns 10-15	The data field, if relevant. Otherwise, the columns are left blank.
Column 16	The relocation flag pertaining to the data field.
Column 17-on	The source line as written and as expanded by the possible recognition of macros.

Program Listing (Continued)

An error flag is a single letter indicating the type of error appearing somewhere in the source line. A parity error on input, for example, produces the flag I in column 3 of the program listing line. Up to three error flags may appear on a given line. Additional errors will not produce flags.

The 5-digit location counter assigned by the assembler to an instruction or datum is output in columns 4-8. The location counter (LC) is immediately followed by a single character indicating the relocation mode of the address. The location counter flags are:

Flag	Meaning
space	absolute
-	page zero relocatable
=	page zero byte relocatable
'	normal relocatable
''	normal byte relocatable

Following the location counter is the 6-column value field which is immediately followed by a single character indicating the type of relocatability associated with the value. The data value flags are:

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

The last item on the line of program listing is the ASCII source line. The line is given as input, except for further expansion that may occur as a result of macro calls.

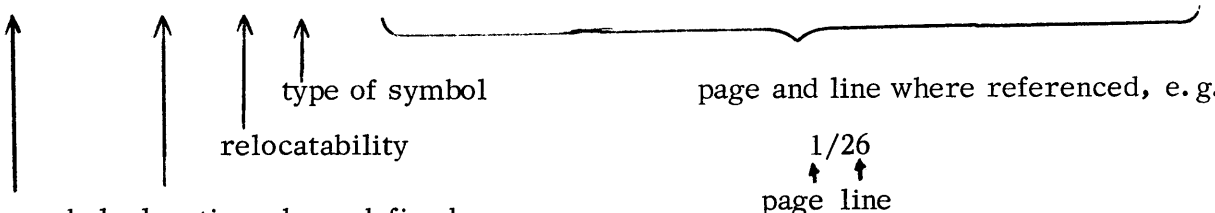
Certain lines of the listing, for example the expansions of macros, may optionally be suppressed by the programmer.

The programmer may choose not to output a program listing.

Cross Reference Listing

As part of the program listing, the macro 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.

```
0009  ALPHA

ALL     000007-          1/26    5/07    5/12    7/05
C11     000001-          1/19    3/17
C15     000002-          1/20    3/12    3/26
C377    000264'          8/07    8/18    8/30
C4      000000-          1/18    2/22
C40     000003-          1/21    4/18
CHR     000010-          1/27    3/06    3/09    3/21
CMPR    034015   MC      1/08    6/04    6/13    6/22    6/31    6/40
CNT     000011-          1/25    3/03    4/26    5/05    7/14
CONCM   000271'          1/07    8/38
D11     000004-          1/22    3/21
D128    000006-          1/24    2/39    4/24
D64     000005-          1/23    5/10    7/04
ERUR    000266'          1/40    4/32    7/11    8/33
FIND    000045'          3/05    4/27
FIND1   000050'          3/08    3/19
FIX     000042'          2/37    3/02
HERE    000012-          1/29    3/07    4/23
1       000006           6/02    6/09    6/14    6/11    6/11    6/13
                         6/20    6/20    6/27    6/28    6/29    6/29
                         6/37    6/38    6/38    6/45    6/46    6/47
LUbT    000235'          1/41    8/04
LNECN   000013-          1/30    2/13    2/38    3/02    5/04    5/06
MUVE    000101'          4/05    4/13
NEW     000216'          5/20    6/06    6/15    6/24    6/33    6/42
                         7/04
ONE     000222'          5/14    7/08
RO      000033'          2/35    2/41
RTN     000265'          8/04    8/11    8/13    8/27    8/31
S1      000014-          1/31    8/15    8/25
S2      000015-          1/32    8/14    8/24
SORT    000137'          5/06    7/15
SPACE   000016-          1/33    3/24    4/03
START   000000'  EN      1/04    2/05    8/46
STBT    000245'          1/42    8/13
```



Cross Reference Listing (Continued)

The meaning of the relocatability values given in the cross reference listing is identical to those given on page 1-5 for the program listing.

The symbol types that may appear in the cross reference listing are as follows:

ΔΔ	user symbol
EN	entry (.ENT)
EO	overlay entry (.ENTO)
XD	external displacement (.EXTD)
XN	external normal (.EXTN)
NC	named common (.COMM)

Error Listing

Error listing output differs from other assembler output in that a small class of errors detected on the first pass over the input will be output at that time. During the second pass, all errors in the program will be included as part of the program listing if the program listing is output. A separate error listing will be output if there is no program listing and is optional if there is a program listing. The separate error listing on the second pass is a subset of the program listing containing only those lines in which errors were detected.

The error listing is always to the teletype.

Processing Input into Output

The initial reading of a source line is the first step taken in assembly to produce the translation into relocatable binary output. To accomplish the translation the assembler must:

1. Build syntactically recognizable elements, called atoms, e.g., numbers, symbols, terminals, etc.

2. Recognize and act upon the basic atom of each line.

RELOCATABILITY

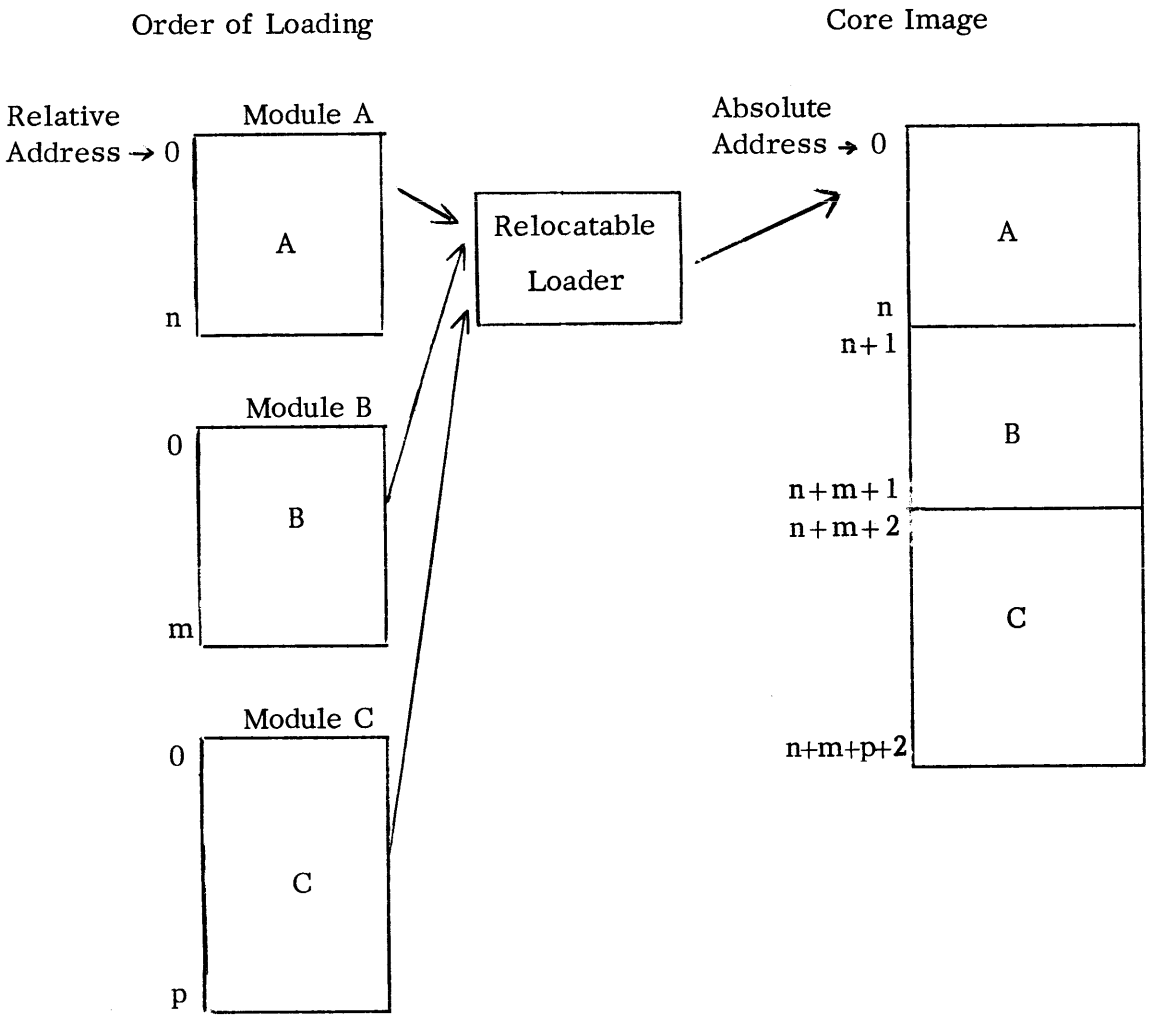
In relocatable assembly, storage words are assigned a relative location counter value. This value is initially zero and is incremented for every storage word generated. At the termination of the assembly, if n words have been generated, they have been assigned to relative locations 0, 1, ... , n-1. The actual addresses assigned to the words generated are determined at load time. The loader maintains the value of the first location available for loading, based on the programs previ-

ously loaded. If we call this value r, then a storage word of relative address a is actually loaded at absolute location $r+a$.

After the entire binary has been loaded, again assuming it took n words, the loader updates its first available location value by

$$r' \leftarrow r+n$$

In this way, any number of separately assembled modules can be loaded together without any conflict in absolute storage assignment. This is the major advantage of relocatability. The action of the loader is shown in the following diagram.



CHAPTER 2

ATOMS

CHARACTER INPUT

The input to the assembler is a string of characters. The assembler scans the input in one of two input modes: normal mode and string mode.

String Mode

In string mode, any ASCII character may be used with no interpretation of the input string. String mode is entered in one of three ways.

1. Comments

   Start of comment: ;
   Terminator: ↵ or ↑

   Example: ;SET MASK BITS ↵

2. Macro Definition Strings

   Start of macro string: .MACRO△<usr_sym>↵
   Terminator: %

   Example:
   ```
   .MACRO X ↵
   LDA 0,2 ↵
   MOVZL 1,1 ↵
   %
   ```

3. Text Strings

   Start of String: .TXT [E / F / O] △< α >

   Terminator: < α >

   Example: .TXT *EXPECTED VALUE= 60% of GROSS $. *

   In this form, < α> can be any character that is not used in the character string. In the example, the character * is used.

Normal Mode

All other input is in normal mode. In normal mode, the input string consists of characters from a subset of the ASCII character set, divided into lines. Each line is a string of characters terminated by either a carriage return or a form feed. The ASCII codes that are recognized by the assembler in normal mode are:

7 - Bit Octal Code	Character	7 - Bit Octal Code	Character	7 - Bit Octal Code	Character	7 - Bit Octal Code	Character
014	FF	066	6	115	M	145	e
015	CR	067	7	116	N	146	f
040	SP	070	8	117	O	147	g
041	!	071	9	120	P	150	h
042	"	072	:	121	Q	151	i
043	#	073	;	122	R	152	j
045	%	074	<	123	S	153	k
046	&	075	=	124	T	154	l
047	'	076	>	125	U	155	m
050	(	077	?	126	V	156	n
051	)	100	@	127	W	157	o
052	*	101	A	130	X	160	p
053	+	102	B	131	Y	161	q
054	,	103	C	132	Z	162	r
055	-	104	D	133	[	163	s
056	.	105	E	134	\	164	t
057	/	106	F	135	]	165	u
060	0	107	G	136	↑	166	v
061	1	110	H	137	←	167	w
062	2	111	I	141	a	170	x
063	3	112	J	142	b	171	y
064	4	113	K	143	c	172	z
065	5	114	L	144	d		

Normal Mode (Continued)

In normal mode, lower case alphabetics are unconditionally translated to their upper case equivalent. Any character not within the subset given above, if encountered during assembly, is given a B (bad character) flag and is ignored syntactically.

In normal mode, the assembler recognizes certain characters and certain groups of characters as different types of atoms. The atoms recognized are numbers, terminals, symbols, and special atoms.

ATOMS

A syntactic element of assembly language recognized by its specific class is called an atom. The classes of atoms are:

1) Terminals
2) Numbers
3) Symbols
4) Special Atoms

TERMINALS

A class of atoms, called terminals, serve the general function of separating numbers and symbols from other numbers and symbols. A terminal is a single character or double character. The terminals are operators and break characters.

Operators

Operators are a class of terminals that are used with single precision integers and symbols to form expressions. The operators are:

Arithmetic	B	Bit Alignment (shift)
	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
Logical	&	And
	!	Or (Inclusive)
Relational	==	Equal
	>=	Greater than or equal
	<=	Less than or equal
	>	Greater than
	<	Less than
	<>	Not equal

Operators (Continued)

The bit shift operator B is distinguishable from a character used in a symbol by the atom that precedes it. A bit shift operator is implied if the type of the last atom is a single precision integer or if the bit shift operator is immediately preceded by a right parenthesis.

Break Atoms

The terminals that are used primarily as separators are:

Δ	Δ represents the class of spaces -a space, a comma, a tabulation, or any number or combination of spaces, commas, and tabulations. The meaning of Δ is changed if followed immediately by a colon or equals sign (: or =) as defined below.
:	A colon (or the class of colons, Δ:) is one means used to define the symbol preceding <usr_sym>:
=	An equal sign (or the class of equal signs, Δ=) is another means of defining the symbol preceding <usr_sym>=
( )	Parentheses may enclose a symbol or an expression.
[ ]	Square brackets may enclose the actual arguments of a macro call.
;	A semicolon indicates the beginning of a comment string.
↵	A carriage return terminates a line of source code.
↓	A form feed terminates a line of source code.

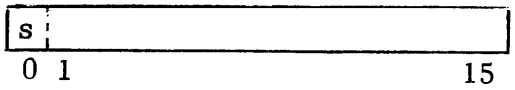
NUMBERS

Three types of numbers are defined for the Macro Assembler. These are:

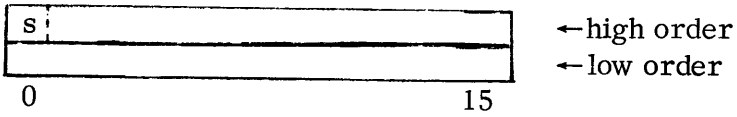
1) Single precision integer - stored in one word
2) Double precision integer - stored in two words
3) Single precision floating point constant - stored in two words

Representation of Numbers

A single precision integer is represented as a single word of 16 bits, having the range $0\text{-}65535_{10}$ ($0\text{-}177777_8$). The integer may be interpreted as signed using two's complement arithmetic in which bit 0 indicates a positive integer if 0 and a negative integer if 1.

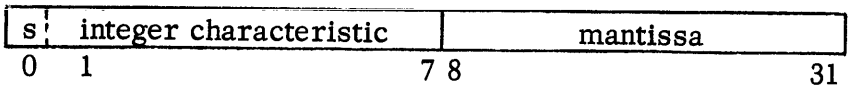


A double precision integer is represented in core in two contiguous words, where the first word is the high order word. Using two's complement notation, a double precision integer is represented as:



where: bit 0 of the high order word is the sign bit.

A single precision floating point constant is represented in core in two contiguous words having the format:



where: s is the sign, 0 = + and 1 = -

The integer characteristic is the integer exponent of 16 in excess 64_{10} (100_8) code. Exponents from -64 to +63 are represented by the binary equivalents of $0\text{-}127_{10}$ ($0\text{-}177_8$). An exponent of zero is represented as 100_8.

The mantissa is represented as a 24-bit binary fraction. It is convenient to view the binary fraction as six, 4-bit hexadecimal digits. The range of the magnitude of the mantissa is:

$$16^{-1} \leq \text{mantissa} \leq (1\text{-}16^{-6})$$

The negative of a number is obtained by complementing bit 0 (from 0 to 1 or 1 to 0). The characteristic and mantissa remain the same. When an expression evaluates to zero,

Representation of Numbers (Continued)

it is represented as true zero, i.e., two words of all zeroes in sign, characteristic, and mantissa bit positions.

The range of magnitude of a floating point number is:

$$16^{-1} * 16^{-64} \leq \text{floating-number} \leq (1-16^{-6}) * 16^{63}$$

which is approximately

$$5.4 * 10^{-79} \leq \text{floating-number} \leq 7.2 * 10^{75}$$

Most routines that process floating point numbers assume that all nonzero operands are normalized, and they normalize a nonzero result. A floating point number is considered normalized if the fraction is greater than or equal to 1/16 and less than 1; in other words it has a 1 in the first four bits (8-11) of the high order word. All floating point conversions by the assembler are normalized.

Use of Numbers

Single precision integers may appear in expressions and data statements, while double precision integers and floating point numbers may appear only in data statements

Source Representation of Numbers

Single Precision Integers

The source format of a single precision integer is:

[+] d [d ... d] [.] <break>

where: each d is a digit within the range of the current input radix. The initial d must be in the range 0-9.

<break> is any character other than a digit within the range of the current radix or a "." .

If a decimal point precedes the break character, the integer is evaluated as decimal. If there is no decimal point, the integer will be evaluated in the current input radix as set by the programmer. The range of input radix values is 2 through 20 as set by the .RDX pseudo-op. The representation of digits is shown in the table following.

Single Precision Integers (Continued)

Digit Representation	Digit Value	Radix Must Be ≥	Digit Representation	Digit Value	Radix Must Be ≥
0	0	any	A	10	11
1	1	any	B	11	12
2	2	3	C	12	13
3	3	4	D	13	14
4	4	5	E	14	15
5	5	6	F	15	16
6	6	7	G	16	17
7	7	8	H	17	18
8	8	9	I	18	19
9	9	10	J	19	20

If the input radix is 11 or greater, a number that would normally begin with a letter must be preceded by an intial 0 to distinguish the number from a symbol, e.g., to represent the decimal numbers, 15, 255, 4095, and 65,535 in hexadecimal:

```
000020 .RDX 16
000017 0F
000377 0FF
007777 0FFF
177777 0FFFF
```

The <break> character normally terminating the single precision integer is one of the following characters:

1. Operator: + - * / B
   ! &
   == <> <= >= > <
2. Terminal: △)] ;

Note the following exception:

The operator B (bit shift operator) will be interpreted as a digit if the radix is 12 or greater. To obtain the correct interpretation as a bit operator, the programmer must use the ← convention. The ← acts as a break character to the number string and is then ignored.

```
000020 .RDX 16
025423 02B13        ;B REPRESENTS DIGIT 11
000010 02←B13       ;B REPRESENTS BIT SHIFT OPERATOR
```

Single Precision Integers (Continued)

Within an expression, one integer might have the current radix and another might be given in radix 10 using the decimal point convention. Some examples of assembled values of expressions in which single precision integers of different radixes are used are shown below.

```
000002 .RDX 2
000012 101+101
000010 .RDX 8
000202 101+101
000012 .RDX 10
000312 101+101
000020 .RDX 16
001002 101+101
```

Special Formats of Single Precision Integers

There is a special input format that is converted to the single precision 7-bit octal value for the single ASCII character following. The input format is:

```
"α
```

where: α represents any ASCII character except line feed (012_8), rubout (177_8), or null (000).

Only the single ASCII character immediately following the quotation mark is interpreted. The ASCII characters, null, rubout, and line feed, which are invisible to the Macro Assembler, cannot be input using this format. However, the other ASCII characters can be represented as single precision integer in this manner.

```
000101 "A
000065 "5
000045 "%
000134 "\
```

The format can be used as an operand within an expression as shown.

```
000103 "A+2
000026 "B/3
177751 "*-"A
```

Note that "/ assembles an octal 15 and also terminates the line.

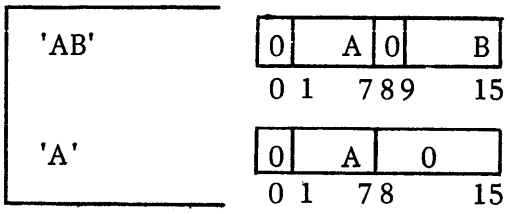
Special Formats of Single Precision Integers

A second format can be used to convert up to two ASCII characters to a single precision integer. The format is:

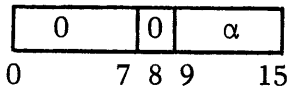
```
'<string>'    or    'string ↵
```

where: <string> can consist of any number of ASCII characters; only the first two characters are used to generate a 16 bit value.

Characters are packed left-to-right in the word:



Note that packing of '<string>' contrasts with packing of a single ASCII character that is input using the "α format. Such a character is stored



A string format consisting of two apostrophes without an intervening character will generate a word containing absolute zero.

Special formats of single precision integers may be used in any context that integers are allowed. The values of some simple expressions using string constants are given below.

```
040502 'AB'
041101 'BA'
000000 ''
000003 ''+5-2
041005 'B'+5
020101 ' A'
040501 "A+'A'
```

If a carriage return is encountered before the second ' , the string is terminated. For example:

```
006400 '
040415 'A
040502 'AB
```

Double Precision Integers

A double precision integer has the following source format:

```
{±} d{dd...d}{.} D <break>
```

where: each d is a digit within the current radix. The initial d must be in the range 0-9.

The character D before the break character indicates a double precision integer.

The optional decimal point is used to indicate that the integer is converted in decimal.

<break> is a terminal character that indicates the end of the double precision integer.

The break character is typically one of the terminals:

Δ ; ↓

Operators may not terminate double precision integers (a format error results).

The radix of a double precision integer may be in the range 2 - 20. If the radix is greater than or equal to 14, the letter D will be interpreted as a digit. To force the assembler to interpret D as indicating double precision, use the ← convention, e.g.,

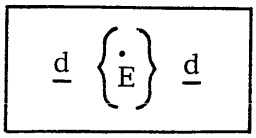
```
000020 .RDX 16

000455 12D     ;D IS INTERPRETED AS THE DIGIT 13
000000 12←D    ;D IS INTERPRETED AS SIGNALING THAT
000022         ;12 IS A DOUBLE PRECISION INTEGER
```

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

```
000010 .RDX 8
000000 1D
000001
177777 -1D
177777
000001 200000D
000000

000003 262147.D
000003
000001 100000.D
103240
```

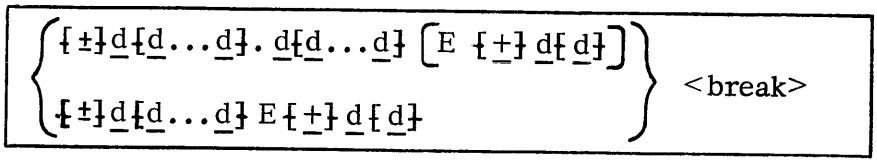
Single Precision Floating Point Constants

Much of the floating point number format is optional. The minimal format recognizable as a floating point number must consist of at least one digit in the range 0 - 9 followed by either a decimal point or the letter E followed by at least one digit in the range 0 - 9. The minimal floating point format is:



where: d is a digit in the range 0 - 9.

A single precision floating point number is represented in source format as:



where: each d is a digit 0 - 9. The mantissa and exponent are always converted in decimal, e.g., 2E9 ⇒ $2 * 10^9$.

One or two digits may represent an exponent following the letter E.

Equivalent floating point numbers may be formatted using either the letter E or the decimal point or both as shown below:

```
141376 -254.33
052172
141376 -254.33E0
052172
141376 -25433E-02
052172
141376 -25433E-2
052172
141376 -2543.3E-1
052172
```

<break> is typically one of the terminals:

△) ;

Single Precision Floating Point Constants (Continued)

If the current radix is radix 15 or larger, a letter E appearing in a number can cause interpretation of the number as an integer in the current radix rather than as a floating point number. To avoid this ambiguity, use the ← convention, for example,

```
000020 .RDX 16
155035 -25E3     ;E IS HEXADECIMAL 14
142141 -25←E3  ;E INDICATES FLOATING POINT
124000
```

Some examples of floating point constants in data statements with their stored values are:

```
040420 1.0
000000
040462 3.1415926
041766
140420 -1E0
000000
040200 +5.0E-1
030000
041421 +273.0E0
010000
```

Examples of Numbers

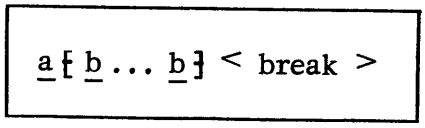
Following are some additional examples of the format of source program numbers and their assembled value.

Examples of Numbers (Continued)

```
000020 .RDX 16

053175 567D          ;HEXADECIMAL SINGLE PRECISION INTEGER
000000 567+D         ;HEXADECIMAL DOUBLE PRECISION INTEGER
002547

001067 567.          ;DECIMAL SINGLE PRECISION INTEGER
000000 567.+D        ;DECIMAL DOUBLE PRECISION INTEGER
001067

002547 567           ;HEXADECIMAL SINGLE PRECISION INTEGER
053175 567D          ;HEXADECIMAL SINGLE PRECISION INTEGER

005316 567+B14       ;HEXADECIMAL SINGLE PRECISION INTEGER, BIT
                     ;SHIFTED ONE BIT
012634 567+B13       ;HEXADECIMAL SINGLE PRECISION INTEGER, BIT
                     ;SHIFTED TWO BITS

042026 567+E1        ;FLOATING POINT CONSTANT (DECIMAL)
023000
```

SYMBOLS

A primary function of the assembler is the recognition and interpretation of symbols. Symbols are used both to direct the action of the assembler and to represent numeric values. The various classes of symbols will be discussed in Chapter 3. Their source representation is given below.



where: a is one of the characters A-Z . ?

b is one of the characters A-Z . ? 0-9

< break > is any character other than A - Z 0 - 9 . ?

If more than five characters precede the < break > character, only the first five are regarded as significant.

? as the first character of a symbol should not be used. It will be used by DGC in system macros to "guarantee" uniqueness of macro names.

SPECIAL ATOMS

There are three atoms that are transparent during the assembly scan. The effect of these atoms upon a line occurs after the entire line has been scanned.

@ An at sign (@) or any number of at signs appearing anywhere in a source program line of a memory reference instruction (MRI) or before an expression has the following effect.

1) When the rest of the MRI has been evaluated, presence of the @ sign or a series of @ signs anywhere in the instruction causes a 1 to be stored in bit 5. In the format of a memory reference instruction, bit 5 is the indirect addressing bit.

```
024020 LDA 1,20

026020 LDA 1,@20
```

2) In the format of a data word, bit 0 is the indirect addressing bit. When the expression has been evaluated, presence of the @ sign or series of @ signs causes bit zero of the word to be set to 1.

```
000025 25

100025 @25
```

A pound sign (#) or any number of pound signs appearing anywhere in a source program line of an arithmetic and logical instruction (ALC) has the following effect.

When the rest of the ALC has been evaluated, a 1 is stored in bit 12. (Bit 12 in the format of the ALC is the no load bit.)

SPECIAL ATOMS (Continued)

** Two consecutive asterisks appearing anywhere in a source program line (or any series of two or more asterisks) cause the suppression of listing of that line.

```
LDA  0,0,2
LDA  0,0,3**
LDA  0,0,3
.END
```

source program

```
00000  021000  LDA  0,0,2
00002  021400  LDA  0,0,3
                .END
```

listing

The atom may occur anywhere in the line. For example, all of the following would suppress the listing of .NOMAC.

```
**.NOMAC 1
  .NOMAC 1**
  .NOMAC** 1
  .NOMAC  **1
```

CHAPTER 3

SYNTAX

EXPRESSIONS

An expression <exp> has the format:

[<opn$_1$>]<opr><opn$_2$>

where: <opr> is a macro assembler operator.

<opn$_1$> and <opn$_2$> are operands which may be single precision integers or symbols or expressions evaluating to single precision integers. An operand preceding the operator is necessary for each operator, except for unary operators + and -. Either unary operator may follow an operator or precede an expression.

The macro assembler operators are:

Operator	Meaning
B	Bit alignment
+	Addition or plus
-	Subtraction or minus
*	Multiplication
/	Division
&	Logical AND. The result in a given bit position is 1 if and only if <opn$_1$> = 1 and <opn$_2$> = 1 in that bit position.
!	Inclusive OR. The result in a given bit position is 1 if either <opn$_1$> or <opn$_2$> or both is 1 in that bit position.
==	Equal to
< >	Not equal to
<=	Less than or equal to
<	Less than
>=	Greater than or equal to
>	Greater than

EXPRESSIONS (Continued)

Operators of more than one type may appear in an expression. Order of evaluation depends upon the precedence of the operators:

Operator	Precedence Level
B	3 (highest precedence)
+ - * / & !	2
< <= > >= == <>	1 (lowest precedence)

When operators are of equal precedence, the operators proceed from left to right. Parentheses around an expression may be used to alter precedence; an expression in parentheses is evaluated first.

Expressions are evaluated with no check for overflow.

Expressions Evaluating to Zero or One

An expression containing one of the operators

== <> <= < >= >

evaluates either to absolute zero or absolute one.

Examples:

```
000025 A=25
177763 B=-15
000000 A==B
000001 A<>B
000001 A+B-10==A-(2*10+5)
000001 A==(-B)+10
000000 A<>(-B)+10
000000 A==(-B)&A
```

Bit Alignment Operator

When the bit alignment operator is used, the operand <opn_1> preceding operator B is the value that is to be aligned. The operand <opn_2> following the operator represents the rightmost bit to which <opn_1> is aligned. The value of <opn_2> has the range:

$$0 \leq \langle opn_2 \rangle \leq 15_{10}$$

The formula for determining the result of a bit alignment operation is as follows:

for <opn_1>B<opn_2>, the resultant value will be

<opn_1>*2**(15.-<opn_2>)

where: <opn_2> is implicitly evaluated in decimal unless parentheses are used, e.g.,

```
.RDX 8
1B15 = 000001
1B(15) = 000004
```

The operator B can be misread as a symbol or part of a symbol. If the operand preceding the operator is a symbol, the operand must be enclosed in parentheses to avoid this misinterpretation. Some examples of bit alignment operations are:

```
        000025 A=25

 00002  100000  (A)B0
 00006  124000  (A)B4
 00011  012400  (A)B7
 00017  000124  (A)B13
 00021  000025  (A)B15
N00022  000000  (A)B16
```

Parentheses around <opn_1> and <opn_2> can be used to insure that the correct value is aligned properly. The effect of parenthesized operands is shown below.

```
000025 A=25
000010 C=10

000640 (A-C)*2B(3+C)
000640 (A-C*2)B(3+C)
177425 A-(C*2)B(3+C)
000640 A-C*2B(3+C)
```

Examples of Expressions

Some examples of expression evaluation are:

```
       000025 A=25
       000015 B=15
00000  000010 A*(B-10)/B
00001  000015 A&B/A!B
00002  000001 (A-10)==B
00003  000016 A/B+B
00004  000000 A&B/(A!B)
00005  000001 ((B/A)+5)>0
```

Relocation Properties of Expressions

Associated with each operand of an expression is its relocation property, and the relocation property of the result of evaluation of an expression will depend upon the relocation properties of the operands. Expressions described thus far have had absolute operands and the result of evaluation has been absolute.

A value, however, may have one of several relocation properties. These are:

absolute
page zero relocatable
page zero byte relocatable
normal relocatable
normal byte relocatable

The relocatable value is converted to absolute during the loading process by the addition of a constant, c, called a relocation constant. A relocation constant is added once if the value is singly relocatable and twice if the value is doubly relocatable (byte relocatable).

Two relocation constants are maintained by the loader, the normal relocation constant and the page zero relocation constant. In RDOS the initial value of the page zero relocation constant is 50, while the initial value of the normal relocation constant is variable but can be found in the User Status Table (UST) at location USTIN.

Page zero relocatable and normal relocatable operands cannot both be operands of a given expression. However, some mixing of like relocation properties is permitted. The relocation properties of operands and the relocation value of the

Relocation Properties of Expressions (Continued)

result are given below. In the list,

a	represents an absolute value
r	represents a relocatable value (either ZREL or NREL)
2r	represents a byte relocatable value (either ZREL or NREL)
kr	represents a relocatable value that can be converted to an absolute quantity by addition of a relocation constant, c, k times. However, if the final value of a expression is k-relocatable, the statement is flagged with a relocation error (R).

Expression	Relocation
a+a	a
a+r	r
r+r	2r
nr+mr	(n+m)r
a-a	a
r-a	r
a-r	-1r
r-r	a
nr-mr	(n-m)r
a*a	a
a*r	ar
r*r	Illegal
a/a	a
kr/a	$\frac{k}{a}$ r (only if k/a yields no remainder)
a/r	Illegal
a&a	a
a!a	a
r&r	Illegal
a&r	Illegal
r!r	Illegal
a!r	Illegal

Relocation Properties of Expressions (Continued)

All expressions involving the operators ≤ < ≥ > == <> result in an absolute value of zero (false) or one (true). When operands of these expressions are of differing relocation properties, all comparisons will result in a value of absolute zero (false) except when the operator is <> (not equal to).

Expressions evaluated using the rules given that result in a value of a, r, or 2r are legal. Expressions that do not evaluate to a legal relocation property will be flagged as relocation errors (R).

An example showing the relocation properties of expressions is given below. The assembler map indicating the relocation properties of each symbol is included.

```
        000002 A=2          ;A IS ABSOLUTE
               .NREL
  00000'000020'.+20
  00001'000000 R: 0       ;R IS RELOCATABLE
        000002'S=R+1      ; RELOCATABLE + ABSOLUTE IS RELOCATABLE
  00002'000001 A/A        ; ABSOLUTE OPERATOR ABSOLUTE RESULTS IN ABSOLUTE
  00003'000002"R+R        ;RELOCATABLE + RELOCATABLE IS BYTE RELOCATABLE
  00004'177777'R-A          ;RELOCATABLE - ABSOLUTE IS RELOCATABLE
  00005'000001 S-R          ;RELOCATABLE - RELOCATABLE IS ABSOLUTE
 R00006'000000'A!R          ;OPERATORS & AND ! MUST BE USED WITH ABSOLUTE
                            ;OPERANDS
  00011'000001'(A*R)/A    ;NO REMAINDER
               .END
 0002  .MAIN

A     000002        1/01    1/06    1/06    1/08    1/10    1/12    1/13
                    1/14    1/14
R     000001'       1/04    1/05    1/07    1/07    1/08    1/09    1/10
                    1/12    1/13    1/14
S     000002'       1/05    1/09
```

SYMBOLS

Symbols recognized by the macro assembler are classified as:

1. Permanent
2. Semi-permanent
3. User

The distinction between these classes is essential to the understanding of the assembly process.

Permanent Symbols

Permanent symbols are defined within the assembler and cannot be altered in any way. These symbols are used for two purposes: 1) they are used to direct the assembly process; and 2) they are used 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 setting the input radix for numeric conversions, setting the location counter mode, assembling ASCII text, etc. They are discussed in detail in Chapter 4.

A number of permanent symbols represent numeric values of internal assembler variables. For example, the symbol .PASS is used to represent the current pass number. On the first assembly pass its value is 0, while on the second its value is 1. If a symbol can be used both as a pseudo-op and as a value, the assembler recognizes which use is intended by the context in which it is used.

This is determined as follows:

1. If the first atom of a line is a pseudo-op, it is used to direct the assembler.
2. If the occurrence of the pseudo-op atom is in any other position within the line, its value is used.

A few examples will illustrate these rules.

The assembler has a pseudo-op, .TXTM, used to direct the packing of text bytes within a word. The two methods are left/right and right/left. The directive takes the form:

```
.TXTM <exp>
```

SYMBOLS (Continued)

Permanent Symbols (Continued)

If <exp> evaluates to zero (the default mode), bytes are packed right/left. If <exp> evaluates to non-zero, bytes are packed left/right.

Example 1

The line

```
.TXTM 1 )
```

directs the assembler to pack bytes left/right.

Example 2

The line

```
(.TXTM) )
```

assembles a storage word containing the value of the last expression used to set the text mode.

Example 3

If the following are given,

```
        000001 .TXTM 1
00000 000005 +.TXTM+4
```

The first line sets text mode to pack left/right while the second line generates a storage word containing absolute 5. (Note that the first atom of the second line is +.)

A list of all permanent symbols is given in Appendix F. These symbols cannot be redefined and must be used as described in this document. These symbols will never be printed as part of the user's symbol table.

Semi-permanent Symbols

Semi-permanent symbols form a very important class usually thought of as operation codes. Using appropriate pseudo-ops, symbols may be defined as semi-permanent; and their future use implies further syntax analysis. For example, a symbol may be defined as "requiring an accumulator". Use of this symbol causes

Semi-permanent Symbols (Continued)

the assembler to scan for an expression following the symbol. If not found, a format error results. If found, the value of the expression determines the value of the accumulator field bit positions to give a 16-bit statement value. Statements are discussed fully in the next section.

Semi-permanent symbols can be saved and used, without redefinition, for all subsequent assemblies. The assembler supplied by DGC contains a number of semi-permanent symbols defined specifically to conform to the Nova family instruction set. A list of these symbols is given in Appendix E. The user can eliminate these symbols and define his own set, or, more commonly, he can add to the given set. (See Chapter 6.)

Semi-permanent symbols will, by default, not be printed as part of the user's symbol table but can be printed if enabled by the global /A switch (see Operating Procedures, Chapter 6.)

User Symbols

The user can define any symbol that does not conflict with the permanent or semi-permanent symbols. Symbolic definitions are used for many reasons: to symbolically name a location, 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 disk file 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 which is known only for the duration of the single assembly in which they are defined. Global symbols have a value which is known at load time, i.e., they are used for interprogram communication.

STATEMENTS

In discussing source line syntax, the concept of a statement must be clearly understood. A statement, in the context of this manual, is the assembly of one or more fields, initiated by the occurrence of a semi-permanent symbol*, to form a 16-bit value with relocation. Fields are separated by the class of space, Δ. This 16-bit value need not generate a storage word, i.e., cause the location counter to be incremented. A statement is terminated by the successful assembly of the necessary number of fields as determined implicitly by the type of semi-permanent symbol.

* Except a .DUSR semi-permanent symbol.

STATEMENTS (Continued)

The Nova family of computers recognizes six basic instruction formats. Corresponding to each format is a pseudo-op enabling the definition of a semi-permanent symbol requiring fields appropriate to each format. Statements then fall into one of six formats:

1. Arithmetic and Logical
2. Input/Output
3. Input/Output with Accumulator
4. Instruction with Accumulator
5. Memory Reference
6. Memory Reference with Accumulator

The instructions corresponding to these statement types are discussed fully in "How to Use the Nova Computers." The syntax required for each statement type is given below. (Note that the semi-permanent symbols listed are those defined by DGC.)

ALC Statements

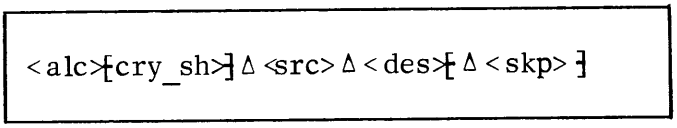
ALC statements are implied wherever the semi-permanent symbol is one of the following:

ADC
ADD
AND
COM
INC
MOV
NEG
SUB

Any one of these symbols will be represented as

<alc>

The ALC statement format is:



ALC Statements (Continued)

where: <alc> is one of the semi-permanent ALC statement symbols.

<cry_sh> represents the optional carry bits and shift options.

<src> and <des> represent the source and destination accumulators, respectively.

<skp > represents the optional skip field.

The special character # may appear as a separator for any field and forces bit 12 of the value to be 1 (no-load). The <src>, <des>, and <skp> fields can be specified by any legal expression. The optional <cry_sh> field must be one of the following alternatives:

<cry_sh>
Z
O
C
L
R
S
ZL
ZR
ZS
OL
OR
OS
CL
CR
CS

Carry bias and shift options, as well as the other fields of the ALC statement, are described in "How to Use the Nova Computers."

Some ALC statements are:

```
ADD     0,1
SUB     2,3,SKP
ADC#    2,1
NEGZL   1,0,SBN
COM     1,0,#SZC
```

STATEMENTS (Continued)

Input/Output Statements

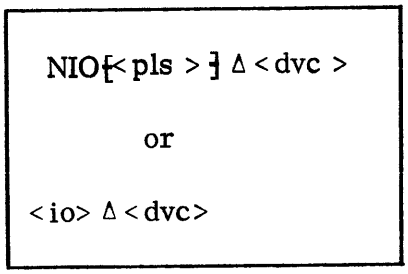
Input/output statements without an accumulator are implied whenever the semi-permanent symbol is one of the following:

NIO
SKPBN
SKPBZ
SKPDN
SKPDZ

Input/output statements with an accumulator are implied whenever the semi-permanent symbol is one of the following:

DIA
DOA
DIB
DOB
DIC
DOC

The format for an I/O statement without an accumulator is:

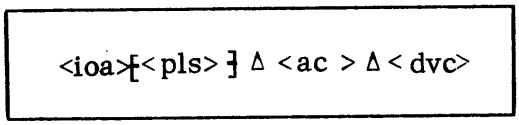


where: <pls> is the optional pulse specification, S, C, or P.

<dvc> is any legal assembler expression specifying a device code.

<io> is one of the semi-permanent symbols SKPBN, SKPBZ, SKPDN, or SKPDZ.

The format for an I/O statement with an accumulator is:



Input/Output Statements (Continued)

where: <ioa> is one of the semi-permanent symbols for an I/O statement with an accumulator.

<pls> is the optional pulse specification, S, C, or P.

<ac> is any legal assembler expression specifying an accumulator.

<dvc> is any legal assembler expression specifying a device code.

Some examples of I/O statements are:

```
NIO     TTI
NIOS    PTR
DOAS    0,PTP
SKPBZ   TTI
DIA     2,CPU
```

Instructions with an Accumulator

The semi-permanent symbols below require an accumulator specification:

INTA
MSKO
READS

The format for an instruction with an accumulator is:

```
<iac> Δ <ac>
```

where: <ac> is any legal expression specifying an accumulator.

For example:

```
READS   1
MSKO    3
```

STATEMENTS (Continued)

Memory Reference Statements

Memory reference statements not requiring an accumulator are:

DSZ
ISZ
JMP
JSR

Those requiring an accumulator are:

LDA
STA

Two formats can be used for memory reference instructions. They are

1. <mr> Δ[<ac> Δ] <dsp > Δ <ndx>

2. <mr> Δ[<ac> Δ] <adr>

In the first format: <ac> is any legal expression specifying an accumulator. It is given for LDA and STA statements.

<ndx> represents an index field and its value must be 0, 1, 2, or 3.

<dsp> represents a displacement that, for index modes 1, 2, and 3, must be in the range

-200 .LE. <dsp> .LT. 200

and for index mode 0 must be in the range

0 .LE. <dsp> .LT. 400

Normally, indexing is based on AC2 or AC3. Occasionally, however, index mode 1 is used to force PC relative addressing. Using explicit index mode 0 is unheard of.

The second form of address specification is most common. Using this form, the assembler attempts to form a correct address representation according to the flow chart shown on page 3- 16 . The final index mode for this implicit type

addressing is either 0 (page zero) or 1 (PC relative).

The basis for this type of address representation is to simplify the specification for the user.

If the address is addressable directly (.ZREL or .EXTD), the index field is set to 00 and the displacement field is set as follows:

1. If absolute, to the <adr> value (0 to 377) with absolute relocation.

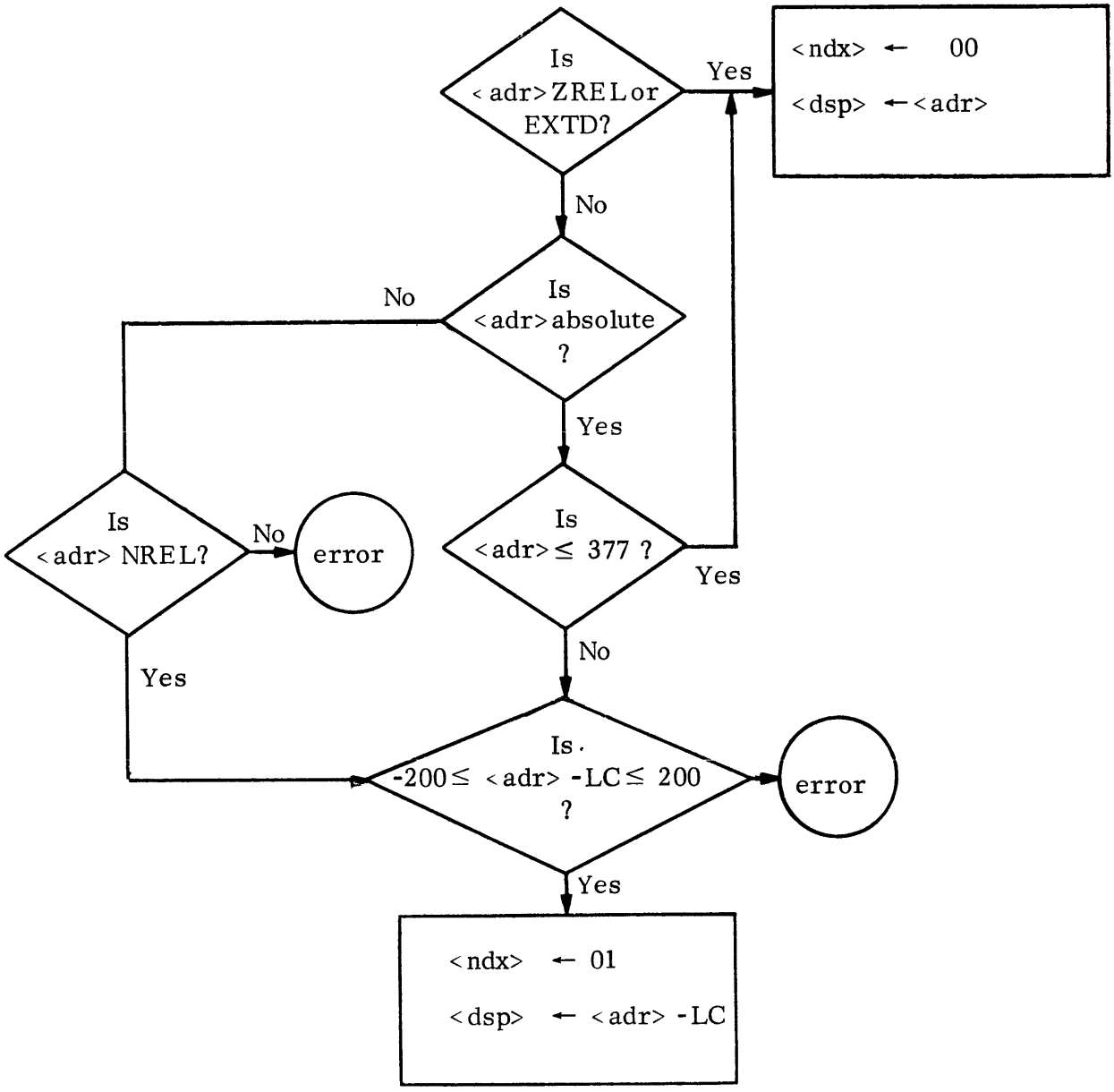
2. If page zero relative (.ZREL), to the relative address <adr> with page zero relocation.

3. If external displacement (.EXTD), to the external's ordinal value (1 to 377) with external displacement relocation.

However, if the <adr> is not directly addressable but is of the same relocation as the current relocation counter and is within addressable range, then the index field is set to 01 and an absolute displacement is computed as

<adr> - LC

ADDRESS EVALUATION



STATEMENTS (Continued)

Semi-Permanent Symbols with No Field Specifications

The discussion of statement types would not be complete without mentioning the one type of semi-permanent symbol that represents a 16-bit value and requires no additional field specification. A number of these symbols are defined and can be used simply as operands within expressions.

For example, these symbols represent:

1. Skip mnemonics

   SKP
   SZR
   SNR
   SZC
   SNC
   SBN
   SEZ

2. Device codes, e.g.,

   TTI
   TTO
   PTR
   PTP

3. Self-complete instructions, e.g.,

   INTEN
   INTDS
   IORST
   HALT

ASSEMBLY

Having discussed expressions and statements, the line by line assembly scan to produce relocatable binary output can now be described. The majority of source lines effect 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 statement, and an address. At assembly time, the address assigned is the current location counter (LC) and, as discussed earlier, this LC may be absolute, relocatable within page zero (ZREL), or relocatable outside page zero (NREL). The generation of each 16-bit storage word causes the LC to be incremented by one. Thus, in general, storage words are assigned to consecutive, increasing LC values.

Labels

The programmer often needs to name a storage word symbolically. This allows him to reference these words without regard for their assembler defined numeric address. The value of the current location counter can be assigned to a user symbol at the start of any source line using the following syntax:

```
< usr_sym > :
```

The value of < usr_sym > will, therefore, be the address of the next storage word assembled. Since some assembler lines do not generate storage words, this definition is not necessarily associated with the statement that it appears within. More than one definition can be made, providing all symbols are defined at the beginning of the line. For example,

```
LAB1: LAB2: LAB3:
```

Equivalence

Another means of defining a symbolic name to a numeric value is by equivalence. An equivalence line associates a value to a symbol, and that symbol can then be used any time the value is required. The syntax of an equivalence line is:

```
< usr_sym>  =  < exp> | < stmt >
```

Equivalence (Continued)

Note that a statement can be used to determine the symbol's value. This is the simplest example of a statement being used without generating a storage word.

Some equivalence lines are:

```
A=    3+5*(4-1)
B=    10.
C=    ADD#    0,1,SKP
```

Storage words can be assembled using a most common syntax:

```
      < exp >  )
or
      < stmt > )
```

These forms account for the majority of assembly source lines. For example, if the current location counter has a value of zero NREL (00000'), the statements below generate words at locations 00000' through 00006'.

```
                1
                3+4 < > (17-10)/2
A:              MOV     Ø, 1
                NIOS    PTR
                JMP     A
B=              6+10.
                B
                A
```

Two number types discussed earlier, double precision integer and floating point, produce 32-bit values. These numbers can never be combined in expressions. They are used to assemble a double storage word. If a label is used to start the line, it is assigned the value of the first word's address. For example, at location 100 the following are assembled:

```
                100.D
                100.0
A:              1E2
```

Two words are produced by each line and A is assigned the value 100 absolute.

CHAPTER 4

PERMANENT SYMBOLS

The permanent symbols are grouped into logical categories. The format for describing permanent symbols is as follows:

1. Permanent symbols that are pseudo-ops are listed as such and their syntax and purpose are given.

2. Permanent symbols that may be either a pseudo-op or a value are listed as pseudo-ops and their syntax, purpose, and their value as a symbol are given.

3. Permanent symbols that are not pseudo-ops are listed as symbols and their value is given.

TITLE PSEUDO-OP

Pseudo-op: .TITL

Syntax: .TITL△<usr_sym>

Function: This pseudo-op names a program. The title given is printed at the top of every listing page. In addition, it is used to identify the relocatable binary output to the loader, library file editor, and the debugger. The <usr_sym> need not be unique from other symbols defined by the program.

Default: .MAIN

Example:

```
.TITL SYB
```

NUMBER RADIX PSEUDO-OPS AND VALUES

Pseudo-op or Symbol: .RDX

Syntax: .RDX△ <exp>

Purpose: This pseudo-op defines the radix to be used for numeric input conversion by the assembler. <exp> is evaluated in decimal and the range of <exp> is:

$$2 \leqslant \text{exp} \leqslant 20$$

Value: The numeric value of .RDX is the current input radix.

Default: The default input radix is 8.

Examples:

```
       000010 .RDX 8
00000  000123 123
       000012 .RDX 10
00001  000173 123
       000020 .RDX 16
00002  000443 123
00003  000020 (.RDX)    ;CURRENT VALUE OF INPUT RADIX
```

Note: Input and output radices are entirely distinct. Setting the input radix has no effect upon the listing radix.

Pseudo-op or Symbol: .RDXO

Syntax: .RDXO△<exp>

Purpose: This pseudo-op defines the radix to be used for numeric conversion of the listing output fields. The <exp> is evaluated in decimal and the range of <exp> is

$$8 \leq \langle exp \rangle \leq 20$$

Value: The numeric value of .RDXO is the current output radix.

Default: The default output radix is 8.

Examples:

```
000012 .RDX  10          input radix 10
 00010 .RDXO 10          output radix 10
 00077 77
 00022 22                } decimal listing
 00045 45
 00008 .RDX  8           input radix 8
000010 .RDXO 8           output radix 8
000077 77
000022 22                } octal listing
000045 45
000020 .RDX  16          input radix 16
  0010 .RDXO 16          output radix 16
  0077 77
  0022 22                } hexadecimal listing
  0045 45
000010 .RDXO 8           output radix 8 (input radix 16)
000167 77
000042 22                } octal listing
000105 45
 00010 .RDXO 10          output radix 10 (input radix 16)
 00119 77
 00034 22                } decimal listing
 00069 45
 00010 .RDX  10          input radix 10
  0010 .RDXO 16          output radix 16
  004D 77
  0016 22                    hexadecimal listing
  002D 45
000010 (.RDXO)           current value of output radix (always prints as 10)
```

Note: Input and output radices are entirely distinct. Setting the output radix has no effect on the input radix.

SYMBOL TABLE PSEUDO-OPS

The symbol table pseudo-ops are of the form:

```
<pseudo-op>△<usr_sym>= <stmt>
```

where: <pseudo-op> is one of the following:

.DALC
.DIAC
.DIO
.DIOA
.DMR
.DMRA
.DUSR

<usr_sym> is a programmer chosen symbol

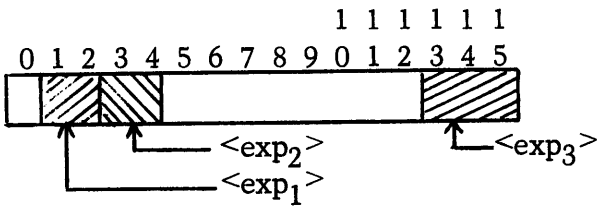
<stmt> is a statement (based on previously defined semi-permanent symbols) or expression

The symbol table pseudo-ops define a user symbol <usr_sym> as a semi-permanent symbol <semi_sym> that has as its value the value of the statement or expression following the equals sign.

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 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 accumulators and the optional skip field. The format for .DALC definition of a symbol and the format of the symbol as it would later be used are:

```
.DALC△<usr_sym>= <stmt>
      :
      :
<usr_sym>△<exp1>△<exp2>△ ⟨<exp3>⟩
```

where: $<exp_1>$, $<exp_2>$, and $<exp_3>$ are stored as:



For example:

```
103120 .DALC MULT4=103120

127120 MULT4  1,1
```

MULT4 is defined 1000011001010000

MULT 4 must be used with two expressions that evaluate within the limits of the ALC instruction accumulator fields (2 bits for each)

1010111001010000

If the field to which an expression is to be added cannot accommodate the value of the expression an overflow error will occur. The field will be unaltered.

```
       103120 .DALC MULT4=103120
000000 107120 MULT4  4,1
```

If the field to which an expression is to be added is not zero, the expression to be added must evaluate to zero. Otherwise, an overflow error will occur.

```
       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 the expressions following a semi-permanent symbol do not fit the implied format a formatting error will result.

```
       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
```

A symbol defined as semi-permanent by a symbol table pseudo-op must, in summary, meet the following conditions:

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

2. If the number of expressions following the semi-permanent symbol do not meet the requirements of the implied format, a format error (F) will result.

3. If an expression does not meet the requirements of the field, i.e., if

   $$\langle exp \rangle \ .GT. \ (2^{field\text{-}width} - 1)$$

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

4. If the field in which the expression is to be stored $\neq 0$, the expression must = 0. Otherwise, an overflow (O) error results, and the field is unaltered.

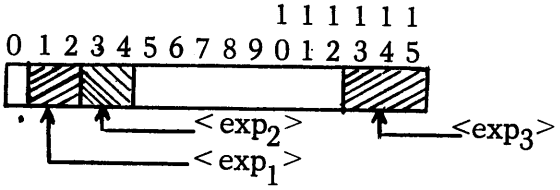
A given <usr_sym> defined in one symbol table pseudo-op may be redefined in another symbol table pseudo-op. The last definition will be the one assigned to <usr_sym>.

The implied format required by each symbol table pseudo-op is given as part of the pseudo-op summary descriptions following.

Pseudo-op: .DALC

Syntax: .DALC△ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. In addition, the use of this symbol implies formatting of an ALC instruction. At least two fields, and optionally three, are required. These fields are assembled as shown below.



The format in which the semi-permanent symbol is used is:

<semi_sym>△<exp$_1$>△<exp$_2$>[△ <exp$_3$>]

Examples:

```
        103400 .DALC ADD=103400


 00000 103400 ADD 0,0
 00001 103402 ADD 0,0,SZC
 00002 133401 ADD 1,2,SKP

F       123400 ADD 1
F       103400 ADD
```

Notes: The atom # may be specified anywhere as a break character. If seen, a 1 is assembled at bit position 12.

A given < usr_sym > defined in one .DALC pseudo-op may be redefined in another .DALC pseudo-op. The last definition will be the one assigned to <usr_sym >.

Notes:
(Continued)

If a three character symbol is defined using this pseudo-op, it can be used and followed immediately by one or two letters of the following format:

<sym> [Z, O, C] [L, R, S]

Use of these letters (or letter) will cause bits 8 - 9, 10 - 11 to be set as follows:

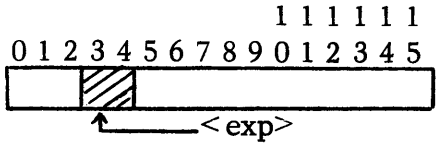
Mnemonic	Bits 8 - 9	Bits 10 - 11
L	01	
R	10	
S	11	
Z		01
O		10
C		11

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DIAC

Syntax: .DIAC Δ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. The use of the symbol implies the formatting of an instruction requiring an accumulator. One field is thus required. The field is assembled as shown below.



The format in which the semi-permanent symbol is used is:

<semi_sym>Δ <exp>

Examples:

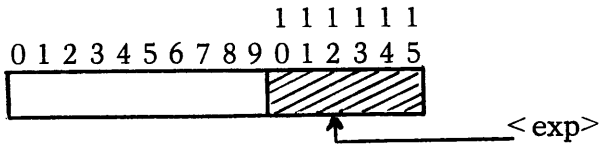
```
        000430 .DIAC RPT=000430


 00000 010430 RPT 2

F00001 000430 RPT 0,0  } Illegal number of expressions
F       000430 RPT     }
```

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DIO

Syntax: .DIO△ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. The use of the symbol implies the formatting of an I/O instruction without an AC field. One field is required; it is assembled as shown below.



The format in which the semi-permanent symbol is used is:

<semi_sym>△<exp>

Examples:

```
        063400 .DIO  SKION=063400

F       063400 SKION

 00002 063402 SKION 2

F00003 063402 SKION 2,3
```

Note: If a three character symbol is defined using this pseudo-op, it can be used and followed immediately by a single letter S, C, or or P. The use of one of these letters causes bits 8-9 of the statement word to be set as follows:

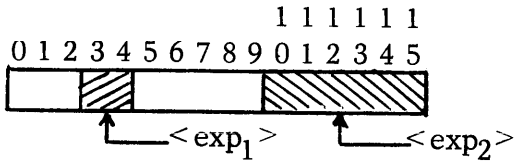
Mnemonic	Bits 8-9
S	01
C	10
P	11

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .DIOA

Syntax: .DIOA△ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. The use of the symbol implies the formatting of an I/O instruction with two required fields. The fields are assembled as shown below.



The format in which the semi-permanent symbol is used is:

<semi_sym>△<exp$_1$>△<exp$_2$>

Example:

```
       060500 .DIOA DIA. =060500
              .NREL
00000'070610 DIA   2,TTI

00001'060645 DIA   0,45
```

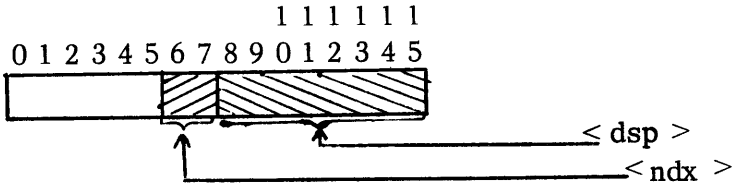
Note: If a three character symbol is defined using this pseudo-op, it can be used and followed immediately by a single letter S, C, or P. The use of one of these letters causes bits 8-9 of the statement word to be set as follows:

Mnemonic	Bits 8-9
S	01
C	10
P	11

Pseudo-op: .DMR

Syntax: .DMR △ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym>as a semi-permanent symbol having the value of <stmt>. In addition, the symbol implies the formatting of an MRI instruction with either one or two required fields (an address or a displacement and index). The fields are assembled as shown below.



The formats in which the semi-permanent symbol is used are:

<semi_sym> △ < dsp >
<semi_sym> △ < dsp > △ < ndx >

The <dsp > and <ndx > fields are set according to the format used and a set of addressing rules as described in Chapter 3.

Examples:

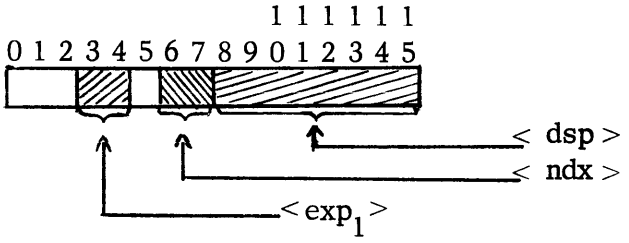
```
       000000 .DMR JMP=000000

              .NREL
00000'000402 JMP .+2

00001'005400 JSR 0,3
00002'003001 JMP @1,2
00003'000401 JMP 1,1
```

Note: The atom @ may be specified anywhere as a break character. If seen, a 1 is assembled at bit position 5.

Pseudo-op: .DMRA

Syntax: .DMRA △ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. In addition, the symbol implies the formatting of an MRI instruction with either two or three required fields. The first field specifies an accumulator. Where there are three fields, the second and third fields are a displacement and index respectively. Where there are two fields, the second is an implied address. The fields are assembled as shown below.



The formats in which the semi-permanent symbol is used are:

<semi_sym>△ <exp$_1$>△<exp$_2$>
<semi_sym>△ <exp$_1$>△<exp$_2$>△ <exp$_3$>

The <dsp> and <ndx> fields are set according to the format used and a set of addressing rules as described in Chapter 3.

Examples:

```
020000 .DMRA LDA=20000
       .NREL
035400 LDA 3,0,3
030402 LDA 2,.B+1
000004 .B: .BLK 4
```

Note: The atom @ may be specified anywhere as a break character. If seen, a 1 is assembled at bit position 5.

Pseudo-op: .DUSR

Syntax: .DUSR△ <usr_sym>=<stmt>

Purpose: This pseudo-op defines <usr_sym> as a semi-permanent symbol having the value of <stmt>. Unlike other semi-permanent symbols, a symbol defined by a .DUSR is merely given a value and has no implied formatting. It may be used anywhere a single precision operand would be used.

Example:

```
                .ZREL

       000025 .DUSR B=25

       000250 .DUSR C=B*10

                .NREL
00000'177555 B-C
00001'006712 B*C+2
```

SYMBOL TABLE PSEUDO-OPS (Continued)

Pseudo-op: .XPNG

Syntax: .XPNG

Purpose: This pseudo-op removes all macro definitions and all symbol definitions, except permanent, from the assembler's symbol table. .XPNG is used primarily as follows:

1. The programmer writes a program containing .XPNG followed by definitions of any semi-permanent symbols.

2. The program is assembled using the global switch S to the MAC command. This causes the assembler to terminate after pass 1.

3. The programmer can then use the assembler containing permanent symbols and those semi-permanent symbols defined by the programmer in step 2.

Example:

```
        .TITL XP
        .XPNG
020000  .DMRA LDA=20000
040000  .DMRA STA=40000

        .END


  MAC/S XP

  R
```

The assembler's symbol table now contains LDA and STA.

LOCATION COUNTER PSEUDO-OPS AND VALUES

Pseudo-op: .BLK

Syntax: .BLK△ <exp>

Purpose: This pseudo-op allocates a block of storage. <exp> is the number of words to be reserved. The current location counter is incremented by < exp > .

Example:

```
                    .NREL
00000'044403 STA 1,.F+1
00001'040403 STA 0,.F+2


00002'000004 .F: .BLK 4
00006'000000 .F1: 0
00007'000000 .F2: 0
```

Pseudo-op: .LOC

Syntax: .LOC△ <exp>

Purpose: This pseudo-op changes the setting of the current location counter to the value and relocation property given by <exp>.

Value: The current location counter value and relocation property. See, however, the exception given below.

Default: Absolute zero.

Example:

```
00000 000000 A: 0

               .NREL

00000'000000 NO: 0

               .ZREL

00000-000000 Z: 0

       000100 .LOC 100
 00100 000000 A
U00101 000000 B
U00102 000000 C

       000001 .LOC A+1
 00001 000000 A

       000001-.LOC Z+1
 00001-000000 A
```

Exception: If .LOC is pushed to the assembler variable stack (see VARIABLE STACK PSEUDO-OPS AND VALUES) and subsequently used to restore the location counter, e.g.,

```
.PUSH .LOC


.LOC .POP
```

then the value is ignored and only the relocation property is changed. This allows the user to save the current relocation mode within a macro and restore it correctly without affecting the relative location counter value which may have been altered within the macro.

LOCATION COUNTER PSEUDO-OPS AND VALUES (Continued)

Symbol: .

Value: The symbol . has the value and relocation property of the current location counter.

Example:

```
                .NREL


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

LOCATION COUNTER

Pseudo-op: .NREL

Syntax: .NREL

Purpose: This pseudo-op causes subsequent source statements to be assembled using normally relocatable addresses. If NREL mode is exited during assembly, the current .NREL value is maintained and used if NREL mode is entered again.

Examples:

```
                    .
                    .
                    .
00003  177775 M:  -3
       000100 .LOC 100
00100  000003 M
                .NREL
00000'000003 .A: M
00001'002777 JMP @.A

       000200 .LOC 200
00200  000003 M
                .NREL
00002'000003 M              ;note next available NREL address
```

LOCATION COUNTER

Pseudo-op: .ZREL

Syntax: .ZREL

Purpose: This pseudo-op causes subsequent source statements to be assembled using page zero relocatable addresses. If ZREL mode is exited during assembly, the current .ZREL value is maintained and is used if .ZREL mode is entered again.

Example:

```
00000 000000 AL: 0
                .ZREL
00000-000000 Z: 0
00001-000000 ZL: 0
       000100 .LOC 100
00100 000000 AL
                .ZREL
00002-000000 AL
```

INTERPROGRAM COMMUNICATION PSEUDO-OPS

Pseudo-op: .COMM

Syntax: .COMM Δ <usr_sym > Δ <exp >

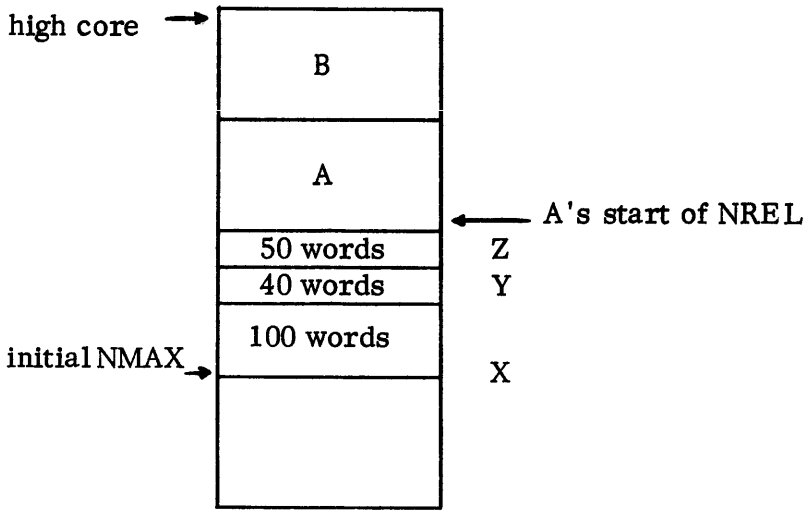
Purpose: This pseudo-op is used to reserve an area for interprogram communication having the name <usr_sym > and the size in words as given by <exp >. This area will be reserved by the first routine loaded that declared the named <usr_sym >. The area is reserved at NMAX, immediately preceding any NREL code loaded. Further routines loaded declaring the same symbol share this original area, provided the sizes specified are the same.

Since <usr_sym > is an entry in the program, it cannot be redefined elsewhere in the program The <usr_sym > may be referenced from other programs loaded together using .EXTN, .EXTD, or .GADD pseudo-ops.

Example:

```
        .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:



Pseudo-op: .CSIZ

Syntax: .CSIZ △ <exp>

Purpose: This pseudo-op specifies the size in words of a program area to be used for interprogram communication.

The assembler evaluates <exp> and passes this value to the loader. More than one .CSIZ pseudo-op may appear in a program. At load time, the largest value specified by all .CSIZ blocks is used to set USTCS at the termination of the load.

Example:

```
          .TITL A
000020    .CSIZ 20          20 words allocated
            .
            .
            .
          .END


          .TITL X
000050    .CSIZ 50          50 words allocated in communicating
            .                 program to be loaded with A.
            .
            .
          .END


RLDR A X
   A
   X
     NMAX 001037
     ZMAX 000050
     CSZE 000050          At load time the largest area is selected
      EST                   for USTCS (prints as CSZE).
      SST
```

Pseudo-op: .ENT

Syntax: .ENT △<usr_sym$_1$>[<usr_sym$_2$> ... △ <usr_sym$_n$>]

Purpose: This pseudo-op declares each <usr_sym> as a symbol that is defined within the program and that may be referenced by separately assembled programs.

A <usr_sym> appearing in a .ENT pseudo-op must be defined as a user symbol within the program in which it is declared. This symbol must be unique from entries defined in other programs loaded together to form a save file. If not, the loader will issue a message indicating multiply defined entries.

Entries are referenced in separately assembled programs using one of the following pseudo-ops:

.EXTD
.EXTN
.GADD
.GLOC

Example:

```
                .TITL A
                .ENT B,.C
                .EXTN C
                .ZREL
00000-177777    .C: C
                .NREL
00000'006000-B: JSR @.C
                 .
                 .
                 .
                .END
```

Pseudo-op: .ENTO

Syntax: .ENTO △<usr_sym>

Purpose: This pseudo-op is used when a program is to become an overlay within an overlay segment. The pseudo-op causes the name <usr_sym> to be associated with the node number and overlay number assigned to the particular overlay. The overlay may then be referenced from another program by <usr_sym> . <usr_sym> must be declared as an .EXTN in the referencing program.

Caution: <usr_sym> cannot appear elsewhere in the program in which is declared as the name of an overlay, since its value is assigned at load time.

Example:

```
.TITLE TIME
.ENTO METER
   .
   .
   .
```

Pseudo-op: .EXTD

Syntax: .EXTD △<usr_sym$_1$>{< usr_sym$_2$> ... △ <usr _sym$_n$ >}

Purpose: This pseudo-op declares each <usr_sym> as a symbol which may be referenced by the program but which is defined in some other program. The <usr_sym> must be declared by an .ENT pseudo-op in the program in which it is defined.

Any symbol declared as an .EXTD may be used as an address or displacement of a memory reference instruction. In addition, it can be used to specify the contents of a 16-bit storage word.

If used as a page zero address or as a displacement, it is the programmer's responsibility to insure the value of the entry meets the specific requirements, i.e.,

$$0 \leq \text{page\_zero\_adr} \leq 377$$

$$-200 \leq \text{displacement} \leq 200$$

Example:

```
                .TITL B
                .EXTD .CR
                .ZREL
                 :
                 :
00000-000001$.CR
00001-006001$.DONE: JSR @.CR
                 :
                 :
                .END
```

Pseudo-op: .EXTN

Syntax: .EXTN △ <usr_sym$_1$> {< usr_sym$_2$> ... △ < usr_sym$_n$>}

Purpose: This pseudo-op declares each <usr_sym> as a symbol that is externally defined in some other program but may be referenced by the current program. The <usr_sym> must be declared using an .ENT pseudo-op in the program in which it is defined.

A symbol declared as a .EXTN can be used only to specify the contents of a 16-bit storage word. The value at load time is therefore a number in the range 0 to 2^{16}-1.

Example:

```
                .TITL B
                .EXTN C
                .ZREL
00000-177777    .C:  C
```

Pseudo-op: .EXTU

Syntax: .EXTU

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

Example:

```
                    .TITL A13
                    .EXTU

    00000 024001$LDA 1,B

                    .END
```

Pseudo-op: .GADD

Syntax: .GADD Δ <usr_sym > Δ <exp>

Purpose: This pseudo-op generates a storage word whose contents is resolved at load time. The value of <usr_sym > is searched for at load time and, if found, its value plus the value of the < exp > are added to form the contents of the storage word. If the <usr_sym > is not found, a loader error results and the storage word will contain just the value of <exp >.

The <usr_sym > must be a symbol defined in some separately assembled program and appearing in that program in an .ENT, .ENTO, or .COMM pseudo-op. .GADD may thus be used in a similar manner to .EXTN with the following differences:

A user symbol appearing 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 .

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

Example:

```
                .TITL Y
                .ENT A
        000200  .LOC 200          Value of A is 200.
                A:
                .END

                .TITL X
        000100  .LOC 100
U00100  000007  .GADD A, 3+4      Value of 7 assigned on assembly.
                .END
```

When loaded in the order Y - X, the value stored will be 207. If X is loaded first, an error message will result and the value stored will be 7.

Pseudo-op: .GLOC

Syntax: .GLOC Δ <usr_sym >

Purpose: This pseudo-op begins a 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 will result.

Within the block, there can be no external references, no label definitions and no label references.

Example:

```
                .TITL A
      000003    .COMM MYAREA,3
                  :
                  :
                .END

                .TITL B          ← Program B, loaded after program A will
                .GLOC MYAREA       initialize program A's named common area.
00000'000001    1
00001'000002    2
00002'000003    3
                .NREL
                  :
                  :
                .END
```

TEXT PSEUDO-OP AND VALUES

Pseudo-op:
.TXT
.TXTE
.TXTF
.TXTO

Syntax:
.TXT Δ α <string> α
.TXTE Δ α <string> α
.TXTF Δ α <string> α
.TXTO Δ α <string> α

Purpose:

These pseudo-ops cause the assembler to scan the input following the character α up to the next occurrence of the character α in string mode. The character α may be any character not used within the string except null, line feed, or rubout. α delimits but is not part of the string. Carriage return and form feed may be used to continue the string from line to line or page to page, but are not stored as part of the text string.

Every two bytes generate a single storage word containing the ASCII codes for the bytes. Storage of a character of a string requires seven bits of an eight-bit byte. The leftmost (parity) bit may be set to 0, 1, even parity, or odd parity as follows:

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

The packing mode can be altered using the .TXTM pseudo-op. If an even number of bytes are assembled, the null word following these packed bytes can be suppressed by the .TXTN pseudo-op. See pages 4-33 and 4-32.

Within the string, angle brackets can be used to delimit an arithmetic expression. The expression will be evaluated, masked to seven bits, and the eighth bit set as specified by the pseudo-op. Note that no logical operators are permitted within the expression. This is the only means, for example, to store a carriage return as part of the text string.

```
.TXT "LINE 1<15>" /
```

Default: Bytes are packed right/left, and a null byte is generated as the terminating byte.

Example:

```
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
```

Pseudo-op or Symbol: .TXTM

Syntax: .TXTM△ <exp>

Purpose: This pseudo-op changes the packing of bytes generated using the text pseudo-ops, .TXT, .TXTE, .TXTF, or .TXTO . If <exp> evaluates to zero, bytes are packed right/left; if <exp> does not evaluate to zero, bytes are packed left/right.

Value: The value of .TXTM is the value of the last expression used within the .TXTM pseudo-op.

Default: Bytes are packed right/left.

Example:

```
       000000 .TXTM 0
00000  041101 .TXT #AB CD#
       041440
       000104
00003  000000 (.TXTM)
       000001 .TXTM 1
00004  040502 .TXT #AB CD#
       020103
       042000
00007  000001 (.TXTM)
```

Pseudo-op or Symbol: .TXTN

Syntax: .TXTN Δ <exp >

Purpose: This pseudo-op determines whether or not a string that contains an even number of characters will terminate with a word consisting of two zero bytes. (When the number of characters in the string is odd, the last word contains a zero byte in all cases.)

If <exp > evaluates to zero, all text strings containing an even number of bytes will terminate with a full word zero. If <exp > does not evaluate to zero, any text string containing an even number of bytes terminates with a word containing the last two characters of the string.

Value: The value of .TXTN is the value of the last expression used within the .TXTN pseudo-op.

Default: The string terminates with a zero word.

Example:

```
        000000 .TXTN 0
00000   031061 .TXT /1234/
        032063
        000000
00003   000000 (.TXTN)
        000001 .TXTN 1
00004   031061 .TXT /1234/
        032063
00006   000001 +.TXTN
```

FILE TERMINATING PSEUDO-OPS

Pseudo-op: .END

Syntax: .END [Δ <exp>]

Purpose: This pseudo-op terminates a source program, providing an end of program indicator for the loader. The <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.)

Example:

```
        .TITL GETCT

102400 GETCT: SUB 0,0 ;INITIATE FOR STARTUP

        .END GETCT
```

Pseudo-op: .EOT

Syntax: .EOT

Purpose: This pseudo-op is used to indicate the end of an input file but not the end of input source. End of file from an input file is an implicit .EOT if other source files follow; end of file from the last input file is an implicit .END.

Example:

```
.TITL PGM
   :
   :
 .EOT
```

REPETITION AND CONDITIONAL PSEUDO-OPS

Pseudo-op: .DO

Syntax: .DO△ <exp>

Purpose: This pseudo-op causes the source program lines between the .DO and the corresponding .ENDC to be assembled the number of times given by <exp>.

Examples:

```
       000000 I=0
       000004 .DO 4
00000  100000 1BI
       000001 I=I+1
              .ENDC
00001  040000 1BI
       000002 I=I+1
              .ENDC
00002  020000 1BI
       000003 I=I+1
              .ENDC
00003  010000 1BI
       000004 I=I+1
              .ENDC
```

Listing

Source Program

```
I=0
.DO 4      ← loop is assembled
1BI          4 times
I=I+1
.ENDC
```

```
       000003 A=3

       000000 .DO 4==A          ← Relational expression evaluates to 0 (false).
              5
              2
              .ENDC

       000001 .DO 4==(A+1)      ← Relational expression evaluates to 1 (true),
00004  000005 5                    so loop is assembled once.
00005  000002 2
              .ENDC
```

Note: The .DO's may be nested to any depth, the innermost .DO corresponding to the innermost .ENDC, etc.

Pseudo-op:
.IFE
.IFG
.IFL
.IFN

Syntax:
.IFE△ <exp>
.IFG△ <exp>
.IFL△ <exp>
.IFN△ <exp>

Purpose: These pseudo-ops cause the statements following to be assembled if the condition defined in the pseudo-op is met. The pseudo-ops define the following conditions:

.IFE△ <exp>	Assemble if <exp> equals 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 value field of the listing is 1 if the condition is true and 0 if the condition is false.

Example:

```
       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.

Note: The .IF's may be nested to any depth, the innermost .IF corresponding to the innermost .ENDC, etc. Note that all .IF conditions are degenerate forms of .DO's. For example:

.IFG A is equivalent to .DO A > 0

Pseudo-op: .ENDC

Syntax: .ENDC

Purpose: This pseudo-op terminates statements for repetitive assembly (statements following a .DO pseudo-op) or statements whose assembly is conditional (statements following one of the pseudo-ops: .IFE, .IFG, .IFL, .IFN).

Example:

```
I=1
.DO 5
I*I
I=I+1
.ENDC
```

MACRO PSEUDO-OPS AND VALUES

Pseudo-op: .MACRO

Syntax: .MACRO△<usr_sym>

Purpose: This pseudo-op defines <usr_sym> as the name of the macro definition that follows. Any line or lines that follow are part of the macro definition up to the first % character encountered.

Once defined, <usr_sym> can be used to call the macro.

Example:

```
                .MACRO TEST         ;<USR-SYM> IS TEST
                ↑1
                ↑2                  ;MACRO DEFINITION
                ↑3
                %

                TEST 4,5,6          ;MACRO CALL WITH ARGS 4,5,6
00000 000004 4
00001 000005 5
00002 000006 6

      000020 .RDX 16
                TEST 0A,0B,0C       ;MACRO CALL WITH ARGS 0A,0B,0C
00003 000012 0A
00004 000013 0B
00005 000014 0C
```

Symbol: .ARGCT

Value: .ARGCT has as a value the number of actual arguments specified by the most recent macro call. If the symbol is used outside a macro, its value is -1.

Example:

```
                .NREL
                .MACRO X
                ↑1+↑2
                .ARGCT
                %

                X 4,5           ;CALL HAS TWO ARGS
00000'000011    4+5
00001'000002    .ARGCT          ;VALUE OF .ARGCT IS TWO
```

Symbol: .MCALL

Value: .MCALL has value 1 if the macro in which it appears has been called previously on this assembly pass, and value 0 if this is the first call on this pass. If used outside a macro, its value is -1.

Example:

```
.MACRO X
.DO .MCALL
JSR @.X ;JSR IF NOT FIRST CALL
.ENDC
.DO .MCALL ==Ø ;IF FIRST CALL , GENERATE SUBROUTINE
.PUSH .LOC ;SAVE LOCATION COUNTER
.ŻREL
.X: X ;POINTER TO SUBROUTINE
.LOC .POP ;RESTORE LOCATION COUNTER
JSR X ; CALL X
JMP XEND ; JUMP PAST X
X:
'
.
.           ;CODE FOR X
JMP Ø,3 ;RETURN
XEND:
.ENDC
%
```

LISTING PSEUDO-OPS AND VALUES

Pseudo-op: .NOCON

Syntax: .NOCON△ <exp>

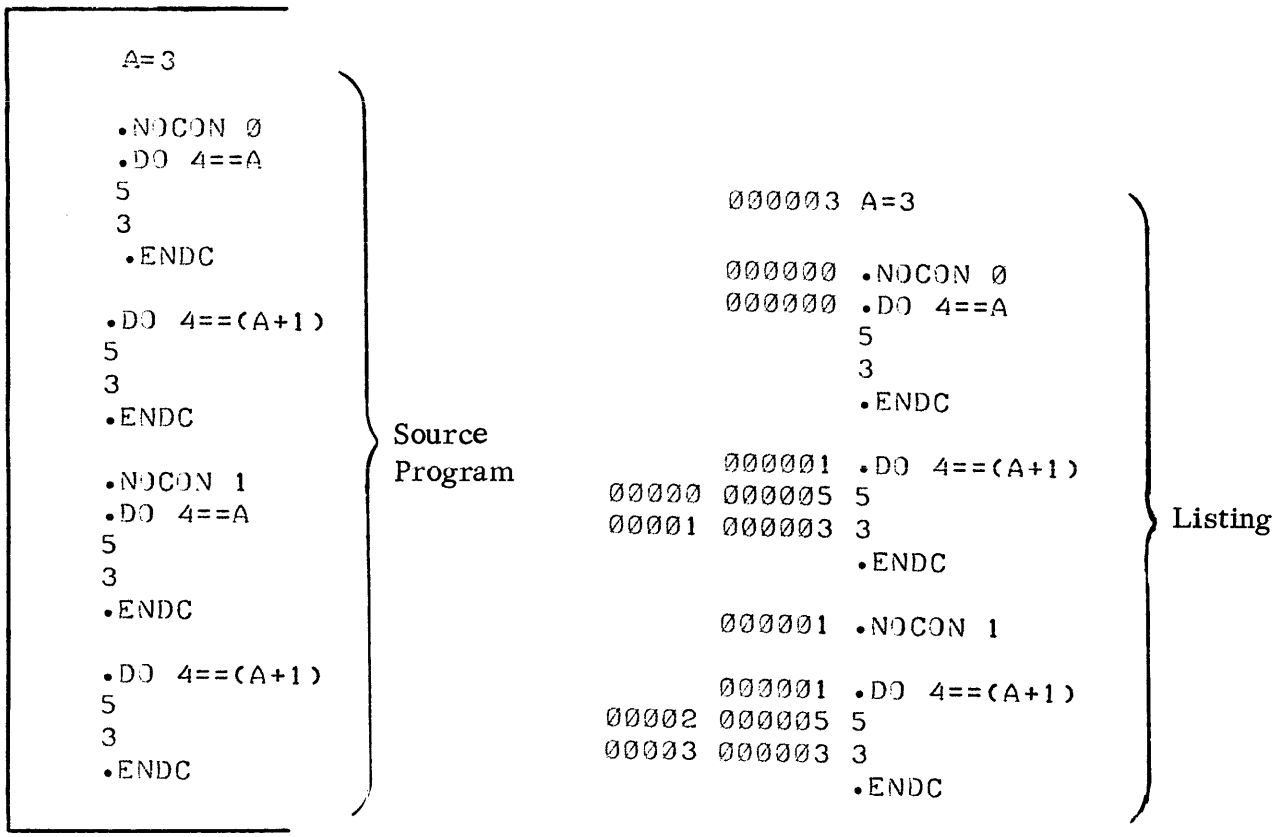
Purpose: This pseudo-op either inhibits or permits listing of those conditional portions of the source program that do not meet the conditions given for assembly. If the value of <exp> ≠ 0, listing is inhibited; if the value of <exp> = 0, listing is permitted.

Conditional portions of the source program that would be assembled are not affected by the .NOCON pseudo-op.

Value: The value of .NOCON is the value of the last expression used within the .NOCON pseudo-op.

Default: Listing is permitted.

Examples:



Pseudo-op: .NOLOC

Syntax: .NOLOC Δ <exp >

Purpose: This pseudo-op is used to inhibit listing of lines which do not include a location field. If the value of <exp > is not equal to zero, listing is inhibited; if the value of <exp > is equal to 0, listing occurs.

Value: The value of .NOLOC is the value of the last expression used within the .NOLOC pseudo-op.

Default: Listing is permitted.

Pseudo-op: .NOMAC

Syntax: .NOMAC△ <exp>

Purpose: This pseudo-op is used to inhibit the listing of macro expansions. If <exp> evaluates to zero, macro expansions will be listed; otherwise, macro expansions are inhibited.

Value: The value of .NOMAC is the value of the last expression used within the .NOMAC pseudo-op.

Default: Expansions are listed.

Examples:

```
                .MACRO OR

                COM ↑1,↑1
                AND ↑1,↑2
                ADC ↑1,↑2
                %

       000001   .NOMAC 1        ;EXPANSION INHIBITED

                OR [1,2]


       000000   .NOMAC 0        ;EXPANSION PERMITTED

                OR [3,0]
00003 174000    COM 3,3
00004 163400    AND 3,0
00005 162000    ADC 3,0

                .MACRO TEST
                5
                6
                .NOMAC 1        ;MACRO EXPANSION MAY BE INHIBITED OR PERMITTED
                7               ;AT ANY TIME WITHIN THE MACRO
                8
                %

                TEST
00000 000005    5
00001 000006    6

                .END
```

Note: .NOMAC can be used within a macro definition to selectively inhibit listing.

VARIABLE STACK PSEUDO-OPS AND VALUES

Pseudo-op: .PUSH

Syntax: .PUSH△ < exp >

Purpose: This pseudo-op allows the programmer to save the value and relocation properties of any legal assembler expression on an internal assembler stack. Additional expressions may be pushed until the stack space is exhausted. The stack is referenced by the permanent symbols .POP and .TOP. As with any push down stack, the last expression "pushed" is the first expression to be "popped."

Example: The current value of the input radix may be saved, its value altered, and then restored by the following statements.

```
000010 .PUSH .RDX
000012 .RDX 10


000010 .RDX .POP
```

Symbol: .POP

Value: The value of .POP is the value and relocation property of the last value pushed on the variable stack (.PUSH pseudo-op). In addition, use of .POP causes the value and relocation property to be popped.

Default: If there are no values on the variable stack, .POP has a value of absolute zero and a zero flag will be given the line in which it is used.

Example:

```
000025 A=25
000025 A

000025 .PUSH A
000015 A=15
000015 A

000025 A=.POP
000025 A
       .END
```

Symbol: .TOP

Value: The value of .TOP is the value and relocation property of the last value pushed to the variable stack. .TOP differs from .POP in that use of the symbol does not pop the last pushed value from the stack. If no values are pushed, absolute 0 is returned and the 0 flag is given.

Example:

```
       000020 .PUSH 20

00000  000020 .TOP
00001  000020 .TOP
```

PASS VALUE

Symbol: .PASS

Value: .PASS has a value of zero on pass 1 and a value of one on pass 2 of assembly.

Example:

```
      000020 .RDX 16
00000 000000 C: 0
00001 000021 D: 08+9
00002 032001 LDA 2,@D
00003 000001 .PASS                ;Value on listing pass (pass 2) is
                                  ; always 1.
```

CHAPTER 5

EXTENDED CAPABILITIES OF THE MACRO ASSEMBLER

THE MACRO FACILITY

The macro facility allows a string of source characters, perhaps consisting of many lines, to be named and subsequently referenced by name. In addition, the definition string may specify formal arguments within the string that are replaced by actual arguments whenever the macro is expanded. The macro is defined only once within a program but may be referenced (called) any number of times after the definition.

The macro definition associates a user symbol with the definition. Then whenever the macro symbol is encountered in assembly of the source program, the definition is substituted for the symbol. The substitution is called macro expansion, and the encounter of the symbol is a macro call.

Macro Definition

The programmer writes a macro definition once. The macro definition can then be substituted in his program anywhere the appropriate macro name is encountered during assembly. A macro definition has the form:

```
.MACRO △ <usr_sym> )
<macro_definition_string> %
```

where: <usr_sym> is the name to be used in the macro call to identify the macro that is to be expanded into the program at that point.

<macro_definition_string> is a string of ASCII characters to be substituted for the macro call.

% terminates <macro_definition_string> and is not part of the definition.

Within <macro_definition_string> two characters (← and ↑) have special meanings. The back arrow (←) is ignored but causes the character immediately following to be stored without any interpretation. It would be used, for example, if the user wishes a percent sign (%) to be part of the macro definition string. (Normally, the % terminates the macro definition.) If the user wants to write a <macro_definition_string> that will be

ABC%D

he would write <macro_definition_string> as:

Macro Definition (Continued)

ABC ← %D

Similarly, if the user wishes to write a <macro_definition_string> that will be

X ← YZ

he would write the <macro_definition_string> as:

X ← ← YZ

The back arrow convention can be used for any ASCII character. For example, if the user writes either

X or ← X

the character will be read as X in either case. However, the back arrow convention is most often used for characters that will otherwise be interpreted, such as %, ↑, and ← itself.

The second character having a special meaning is the up arrow (↑). An up arrow is followed by an alphanumeric character representing a formal argument interpreted as follows:

↑n where n is 1-9

↑a where a is A-Z

A digit following ↑ represents the positional value of an actual argument in the argument list of the macro call that will replace the formal argument ↑n wherever it appears within the macro definition. For example, if the formal argument ↑3 occurs in the <macro_definition_string> , then it will be replaced by the third argument in the macro call as described in the next section. (A zero following ↑ is unconditionally replaced by the null string.)

A letter following ↑ is a symbol whose value is looked up when macro expansion occurs. The value of the symbol is used as the positional value of the actual argument to the macro call that is to replace ↑a wherever it appears within the macro definition.

The carriage return following <user_sym> is required to distinguish <user_sym> from the macro definition string. Except for the characters previously noted (← , ↑ , %), all characters in the <macro_definition_string> are returned during macro expansion exactly as written. For example, if the definition consists of a string that is to be expanded into several lines of source language, each line must be terminated by a carriage return.

If the definition consists of a single expression that is to be substituted as, for example, the second operand of a three-operand instruction, the macro definition string cannot contain carriage returns, comments, etc. To define an expression within a macro definition string, the expression must fall within the line limitations of the assembler (132 characters maximum).

The % terminating the macro definition string will appear in the macro definition as the first character of the last line of the macro definition if the macro definition is one or more complete lines of assembly source program. If the definition is not a complete line , the % will appear immediately following the string that constitutes the definition.

The user symbol that names the macro must follow assembler rules for user symbols.

Examples of macro definitions are:

```
.MACRO  T↵
LDA  0,0,3↵
MOV  0,0,SNR↵
%

.MACRO EXP↵
TEST↑1+↑2%

.MACRO  COM↵
;TEST FOR 95 ← % DONE↵
%
```

The definition of a macro may be temporarily terminated and then continued. This is especially useful if a first macro is used to define a second macro. The first macro may terminate definition of the inner macro temporarily, assign new equivalences, and continue. The macro VFD given later illustrates this continuation property.

Syntactically, if a macro of the same name as the last defined macro is encountered, the second and subsequent "definitions" are appended, in order, to the first definition. For example,

```
    .MACRO TEST ↵
I = 0 ↵
%
    .MACRO TEST↵
J = 0 ↵
%
    is equivalent to:

    .MACRO TEST↵
I = 0 ↵
J = 0 ↵
%
```

Macro Calls

For a given macro definition, any number of macro calls of that definition may appear in the source program. A macro instruction consists of the user symbol given in a macro definition followed by any actual arguments to replace formal arguments in the macro definition string.

A macro call has one of the following forms:

(1) <mac_sym> ↵

(2) <mac_sym>Δ <string$_1$> ↑ <sp><string$_2$><sp>... <string$_n$>↑ ↵

(3) <mac_sym> ↑Δ↑ [<string$_1$>↑ <sp><string$_2$><sp>... <string$_n$>↑] ↵

where: <mac_sym> is the name of some macro definition.

Each <string$_i$> is an actual argument that is to be substituted for the appropriate formal argument during macro expansion.

The first form of the macro instruction presumes that there are no formal arguments within the macro definition. Forms two and three presume that one or more formal arguments must be replaced by actual arguments.

Substitution of actual arguments is accomplished by using <string$_1$> to replace every occurrence of ↑1 (or to replace ↑a where a evaluates to 1), <string$_2$> to replace every occurrence of ↑2, etc. If no formal arguments were specified in the definition, no arguments can be specified by the call. If more arguments are given by the call than specified by the definition, they are ignored.

The list of arguments of a macro call may either be enclosed in square brackets (form 3) or not enclosed in square brackets (form 2). The difference is that form 2 terminates with a carriage return before the first byte of the macro expansion, whereas form 3 does not. If a macro is to replace the index field of an instruction, form 3 should be used. In general, form 2 type calls are more common. For example:

Macro Definition	Alternative Instructions and their Expansions	
.MACRO D ↵ TEMP↑1%	STA 3, D[2]	← macro call
	STA 3, TEMP2	← after expansion

	D 2 ↵ +3	← macro call
	TEMP2+3	← after expansion

Macro Calls (Continued)

Argument strings, like text strings, may consist of any characters. Argument strings are separated by a single break character <sp>. The effect is that leading commas, spaces or tabulations may be part of the argument.* The argument string is terminated by the first <sp> encountered.

Listing of Macro Expansions

The manner of substituting a macro definition for a macro instruction follows the procedures just described. However, the listing output showing the expanded source text is not the same as the macro expansion used to generate the object file output. The listing will show both the macro call and the macro expansion, while the object file will contain only the object code equivalents of the macro expansion with the appropriate actual arguments. An example is:

```
.MACRO  DSP↓                    ←macro definition
↑1%

LDA     0,DSP [ 121 ], 3        ←source listing line with macro

LDA     0, 121, 3               ←expanded line to be translated to
                                   object file
LDA     0, DSP[ 121 ] 121, 3    ←expanded line as it appears in the
                                   listing file
```

The listing of macro expansions may be suppressed using the pseudo-op .NOMAC. If suppressed, the load instruction above would appear on the listing exactly as it appears in the source listing line.

```
                .MACRO .
                5
                LDA ↑1,↑2
                %

                . 0,4
00000 000005 5
00001 020004 LDA 0,4

       000001 .NOMAC 1
                . 0,4
       000000 .NOMAC 0

                . 0,4
00004 000005 5
00005 020004 LDA 0,4
                .END
```

Macro expansions are listed by default.

An expression evaluating to a value other than zero following .NOMAC causes suppression of listing of macro expansions.

An expression expansion evaluating to zero following .NOMAC restores the listing of macro expansions.

* This applies to all argument strings except for the first argument string of an argument list that is not enclosed by square brackets.

Macro Examples

A number of macro examples follow. Note that use of the recursive property of the macro, FACT, and the use of macro continuation and the special character ← within VFD.

The first example is a macro to compute the logical OR of two accumulators. Its call takes a form similar to an ALC instruction, i.e.,

OR△ <src><sp><des>

The source accumulator is unchanged by the call. Note also that actual arguments replace formal arguments within the comments.

Definition

```
; LOGICAL OR MACRO

; CALL:
;                   OR      <OP-0>,<OP-1>

; WHERE THE RESULT IS:

;                   <OP-0> .OR. <OP-1>


          .MACRO  OR
          COM     ↑1,↑1
          AND     ↑1,↑2             ; CLEAR "ON" BITS OF AC↑1
          ADC     ↑1,↑2             ; OR RESULT TO AC↑2
%
```

Calls

```
          .NREL

          OR      1,2
00000'124000      COM     1,1
00001'133400      AND     1,2       ; CLEAR "ON" BITS OF AC1
00002'132000      ADC     1,2       ; OR RESULT TO AC2
          OR      0,1
00003'100000      COM     0,0
00004'107400      AND     0,1       ; CLEAR "ON" BITS OF AC0
00005'106000      ADC     0,1       ; OR RESULT TO AC1
```

Macro Examples (Continued)

A somewhat more illustrative example is that of logical exclusive OR. This macro allows an optional third argument. If absent, AC3 is used as a temporary accumulator and is destroyed. If the third argument is given, it is used as a temporary storage location for saving and restoring AC3. The absence of an argument is conveniently tested for by making a comparison with the null string, e.g.,

```
.DO  '↑3' == ''
```

In addition, this macro saves the state of the no conditionals list option, turns them off, and restores the original state upon exit. Further, since a number of the listing control and variable stack manipulation pseudo-ops print, they have been suppressed using the ** atom. This enables a "clean" listing output.

Macro Examples (Continued)

```
                ; EXCLUSIVE OR

                ; CALL:
                ;                XOR     <SRC>,<DES>[,<TMP>]

                ;       IF NO <TMP>, AC3 IS USED AS THE TEMPORARY AC
                ;       IF <TMP>, MEM LOC <TMP> IS USED TO
                ;                SAVE AND RESTORE AC3


                        .MACRO  XOR
                **      .PUSH   .NOCON
                **      .NOCON  1
                **      .DO     '↑3'<>''
                        STA     3,↑3            ; SAVE AC3 IN ↑3
                **      .ENDC
                        MOV     ↑2,3
                        ANDZL   ↑1,3            ; 2*(AC↑1 .AND. AC↑2)
                        ADD     ↑1,↑2           ; AC↑1 + AC↑2
                        SUB     3,↑2            ; AC↑1 .XOR. AC↑2
                **      .DO     '↑3'<>''
                        LDA     3,↑3            ; RESTORE AC3 FROM ↑3
                **      .ENDC
                **      .NOCON  .POP

                %

                        .ZREL
00000-000001 TEMP:      .BLK    1

                        .NREL

                        XOR     1,2
00000'155000            MOV     2,3
00001'137520            ANDZL   1,3             ; 2*(AC1 .AND. AC2)
00002'133000            ADD     1,2             ; AC1 + AC2
00003'172400            SUB     3,2             ; AC1 .XOR. AC2

                        XOR     0,1,TEMP
00004'054000-           STA     3,TEMP          ; SAVE AC3 IN TEMP
00005'135000            MOV     1,3
00006'117520            ANDZL   0,3             ; 2*(AC0 .AND. AC1)
00007'107000            ADD     0,1             ; AC0 + AC1
00010'166400            SUB     3,1             ; AC0 .XOR. AC1
00011'034000-           LDA     3,TEMP          ; RESTORE AC3 FROM TEMP
```

Macro Examples (Continued)

The recursive property of macros is illustrated by the factorial macro, FACT. Its input consists of a variable, v, and an integer, i, where the following is computed:

$$v = i!$$

using the recursive formula

$$i! = i * (i-1)!$$

The macro expands as follows:

> Until the input integer becomes 1, the second conditional expands and recursively calls FACT. (Note that when return is finally made after these calls, the macro will return the string
>
> ↑2 = (↑1)*↑2)
>
> and terminate.) When the input becomes 1, the first conditional expands and terminates its expansion. This begins the succession of returns to each level at which a recursive call was made, in the process computing i !

```
                    .MACRO   FACT
                    .DO      ↑1==1
          ↑2=       1
                    .ENDC
                    .DO      ↑1<>1
                    FACT     ↑1-1,↑2
          ↑2=       (↑1)*↑2
                    .ENDC
          %
                    FACT     6,I
000000              .DO      6==1
          I=        1
                    .ENDC

000001              .DO      6<>1
                    FACT     6-1,I
000000              .DO      6-1==1
          I=        1
                    .ENDC

000001              .DO      6-1<>1
                    FACT     6-1-1,I
000000              .DO      6-1-1==1
          I=        1
                    .ENDC

000001              .DO      6-1-1<>1
                    FACT     6-1-1-1,I
000000              .DO      6-1-1-1==1
          I=        1
                    .ENDC

000001              .DO      6-1-1-1<>1
                    FACT     6-1-1-1-1,I
000000              .DO      6-1-1-1-1==1
          I=        1
                    .ENDC

000001              .DO      6-1-1-1-1<>1
                    FACT     6-1-1-1-1-1,I
000001              .DO      6-1-1-1-1-1==1
000001    I=        1
                    .ENDC

000000              .DO      6-1- -1- -1<>1
                    FACT     6-1-1-1-1-1-1,I
          I=        (6-1-1-1-1-1)*I
                    .ENDC
000002    I=        (6-1-1-1-1)*I
                    .ENDC
000006    I=        (6-1-1-1)*I
                    .ENDC
000030    I=        (6-1-1)*I
                    .ENDC
000170    I=        (6-1)*I
                    .ENDC
001320    I=        (6)*I
                    .ENDC
```

Macro Examples (Continued)

A macro to output "packed decimal" is given next. It illustrates a number of useful techniques for use within macros.

In packed decimal, each decimal digit is represented as a 4-bit binary nibble. The sign of the number always occupies the least significant nibble. The translation of of decimal to 4-bit binary is

decimal	4-bit binary
+	0011
-	0100
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

The input to PACK is the decimal string of digits, separated by spaces, followed by an explicit sign (+ or -) and the precision in Nova words. The macro produces the output least significant word first.

Some further explanation is necessary.

1. The input radix within the macro must be decimal. Therefore, it is saved, set to decimal, and restored within the macro body.

2. To present the output as 4-bit nibbles, the output radix within the macro must be hexadecimal. Therefore, the output radix is also saved, set to hexadecimal, and restored. Note the order of the save for these radices is the opposite of the order of the restore. (See .PUSH, .POP descriptions in Chapter 4.)

3. Many statements are assembled with each macro call, but expansion is inhibited except for the storage words assembled.

Macro Examples (Continued)

```
                ; PACKED DECIMAL

                ; CALL:
                ;                   PACK    D D ... D S,W
                ; WHERE D'S REPRESENT DIGITS, S THE SIGN (+ OR -), AND
                ; W THE NUMBER OF WORDS.


                        .MACRO  PACK
                **      .PUSH   .NOMAC
                **      .NOMAC  1
                        .PUSH   .RDX
                        .PUSH   .RDXO
                        .RDX    10
                        .RDXO   16
                I=      .ARGCT
                J=      I-1
                B=      11
                W=      3+("↑J-"+/2)
                J=      J-1
                        .LOC    .+↑I-1
                        .DO     ↑I
                        .DO     B+1/4
                W=      W+0↑JBB
                B=      B-4
                        .DO     J<>0
                J=      J-1
                        .ENDC
                        .ENDC
                **      .NOMAC  0
                        W
                **      .NOMAC  1
                W=      0
                B=      15
                        .LOC    .-2
                        .ENDC
                        .LOC    .+↑I+1
                        .RDXO   .POP
                        .RDX    .POP
                **      .NOMAC  .POP
                %

      000100            .LOC    100
                        PACK    1 2 3 4 5 +,3
00042   3453            W
00041   0012            W
00040   0000            W
                        PACK    1 2 3 4 5 -,3
00045   3454            W
00044   0012            W
00043   0000            W
                        PACK    6 5 4 3 2 1 -,4
00049   3214            W
00048   0654            W
00047   0000            W
00046   0000            W
```

Macro Examples (Continued)

```
                                PACK    3 2 7 6 8 +,6
    0004F    7683               W
01  0004E    0032               W
02  0004D    0000               W
03  0004C    0000               W
04  0004B    0000               W
05  0004A    0000               W
06
```

Macro Examples (Continued)

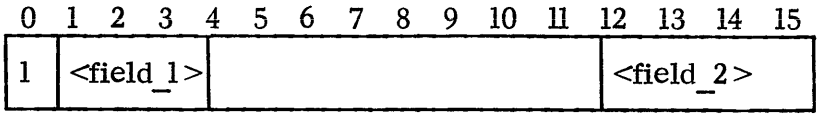
A powerful macro, used to associate a specified field layout with a given name, is given below. The macro, VFD, is used to define a new macro named as the first argument in the call to VFD. Subsequent use of the name given in the VFD call generates a 16 -bit storage word having a primary value to which fields are assembled as described in the call to VFD. The call is of the form:

```
VFD△ <type_name><sp><primary_value><sp><field1 right bit><sp>
     <field1 mask><sp>... <fieldi right bit><sp><fieldi mask><sp>...
```

The 3rd, 5th, ... arguments specify the rightmost bit positions of the 1st, 2nd, ... fields. The 4th, 6th, ... arguments specify the field masks for the 1st, 2nd, ... fields. To assemble the fields in the proper bit positions, with overflow and field zero checking, a call is made of the form:

```
<type_name>△ <field_1><sp><field_2><sp>...
```

The example below defines a <type_name> of SPECL. This name is for words of the following layout:



The definition of VFD is:

```
          .MACRO VFD

          I=4
                              .MACRO ↑1

                              **.PUSH .NOMA
                              **.NOMA 1
                              VALU=↑2
                              J=1
←%
          .DO .ARGCT/2-1
                              .MACRO ↑1

                              .IFN ↑I>=←↑J
                              MASK=↑I
                              DATA=←↑J
←%
          I=I-1
                              .MACRO ↑1

                                        .DO 15.-↑I
                                        MASK=MASK*2
                                        DATA=DATA*2
                                        .ENDC
←%
          I=I+1
                              .MACRO ↑1

                                        .IFN VALU&MASK
                                        ERROR [FIELD NON-ZERO]
                                        .ENDC

                                        .IFE VALU&MASK
                                        VALU=(VALU&(-MASK-1))+DATA
                                        .ENDC
                              .ENDC

                              .IFE ↑I>=←↑J
                              ERROR[FIELD OVERFLOW]
                              .ENDC
←%
          I=I+2
                              .MACRO ↑1

                              J=J+1
←%
          .ENDC
                              .MACRO ↑1

                              **.NOMA Ø
                              VALU
                              **.NOMAC          .POP
←%
%
```

Macro Examples (Continued)

```
          .MACRO  ERROR

          **.PUSH .NOMAC
          **.NOMA Ø
;*******************************

;     ↑1 ↑2 ↑3 ↑4 ↑5 ↑6 ↑7 ↑8 ↑9

;*******************************

          **.NOMAC          .POP
%
```

Macro Examples (Continued)

The call to define SPECL produces:

```
                    VFD     SPECL,100000,3,7,15.,17
000004              I=4
                                    .MACRO SPECL

                                    **.PUSH .NOMA
                                    **.NOMA 1
                                    VALU=100000
                                    J=1
                %
000002              .DO .ARGCT/2-1
                                    .MACRO SPECL

                                    .IFN 7>=^J
                                    MASK=7
                                    DATA=^J
                %
000003              I=I-1
                                    .MACRO SPECL

                                            .DO 15.-3
                                            MASK=MASK*2
                                            DATA=DATA*2
                                            .ENDC
                %
000004              I=I+1
                                    .MACRO SPECL

                                            .IFN VALU&MASK
                                            ERROR [FIELD NON-ZERO]
                                            .ENDC

                                            .IFE VALU&MASK
                                            VALU=(VALU&(-MASK-1))+DATA
                                            .ENDC
                                    .ENDC
```

Macro Examples (Continued)

```
                                    .IFE 7>=^J
                                    ERROR[FIELD OVERFLOW]
                                    .ENDC
        %
000006          I=I+2
                                    .MACRO SPECL

                                    J=J+1
        %
                .ENDC
                                    .MACRO SPECL

                                    .IFN 17>=^J
                                    MASK=17
                                    DATA=^J
        %
000005          I=I-1
                                    .MACRO SPECL

                                            .DO 15.-15.
                                            MASK=MASK*2
                                            DATA=DATA*2
                                            .ENDC
        %
000006          I=I+1
                                    .MACRO SPECL

                                            .IFN VALU&MASK
                                            ERROR [FIELD NON-ZERO]
                                            .ENDC

                                            .IFE VALU&MASK
                                            VALU=(VALU&(-MASK-1))+DATA
                                            .ENDC
                                    .ENDC

                                    .IFE 17>=^J
                                    ERROR[FIELD OVERFLOW]
                                    .ENDC
        %
000010          I=I+2
                                    .MACRO SPECL

                                    J=J+1
        %
                .ENDC
                                    .MACRO SPECL

                                    **.NOMA 0
                                    VALU
                                    **.NOMAC        .POP
        %
```

Macro Examples (Continued)

And, finally, calls of SPECL for fields having values of 1,1 and 7,17 gives:

```
                        SPECL   1,1
0000001 1100001                         VALU
                        SPECL   7,17
0000002 1700017                         VALU
```

GENERATED LABELS

In non-string mode, each occurrence of the character $ is replaced by three characters from the set Ø-9, A-Z. The three characters are determined by converting a count of the number of macro calls in radix 36 to ASCII. In nested macros, the replacement string for $ in the outer macro is saved and restored upon completion of the expansion of the inner macro. $ can therefore be used, for example, to generate unique labels in macros. When used in labels,$ should generally not be the first character as the first replacement character may be a digit.

Example:

```
      .MACRO X
X$:   ↑1
%
      .MACRO Y
Y1$:  ↑1+↑2
      X[↑Z]
Y2$:  ↑3+↑4
%
      .NREL
A$:   123
      Y[1,2,3,4]
B$:   456
      Y[5,6,7,8.]
```

LITERALS

All memory reference instructions must specify an address field. This address is used to:

1. Access the contents of the memory location in the case of an LDA.
2. Modify the memory location in the case of an STA, ISZ, or DSZ.
3. Transfer control in the case of a JMP or JSR.

Often, however, the programmer merely wishes to specify the contents of a memory location and is not concerned about its address. This is referred to as a literal reference.

Literals are permitted for all memory reference instructions. The macro assembler dumps these literals and assigns memory locations using the first and subsequent .ZREL locations available after pass 1. Therefore, all literal references are directly addressable.

The syntax of a literal reference is as follows:

< mem_ref > [< ac >,] = < statement >

Note that a literal may be any expression or statement.

Frequently literals are used to load an accumulator with some constant. For example,

```
LDA  1,=3
```

indicates that the programmer wishes to load AC1 with the value 3.

Expressions are acceptable

```
LDA  0,=1B0+"A/2
```

indicates that the programmer wishes to load AC0 with the value 40040.

Statements are also acceptable:

```
LDA  1,=SUBZ# 2,3,SNC
```

indicates the programmer wishes to load AC1 with the value 156433.

The previous examples have given absolute expressions as literals. However, any

relocatable expression is legal.

```
        .NREL
        .
A:      .
        .
        LDA  2,=A
```

indicates the value of "A" is to be loaded into the index register 2. If the programmer needs a byte pointer to a text string labeled "TX" for example, it is a simple matter:

```
        LDA     1,=2*TX
        .
        .
        .
TX:     .TXT    "TEXT STRING"
```

The use of a label as a literal can make subroutine communication without concern for addressing errors simple. If a call to "XOR" is to be made and "XOR" may not be directly addressable, the following creates a directly addressable reference

```
        JSR     @=XOR
```

A literal can be used as a temporary. For example

```
        STA     1,=0
```

would allocate a page zero word containing 0. Obviously, since 0 can always be generated by a SUB instruction, the programmer does not intend to define a constant of 0. He has merely generated a temporary address, using =0 by convention.

CHAPTER 6

OPERATING PROCEDURES

LOADING THE MACRO ASSEMBLER

The Macro Assembler is supplied as three files of dumped tapes of the RDOS system. The files are:

MAC.SV	-	Macro Assembler
MACXR.SV	-	Cross Reference Listing
MAC.PS	-	Permanent Symbols

The files must be LOADed by the user before the CLI command, MAC, can be used to assemble programs.

MAC COMMAND LINE

The command to assemble files using the Macro Assembler has the format:

MAC△filename$_1$†△ ... filename$_n$†

The MAC command line is used to build a command file as described in the the RDOS User's Manual, 093-000075.

A MAC command causes one or more source files (filename$_i$) to be assembled. Output may be a relocatable binary file, a listing file, or both. The command name MAC must be used in assembling programs; the name MAC cannot be changed.

Command line switches, conventions, and examples are given below:

Switches:

Global: By default, output of an assembly is a relocatable binary file without a listing file.

- /A - add semi-permanent symbols to cross reference.
- /L - listing file is produced. Listings include a cross referencing of the symbol table. MACXR.SV must be available on disk.
- /N - no relocatable binary file is produced.

MAC COMMAND LINE (Continued)

Global Switches: (Continued)

/U	-	user symbols are included in the relocatable binary output.
/E	-	error printouts on the TTO are suppressed unless there is no listing file.
/S	-	skip pass 2 and save a version of the assembler's symbol table, MAC.PS, that contains new symbols and macro definitions. (See page 6-3.)
/O	-	override all listing suppression controls.

Local Switches:

/E	-	error output is directed to the given file name.
/B	-	relocatable binary output directed to the given file name.
/L	-	listing output directed to the given file name.
/S	-	skip this file on pass 2 of assembly (This switch should be used only if the file does not assemble any storage words. Macro definition files can be skipped on pass 2.)

Asterisk: Not permitted.

Errors:

NO SOURCE FILE SPECIFIED.	
ILLEGAL FILE NAME.	
FILE DOES NOT EXIST.	(input file)
FILE ALREADY EXISTS.	(output file)
FILE WRITE PROTECTED.	(output file)
FILE READ PROTECTED.	(input file)
SWITCH ERRORS.	(listing and binary files cannot be the same)

Extensions: On input, search for filename.SR . If not found, and the filename did not have an extension, search for filename.

On default output, produce filename.RB for relocatable binary and filename.LS for listing (global L switch), where filename will be the name portion of the first source file specified without a /S local switch.

MAC COMMAND LINE (Continued)

Examples: MAC Z

causes assembly of source file Z, producing a relocatable binary file called Z.RB.

MAC LIB/S A B C $LPT/L

causes assembly of relocatable binary files A, B, and C. File LIB contains macro definitions and thus is skipped during the second pass. A listing and cross-referenced symbol table are produced on the line printer.

The standard semi-permanent symbols (macro, floating point, and RDOS) are defined in three source files, NBID.SR, FPID.SR, and OSID.SR. To incorporate these symbols into the macro assembler, use the following command:

MAC/S/N NBID FPID OSID)

THE MACRO ASSEMBLER'S SYMBOL TABLE FILES

The macro assembler maintains its symbol table and macro definition table in a disk file called MAC.ST. At the start of each assembly, the "permanent" symbol table file, MAC.PS, is copied to create MAC.ST. Thus, MAC.PS can be used to save symbol and macro definitions from one assembly to the next.

When the pseudo-op .XPNG is seen, MAC.ST is deleted and a new, empty MAC.ST is created. The global /S switch causes the assembler to terminate at the end of pass 1 and to rename MAC.ST to MAC.PS.

Therefore, the programmer can create an entirely new set of retained symbols and macros by performing MAC/S on a source file beginning with an .XPNG; or he can add to his retained symbols and macros by performing MAC/S on a source file not containing an .XPNG.

The symbol table portion of MAC.ST can hold approximately 8000 symbols while the macro definition table portion can hold about 1/2 million bytes of macro definition strings.

APPENDIX A

ERROR MESSAGES

Assembler error messages are single letter codes that are output in the first three character positions of a listing line. The first error code will appear in character position three of the line in which the error occurred. If there is a second error, the code is output in character position two, and if there is a third error, the code appears as the first character of the listing line.

Assembler errors are output as part of the assembly listing to any device and also to the teletypewriter. If the listing is suppressed, the error listing is always output at the teletypewriter. If there is a listing device, output of errors at the teletypewriter can be suppressed. Certain errors encountered on the first pass will be output, since they may not be detected on the second pass.

The list of possible assembler error codes is as follows.

A	Address error.
B	Bad character.
C	Macro error.
D	Radix error.
E	Equivalence error.
F	Format error.
G	Global error.
I	Parity error on input.
K	Conditional or repetitive assembly error
L	Location counter error.
M	Multiply defined symbol error.
N	Number error.
O	Overflow field or stack error.
P	Phase error.
Q	Questionable line.
R	Relocation error.
U	Undefined symbol error.
X	Text error.

Some typical causes of error codes are given on pages following. However, it is not possible to pinpoint all possible causes of assembly errors.

ADDRESSING ERROR (A)

An A flags an error appearing in a memory reference instruction (MRI) and indicates an illegal address. For example:

1. A page zero relocatable instruction references an NREL address.

   Example:

```
                    .NREL
 00003'000010 G:  10      ;NREL ADDRESS
                    .ZREL
A00000-044000 STA 1,G  ;ZREL ADDRESS
```

2. A normal relocatable address references an address outside the range of the program location counter's relative address range: (. -200 $\leq$ displacement $\leq$.+177)

   Example:

```
                    .NREL
A00004'020000'LDA 0,Y          ;Y IS OUTSIDE THE
       000423'.LOC .+416       ;INSTRUCTION'S RANGE
 00423'000002 Y:  2
```

BAD CHARACTER (B)

Error code B indicates an illegal character in some symbol. The statement containing a symbol that has an erroneous character will be flagged with a B. A bad character error often leads to other errors as shown in the example below:

Example:

```
                    .NREL
B00000'024023 .A%: LDA 1,23   ;% IN LABEL SYMBOL CAUSES
                              ;BAD CHARACTER ERROR
```

MACRO ERROR (C)

The macro error code C occurs under the following circumstances:

1. At attempt is made to continue the definition of a macro when it is not the last macro defined.

   Example:

```
        .MACRO△A)
        <macro_definition>%
          .
          .
          .
        .MACRO△A)               ;LEGAL CONTINUATION
        <macro_definition> %
          .
          .
          .
        .MACRO△B)
        <macro_definition>%
          .
          .
          .
C       .MACRO△A)               ;ILLEGAL TO CONTINUE ANY
                                ;MACRO EXCEPT B
```

2. A macro error will occur if a macro exhausts assembler working space. However, this should only occur if the macro definition causes endless recursion.

RADIX ERROR (D)

Error code D occurs on a .RDX or .RDXO pseudo-op when .RDX contains an expression that is not in the range 2-20 and when .RDXO contains an expression that is not in the range 8-20, or when a digit is used that is not within the current input radix.

Examples:

```
D       000030  .RDX  4*6
                .END


        000002  .RDX 2
D00000  000013 B: 35
```

EQUIVALENCE ERROR (E)

Error code E occurs when an equivalence statement contains an undefined symbol on the righthand side of the equals sign. This may occur on pass one before the symbol on the righthand side has been defined or on pass two if the symbol is never defined.

Examples:

```
EUU             A=B       ;PASS ONE; B UNDEFINED

01              .NREL
 EU     000000  A=B       ;PASS TWO; B UNDEFINED
                .END
```

FORMAT ERROR (F)

An F error results from any attempt to use a statement format that is not legal for the type of statement and often often occurs in conjunction with other errors.

When a format error occurs in an instruction, the code generated by the instruction reflects only those fields assembled before the error was detected.

Examples:

```
F       143000 ADD 2              ;NOT ENOUGH OPERANDS

F00425'041410 STA 0,10,3,SNC ;TOO MANY OPERANDS AND WRONG
                                  ;OPERAND FOR INSTRUCTION TYPE

F       000000-.ZREL -1           ;ZREL CANNOT HAVE ARGUMENT

        060512 .DUSR C = DIAS 0,PTR
F00426'060512 C 1                 ;ATTEMPT TO GIVE ARGUMENT TO A
                                  ;SYMBOL DEFINED IN A .DUSR
                                  ;PSEUDO-OP
```

EXTERNAL/INTERNAL SYMBOL ERROR (G)

A G error code results when there is an error in the declaration of an external or entry symbol.

Examples:

```
 GU             .ENT HH          ;HH NEVER DEFINED
02              .END

  G             AA:              ;AA IS ENTRY IN PROGRAM IN
                .EXTN AA         ;WHICH THE SYMBOL IS DECLARED
                .END             ;EXTERNAL
```

INPUT (PARITY) ERROR (I)

An I error code occurs when an input character does not have even parity. The assembler will substitute a back slash (\) for any incorrect character and flag the line containing the error with an I.

CONDITIONAL ASSEMBLY ERROR (K)

A K error code occurs when an .ENDC pseudo-op does not have a preceding .DO or .IFx pseudo-op.

Example:

```
          000002 .DO 2
                   :
                   :
                   :

                 .ENDC
K                .ENDC
```

LOCATION ERROR (L)

The L code occurs when an error is detected in a statement that affects the location counter.

Examples:

1. The expression in a .LOC evaluates to less than zero or cannot be evaluated on the first pass of the assembler. If the expression is outside the range of locations or cannot be evaluated, the .LOC is ignored, and the location counter is unchanged.

```
L       177777 .LOC -1
```

2. The expression in a .BLK statement cannot be evaluated on the first pass of the assembler or its value, when added to the current value of the program location counter, is less than zero. If an L error occurs, the .BLK statement is ignored and the location counter is unchanged.

```
  77711'000000 A:   0
L       100012'.BLK .+100
```

MULTIPLE DEFINITION ERROR (M)

The M code flags a multiply defined symbol. Within an assembly program a symbol appearing, for example, as a label cannot be redefined as another unique label. Any multiply defined symbol will be flagged M at each appearance of the symbol.

Example:

```
                    .NREL
 M000000'000000 A:  0
PM000001'000001 A:  1
```

Note that the second definition of A is also flagged as a phase error (P) on the second assembler pass. (See Phase Error).

NUMBER ERROR (N)

The N code is given when a number exceeds the proper storage limitations for the type of number; the N error occurs under the following conditions:

1. An integer is greater than or equal to 2^{16}. The number is evaluated modulo 2^{16}.

```
        000012  .RDX  10
N000014 000003  65539
```

2. A double precision integer is greater than or equal to 2^{32}. The number is evaluated modulo 2^{32}.

```
N000005 167153 40000000000.D
        024000
```

3. A floating point number is larger than $7.2*10^{75}$.

```
N000013 077777 7.3E75
        177777
```

FIELD OVERFLOW ERROR (O)

A field overflow error occurs when variable stack space is exceeded, when a .TOP or .POP is given with no previous .PUSH, or when a instruction operand is not within the required limits, e.g., 0-3 for an accumulator, 0-7 for a skip field, etc. When overflow occurs in an instruction field, such as an accumulator field, the field will remain unchanged.

Examples:

```
000000 020775 LDA 5,.-3
       014000 .DIAC R=14000
000001 014000 R 1

FO     000000 .POP

FO     000000 .TOP
```

PHASE ERROR (P)

A phase error is caused when the assembler detects on pass 2 some unexpected difference from the source program scan on pass 1. For example, a symbol defined on the first pass which has a different value on the second pass will cause a phase error. If, as in the example, a symbol is multiply defined, the M error flags each statement containing the symbol while the phase error will flag the second (and any subsequent attempt to redefine the symbol.)

Example:

```
                    .NREL

  M00001'000000 B: 0
 PM00002'000000 B: 0
```

QUESTIONABLE LINE (Q)

A Q error occurs when a # or @ atom has been used improperly or when a ZREL value is used where an absolute value is expected.

Examples:

```
000002 113000 ADD 0,@2     ;INCORRECT USE OF SPECIAL ATOM
                           ;CAUSES @ ERROR

                .ZREL
 00000-000010 FLD: .BLK 10

                .NREL
Q00000'000000 LDA 0,FLD,2  ;ASSEMBLER EXPECTS ABSOLUTE
                           ;FOR FLD
                .END
```

RELOCATION ERROR (R)

The R error indicates that an expression cannot be evaluated to a legal relocation type (absolute, relocatable, or byte relocatable as described in Chapter 3) or that the expression is a mix of ZREL and NREL symbols.

Example:

```
                .NREL
 00000'000010 E: 10        ;CONTENTS ABSOLUTE
 00001'000000"E+E          ;CONTENTS NREL BYTE
R00002'000000'E+E+E        ;CONTENTS NOT ABSOLUTE, RELOCATABLE
                           ;OR BYTE RELOCATABLE
```

UNDEFINED SYMBOL ERROR (U)

The U error occurs on pass 2 when the assembler encounters a symbol whose value was never known on pass one. The error occurs on pass one when the definition of a symbol (by equivalence) depends upon another symbol whose value is unknown at that point.

Example:

```
U00022'030000 LDA 2,B'          ;WHERE B IS UNDEFINED
```

See also the example given for equivalence error E.

TEXT ERROR (X)

An error occurring in a string is flagged as a text error (X). A text error occurs if the expression delimiters < and > within a string do not enclose a recognizable arithmetic or logical expression. (Relational expressions cannot be used within text strings.)

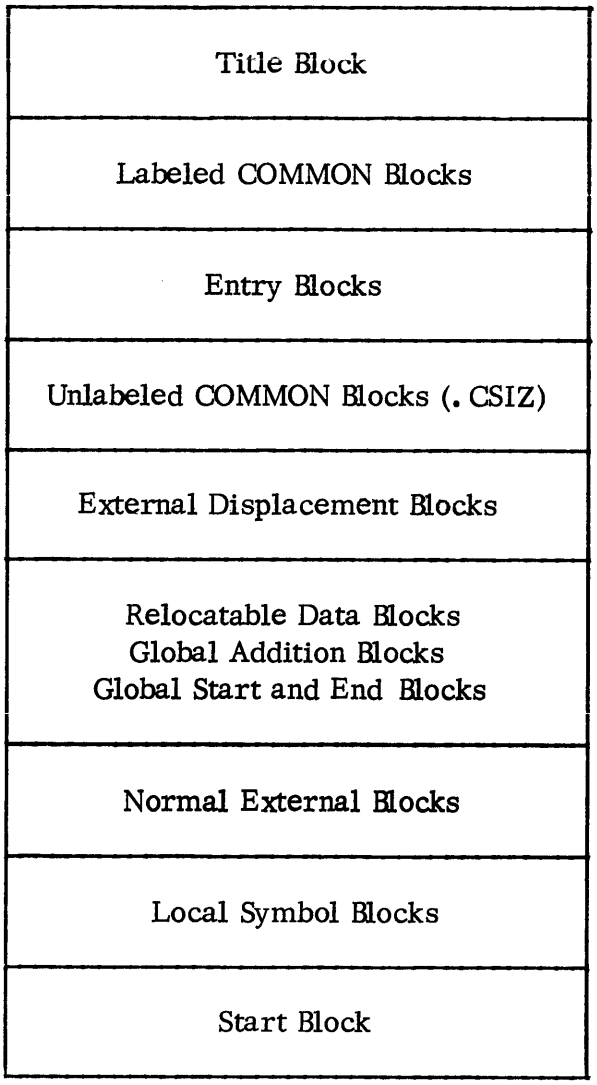
Examples:

```
XU00000 047516 .TXT #NO SPACE ALLOWED IN AN EXPRESSION <  X+ Y>#
        .
        .
        .
X00023 054105 .TXT #EXPRESSIONS MUST HAVE OPERANDS <+>#
        .
        .
        .
XU00043 052101 .TXT #ATTEMPT TO USE RELATIONAL OPERATOR <X<=Y>#
```

APPENDIX B

RELOCATABLE BINARY BLOCK TYPES

The relocatable binary output of the Macro Assembler is divided into a series of blocks. The order in which blocks appear, if each type of block is present, is shown in the figure following.



The relocatable binary output must contain at least a Title Block and a Start Block. Presence of one or more of the other types of blocks will depend upon source input. The pages following describe the content of each of the blocks.

The first word of each block contains a number indicating the type of block. The number is in the range 2 - 17_8. Block type formats are described later in the appendix in numerical order.

The second word of each block is the word count. It is always in two's complement format, and the counter never exceeds 15. Where the word count is a constant for every block of the particular type, the word count constant is shown in parentheses in the format.

Words 3-5 are reserved for relocation flags. The relocation property of each address, datum, or symbol is defined in three bits. For example, for a Relocatable Data Block, bits 0-2 of word 3 apply to the address, bits 3-5 apply to the first data word, bits 6-8 apply to the second data word, etc. The meaning of the bit settings is given in the table following.

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

All other blocks use bits of word 3 only and set words 4 and 5 to zero.

Word 6 contains a checksum, such that the sum of all words in the block is 0.

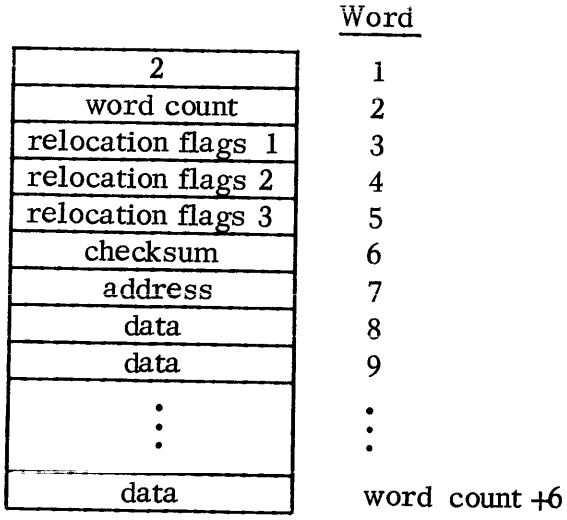
For those blocks containing user symbols, each symbol entry is three words in length.

The first 37 bits of the three-word entry contain the user symbol name in radix 50 form. (See Appendix C for radix 50 notation.) The last five bits of the second word are used as a symbol type flag, where the currently defined types are:

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

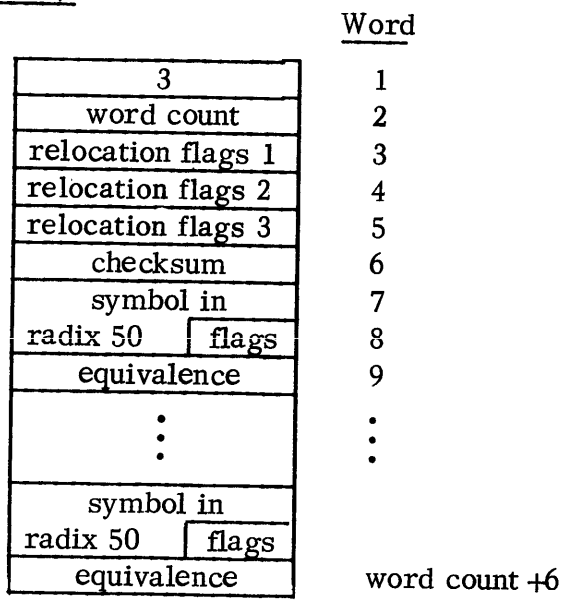
The setting of the third word allocated for each user symbol entry varies with the type of block and is described in the format writeups of each block.

RELOCATABLE DATA BLOCK



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

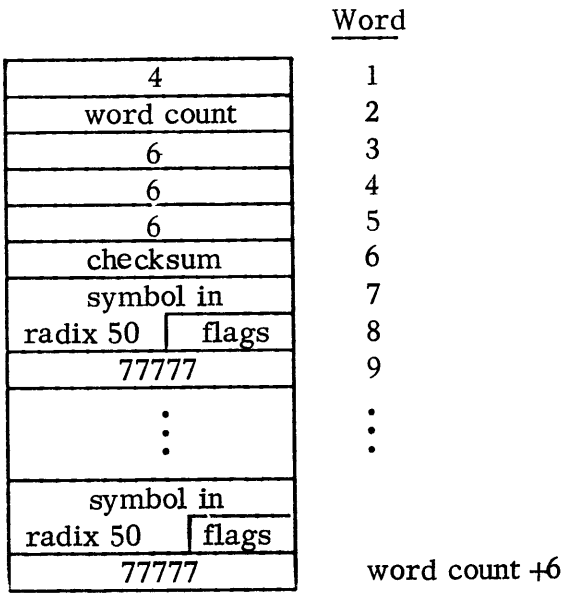
ENTRY BLOCK (.ENT)



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.

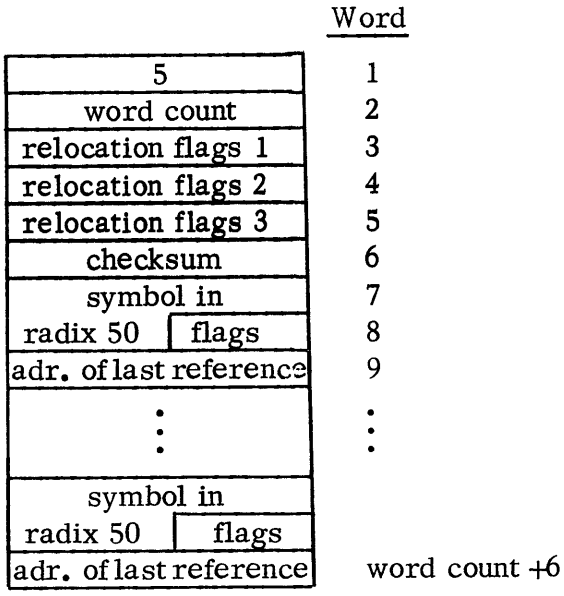
The overlay block .ENTO is the same as the .ENT block except for different flags value in word S1 (word 8, etc.).

EXTERNAL DISPLACEMENT BLOCK (.EXTD)



The third word of each user symbol entry in the External Displacement Block is set to 77777.

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

	Word
6	1
word count (-2)	2
relocation flags 1	3
0	4
0	5
checksum	6
address	7
0	8

TITLE BLOCK (.TITL)

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

13
word count (-4)
relocation flags 1
0
0
checksum
symbol in radix 50 | flags
0
expression value

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)

14
word count (-5)
relocation flags 1
0
0
checksum
address
symbol in radix 50 | 00000
0
expression value

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 expression (<exp> following .CSIZ). 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	checksum
symbol in radix 50 | 00000	7	0
0	8	
	9	

APPENDIX C

RADIX 50 REPRESENTATION

Radix 50 representation is used to condense symbols of five characters into two words of storage using only 27 bits. Each symbol consists of from 1 to 5 characters and a symbol having five characters may be represented as:

$$a_4 a_3 a_2 a_1 a_0$$

where: Each a_i may be one of the following characters: A-Z (26 characters)
0 - 9 (10 characters)
. or ? (2 characters)

All symbols are padded, if necessary, with nulls. Each character can be translated into octal representation as follows:

Character a_i	Translation b_i
null	0
0 to 9	1 to 12_8
A to Z	13_8 to 44_8
.	45_8
?	46_8

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

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

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

$N_2 = (b_1 * 50) + b_0$

$N_{2\ \text{maximum}} = (50)^2 - 1 = 3077$, 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.

APPENDIX D

BASIC SYNTAX SUMMARY

The basic syntax of the macro assembler is defined here in Backus Normal Form notation. In following the language definition, note that the last term of each list of alternative terms is defined in full, then the next to last, etc. This summary covers Chapter 2 and the permanent macro assembler symbols. All DGC-defined semi-permanent symbols are given in Appendix E.

<input_mode > ::= <string_mode > | <normal_mode >

<normal_mode> ::= <atom> ... <atom>

<atom> ::= <symbol > | <number> | <terminal > | <special_atom>

<special_atom> ::= @| #| **

<terminal> ::= <break> | <operator>

<operator> ::= <relational_op> | <logical_op> | <arithmetic_op>

<arithmetic_op> ::= + | - | * | / | B

<logical_op ::= & | !

<relational_op> ::= > | >= | ==| < | <= | <>

<break> ::= (|) | [|] | ; | = | ↓ |) |<c_sp> | <sp>

<sp> ::= , | | tabulation

<c_sp> ::= <sp>... <sp>

<number> ::= <integer> | <floating_point>

<floating_point> ::= <sign> <decimal_digit>...<decimal_digit> . <decimal_digit>...<decimal_digit> E <sign> <decimal_digit> <decimal_digit>

<decimal_digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<sign> ::= + | -

<integer> ::= <single_precision_integer> | <double_precision_integer>

<double_precision_integer> ::= <sign> <decimal_digit> <digit>...<digit> . D

<digit> ::= <decimal_digit> | A | B | C | D | E | F | G | H | I | J

<single_precision_integer> ::= "<character> | ' <string>' | <sign> <decimal_digit> <digit>...<digit>. | '<string>

<string> ::= <character>... <character>

<character> ::= <letter> | <decimal_digit> | <operator> | <break> | <special_atom> | " | ' | \ | ↑ | ← | ?

<letter> ::= A | B | C ... | Z | a | b ... | z

<symbol> ::= <initial_symbol_character > <symbol_character > ... <symbol_character >

<symbol_character > ;:= <initial_symbol_character > |<decimal_digit > | $

<initial_symbol_character> ::= <letter > | . | ?

<string_mode > ::= <text_string> | <macro_definition_string> | <comment_string>

<comment_string> ::= ; <string> <comment_terminator>

<comment_terminator> ::=) | ↓

<macro_definition_string> ::= .MACRO <c_sp> <string > %

<text_string> ::= <text_initiator_op> <c_sp> <character_not_in_string> <string> <character_not_in_string>

<text_initiator_op> ::= .TXT | .TXTE | .TXTF | .TXTO

<permanent_symbol> ::= <pseudo_op> | <value>

<value> ::= .ARGCT | .PASS | . | .POP | .TOP | .MCALL

<pseudo_op>::= .TITL | .MACRO | .PUSH | <radix_op> | <symbol_table_op> | <location_counter_op> | <text_op>
| <interprogram_communication_op> | <file_terminator_op> | <listing_op> | <conditional_or_repeat_op>

<conditional_or_repeat_op> ::= .DO | .IFE | .IFG | .IFL | .IFN | .ENDC

<listing_op> ::= .NOCON | .NOMAC | .NOLOC

<file_terminator_op> ::= .END | .EOT

<interprogram_communication_op> ::= .COMM | .CSIZ | .ENT | .ENTO | .EXTD | .EXTN | .EXTU | .GADD | .GLOC

<text_op> ::= .TXT | .TXTE | .TXTF | .TXTM | .TXTN | .TXTO

<location_counter_op> ::= .BLK | .LOC | .NREL | .ZREL

<symbol_table_op> ::= .DALC | .DIAC | .DIO | .DIOA | .DMR | .DMRA | .DUSR | .XPNG

<radix_op> ::= .RDX | .RDXO

```
;*****************************************************************
;
; NAME: NBID.SR                              PART NUMBER: 090-001482
;
;
; DESCRIPTION: NOVA BASIC INSTRUCTION DEFINITIONS
;
;
; DOCUMENTATION REFERENCES:
;
;       TITLE                                DOCUMENT NO.
;
;       EXTENDED ASSEMBLER                   007-000164
;       RDOS EXTENDED ASSEMBLER              007-000285
;       SOS EXTENDED ASSEMBLER               007-000322
;
;
; REVISION HISTORY:
;
;       REV.            DATE
;
;       00              04/27/73
;
;
; COPYRIGHT (C) DATA GENERAL CORPORATION, 1973
; ALL RIGHTS RESERVED.
;*****************************************************************
```

```
;INSTRUCTION DEFINITON FILE

        .XPNG           ;DELETE ALL SYMBOLS

;DEFINE STANDARD IO DEVICES


.DUSR MDV =01   ;MULTIPLY-DIVIDE
.DUSR MAP=2     ;1200/800 MAP BOX
.DUSR MAP0=02   ;MEMORY ALLOCATION AND PROTECTION
.DUSR MAP1=03   ;  "
.DUSR MAP2=04   ;  "
.DUSR MCAT=06   ;MULTI-PROCESSOR COMMUNICATIONS ADAPTER TRANSMITTER
.DUSR MCAR=07   ;MULTI-PROCESSOR COMMUNICATIONS ADAPTER RECEIVER
.DUSR TTI =10   ;TELETYPE READER/KEYBOARD
.DUSR TTO =11   ;TELETYPE PUNCH/PRINTER
.DUSR PTR =12   ;PAPER TAPE READER
.DUSR PTP =13   ;PAPER TAPE PUNCH
.DUSR RTC =14   ;REAL TIME CLOCK
.DUSR PLT =15   ;INCREMENTAL PLOTTER
.DUSR CDR =16   ;CARD READER
.DUSR LPT =17   ;LINE PRINTER
.DUSR DSK =20   ;FIRST FIXED HEAD DISK CONTROLLER
.DUSR ADCV=21   ;A/D CONVERTER
.DUSR MTA =22   ;FIRST MAG TAPE CONTROLLER
.DUSR DACV=23   ;D/A CONVERTER
.DUSR DCM =24   ;DATA COMMUNICATIONS MULTIPLEXOR
.DUSR QTY =30   ;QUAD MULTIPLEXOR
.DUSR IBM1=31   ;IBM 360/370 INTERFACE
.DUSR IBM2=32
.DUSR DKP =33   ;FIRST MOVING HEAD DISK CONTROLLER
.DUSR CAS =34   ;FIRST CASSETTE CONTROLLER
.DUSR IVT =35   ;INTERVAL TIMER
.DUSR IPB =36   ;INTER-PROCESSOR BUS
.DUSR DPI =40   ;DUAL PROCESSOR INPUT
.DUSR DPO =41   ;DUAL PROCESSOR OUTPUT
.DUSR TTI1=50   ;SECOND TTY
.DUSR TTO1=51   ;
.DUSR PTR1=52   ;SECOND PAPER TAPE READER
.DUSR PTP1=53   ;SECOND PAPER TAPE PUNCH
.DUSR PLT1=55   ;SECOND PLOTTER
.DUSR CDR1=56   ;SECOND CARD READER
.DUSR LPT1=57   ;SECOND LINE PRINTER
.DUSR DSK1=60   ;SECOND FIXED HEAD DISK CONTROLLER
.DUSR MTA1=62   ;SECOND MAG TAPE CONTROLLER
.DUSR DKP1=73   ;SECOND MOVING HEAD DISK CONTROLLER
.DUSR CAS1=74   ;SECOND CASSETTE CONTROLLER
.DUSR FPU1=74   ;SINGLE-PRECISION FLOATING POINT
.DUSR FPU2=75   ;DOUBLE-PRECISION FLOATING POINT
.DUSR FPU=76    ;FLOATING-POINT CONTROLLER
.DUSR CPU =77   ;CENTRAL PROCESSING UNIT
```

```
;MULTIPLY/DIVIDE
.DUSR DIV=073101
.DUSR MUL =073301

;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT DON'T REQUIRE AC'S
.DMR JMP = 000000
.DMR JSR=004000
.DMR ISZ=010000
.DMR DSZ=014000

;DEFINE MEMORY REFERENCE INSTRUCTIONS THAT REQUIRE AC'S
.DMRA LDA=020000
.DMRA STA=040000

;DEFINE THE ALC INSTRUCTIONS
.DALC COM=100000
.DALC NEG=100400
.DALC MOV=101000
.DALC INC=101400
.DALC ADC=102000
.DALC SUB=102400
.DALC ADD=103000
.DALC AND=103400

;DEFINE THE ALC SKIPS
.DUSR SKP=1
.DUSR SZC=2
.DUSR SNC=3
.DUSR SZR=4
.DUSR SNR=5
.DUSR SEZ=6
.DUSR SBN=7
```

```
;DEFINE THE IO INSTRUCTIONS
.DIO  NIO=060000
.DIOA DIA=060400
.DIOA DOA=061000
.DIOA DIB=061400
.DIOA DOB=062000
.DIOA DIC=062400
.DIOA DOC=063000

;DEFINE THE IO SKIP INSTRUCTIONS
.DIO SKPBN=063400
.DIO SKPBZ=063500
.DIO SKPDN=063600
.DIO SKPDZ=063700


;DEFINE SPECIAL INSTRUCTIONS
.DUSR INTEN=NIOS CPU              ;INTERRUPT ENABLE
.DUSR INTDS=NIOC CPU              ;INTERRUPT DISABLE
.DIAC READS=DIA 0,CPU             ;READ THE SWITCHES
.DIAC INTA= DIB 0,CPU             ;INTERRUPT ACKNOWLEDGE
.DIAC MSKO= DOB 0,CPU             ;MASK OUT
.DUSR IORST=DICC 0,CPU            ;IO RESET
.DUSR HALT= DOC 0,CPU             ;HALT

        .EOT
```

```
;*****************************************************************
;
; NAME: FPID.SR                                 PART NUMBER: 090-001483
;
;
; DESCRIPTION: FLOATING POINT INTERPRETER INSTRUCTION DEFINITIONS
;
;
; DOCUMENTATION REFERENCES:
;
;       TITLE                                   DOCUMENT NO.
;
;       EXTENDED ASSEMBLER                      007-000164
;       RDOS EXTENDED ASSEMBLER                 007-000285
;       3OS EXTENDED ASSEMBLER                  007-000322
;
;
; REVISION HISTORY:
;
;       REV.            DATE
;
;       00              04/27/73
;
;
; COPYRIGHT (C) DATA GENERAL CORPORATION, 1973
; ALL RIGHTS RESERVED.
;*****************************************************************
```

```
; ABSOLUTE FLOATING INTERPRETER INSTRUCTIONAL ENTRIES

        .DUSR   FETR=   JSR @4  ; ENTER
        .DUSR   FINI=   JSR @5  ; INITIALIZE

;
; FLOATING ALC INSTRUCTIONS WITH NORMALIZE/NO NORMALIZE OPTION
;
        .DALC   FNEG= 100400
        .DALC   FNEGU=100420
        .DALC   FMOV= 101000
        .DALC   FMOVU=101020
        .DALC   FPOS= 101400
        .DALC   FABS=FPOS 0,0
        .DALC   FDABS=FPOS 0,0
        .DALC   FPOSU=101420
        .DALC   FMNS= 102000
        .DALC   FMNSU=102020
        .DALC   FSUB= 102400
        .DALC   FSUBU=102420
        .DALC   FADD= 103000
        .DALC   FADDU=103020
        .DALC   FRND= 103400
        .DALC   FRNDU=103420
;
; FLOATING ALC INSTRUCTIONS WITHOUT NORMALIZE/NO NORMALIZE OPTION
;
        .DALC   FMPY=100100
        .DALC   FDIV=100200
        .DALC   FHLV=100300
;  ONLY AVAILABLE WITH EXTENDED FLOATING POINT PACKAGE
        .DALC   FALG=100020
        .DALC   FALOG=FALG 0,0
        .DALC   FDLOG=FALG 0,0
        .DALC   FATN=100040
        .DALC   FATAN=FATN 0,0
        .DALC   FDATAN=FATN 0,0
        .DALC   FCOS=100060
        .DALC   FDCOS=FCOS 0,0
        .DALC   FSIN=100120
        .DALC   FDSIN=FSIN 0,0
        .DALC   FTAN=100140
        .DALC   FDTAN=FTAN 0,0
        .DALC   FEXP=100220
        .DALC   FDEXP=FEXP 0,0
        .DALC   FSGN=100260 ; (RELO. INTER. ONLY)
        .DALC   FHLV=100300
        .DALC   FSQR=100240
        .DALC   FSQRT=FSQR 0,0
        .DALC   FDSQR=FSQR 0,0
        .DALC   FLFX=100320 ; (RELO. INTER. ONLY)
        .DALC   FXFL=100340 ; (RELO. INTER. ONLY)

        .DALC   FDVD=   FDIV 0,0
```

```
;
; FLOATING MP INSTRUCTIONS
;
; WITH AN ACCUMULATOR
;
        .DMRA   FLDA= 020000
        .DMRA   FSTA= 040000
;
; WITHOUT AN ACCUMULATOR
;
        .DMR    FJMP= 000000
        .DMR    FJSR= 004000
        .DMR    FISZ= 010000
        .DMR    FDSZ= 014000
        .DMR    FFLO= 060000
        .DMR    FLD3= 064000
        .DMR    FST3= 070000
        .DMR    FFIX= 074000
;
; SPECIAL FUNCTION INSTRUCTIONS
;
        .DUSR   FEXT= 00
        .DUSR   FIC2= 104000
        .DUSR   FIC3= 110000
        .DUSR   FHLT= 114000
;
; INSTRUCTIONS REQUIRING ONE ACCUMULATOR
;
        .DIAC   FFDC= 140000
        .DIAC   FDFC= 120000
        .DIAC   FFDCF=140001
        .DIAC   FDFCI=120001

;
; SKIP DEFINITIONS
;
        .DUSR   FSGT= 1
        .DUSR   FSLT= 2
        .DUSR   FSNR= 3
        .DUSR   FSZR= 4
        .DUSR   FSGE= 5
        .DUSR   FSLE= 6
        .DUSR   FSKP= 7
;
```

```
; DEFINE THE PRECISION INSTRUCTION

        .DIO    FPRC=060000 ; (RELO. INTER. ONLY)

; DEFINE THE FORTRAN INTERPRETER MNEMONICS
; (ONLY AVAILABLE WITH THE RELOCATABLE INTERPRETER)

        .DALC   FCEQ=100170 ; EQUALITY TEST
        .DALC   FCLE=100172 ; LESS THAN OR EQUAL TEST
        .DALC   FCLT=100162 ; LESS THAN TEST

        .DALC   FSTM=100161 ; STA TO    ADDRESS FOLLOWING
        .DALC   FLDM=100163 ; LDA FROM ADDRESS FOLLOWING
        .DALC   FADM=100165 ; ADD FROM ADDRESS FOLLOWING
        .DALC   FSMR=100167 ; R - MEMORY TO R
        .DALC   FSRM=100171 ; MEMORY - R TO R
        .DALC   FMLM=100173 ; MULTIPLY BY ADDRESS FOLLOWING
        .DALC   FDMR=100175 ; MEMORY/R TO R
        .DALC   FDRM=100177 ; R/MEMORY TO R

        .DUSR   FSNGL=FPRC 2    ; SINGLE
        .DUSR   FDBLE=FPRC 4    ; DOUBLE

        .EOT
```

```
;*****************************************************************
;
; NAME: OSID.SR                                 PART NUMBER: 090-001484
;
;
; DESCRIPTION: OPERATING SYSTEMS INSTRUCTION DEFINITIONS
;
;
; DOCUMENTATION REFERENCES:
;
;       TITLE                                   DOCUMENT NO.
;
;       EXTENDED ASSEMBLER                      007-000164
;       RDOS EXTENDED ASSEMBLER                 007-000285
;       SOS EXTENDED ASSEMBLER                  007-000322
;
;
; REVISION HISTORY:
;
;       REV.            DATE
;
;       00              04/27/73
;
;
; COPYRIGHT (C) DATA GENERAL CORPORATION, 1973
; ALL RIGHTS RESERVED.
;*****************************************************************
```

```
; DEFINE AS PERMANENT SYMBOLS ALL MONITOR RELATED SYMBOLS

; DEFINE THE NOVA SYSTEM CALL

        .DUSR   .SYSTM= JSR     @17

; DEFINE THE USER STACK POINTER LOCATION

        .DUSR   USP=    16


; DEFINE THE MONITOR CALLS


; COMMANDS WHICH DO NOT REQUIRE DEVICE ACTION OR CHANNEL NUMBER

        .DUSR   .CREAT= 0B7             ; CREATE FILE
        .DUSR   .DELET= 1B7             ; DELETE FILE
        .DUSR   .RENAM= 2B7             ; RENAME A FILE
        .DUSR   .MEM=   3B7             ; RETURN MEMORY LIMITS
        .DUSR   .BREAK= 4B7             ; BREAK
        .DUSR   .RLSE=  5B7             ; RELEASE A DEVICE
        .DUSR   .DIR=   6B7             ; CHANGE BASE DIRECTORY
        .DUSR   .EXEC=  7B7             ; EXECUTE A PROGRAM OVERLAY
        .DUSR   .INIT=  10B7            ; INIT DISK DEVICE
        .DUSR   .RTN=   11B7            ; SYSTEM RETURN
        .DUSR   .RESET= 12B7            ; I/O RESET
        .DUSR   .ERTN=  15B7            ; ERROR RETURN FROM COMMAND
        .DUSR   .CRAND= 16B7            ; CREATE RANDOM
        .DUSR   .GCHAR= 17B7            ; GET TTY CHAR
        .DUSR   .PCHAR= 20B7            ; TTY PUT CHAR
        .DUSR   .DELAY= 21B7            ; WAIT N CYCLES
        .DUSR   .MEMI=  22B7            ; ALLOCATE MEMORY INCREMENT
        .DUSR   .CCON=  41B7            ; CREATE CONTIGUOUS
        .DUSR   .EXFG=  43B7            ; EXECUTE FOREGROUND
        ;RESERVE        44B7            ;IOCS
        ;RESERVE        45B7            ;IOCS
```

```
;       COMMANDS WHICH REQUIRE CHANNEL NUMBER

        .DIO    .ROPEN= 2387    ; OPEN FOR READING
        .DIO    .MTOPO= 2387    ; OPEN MAG TAPE FOR DIRECT I/O
        .DIO    .OVOPN= 2487    ; OPEN OVERLAYS
        .DIO    .CHATR= 2687    ; CHANGE THE FILE ATTRIBUTES
        .DIO    .GTATR= 2787    ; GET THE FILE/DEVICE ATTRIBUTES
        .DIO    .RDB=   1387    ; READ BLOCK
        .DIO    .WRB=   1487    ; WRITE BLOCK
        .DIO    .APPEND=2587    ; OPEN FILE FOR APPENDING
        .DIO    .OPEN=  3087    ; OPEN FILE
        .DIO    .CLOSE= 3187    ; CLOSE FILE
        .DIO    .RDS=   3287    ; READ SEQUENTIAL CHARACTERS
        .DIO    .RDL=   3387    ; READ SEQUENTIAL LINE
        .DIO    .RDR=   3487    ; READ RANDOM
        .DIO    .WRS=   3587    ; WRITE SEQUENTIAL CHARACTERS
        .DIO    .WRL=   3687    ; WRITE SEQUENTIAL LINE
        .DIO    .WRR=   3787    ; WRITE RANDOM
        .DIO    .OVLOD= 4087    ; LOAD OVERLAY
        .DIO    .SCALL= 4287    ; GENERAL CALL
        .DIO    .MTDIO= 4687    ; MAG TAPE DIRECT I/O
        .DIO    .SPOS=  4787    ; SET FILE POSITION
        .DIO    .GPOS=  5087    ; GET FILE'S CURRENT POSITION
        .DIO    .EOPEN= 5187    ; OPEN FOR EXCLUSIVE USE
        .DIO    .TOPEN= 5287    ; TRANSPARENT OPEN
        .DIO    .CHLAT= 5387    ; CHANGE LINK ACCESS ATTRIBUTES
        .DIO    .CHSTS= 5487    ; GET CHANNEL STATUS
```

```
; THE FOLLOWING CALLS ARE SCALLS

        .DUSR   .GHRZ=  .SCALL 0        ;GET CLOCK FREQ
        .DUSR   .DUCLK= .SCALL 1        ;DEF USER CLOCK
        .DUSR   .RUCLK= .SCALL 2        ;REMAOVE USER CLOCK
        .DUSR   .GTOD=  .SCALL 3        ;GET TOD
        .DUSR   .STOD=  .SCALL 4        ;SET TOD
        .DUSR   .SDAY=  .SCALL 5        ;SET DAY
        .DUSR   .GDAY=  .SCALL 6        ;GET DAY
        .DUSR   .IDEF=  .SCALL 7        ;DEFINE DEVICE INT
        .DUSR   .IRMV=  .SCALL 10       ;REMOVE DEV INT
        .DUSR   .SPKL=  .SCALL 11       ;SPOOL KILL
        .DUSR   .SPDA=  .SCALL 12       ;SPOOL DISABLE
        .DUSR   .SPEA=  .SCALL 13
        .DUSR   .ENQ=   .SCALL 14       ;STREAM OUTPUT
        .DUSR   .CPART= .SCALL 15       ;CREATE PARTITION
        .DUSR   .CDIR=  .SCALL 16       ;CREATE SUBDIRECTORY
        .DUSR   .LINK=  .SCALL 17       ;LINK ENTRY
        .DUSR   .EDIV=  .SCALL 20       ;CHANGE DIRECTORY SPECIFIER
        .DUSR   .GDIRS= .SCALL 21       ;GET DIRECTORY SPECIFIER
        .DUSR   .SYSI=  .SCALL 22       ;SOS COMPATIBLE CALL
        .DUSR   .WCHAR= .SCALL 23       ;WAIT FOR TTY CHAR
        .DUSR   .ICMN=  .SCALL 24       ;INIT COMMON
        .DUSR   .WRCMN= .SCALL 25       ;WRITE TO COMMON
        .DUSR   .RDCMN= .SCALL 26
        .DUSR   .DDIS=  .SCALL 27       ;DISABLE INT (CONTL A,C,F)
        .DUSR   .OEBL=  .SCALL 30       ;ENABLE INT
        .DUSR   .DEBL=  .SCALL 31       ;ENABLE MAPPED DEV ACCESS
        .DUSR   .DDIS=  .SCALL 32       ;DISABLE MAPPED DEV ACCESS
        .DUSR   .RDOPR= .SCALL 33       ;READ OPERATOR
        .DUSR   .WROPR= .SCALL 34       ;WRITE OPERATOR
        .DUSR   .STMAP= .SCALL 35       ;DCH MAP REQ FOR USER
        .DUSR   .GCIN=  .SCALL 36       ;GET CONSOLE INPUT DEV
        .DUSR   .GCOUT= .SCALL 37       ;GET CONSOLE OUTPUT DEV
        .DUSR   .STAT=  .SCALL 40       ;GET STATUS OF FILE
        .DUSR   .ECLR=  .SCALL 41       ;RELEASE A FILE
        .DUSR   .TCRET= .SCALL 42       ;TRANSPARENT .CREATE
        .DUSR   .TCRND= .SCALL 43       ;TRANSPARENT .CRAND
        .DUSR   .TCCON= .SCALL 44       ;TRANSPARENT .CCON
        .DUSR   .FGND=  .SCALL 45       ;IS THERE A FOREGROUND
        .DUSR   .GMEM=  .SCALL 46       ;GET MEM PARTITIONS
        .DUSR   .SMEM=  .SCALL 47       ;SET MEM PARTITIONS
        .DUSR   .BOOT=  .SCALL 50       ;INVOKE HIPBOOT
        .DUSR   .MDIR=  .SCALL 51       ;GET MASTER DIR. SPECIFIER
        .DUSR   .GCHN=  .SCALL 52       ;GET A FREE CHANNEL
        .DUSR   .ULNK=  .SCALL 53       ;DELETE A LINK ENTRY


        .END
```

APPENDIX F

PERMANENT SYMBOLS

Permanetnt Symbol	Pseudo-op (directive)	Value
.	No	Yes
.ARGC	No	Yes
.BLK	Yes	No
.COMM	Yes	No
.CSIZ	Yes	No
.DALC	Yes	No
.DIAC	Yes	No
.DIO	Yes	No
.DIOA	Yes	No
.DMR	Yes	No
.DMRA	Yes	No
.DO	Yes	No
.DUSR	Yes	No
.END	Yes	No
.ENDC	Yes	No
.ENT	Yes	No
.ENTO	Yes	No
.EOT	Yes	No
.EXTD	Yes	No
.EXTN	Yes	No
.EXTU	Yes	No
.GADD	Yes	No
.GLOC	Yes	No
.IFE	Yes	No
.IFG	Yes	No
.IFL	Yes	No
.IFN	Yes	No

PERMANENT SYMBOLS (Cont'd)

Permanent Symbol	Pseudo-op (directive)	Value
.LOC	Yes	Yes
.MCALL	No	Yes
.MACRO	Yes	No
.NOCON	Yes	Yes
.NOLOC	Yes	Yes
.NOMAC	Yes	Yes
.NREL	Yes	No
.PASS	No	Yes
.POP	No	Yes
.PUSH	Yes	No
.RDX	Yes	Yes
.RDXO	Yes	Yes
.TITL	Yes	No
.TOP	No	Yes
.TXT	Yes	No
.TXTE	Yes	No
.TXTF	Yes	No
.TXTM	Yes	Yes
.TXTN	Yes	Yes
.TXTO	Yes	No
.XPNG	Yes	No
.ZREL	Yes	No

MACRO ASSEMBLER ERROR CODES

Up to three error codes may be output per source line. The error codes are output in the first three character positions of the listing line. The first error encountered causes a code to be placed in column 3, the second in column 2, and the third in column 1.

Code	Error
A	Addressing error
B	Bad character
C	Macro error
D	Radix error
E	Equivalence error
F	Formatting error
G	Global symbol error
I	Parity error on input
K	Conditional or repetitive assembly error
L	Location counter error
M	Multiply defined symbol error
N	Number error
O	Overflow field or stack error
P	Phase error
Q	Questionable line error
R	Relocation error
U	Undefined symbol error
X	Text input error

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) _____

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

Postage will be paid by:

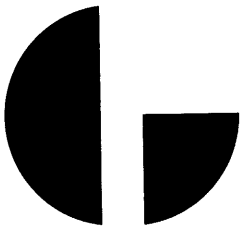
Data General Corporation

Southboro, Massachusetts 01772

ATTENTION: **Programming Documentation**

FOLD UP SECOND FOLD UP

STAPLE



DATA GENERAL
CORPORATION
Southboro,
Massachusetts 01772
(617) 485-9100