SPERRY UNIVAC
1100 Series

Meta–Assembler (MASM)

Programmer Reference

SPERRY UNIVAC
COMPUTER SYSTEMS

UP-8453 Rev. 1

This document contains the latest information available at the time of publication. However, Sperry Univac reserves the right to modify or revise its contents. To ensure that you have the most recent information, contact your local Sperry Univac representative.

Sperry Univac is a division of Sperry Rand Corporation.

This document corresponds to level 2R1 of the MASM software.

PRINTED IN U.S.A.

Page Status Summary

Issue: UP-8453 Rev. 1

Section	Pages	Update
Cover/Disclaimer		
PSS	1	
Contents	1 thru 5	
1	1 thru 80	
2	1 thru 21	
Index	1 thru 5	
User Comment Sheet		
Total: 114 pages and cover		

Section	Pages	Update

Section	Pages	Update

Contents

Page Status Summary

Contents

TABLES

1. Assembler Language

1.1. INTRODUCTION

This manual describes the SPERRY UNIVAC 1100 Series Meta-Assembler (MASM) processor and language. This manual is directed to users with basic Assembler programming knowledge and experience. Definition of the machine language which is to be assembled by MASM is not given in this document. This information is available in the relevant hardware manuals.

MASM is called a meta-assembler because it is not specifically bound to generating code for a particular hardware architecture. With an unaltered environment, MASM will generate code for an 1100 Series hardware architecture. However, with the directives and built-in functions provided, the user may alter the environment to generate code for any hardware architecture. This assumes the output of MASM (1100 Series Relocatable Binary Format) can be converted to an acceptable form for to the operating system on the alternate architecture.

The processor accepts both Fieldata and ASCII input and maintains character constants in either code as specified by the user. MASM uses an internal code to store character constants which do not have to be maintained in a specific character code.

MASM performs specified tasks based on the interpretation of statements received primarily via the Source Input Routine (SIR$) and produces an output. The output produced depends upon the user's request. If a relocatable binary element is requested, it is produced by the Relocatable Output Routine (ROR). MASM optionally produces a printed listing of the the input and its processed form. The structure of both the input and output forms are presented elsewhere in this manual.

MASM performs its function in two scans of the input. The first scan is known as the summary pass, and the second is known as the generative pass. These two passes of the source input, that is, from the first source image encountered to the last source image, are known as the main assembly. Assemblies invoked within the main assembly are known as subassemblies. Certain initialization is done at the start of each pass (see 1.8).

1.1.1. Dictionary

To use MASM effectively, one must have a general knowledge of the storage mechanism known as the dictionary. The most elementary function of the dictionary is to retain knowledge of labels and values associated with the labels, such as the value and number of the location counter at the time the label is defined. At processor initialization the dictionary contains the directives and functions built into MASM.

A name and its value are automatically entered into the dictionary upon detection of a label on an assembler statement. The value associated with the label is determined by its use in the label field and the rest of the assembler statement. For example the "value" associated with built-in directives and functions is not really a value in the normal sense of the word, but rather control information, which in this case acts not as data to be manipulated, but rather as data to govern the next stage of manipulation.

Each value entered into the dictionary has a type associated with it. See Section 1.7.25 for definition of the types available.

The dictionary is structured by levels. These levels define the scope of labels and have the range 0 to *n*, with 0 being the highest level and *n* being the lowest level. Labels defined at level 0 are known outside the program. Labels defined at level 1 are known only to the program. Labels defined at levels lower than level 1 are known to selected portions of the program. All operation mnemonics and built-in directives and functions are also known at level 1. The dynamic nesting of subassemblies causes lower levels to be employed. Labels defined at a particular level stay at that level unless it is specifically requested that they be known at some higher level.

A value is retrieved upon the presentation of a symbol. The normal retrieval is accomplished by starting at the level corresponding to the current subassembly and searching progressively higher levels (i.e., lower numbered levels) until the symbol is found.

1.2. MASM USAGE

1.2.1. Processor Call

MASM is normally a standard processor in SYS$*LIB$ of an 1100 Series system. The 1100 Series Executive System, Volume 3, System Processors Programmer Reference, UP-4144.3 (current version) and 1100 Series Executive System, Volume 2, EXEC Programmer Reference, UP-4144.2 (current version), describe the standard form for calling an 1100 Series language or systems processor. This information is directly applicable to MASM. Unique to each processor, however, is a set of option letters in addition to the Source Input Routine options. The MASM processor call statement has the format:

@MASM,*options si,ro,so*

where *si* denotes the name of the source input element, *ro* the relocatable output element or, in the case of a definition mode assembly, the omnibus output element, and *so* the source output element. The *options* are defined in Table 1-1.

Table 1-1. MASM Options

Option	Description
A	Not used.
B	Use batch format in demand mode (assumed if a @BRKPT PRINT$ is active when MASM is initiated).
C	Print source images and $DISPLAY strings.
D	Double space output.
E	Display walkback information in case of error.
F	Not used.
G	SIR$ option.
H	SIR$ option.
I	SIR$ option.
J	SIR$ option.
K	SIR$ option.
L	Same as both S and R options.
M	Allows directive redefinition by PROCs.
N	Assume an implied $UNLIST precedes line 1 of the source language.
O	Print octal information, including details of the preamble of the output RB element and values produced by $DISPLAY (other than strings).
P	SIR$ option.
Q	SIR$ option.
R	Print detailed relocation information for generated data (implies the O option).
S	Both C and O options.
T	Not used.
U	SIR$ option.
V	Print both input and updated line numbers with the correction lines.
W	SIR$ option.
X	Terminate the run if errors occur during the assembly (meaningful for batch runs only).
Y	Print cross reference listing immediately after the list of entry points.
Z	Print console message if the assembly contains errors.

If no specification fields (*si*, *ro*, *so*) are specified, the assumed source for input is the run stream. NAME$ in TPF$ is the assumed name for the *ro* field. No listing options are assumed to be set.

1.2.2. Reusability

MASM is a reusable processor. This means that if successive calls on MASM are separated only by transparent control statements (such as @MSG, @LOG, @HDG), MASM is not reloaded from mass storage but reads its own control statement, reinitializes itself, and processes the next element. This saves considerable time and I/O resources. Note that this capability is available even if MASM is being called from a user file (rather than from SYS$*LIB$) provided that all calls on MASM after the first in a sequence do not specify the user file (i.e., call is @MASM...) from which MASM was obtained. If a reload of MASM is desired for some reason, the @ENDX control statement may be used to terminate the reusability sequence. Note that when in reuse mode, MASM attempts to minimize the cost of dynamic storage expansion and storage use by starting the next assembly with a storage size slightly smaller than the size at the end of the previous assembly.

1.2.3. Output

MASM produces three types of output that are under the control of the user:

1. a printed listing,
2. a relocatable binary (RB) and
3. an omnibus element for subsequent use by MASM.

The control available over the printed listing allows blank or text lines to be inserted and new pages started for readability. The ease of insertion of commentary text is intended to encourage clear and complete program documentation. The listing may be partly or completely inhibited.

Reading from left to right, the printed output may be divided into five fields. Field 1 starts in the first print position and contains error flags, if any. Field 2 contains the location counter number and value. Field 3 contains the binary (octal or hexadecimal) representation of the value generated. Field 4 contains source line numbers, and Field 5 contains the source image as seen by MASM.

Field 2 is void unless an RB was produced. Field 3 contains the octal or hexadecimal representation of the binary value associated with the interpretation of field 5. Field 4 may have several formats. If two columns of numbers are present, then the left column contains the line numbers of the source output and the right column contains the line numbers of the source input (V option listing). If only one column of numbers is present, then these numbers are the line numbers of the source output. If the element contains only images produced by the Conversational Time Sharing System (CTS), (which implies no corrections were applied through SIR$), the single column of line numbers will be the CTS line numbers.

At the end of the source input and generated output listing, a summary of the preamble of the generated RB element is given. This includes the numbers and values of the location counters used, the locations of the externalized symbols, and the names of the undefined symbols.

Following the preamble, some statistics concerning the processor behavior are given. These include the number of lines encountered including lines from procedure bodies, the assembly time (accumulated CPU usage) in seconds, and storage utilization. Storage utilization is given in the format *a/b/c/d*, where *a* is the starting size in words of storage pool, *b* is the number of storage compactions done, *c* is the number of ER MCORE$'s done and *d* is the final size, in words, of the storage pool.

1.2.4. Input

1.2.4.1. Statements

MASM processes the input presented to it in terms of *statements*, where *statement* is one or more *lines* of input text and a *line* is an 80-character image. A statement can be considered to have two parts: the *functional* part, which will be interpreted by MASM, and a *comment* part, which serves to provide additional information to people reading the program.

The functional part of a statement may be divided into three fields:

1. a label field,
2. an operation field and
3. an operand field. Each field may contain subfields.

All of the fields and subfields following the label field are in free form. The label field must begin in column 1 of the the symbolic line . Any or all of the fields may be void. Fields are generally bounded by one or more spaces and subfields are bounded by commas.

MASM completes the interpretation of the functional part of a line when one of the following four events occur:

1. the maximum number of fields and subfields required by the operation is encountered;

2. the 80th character is read;

3. the line terminator space period space (.) is encountered; or

4. a line continuation character (;) is encountered and MASM is not currently processing a string enclosed in quotes.

Example:

```
1.  P F      F O R M   1 2 , 6 , 1 8     . F O R M   D E F I N I T I O N
2.  . A  C O M M E N T
3.  E N D
```

Explanation :

Line 1 uses all four fields. PF is the label, FORM is the operation field, 12,6,18 are three subfields in the operand field. Characters to the right of the period are comments.

Line 2 contains only the comment field. It could indicate a logical break in the symbolic code or give additional commentary.

Line 3 contains only an operation field. MASM knows this is an operation field because it is the first field not starting in column 1.

1.2.4.2. Symbols

Labels, operations and many operands are generally specified as strings of characters called "symbols", which are subject to certain restrictions. These restrictions eliminate ambiguities and protect the processor.

Legitimate MASM symbols may contain the characters A–Z, 0–9, and $. A symbol may not begin with a digit, and only MASM system symbols may begin with the $ character. Therefore, all user-defined symbols must begin with an alphabetic character. No distinction is made between uppercase and lowercase characters when used in symbol names, if an element is being maintained in ASCII. Thus "ABC" and "abc" are the same symbol. The maximum length of a symbol is 12 characters. Any excess beyond this is ignored.

Example:

1. AB1 – a valid MASM symbol.

2. 1AB – an invalid MASM symbol.

3. A#7 – an invalid MASM symbol.

4. $EQU – an illegal symbol if user tries to insert it into the dictionary.
   – a legitimate symbol if referencing the MASM directive $EQU.

5. A/B – an invalid symbol; a valid expression.

1.2.5. Library Searching

The rules for directive type symbol lookup are as follows:

1. If the M option on the MASM processor call statement is not set, the dictionary is searched.
2. If the file name ASM$PF is attached to a file assigned to the run that file is searched.
3. If there is a source input file, it is searched. If not, the source output file is searched. If none, the relocatable output file is searched.
4. If the M option on the MASM processor call statement is set, the dictionary is searched.
5. SYS$*RLIB$ is searched.

If a find is made at any stage of this search, no further searching is done and the definition or sample is read in from the file where it was found. Note that at step 3, only one of the three files was actually searched. Once the definition or sample has been found and read in from a file, its definition is placed in the dictionary and file searching for the symbol will not take place again provided the definition is not deleted.

1.3. FIELDS

As mentioned earlier, the three fields of the functional part of a MASM statement are (1) label, (2) operation and (3) operand.

1.3.1. Labels

This field is optional and is used to introduce symbols into the dictionary. The label field may be divided into subfields. These subfields may be further divided into items. The entities which may be entered as items are:

1. page control,
2. line levelers,
3. node selectors,
4. location counter specifications,
5. dictionary insertion control and
6. MASM symbols.

The first item may be a "/" to indicate page eject. The next item on the line may be a line leveler of the form "%*n*:" where *n* is an unsigned integer (see 1.11). The leveler may be followed by a location counter specification of the form "$(*e*) where *e* is a expression which evaluates to an integer in the range of 0 to 63. The next item in the subfield may be a legal MASM symbol, which is the label. If both a location counter specification and a symbol appear in the label field, they must be in separate subfields. Dictionary control items ,"*", may follow the MASM symbol. The next item may be a node selection in the form ($e_1, e_2, \ldots$) where *e* indicates some expression that can be converted to an integer. After the node selectors the dictionary control items may also appear. They may not appear in both places. As a special case the label field may consist of one or more asterisks. (See 1.8).

There may be more than one subfield with a legal MASM symbol. The symbols are set to the value of the current location counter. If the statement has a directive which utilizes a label in its interpretation, the last MASM symbol in the label field are associated with the directive.

Examples:

1. /

The label field consists of a single item, the "/" which causes a page eject.

2. /ABC

The label field of line 2 consists of two items, a page eject character and a symbol. The symbol is implicitly defined.

3. %1:$(3),TAG

The label field of line 3 consists of two subfields. The first subfield contains two items: a line leveler and a location counter specification. The second subfield consists of a symbol.

4. EOFADR*

The label field of line 4 consists of two items: a symbol and a dictionary control character.

5. ARG*(1,5)
6. ARG(1,5)*

The label fields of these lines have the same effect. The label field of both lines consists of three items: a symbol, a node selection, and a dictionary control character.

7. TAG,IOG

The label field consists of two subfields each containing a symbol. The value assigned to each symbol depends upon its use.

8. TAG,K $DO 10 , +K

The symbol TAG will be assigned the value of the current location counter. The symbol "K" is assigned the incrementing values (1, 2, ...). (See 1.6.7.)

1.3.1.1. Location Counter Specification

MASM allocates storage for instructions and data under the control of storage location counters. There are 64 location counters available in MASM, numbered 0 through 63. Any location counter may be used or referenced in any sequence. The initial location counter number is zero (0). A program remains under control of that location counter number until a new location counter specification is introduced. When a specific location counter is specified, all subsequent coding is under its control until another location counter is explicitly specified.

1.3.1.2. Externalized Labels

When enough asterisks are suffixed to a MASM label symbol to insert the label at level 0 of the dictionary, the label is known outside the program and is said to be externalized. Such labels are entered into the entry point table in the preamble of the RB element. Some label symbols may not be externalized (see 1.12).

Example:

Assume a processing level of 1, then:

```
TAG*    EQU    6
```

inserts the symbol "TAG" at level 0 of the dictionary, which causes the symbol to be known outside of the program.

1.3.1.3. Node Selectors

MASM permits usage of nodes and selectors as labels. A selector may be any legitimate assembler item, expression, or another node and selector. A label may not be used as its own selector (see 1.5.13).

A selector is enclosed in parentheses immediately following the symbol with no intervening spaces.

Example:

1. A(4)

2. A(3,2)

3. X(1,Y(1))

4. X(4,3,SIZE//2)

1.3.2. Operation

The operation field starts with the first nonblank character following the label field and is terminated by a blank. The first subfield must evaluate to a MASM directive, PROC reference, function reference, or an instruction; in other words, control information. If this is not the case, the operation field is considered void. Subsequent subfields act as operand input for the operation specified.

```
1.          R E S 5
2.  A       E Q U       2
3.          A P R O C , 1 , 2
4.          L A , U
5.          +     14
```

Explanation:

Line 1:

The operation field consists of one subfield and is a valid MASM directive.

Line 2:

The operation field consists of one subfield and is a valid MASM directive.

Line 3:

The operation field consists of three subfields. Subfield 1 is a user-defined procedure. Subfields 2 and 3 are objects which the procedure may reference.

Line 4:

The operation field consists of two subfields. Subfield 1 may be an instruction mnemonic, subfield 2 is operand information to be used when generating the instruction.

Line 5:

The operation field is void.

1.3.3. Operand

The operand field is more precisely defined as the operand part of the functional portion of a MASM statement because it may consist of multiple fields.

The operand part begins with the first nonblank character after the operation field or label field (if the operation field is void) and continues until the end of the functional portion of the statement. As mentioned previously, the operand field may consist of more than one field, or it may be void.

It is not necessary for the operand field to contain the maximum number of subfields implied by the operation field. When omitting a subfield, other than the normal first field or last field, the construct comma-zero-comma (,0,) or two contiguous commas (,,) is necessary. If the last subfield is omitted, a comma is not required after the last coded subfield.

Any subfield referenced but not specified in the operand part is evaluated to zero.

Any subfield of any field of the operand portion may be "flagged" by prefixing the subfield with an asterisk (*). An asterisk cannot stand alone in a subfield, although *0 is acceptable (see 1.8.3).

Example:

```
1.  A       EQU     2
```

The operand part consists of a single field, the value 2.

```
2.          APROC   DOG,*0
```

The operand part consists of a single field with two subfields. The first subfield is the symbol "DOG" and the second subfield is a flagged item with the value 0.

```
3.          LA,U    A0, TAG
```

The operand part consists of a single field with two subfields. The first subfield is the symbol "AO" and the second subfield is the symbol "TAG". The leading space in the second subfield is ignored.

4. APROC APPLE TREE

The operand consists of two fields. Field 1 is the symbol "APPLE" and field 2 is the symbol "TREE".

5. APROC APPLE,,02

The operand part consists of a single field with three subfields. The first subfield is the symbol "APPLE", the second subfield is void and will be evaluated to zero, and the third subfield evaluates 2.

1.3.4. Comments

As mentioned previously, the part of a MASM statement not occupied by the functional part is the comment part. All characters are allowed in the comment part.

Example:

1. APROC1 COMMENT PART OF STATEMENT

APROC1 is a user-defined procedure that does not reference any operand fields, therefore the functional portion of the statement ends with the space following the symbol "APROC1".

2. APROC2 APPLE; A COMMENT

APROC2 is a user-defined procedure which references two fields. The line continuation marks the end of the functional portion of this line.

3. TREE ANOTHER COMMENT

This line contains the continuation of the functional part of Line 2. "TREE" is the second field of the procedure reference. The rest of the line is commentary.

4. APROC4 APPLE . A COMMENT MAY GO HERE

APROC4 is a user-defined procedure which may or may not reference more than one field. The construct space-period-space (.) marks the end of the functional portion of the MASM statement. Any references to fields beyond "APPLE" will evaluate to zero.

1.3.5. Other Considerations

1.3.5.1. Line Continuation

The line continuation character is the semicolon (;). If a MASM statement is longer than 80 characters, a semicolon may be used to continue the statement onto the next line. There is no limit on the number of continuation lines. However, readability should be taken into consideration in complex statement constructs.

If MASM encounters a semicolon outside a quoted string, the scanning of the functional part of the line is terminated and the remainder of the line is assumed to be the comment portion. The functional part of the next line is assumed to begin with the first nonblank character. Text on the continuation line may begin in column 1.

Example:

```
1.  TAG       RES     ;     A COMMENT
2.            15              ANOTHER COMMENT
3.  TAG ;
4.  RES ;
5.            15
```

Explanation:

Lines 1 and 2 produce the same results as lines 3, 4, 5.

If the user wants to continue a quoted string, he may terminate the string of the current line with a single quote immediately followed by a semicolon. If the first nonblank character on the continuation line is a single quote and the string on the continuation line is a valid MASM quoted string the two strings is concatenated.

Example:

```
1.            'ABCD';  THE COMMENT PORTION
2.      'EFG'
```

Explanation:

Lines 1 and 2 produce a string 'ABCDEFG'

1.3.5.2. Paper Ejection

A slash (/) appearing in column 1 (see 1.3.1) advances the printing to the top of the next page. The slash appears on the new page. The MASM directive "EJECT" also advances to the next page (see 1.6.8).

1.4. DATA GENERATION

A MASM statement with a void operations field and a nonvoid operand field generates data that is output to the relocatable binary element. MASM accomplishes this by an implied call to the $GEN directive (see $GEN directive in 1.6.21).

The operand field has the format:

$$e_1, e_2, \ldots, e_n$$

where e is any valid MASM expression. Each e constitutes a subfield (see 1.2.4.1). If the number of subfields, n, is a divisor of the word size, b, an n-field word is generated with each field n/b bits in size. If n is not a divisor of the word size, then an E-flag is generated.

Each e may have a monadic + or –, or the entire operand may have a monadic + or –. If the entire operand is to have a sign, the operand field must be enclosed in parentheses preceded by the desired sign.

Example:

```
1.            WRD 36
              + 15
```

Example 1 generates one 36-bit word with the value 15. The generated octal output appears as: 000000000017.

```
2.      WRD 32
        -(-4,5,6,7)
```

Example 2 generates a 32-bit word consisting of four 8-bit fields. The generated octal output appears as 004 372 371 370 if broken down into its fields; or 00476574770 as a 32-bit word.

```
3.      WRD 16
        + 3,2,1,0,1,2,3,2
```

Example 3 generates a 16-bit word consisting of 8 fields of 2.

1.4.1. Signed Character Strings

If the character string is signed, the data generated will be right justified, zero filled. The number of words generated, n, varies according to the following relationship:

$$n = (c * b)//k$$

where:

c = numbers of characters
b = number of bits per character
k = number of bits per word

If the value of n is larger than 2, a T-flag is generated. If the word size is greater than 36 bits, n cannot be larger than 1.

Example:

```
1.      $INSERT 'WRD 36',' $FDATA'
```

This instruction sets the word size to 36 bits and the character set to Fieldata. (See 1.6.28.)

```
2.      + 'ABCEF'
```

This line generates the following value:

000607101213

```
3.      - 'ABCEFGH'
```

MASM generates the following octal value:

777777777771
706765646362

1.4.2. Unsigned Character Strings

If the operand field is a string enclosed in single quotes, without a leading sign, a variable number of data words may be generated. The value generated is a left justified space filled set of words. There is no limit on the number of words that may be generated.

Example:

```
1.  $FDATA
2.  'ABCD'
3.  'ABCDEFH'
4.  $ASCII
5.  'ABCD'
```

Explanation:

Line 1:

The character is set to Fieldata.

Line 2:

The octal value generated is:

060710110505

assuming the system character set is Fieldata.

Line 3:

The octal value generated is:

060710111213
150505050505

assuming the system character set is Fieldata.

Line 4:

The system character set is set to ASCII.

Line 5:

The octal value generated is:

101102103104

1.5. VALUES AND EXPRESSIONS

Values are the fundamental elements of MASM subfields. They are computed by the interpretation of expressions, and are retained by assigning them to symbols and selectors of nodes. MASM has a large number of available data types and allows explicit use of typing. This subsection describes the various data types and the syntax and semantics of the expressions that may be constructed to evaluate them.

1.5.1. Values

A knowledge of the various kinds of values (also referred to as data types) is necessary to utilize the full power of MASM for constructing programs. Extremely general PROCs may be constructed which base their operation on the nature of the data submitted as parameters. A full set of built-in functions is available for performing transfer from one data type to another, and for testing the data type of a parameter. These are described in 1.7.

1.5.1.1. Numeric Values

The values which occur most commonly in a program are numeric values. All data generated for the output RB element are numeric data, possibly produced by conversion of other data types. Transfer functions (functions which perform type conversions) may be invoked explicitly by the source language (through calling such a function by name) or implicitly by context.

This subsection details the nature of numeric values and the syntax of their external representation in source language.

1.5.1.1.1. Binary Values

Binary values in MASM may be thought of as integers. The internal arithmetic precision of MASM for binary arithmetic is 72 bits plus sign. All binary computations performed by MASM are done in this 73-bit arithmetic, including those leading to a single precision word being generated in the output.

Elementary numeric items are usually referred to as numbers. Binary numbers may be decimal, octal, or hexadecimal. The interpretation of a number as octal or hexadecimal depends on the setting of a global assembly switch, which is controlled by directives.

Decimal numbers begin with a digit 1, 2, 3, 4, 5, 6, 7, 8, or 9. They may not begin with a 0 and may not contain a decimal point. Any of the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 may be used in a decimal number.

Octal or hexadecimal numbers must begin with a 0. If current mode is octal, only the digits 0, 1, 2, 3, 4, 5, 6, and 7 are permitted in an octal number. If the mode is hexadecimal, the digits permitted are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. If the last character of a hexadecimal constant is a D, it is interpreted as a digit, not the double precision postfix operator. Therefore, parentheses must be placed around the hexadecimal constant with the D suffixed to the right parenthesis to generate a double precision hexadecimal constant.

Another form of binary item is the label reference, which retrieves a value computed earlier by MASM and assigned to the label, by the use of the $EQU directive, the $EQUF directive, or implicit definition through having appeared in the label field of an instruction or other generative line. Labels may have relocation relative to internal location counters of the element being assembled or relative to external symbols. Two relocatable values are not considered equal unless both the absolute part and the relocation are the same. Binary values may also have a FORM attached. A form defines the layout of fields within a word for an item. Such a FORM may be created by use of the I$ built-in form (see 1.6.20), the $EQUF directive, an instruction, or a programmer-defined form (created with the $FORM directive).

Example:

This line is interpreted as a decimal number.

2. 014

This line is interpreted as an octal number provided the $OCTAL directive is in effect.

3. 018
4. 0F

These lines are interpreted as a hexadecimal number provided the $HEX directive is in effect.

5. ABC14

This line is a label reference and the symbol is looked up in the dictionary.

1.5.1.1.2. Parenthetic Expression Items

An expression may be enclosed by parentheses and be preceded by an operator. Such an expression is known as a parenthetic expression. Its primary function is that of algebraic grouping.

Example:

1. 4*(5+2)

The parenthetic expression (5+2) is used to sum 5 and 2 before multiplying by 4.

2. +(6+10)

The "+" preceding the parenthetic expression (6+10) is used to prevent literal generation.

1.5.1.1.3. Line Items

A line item is an expression involving some manipulation other than the operators listed in 1.5.2. The information within the parentheses must be a valid MASM line, exclusive of the label. This means that an operation field (possibly void) must be present. A line item may thus reference an instruction, a PROC, a FORM, or may only generate data. If a PROC is called, it may not increment the current location counter; the current location counter is said to be blocked. The PROC called may call other PROCs inside its own line items; in this way, a number of location counters may come to be blocked. An attempt to alter a blocked location counter results in a T flag.

To detect a line item, MASM evaluates the first expression following a left parenthesis. If the next character after the expression is not a right parenthesis, the character must be a comma or a space. If it is a comma and no significant (not following an operator) space is found before the right parenthesis, an implicit call to $GEN is made. Otherwise, the first expression after the left parenthesis must be a directive, instruction, or PROC call. Note that this means MASM recognizes the format (LA,U A0,1) as a valid line item.

Example:

1. 046+(J TAG)
2. 024+(1,2,3,6)
3. 054+(PVM,U TAG)

Explanation:

Line 1:

The line item, (J TAG), is an 1100 Series instruction mnemonic which when generated is added to the constant 046.

Line 2:

The designated word size is divided into four equal parts and each expression of the line item, (1,2,3,6), is placed into the parts of the word. The result is added to the constant 024.

Line 3:

The line item (PVM,U TAG) calls procedure PVM. The result is then added to the constant 054.

1.5.1.1.4. Literal Items

A literal is an expression enclosed in a set of parentheses; that is, there may be no operators at the same level as the enclosing parentheses (with the exception of unary * and the conditional operators, as described in 1.5.2). Literals are essentially line items without the preceding operator. Since literals are relocatable values, all the rules which apply to relocatable values apply to literals. They usually have an attached FORM as well. To use the value of a literal in an expression, an extra set of parentheses is required. The expression 46+(J BEGIN) does not cause a literal to be generated.

The value of a literal on a summary pass is zero. It is given its true value only in the generative pass. Therefore, caution should be used if the value of a literal is ever tested for conditional interpretation (that is, used before the -> operator or as an operand of a $DO or $IF directive).

Example:

```
1.    A          EQU        (ADR1,ADR2)
2.               (1,TABLE)
3.               LR           R5,(PVM,15 2,ABBC)
4.    AZ         EQU        (F12618 1,14,ADDR)
5.    Z
```

Explanation:

Line 1:

The symbol A is associated with the address of the literal

(ADR1,ADR2).

Line 2:

The address of the literal (1,TABLE) is generated.

Line 3:

The address of the literal (PVM,15 2,ABBC) is used as an input parameter to the control information associated with the symbol LR.

Line 4:

The symbol AZ is associated with the address of the literal:

(F12618 1,14,ADDR).

Line 5:

The symbol Z is set to the value resulting from the evaluation of expression +(ADR1,ADR2). No literal is involved.

1.5.1.1.5. Floating Point Values

Floating point values are used less frequently than binary values. MASM maintains floating point numbers internally with a 12-bit characteristic, a 70-bit mantissa, and two sign bits, irrespective of the final precision of the number to be generated.

Floating point numbers may be either decimal, octal, or hexadecimal, the choice between octal and hexadecimal again depending on the global switch. A floating point number must contain a decimal point (period), which may be the first character, the last character, or embedded. If the decimal point is the last character, a blank may not follow it, or else the period is interpreted as the start of a comment field, rather than as part of a number. (The term "decimal point" is meant to encompass the octal point or hexadecimal point when a floating point number is represented in either of these bases.)

A decimal floating point number may begin with a zero if and only if the next character is the decimal point. Octal (or hexadecimal) floating point numbers always begin with a zero which is followed by a digit. Other rules for binary numbers apply, particularly that for trailing D in hexadecimal mode.

Example:

```
1.      1.0
2.      0.7
3.      00.7
4.      011.7
5.      00.00006
6.      00.8
7.      0A.4B
```

Explanation:

Line 1:

This line is interpreted as a decimal floating point number.

Line 2:

The value is interpreted as a decimal floating point number because the 0 is followed immediately by a decimal point.

Lines 3,4 & 5:

These lines are interpreted as octal floating point numbers if an $OCTAL directive is in effect.

Line 6 & 7:

These values are interpreted as hexadecimal floating point numbers provided the $HEX directive is in effect.

1.5.1.2. String Values

MASM treats strings as an independent data type, converting them to binary only when necessary to generate output data or when required by the context of the expression. In addition to string constants, string values are returned by some of the built-in functions.

String constants are written by coding a single quote ('), followed by any combination of characters (other than a quote), and terminated by another single quote. A single quote may be included in a string by writing two single quotes (' ') at the point where it is desired to generate a single quote. The continuation of a string onto a following line has already been described in 1.3.5. A string may have up to 262143 characters.

Example:

1. A EQU 'A'R

   The symbol A is associated with the value 06 with string attributes, indicating right justification. This assumes the system character set is Fieldata.

2. A EQU 'ABCDEF'DL

   The symbol A is associated with the value 0607101112 with string attributes indicating left justification and double precision (defined in 1.5.2.11). This assumes the system character set is Fieldata.

3. STG EQU 'abc'

   The symbol STG is associated with the value 0141142143 with string attributes indicating right justification and single precision. This assumes the system character set is ASCII.

1.5.1.3. Nodes and Selectors

A node is a point of departure for a tree structure. Each node may have a set of selectors defined for it. The number of selectors is limited only by the amount of storage available. Each selector is defined by a unique nonnegative integer. The integers used need not be in any particular sequence. A particular selector of a node is obtained by writing the number of the selector in parentheses following the node reference. If the value of the selector is itself a node, further selections may be made by writing the number, separated by commas, of each selection within the same set of parentheses. The selector number may be computed by MASM expressions.

The value given to any particular selector of a node may be any legitimate MASM value, including another node. The values of various selectors of the same node need not be of the same type. If a node reference is used in a context which requires a numerical value, the value of the node reference is the number of selectors defined. Just as may be done with other MASM values, node references may be passed as arguments and assigned to labels. A label whose value is a node reference thereby may be referenced as a subscripted label (see 1.3.1). If a node reference is passed as a parameter to a PROC (see 1.8), the number of subscripts of the associated paraform is 2 plus the number permitted for any selection sequence of the node reference itself.

Node references are created by the PROC paraform mechanism, some built-in functions, and the simple writing of a subscripted label in the label field, where the label was not previously defined. Node references may be deleted by removing all references to them through reassignment or through the use of the $DELETE directive. Since the ordinary equality test of MASM tests only for numerical equality (and hence will be satisfied by two nodes with the same number of selectors), a pair of operators is provided which tests two nodes for exact identity (and nonidentity); that is, the test is satisfied only if the operands are the same node.

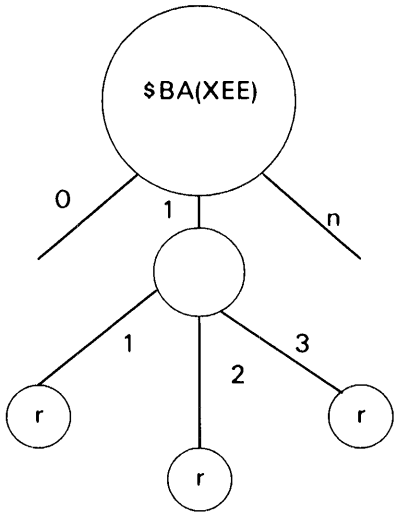
Examples of node references:

1. $BA(XEE)(1)
2. A(4)

Explanation:

Line 1:

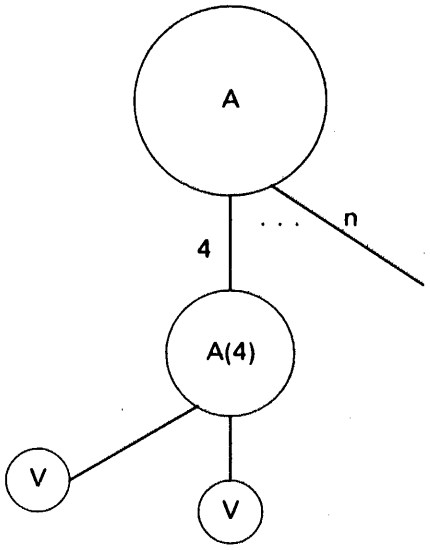
Selector 1 of the node $BA(XEE) points to a node; therefore line 1 is a node reference. (See the $BA function in 1.7.2).



where r indicates relocation information.

Line 2:

Assume A is a node with at least one selector, 4, which also has selectors. Then A(4) is a node reference.



where v indicates some value.

Examples of node reference selectors:

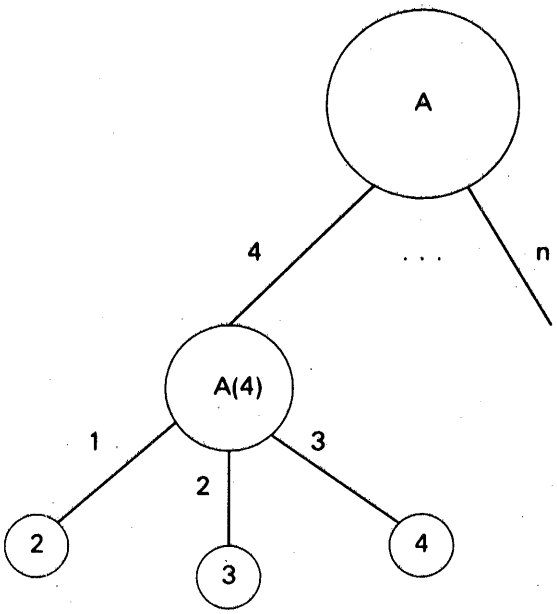
1. A(4,1)

2. $BA(XEE)(1,3)

Explanation:

Line 1:

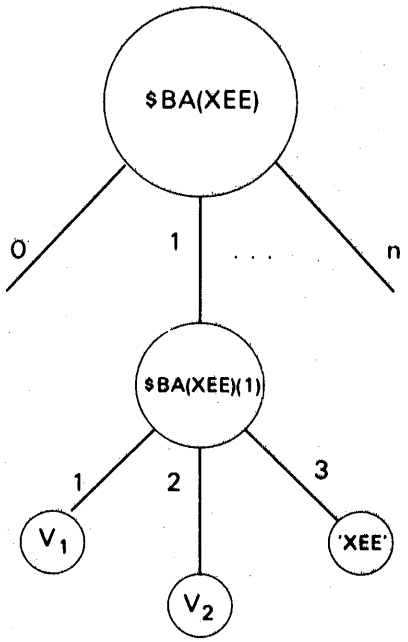
A graphic representation of this is:



If the node A(4) had three values associated with it and they were 2,3,4, then the value of selector A(4,1) would be 2.

Line 2:

A graphic representation is:



If XEE is undefined, the value associated with $BA(XEE)(1,3) would be the string 'XEE' (see 1.7.2).

1.5.1.4. Control Information

Control information is the name used to refer to values associated with MASM directives, built-in functions, PROC names, and function names; in general, this means a value not associated with data, but rather values to control the next stage of processing. Control information requires a special context, which is usually that associated with the operation field of a MASM line or line item.

Any MASM symbol (or node reference selector) may have both a normal data value and a control information value. The value used by MASM at any given time depends on the context being processed. Thus, the operation field of a MASM line (or line item) retrieves the control information associated with an expression, while other contexts require a data type value. The $EQU directive assigns both values; thus it may be seen to have a special context. The slash (/) forces the context for the following expression to be that for control information (which means that that expression is usually a label or other elementary item). This operator may be used to pass a directive into a procedure.

Example:

```
1.  ADD      EQU     AA
2.  MACRO    EQU     $PROC
3.           PVM     14,/LA
```

Explanation:

Line 1:

The symbol ADD may be used as the 1100 Series instruction mnemonic AA (Add to A).

Line 2:

The symbol MACRO may be used as the MASM directive $PROC.

Line 3:

Assuming the symbol PVM is a procedure call, then the symbol LA is passed into the procedure. Without the prefix operator, /, the symbol LA would be evaluated, with the procedure receiving the resultant value.

1.5.2. Expressions and Operators

MASM contains both unary and binary (monadic and dyadic) operators, some of which use the same symbol. Where a symbol has more than one meaning, the context is used to determine which operator is intended.

Because all MASM arithmetic is performed in high precision, the programmer need not be concerned with single and double precision mixed mode arithmetic. The distinction between precision is made only when data is to be generated, and is fully under the control of the programmer when the level 10 operators are used (see 1.5.2.11).

Some operators demand a particular form for their operands. In some cases a transfer function is automatically invoked to insure that the operands are of the proper type. Thus, binary numbers are converted to floating point if used in mixed mode, and similarly, strings are converted to binary.

Certain transfer functions are not defined, such as those converting relocatable values to floating point. If mixed mode arithmetic of this type is attempted, an R-flag is generated.

Relocatable values may not be operated on by logical operators, string operators, or scaling operators, nor may they be multiplied by a value other than absolute 1 or 0, divided by a value other than 1, or combined with floating point numbers. Violation of these restrictions will produce an R-flag and relocation is lost.

Table 1-2. The Hierarchy of Operators in MASM

Level	Operators	Description
0	-> ! * /	conditional if-then conditional else (exclamation point) (unary) set leading asterisk flag (unary) control information context
1	\	(unary) negation operator (NOT)
2	<	less than
	<= > >= = <> == =/=	less than or equal greater than greater than or equal equal not equal node identity node nonidentity
3	:	string concatentation
4	++ —	bit logical OR bit logical XOR
5	**	bit logical AND

Table 1-2. The Hierarchy of Operators in MASM (continued)

Level	Operators	Description
6	+	arithmetic addition
	-	arithmetic subtraction
7	*	arithmetic multiplication
	/	arithmetic division quotient
	//	arithmetic division covered quotient
	///	arithmetic division remainder
8	*/	fixed point binary scaling
	*+	floating point power of 10 positive scaling
	*-	floating point power of 10 negative scaling
	*//	floating point power of 2 scaling
9	+	(unary) positive number
	-	(unary) arithmetic negative
10	D	(postfix) double precision
	S	(postfix) single precision
	L	(postfix) left justify space fill
	R	(postfix) right justify zero fill

Generally, these operators are left associative; that is, operations are performed from left to right in an expression. However, there are some exceptions. Relational operators function globally, so that the expression A>B>C means the same as (A>B)**(B>C).

The concatenation operations in an expression are performed all at once (other operators and parentheses permitting) to save the storage otherwise required for intermediate results. The operators at level 0 are not always left associative.

Level *n* operators operate an expression whose level is greater than *n* and produce level *n* expressions.

Example:

Let: # be a *n* binary operator and
a, *b*, and *c* be expressions.

Then in the expression:

c <= *a* # *b*

c is a level *n* expression *a* and *b* are expressions whose level is greater than *n*.

1.5.2.1. Level 0 Operators

The level 0 operators include conditional expression operators and two special unary operators. Their structure is perhaps the most complex of all the MASM operators, primarily because they do not follow simple rules of left associativity, and because, for conditional expressions, more than one operator is used to form an expression.

Conditional expressions allow the alternative generation of values without multiplying various expressions by zero or one. Moreover, the unused expression in a conditional expression is not evaluated, thus permitting the use of functions with side effects in cases where the zero–one multiplication technique would forbid them. Since microstrings (see 1.10) are evaluated prior to the evaluation of the line, this does not apply to them. On the other hand, this means that some errors may fail to be detected, because the expression containing them is not evaluated.

If *a*, *b*, *c*, etc. denote level one expressions, then level zero expressions are constructed from them in the following way. A level one expression is a level zero expression. So are constructions of the form:

a -> *b* ! *c*
a -> *b*

which are said to be conditional expressions. The first is said to be complete and the second incomplete (because the ! *c* is missing). Conditional expressions may be substituted for *b* and *c* in the above expressions with the restriction that the *b* in *a* -> *b* ! *c* may be replaced only by a complete conditional expression. In this fashion, complex conditional expressions may be built. If all of the conditional expressions involved in such a construction are complete, then the resulting expression is complete; otherwise, it is incomplete. With these rules, the operators -> and ! in a conditional expression can be matched as follows: As the expression is examined from left to right, then each ! matches with the most recent unmatched -> operator. For example, the expression:

a -> *b* -> *c* ! *d* -> *e* ! *f* ! *g* - > *h*

may be parenthesized as:

a ->(*b* -> *c* !(*d* -> *e* ! *f*)) !(*g* -> *h*).

Note that this means that the matching process is performed only at the same parenthesis level; parentheses create new higher level expressions to be treated as units when scanning conditional expressions. The two level zero monadic operators * and / may appear at the beginning of a level zero expression or following a -> or ! operator.

The operators "->" and "!" are used together to form conditional expressions of the form *a* -> *b* ! *c* and *a* -> *b* as described above. To evaluate these expressions, *a* is first evaluated to a binary value without relocation. If *a* is not equal to zero, then the value of the conditional expression is the value of *b*. If *a* is equal to zero, then the value of the conditional expression is the value of *c* or void, if *c* is not present.

Since the value of 0 in MASM may be thought of as "false" and any nonzero value as "true", the expression *a* -> *b* ! *c* may be interpreted as "IF *a* THEN *b* ELSE *c*", while *a* -> *b* may be understood as "IF *a* THEN *b* ELSE *void*". Note that *void* is not the same as a value of zero or the null string. There are contexts where this distinction is important, such as the parameters on a $PROC directive line.

A conditional expression may be used to compute control information for use as a function or as a directive. Thus, the line:

```
T->LA! LNA          A0,TAG
```

will generate either:

```
LA          A0,TAG      (T true)
```

or

```
LNA         A0,TAG      (T false)
```

depending on the value of T.

The level zero unary operators "*" and "/" may appear at the beginning of a level zero expression or after any occurrence of the "->" or "!" operators. If these operators appear in front of a conditional expression, then they are applied to the expression; otherwise, they are applied to the level one expression which they precede. For example:

```
*a->b-*cld
```

and

```
/a-b-/cld
```

The first monadic operator is applied to the whole expression, and the second is applied only to *c*.

The unary operator "*" causes the leading asterisk flag to be set. This flag may be tested by the appropriate node selector reference.

The unary operator "/" causes the operand to be converted to control information, if possible. If this is not possible, an error is noted, and the value of the expression is zero. The operator allows the user to pass directives as parameters without the directives being evaluated when they are passed. The control information is passed so the directive may be evaluated later.

Some of the level 0 operators are exceptions to the usual rule that literals are not generated if there is an operator at the same parenthesis level. In particular, the unary "*" does not suppress literal creation, nor does the conditional operators for the consequence of the conditional expression. Therefore, in the expressions:

```
(a)->(b)!(c)
```

and

```
*(d)
```

only the "(*a*)" is not a literal.

1.5.2.2. Level 1 Operators

The only level 1 operator is the unary operator "\", the NOT operator. Its operand is converted to binary without relocation. If any bit of the operand conversion result is one, then the result is zero; otherwise, the value is one.

Two consecutive NOT operators will convert any nonzero value to one, while leaving zero intact. The unary NOT operator is distinct from the negation operator (unary "-"), in that it works on MASM truth values, such as those expected by the $IF directive.

1.5.2.3. Level 2 Operators

The level 2 operators include all the relational operators ("=", "<>", "<", "<=", ">", ">=", "==", and "=/=") and return values of one or zero, according to whether the relation specified is true or false. A simple relation has the form $e_1\ r\ e_2$, where r is a level 2 operator, and e_1 and e_2 are level 3 expressions. A compound relation has the form:

$$e_1\ r_1\ e_2\ r_2\ \cdots\ r_n\ e_{n+1}$$

with notation as before. The value of this expression is 1 if all of the relations $e_m\ r_m\ e_{m+1}$ for $m = 1,...,n$ are true, and zero otherwise. The expressions $e_1, ..., e_n$ are evaluated only up to the first pair for which the relation is false. For this reason, some errors may not be detected in the unevaluated portion of the level 2 expression. This does not apply to microstring substitution, which takes place before expression evaluation commences.

Since the mode of e_{m-1} and e_{m+1} may differ from each other and from e_m, the value of e_m may be converted twice. Both of these conversions are made from the same original value, rather than one being converted from the other. This achieves maximum consistency.

Except for the operators "==" and "=/=" (which operate on node references), the level 2 operators operate on binary, floating, and string values. For a simple relation formed from one of these operators, if one of the operands is floating, then both operands are converted to floating; otherwise, if either operand is binary, then both operands are converted to binary.

Binary and floating values are compared using the natural orderings of these number systems. In addition, the "=" and "<>" operators also compare relocation for binary operands; two relocatable binary operands are equal only if the absolute part (offset) and all relocations are the same. For the operators "<", "<=", ">", and ">=", the relocation of two binary operands must be the same or an R-flag results.

When strings are compared, they are first converted to strings with the same character size as follows: If either string is in a data character set, then both strings are converted to the data character set. If there is currently no data character set, or if the character sizes of the resulting strings differ, then a V-flag is is indicated. Otherwise, if one of the strings is ASCII, then the other is converted to ASCII. If either string has the left justification attribute, then the strings are compared as left justified strings followed by arbitrarily large numbers of space characters. Otherwise, the strings are compared as right justified strings preceded by arbitrarily large numbers of characters with a code of zero. Once the strings are justified, they are compared according to the lexical ordering based on the collating sequence formed by the character codes.

The relation "=" is true if the operands are equal and false if they are not equal. For binary operands, the relation is true if the values are equal and the relocation matches; otherwise the relation is false.

The relation "<>" is true if and only if the relation "=" is false.

The relation "<" is true if the first operand is less than and not equal to the second operand. For binary operands, the relocation of the two operands must match.

The relation "<=" is true if the first operand is less than or equal to the second operand. For binary operands, the relocation of the two operands must match.

The relation ">" is true if the first operand is greater than and not equal to the second operand. For binary operands the relocation of the two operands must match.

The relation ">=" is true if the first operand is greater than or equal to the second operand. For binary operands the relocation of the two operands must match.

The relation "==" is computed by converting both operands to node references. If both node references reference the same node, then the relation is true; otherwise, it is false.

The relation "=/=" is computed by converting both operands to node references. If both node references reference the same node, then the relation is false; otherwise, it is true.

1.5.2.4. Level 3 Operators

The only level 3 operator is the infix operator " : ". This operator is used to concatenate strings. A level 3 expression has the form:

$$e_1 : e_2 : e_3 : \ldots : e_n$$

where $e_1, \ldots, e_n$ are level 4 expressions which are converted to strings. The value of the expression is the result of concatenating the strings $e_1, \ldots, e_n$. If any of the strings is in a data character set, then all of the strings are converted to a data character set. If there is presently no data character set, or if the character sizes of the resulting strings fail to match, an error is noted. Otherwise, if any of the strings is in ASCII, then all the strings are converted to ASCII. After the conversions, the strings are concatenated in the order they appear. The justification and space attributes of the resulting string are those of the last operand e_n. If any of the operands is double precision, then the result is double precision; otherwise, the result is single precision.

1.5.2.5. Level 4 Operators

The level 4 operators perform bitwise logical operations on binary values with appropriate extensions for 73-bit arithmetic.

For the operator "++", the operands are converted to binary without relocation. The result is the binary value formed by ORing the two operands. A bit in the result is set to one if and only if at least one of the corresponding bits in the operands is one.

The result of "++" is single precision if and only if both operands are single precision. If only one of the operands has a form, or if both of the operands have the same form, the result retains that form. Otherwise the result does not have a form.

For the operator "--", the operands are converted to binary without relocation. The result is the binary value formed by XORing the two operands together. A bit in the result is one if and only if exactly one of the corresponding bits in the operands is one. The precision and form attributes are treated in the same way as for the "++" operator.

1.5.2.6. Level 5 Operators

The level 5 operator performs a bitwise logical operation on binary values, with appropriate extensions for 73-bit arithmetic.

For the operator "**", the operands are converted to binary without relocation. The result is the binary value formed by ANDing the two operands together. A bit in the result is set to one if and only if both of the corresponding bits in the operands are one. The precision and form attributes are treated in the same way as for the "++" operator.

1.5.2.7. Level 6 Operators

The level 6 operators perform arithmetic addition and subtraction on binary or floating point operands. If at least one operand is floating point, the other operand is converted to floating point and the result is floating point. Otherwise, the operands are converted to binary, with a truncation error noted if a string exceeds 72 bits. The "+" operator gives the arithmetic sum of its two operands, while the "-" operator gives the arithmetic difference of its two operands.

Relocation information is preserved by these operators. However, it should be noted that a result may be produced which cannot be placed in the output element (such as negative relocation relative to the base of a location counter). These limitations are due to the nature of 1100 Series RB format and the 1100 Series Collector, and are not inherent limitations of MASM. Such values may be kept and used for later computation without restriction.

1.5.2.8. Level 7 Operators

The level 7 operators perform arithmetic operations related to multiplication and division; their conversion requirements are the same as for the level 6 operators.

Relocation information is preserved only in the case that a relocatable value is multiplied or divided by one. Relocation information is lost if a relocatable value is multiplied by zero, but no error indication is given. In all other cases, relocation is lost, the binary value without relocation is used, and an R-flag results.

The "*" operator computes the arithmetic product of its two operands. A result exceeding 72 bits produces a T-flag.

The "/" operator computes the arithmetic quotient of its two operands. A divide check condition produces a T-flag.

The "//" operator computes the arithmetic covered quotient of its two operands, and is meaningful only for binary arithmetic. The covered quotient is equal to the ordinary quotient if the division remainder is zero, and is otherwise one greater than the ordinary quotient.

The "///" operator computes the remainder from the arithmetic division of its two operands, and is meaningful only for binary arithmetic.

1.5.2.9. Level 8 Operators

The level 8 operators perform shifting and scaling on binary and floating point values. The right hand operand of each of these operators may not be a floating point value; nonnumeric operands are converted to binary. Relocatable operands cause R-flags to be indicated.

The "*/" operator causes its left hand operand to be multiplied by the power of two given by the right hand operand. If the right hand operand is positive, this is a left (logical) shift. If the right hand operand is negative, this is a right (arithmetic) shift. An error results if either operand is floating point (T-flag), the second operand is relocatable (R-flag), or if too large a value (T-flag) is generated. The result is a binary value.

The "*+" operator causes its left hand operand to be converted to floating point, if necessary, and multiplies it by the power of 10 given by the right hand operand. The right hand operand may not be floating point, and neither operand may have relocation. The result is always floating point.

The "*-" operator causes its left hand operand to be converted to floating point, if necessary, and divides it by the power of 10 given by the right hand operand. The right hand operand may not be floating point, and neither operand may have relocation. The result is always floating point.

The "*//" operator performs floating point scaling by a power of two, and is intended for use primarily with octal or hexadecimal floating point numbers. It is similar in all respects to the "*+" operator, except that multiplication is by a power of two (possibly negative) instead of a power of 10.

1.5.2.10. Level 9 Operators

The level 9 operators are the unary "+" and unary "-". Both of them force conversion of nonnumeric operands to binary, and both permit relocation. Additionally, the "-" operator returns the ones complement of its operand. The unary "+" operator is used primarily to establish a numeric context, such as converting a node reference to its selector count in situations where either a node reference or a binary value are permitted. The unary "+" is also used to prevent a literal from being generated when a line item is written and the full 73-bit value and form are required.

1.5.2.11. Level 10 Operators

The level 10 operators are postfix operators, and are used primarily to specify attributes of a value used when data is being generated.

The "S" postfix operator converts its operand to a value and sets the precision attribute to single precision.

The "D" postfix operator converts its operand to a value and sets the precision attribute to double precision. If the binary representation mode is hexadecimal, a trailing D on a hexadecimal number is interpreted as part of the number rather than as the double precision postfix operator; therefore, a pair of parentheses must be used in this case, as in (05FF)D.

The "L" postfix operator converts its operand to a string and sets the justification attribute to left justification, space fill.

The "R" postfix operator converts its operand to a string and sets the justification attribute to right justification, zero fill.

Examples:

```
1.      047D
2.      'abc'D
3.      'const'R
4.      FCTN( 'X'L)
```

1.5.2.12. The Flag Attribute

The flag attribute of a value, also referred to as the leading asterisk flag, is clear for the results of most of the operators described above, even if one or more of the operands had the flag set. The unary "*" is generally the only way to set the flag, although some built-in functions return a node some of whose selectors may have the flag set. Therefore, the programmer should exercise caution when computing with flagged values, so that tests are not made for the flag on the results of such computations. (See 1-8 for referencing flagged values.)

The flag attribute is generally used only on values which are selectors of node references. Therefore, the flag attribute for a selection is tested by prefixing the last selector with an asterisk (as in ABC(1,3,*5)). A built-in function ($IBITS) is also available to test for the flag.

1.6. ASSEMBLER DIRECTIVES

All MASM directives (except for machine instructions), like all MASM built-in functions, begin with a "$" character. This means that they may not be written in the label field of a MASM line and, therefore, redefinition of directives is impossible.

In addition to the basic directive forms with the leading "$", all MASM directives have synonyms without the "$" character. These synonyms, unlike the basic forms, may be redefined by the programmer, thus providing flexibility without loss of control. Some directives have more than one synonym. In the following subsections, only those synonyms which cannot be derived from the directive name by deleting the "$" will be noted. In other words, if $ABC is a directive, the user may assume that ABC is a synonym for $ABC, even if it is not stated explicitly. In the examples throughout this manual, both the basic forms and their synonyms are used, so that the programmer may become familiar with both alternatives.

1.6.1. $ANDF (And If)

The $ANDF directive is called by:

```
$ANDF e
```

where *e* is a binary value with no relocation. This directive is used in conjunction with the $IF, $ELSE, $ENDF, and $ELSF directives. This directive will be ignored if MASM is already skipping images within a conditional construction. If not, *e* is evaluated. If *e* is nonzero, no action is taken; if *e* is zero, MASM begins skipping images.

If *d* denotes one of the directives $ENDF, $ELSE, or $ELSF, the first construction is equivalent to (and shorter than) the second:

```
.                    .
.                    .
.                    .
$ANDF    e           $IF      e
.                    .
.                    .
.                    .
d                    $ENDF
                     d
```

Example:

```
1.      $IF     A>0
2.      +       A
3.      $ANDF   B>0
4.      +       B
5.      $ENDF
```

Explanation:

Line 1:

If the expression A>O is true then the statements on lines 2 and 3 are interpreted. If A>O is false then lines 2 through 4 are skipped.

Line 3:

If line 3 is interpreted and the expression B>O is true then line 4 is interpreted. If the expression B>O is false then line 4 is skipped.

Line 5:

The $ENDF marks the end of the conditional construct.

1.6.2. $ASCII (Set Character Mode to ASCII)

This directive requires no parameters. It sets the system character set to ASCII. This directive has special interaction with the $CHAR directive.

1.6.3. $CHAR (Define a Data Character Set)

$CHAR is called as follows:

$CHAR,$e_o$ e_i,f_i,... e_n, f_n

where e_o indicates character frame size and in converted by the rules for parameter conversion to a non negative number is the range 1 to 36.

The order pairs (e,f) indicates the current and future character codes, respectively. These are converted by the rules for parameter conversion to non negative numbers.

If e_o is void the previous character frame size remains 1 in effect.

If all parameters are void then any data character character set in inactivated and MASM reverts to the system character set.

1. If there is no data character set, in effect prior to the use of $CHAR, a table is constructed which translates each character of the system character set into the character of the new character set.

2. If there is a data character set with a translation table constructed for the same system character set, a copy of that table is used.

3. If there is a data character set with a translation table constructed for a system character set different from the one specified with the current $CHAR, the translation table is constructed as follows: Each character of the current system character set is translated into the equivalent character of the previous system character set and then, via the translation table, into the final code.

Once the table has been constructed, then the parameter pairs e_i,f_i are taken in order and modify the table to indicate that the character of the system character set with the code e_i is translated into f_i. If the system character set is Fieldata, then the number of pairs may not exceed 64; if the system character set is ASCII, the number of pairs may not exceed 128. No value of f_i may exceed 1*/36-1.

Note that this means that the translate table need not be completely redefined in order to alter only a few characters of it.

The existence of a character translation table overrides the setting of the system character set by the $ASCII or $FDATA directives.

Example:

```
1.      $CHAR 'A',077
2.      +          'AB'
3.      $ASCII
4.      +          'AB'
5.      $CHAR
6.      +          'AB'
```

Explanation:

The example assumes the system character set is Fieldata.

Line 1:

A character translation table is built which converts a value of 06 to 077.

Line 2:

Using the character translation table MASM generates a 07707.

Line 3:

The $ASCII directive sets the system character set to ASCII.

Line 4:

A value of 07707 is generated because the character translation table overrides the system character set.

Line 5:

A $CHAR directive with no parameters removes the character translation table.

Line 6:

MASM generates a value of 101102 because the system character set is ASCII.

1.6.4. $DEF (Establish Definition Mode)

The $DEF directive (see 1.13) establishes definition mode for the main assembly. This directive must be interpreted before the beginning of the second pass of the main assembly. If an attempt is made to generate data in a definition mode assembly, an I-flag is generated. This directive is ignored if encountered during the second pass of the main assembly.

The $DEF directive of MASM is distinct from the DEF directive used by PDP. Since DEF should not occur inside PROCs, the two never interfere with each other. The PDP DEF may still be used in PROCs intended to be processed by MASM. See SPERRY UNIVAC 1100 Series, Vol. 3, System Processors Programmer Reference, UP-4144.3, current version. for details of the use of DEF with PDP.

1.6.5. $DELETE (Delete a Definition)

The $DELETE directive is called as follows:

label $DELETE

where there are no parameters, but the label field is required. This directive deletes the final relationship of a definition, which may delete both data and control information, since an identifier may reference both. If the *label* is a selection $a(s_1, ..., s_n)$, the effect of $DELETE is to delete the selector s_n of the selection $a(s_1, ... s_{n-1})$. This means that $DELETE may be used to prune trees built up of nodes and selectors. Nodes and PROC sample blocks, which may be referenced from more than one place, are deleted when all references to them are deleted. $DELETE may be used in definition mode to remove PROC sample blocks when the PROC is called solely to establish definitions.

1.6.6. $DISPLAY (Display Information)

The $DISPLAY directive is called as follows:

$DISPLAY $e_1, e_2, ..., e_n$

where $e_1, ..., e_n$ are binary values or strings in a system character set. If the current subassembly pass is not generative, this directive is ignored. Otherwise, the strings and the binary values are displayed in the printed listing, if any. The strings are printed in the position normally occupied by the source image, except that line numbers do not appear. If a string is flagged, an E-flag is produced. A binary value is printed in the position normally occupied by the assembler output except that no location counter is specified. A binary value is always printed as soon as it is encountered, but a string is only printed when another string is encountered in the parameter list, a binary value is printed, or the end of the parameter list is reached.

If V is a binary value, then of the two statements:

```
DISPLAY   'V',V
DISPLAY   V,'V'
DISPLAY   *'ERROR',V
```

the first displays the information on one line, the second requires two lines for the display, while the third will produce an E-flag and display the 'ERROR' and V on the same line.

This directive is intended to be used to provide error indication messages from inside PROCs.

This directive may also be used to document assembly-time actions such as large table generations which are not otherwise readable. Strings may be composed dynamically with the use of the concatenation operators.

1.6.7. $DO (Repetitive Generation of a Line)

The $DO directive is called as follows:

label $DO *rpt ,line*

where *label* is optional and may be any identifier (selections not permitted), *line* is any valid MASM line (excluding leveler and page ejector), and *rpt* is from one to three binary values separated by commas. At least one space must occur between *rpt* and the following comma, and if the line to be generated has a label, it must follow the comma immediately. If *rpt* is only one value, it is

interpreted as *end*; two values are interpreted as *start* and *end*; while three values are interpreted as *start*, *end*, and *step*. If *step* or *start* are not specified, values of 1 are used. The *label* is set to the value *start* and incremented by *step* until the value *end* is reached or passed. For each value given to the label, the specified line is interpreted once.

The *start* and *end* values must be nonnegative binary values without relocation and may not exceed 262143. The *step* field may be positive or negative, but not zero, and may not exceed 131071 in magnitude. If (*end–start*)/*step* is negative, the line is interpreted zero times, and the label is not defined. Any microstrings in the object line are interpreted each time the line is interpreted; microstrings preceding the space–comma pair are interpreted once (before initiating the $DO). $DO directives may be nested. A $DO repetition may be terminated by the $ENDD directive before the full number of repetitions are performed.

1.6.8. $EJECT (Eject the Page)

This directive requires no parameters. If the current subassembly pass is not generative, this directive is ignored. Otherwise, the paper is advanced so that the following printing begins on a new page.

1.6.9. $ELSE (Conditional Interpretation Alternative)

The $ELSE directive requires no parameters. It is used with the $IF and $ENDF directives to establish an alternate set of code to be interpreted. If MASM is conditionally skipping images when $ELSE is encountered, skipping is discontinued and interpretation is begun. If MASM is conditionally interpreting images when $ELSE is encountered, interpretation is discontinued and skipping is begun.

1.6.10. $ELSF (Cond. Interp. Conditional Alternative)

The $ELSF directive is called by:

```
$ELSF
```

where *e* is a binary value without relocation. This directive is used in conjunction with the $IF–$ENOF directives. This directive behaves as a $ELSE directive if MASM is already interpreting images. If MASM is skipping images when this directive is encountered, MASM evaluates the expression and either interprets or skips the images following.

Example:

```
1:      $IF     A          2:      $IF     A
          .                          .
          .                          .
          .                          .
          a                          a
          .                          .
          .                          .
          .                          .
        $ELSF  B                   $ELSE
          .                        $IF       B
          .                          .
          .                          .
          b                          .
          .                        b
          .                          .
          .                          .
        $ENDF                        .
                                   $ENDF
                                   $ENDF
```

Examples 1 and 2 are logically the same; however, example 1 is shorter.

1.6.11. $END (End of a Subassembly)

The $END directive is called as follows:

```
$END        e
```

where *e* is converted according to the rules for parameter conversion. The $END directive terminates a subassembly and ends a PROC or function sample. When used to terminate a PROC or function sample, the $END directive may not be conditionally generated. When a sample is being interpreted, however, an $END directive may be conditionally generated and thus terminates execution of that particular PROC or function.

For a PROC, the expression *e* is ignored if present. For a function, the value of *e* is returned as the value of the function call, and *e* may have any value, including strings, nodes, or control information. If *e* is present on the $END directive which terminates the main assembly, the value of *e* must be binary with exactly one relocation item (which is not an external reference). This indicates to the Collector (and ultimately to the Executive) the address at which the execution of the generated absolute program is to begin. Naturally, if more than one element in a collection has a transfer address *e* specified by a main assembly $END directive, ambiguity exists which must be resolved by Collector source language. An element which has a starting transfer address defined is colloquially said to be a main program. For further details, refer to the 1100 Series, Volume 2, EXEC Programmer Reference, UP-4144.2 (current version).

1.6.12. $ENDD (End $DO Iteration)

This directive requires no parameters and may be used only if a $DO repetition is being performed. The $ENDD terminates the current active group of nested $DO repetitions.

1.6.13. $ENDF (End Conditional Interpretation Group)

This directive ends the conditional code generation group introduced by the most recent use of the $IF directive. It requires no parameters. Conditional interpretation or skipping for this group stops, and the mode of interpretation reverts to that effective for the next outer level of $IF–$ENDF, if any.

$ENDF has the synonym OFF.

1.6.14. $ENDI (End $REPEAT Iteration)

The $ENDI directive requires no parameters and may be used only if a $REPEAT construction is active. It terminates the iteration of the currently active $REPEAT group, thereby giving control to the next outer $REPEAT group, if any, or else to the main assembly.

1.6.15. $ENDR (End a $REPEAT Construction)

This directive requires no parameters and may not have a leveler or label field because it is not saved as part of $REPEAT sample. This directive has two functions. First, when a $REPEAT group sample is being picked up prior to beginning iteration, it serves to indicate the end of a $REPEAT group construction, and may therefore not be conditionally generated when it serves this purpose.

When $ENDR is encountered during repetition, either at the end of the $REPEAT group, or through its conditional generation, the current iteration is ended, the counter is incremented, and the next iteration begins.

1.6.16. $EQU (Equate a Value)

The $EQU directive is called as follows:

```
label       $EQU       e
```

where *e* is an expression. If the label is absent, no action is taken. Otherwise, the expression *e* is converted according to the rules for parameter conversion, with one exception. Before an undefined identifier is converted to binary, the dictionary is searched to see if the identifier is defined as a directive, an internal name, or a procedure name. If such a definition is found, it is taken as the value of the expression. The label is then given the value of the conversion as its definition.

Of the two statements:

```
MACRO     EQU       PROC
MACRO     EQU       /PROC
```

the second is preferred, since it is always unambiguous, although they both have the same effect.

Symbols given definitions by use of the $EQU directive are known as explicit definitions and may be redefined at will without generation of a D–flag.

1.6.17. $EQUF (Equate a Field)

The EQUF directive is called as follows:

label EQUF *u,x,j*

where *u,x,j* are converted to binary values. For a detailed description, see Section 2.

1.6.18. $FDATA (Set System Character Set to Fieldata)

The $FDATA directive requires no parameters and sets the system character set to Fieldata. It has the synonym FIELDATA. This directive has special interaction with the $CHAR directive as described in 1.6.3.

1.6.19. $FORM (Define a FORM)

The $FORM directive is called as follows:

label $FORM $e_1, e_2, \ldots, e_n$

where $e_1, \ldots, e_n$ are integer values greater than zero whose sum is less than 73. The integer values in the operand field represent the length in bits of the field of a word (or double word).

The label used on the $FORM is defined as a FORM name, which may be used in the operation field of a MASM line to specify generation of a datum. The label is used as a FORM reference as follows:

label $d_1, d_2, \ldots, d_n$

each d_i is converted to a binary value and mapped into a field of size e_i, as specified on the $FORM line defining the FORM name. The fields of a form are adjacent and right justified within a word or double word.

Example:

```
1.  PF      FORM    12,6,18
2.          PF      5,1,TAG
```

Explanation:

Line 1:

The symbol PF is the form name and is associated with a 36-bit word divided into three fields of 12, 6, and 18 bits.

Line 2:

Line 2, a form reference line, produces a 36-bit word with the values 5,1,TAG in the fields defined by the symbol PF. If the symbol TAG was associated with the value 01000, then the octal representation is:

000501001000

A field in a FORM reference can be a line item. If the form of the line item is identical to the form referenced, and is not a literal, the corresponding fields from both forms referenced are ORed.

Example:

```
1.  FA      FORM    12,6,18
2.  FB      FORM    12,6,18
3.  S1      EQU     +(FA   0,1,TAG)
4.          FB      4,S1,0
```

Explanation:

Lines 1 and 2:

Two symbols are defined, each having an associated form.

Line 3:

A line item is created with one of these forms, FA. If the symbol TAG has the value 01000, then the octal representation is:

000001001000

The + preceding the line item inhibits literal generation.

Line 4:

A form reference line using S1 as one of the values may be represented in octal as:

000401001000

The result was produced as follows:

```
    S1 = 000001001000
    FB = 000400000000
result = 000401001000
```

MASM provides a built-in FORM with the name I$ which corresponds to the 1100 Series instruction word format. It may be thought of as having the definition:

```
I$          $FORM     6,4,4,4,2,16
```

and may be used at will by the programmer.

1.6.20. $FUNC (Define a function)

The $FUNC directive is dealt with in detail in 1.9. The label on the $FUNC directive is assigned the value of the parameter tree when the function body is being interpreted. If this label is externalized, the label is defined as a function name with no entry parameter whose entry point is at the beginning of the function body.

1.6.21. $GEN (Data Generation)

The $GEN directive is invoked by the call:

$GEN $e_1,\ldots,e_n$

where the action depends on the number of operands. If only one operand is present, its value is generated as data for the output element. When n is greater than one, the current word size is divided into n equal fields, each e_i is converted to a binary value, and the value of e_i is generated in the i^{th} field. The $GEN directive is invoked implicitly for lines which have no operation field or a void operation field.

1.6.22. $GFORM (Generalized FORM)

The $GFORM directive is called by:

$GFORM $f_1,e_1,f_2,e_2,\ldots,f_n,e_n$

where all f_i and e_i are converted to binary values, and the f_i must be unrelocated positive integers whose sum is less than 73. The $GFORM directive provides the same effect as if the two lines

```
F       FORM    f1,...,fn
        F       e1,...,en
```

are written, without actually creating the form F. If any f_i happens to be 0, the corresponding e_i is ignored after conversion to binary.

1.6.23. $GO (Transfer to a NAME)

$GO is called by:

$GO n

where n must evaluate to an internal NAME (a label which appears on a $NAME directive). If the $GO is in the main assembly, only forward transfer is possible; MASM will begin skipping images until the specified NAME is encountered. If the $GO is in a procedure or function, transfer may be made to any NAME of that procedure or function, or any external NAME of any other function or procedure. If the transfer is out of the present function or procedure, a diagnostic G-flag is produced. Such transfers are lateral transfers and do not change the subassembly nesting level. A forward $GO within a procedure/function to a nonexternalized NAME is done by skipping images and may thus be slower than other $GO operations. To terminate the present procedure or function interpretation, it is not necessary to do a $GO to a NAME immediately before the $END at the end of the sample. The following alternatives are preferable:

```
DO    1 , END or 1-END using conditional operators
DO    e , END or e-END using conditional operators
```

The first of these is an unconditional termination, while the second is conditioned on the value of the expression e.

1.6.24. $HEX (Set Binary Representation to Hexadecimal)

The $HEX directive requires no parameters and sets the binary representation mode to hexadecimal.

1.6.25. $IF (Conditional Interpretation)

The $IF directive is called by:

```
$IF        e
```

where *e* is a binary value without relocation. If *e* is omitted, a value of 0 is used. $IF increments the conditional nesting level by 1. If MASM is already skipping images for an outer conditional construction, this action continues. If not, the expression *e* is evaluated and compared with 0. If *e* is not equal to zero, MASM continues to interpret images. If *e* is zero, MASM begins to skip images and continues to do so until a matching $ELSE, $ELSF, or $ENDF is found. $IF has the synonym ON.

1.6.26. $INCLUDE (Include Definitions)

The $INCLUDE directive is called as follows:

```
$INCLUDE  e
```

where *e* is a string in the system character set which is of the form *'n'* or *'n/v'*. Both *n* and *v* must be from 1 to 12 characters from the characters A to Z,0 to 9, "$", and "-". If the current assembly pass is generative, there is no action taken. Otherwise, *n* or *n/v* is assumed to be the name of an omnibus element produced by a MASM definition mode assembly. The Assembler libraries are searched for the element, and, if it is found, the definitions contained in the element are added to the dictionary for the present assembly (see 1.2.5). Any previous definitions for the same symbols are replaced (see 1.13). Done only in Pass 1.

1.6.27. $INFO (Special Information)

The $INFO directive is used to communicate between MASM and the Collector. It is called by:

```
label      $INFO      e0,e1, ... ,en
```

where e_0 is a binary value without relocation in the range 1 to 8. The meanings of $e_1,\ldots,e_n$ and that of the call itself depend on the value of e_0, which is referred to as the group number.

1.6.27.1. Group Number 1 (Mode Settings)

This type of call controls the arithmetic fault mode and quarter—or third-word sensitivity of the output element. The parameter e_1 is a binary value in the range 0 to 077, which is treated as a bit mask whose bits have the meanings as shown in Table 1-3.

Table 1-3. Bit Meanings for $INFO Group Number 1

Bit	Meaning
0*	Specify quarter-or third-word sensitivity
1	Quarter-word sensitive
2	Third-word sensitive
3	Specify arithmetic fault mode
4	Arithmetic fault compatibility mode
5	Arithmetic fault noninterrupt mode

*least significant bit.

If the current subassembly pass is not generative, then the directive is ignored. If bit 0 is set, the values of bits 1 and 2 are substituted for bits 25 and 26 of the flag bits word of the element table entry for the output element. Similarly, if bit 3 is set, then the values of bits 4 and 5 are substituted for bits 29 and 30 of the flag bits word.

For example, the output relocatable element may be marked as quarter-word sensitive by the line:

```
$INFO       1 3
```

1.6.27.2. Group Number 2 (Common Block)

This call specifies that a location counter is a common block. The parameter's meanings are:

- e_1 A string specifying the common block name.
- e_2 The location counter number to be used to refer to the common block.
- e_3 The minimum address of the location counter (optional).

The parameters e_2 and e_3 must be binary values without relocation. The string e_1 must satisfy the Collector requirements for a common block name (no embedded blanks, commas, or periods). The effect of this call is to make e_2 refer to the common block named by e_1 with minimum address e_3. If there are several directives with e_1 and e_2 identical but different values for e_3, the largest of these values is used.

A location counter must have some space allocated if it is to be included in the output element preamble. Therefore, a common block location counter must be incremented somewhere in the element referencing it, either by the $RES directive or by data generation.

For example, the common block COMDATA would be made available as location counter 4 by the line:

```
$INFO       2 'COMDATA',4
```

1.6.27.3. Group Number 3 (Minimum D-Bank Specification)

This call specifies the minimum address for the D-bank. The single parameter e_1 is a nonnegative binary value without relocation which specifies the minimum address. This directive is ignored for nongenerative passes. If there are several such $INFO directives, the largest value specified is used.

1.6.27.4. Group Number 4 (Blank Common Block)

This call specifies that a location counter is a blank common block. The parameter's meanings are:

e_1 The location counter number used to refer to blank common.
e_4 The minimum location counter address.

The interpretations for e_1 and e_2 are the same as for e_2 and e_3, respectively, for a group number of 2. Blank common may also be referred to by a group number 2 $INFO directive with a name of 'BLANK$COMMON' for e_1.

1.6.27.5. Group Number 5 (External Reference Definition)

This call allows the user to create an external reference to a symbol which is spelled using characters not permitted in a MASM identifier. The parameter e_1 is a string which specifies the name of the external identifier. It is limited to 12 characters, left justified, space filled, from the Fieldata character set. The label in the label field is equated to a value of 0 with full value relocation by the identifier e_1.

For example, the line:

```
PLSCALL    INFO        5  'PL\SCAN'
```

allows a MASM element to reference the external symbol PL\SCAN, which is defined in another element and written in a different language, by using the label PLSCALL.

1.6.27.6. Group Number 6 (Entry Point Definition)

This call performs the same operation as group number 5, but applied to an entry point name. It allows creation of an entry point whose external name contains characters not allowed in a MASM identifier. The parameter e_1 is a string (restricted to 12 characters, left justified space filled, from the Fieldata character set) specifying the name of the identifier, and e_2 is a binary value which specifies the value of the entry point. If the identifier specified by the string e_1 is given more than one definition, only the last one is transmitted to the preamble of the relocatable output element.

For example, the line:

```
            $INFO       6  'VAL\REF',VALR
```

would make the value of VALR, an internally defined symbol, available to programs written in another language as the external symbol VAL\REF.

1.6.27.7. Group Number 7 (Even Starting Address)

This call specifies that a location counter must be given a program absolute starting address which is even. The parameter e_1 specifies the number of the location counter and must be binary without relocation.

1.6.27.8. Group Number 8 (Static Diagnostic Information)

This call specifies that a location counter is to be a part of the static diagnostic information which is a part of the absolute element diagnostic tables, rather than a part of a segment. The parameter e_1 is a binary value without relocation which specifies the number of the location counter to be used. Data generated under a group number 8 (formerly known as INFO-010) location counter are given their correct values, except that the relocation base of a group 8 location counter is always set to 0 by the Collector. This information may be referenced by diagnostic routines at execution time.

1.6.27.9. Restrictions

No location counter may be used in connection with more than one of the group numbers 2, 4, 7, and 8. At most one location counter can be a blank common block or a labeled common block with a given label.

The first six characters of an identifier used for a common block name, an external reference, or an entry point may not be zero or negative zero when converted to Fieldata.

1.6.28. $INSERT (Insert Images)

The $INSERT directive is called as follows:

$INSERT $e_1,...,e_n$

where $e_1,...,e_n$ are strings in a system character set. Each e_i is treated as a line to be interpreted by MASM, beginning with a label field as the first character of the string and so forth. The lines defined by the strings e_i are interpreted in order from left to right. $INSERT directives may be nested, with lines interpreted at inner levels being interpreted at the proper place between lines interpreted at higher levels.

Example:

```
        INSERT    'RPT JGD R4,TOP',' J EXIT'
```

This will have the same effect as the two lines:

```
RPT     JGD       R4,TOP
        J         EXIT
```

1.6.29. $LEVEL (Dictionary Level Control)

The $LEVEL directive is called by:

```
$LEVEL      e1,e2,e3
```

where e_1, e_2, and e_3 are binary values without relocation. For each subassembly (including the main assembly) there is a principal level of definition in the MASM symbol dictionary. Each new subassembly in a nest introduces a deeper level of definition in the dictionary as its principal level.

The value of e_1 on the $LEVEL directive establishes the dictionary level for the following lines (up to the next $LEVEL or the end of the subassembly) as being e_1 levels deeper than the current principal level. The value of e_2 is used to determine the dictionary insertion level for new symbols. e_2 is always positive and indicates that symbols must be defined at more shallow levels, just as if they had been written in the label field with trailing asterisks. The value of e_3 determines the level at which the dictionary search for identifiers begins. Identifiers defined deeper than this level are not found.

For the main assembly, the level more shallow than its initial principal level is that of symbols external to the assembly itself. These symbols are either the external definitions in the preamble of the relocatable element or, for definition mode assembly, the symbols retained in the dictionary snapshot written out as the omnibus element. Therefore, the line:

```
LEVEL       0,1,0
```

causes, for an ordinary assembly, all following symbols defined at the current level to be externally defined; for a definition mode assembly, all following symbols are retained in the dictionary snapshot in the output element. The use of this form eliminates the need for explicitly externalizing (with an asterisk) all of the symbols defined by the element.

1.6.30. $LIST (Resume Listing)

The $LIST directive requires no parameters. It is ignored if the subassembly pass is not generative. Otherwise, any UNLIST condition in existence (due either to an initial N option or an $UNLIST directive) is removed, and the printed listing is resumed under control of whichever listing options were specified on the MASM processor call statement.

1.6.31. $LIT (Literal Pool Definition)

The $LIT directive has two distinct forms:

```
$LIT
```

and

```
label       $LIT
```

If the current subassembly pass is not generative, the directive will be ignored. If there is no label, the implied literal pool counter number is set equal to the current location counter number. If there is a label field, then a literal function is created which places literals into the pool corresponding to the current location counter and the label is set equal to that function. At the beginning of the main assembly, the implied literal pool location counter number is zero. If there is more than one labeled $LIT directive for the same location counter, the labels are the same function; that is, literal pools are unique by location counter only, not by name.

For example, if the lines:

```
$(2),ABC  $LIT
$(4)      $LIT
$(1)      (1, TABLE)
          ABC(1,0)
```

are interpreted, then the literal (1,TABLE) is placed in the location counter 4 literal pool, while the literal ABC(1,0) is placed in the location counter 2 literal pool.

1.6.32. $NAME (Define an Internal Name)

The $NAME directive is called as follows:

label $NAME *e*

where *e*, which may be void, is converted according to the rules for parameter conversion. The label specified is given the value of an internal name, whose associated entry value is the value of *e*. An internal name may be used to provide an alternate entry point to a PROC or function, or it may provide a forward transfer point within the main assembly. The object of a $GO directive must be an internal name. For an internal name to be known outside the PROC or function containing it, it must be externalized. Nonexternalized names may only be used from within the PROC or function containing them (or deeper nested calls), and are usually employed only as $GO destinations. For $NAME directives contained within a PROC or function sample, the expression *e* is evaluated only once, at the time the sample is scanned initially. This means that the value of *e* does not change from call to call. The value of *e* is obtainable as P(0,0) or F(0), assuming that P is the relevant $PROC label or F the relevant function label.

1.6.33. $NEG (Transform Negative Values)

The $NEG directive is called as follows:

$NEG *function name*

where *function name* is an entry point to a user-defined function which is called internally by MASM to transform negative values. If the function name is void, the effect of the $NEG directive is nullified. There are two instances when the function is called:

1. when a negative value is being output to the relocatable binary element.

2. when a negative value which is part of a larger value is being built. This is the case of forms both explicit and implicit. This means the transformed value may be entered into the dictionary.

Example:

```
1: F$      $FUNC
2: TWOS*   $NAME
3: MSK     $EQU     F$(2)=72->-0D!1*/F$(2)-1
4:         $END     (F$(1)+1)**MSK
5:         $NEG     TWOS
6: AZ      $EQU     +(-2,-3)   .
7: AF      $FORM    7,13,11,5  .
8:         AF       -3,-10,4,-2
```

```
9:  AX      $EQU    -2      .
10:         +       AX      .
11:         -       10      .
```

Explanation:

Lines 1-4 defines the function MASM uses to perform the transformation of negative values from ones complement to twos complement

Line 5 indicate negative values are to be transformed by the function TWOS.

Line 6 the transformed value 0777776777775 will be associated with the symbol 'AZ' and entered into the dictionary.

Line 7 defines the symbol 'AF' as a form reference.

Line 8 uses the form reference 'AF' to generate the 36 bit value 0767775400236 or broken down into its input fields 0175 017766 0004 036.

Line 9 associates the value -2(ones complement) with the symbol 'AX' and enters it into the dictionary.

Line 10 the value associated with the symbol 'AX' is output to the relocatable binary element after being transformed to a twos complement negative value.

Line 11 the value -10 is transformed to a twos complement number and output to the RB element.

Care must be exercised when performing arithmetic operations on values which consist wholly or partially of values which have been transformed.

The $NEG directive and its associated function is in effect for all lines following it or until another $NEG directive is encountered.

MASM supplies two parameters to the user-defined function:

1. as selector 1 the value

2. as selector 2 the field size of the value in bits

The function should AND the transformed value with a value consisting of one bits whose size is equal to the field size. This is to remove any possible side effects which may have propagated from the function.

Example:

```
1:  F$      $FUNC
2:  SBMAG*          $NAME
3:          $END    (-F$(1)++1*/(F$(2)-1))**(1*/F$(2)-1)
```

Explanation:

Lines 1-3 define a function which transform ones complement negative values to sign bit magnitude negative values.

Line 3 expression complements the value specified by F$(1), sets the sign bit then ANDs the result with a value of all one bits whose size is the field length.

1.6.34. $NIL (No Action)

The $NIL directive requires no parameters and generates no code. If a label (including a waiting label) is specified, the label is marked as used, preventing definition by implication, but is not given a value.

1.6.35. $OCTAL (Set Binary Representation to Octal)

The $OCTAL directive requires no parameters. It sets the binary representation mode to octal.

1.6.36. $PROC (Define a PROC)

The $PROC directive is called as follows:

label $PROC e_1, e_2, e_3

where e_1, e_2, and e_3 are nonnegative binary values without relocation. A detailed discussion of procedures is found in 1.8. The $PROC directive serves to introduce a procedure definition. The label field, if any, may not have selectors; the label is used within the procedure to identify the parameter tree defined by the call to the procedure. If the label is externalized, it is also given the value (at the appropriate level) of an entry point to the procedure, with no entry parameter, which enters the procedure at the first statement.

The parameter e_1 specifies the maximum number of parameter lists allowed (in addition to list 0). If e_1 is void, then the number of lists is unlimited. If e_1 is flagged, the procedure is defined as being one pass.

The parameter e_2, if coded, specifies the number of words generated by the PROC. This value is computed only once, when the sample is scanned, and may not be changed from call to call. Such a PROC is called a words-given PROC.

The parameter e_3, if coded, specifies the location counter to be used for generation of the code under the PROC. If omitted or void, the location counter used is the one active at the point the PROC was called.

1.6.37. $REPEAT (Repeat a Statement Group)

The $REPEAT directive has the form:

label $REPEAT *rpt*

where *label*, if present, may not have selectors, and *rpt* is a field of zero to three binary expressions without relocation. If there is one expression in *rpt*, it is taken to be *end*; two expressions are assumed to be *start* and *end*; while three expressions are assumed to be *start*, *end*, and *step*. If *start* or *step* is omitted, the value 1 is used. If *end* is omitted, the value 262143 is used. Both *start* and *end* must be in the range 0 to 262143, while *step* must be nonzero with magnitude less than 131072.

The lines between $REPEAT and the next (unconditional) $ENDR at the same $REPEAT nesting level are saved as sample. If *step* does not have the same sign as *end–start*, the sample statements are not interpreted. Otherwise, they are interpreted $1+(end-start)/step$ times, with the label set first to *start*, then to *start* + *step*, and so on. Each iteration is terminated by encountering an $ENDR directive, either the unconditional one at the end of the sample, or one generated conditionally. If an $ENDI directive is encountered, the entire $REPEAT operation is terminated. $REPEAT introduces a new sample level but not a new dictionary level.

1.6.38. $RES (Reserve Space)

The $RES directive has the form:

$RES *e*

where *e* is a binary value without relocation. If the current location counter is blocked, this line is marked with an I-flag and no action will be taken. Otherwise, the value of *e* is added to the current location counter. If the directive appears on a source image, the original value of the location counter will be printed in the listing. Note that the expression *e* must be fully computable in the summary pass of the subassembly, since the location counter value is affected. This means that any identifiers used in computing *e* must have their values determined by previously interpreted lines.

1.6.39. $UNLIST (Inhibit Listing)

The $UNLIST directive requires no parameters. If the current subassembly pass is not generative, this directive is ignored. Otherwise, any listing being produced is inhibited and remains so until a $LIST directive is encountered. The processor call *N* option has the effect of placing an $UNLIST directive before the first line of the assembly. This is not the same as requesting no listing, since a $LIST directive may turn on the listing. To insure that there is no listing, there must be an absence of all of the listing options (C, D, E, L, O, R, or S).

1.6.40. $WRD (Specify Word Size)

The $WRD directive is called by

$WRD *e*

where *e* is a positive binary value without relocation not exceeding 36. The current word size in bits is set to the value of *e*. Internally, MASM can handle a word size up to 72 bits; however, the present Operating System interface (ROR) does not support a word size larger than 36 bits.

1.7. ASSEMBLER FUNCTIONS

MASM has a large number of functions built into it for determining the status of the assembly and the characteristics of an expression, and manipulating the data types of node references and strings.

All built-in functions in MASM begin with the "$" character, which prevents the programmer from redefining them, since a $ may not appear in column 1 of a line (except for location counter change). Some built-in functions do not require any arguments. This is indicated by writing the function name without an argument list following it (that is, "$F" is written, not "$F( )"). This is true also for those functions for which arguments are optional.

The function name itself may be computed by an expression (such as a conditional expression or a PROC parameter) which evaluates to control information. The statement:

```
+           (K->$LCB !$LCV)(1)
```

computes either $LCB(1) or $LCV(1), depending on the value of K. Similarly, the $EQU directive may be used to create new names for built-in functions, as in

```
A       $EQU    /K->$LCB !$LCV
        +       A(1)
```

Built-in functions generally expect a certain context for their arguments. If an argument is not of the proper data type, conversion is performed. If the conversion necessary is not defined, a V-flag is produced.

1.7.1. $AP(e) (Absolute Part)

The value of $AP($e$), where e is a binary expression, is the absolute part of e with all relocation information deleted. None of the other attributes of e is affected by this function.

An expression of the form "$AP($e$)=$e$" is true if and only if e has no relocation. An expression of the form:

$$e_1\text{-}\$AP(e_1)=e_2\text{-}\$AP(e_2)$$

is true if and only if the relocation for e_1 and e_2 is the same.

Examples:

If ABC is 047 relative to the base of location counter 1, then $AP(ABC) is 047. The value of $AP($LCV) is $LCV-$LCB.

1.7.2. $BA(e) (Binary Attributes)

The value of $BA($e$), where e is a binary expression, is a node reference whose elements describe the binary attributes of e. If a FORM is attached to e, then selector 0 of the node is defined, and its selectors (starting at 1) are the field sizes of the FORM. If e has m relocation items, then the selectors $1,\ldots,m$ of the value of $BA are defined, and each of them has three subselectors. The first of the three is the leftmost bit of relocation, the second is the rightmost bit of relocation, and the third is the relocation itself. The relocation is binary if relocation is by a location counter. It is a string (the external reference name) if relocation is by an external reference. If the relocation is negative, the third subselector is flagged. If e has neither a form nor any relocation, the value returned by $BA is an empty node. This is summarized in Table 1-4.

Table 1-4. Selectors Defined on the Result of $BA(e)

Selector	Description
(0,*j*)	Field size in bits of the j^{th} field of the attached FORM, if any. FORM fields are counted from the left.
(*i*,1)	Leftmost bit of relocation for the i^{th} relocation item. Bits are numbered from right to left, starting at 0.
(*i*,2)	Rightmost bit of relocation for the i^{th} relocation item.
(*i*,3)	If relocation is by a location counter, this is the number of that location counter. If relocation is by an external reference, this is a string whose characters are the name of the external symbol.
(*i*,*3)	If relocation is to be subtracted, this value is one.

Example:

If the lines:

```
ABC     EQU     +(J EOR)
Z       EQU     $BA(ABC)
```

have been interpreted, and EOR is external, then the value of Z is:

$LO($L1(6,4,4,4,2,16),$L1(15,0,'EOR')).

1.7.3. String Conversion Functions

The functions described in this subsection create strings from other data types.

1.7.3.1. $CAS(e) (Convert to ASCII String)

The expression *e* is converted to the ASCII character set If *e* is a string then the string will be converted from its original character set to its ASCII equivalent. If the expression *e* is a binary value then the value is grouped into 9-bit groups with the lower eight bits as significant bits and the resultant value is marked as an ASCII string.

Example:

```
1.              +$CAS('ABCD')
```

Assuming the system character set is Fieldata, then the string 'ABCD' is converted to ASCII and output as:

```
0101102103104
```

2. +$CAS(012040777)

The binary value 012040777 is grouped into 9-bit groups with the lower eight bits as significant bits. The resultant value is:

012040377

This example would produce a T-flag because bits were lost when only the eight least significant bits were output.

1.7.3.2. $CB (e1,e2) (Convert to Binary Representation)

The function $CB($e_1$,$e_2$) requires e_1 and e_2 to be integers, with $0 \leq e_2 \leq 26$. If e_2 is omitted, a value of zero is assumed. The result of $CB is a string in the system character set which is the representation of the value of e_1 in the binary representation mode. The significant digits are right justified within the string, and there are at least one leading zero digit. If e_1 is negative, then the string begins with "-". If possible, enough leading zeros are included so that the total length of the string is e_2; otherwise, the length of the string is the minimum size necessary to hold the nonzero digits (and leading zero) of the representation of e_1.

Example:

The value of $CB(27,6) is '000033'.

1.7.3.3. $CD(e) (Convert to Decimal)

If e is a binary value without relocation, the value of $CD($e$) is a string in the system character set which contains the decimal representation of e. If e is negative, a leading "-" is present. The length of the result is the minimum necessary to contain the significant digits of e.

Example:

The value of $CD(0144) is '100'.

1.7.3.4. $CFS(e) (Convert to Fieldata String)

This function works like $CAS, but the resulting string is in Fieldata. If the expression e is a binary value, the value is grouped into 6-bit groups, and the resultant value is marked as Fieldata.

1.7.3.5. $CS(e) (Convert to String)

This function works like $CAS and $CFS, except that the result is in the data character set, if any. If there is no data character set, the result is in the system character set.

1.7.4. $FN (e1,e2) (Form a Name)

The value of $FN($e_1$,$e_2$) is a PROC or function name. The e_1 parameter must be a previously defined procedure or function name, and e_2 is converted according to the rules for parameter conversion, except that a void expression is also allowed. If e_2 is omitted, a void expression is assumed. The result of this function is a new entry point to the PROC or function named by e_1, except that the entry parameter is the value of e_2. If e_2 is void, the new entry point acts like a PROC or function label, while a nonvoid e_2 produces a new entry point of the NAME type (the zero selector is defined).

The new entry point is assumed to be at the first of the procedure or function body.

Example:

If PVT is a PROC which has no NAME entry point, the value of $FN(PVT,6) is the same as that of the label on the line:

label * NAME 6

1.7.5. $FP (Final Pass)

The value of $FP, which requires no parameters, is 1 if the current subassembly pass is the final pass of the current subassembly and is zero otherwise. This function should be used to control actions which are to be performed only once during a subassembly, even though the subassembly may require more than one pass.

1.7.6. $GP (Generative Pass)

The value of $GP, which requires no parameters, is 1 if the current subassembly pass is generative and is zero otherwise. This function should be used to control actions associated with the output such as listing control and printed displays.

1.7.7. $IBITS(e) (Indicator Bits for Expression)

The expression *e* is converted according to the rules for parameter conversion. Table 1-5 indicates the bits set in the result for various characteristics of *e*.

Table 1-5. Expression Characteristic Indicator Bits

Bit	Meaning
0*	Flagged expression
1	Double precision
2	Negative arithmetic value
3	Left justification
4	Form attached
5	Fieldata string
6	ASCII string

*least significant bit

For example, the value of $IBITS(*'ABC'LD) is 0113 if the current character representation mode is ASCII.

1.7.8. $IC(e) (Identifier Class)

The $IC function takes one argument, which must be a binary value without relocation in the range 0 to 127. It returns a portion of the MASM symbol dictionary containing all those identifiers whose MASM system hash code is *e*. The value of $IC(*e*) is a new node reference. The selector *m* is defined for this node if there are class *e* identifiers at level *m*. Levels refer to the hierarchy of definitions established by the nest of subassemblies active at the time the $IC function is invoked. For each selector *m*, the value of the selection is a reference to a node, which is distinct from all previously allocated nodes. The selectors defined for this node are consecutive integers starting from 1 which select strings in the system character set corresponding to the identifiers defined at level *m*. If the identifier defines a value or node reference, then the string is not flagged. If the identifier defines control information, the string is flagged.

The following function computes the class number of an identifier (that is, it computes the system hash value of an identifier) supplied as a string:

```
F         function
CLASS*    NAME
          CHAR
          FDATA
          WRD       36
S         EQU       $CFS(F(1))
          END   127**('RANDOM'*($SS(S,1,6)--$SS(S,7,6)))*/-29
```

Level 0 in the dictionary is the level of symbols defined outside the main assembly (external symbols for generative assemblies, saved symbols for DEF-mode assemblies). Level 1 is the level of the main assembly itself. Higher numbered levels are those of progressively deeper nested subassemblies (PROCs and functions).

Example:

$IC(14)(1,5) is a string which is the name of the fifth identifier in class 14 defined at the level of the main assembly. This string can be used in a microstring expression to retrieve the identifier itself, so a symbol table can be constructed for use at execution time.

1.7.9. $ILCN (Initial Location Counter Number)

The function $ILCN requires no parameters and returns the location counter number in effect at the beginning of the current subassembly pass. For the main assembly, the value is always zero. If the current subassembly is a call to a procedure with a specified location counter number, then $ILCN is the value of that location counter number. For other subassemblies, $ILCN is the value of $LCN at the point at which the subassembly was invoked.

Examples:

Inside a PROC started by the line:

```
P         PROC      ,,5
```

the value of $ILCN is 5. Inside a PROC started by the line:

```
P         PROC
```

called from a point where the active location counter is 3, the value of $ILCN is 3.

1.7.10. $LCB(e) (Location Counter Base)

The $LCB function requires one parameter or no parameters. The value of *e* should be in the range 0 to 63, if given. If *e* is not within this range, a T-flag is produced. If no parameter is given, the value of $LCN is used. The value returned by $LCB is 0 relocated by the location counter specified by the value of *e*. In other words, the value of $LCB is the address of the first word of location counter *e*.

1.7.11. $LCN (Current Location Counter Number)

The $LCN function requires no parameters. Its value is the number of the current location counter.

1.7.12. $LCV(e) (Location Counter Value)

This function returns the current value of the location counter designated by the value of *e*. If *e* is omitted, the value of $LCN is used for *e*. This function has the "$" function as a synonym for compatibility with existing programs and for shorthand convenience.

1.7.13. $LEV (Principal Dictionary Level)

The $LEV function requires no parameters. Its value is the number of the principal dictionary level. The value of the principal dictionary level is 1 on the main assembly and is incremented by 1 for each nested subassembly.

Example:

```
1.  P*      PROC
2.          +$LEV
3.          END
4.          +$LEV
5.          P
6.          END
```

Explanation:

Line 4:

The value of $LEV is 1.

Line 5:

The value of $LEV at line 2 is 2 when procedure P is called.

1.7.14. $LF(e) (Label Field Description)

The value of $LF is a description of the waiting label for subassembly *e*, where *e* is a positive integer. For purposes of this function, the active subassemblies are assumed to be numbered from zero starting with the current subassembly. A T-flag is produced if the subassembly does not exist. A V-flag is produced if the subassembly is protected. The value of $LF is a new node reference. If the e^{th} subassembly has no waiting label, the node has no selectors defined. If there is a waiting label, the zero selector points to a string in the system character set which represents the identifier. If the waiting label is a selector definition (i.e., a subscripted label), then selector *i* is the binary value

of the i^{th} subscript. Note that $LF(0) is always illegal, since the current subassembly is the line containing the $LF call and is always protected. The argument of 1 for $LF retrieves the waiting label for the PROC containing the line on which $LF is called.

For example:

The value of $LF($e$)=0 is 1 if there is no waiting label. The value of $LF($e$)=1 is 1 if the waiting label is a simple identifier. The value of $LF($e$)>1 is 1 if the waiting label is a selector definition.

Example 1:

If the current subassembly has a waiting label of CLB(4,3), then the value of $LF(1) is $LO('CLB',4,3).

Example 2:

The following function converts the output of $LF to a string corresponding to the label:

```
F       function
CLF*    NAME
        IF      F(1)>1
A       EQU     '('$CD(F(1,1))
I       DO      2,F(1)-1 ,A  EQU  A:',':$CD(F(1,I))
A       EQU     A:')'
        ELSE
A       EQU     ''                      . THE NULL STRING
        ENDF
        END     F(1,0):A
```

This function may then be used, in a PROC:

```
INCR*   PROC    *0
*       EQU     [CLF($LF(1))]+1
        END
```

which increments the label in the label field of the PROC call line, as in:

```
K       INCR
```

which has the same effect as the line:

```
K       EQU     K+1
```

1.7.15. $LINES (Line Counter)

The function $LINES requires no parameters. It returns a count of the number of lines scanned by MASM since the beginning of the assembly. The lines counted include those from source input, library PROCs, PROC and function interpretation, sample scanning, $REPEAT and $DO repetitions, $INSERT images, and images skipped by $GO and $IF. Continuation images are considered part of the initial line and are not counted separately. The final value of the line counter is printed at the end of an assembly, if the appropriate type of listing is requested.

The line counter provides a simple measure of the cost of the assembly process, which is adequate for most purposes of optimization. The following lines illustrate how this function may be used to

measure the cost of a given section of code:

```
K       EQU     $LINES
        .
        .   code being measured
        .
        DISPLAY $CD($LINES-K-1)):' LINES'
```

1.7.16. $LP (Last Pass)

The $LP function requires no parameters. Its value is 1 if the generative pass of the main assembly is being performed and zero otherwise. This function is used to control actions which are performed only after the first (summary) pass of the main assembly is complete.

1.7.17. $L0 (e0,...,en) (Form a List Starting at 0)

The parameters $e_0,...,e_n$ are converted according to the rules for parameter conversion. A new node is constructed whose $n+1$ selectors run from 0 to n and whose i^{th} selector value is e_i. This function may be used to construct complex tree structures such as those found in languages such as LISP, SNOBOL, or PL/I.

For example, if:

```
A       EQU     $L0('X',6,$L0(014))
```

then A(0) is 'X', A(1) is 6, and A(2) is a node with A(2,0) having the value 014. No other selectors of A are defined.

1.7.18. $L1 (e1,...,en) (Form a List Starting at 1)

The $L1 function performs an operation identical to that of $L0, except that the first defined selector of the result is 1 rather than 0.

1.7.19. $NODE (Form a Node)

The $NODE function requires no parameters. It returns a reference to a node which is distinct from every other node so far created. The node thus produced has no selections defined. This may be required if the identifier is assigned a value other than a node and its content expected a node value.

For example, if the line:

```
A       EQU     15
```

has been interpreted by MASM, then the use of A with a subscript is illegal, as in A(3). Therefore, the programmer must write:

```
A       EQU     $NODE
A(3)    EQU     ...
```

in order to achieve the desired results.

1.7.20. $NS (e1,e2) (Find nth Selector)

A nonempty node may have any set of selector numbers in use; they need not be successive integers, nor need they start at any particular integer. The $NS function requires e_1 to be a node reference and e_2 to be an ordinal integer less than or equal to the number of selectors defined for e_1. Then the value of $NS($e_1$,$e_2$) is the value of the e_2^{th} selector of e_1, giving the selectors of e_1 the usual numerical ordering, so that the smallest selector of e_1 is one, etc. An error results if e_2 is larger than the number of selectors defined for e_1. (The first selector is found when $e_2 = 1$.)

Since the selectors for a node are ordered by increasing value, the selection mechanism may be used for sorting. For example, if each sort key K has an associated value VK, then one may initialize

```
A           EQU         $NODE
```

and then perform for each key K:

```
A(K)        EQU         VK
```

The sorted values may be retrieved in order by referencing the selector numbers $NS(A,1), $NS(A,2), and so on.

Example:

If A(2) and A(4) are the only selectors defined for A, then A($NS(A,1)) is the same as A(2), and A($NS(A,2)) is the same as A(4).

1.7.21. $PAR(e) (Processor Call Parameter)

The $PAR function provides access to the parameters on the MASM processor call statement. The argument e must be a binary value without relocation in the range 0 to 63. If e is zero, the value of $PAR is the option bits from the MASM call in master bit notation (bit 0 corresponds to Z, bit 1 to Y, and so on). For $e > 0$, the function returns the element name subfield of field e of the MASM processor call statement. If no such field exists, a void string is returned. This function can make assembly actions depend on parameters specified from outside the MASM environment.

1.7.22. $SL(e) (String Length)

The parameter e must be a string. The value returned is the number of characters in the string e.

Example:

The value of $SL('SIX+ONE') is 7.

1.7.23. $SN (e1,e2) (Find Selector Number)

This function is the converse of the $NS function. Where $NS uses an ordinal number to return the defined selector for a node, $SN uses a defined selector e_2 for the node e_1 and returns its appropriate ordinal. Therefore, e_1 must be a node reference and e_2 should be a binary value without relocation. If the selector e_2 is defined for e_1, the value of $SN($e_1$,$e_2$) is the ordinal number of that selector. Otherwise, the value is 0.

The expression \\$SN(e_1,e_2) is 1 if e_2 is a selector for e_1 and zero otherwise.

Example:

If A(2) and A(4) are the only selectors defined for A, then $SN(A,2) is 1 and $SN(A,4) is 2.

1.7.24. $SR (e1,e2) (String Repetition)

The function $SR returns a string which is constructed by concatenating e_2 copies of the string e_1. Thus, e_2 should be a nonnegative binary value without relocation. If e_2 is zero, a void string is returned.

Example:

The value of $SR('ABC',3) is 'ABCABCABC'.

1.7.25. $SS (e1,e2,e3) (Substring Extraction)

For this function, e_1 is a string and e_2 and e_3 are integers with $e_2 \geq 1$ and $e_3 \geq 0$. If e_3 is omitted, 1 is used. $SS returns the substring of e_1 starting at character e_2 (numbered from the left beginning with 1) of length e_3. If e_3 requests more characters than are present to the end of string e_1, the result is blank filled to e_3 characters. If e_3 is zero, the result is a void string.

This function may be used to left justify strings within a given field size.

Example:

```
1.      $SS('A',1,10):$SS('EQU',1,10):'1'
2.      $SS('Example',3,3)
```

Explanation:

Line 1:

The result has "A", "EQU", and "1" beginning at character positions 1, 11, and 21, respectively.

Line 2:

The result is the string "amp". This example assumes ASCII.

1.7.26. $SSS (e1,e2,e3,e4) (Substring Substitution)

For this function, e_1 and e_2 are strings, while e_3 and e_4 are integers, with $e_3 \geq 1$ and $e_4 \geq 0$. If e_4 is omitted a value of 1 is assumed. The value of $SSS is a string constructed by substituting the string e_2 for the e_4 characters of string e_1 beginning at character e_3 of e_1. The other portions of the result are the rest of string e_1. The string e_1 is extended by blanks on the right if necessary to make the expression meaningful. If e_4 is zero, the insertion is done before character e_3. If e_2 is void, this function deletes a substring of e_1. The type of resultant string is determined by the following rules in decreasing precedence:

1. If e_1 or e_2 is data character set, the result is the data character set.

2. If e_1 or e_2 is ASCII, the result is ASCII.

3. If e_1 or e_2 is Fieldata, the result is Fieldata.

The result of the $SSS function can usually be computed alternatively using the $SS function and concatenation operators, but $SSS provides greater clarity of intent. This function, when combined with the $DISPLAY directive, may be used to construct commentary display, at assembly time.

Example:

The value of $SSS('ABCDEF','HIJ',3,2) is 'ABHIJEF'.

1.7.27. Typing Functions

MASM has a large number of functions which allow the programmer to interrogate the data type of an expression.

1.7.27.1. $TYPE(e) (Compute Data Type Number)

The expression *e* is converted according to the rules for parameter conversion. The value of $TYPE is an integer corresponding to the type of the expression *e* as given by Table 1-6.

Table 1-6. Data Type Numbers

Number	Type
1	Binary value.
2	Floating point value.
3	String value.
4	Node reference.
5	Internal name (NAME line label).
6	PROC name (label on PROC line).
7	function name (label on function line).
8	MASM directive (including instruction mnemonics).
9	MASM built-in function.

1.7.27.2. Type Testing Functions

All the functions described in this subsection require one parameter which is converted according to the rules of parameter conversion. They all return either a 0 or 1, depending on the similarity between the type of the parameter and the function used.

Table 1-7 indicates for what types the various type testing functions return a 1.

Table 1-7. Description of Type Testing Functions

Function Name	Data Type For Which the Value is 1	Description
$TBIN	1	Test for binary
$TCON	5,6,7,8,9	Test for control information
$TDAT	1,2,3,4	Test for data
$TDIR	8	Test for directive
$TFLT	2	Test for floating
$TFNM	7	Test for a function name
$TFUN	9	Test for a built-in function
$TINM	5	Test for an internal name
$TNAM	5,6,7	Test for a name
$TNOD	4	Test for a node
$TPNM	6	Test for a PROC name
$TSTR	3	Test for a string
$TVAL	1,2,3	Test for a value

1.7.28. $TMODES (Test Modes)

This function requires no parameters. It returns a binary value with bits set according to Table 1-8.

Table 1-8. Mode Bit Settings for $TMODES

Bit	Setting	Meaning
0*	1	$ASCII directive in effect
	0	$FDATA directive in effect
1	1	$LIST directive in effect
	0	$UNLIST directive in effect
2	1	$OCTAL directive in effect
	0	$HEX directive in effect

*least significant bit

1.7.29. $(e) (Location Counter Value)

This function is the same as $LCV when used as an expression element. When written in the label field of a line (the argument is mandatory), this function indicates a change in the number of the active location counter to the value of *e*. The following data are then generated under location counter *e*. This is the only built-in function which may be written in the label field of a line; therefore, built-in functions may not be redefined.

1.8. PROCEDURES

Procedures are one means of invoking separate subassemblies within the main assembly. Using these subassemblies the user may:

1. extend the set of directives and instructions mnemonics provided by MASM,

2. build data structures and

3. generate sequences of coding or data.

PROCs may make use of the full capabilities of MASM, subject only to the restrictions given below on each type of PROC and restrictions imposed by the context of the call (as in a line item).

A procedure is bounded by a $PROC–$END pair of directives. The lines between the $PROC and $END are called the procedure sample, and are interpreted by MASM when the procedure is invoked. The $END terminating the body of the procedure must be unconditional (that is, not the object of a $DO, within an $IF–$ENDF pair, created by $INSERT or micro substitution, and so forth).

When supplying expressions on the $PROC directive, it is important to recognize that a void subfield may be generated by a conditional expression and has a meaning different from the meaning of a subfield whose value is zero. This is true for all three subfields. Since the $PROC directive is interpreted by MASM when encountered and not saved as part of the PROC body, the expressions in the $PROC operand field are evaluated only once, when the PROC sample is saved. The label on the $PROC directive line, however, is defined at the level of the body of the PROC and must be externalized if it is to provide an entry point to the PROC. The same is true for the label on any NAME lines inside the PROC; NAME lines also have their expression evaluated only once, at the time PROC sample is saved. (If the body of the PROC is to be computed by actions taken at the time sample is saved, this can be done by using levelers, as described in 1.11.) A PROC sample is terminated by an unconditional, nongenerated $END directive. The interpretation of a PROC is terminated by encountering any $END directive, either one conditionally generated, or the one at the end of the PROC body.

Generally, procedures are used to generate sequences of coding or data which vary based on some set of parameters, which may be explicitly given to the procedure on the PROC call line or implicitly determined from values defined at higher levels. Procedures may generate any number of words of data or instructions, including zero; however, they must be consistent in the incrementation of location counters from pass to pass of the higher level subassemblies. If this restriction is ignored, the output is likely to contain C and D flags, which indicate the values of the location counters were different in different passes.

Procedures must be defined before they are called. This can be done in three ways. The PROC sample may be included in the source language given to MASM. The PROC sample may be contained in the dictionary information saved by a definition mode assembly and loaded by the $INCLUDE directive, or the PROC sample may be in a file in an assembler PROC element processed by the Procedure Definition Processor (PDP).. The rules for PROC name lookup in files have been discussed in 1.2.5.

1.8.1. Types of PROCs

There are three types of PROCs possible in MASM, depending on whether the first subfield is flagged or whether the second subfield is nonvoid. A nonvoid second subfield overrides a flagged first subfield. The three types are known as two-pass (no flag, void second subfield), one-pass (flagged first subfield, void second subfield), and words-given (second subfield nonvoid). Their characteristics are summarized briefly in Table 1-9, where the passes referred to in the heading are the passes made by the next higher subassembly:

Table 1-9. Characteristics of PROC Types

Type of PROC	Action Taken on Summary Pass	Action Taken on Generative Pass	Restrictions
two-pass	summary pass	summary and generative passes	none
one-pass	summary pass	generative pass	no forward references
words-given	increment location counter by number of words given	generative pass	No ext defs no forward references, generate number of words specified

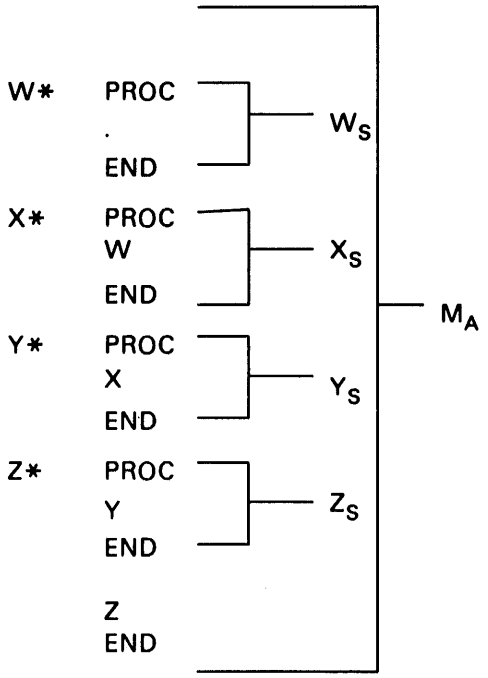
Note that the number of passes performed on a PROC depend on the kind of pass being performed by the next outer subassembly and on the type of PROC being used. This leads to considerable differences in efficiency among the types of PROCs.

On the PROC call line itself, if the number of fields actually specified is less than the number permitted by the first $PROC directive subfield value, the PROC call must be terminated by a period-space to avoid scanning a comment as possible parameters for the PROC.

1.8.1.1. Two-Pass PROCs

Two-pass PROCs are not restricted in any way. All operations permitted in the main assembly are permitted in a two-pass PROC, with the addition of the ability to use the $GO directive to transfer backward or to another PROC, as well as forward. (In the main assembly, a $GO may only go forward, not backward or into a PROC.) The number of words generated by distinct calls on a two-pass PROC need not be the same. Forward references to labels local to the PROC may be employed, and variables external to the PROC may be manipulated at will. The flexibility achieved may require substantial processing by MASM. Since a two-pass PROC requires two passes during the higher level generative pass as well as the summary pass in the higher level summary pass, the number of passes made by the innermost PROC in a nest of calls to two-level PROCs is an exponential function of the depth of nesting. This can be extremely expensive. Consequently, two-pass PROCs should be converted to one-pass or words-given PROCs where possible.

The following example is a series of nested two pass procedures. Table 1-10 indicates the number of passes performed and the values of the built-in functions.



W_S, X_S, Y_S, and Z_S are procedure subassemblies. M_A is the main assembly.

Table 1-10. Two Pass Summary Table

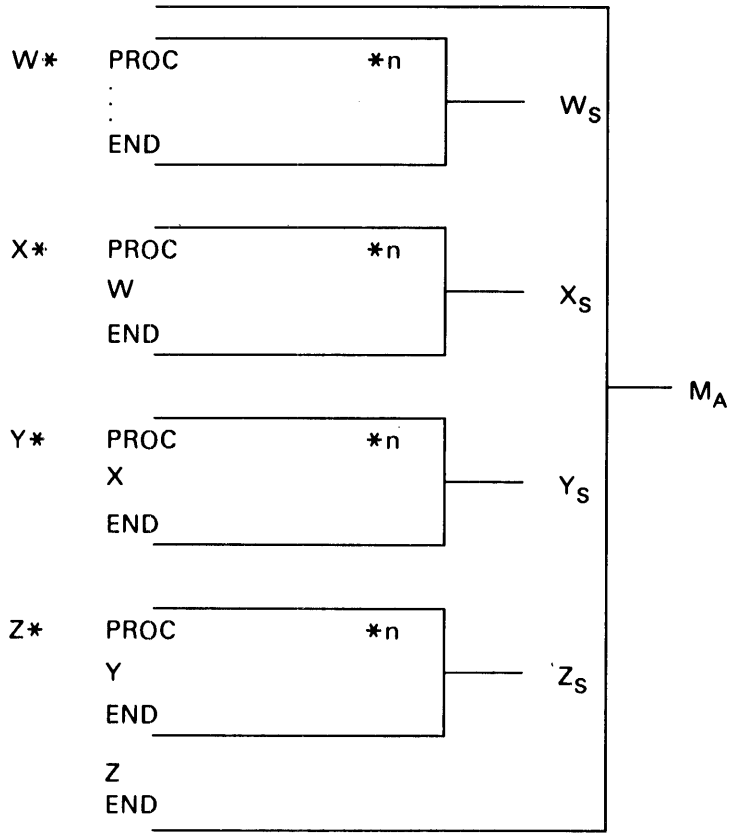
M	Z	Y	X	W	$FP	$LP	$GP
S	S	S	S	S	1	0	0
G	S	S	S	S	1	1	0
	G	S	S	S	1	1	0
		G	S	S	1	1	0
			G	S	0	1	0
				G	1	1	1

S and **G** indicate a summary and generative pass, respectively, being performed. The values associated with the built-in functions $FP, $LP, and $GP are as if the functions were in procedure **W**.

1.8.1.2. One-Pass PROCs

One-pass PROCs, as their name implies, require only one pass during the generative pass of the next higher level subassembly. Since the omitted pass is a summary pass, the definitions of labels local to the subassembly are not available until after they occur. This is why forward references are not allowed. By eliminating one summary pass, the number of passes made for the innermost nested PROC call does not grow exponentially with the nesting depth. One-pass PROCs also avoid the problem of double alteration of external variables without the need for the $FP function.

The following example is a series of nested one pass procedures. Table 1-11 indicates the number of passes performed and the value of the built-in function.



W_S, X_S, Y_S, and Z_S are procedure subassemblies. M_A is the main assembly.

Table 1-11. One Pass Summary Table

M	Z	Y	X	W	$FP	$LP	$GP
S	S	S	S	S	1	0	0
G	G	G	G	G	1	1	1

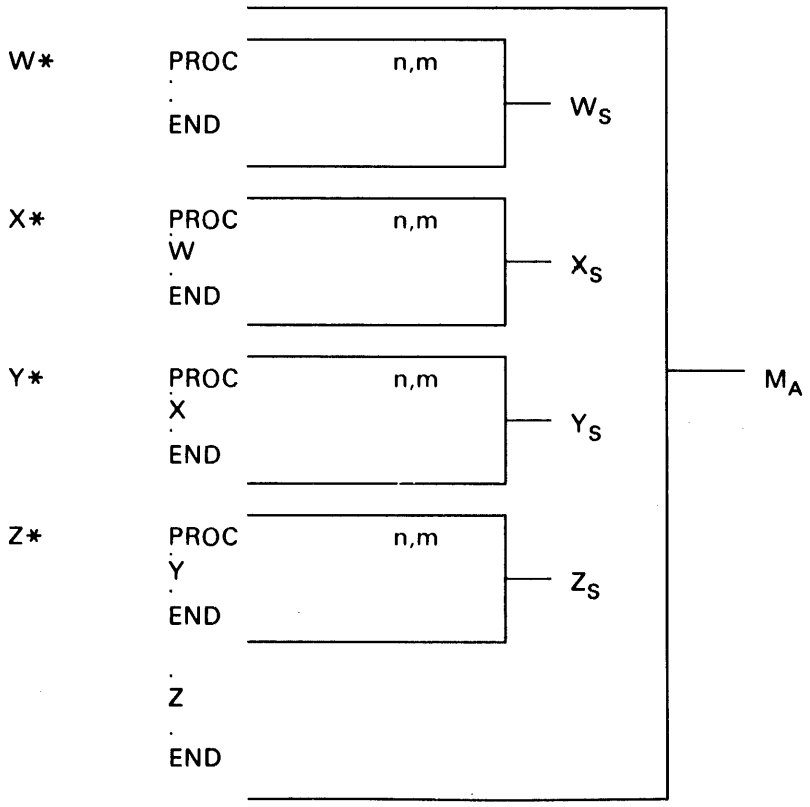
S and G indicate a summary and generative pass, respectively, is being performed. The values associated with the functions $FP, $LP, and $GP are as if the functions were in procedure W.

1.8.1.3. Words-Given PROCs

As their name indicates, words-given PROCs must generate the same number of words on all calls. The number of words must of course be the value computed by the expression in subfield 2 of the $PROC directive. Words-given PROCs need not be scanned at all during the summary pass of the next higher subassembly, since the primary reason for such a scan is to compute the number of words generated by the PROC call, which is already known. As for one-pass PROCs, no summary pass is made for a words-given PROC during the generative pass of the next higher subassembly, so forward references are not permitted in a words-given PROC.

Because the body of a words-given PROC is not scanned during the summary pass of the next higher subassembly, there are some additional restrictions imposed because their violation would result in incorrect values for location counter sizes, dictionary values, and so forth. Therefore a words-given PROC may not have a change of location counter, any externalized definitions (other than entries to the PROC), or a definition of a waiting label. If all of these restrictions can be met, the words-given PROC type should be used, as it is the form of PROC which is least expensive in terms of assembly time.

The following example is a series of nested words-given procedures. Table 1-12 indicates the number of passes performed and the values of the built-in functions.



W_S, X_S, Y_S, and Z_S are procedure subassemblies and M_A is the main assembly.

Table 1-12. Word Given Procedure Summary Table

M	Z	Y	X	W	$FP	$LP	$GP
S | G	G	G	G	G	1	1	1

S and G indicate a summary and generative pass, respectively, is being performed. The values associated with the functions $FP, $LP, and $GP are as if they are in procedure W.

1.8.2. Speeding Up a Two-Pass PROC

If a two-pass PROC can determine how many words it generates during the first summary pass, it is possible to do only the global operations and skip the interpretation of the generative directives during the first summary pass, instead performing a RES of the proper number of words. This can provide a considerable increase in speed. Similar techniques may be applicable for a one-pass PROC; the method requires an understanding of the $FP, $GP, and $LP built-in functions.

Example:

```
1.  P1*     PROC
2.          $IF       (/$LP)**$FP
3.          $RES      P1(1,1)
4.          $ELSF     $LP**$FP
5.  I       $DO       P1(1,1),;
6.          +         P1(1,1,I)
7.          $ENDF
8.          $END
9.  B       $EQU      $L1(1,2,3,4,5)
10. C       $EQU      $L1(10,12,14)
11.         P1        B
12.         P1        C
13.         $END
```

Explanation:

Procedure P1 consists of lines 1 through 8. Line 9 defines a node B with five selectors which reference the values 1, 2, 3, 4, and 5. Line 10 defines a node C with 3 selectors which reference the values 10, 12, and 14. Line 11 is a call to procedure P1. When both the main assembly and the subassembly are in the summary pass, the expression (/$LP)**$FP from line 2 will be true, causing line 3 to be interpreted. As a result of line 3, the location counter will be incremented by 5. Line 4 will cause all images up to line 7 to be skipped. Line 8 indicates the end of the procedure.

When the main assembly is in the generative pass, and the procedure is in the summary pass, the expressions (/$LP)**$FP in line 2 and $LP**$FP in line 4 are both false. Therefore, statements 3, 5, and 6 are skipped.

When both the main assembly and the procedure are in the generative pass, the expression at line 2 is false. The statement at line 3 is skipped, and the expression in line 4 is true. Therefore, the statements at lines 5 and 6 are interpreted and the result is output.

Line 12 is another call to procedure P1. The same process is repeated with the appropriate location counter increment and the number and value of the words output.

1.8.3. Calling a PROC

A PROC is called by writing the name of an entry point to the PROC in the operation field of a MASM line or line item. PROC entry points are created by externalization of labels on $PROC directives and $NAME directives, or as the value returned by a call on the $FN built-in function. The operand fields (and possibly further subfields of the operation field) are passed to the PROC as parameters by creating a new node whose selections are defined as follows:

(0,0) PROC entry parameter (value of expression on $NAME line if entry made by $NAME label, or value specified by $FN built-in function). Not defined if entry made via $PROC label.

(0,*j*) If (0,0) is a defined selection, this is the value of the expression in the j^{th} subfield of the operation field, numbering the PROC entry name itself as 0.

(*i*,*j*) This is the value of the expression in the j^{th} subfield of the i^{th} operand field. Operand fields and subfields are numbered consecutively beginning with 1.

(*i*,**j*) This is 1 if the expression in the j^{th} subfield of the i^{th} operand field was flagged, and zero otherwise. The value of *i* may be zero, if (0,0) is a defined selector, and this will retrieve the flag attribute of the $NAME line expression if j = 0.

The node for which these selections are defined is given as the local definition of the label which appeared on the $PROC directive line. Thus, if the PROC began with the line:

```
P       PROC    ...
```

the value of the third subfield of the first operand field is obtainable as P(1,3), while a NAME entry parameter is retrieved by P(0,0).

The parameters in the operand (and operation) subfields of a PROC call are converted according, for instance, to the rules for parameter conversion. Control information is possible, thus allowing P(1,1) to appear in the operation field of a line within the PROC.

1.8.4. Waiting Labels

When a label is written in the label field of a PROC call, a definition is not established for the label immediately (except for a words-given PROC, which is not considered further in this subsection). Instead, a summary pass is made over the PROC body. If a line is encountered during the summary pass with an asterisk in column 1, the label for the PROC call line is defined with the value it should be given if it appeared in the label field of the line with the asterisk in column 1 (but at its proper dictionary level external to the PROC, of course). If no such line is found, the label on the PROC call line is given the default value of the address of the first word generated by the PROC, or if the PROC generates no words, the current location counter value at the start of the PROC subassembly. Within the PROC, such a label is said to be a waiting label, and it may be examined with the $LF built-in function. Note also that the $NIL directive may be employed to keep a waiting label from being given the default definition without giving it any other definition instead.

1.8.5. Location Counter Control in PROCs

Any PROC may specify an initial location counter to be used for its generated data by coding the third field of the $PROC directive. If that field is void, the location counter in effect at the time of the PROC call is used as the initial location counter. The number of the initial location counter may be retrieved by the $ILCN function. The location counter in use is reset to the initial location counter at the start of each pass, so, a two-pass PROC need not take special action if a permanent location counter change is coded within the PROC. After completing all passes of a PROC, the location counter in use is reset to the one in use at the time of the PROC call, if it is different from the initial location counter and no permanent location counter change is made. As noted previously, a words-given PROC may not change the location counter in use, other than by specification on the $PROC directive line. Care should be exercised when changing location counters in PROCs, as unintended effects may be propagated through higher assembly levels.

1.8.6. Nesting of PROCs

PROCs may be nested statically and dynamically. Static nesting of PROCs occurs if the lines composing the body of one PROC are physically included in the body of another outer PROC. Dynamic nesting occurs when one PROC calls on another. A PROC which is statically nested can only be referenced from the PROC containing it, initially, although externalization of an entry point to an inner PROC can be performed by deliberate action of an outer PROC. The PROC sample for an inner PROC is saved when the outer PROC is called, and discarded when the outer PROC is terminated, unless some reference to the inner PROC sample is still in existence. This is an expensive operation and should be avoided for the sake of efficiency.

Dynamic nesting of PROCs (and functions) is the way new levels of the dictionary arise. Each new PROC (or function) call begins a subassembly and creates a new principal level of the dictionary for symbol lookup and insertion. PROCs written at the same static physical level may nevertheless be nested at greatly differing dynamic levels. For each subassembly, the symbols at more shallow levels in the dictionary are available for definition and reference, while a local level of the dictionary exists for symbols whose existence is limited to the current subassembly. An external symbol may be referenced as an operand at will if there is no local label with the same name; this includes the parameter node of higher level PROCs if all the PROCs have distinct $PROC directive line labels . An external symbol may be used in the label field of a line only by affixing the proper number of asterisks to it (and before any selector group). Each asterisk indicates one level of externalization. Externalization may also be used to create new external symbols at a higher level in the same fashion. The number of asterisks used must be the same for each label field reference to insure that the same variable is being obtained. Waiting labels may be externalized also; one of the asterisks is ignored in counting levels of externalization, since the first asterisk indicates that waiting label definition is intended, and level counting begins at the level at which the waiting label exists, which is already an external level.

1.8.7. Use of $NAME and $GO Directives

In addition to providing an entry point for a PROC, the $NAME directive may also create a target for the $GO directive. In the main assembly, a $GO can only transfer control forward to a $NAME label not yet encountered. In a PROC, $GO may transfer control forward, backward, or outside the PROC altogether. The label on a $NAME line need not be externalized if it is only used as a $GO target from within the same PROC, since a forward $GO to an unknown name searches to the end of the PROC while a backward $GO is to a name already passed and therefore known. (Exception: If the name is defined by a $NAME line which appears earlier in the PROC body than the point at which the PROC is entered, a backwards $GO to a nonexternalized name is unsuccessful. This situation should be rather rare.) If a $GO transfers outside the PROC to another PROC, a diagnostic G-flag is generated, since there is a possibility that this is a mistake. Such a $GO is considered to be a lateral

transfer and does not introduce a new level of definition for the dictionary; the environment in the destination PROC, including the parameter tree, is the same as before the $GO was interpreted. Thus a lateral transfer can make use of common code without a PROC call and thereby save the time required to create and later delete a new dictionary level.

A $NAME line label, if external, may be used as an entry to the PROC containing it. Only if entry is made via a $NAME label (a so-called internal name) can the zero parameter list be referenced. A set of internal names may also be used to allow one PROC to perform a number of distinct but related actions through the different locations of its entry points. Interpretation of a PROC body begins with the line following the entry point.

1.8.8. Using the $GP, $FP, and $LP Functions

The $GP, $FP, and $LP built-in functions are usually of value only within a PROC. They allow for conditional interpretation of code inside a PROC and can aid in speeding up the operation of complex PROCs.

The $GP built-in function is used to suppress calculations which lead only to values for DISPLAY or data generation until the generative pass is being performed. Computations which are essential to determining the value of an external symbol or the amount of incrementation of a location counter cannot be postponed to a generative pass, since these determinations are the reason for performing a summary pass.

The $FP built-in function is used most often to ensure that certain computations are not performed twice. It is meaningful only in two-pass PROCs, since no other PROC performs more than one pass per higher level subassembly pass. If an external variable is being incremented in a two-pass PROC, it is incremented on both passes unless protected by $FP.

The $LP built-in function is of use for both one-pass and two-pass PROCs, but not for words-given PROCs. It can be used to do RES operations during the main assembly summary pass and data generation based on later defined symbols during the main assembly generative pass, assuming that the number of words generated on the second pass is the same as the incrementation of the location counter on the first pass.

The applicability of these functions is summarized in Table 1-13.

Table 1-13. PROC Types Using Pass-Determination Functions

Type	Description
two-pass	$LP, $FP, $GP
one-pass	$LP, $GP
words-given	none (only one pass made)

1.8.9. Pass Initialization

MASM initializes to Fieldata at the start of each pass. On subassemblies, MASM will initialize to the mode when the procedure was called at each pass through the procedure.

1.9. FUNCTIONS

Note about PDP.

Functions provide the programmer with a means for extending the set of functions provided by MASM as built-in functions with new user-created functions. Functions may use the full capabilities of MASM, but are generally restricted in the generation of data because the current location counter is blocked.

A function is delimited by a $FUNC–$END pair of directives, where the $END terminating the text of the $FUNC must be unconditional (that is, not the object of a $DO, within a $IF–$ENDF pair, created by $INSERT, or by other conditional construction). The lines between the $FUNC and $END are called the function sample, and are interpreted by MASM when the function is invoked. Interpretation of the function is terminated by the first $END image which is encountered whether or not this $END was conditionally generated. The value returned by the function is given by an expression in the first operand field of the $END directive which terminates the function interpretation.

A label is generally present on the $FUNC directive line. When interpreting the function sample, this label is given the value (local to the function subassembly) of a node reference with either n or $n+1$ selectors defined. The value of selector 1 is that of argument 1 of the function call, selector 2 is given the value of argument 2, and so on up to argument n for selector n. If the function was entered via a NAME line, the expression on the NAME line (which was evaluated when the function sample was first scanned) is used as the value of selector 0 of the function name node reference. Outside the function, the value of the label on the $FUNC directive line will be known only if it was externalized. The label is then given the value of an entry point to the function with no entry parameter. This value is control information, so most uses of the value in expressions where a function call is NOT intended will require the use of the "/" operator. Similarly, any NAME line used to provide an entry to the function must have its label externalized, and such a label will be control information. NAME line labels may be used as $GO objects within the function. The$NAME similarity between the function parameter and entry mechanism and that for PROCs should be evident. (Note that the PROC parameter tree has two levels of selection, however.)

Functions may call other Functions and invoke procedures. Any MASM operation is permitted within a function or in any function or PROC called from a function with the exception of the incrementation of a blocked location counter. Since a procedure may generate code under an unblocked location counter and then call a function, more than one location counter may be blocked at a given time.

Functions, like procedures, are subassemblies. They introduce a new dictionary level for local definitions, and the lookup and definition of local and external symbols is the same as it is for PROCs (except that a function may not have a waiting label). On the other hand, a function is never processed twice by the same higher level subassembly. Since functions normally do not generate data, a generative pass is not needed. Therefore, forward references cannot occur. However, a function may be computed on either the summary or generative pass of the next higher subassembly.

Example 1:

This function calculates the character position in the first argument of the first substring which is equal to the second argument. Both arguments are assumed to be strings.

```
1.  F          function
2.  INDEX*     NAME       0
3.  I          REPEAT     $SL(F(1) )-$SL(F(2) )+1
4.             IF         F(2)=$SS(F(1),I,$SL(F(2) ) )
5.             ENDI
6.             ENDF
7.             ENDR
8.             END        I
```

Note that I is incremented until the ENDI directive is interpreted. The value of I is available outside the REPEAT group, and it is returned by the function as its value. If the substring is not found, this function will still return a value for an index, even though it is not correct. Thus the value of INDEX('ABCABCABC','BCA') is 2, but also the value of INDEX('ABCABCABC','E') is 9. Correcting this function to return 0 in the no find case is left as an exercise for the reader.

Example 2:

A function which converts a character string into a node reference whose selector *i* is a single character string containing the *i*th character of the argument string may be written:

```
1.  F          function
2.  STRNOD*    NAME       0
3.  A          EQU        $NODE
4.  I          DO $SL(F(1) ) ,A(I) EQU $SS(F(1),I)
5.             END        A
```

The value of STRNOD('ABC') looks like the node produced by the expression $L1('A','B','C').

Example 3:

STRNOD has an inverse function which converts a node reference whose selections are string-valued into a string and may be written:

```
1.  F          function
2.  NODSTR*    NAME       0
3.  A          EQU        ''
4.  I          DO F(1) ,A EQU A:F(1,I)
5.             END        A
```

The value of NODSTR($L1('A','B','C')) is 'ABC'.

1.10. MICROSTRINGS

Microstrings are a means of computing the line to be interpreted by MASM through the substitution of string expressions as parts of a line. Any portion of a line may be generated by microstring substitution with the exception of the line leveler (see 1.11). A microstring is introduced by a left bracket ([ ) and terminated by a right bracket ( ]). Brackets appearing between single quotes as part of a character string are not recognized as beginning a microstring. However, this effect may be obtained through the use of the concatenation operator. The expression between the brackets must be convertible to a string value in a system character set. The characters of the microstring

expression value replace the bracketed expression in the line MASM is to assemble. All microstring substitution operations take place before any normal interpretation functions, such as label definition, directive recognition, and so on. This means that the directive itself can be computed partially or entirely by microstring substitution.

For example, the set of definitions:

```
1.  A0          EQU         12
2.  A1          EQU         13
      .           .
      .           .
      .           .
16. A15         EQU         27
```

may be effected by the single line:

```
    I           DO 0,15  ,A[$CD(I) ] EQU I+12
```

since the value of $CD(I) is progressively '0', '1', '2', and so on. A less trivial example is provided by a PROC to create conditional jump procedures, as follows:

```
1.  F*          function
2.              END         [P(F(1),*F(2) )->'*'!' ']P(F(1),F(2) )
3.  P           PROC        2,2
4.  JE*         NAME        *'TNE'
5.  JNE*        NAME        *'TE'
6.  JEZ*        NAME        'TNZ'
7.  JNEZ*       NAME        'TZ'
8.  K           EQU         P(0,*0)
9.          [P(0,0)]    [K->'P(1,1),'!' ']F(1,1+K),;
10.                         F(1,2+K),P(0,1)+P(1,3+K)
11.             J           F(2,1),F(2,2)
12.             END
```

which creates a number of NAMEs callable by the programmer with two operand fields, the first representing the comparison field (where an A register is required for some tests but not others) and the second field being the jump destination. Thus, the above PROC may be used as in the following examples:

```
                JNE,S3      A0,TABLE,*X6  0,X11
```

or

```
                JEZ         FIELD,,H2  *RETURN
```

where the first generates the following instructions:

```
                TE          A0,TABLE,*X6,S3
                J           0,X11
```

and the second generates the following:

```
                TNZ         FIELD,,H2
                J           *RETURN
```

Note that in this example, three separate microstrings are used. In the function, a microstring computes either the string * or the void string, and the result is used as an operator, making the value returned by the function flagged or not. This function also illustrates access from one subassembly (the function) to variables defined in a higher level subassembly (P, which is the parameter tree of the PROC P which called the function F). In the PROC itself, microstrings are used to compute the directive of the test instruction based on which entry point to the PROC was used, and also, depending on whether an A-field is needed by the instruction, the value for the A-field. The F function is used by the PROC to set up the U- and X-fields, either of which may have a flag (for indirect addressing or index incrementation).

1.11. LEVELERS

Levelers are perhaps the most difficult to understand of all the MASM concepts. This is partly because, unlike all previous discussion of levels which pertained to dynamic call nesting levels, levelers pertain to the STATIC nesting levels of PROCs, functions, and REPEAT groups, and they apply at the time the sample is saved rather than when it is called for interpretation.

MASM lines and microstrings both may have levelers. A leveler has the form *%n:*, where *n* is an unsigned integer. If the leveler for any line or microstring is omitted, an implicit leveler of "%0:" is used. The leveler for a line is written preceding the label field, while the leveler for a microstring is written immediately after the left bracket.

For example:

```
%1:ABC    EQU      14
          +        [%2:$CD(I) ]
```

The static level of a line is determined by the following algorithm:

1. The level at the start of the main assembly is 0.
2. The level is incremented by 1 for each $PROC, $FUNC, or $REPEAT directive encountered.
3. The level is decremented by 1 for each $END or $ENDR encountered.

Thus, when saving a sample, levels are greater than zero. A line is interpreted if the leveler for the line is the same as the level of the line. A microstring is substituted if the leveler for the line plus the leveler for the microstring is equal to the level of the line. Therefore, if the present level is 2 (as it would be inside a function inside a PROC), the first line and the microstring on the second line would both be interpreted:

```
%2:A      EQU      'P(1,1)'
%1:       [%1:DIR]
```

It is important to understand that when the static nesting level is being calculated by the programmer, the same line may be encountered by MASM at different times with different levels. For example, if F is a function nested statically within the PROC P, when the sample for P is picked up, lines inside F are at level 2. When P is called, the sample for F as a function (rather than as part of P) is saved, and these lines are at level 1, since the PROC is being interpreted, which makes it part of the main assembly.

Some examples may help to make this clear. The first example is a PROC (perhaps a debugging PROC) which generates code only if a global assembly variable is set. The use of levelers allows the code inside the PROC to be deleted from the sample so that any calls to the PROC is faster than if the code were skipped each time the call was performed. In this and the following examples, the level of the line is indicated to the left of the line's label field.

```
0       P          PROC       0,2*DEBUG
1       SNOOP*     NAME       0
1       %1:        IF         DEBUG
1                  SLJ        SNOOPY
1                  +          P(1,2),P(1,1)
1       %1:        ENDF
1                  END
```

Note that the PROC produces the same results without the levelers, but the IF–ENDF group is skipped each time the PROC is called if DEBUG=0. The use of levelers thus saves time in this case. Note that DEBUG must be either 1 or 0 and may not be changed in the assembly.

A much more complicated example is taken from Church's lambda calculus. In this case, a PROC is defined which, when called, causes its waiting label to be defined as function with a given set of arguments. The function computes and returns the value of an expression using those arguments. The durmmy arguments and the expression are specified as parameters to the PROC.

```
0       P          PROC       *2
1       LAMBDA*    NAME       0
1       F$         function
2       *          NAME
2       %1:I       REPEAT     P(1)
3       [%1:P(1,I)]  EQU      F$([%1:$CD(I)])
3                  ENDR
2                  END        [%1:P(2,1)]
1                  END
0                  etc.
```

Note that when the sample for this PROC is picked up, no levelers match the level of their line, so no lines are interpreted. Now assume that LAMBDA is called as follows:

```
0       ADD        LAMBDA   'X','Y' 'X+Y'
```

Then the PROC P is interpreted, resulting in the following:

```
0       F$         function
1       *          NAME
1       %1:I       REPEAT     P(1)
2       [%1:P(1,I)]  EQU      F$([%1:$CD(I)])
2                  ENDR
1                  END        [%1:P(2,1)]
0                  END
```

Note that now several line levels match levelers. Therefore, MASM, as it now scans the body of PROC P attaches the waiting label to the function F$ NAME line. The REPEAT is then interpreted, since its leveler matches its level. This means that MASM generates the following line P(1) times under control of the REPEAT, having first picked it up as REPEAT sample:

```
1       [ %1 : P(1,I) ] EQU    F$( [ %1 : $CD(I) ] )
```

Since the level of this line is matched by the sum of the line's leveler and the leveler of each of the microstrings, MASM will perform the micro substitution. For the particular LAMBDA call being considered, MASM thus places the lines:

```
X          EQU        F$(1)
```

```
Y           EQU         F$(2)
```

into the function body as part of the process of picking up the function sample. This completes the action of the REPEAT. Finally, the line:

```
1           END     [ %1 : P(2,1) ]
```

is reached. Again, the microstring leveler plus the (implicit) line leveler matches the level of the line, so the substitution is made, resulting in the function body being completed with the line:

```
            END         X+Y
```

This completes the work of the PROC. ADD is now defined as a function which computes the sum of its arguments, just as if the written the lines are:

```
F$          function
ADD*        NAME
X           EQU         F$(1)
Y           EQU         F$(2)
            END         X+Y
```

since this is exactly what has been stored as the function body for ADD. We may then write:

```
            +           ADD(2,3)
```

and MASM generates the constant 5.

The LAMBDA PROC may be used in a more general way to build up the rest of the lambda calculus in the fashion of LISP, e.g.,

```
FACTORL     LAMBDA      'N'   'N ->N*FACTORL(N-1) ! 1'
            +           FACTORL(4)   . 24 GENERATED
```

or at a higher level:

```
APPLY       LAMBDA      'F','X'   'F(X)'
            +           APPLY( /$,2)    . $(2) GENERATED
            +           APPLY( /FACTORL,5)120 GENERATED
```

Note that, unlike LISP, MASM requires functions to be identified by the "/" operator.

A more practical use for levelers than LAMBDA might be the inclusion or deletion of debugging DISPLAYs from a PROC so that skipping would not have to be done for each call on the PROC. For example:

```
P           PROC        *1
DINSERT*    NAME
I           REPEAT      P(1)
%2:         IF          DEBUG
            DISPLAY     P(1,I)
%2:         ENDF
            INSERT      P(1,I)
            ENDR
            END
```

which always inserts the lines specified on the DINSERT call, but displays them only if DEBUG is set at the time the PROC is defined.

There are a few exceptions to the rules which should be noted. First of all, observe that $PROC, $FUNC, and $REPEAT are at the level below the sample which follows them. Second, note that a leveler is not needed on $END or $ENDR lines, since they are assumed to match up with the associated $PROC, $FUNC, or $REPEAT and thus have an implied leveler which is the same as the associated start of sample directive. Finally, it is important to remember that the expression in the operand field of a $NAME line is evaluated when sample is picked up, sc that a leveler of 1 is implied for the expression itself in all cases.

1.12. ERROR AND WARNING DIAGNOSTICS

MASM generates three different diagnostic flags (Table 1-14) and nine different error flags (Table 1-15). These flags may be found in the first portion of each line; a line may have none of these, or it may have more than one.

Table 1-14. MASM Diagnostic Flags

Flag	Meaning
U	Undefined identifier used on this line.
G	A $GO has transferred control from one PROC to a NAME defined in a different PROC (lateral transfer of control without change of nesting level).
?	Improbable coding sequence. The results may not be what the programmer intended to generate.

Table 1-15. MASM Error Flags

Flag	Meaning
C	Discrepancy between location counter values of pass 1 and pass 2.
D	Redefinition of a label originally defined by implication.
E	Error in syntax, or other miscellaneous errors not included in a more specialized flag.
I	Error in directive field: Unknown directive, or attempt to increment a blocked location counter by use of $RES.
L	Level errors, such as incorrect number of $END directives.
M	Microstring error.
Q	Missing quote terminator.
R	Relocation error—loss of relocation information due to mixed mode arithmetic, multiplication or division of a relocatable value by a value other than one or zero, etc.
T	Truncation of significant bits, value out of range, miscellaneous lost data.
V	Inappropriate value—integer where control information expected, etc.

In general, the error flags mark the output element in error, which is later noted by the Collector when an absolute program is built. The diagnostic flags do not mark the output RB as in error, although the presence of "?" flags may indicate a programming problem which causes the execution to be erroneous.

If the E option is used on the MASM control statement, the listing produced by MASM contains information which may help determine how errors were generated, if any were present. This includes a form of walkback from within a PROC nest.

If a user attempts to define, at Level 0, a symbol, which is relocated by an external reference, the symbol is dropped from the entry point table and the element is marked in error.

If a user attempts to define, at Level 0, a symbol which has associated with it the I$ form, the symbol is dropped from the entry point table and the element is marked in error.

1.13. DEFINITION MODE ASSEMBLY

Definition mode assembly saves the results of processing a set of definitions for use in several assemblies. The dictionary is built by MASM as it would be for any assembly. When the assembly is completed, the set of definitions external to the main assembly are saved in an omnibus element whose name and file are determined by field 2 of the MASM processor call statement. An assembly is determined to be definition mode if the $DEF directive is used in it. A definition mode assembly

may not contain any operations which generate code, as noted in the description of the $DEF directive. The results of a definition mode assembly are retrieved by using the $INCLUDE directive, which specifies the name of the omnibus element created by the definition mode assembly. The use of definition mode is considerably faster than the other method for processing definitions, since MASM is required to read the source language only once. Definitions retrieved by $INCLUDE are in an internal format and can be placed in the dictionary far faster than would be possible by scanning various directives, such as $EQU, $EQUF, and so on.

As an example, a set of register definitions may be loaded using PDP:

```
@PDP,I  AXR$
        DEF
X1      EQU     1
X2      EQU     2
        etc.
R15     EQU     79
AXR$*   PROC    0,0
        END
@MASM,I PROGRAM
        AXR$
        etc.
        END
```

in which case, the definition lines must be assembled each time the AXR$ PROC is called. The DEF directive in PDP functions in a completely different manner from the DEF in MASM; see the PDP descriptions in the 1100 Series Executive System, Volume 3, System Processors Programmer Reference, UP-4144.3 (current version).

The way to do this, using a MASM definition mode assembly is as follows:

```
@MASM,I AXR$
        DEF
        LEVEL   0,1,0
X1      EQU     1
        etc.
R15     EQU     79
        END
@MASM,I PROGRAM
        INCLUDE 'AXR$'
        etc.
        END
```

In this case, the definitions need to be assembled from source language only once. Note that the LEVEL directive is used to make the definitions given external to the main assembly level. This can also be done by externalizing each label defined, but that requires more work in writing. This is one of the most frequent uses of the $LEVEL directive.

MASM does not resolve relocation by an external reference if the symbol is INCLUDEd in an element in which the external reference is defined. The collector must resolve the relocation.

Example:

```
1.  @MASM,IS ELT1
2.           DEF
3.           LEVEL   0,1,0
```

```
4.   AZ          EQUF      TAG
5.               END
6.   @MASM,IS    ELT2
7.               INCLUDE   'ELT1'
8.   TAG*        +         2
9.               LA        A0,AZ
10.              END
```

Explanation:

Lines 1 through 5 constitute a definition mode assembly in which symbol AZ (line 4) is defined and has relocation by external reference by TAG.

Lines 6 through 10 constitute an element which has the symbol TAG (line 8) defined. The statement at line 9 produces a relocatable binary output which has relocation by external reference TAG. The collector must satisfy this.

2. Built-in 1100 Series Features

2.1. GENERAL

As mentioned earlier, MASM, with an unaltered environment, generates code for an 1100 Series hardware architecture. This section deals with 1100 Series features which are built into MASM.

2.2. EQUF (EQUATE A FIELD)

For documentation purposes, or if repeated references are made to the *u,x,j* fields of a word, it may be desirable to have a symbol reference these fields. The format is:

label EQUF *u,x,j*

where *u, x* and *j* are converted to binary values. The *label* is assigned a binary value with the I$ FORM attached, and the values of *u*, *x*, and *j* in bits 0–15, 18–21, and 26–29 respectively, with appropriate relocation items attached. If *u* is flagged, bit 16 is set. If *x* is flagged, bit 17 is set.

Example:

```
1.  IOBUFAD   EQUF    4,X3,H2
```

The symbol IOBUFAD is associated with the values 4,3,1 and the I$ form is attached. The result in instruction format is:

```
00  01  00  03  0    000004
```

If the symbol IOBUFAD is used with the 1100 Series instruction LA:

```
LA          A0,IOBUFAD
```

the result is:

```
10  01  00  03  0  000004
```

This is the same as:

```
LA,H2       A0,4,X3
```

2. IOFUNC EQUF 3,X3,S2

The symbol IOFUNC is associated with the values 3,3,14 and the I$ form is attached. The result in instruction format is:

```
00  14  00  03  0  000003
```

If the symbol IOFUNC is used with the SA instruction such as:

```
SA          A2,IOFUNC
```

the result is:

```
01  14  00  03  0  000003
```

This is the same as if the user writes:

```
SA,S2       A2,3,X3
```

2.3. WRD (DEFINE THE WORD SIZE)

For all assemblies, the word size is assumed to be 36 bits.

2.4. INSTRUCTION MNEMONIC REDEFINITION

As discussed in Section 1, the M option on the processor call card allows all 1100 Series instruction mnemonics to be redefined. This option increases assembly time substantially because the libraries are searched for every directive which is not typed as a procedure.

There exists a subset of the 1100 Series instruction repertoire which is always capable of redefinition regardless of the presence or the absence of the M option. This subset contains the following instructions.

f	j	a	Mnemonic	Description
05	17	01	SNZ	store negative zero
05	17	10	INC	increment by one
05	17	11	DEC	decrement by one
33	02		BTT	byte translate and test
33	05		BPD	convert byte to packed decimal
33	06		PDB	convert packed decimal to byte
33	07		EDIT	edit
37	00		QB	compress quarter word byte to binary
37	01		BQ	expand binary to quarter word byte
37	02		BHQ	compress quarter word byte to halves
37	04		QDB	quarter word to double binary
37	05		DBQ	double binary to quarter word byte
72	14		SCN	store channel number
73	17	00	TS	test and set
74	04		JK	console selective jump

Another way to redefine instructions is to introduce the redefinition procedures via a definition mode assembly. The user includes the procedure definitions in the element. The procedures replaces the standard instruction definitions.

2.5. 1100 SERIES INSTRUCTION REPERTOIRE

Tables 2-1 and 2-2 indicate the user and Executive instruction repertoire, respectively, currently supported by MASM. For detailed descriptions, see the appropriate 1100 System Processor and Storage Programmer Reference, UP-7970 (current version).

Table 2-1. User Instruction Repertoire

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
00	0-17	-	Illegal Operation	Causes Illegal Operation Fault Interrupt to MSR + 241_8
01	0-15	S,SA	Store A	$(A_a) \rightarrow U$
02	0-15	SN,SNA	Store Negative A	$-(A_a) \rightarrow U$
03	0-15	SM,SMA	Store Magnitude A	$\lvert (A_a) \rvert \rightarrow U$
04	0-15	S,SR	Store R	$(R_a) \rightarrow U$
05	00-17	SZ a = 00	Store Zero	Store constant 000000 000000, zeros, in location specified by operand address
05	00-17	SNZ a = 01	Store Negative Zero	Store constant 777777 777777, all ones, in location specified by operand address
05	00-17	SP1 a = 02	Store Positive One	Store constant 000000 000001, positive one, in location specified by operand address
05	00-17	SN1 a = 03	Store Negative One	Store constant 777777 777776, negative one, in location specified by operand address
05	00-17	SFS a = 04	Store Fieldata Spaces	Store constant 050505 050505, Fieldata spaces, in location specified by operand address
05	00-17	SFZ a = 05	Store Fieldata Zeros	Store constant 606060 606060, Fieldata zeros, in location specified by operand address
05	00-17	SAS a = 06	Store ASCII Spaces	Store constant 040040 040040, ASCII spaces, in location specified by operand address
05	00-17	SAZ a = 07	Store ASCII Zeros	Store constant 060060 060060, ASCII zeros, in location specified by operand address
05	00-17	XX	Increase/Decrease Instructions	
05	00-17	INC a = 10	Increase Operand by one	Increase operand by one. If initial operand or result is zero, execute NI; if not zero, skip NI.

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
05	00-17	DEC a = 11	Decrease Operand by one	Decrease operand by one. If initial operand or result is zero, execute NI; if not zero, skip NI.
05	00-17	INC2 a = 12	Increase Operand by two	Increase operand by two. If initial operand or result is zero, execute NI; if not zero, skip NI.
05	00-17	DEC2 a = 13	Decrease Operand by two	Decrease operand by two. If initial operand or result is zero, execute NI; if not zero, skip NI.
05	00-17	ENZ a = 14-17	Increase Operand by zero	Increase operand by zero. If initial operand or result is zero execute NI; if not zero, skip NI.
06	0-15	S,SX	Store X	$(X_a) \rightarrow U$
07	12	LDJ	Load D-bank Base and Jump	Ignore X_a bit positions 34-33; if D12 = 0, select BDR2; if D12 = 1, select BDR3
07	13	LIJ	Load I-bank Base And Jump	Ignore X_a bit positions 34-33; if D12 = 0, select BDR0; if D12 = 1, select BDR1
07	14	LPD	Load PSR Designators	$U_{6,5,3-0} \rightarrow$ PSRM Bit 6 → D20 Bit 2 → D8 Bit 5 → D17 Bit 1 → D5 Bit 3 → D10 Bit 0 → D4
07	15	SPD	Store PSR Designators	PSRM D-bits → U_{6-0} D20 → Bit 6 D8 → Bit 2 D17 → Bit 5 D5 → Bit 1 D12 → Bit 4 D4 → Bit 0 D10 → Bit 3
07	17	LBJ	Load Bank And Jump	Load BDR; jump to location specified by the operand address
10	0-17	L,LA	Load A	(U) → A
11	0-17	LN,LNA	Load Negative A	- (U) → A
12	0-17	LM,LMA	Load Magnitude A	| (U) | → A
13	0-17	LNMA	Load Negative Magnitude A	- | (U) | → A

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
14	0-17	A,AA	Add to A	(A) + (U) → A
15	0-17	AN,ANA	Add Negative To A	(A) - (U) → A
16	0-17	AM,AMA	Add Magnitude To A	(A) + | (U) | → A
17	0-17	ANM, ANMA	Add Negative Magnitude to A	(A) - | (U) | → A
20	0-17	AU	Add Upper	(A) + (U) → A+1
21	0-17	ANU	Add Negative Upper	(A) - (U) → A+1
22	0-15	BT	Block Transfer	$(X_x + u) \rightarrow X_a + u$; repeat k times
23	0-17	L,LR	Load R	$(U) \rightarrow R_a$
24	0-17	A,AX	Add to X	$(X_a) + (U) \rightarrow X_a$
25	0-17	AN,ANX	Add Negative to X	$(X_a) - (U) \rightarrow X_a$
26	0-17	LXM	Load X Modifier	$(U) \rightarrow X_{a17-0}$; X_{a35-18} unchanged
27	0-17	L,LX	Load X	$(U) \rightarrow X_a$
30	0-17	MI	Multiply Integer	(A) x (U) → A, A+1
31	0-17	MSI	Multiply Single Integer	(A) x (U) → A
32	0-17	MF	Multiply Fractional	(A) x (U) → A, A+1, left circular one bit
33#	00	BM	Byte Move	Transfer LJ0 bytes from source string to receiving string. Truncate or fill receiving string as required
33#	01	BMT	Byte Move With Translate	Translated and transfer LJ0 bytes from source string to receiving string. Truncate or fill receiving string as required
33#	03	BTC	Byte Translate and Compare	Translate and compare LJ0 bytes from string SJ0 to LJ1 bytes from string SJ1; terminate instruction on not equal or if both LJ0 and LJ1 are zero, when: (A_a) + ; string SJ0 > SJ1 (A_a) 0 ; string SJ0 = SJ1 (A_a) - ; string SJ0 < SJ1

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
33#	04	BC	Byte Compare	Compare LJ0 bytes from string SJ0 to LJ1 bytes from string SJ1; terminate instruction on not equal or if both LJ0 and LJ1 are zero
33#	05	BPD	Byte to Packed Decimal Convert	Convert (SJC) → packed decimal SJ1
33#	06	PDB	Packed Decimal to Byte Convert	Convert packed decimal (SJ0) → SJ1
33#	07	EDIT	Edit	Edit string SJ0 and transfer to string SJ1 under the control of string SJ2
33#	10	BI	Byte to Binary Single Integer Convert	Convert LJ0 bytes in string SJ0 to a signed binary integer in register A
33#	11	BDI	Byte to Binary Double Integer Convert	Convert LJ0 bytes in string SJ0 to a signed binary integer in registers A and A + 1
33#	12	IB	Binary Single Integer to Byte Convert	Convert signed binary integer in A to byte format and store in string SJ0
33#	13	DIB	Binary Double Integer to Byte Convert	Convert the binary integer in A and A + 1 to byte format and store in string SJ0
33#	14	BF	Byte to Single Floating Convert	Convert LJ0 bytes in string SJ0 to a single length floating point format in register A
33#	15	BDF	Byte to Double Floating Convert	Convert LJ0 bytes in string SJ0 to a double length floating point format in registers A and A + 1
33#	16	FB	Single Floating to Byte Convert	Convert the single length floating point number in A to byte format and store in string SJ0
33#	17	DFB	Double Floating to Byte Convert	Convert double length floating point number in A and A + 1 to byte format and store in string SJ0
34	0-17	DI	Divide Integer	(A, A + 1) divided by (U) → A; REMAINDER → A + 1
35	0-17	DSF	Divide Single Fractional	[(A, 36 sign bits) right algebraic shift 1 place] divided by (U) → A + 1

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
36	0-17	DF	Divide Fractional	[(A, A + 1) right algebraic shift 1 place] divided by (U) → A REMAINDER → A + 1
37#	00	QB	Quarter-Word Byte to Binary Compress	Discard $(A)_{35}$, $(A)_{26}$, $(A)_{17}$, and $(A)_{8}$; remaining bits (A) → A_{31-0}; $(A)_{31\ 18}$ → A_{35-32}
37#	01	BQ	Binary to Quarter-Word Byte Extend	Discard $(A)_{35-32}$; $(A)_{31-0}$ → A_{34-27} , A_{25-18} , A_{16-9} , and A_{7-0} ; zero fill A_{35} , A_{26} , A_{17} , and A_{8}
37#	02	QBH	Quarter-Word Byte to Binary Halves Compress	Discard $(A)_{35}$, $(A)_{26}$, $(A)_{17}$, and $(A)_{8}$; remaining bits (A) — A_{33-18} and A_{15-0}; $(A)_{33}$ → A_{35-34} $(A)_{15}$ → A_{17-16}
37#	03	BHQ	Binary Halves to Quarter-Word Byte Extend	Discard $(A)_{35-34}$ and $(A)_{17-16}$; remaining bits (A) → A_{34-27} , A_{25-18} , A_{16-9} , and A_{7-0} ; zero fill A_{35} , A_{26} , A_{17} , and A_{8}
37#	04	QDB	Quarter-Word Byte to Double Binary Compress	Discard A_{35} , A_{26} , A_{17} , A_{8} , A+135, A+126, A+117, and A+18; remaining bits (A,A+1) → A_{27-0} and A+1; $(A)_{27}$ A_{35-28}
37#	05	DBQ	Double Binary to Quarter-Word Byte Extend	Discard $(A)_{35-28}$; remaining bits (A,A+1) → A_{34-27} , A_{25-18} , A_{16-9} , A_{7-0} , $A+1_{34-27}$, $A+1_{25-18}$, $A+1_{16-9}$, and $A+1_{7-0}$; zero fill A_{35} , A_{26} , A_{17} , A_{8} , $A+1_{35}$, $A+1_{26}$, $A+1_{17}$, and $A+1_{8}$
37#	06	BA	Byte Add	Add the LJ0 bytes in string SJ0 to the LJ1 bytes in string SJ1 and store the results in string SJ2
37#	07	BAN	Byte Add Negative	Subtract the LJ0 bytes in string SJ0 from the LJ1 bytes in string SJ1 and store the results in string SJ2
40	0-17	OR	Logical OR	(A) OR (U) → A + 1
41	0-17	XOR	Logical Exclusive OR	(A) XOR (U) → A + 1
42	0-17	AND	Logical AND	(A) AND (U) → A + 1
43	0-17	MLU	Masked Load Upper	[(U) AND (R2)] OR [(A) AND NOT (R2)] → A + 1
44	0-17	TEP	Test Even Parity	Skip NI if (U) AND (A) has even parity

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
45	0–17	TOP	Test Odd Parity	Skip NI if (U) AND (A) has odd parity
46	0–17	LXI	Load X Increment	(U) → $(X_a)_{35-18}$; $(X_a)_{17-0}$ unchanged
47	0–17	TLEM	Test Less Than or Equal to Modifier	Skip NI if $(U)_{17-0} \leq (X_a)_{17-0}$; always $(X_a)_{17-0} + (X_a)_{35-18} \rightarrow X_{a17-0}$
		TNGM	Test Not Greater Than Modifier	
50	0–17	TZ	Test Zero	Skip NI if (U) = ± 0
51	0–17	TNZ	Test Nonzero	Skip NI if (U) ≠ ± 0
52	0–17	TE	Test Equal	Skip NI if (U) = (A)
53	0–17	TNE	Test Not Equal	Skip NI if (U) ≠ (A)
54	0–17	TLE	Test Less Than or Equal	Skip NI if (U) ≤ (A)
		TNG	Test Not Greater	
55	0–17	TG	Test Greater	Skip NI if (U) > (A)
56	0–17	TW	Test Within Range	Skip NI if (A) < (U) ≤ (A + 1)
57	0–17	TNW	Test Not Within Range	Skip NI if (U) ≤ (A) or (U) > (A + 1)
60	0–17	TP	Test Positive	Skip NI if $(U)_{35} = 0$
61	0–17	TN	Test Negative	Skip NI if $(U)_{35} = 1$
62	0–17	SE	Search Equal	Skip NI if (U) = (A), else repeat
63	0–17	SNE	Search Not Equal	Skip NI if (U) ≠ (A), else repeat
64	0–17	SLE	Search Less Than or Equal	Skip NI if (U) ≤ (A), else repeat
		SNG	Search Not Greater	
65	0–17	SG	Search Greater	Skip NI if (U) > (A), else repeat
66	0–17	SW	Search Within Range	Skip NI if (A) < (U) ≤ (A + 1), else repeat
67	0–17	SNW	Search Not Within Range	Skip NI if (U) ≤ (A) or (U) > (A + 1), else repeat

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
70		JGD	Jump Greater And Decrement	Jump to U if (Control Register)$_{ja}$>0; go to NI if (Control Register)$_{ja}$ ≤ 0; always (Control Register)$_{ja}$ −1 → Control Register$_{ja}$
71	00	MSE	Mask Search Equal	Skip NI if (U) AND (R2) = (A) AND (R2), else repeat
71	01	MSNE	Mask Search Not Equal	Skip NI if (U) AND (R2) ≠ (A) AND (R2), else repeat
71	02	MSLE MSNG	Mask Search Less Than or Equal Mask Search Not Greater	Skip NI if (U) AND (R2) ≤ (A) AND (R2), else repeat
71	03	MSG	Mask Search Greater	Skip NI if (U) AND (R2) > (A) AND (R2), else repeat
71	04	MSW	Masked Search Within Range	Skip NI if (A) AND (R2) < (U) AND (R2) ≤ (A + 1) AND (R2), else repeat
71	05	MSNW	Masked Search Not Within Range	Skip NI if (U) AND (R2) ≤ (A) AND (R2) or (U) AND (R2) > (A + 1) AND (R2), else repeat
71	06	MASL	Masked Alphanumeric Search Less Than or Equal	Skip NI if (U) AND (R2) ≤ (A) AND (R2), else repeat
71	07	MASG	Masked Alphanumeric Search Greater	Skip NI if (U) AND (R2) > (A) AND (R2), else repeat
71	10	DA	Double-Precision Fixed-Point Add	(A, A + 1) + (U, U + 1) → A, A + 1
71	11	DAN	Double-Precision Fixed-Point Add Negative	(A, A + 1) − (U, U + 1) → A, A + 1
71	12	DS	Double Store A	(A, A + 1) → U, U + 1
71	13	DL	Double Load A	(U, U + 1) → A, A + 1
71	14	DLN	Double Load Negative A	− (U, U + 1) → A, A + 1
71	15	DLM	Double Load Magnitude A	| (U, U + 1) | → A, A + 1

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
71	16	DJZ	Double-Precision Jump Zero	Jump to U if (A, A + 1) = ± 0; go to NI if (A, A + 1) ≠ ± 0
71	17	DTE	Double-Precision Test Equal	Skip NI if (U < U + 1) = (A, A + 1)
72	01	SLJ	Store Location And Jump	Relative P + 1 → U_{17-0}; jump to U + 1
72	02	JPS	Jump Positive And Shift	Jump to U if $(A)_{35} = 0$; go to NI if $(A)_{35} = 1$; always shift (A) left circularly one bit position
72	03	JNS	Jump Negative And Shift	Jump to U if $(A)_{35} = 1$; go to NI if $(A)_{35} = 0$; always shift (A) left circularly one bit position
72	04	AH	Add Halves	$(A)_{35-18} + (U)_{35-18}$; → $(A)_{35-18}$; $(A)_{17-0} + (U)_{17-0}$ → A_{17-0}
72	05	ANH	Add Negative Halves	$(A)_{35-18} - (U)_{35-18}$ → $(A)_{35-18}$; $(A)_{17-0} - (U)_{17-0}$ → A_{17-0}
72	06	AT	Add Thirds	$(A)_{35-24} + (U)_{35-24}$ → A_{35-24}; $(A)_{23-12} + (U)_{23-12}$ → A_{23-12}; $(A)_{11-0} + (U)_{11-0}$ → A_{11-0}
72	07	ANT	Add Negative Thirds	$(A)_{35-24} - (U)_{35-24}$ → A_{35-24}; $(A)_{23-12} - (U)_{23-12}$ → A_{23-12}; $(A)_{11-0} - (U)_{11-0}$ → A_{11-0}
72	10	EX	Execute	Execute the instruction at U
72	11	ER	Executive Request	Interrupt to MSR + 242_8
72	16	SRS	Store Register Set	A_a contains address and count for each of two GRS areas
72	17	LRS	Load Register Set	Move specified storage area to GRS area(s)
73	00	SSC	Single Shift Circular	Shift (A) right circularly U places
73	01	DSC	Double Shift Circular	Shift (A, A + 1) right circularly U places
73	02	SSL	Single Shift Logical	Shift (A) right U places, zero fill
73	03	DSL	Double Shift Logical	Shift (A, A + 1) right U places, zero fill
73	04	SSA	Single Shift Algebraic	Shift (A) right U places, sign fill

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
73	05	DSA	Double Shift Algebraic	Shift (A, A + 1) right U places, sign fill
73	06	LSC	Load Shift And Count	(U) → A; shift (A) left circularly until $(A)_{35} \neq (A)_{34}$; number of shifts → A + 1
73	07	DLSC	Double Load Shift and Count	(U, U + 1) → A, A + 1; shift (A, A + 1) left circularly until $(A, A + 1)_{71} \neq (A, A + 1)_{70}$; number of shifts → A + 2
73	10	LSSC	Left Single Shift Circular	Shift (A) left circularly U places
73	11	LDSC	Left Double Shift Circular	Shift (A, A + 1) left circularly U places
73	12	LSSL	Left Single Shift Logical	Shift (A) left U places, zero fill
73	13	LDSL	Left Double Shift Logical	Shift (A, A + 1) left U places, zero fill
73	17	TS a = 00	Test And Set	If $(U)_{30} = 1$, interrupt to MSR + 244_8; if $(U)_{30} = 0$, go to NI; always $01_8 \rightarrow U_{35-0}$
73	17	TSS a = 01	Test And Set And Skip	if $(U)_{30} = 0$, skip NI; if $(U)_{30} = 1$, go to NI; always $01_8 \rightarrow U_{35-30}$
73	17	TCS a = 02	Test and Clear And Skip	If $(U)_{30} = 0$, go to NI; if $(U)_{30} = 1$, skip NI; always clear $(U)_{35-30}$
74	00	JZ	Jump Zero	Jump to U if (A) = ± 0 go to NI if (A) ≠ ± 0
74	01	JNZ	Jump Nonzero	Jump to U if (A) ≠ ± 0; go to NI if (A) = ± 0
74	02	JP	Jump Positive	Jump to U if $(A)_{35} = 0$; go to NI if $(A)_{35} = 1$
74	03	JN	Jump Negative	Jump to U if $(A)_{35} = 1$; go to NI if $(A)_{35} = 0$
74	04	J JK	Jump Jump Key	Jump to U if a = 0 or if a = set SELECT JUMPS control circuit; go to NI if neither is true
74	05	HJ HKJ	Halt Jump Halt Keys and Jump	Stop if a = 0 or if [a field AND set SELECT STOPS control circuits] ≠ 0; on restart or continuation jump to U

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
74	06	NOP	No Operation	Proceed to next Instruction
74	07	AAIJ	All All I/O Interrupts And Jump	Allow all I/O interrupts and jump to U
74	10	JNB	Jump No Low Bit	Jump to U if $(A)_0 = 0$; go to NI if $(A)_0 = 1$
74	11	JB	Jump Low Bit	Jump to U if $(A)_0 = 1$; go to NI if $(A)_0 = 0$
74	12	JMGI	Jump Modifier Greater and Increment	Jump to U if $(X_a)_{17-0} > 0$; go to NI if $(X_a)_{17-0} \leq 0$; always $(X_a)_{17-0} + (X_a)_{35-18} \rightarrow Xa_{17-0}$
74	13	LMJ	Load Modifier and Jump	Relative P + 1 $\rightarrow (X_a)_{17-0}$; jump to U
74	14	JO a = 00	Jump Overflow	Jump to U if D1 = 1; go to NI if D1 = 0
74	14	JFU a = 01	Jump Floating Underflow	Jump to U if D21 = 1, clear D21; go to NI if D21 = 0
74	14	JFO a = 02	Jump Floating Overflow	Jump to U if D22 = 1, clear D22; go to NI if D22 = 0
74	14	JDF a = 03	Jump Divide Fault	Jump to U if D23 = 1, clear D23; go to NI if D23 = 0
74	15	JNO a = 00	Jump No Overflow	Jump to U if D1 = 0; go to NI if D1 = 1
74	15	JNFU a = 01	Jump No Floating Underflow	Jump to U if D21 = 0; go to NI if D21 = 1; clear D21
74	15	JNFO a = 02	Jump No Floating Overflow	Jump to U if D22 = 0; go to NI if D22 = 1; clear D22
74	15	JNDF a = 03	Jump No Divide Fault	Jump to U if D23 = 0; go to NI if D23 = 1; clear D23

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
74	16	JC	Jump Carry	Jump to U if D0 = 1; go to NI if D0 = 0
74	17	JNC	Jump No Carry	Jump to U if D0 = 0; go to NI if D0 = 1
76	00	FA	Floating Add	(A) + (U) → A; RESIDUE → A + 1 if D17 = 1
76	01	FAN	Floating Add Negative	(A) – (U) → A; RESIDUE → A + 1 if D17 = 1
76	02	FM	Floating Multiply	(A) x (U) → A (and A + 1 if D17 = 1)
76	03	FD	Floating Divide	(A) divided by (U) → A; REMAINDER → A + 1 if D17 = 1
76	04	LUF	Load and Unpack Floating	$\lvert (U) \rvert_{34-27} \rightarrow A_{7-0}$, zero fill $(U)_{26-00} \rightarrow A + 1_{26-00}$, sign fill $(U)_{35} \rightarrow A + 1_{35}$;
76	05	LCF	Load and Convert To Floating	$(U)_{35} \rightarrow A + 1_{35}$, [NORMALIZED $(U)]_{26-0} \rightarrow A + 1_{26-0}$; if $(U)_{35} = 0$, $(A)_{7-0} \pm$ NORMALIZING COUNT → A + 1_{34-27}; if $(U)_{35} = 1$, ones complement of [$(A)_{7-0} \pm$ NORMALIZING COUNT] → A + 1_{34-27}
76	06	MCDU	Magnitude of Characteristic Difference To Upper	$\lvert \lvert (A) \rvert_{35-27} - \lvert (U) \rvert_{35-27} \rvert \rightarrow A + 1_{8-0}$; ZEROS → A + 1_{35-9}
76	07	CDU	Characteristic Difference To Upper	$\lvert (A) \rvert_{35-27} - \lvert (U) \rvert_{35-27} \rightarrow A + 1_{8-0}$ SIGN BITS → A + 1_{35-9}
76	10	DFA	Double-Precision Floating Add	(A, A + 1) + (U, U + 1) → A, A + 1
76	11	DFAN	Double-Precision Floating Add Negative	(A, A + 1) – (U, U + 1) → A, A + 1
76	12	DFM	Double-Precision Floating Multiply	(A, A + 1) x (U, U + 1) → A, A + 1
76	13	DFD	Double-Precision Floating Divide	(A, A + 1) divided by (U, U + 1) → A, A + 1
76	14	DFU	Double Load and Unpack Floating	$\lvert (U, U + 1) \rvert_{70-60} \rightarrow A_{10-0}$, zero fill; $(U, U + 1)_{59-36} \rightarrow A + 1_{23-0}$, sign fill; $(U, U + 1)_{35-0} \rightarrow A + 2$

Table 2-1. User Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
76	15	DLCF, DFP	Double Load and Convert To Floating	$(U)_{35} \rightarrow A + 1_{35}$; [NORMALIZED $(U < U + 1)]_{59-0} \rightarrow A + 1_{23-0}$ and A + 2; if $(U)_{35}$, $(A)_{10-0} \pm$ NORMALIZING COUNT $\rightarrow A + 1_{34-24}$; if $(U)_{35} = 1$, ones complement of $[(A)_{10-0} \pm$ NORMALIZING COUNT$] \rightarrow A + 1_{34-24}$
76	16	FEL	Floating Expand and Load	If $(U)_{35} = 0$; $(U)_{35-27} + 1600_8 \rightarrow A_{35-24}$ If $(U)_{35} = 1$; $(U)_{35-27} - 1600_8 \rightarrow A_{35-24}$ $(U)_{26-3} \rightarrow A_{23-0}$; $(U)_{2-0} \rightarrow A + 1_{35-33}$; $(U)_{35} \rightarrow A + 1_{32-0}$
76	17	FCL	Floating Compress and Load	If $(U)_{35} = 0$; $(U)_{35-24} - 1600_8 \rightarrow A_{35-27}$; if $(U)_{35} = 1$; $(U)_{35-24} + 1600_8 \rightarrow A_{35-27}$ $(U)_{23-0} \rightarrow A_{26-3}$; $(U+1)_{35-33} \rightarrow A_{2-0}$

# indicates for 1110, 1100/40 only
* indicates for 1100/80 only
** not 1100/80

Table 2-2. Executive Instruction Repertoire

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
07	01	SOA	Store Output Access Control Word	$A \rightarrow$ OACR; channel number per U_{5-0}
07	02	SIP	Store Input Pointer Word	$(A) \rightarrow$ ICPR; channel number per U_{5-0}
07	03	SOP	Store Output Pointer Word	$(A) \rightarrow$ OCPR; channel number per U_{5-0}
07	04	LIA	Load Input Access Control Word	$(IACR) \rightarrow A$; channel number per U_{5-0}
07	05	LOA	Load Output Access Control Word	$(OACR) \rightarrow A$; channel number per U_{5-0}
07	06	LIP	Load Input Pointer Word	$(ICPR) \rightarrow A$; channel number per U_{5-0}

Table 2-2. Executive Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
07	07	LOP	Load Output Pointer Word	(OCPR) → A; channel number per U_{5-0}
07	10	LCB	Load Chain Base Register	If a=0, $(U)_{14-0}$ → CBR of IOAU channels 0–23 If a=1, $(U)_{14-0}$ → CBR of IOAU channels 24–47
07	11	LPI	Load Processor Interrupt Pointer	If a=0, $(U)_{1-0}$ → PIP register of IOAU for channels 0–23 If a=1, $(U)_{1-0}$ → PIP register of IOAU for channels 0–23
07	16	LBR (a = 0)	Load Breakpoint Register	(U) → Breakpoint Register
07	16	SJS (a = 1)	Store Jump Stack	(Jump History Stack) → U, repeat NOTE: The SJS instruction is noninterruptible
72	00	IMI	Initiate Maintenance Interrupt	Send Attention Interrupt to Maintenance Processor. If in Maintenance Mode, otherwise NO-OP
72	13	PAIJ	Prevent All I/O Interrupts And Jump	Prevent all I/O interrupts and jump to U
72**	14	SCN	Store Channel Number	If a=0, channel number → U_{3-0} If a=1, channel number → U_{3-0} ; CAU number → U_{5-4} If a=2, channel number → U_{3-0} ; CAU number → U_{5-4} If a=3, channel number → U_{5-0} ; CAU number → U_{14-12}
72	15	TRA	Test Relative Address	Used to determine whether a relative address is within a given relative addressing range
72**	15	LPS (a = 0)	Load Processor State Register	(U) → PSRM0
72**	15	LMP (a = 1)	Load Main Processor State Register	(U,U+1) → PSRM0, PSRM1
72**	15	LUP (a = 2)	Load Utility Processor State Register	(U,U+1) → PSRU
72**	16	LSL (a = 0)	Load Main Storage Limits Register	(U) → SLRM

Table 2-2. Executive Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
72**	16	LUS (a = 1)	Load Utility Storage Limits Register	(U) → SLRU
72**	16	SL (a = 2)	Store Main Storage Limits Register	(SLRM) → U
72**	16	SUL (a = 2)	Store Utility Storage Limits Register	(SLRU) → U
73**	14	III (a = 0 through 5)	Initiate Interprocessor Interrupt	Initiate interprocessor–interrupt per a; a = 0: Interrupt CAU Number 0 a = 1: Interrupt CAU Number 1 a = 2: Interrupt CAU Number 2 a = 3: Interrupt CAU Number 3 a = 4: Interrupt CAU Number 4 a = 5: Interrupt CAU Number 5
73**	14	ESDC (a = 10_8)	Enable Second Day Clock	Enable dayclock in IOAU having channels 24–47
73	14	EDC a = 11	Enable Day Clock	Enable dayclock in IOAU having channels 0–23
73	14	DDC a = 12	Disable Day Clock	Disable dayclock
73	14	SDC a = 13	Select Day Clock	Select Internal Day Clock
73**	14	ES (a = 14_8)	Enable Storage Reference Counters	Enable storage reference counters on next instruction
73	15	SIL a = 00	Select Interrupt Locations	$(U)_{8-0}$ → MSR
73	15	LBRX a = 02	Load Breakpoint Register	Transfer operand to Breakpoint Register
73	15	LQT a = 03	Load Quantum Timer	Place full-word operand in Quantum Timer

Table 2-2. Executive Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
73	15	IIIX a = 04	Initiate Interprocessor Interrupt	Interrupt processor specified by operand address value
73	15	SPID a = 05	Store Processor ID	Store: binary serial number in first third; 2-character Fieldata revision level in second third; processor in last sixth of operand
73	15	RAT a = 06	Reset Auto-Recovery Timer	Reset auto-recovery timer in system transition unit
73	15	TAP a = 07	Toggle Auto-Recovery Path	Toggle path selection after each auto-recovery attempt
73	15	LB a = 10	Load Base	Place operand bits 0 through 17 in base value field of BDR specified by bits 33 and 34 of X_x
73	15	LL a = 11	Load Limits	Place operand bits 15 through 23 and 24 through 35 in BDR limits fields specified by X_x bits 33 and 34
73	15	LAE a = 12	Load Addressing Environment	Place the double-word operand in GRS location 046 and 047 and the four respective Bank Descriptor Registers
73	15	SQT a = 13	Store Quantum Time	Store Quantum Timer value at the storage location specified by operand address. Executing this instruction has no effect on D29. It may be in GRS.
73	15	LD a = 14	Load Designator Register	Place full-word operand in Designator Register
73	15	SD a = 15	Store Designator Register	Store Designator Register contents at location specified by operand address
73	15	UR a = 16	User Return	Provides an orderly return to a user program
73	15	SSS a = 17	Store System Status	Store two system status words at the location specified by operand address

Table 2-2. Executive Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
73**	16	LCR a = 00	Load Channel Select Register	$(U)_{5-0} \rightarrow$ CSR; if $(U)_9 = 1$, select back-to-back transfer mode
73*	16	LLA (a = 1)	Load Last Address Register	$(U)_{8-0} \rightarrow$ LAR
75*	00	LIC	Load Input Channel	For channel [a OR CSR]: (U) → IACR; set input active; clear input monitor
75*	01	LICM	Load Output Channel and Monitor	For channel [a OR CSR]: (U) → IACR; set input active; set input monitor
75*	02	JIC	Jump On Input Channel Busy	Jump to U if input active is set for channel [a OR CSR]: go to NI if input active is clear
75*	03	DIC	Disconnect Input Channel	For channel [a OR CSR]: clear input active; clear input monitor
75*	04	LOC	Load Output Channel	For channel [a OR CSR]: (U) → OACR (ISI only); set output active; clear output monitor; clear function active (ISI only)
75*	05	LOCM	Load Output Channel and Monitor	For channel [a OR CSR]: (U) → OACR (ISI only); set output active; set output monitor; clear external function active (ISI only)
75*	06	JOC	Jump On Output Channel Busy	Jump to U if output active is set for channel [a OR CSR]: go to NI if output active is clear
75*	07	DOC	Disconnect Output Channel	For channel [a OR CSR]: clear output active; clear output monitor; clear function active (ISI only)
75*	10	LFC	Load Function In Channel	For channel [a OR CSR]: (U) → OACR; set output active (ISI only), function active (ISI only), and force external function; clear output monitor (ISI only)
75*	11	LFCM	Load Function In Channel and Monitor	For channel [a OR CSR]: (U) → OACR; set output active (ISI only), function active (ISI only), force external function, and output monitor (ISI only)
75*	12	JFC	Jump On Function In Channel	Jump to U if force external function is set for channel [a OR CSR]; go to NI if force external function is clear

Table 2-2. Executive Instruction Repertoire (continued)

Function Code (Octal)		Mnemonic	Instruction	Description
f	j			
75*	14	AACI	Allow All Channel External Interrupts	Allow all external interrupts
75*	15	PACI	Prevent All Channel External Interrupts	Prevent all external interrupts
75*	16	ACI	Allow Channel Interrupts	If a=0, allow interrupts on channels 23-0 specified by one bits in $(U)_{23-0}$ If a=1, allow interrupts on channels 47-24 specified by one bits in $(U)_{23-0}$
75*	17	PCI	Prevent Channel Interrupts	If a=0, prevent interrupts on channels 23-0 specified by one bits in $(U)_{23-0}$ If a=1, prevent interrupts on channels 47-24 specified by one bits in $(U)_{23-0}$
75	00	SRL	Select Release	Initiates the execution of a CCW list
75	01	SIOF	Start I/O Fast Release	Initiates operation specified by bit 00 through 15 of CAW
75	02	TIO	Test I/O	Interrogates the channel, subchannel and device
75	03	TSC	Test Subchannel	Interrogates the channel and subchannel
75	04	HDV	Halt Device	Terminates current operation on channel and subchannel
75	05	HCH	Halt Channel	Terminates current operation on channel
75	10	LCR	Load Channel Register	Load the interrupt mask register
75	11	LTCW	Load Control Words	Loads the status table subchannel
75#	16	ACI	Allow Channel Interrupts a=o	Allow interrupts on channel 23-0 specified by one bits in $(U)_{23-0}$
75#	16	ACI	Allow Channel Interrupts a=1	Allow interrupts on channels 47-24 specified by one bits in $(U)_{23-0}$
75#	17	PCI	Prevent Channel Interrupts a=0	Prevent interrupts on channel 23-0 specified by one bits in $(U)_{23-0}$
75#	17	PCI	Prevent Channel Interrupts a=1	Prevent interrupts on channels 47-24 specified by one bits in $(U)_{23-0}$

# indicates for 1110, 1100/40 only
* indicates for 1100/80 only
** not 1100/80

Index

USER COMMENT SHEET

Comments concerning the content, style, and usefulness of this manual may be made in the space provided below. Please fill in the requested information.

Requests for copies of manuals, lists of manuals, pricing information, etc. should be made through your 1100 Series site manager to your Sperry Univac representative or the Sperry Univac office serving your locality.

System: _____

Manual Title: _____

UP No: _____ Revision No: _____ Update: _____

Name of User: _____

Address of User: _____

Comments:

CUT

FOLD

FIRST CLASS

PERMIT NO. 21

BLUE BELL, PA.

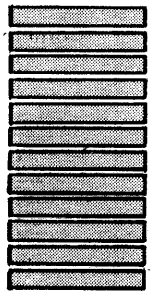
BUSINESS REPLY MAIL

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

POSTAGE WILL BE PAID BY

SPERRY UNIVAC

SYSTEMS SUPPORT
ATTN: INFORMATION SERVICES M.S. 4533
P.O. BOX 3942
ST. PAUL, MINNESOTA 55165



CUT

FOLD