Assembler as a Higher Level Language: Macros and Conditional Assembly Techniques

SHARE 83, Session 4885-4886

August 10, 1994

John R. Ehrman

International Business Machines Corporation
Software Solutions Division
Santa Teresa Laboratory
555 Bailey Avenue
San Jose, California 95141

Synopsis:

This presentation is taken from a forthcoming "IBM High Level Assembler/MVS & VM & VSE Tutorial Guide". The examples in this document are for purposes of illustration only, and no warranty of correctness or applicability is implied or expressed.

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States and/or other countries:

IBM	ESA	HLASM
MVS/ESA	MVS/SP	High Level Assembler
MVS/XA	System/360	High Level Assembler/MVS & VM & VSE
System/370	System/370/390	IBM High Level Assembler/MVS & VM & VSE
System/390	VM/ESA	
VM/XA	VSE	
VSE/ESA		

Publications

The currently available product publications for High Level Assembler/MVS & VM & VSE are:

- High Level Assembler/MVS & VM & VSE *Diagnosis Guide*, SC26-3110
- High Level Assembler/MVS & VM & VSE *Fact Sheet*, GC26-3189
- High Level Assembler/MVS & VM & VSE *General Information*, GC26-4943
- High Level Assembler/MVS & VM & VSE *Installation*, SC26-4942
- High Level Assembler/MVS & VM & VSE *Language Reference*, SC26-4940
- High Level Assembler/MVS & VM & VSE *Licensed Program Specifications*, GC26-4944
- High Level Assembler/MVS & VM & VSE *Programmer's Guide*, SC26-4941

- High Level Assembler/MVS & VM & VSE *Presentation Guide*, GG24-3910

Soft-copy High Level Assembler publications are available on the following *IBM Online Library Omnibus Edition* Compact Disks:

- *Application Development Collection*, SK2T-1237
  - The HLASM component is SK2T-2364
- *VM Collection*, SK2T-2067
- *MVS Collection*, SK2T-0710
- *VSE Collection*, SK2T-0060
- *Transaction Processing and Data Collection*, SK2T-0730

Related useful publications include:

- *Program Update Information, VMSES and VMSES/E, Support for the High Level Assembler, VM/ESA™ Release 1.0 (370 Feature), Release 1.1 and Release 2.0, APAR VM54803*, GC24-5661.

Electronic Reader Comments may be submitted via E-mail to:

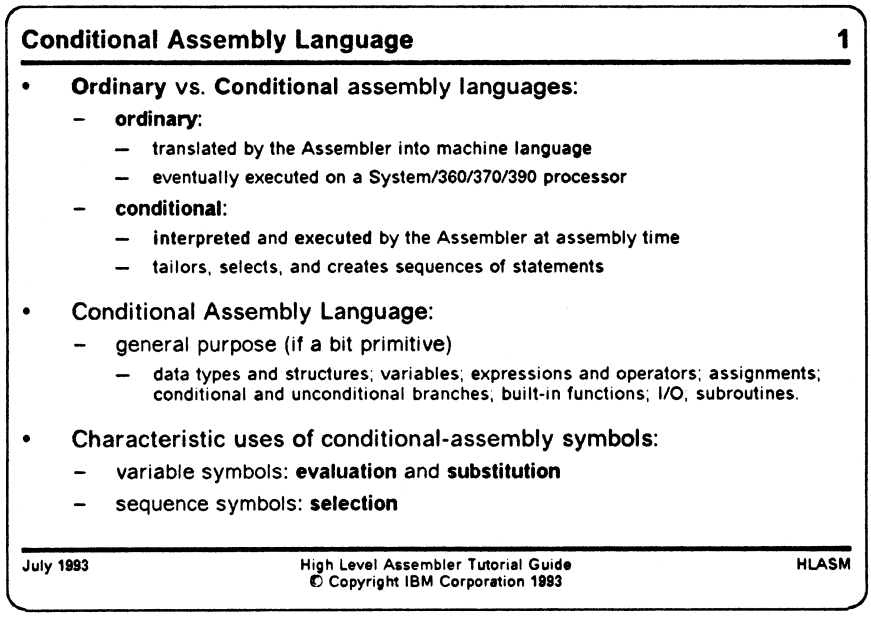
Internet: HLASMPUB@STLVM27.VNET.IBM.COM
IBMMAIL: USIB5V7X at IBMMAIL

Contents

Figures

The Conditional Assembly Language

The "ordinary" assembly language is translated by the Assembler into the machine language of the System/360/370/390 processors, for eventual execution by such a processor. The "conditional" assembly language is *interpreted* and *executed* by the Assembler *at assembly time* to tailor, select, and create desired sequences of statements.

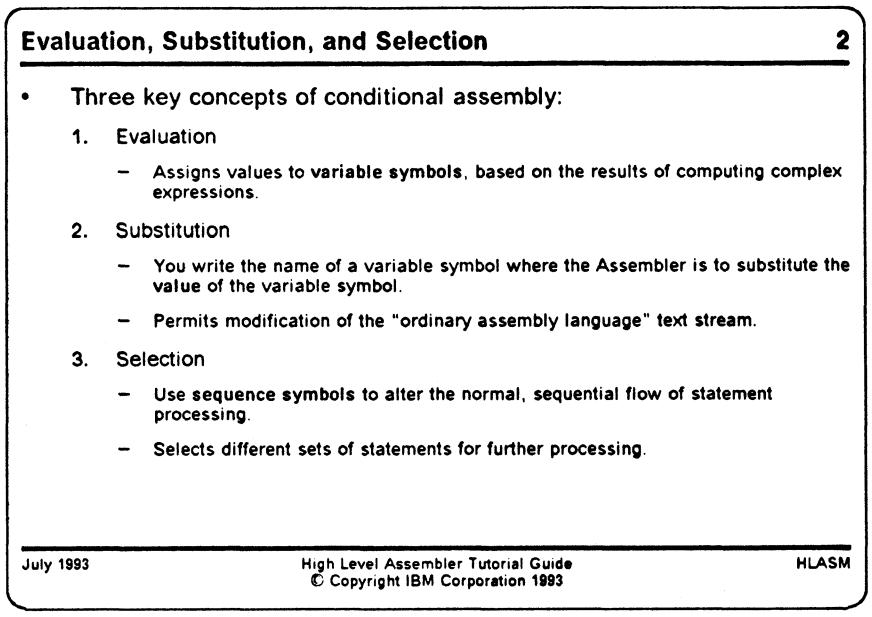


Though primitive in many respects, the conditional assembly language has most of the basic elements of a general purpose programming language: data types and structures, variables, expressions and operators, assignments, conditional and unconditional branches, some built-in functions, simple forms of I/O, and subroutines.

The distinctive feature of the conditional assembly language is the introduction of two new classes of symbols:

- **variable symbols** are used for evaluation and substitution
- **sequence symbols** are used for selection among alternative actions.

Just as "normal" or "ordinary" assembly deals with ordinary symbols — assigning values to symbols and using those values to evaluate various kinds of expressions — the conditional assembly language uses variable and sequence symbols.



Evaluation, Substitution, and Selection

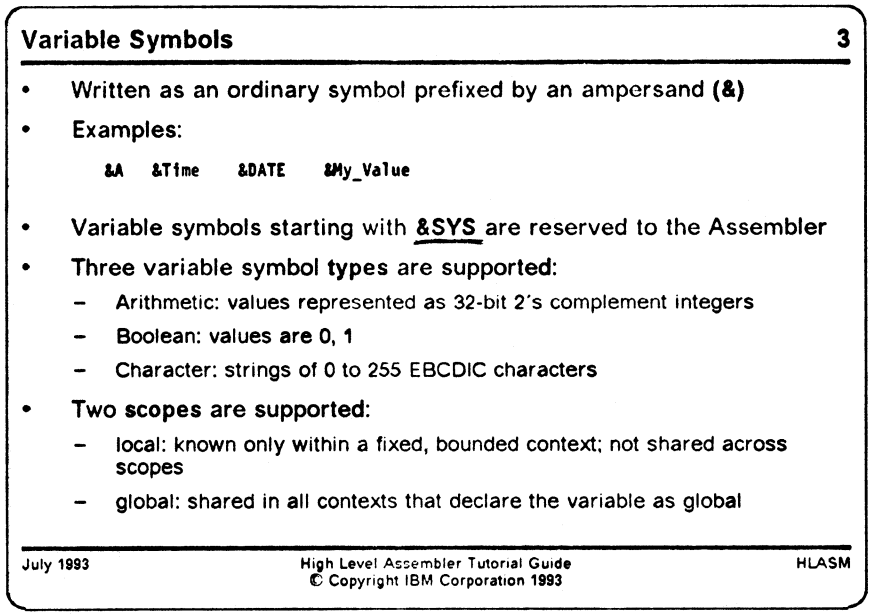
There are three key concepts in the conditional assembly language:

- evaluation
- substitution
- selection

Evaluation allows you to assign values to variable symbols based on the results of computing complex expressions.

Substitution is achieved by writing the name of a special symbol, a *variable symbol*, in a context that the Assembler will recognize as requiring substitution of the *value* of the variable symbol. This permits *modification* of the "ordinary assembly language" text stream to be processed by the assembler.

Selection is achieved by using *sequence symbols* to alter the normal, sequential flow of statement processing. This permits different sets of statements to be presented to the Assembler for further processing.



Variable Symbols

In addition to the familiar "ordinary" symbols managed by the assembler – internal and external – there is also a class of **variable symbols**. Variable symbols obey scope rules supporting two types that roughly approximate internal and external ordinary symbols, but they are not retained past the end of an assembly, and do not appear in the object text produced by the assembly.

Variable symbols are written just as are ordinary symbols, but with the ampersand character (&) prefixed. Examples of variable symbols are:

```
&A          &a              (these two are treated identically)
&Time
&DATE
&My_Value
```

As indicated in these examples, variable symbols may be written in mixed-case characters; all appearances will be treated as being equivalent to their upper-case versions. Variable symbols starting with the characters **&SYS** are reserved to the Assembler.

There are three types of variable symbols, corresponding to the values they may take:

arithmetic
The allowed values of an arithmetic variable symbol are those of 32-bit (fullword) two's complement integers (i.e., between -2^{31} and $+2^{31}-1$. (Be aware that in certain contexts, their values may be substituted as *unsigned* integers!)

boolean
The allowed values of a boolean variable symbol are 0 and 1.

character
The value of a character variable symbol may contain from 0 to 255 characters, each being any EBCDIC character. (A character variable symbol containing no characters is sometimes called a *null string*.)

Declaring Variable Symbols 4

- There are six **explicit** declaration statements (3 types × 2 scopes)

	Arithmetic	Boolean	Character
Local Scope	**LCLA**	**LCLB**	**LCLC**
Global Scope	**GBLA**	**GBLB**	**GBLC**
Initial Values	0	0	null

- Examples of scalar-variable declarations:

```
LCLA  &J,&K
GBLB  &INIT
LCLC  &Temp_Chars
```

- May be subscripted, in a 1-dimensional array (positive subscripts)

```
LCLA  &F(15)
```

- May be **created**, in the form &(e) (where **e** starts with an alphabetic character)

```
&(B&J&K)   SETA  &(XY&J.Z)-1
```

July 1993 High Level Assembler Tutorial Guide HLASM

Declaring Variable Symbols ... 5

- Three forms of **implicit** declaration:

1. by the Assembler (for **System Variable Symbols**)
   - names always begin with characters **&SYS**
   - most have local scope
2. by appearing as **symbolic parameters** (dummy arguments) in a macro prototype statement
   - symbolic parameters always have local scope
3. as local variables, if first appearance is as target of an assignment

July 1993 High Level Assembler Tutorial Guide HLASM

Declaration

Variable symbols are declared in several ways:

- explicitly, through the use of declaration statements
- by the Assembler (these are known as *System Variable Symbols*)
- implicitly, by their appearance as dummy arguments in a macro prototype statement (these are known as *symbolic parameters*; they are of character type, and are local in scope)
- implicitly, as local variables, through appearing for the first time as the target of an assignment.

Explicitly declared variable symbols are declared using two sets of statements that specify their type and scope:

Figure 1. Explicit Variable Symbol Declarations and Initial Values			
	Arithmetic	**Boolean**	**Character**
Local scope	LCLA	LCLB	LCLC
Global scope	GBLA	GBLB	GBLC
Initial Values	0	0	null

These declared variables are automatically initialized by the Assembler to zero (arithmetic and boolean variables) or to null (zero-length) strings (character variables).

The two scopes of variable symbols — local and global — will be discussed in greater detail later, in "Variable Symbol Scope Rules: Summary" on page 65. For the time being, we will be concerned almost entirely with local variables.

For example, to declare the three local variable symbols &A as arithmetic, &B as boolean, and &C as character, we would write

```
LCLA  &A
LCLB  &B
LCLC  &C
```

More than one variable symbol may be declared on a single statement. The ampersand preceding the variable symbols may be omitted in LCLx and GBLx statements, if desired.

Variable symbols may also be *subscripted*: that is, you may declare a one-dimensional array of variable symbols all having the same name, by specifying a parenthesized integer expression following the name of the variable. For example,

```
LCLA  &F(15)
LCLB  &G(15)
LCLC  &H(15)
```

would declare the three subscripted local variable symbols F, G, and H to have 15 elements. We will see in practice that the declared size of an array is ignored, and any valid (positive) subscript value is permitted. Thus, it is sufficient to declare

```
LCLA  &F(1)
LCLB  &G(1)
LCLC  &H(1)
```

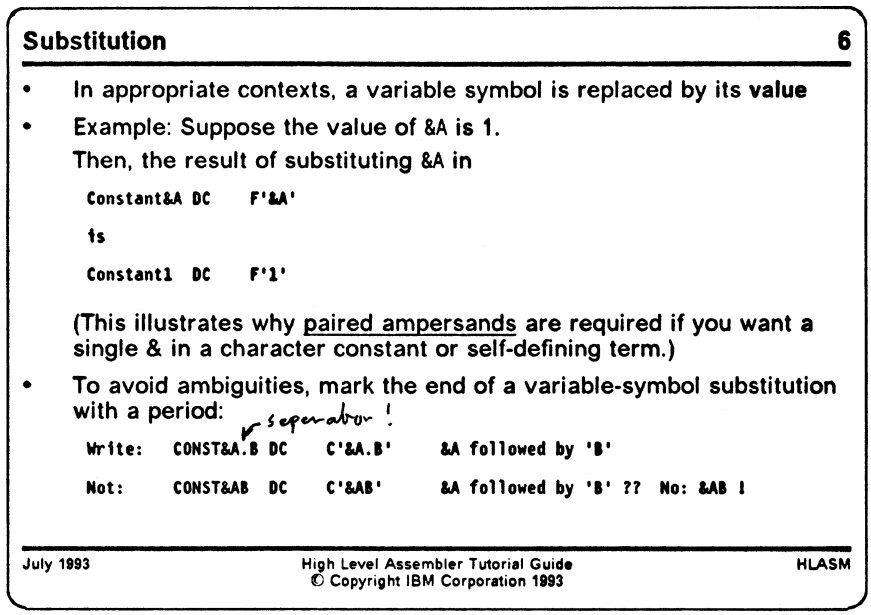
You can determine the maximum subscript used for a subscripted variable symbol with a Number attribute reference (to be discussed later, at "Macro-Instruction Argument Properties: Number Attribute" on page 60).

Subscripted variable symbols may appear anywhere a scalar (non-subscripted) variable symbol appears.

Created variable symbols may be created "dynamically", using characters and the values of other variable symbols. The general form of a created variable symbol is &(**e**), where **e** must (after substitutions) begin with an alphabetic character. Created variable symbols may also be subscripted; like other variable symbols they may be declared explicitly or implicitly.

System variable symbols and their properties are discussed in "System (&SYS) Variable Symbols" on page 129.

In the examples that follow, we will typically enclose string values in apostrophes, as in 'String', to help make the differences clearer between strings and descriptive text. However, the enclosing quotes are only *sometimes* required by the syntax rules of a particular statement or context.



Substitution

The value of a variable symbol is used by *substituting* its value, converted into a character string if necessary, into some element of a statement. For example, if the value of &A is 1 (at this point, it doesn't matter whether &A is an arithmetic, boolean, or character variable), and we write the following DC statement:

```
Constant&A DC    F'&A'
```

then the resulting statement would be

```
Constant1  DC    F'1'
```

That is, at each appearance of the variable symbol &A, *its value is substituted in place of the symbol.* (This behavior explains why you were required to write a pair of ampersands in character constants and self-defining terms where you wanted a single ampersand to appear in the character constant or self-defining term: a single ampersand would indicate to the Assembler that a variable symbol is expected to appear in that position.)

The positions where substitutable variable symbols appear, and at which substitutions are done, are sometimes called *points of substitution.*

Suppose we need to substitute the value of &A into a character constant, such that its value is followed by the character 'B'. If we wrote

```
CONST&AB  DC    C'&AB'      &A followed by 'B' ??
```

the assembler has a problem: should &AB be treated as the variable symbol &AB or as the variable symbol &A followed by 'B'? If the assembler made the latter choice, it could never recognize the variable symbol &AB (nor any other symbols beginning with &A, like &ABCDEFG)!

To eliminate such ambiguities, you should indicate the end of the variable symbol with a period (.). Thus, the constant should be written as

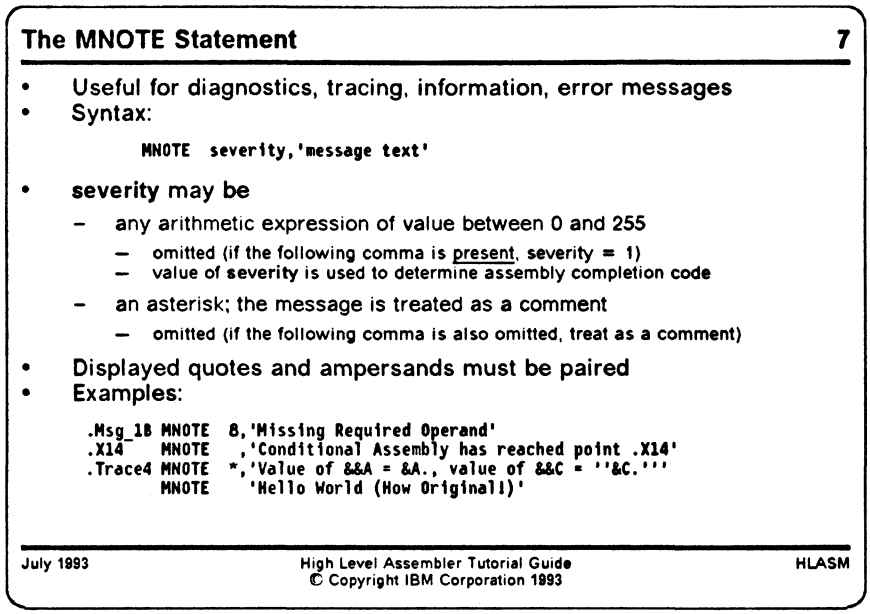
```
CONST&A.B DC    C'&A.B'     &A followed by 'B'
```

giving

```
CONST1B   DC    C'1B'       &A followed by 'B' !!
```

(Note that variable symbols are not substituted in remarks fields or in comments statements.)

While the terminating period is not required in all contexts, it is a good practice to specify it wherever substitution is intended. (The two places where the period most definitely *is* required are when the point of substitution is to be followed by a period or a left parenthesis.)



The MNOTE Statement

The "inputs" to conditional assembly activities are usually values of variable symbols, and ordinary statements that may or may not be affected by substitution and/or selection. Similarly, the "outputs" are normally sequences of statements on which selection and substitution have been performed.

There is another way for the conditional assembly language to "communicate" to the program and the programmer, by way of the MNOTE statement.

The MNOTE statement can be used in both "open code" and in macros to provide diagnostics, trace information, and other data in an easily readable form. By providing suitable controls, you can produce or suppress such messages easily, which facilitates debugging of macros and of programs with complex uses of the conditional assembly language. For example, a program could issue MNOTE statements like the following:

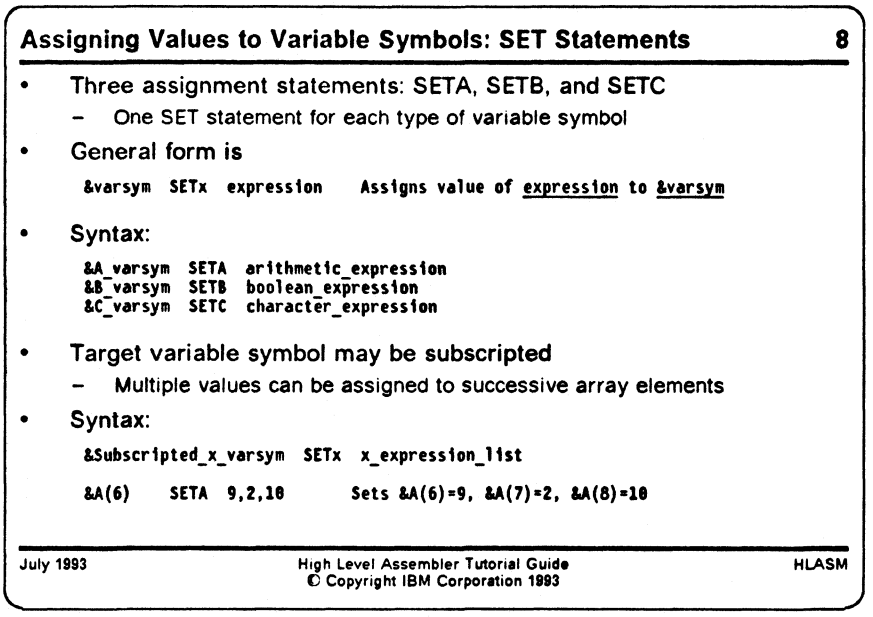
```
.Msg_1B MNOTE  8,'Missing Required Operand'
.X14    MNOTE   ,'Conditional Assembly has reached point .X14'

.Trace4 MNOTE  *,'Value of &&A = &A., value of &&C = ''&C.'''
        MNOTE    'Hello World (How Original!)'
```

The first MNOTE sets the return code for the assembly to be at least 8 (presumably, due to an error condition); the second could indicate that the flow of control in a conditional assembly has reached a particular point (and will supply a default severity code value of 1); the third provides information about the current values of two variable symbols; and the fourth illustrates the creation of a simple message.

Any quotation marks and ampersands intended to be part of the message must be paired, as illustrated in the example above.

The first two MNOTEs are treated as "error" messages, which means that they will be flagged in the error summary in the listing and will appear in the SYSTERM output (if the TERM option was specified, and the setting of the FLAG option has not suppressed them). A

setting of an assembly severity code is also performed. The latter two MNOTEs will be treated as comments, and will appear only in the listing.



Assigning Values to Variable Symbols: SET Statements

Assignment of new values to variable symbols occurs in three ways, corresponding to the types of declaration.

- Explicitly and implicitly declared variable symbols of arithmetic, boolean, and character type are assigned values by the SETA, SETB, and SETC statements, respectively. (Since the type of the assigned variable is generally known in advance, having three separate SET statements is somewhat redundant; it does help, however, by allowing implicit declarations.)
- System variable symbols are assigned values by the Assembler (and only by the Assembler). They may not appear in the name field of a SETx statement.
- Symbolic parameters have their values assigned by appearing as actual arguments in a macro call statement. They may not appear in the name field of a SETx statement.

At this point, we will discuss only assignments to declared variable symbols.

Multiple array elements may have values assigned in a single SET statement by specifying a list of operand-field expressions of the proper type, separated by commas. For example:

```
&A(6)    SETA  9,2,10       Sets &A(6)=9, &A(7)=2, &A(8)=10
```

would assign 9 to &A(6), 2 to &A(7), and 10 to &A(8). (If you wish to leave one of the array elements unchanged, simply omit the corresponding value from the expression list.)

Occasionally, the three declarable types of variable symbol (arithmetic, boolean, and character) are referred to as SETA, SETB, and SETC variables, respectively, and declarable variable symbols are referred to as SET symbols.

Evaluating Conditional-Assembly Expressions

As in any programming language, it is useful to evaluate expressions involving variable symbols and other terms, and to assign the results to other variable symbols.

Divide by zero gives Ø as result

Evaluating and Assigning Arithmetic Expressions 9

- Syntax:

```
&Arithmetic_Var_Sym  SETA  arithmetic_expression
```

- Follows same evaluation rules as ordinary-assembly expressions
  - Simpler, because no relocatable terms are allowed
- Terms include:
  - arithmetic and boolean variable symbols
  - self-defining terms (binary, character, decimal, hexadecimal)
  - character variable symbols whose value is a self-defining term
  - predefined absolute ordinary symbols
  - numeric-valued attribute references (Count, Definition, Integer, Length, Number, Scale)
- Example:

```
&A  SETA  &D*(2+&K)/&G+ABSSYM-C'3'+L'&PL3
```

July 1993 High Level Assembler Tutorial Guide HLASM

Evaluating and Assigning Arithmetic Expressions: SETA

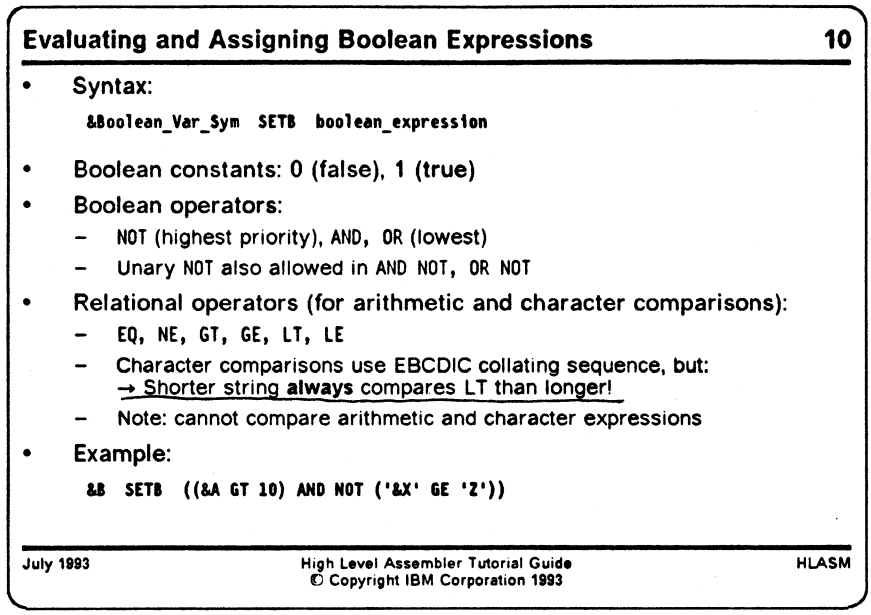
The rules for evaluating conditional-assembly arithmetic expressions are very similar to those for ordinary expressions, with the added great simplification that none of the terms in a conditional-assembly expression may be relocatable. In addition to self-defining terms, predefined absolute ordinary symbols may be used as terms, as may variable symbols whose value can be expressed as a self-defining term (whose value in turn can be represented as a signed 32-bit integer).

As usual, parentheses may be used in expressions to control the order and precedence of evaluation.

Numeric-valued attribute references to ordinary symbols may also be used as terms; these are normally attribute references to character variable symbols whose value is an ordinary symbol. The numeric-valued attribute references are:

- Count (K')
- Definition (D')
- Integer (I')
- Length (L')
- Number (N')
- Scale (S')

We will illustrate applications of attribute references later, particularly when we discuss macros. Attribute references may, of course, be used in "open code".



Evaluating and Assigning Boolean Expressions: SETB

Boolean expressions provide much of the conditional selection capability of the conditional assembly language. In practice, many boolean expressions are not assigned to boolean variable symbols; rather, they are used in AIF statements to describe a condition to control whether or not a conditional-assembly "branch" will or will not be taken.

Boolean primaries include boolean variable symbols, the boolean constants 0 and 1, and (most useful) comparisons. Two types of comparison are allowed: between arithmetic expressions, and between character expressions (which will be described in "Evaluating and Assigning Character Expressions: SETC" on page 13 below). Comparisons between arithmetic and character terms is not allowed.

The comparison operators are

- EQ (equal)
- NE (not equal)
- GT (greater than)
- GE (greater than or equal)
- LT (less than)
- LE (less than or equal)

In an arithmetic relation, the usual integer comparisons are indicated. (Remember that predefined absolute ordinary symbols are allowed as arithmetic terms!)

```
N       EQU  10
&N      SETA 5
&B1     SETB (&N GT 0)      &B1 is TRUE
&B2     SETB (&N GT N)      &B2 is FALSE
```

For character comparisons, a test is first made on the *lengths* of the two comparands: if they are not the same length, the shorter operand is *always* taken to be "less than" the longer. *Note that this may* ***not*** *be what you would get if you did a "hardware" comparison!* The following example illustrates the difference.

```
('BB' GT 'AAA')      is always FALSE in conditional assembly
CLC  =C'BB',=C'AAA'  indicates that the first operand is "high"
```

If the character comparands are the same length, then the usual EBCDIC collating sequence is used for the comparison, so that

```
('BB' GT 'AA')        is always TRUE in conditional assembly
```

The boolean operators are the usual logical operators AND, OR, and NOT; note that no exclusive-or (XOR) operation is provided. (This deficiency can sometimes be remedied by noting that

```
(A XOR B) = (A OR B) - (A AND B)
```

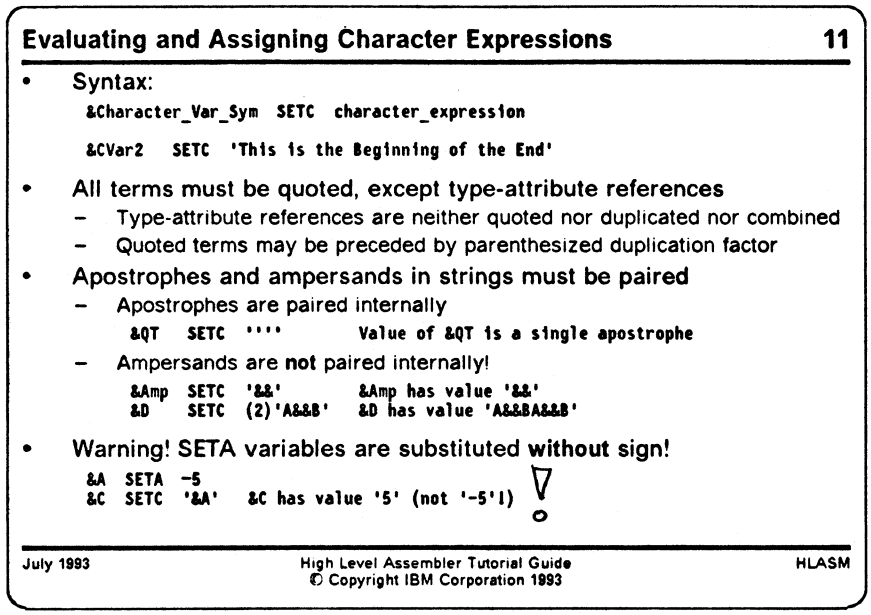
if the terms can readily be cast in the arithmetic form needed for the subtraction.)

NOT may be used as a unary operator, as in the following:

```
&Bool_var  SETB  (NOT ('BB' EQ 'AA'))
```

which would set &Bool_var to 1, meaning TRUE.

In a compound expression involving mixed operators, the NOT operation has highest priority; AND has next highest priority; and OR has lowest priority.



Evaluating and Assigning Character Expressions: SETC

The major elements of character expressions are quoted strings. For example, we may assign values to character variable symbols using quoted strings, as follows:

```
&CVar1   SETC  'AaBbCcDdEeFf'
&CVar2   SETC  'This is the Beginning of the End'
&Digits  SETC  '0123456789'
&Hex     SETC  '0123456789ABCDEF'
```

Type attribute references may also be used as terms in character expressions, but they must appear as the only term in the expression:

```
&TCVar1  SETC  T'&CVar1
```

Character-string constants in SETC expressions are quoted, and internal apostrophes and ampersands must be written in pairs, so that the term may be recognized correctly by the assembler. Thus, character strings in character (SETC) expressions look like character constants and character self-defining terms in other contexts.

However, when the assembler determines the *value* of a character term in a SETC expression, there is one key difference: while apostrophes are paired to yield a single internal apostrophe, ampersands are *not* paired to yield single internal ampersands! Thus, if we assign a string with a pair of ampersands, the result will still contain that pair:

```
&QT    SETC  ''''        Value of &QT is a single apostrophe
&Amp   SETC  '&&'        &Amp has value '&&'
&C     SETC  'A&&B'      &C has value 'A&&B'
&D     SETC  (2)'A&&B'   &D has value 'A&&BA&&B'
```

If the value of such a variable is substituted into an ordinary statement, then the ampersands will be paired to produce a single ampersand, according to the familiar rules of the Assembler Language:

```
&C       SETC  'A&&B'     &C has value 'A&&B'
AandB    DC    C'&C.'     generated constant is 'A&B'
```

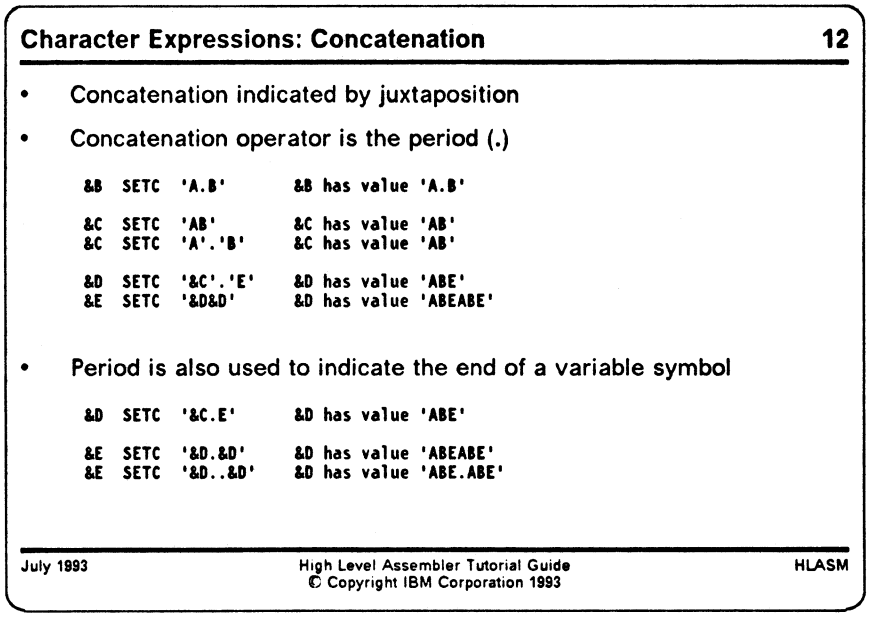
If a single ampersand is required in a character expression, then a substring (described below) of a pair of ampersands should be used.

One reason for this behavior is that it prevents unnecessary proliferation of ampersands. For example, if we had wanted to create the character string 'A&&B', a requirement for paired ampersands in SETC expressions would require that we write

```
&C   SETC  'A&&&&B'   ???
```

which would clearly make the language become even more awkward. The existing rules represent a trade-off between inconvenience and inconsistency, in favor of greater convenience.

Character expressions introduce two new concepts: *string concatenation*, and *substring operations*.

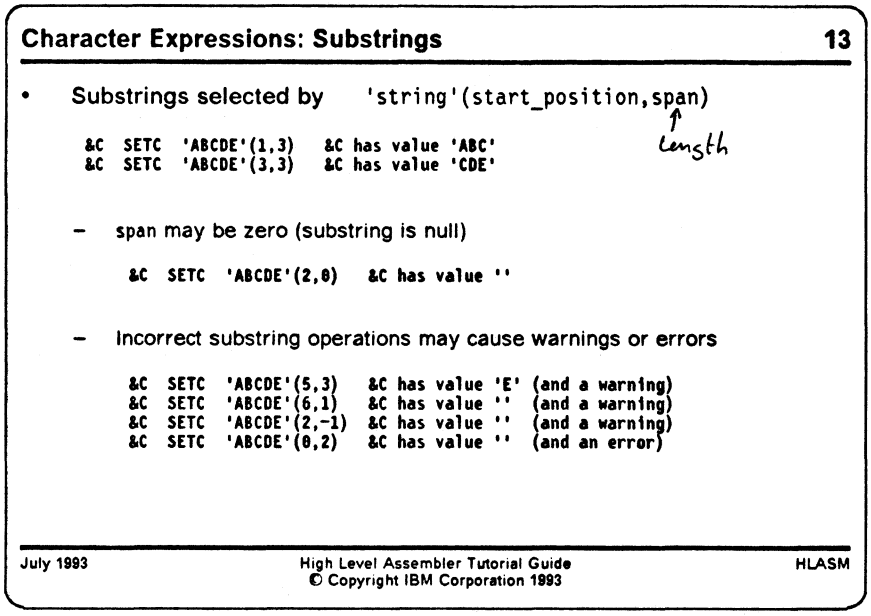


String Concatenation

We are somewhat familiar with the notion of string concatenation from some of the earlier examples of substitution, where a substituted value is concatenated with the adjoining characters to create the completed string of characters. As before, the end of a variable symbol may be denoted with a period. The period is also used as the concatenation operator, as shown in the following examples:

```
&C  SETC  'AB'       &C has value 'AB'
&B  SETC  'A.B'      &B has value 'A.B'
&C  SETC  'A'.'B'    &C has value 'AB'
&D  SETC  '&C'.'E'   &D has value 'ABE'
&D  SETC  '&C.E'     &D has value 'ABE'
&E  SETC  '&D&D'     &D has value 'ABEABE'
&E  SETC  '&D.&D'    &D has value 'ABEABE'
&E  SETC  '&D..&D'   &D has value 'ABE.ABE'
```

As these examples show, there may be more than one way to specify desired concatenation results.



Substrings

Substrings are defined by a somewhat unusual (and sometimes awkward) notation, as follows:

```
substring = 'source_string'(start_position,span)
```

where *start_position* is the position in the *source_string* where the substring is to begin, and *span* is the length of the substring to be extracted.

To illustrate, consider the following examples:

```
&C  SETC  'ABCDE'(1,3)   &C has value 'ABC'
&C  SETC  'ABCDE'(3,3)   &C has value 'CDE'
&C  SETC  'ABCDE'(5,3)   &C has value 'E' (and a warning)
```

So long as the substring is entirely contained within the *source_string*, the results are intuitive. For cases where one or another of the many possible boundary conditions would cause the substring not to be entirely contained within the *source_string*, the following rules apply:

1. The *length* of the *source_string* must be between 1 and 255.
2. The *span* of the *substring* must be between 0 and 255.
3. If $1 \le start\_position \le length$, and $1 \le span \le length$, and $start\_position + span \le length + 1$, then a normal substring will be extracted.
4. If $start\_position \le 0$, then the assembler will issue an error message, and the substring will be set to null.
5. If $start\_position > length$, then the assembler will issue a warning message, and the substring will be set to null.
6. If $span = 0$, then the substring will be set to null. No error message will be issued.

7. If *span*<0, then the assembler will issue a warning message, and the substring will be set to null.

8. If *start_position*+*span*>*length*+1, then the substring will be that portion of the *source_string* starting at *start_position* to the end. The assembler will issue a warning message.

Unfortunately, there is no substring notation meaning "from here to the end of the string", which some other languages support.

String expressions are constructed using the operations of substitution, concatenation, and substringing. One may also use type attribute references as character terms, but they are limited to "single-term" expressions with no duplication factors.

Be aware that substitution of arithmetic-valued variable symbols into character (SETC) expressions will *not* preserve the sign of the arithmetic value! For example:

```
&A  SETA  -5
&C  SETC  '&A'   &C has value '5' (not '-5'!)
```

If signed arithmetic is important, use arithmetic expressions and variable symbols; if signed valued must be substituted into ordinary statements with the proper sign, then you must construct a character variable with the desired sign, as in the following example. (Uses of the AIF and ANOP statements, and the sequence symbol .GenCon will be discussed shortly.)

```
&A        SETA  -5
BadCon1   DC    F'&A'              Constant has value 5
&C        SETC  '&A'               &C has value '5' (not '-5'!)
BadCon2   DC    F'&C'              Constant has value 5
          AIF   (&A GE 0).GenCon   Check sign of &A
&C        SETC  '-'.'&C'           Prefix minus sign if negative
.GenCon   ANOP
GoodConst DC    F'&C'              Correctly signed constant with value -5
```

Character Expressions: String Lengths **14**

- Use a Count Attribute Reference (K') to determine string lengths

```
&N    SETA  K'&C    Sets &A to number of characters in &C

&C    SETC  '12345'     &C has value '12345'
&N    SETA  K'&C        &N has value 5

&C    SETC  ''          null string
&N    SETA  K'&C        &N has value 0

&C    SETC  (3)'AB'     &C has value 'ABABAB'
&N    SETA  K'&C        &N has value 6
```

July 1993 High Level Assembler Tutorial Guide HLASM

String Lengths

The number of characters in a character variable symbol can be determined using a Count attribute reference (K'). For example:

```
&C    SETC  '12345'          &C has value '12345'
&N    SETA  K'&C             &N has value 5

&C    SETC  ''               null string
&N    SETA  K'&C             &N has value 0

&C    SETC  (3)'AB'          &C has value 'ABABAB'
&N    SETA  K'&C             &N has value 6
```

The Count attribute reference is very useful in cases where strings must be scanned from right to left; thus,

```
&X    SETC  '&C'(K'&C,1)   Extract rightmost character of &C
```

assigns the rightmost character in the value of &C to &X.

Comments on Substitution, Evaluation, and Re-Scanning

The assembler uses a method of identifying points of substitution that may be different from the methods used in some other languages.

1. Points of substitution are identified *only* by the presence of variable symbols. Ordinary symbols (or other strings of text) are never substituted.

2. Statements are scanned only once to identify points of substitution. This means that if a substituted value seems to cause another variable symbol to "appear" (possibly suggesting further points of substitution), these "secondary" substitutions will not be performed.

   To illustrate, you might ask what happens in this situation: will the substituted value of &B in the DC statement be substituted again?

   ```
   &C       SETC   '&&B'        &C has value '&&B'
   &C       SETC   '&C'(2,2)    &C has value '&B'
   &B       SETC   'XXX'        &B has value 'XXX'
   Con      DC     C'&C'        Is the result '&B' or 'XXX'?
   ```

   The answer is "no". In fact, this DC statement results in an error message:

   ```
   ASMA127S *** ERROR *** ILLEGAL USE OF AMPERSAND
   ```

   Because the assembler does not re-scan the DC statement to attempt further substitutions for &C, there will be a single ampersand remaining in the nominal value ('&B') of the C-type constant.

   As a further example, note that substitution uses a left-to-right scan, and that new variable symbols are not created "automatically". For example, if the two character variable symbols &C1 and &C2 have values 'X' and 'Y' respectively, then the substituted value of '&C1&C2' is 'XY', and not the value of '&C1Y'. Similarly, the string '&&C1.C2' represents '&&C1.C2', and *not* the value of '&XC2'!

   The only mechanism for "manufacturing" variable symbols is that of the created variable symbol, whose recognition requires the specific syntax previously described.

3. This single-scan rule applies both to ordinary-statement substitutions, and to conditional-assembly statements. Thus, statements once scanned for points of substitution will not be re-scanned (or "re-interpreted") further.

Consider the arithmetic expression '5*&A'. We would expect it to be evaluated by substituting the value of &A, and then multiplying that value by 5.

If this is used in statements such as

```
&A      SETC    '10'
&B      SETA    5*&A
```

then we would find that &B has the expected value, 50. However, in the following statements:

```
&A      SETC    '3+4'
&B      SETA    5*&A
```

we are faced with several possibilities. First, is the value of &B now 35 (corresponding to "5*(3+4)")? That is, is the sum 3+4 evaluated before the multiplication? Second, is the value of &B now 19 (corresponding to "(5*3)+4")? That is, is the string "5*3+4" evaluated according to the rules for arithmetic expressions?

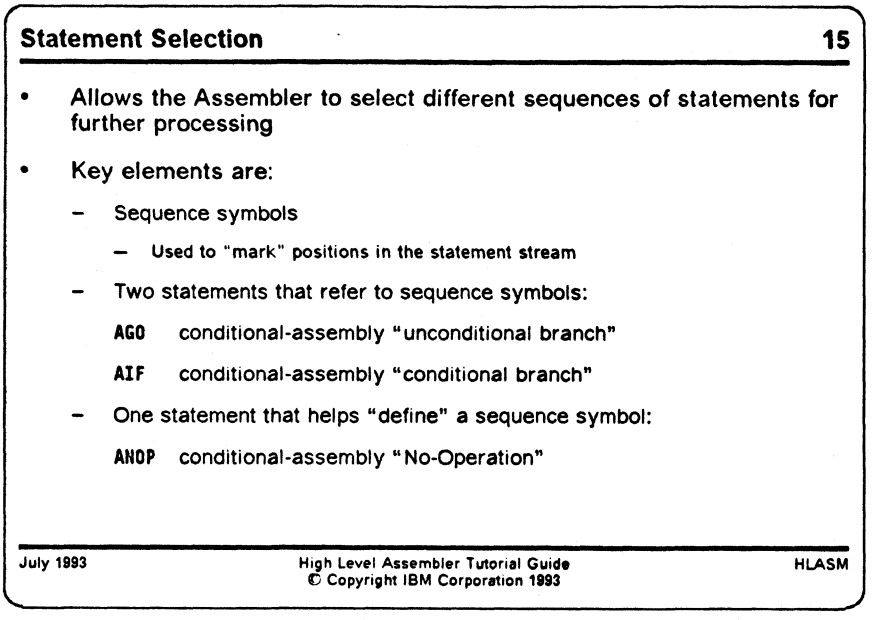
In fact, a third situation occurs: because the expression '5*&A' is *not* re-scanned in any way, the value of &A must be a self-defining term. Because it is not, the assembler produces this error message:

```
ASMA102E *** ERROR *** Arithmetic term is not self-defining term; default = 0
```

indicating that the substituted "term" 3+4 is improperly formed.

A similar result occurs if predefined absolute symbols are used as terms. If they are used directly (without substitution), they are valid; however, the name of the symbol may not be substituted as a character string. To illustrate:

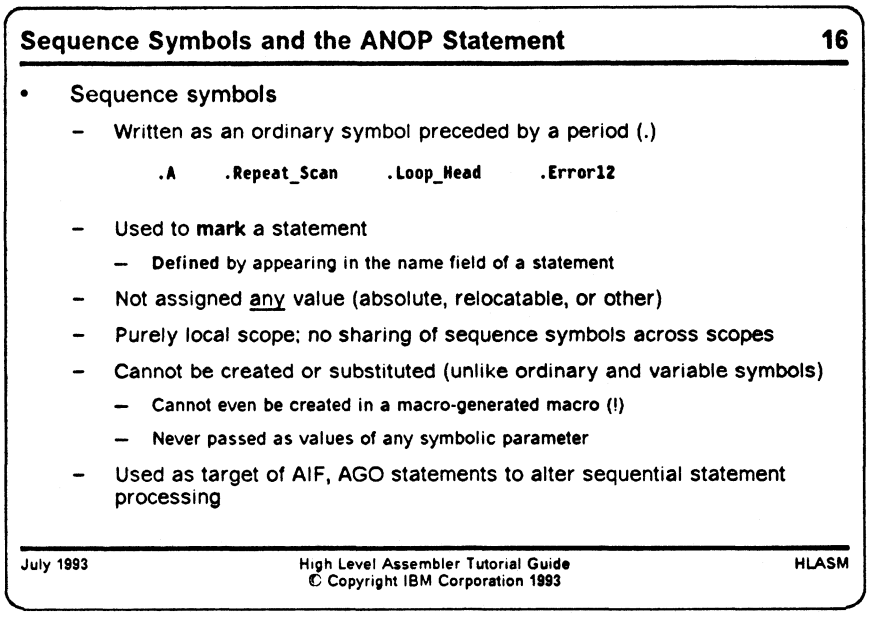
```
N     Equ   3+4          N has value 7
&B    SetA  5*N          &B has value 35

&N    SetC  'N'          Set &N to the character 'N'
&C    SetA  5*&N         Error message for invalid term!
```

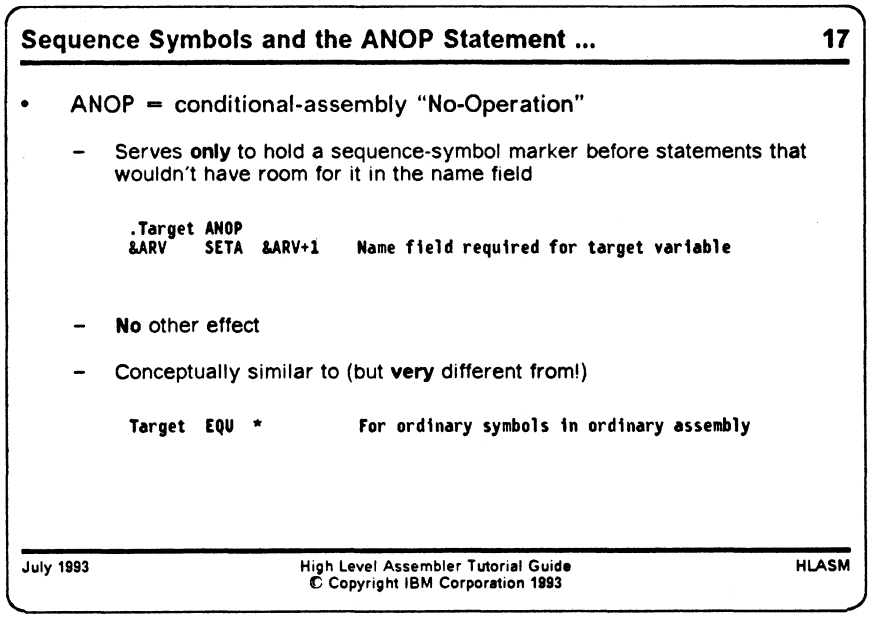


Statement Selection

The full power of the conditional assembly language lies in its ability to direct the Assembler to *select* different sequences of statements for processing. This allows you to tailor your program in many different ways, as we will see.

The key facilities required for statement selection are *sequence symbols*, which are used to mark positions in the statement stream for reference by other statements, and the AIF and AGO statements, which allow the normal sequence of statement processing to be altered, based on conditions specified by the programmer. The ANOP statement is provided as a "place holder" for a sequence symbol.





Sequence Symbols and the ANOP Statement

Sequence symbols are the key to statement selection: they "mark" the position of a specific statement in the stream of statements to be processed by the assembler. They are written as an ordinary symbol preceded by a period (.), as in the following examples:

```
.A      .Repeat_Scan      .Loop_Head      .Error12
```

Sequence symbols have some unusual properties compared to ordinary symbols.

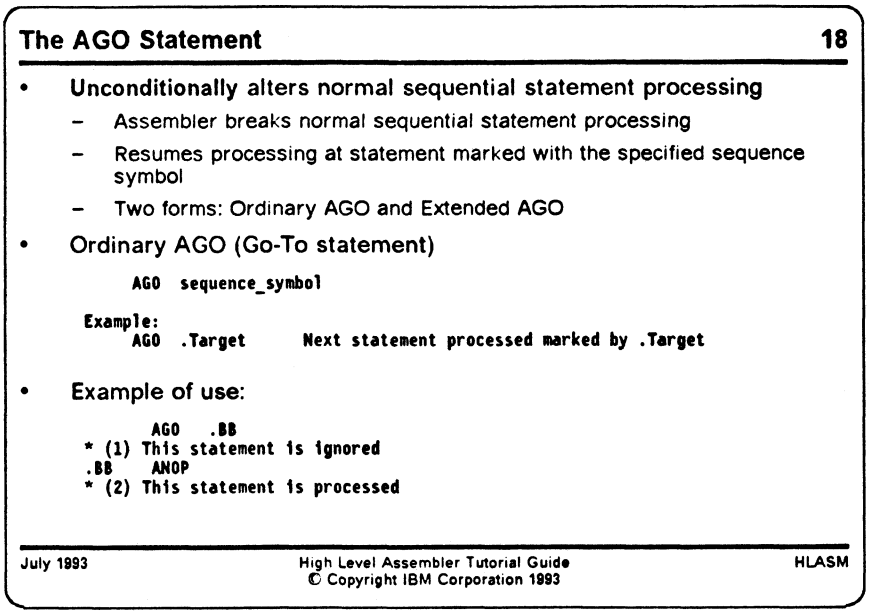
- Sequence symbols are defined by appearing in name field of any statement. They may appear on ordinary-assembly statements and on conditional-assembly statements, with no difference in meaning or behavior.

- Sequence symbols are not assigned an absolute or relocatable value, and they do not appear in the assembler's Symbol Table. They cannot be used in expressions of any kind.
- Sequence symbols have purely local scope. That is, there is no sharing of sequence symbols between macros, or between macros and ordinary "open code" assembly.
- Sequence symbols cannot be created or substituted (unlike ordinary and variable symbols).
- Sequence symbols are never passed as values of any symbolic parameter. Thus, although they can appear in the name field of a macro instruction statement (or macro "call"), they are never made available to the macro definition as the value of a name-field variable symbol.
- Sequence symbols are used as the target of AIF and AGO statements to alter sequential statement processing, and for no other purpose.

The ANOP statement is provided as a "place holder" for a sequence symbol that could not otherwise be attached to a desired statement. This is illustrated in the following example, where the desired "target" is a SETA statement, which requires that an arithmetic variable symbol appear in the name field:

```
.Target ANOP
&ARV    SETA  &ARV+1    Name field required for target variable
```

Thus, the ANOP statement provides a way for other AIF and AGO statements to refer to the SETA statement.



The AGO Statement

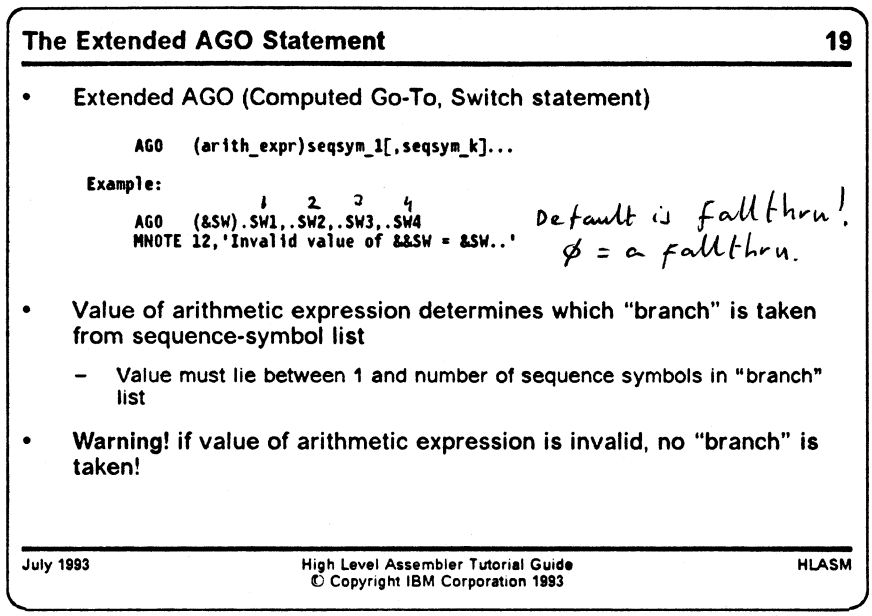
The function of the AGO statement is to unconditionally alter the sequence of statement processing, which resumes at the statement "marked" with the specified sequence symbol. It is written in the form

```
      AGO  sequence_symbol

  Example:
      AGO  .Target       Next statement processed marked by .Target
```

The Assembler breaks its normal sequential statement processing, and resumes processing at the statement "marked" with the specified sequence symbol. For example,

```
        AGO   .BB
* (1) This statement is ignored
.BB     ANOP
* (2) This statement is processed
```

the AGO statement will cause the following comment statement (1) to be skipped, and processing will resume at the ANOP statement.



The Extended AGO Statement

The assembler provides a convenient extension to the simple imperative (unconditional) AGO statement, in the form of the "Computed AGO" statement, analogous to a "switch" or "case" statement in other languages. The operand field contains a parenthesized arithmetic expression, followed by a list of sequence symbols, as shown in the following example.

```
AGO   (arith_expr)seqsym_1[,seqsym_k]...
```

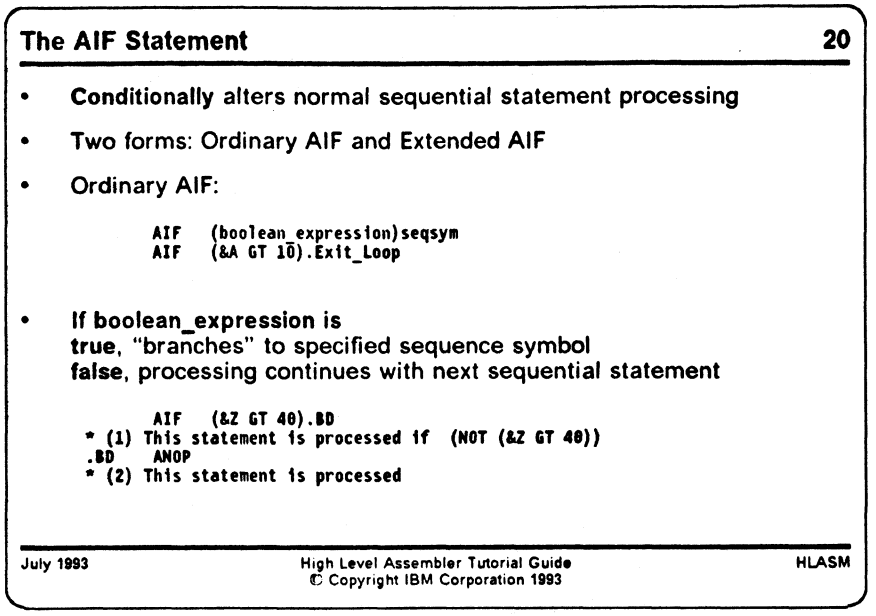
Figure 2. General Form of the Extended AGO Statement

The operation of this extended AGO statement is simple: the value of the *arithmetic_expression* is used to select one of the sequence symbols as a "branch target": if the value is 1, the first sequence symbol is selected; if the value is 2, the second sequence symbol is selected; and so forth. However, because it is possible that the value of the arithmetic expression does not correspond to *any* entry in the list (e.g., the value of the expression may be less than or equal to zero, or larger than the number of sequence symbols in the list), the assembler will not take *any* branch, and *will not issue any diagnostic message about the "failed" branch!* Thus, it is important to verify that the values of arithmetic expressions used in extended AGO statements are always valid.

The operation of the extended AGO statement illustrated in Figure 2 is precisely equivalent to the following set of AIF statements (which will be described shortly):

```
AIF  (arith_expr EQ 1)seqsym_1
AIF  (arith_expr EQ 2)seqsym_2
- - -
AIF  (arith_expr EQ k)seqsym_k
```

This construction helps to illustrate how and when it is possible for no "branch" to be taken.



The AIF Statement

The AIF statement provides a method for conditionally selecting a sequence of statements, by testing a condition before deciding to "branch" or not to the statement designated by a specified sequence symbol. The ordinary AIF statement is written in this form:

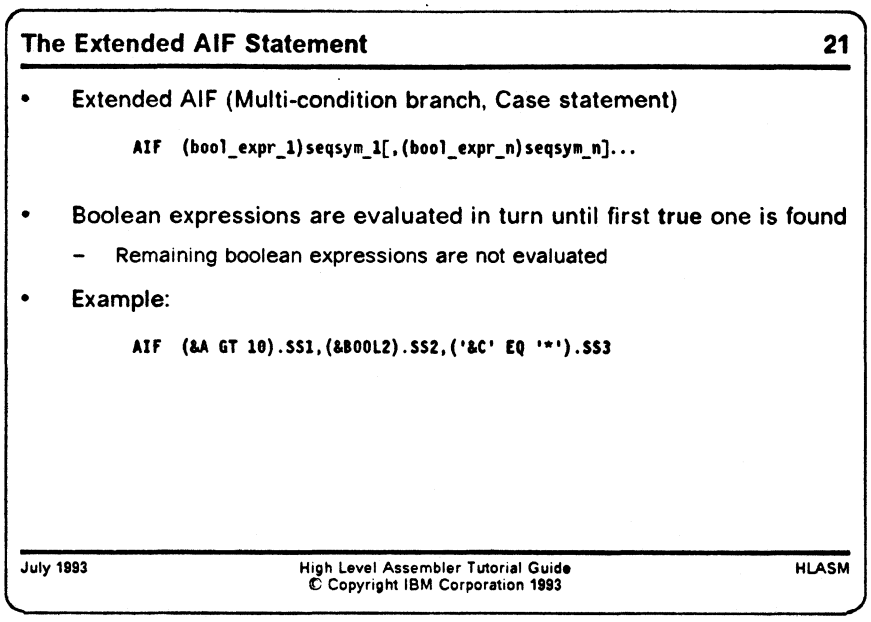
```
     AIF  (boolean_expression)seqsym
```

```
Example:
     AIF  (&A GT 10).Exit_Loop
```

If the "boolean_expression" is true, statement processing will continue at the statement marked with the specified sequence symbol. If the "boolean_expression" is false, processing continues with the next sequential statement following the AIF. For example:

```
       AIF   (&A GT 10).BD
* (1) This statement is processed if   (NOT (&A GT 10))
.BD    ANOP
* (2) This statement is processed
```

In this case, the statement following the AIF will be processed if the boolean expression (&A GT 10) is false; if the condition defined by the boolean condition is true, the next statement to be processed will be the ANOP statement.



The Extended AIF Statement

The extended, or multi-condition, form of the AIF statement allows you to write multiple conditions and "branch" targets on a single statement, as shown in the following:

```
AIF  (bool_expr_1)seqsym_1[,(bool_expr_n)seqsym_n]...
```

Figure 3. General Form of the Extended AIF Statement

The boolean expressions are evaluated in turn until the first **true** expression is found; the next statement processed will be the one "marked" by the corresponding sequence symbol. The remaining boolean expressions are not evaluated after the first true expression is found.

An example of an extended AIF statement is:

```
AIF  (&A GT 10).SS1,(&BOOL2).SS2,('&C' EQ '*').SS3
```

The extended AIF statement illustrated in Figure 3 is entirely equivalent to the following sequence of ordinary AIF statements:

```
AIF  (bool_expr_1)seqsym_1
AIF  (bool_expr_2)seqsym_2
- - -
AIF  (bool_expr_n)seqsym_n
```

The primary advantage of the extended AIF statement is in providing a concise notation for what would otherwise require multiple AIF statements.

Examples of Conditional Assembly

We will now describe some simple examples of open-code conditional assembly. Further examples of conditional assembly techniques will be illustrated later, when we discuss macros.

Example: Generate a Byte String with Values 1-N 22

- Sample 0: write everything by hand

```
N       EQU   5               Predefined absolute symbol
        DC    AL1(1,2,3,4,N)  Define the constants
```

  - Defect: if the value of N changes, must rewrite the DC statement

- Sample 1: generate separate statements

```
N       EQU   5               Predefined absolute symbol
        LCLA  &J              Local arithmetic variable symbol
.Test   AIF   (&J GE N).Done  Test for completion (N could be LE 0!)
&J      SETA  &J+1            Increment &J
        DC    AL1(&J)         Generate a byte
        AGO   .Test           Go to check for completion
.Done   ANOP                  Generation completed
```


Example: Generate a Byte String with Values 1-N ... 23

- Sample 2: generate a single string

```
N       EQU   5               Predefined absolute symbol
        LCLA  &K              Local arithmetic variable symbol
        LCLC  &S              Local character variable symbol
&K      SETA  1               Initialize counter
        AIF   (&K GT N).Done2 Test for completion (N could be LE 0!)
&S      SETC  '1'             Initialize string
.Loop   ANOP                  Loop head
&K      SETA  &K+1            Increment &K
        AIF   (&K GT N).Done1 Test for completion
&S      SETC  '&S'.',&K'      Continue string
        AGO   .Loop           Branch back to check for completed
.Done1  DC    AL1(&S.)        Generate the byte string
.Done2  ANOP                  Generation completed
```

- Try it with "N EQU 30" ... What happens to the DC statement?
- Try it with "N EQU 90" ... What happens?

Generate a Sequence of Byte Values

Suppose we wish to generate DC statements defining a sequence of byte values from 1 to N, where N is a predefined value. This could naturally be done by writing statements like

```
N       EQU   12
        DC    AL1(1,2,3,...,N)
```

but this requires knowing the exact value of N every time the program is modified and re-assembled.

Conditional assembly techniques can be used to solve this problem so that changing the EQU statement defining N will not require any rewriting. We can generate the sequence of DC statements as follows:

```
N       EQU   5               Predefined absolute symbol
        LCLA  &J              Local arithmetic variable symbol
.Test   AIF   (&J GE N).Done  Test for completion (N could be LE 0!)
&J      SETA  &J+1            Increment &J
        DC    AL1(&J)         Generate a byte
        AGO   .Test           Go to check for completion
.Done   ANOP                  Generation completed
```

Figure 4. Generating a Sequence of Bytes, Individually Defined

The operation of this loop is simple. The LCLA declaration of &J also initializes it to zero (we could not have omitted the declaration in this example, because the first appearance of &J is not in a SETA statement). The AIF statement compares &J to N (a predefined absolute symbol), and if it exceeds N, a "branch" is taken to the label .Done. (In fact, the Assembler implements the "branch" by searching the source file for an occurrence of the sequence symbol in the local context of "open code".) If the AIF test does not change the flow of statement processing, the next statement increments &J by one, and its new value is then substituted in the DC statement. The following AGO then returns control to the test in the AIF statement.

Alternatively, we could generate only a single DC statement by using a technique that constructs the nominal value string for the DC statement, like the following:

```
N       EQU   5               Predefined absolute symbol
        LCLA  &K              Local arithmetic variable symbol
        LCLC  &S              Local character variable symbol
&K      SETA  1               Initialize counter
        AIF   (&K GT N).Done2 Test for completion (N could be LE 0!)
&S      SETC  '1'             Initialize string
.Loop   ANOP                  Loop head
&K      SETA  &K+1            Increment &K
        AIF   (&K GT N).Done1 Test for completion
&S      SETC  '&S'.',&K'      Continue string
        AGO   .Loop           Branch back to check for completed
.Done1  DC    AL1(&S.)        Generate the byte string
.Done2  ANOP                  Generation completed
```

Figure 5. Generating a Sequence of Bytes, as a Single Operand String

In this program fragment, a single character string is constructed with the desired sequence of values separated by commas. The first SETC statement sets the local character variable symbol &C to '1', and the following loop then concatenates successive values of the arithmetic variable symbol &K onto the string with a separating comma, on the right. When the loop is completed, the DC statement inserts the entire string of numbers into the nominal values field of the AL1 operand.

It is instructive to test this example with values of N large enough to cause the string &S to become longer than (say) 60 characters; try assigning a value of 30 to N, and observe what the assembler does with the generated DC statement. (Answer: it creates a continuation automatically!)

Both these examples share a shortcoming: if more than one such sequence of byte values is needed in a program, with different numbers of elements in each sequence, these "blocks" of conditional assembly statements must be repeated. We will see in "Generating a Byte Sequence" on page 77 that a simple macro definition can make this task easier to solve.

Example: System-Dependent I/O Statements 24

- Suppose a module declares I/O blocks for MVS, CMS, and VSE:

```
&OpSys   SETC  'MVS'              Set desired operating system
         - - -
         AIF   ('&OpSys' NE 'MVS').T1   Skip if not MVS
Input    DCB   DDNAME=SYSIN,...etc...   Generate MVS DCB
         - - -
         AGO   .T4
.T1      AIF   ('&OpSys' NE 'CMS').T2   Skip if not CMS
Input    FSCB  ,LRECL=80,...etc...      Generate CMS FSCB
         - - -
         AGO   .T4
.T2      AIF   ('&OpSys' NE 'VSE').T3   Skip if not VSE
Input    DTFCD LRECL=80,...etc...       Generate VSE DTF
         - - -
         AGO   .T4
.T3      MNOTE 8,'Unknown &&OpSys value ''&OpSys''.'
.T4      ANOP
```

- Setting of &OpSys selects statements for running on **one** system
  - Assemble the module with a system-specific macro library

July 1993 High Level Assembler Tutorial Guide HLASM

Generating System-Dependent I/O Statements

Suppose you are writing a module that provides operating system services to a larger application. As a simple example, suppose one portion of the module must read input records, and that you wish to use the appropriate system-interface macros for each of the System/360/370/390's MVS, CMS, and VSE operating systems.

This is very simply solved using conditional-assembly statements to select the sequences appropriate to the system for which the module is intended. Suppose you have defined a character-valued variable symbol &OpSys whose values may be MVS, CMS, or VSE. Then the needed code sequences might be defined as in Figure 6 on page 29:

```
&OpSys   SETC  'MVS'              Set desired operating system
         - - -
         AIF   ('&OpSys' NE 'MVS').T1   Skip if not MVS
Input    DCB   DDNAME=SYSIN,...etc...   Generate MVS DCB
         - - -
         AGO   .T4
.T1      AIF   ('&OpSys' NE 'CMS').T2   Skip if not CMS
Input    FSCB  ,LRECL=80,...etc...      Generate CMS FSCB
         - - -
         AGO   .T4
.T2      AIF   ('&OpSys' NE 'VSE').T3   Skip if not VSE
Input    DTFCD LRECL=80,...etc...       Generate VSE DTF
         - - -
         AGO   .T4
.T3      MNOTE 8,'Unknown &&OpSys value ''&OpSys''.'
.T4      ANOP
```

Figure 6. Generating a Sequence of Bytes, as a Single Operand String

In this example, different blocks of code contain the necessary statements for particular operating environments. In any portion of the program that contains statements particular to one of the environments, conditional assembly statements allow the assembler to select the correct statements. By setting a single variable symbol &OpSys to an appropriate value, you can tailor the application to a chosen environment without having to make into multiple copies of its processing logic, one for each environment.

Thus, for example, the first AIF statement tests whether the variable symbol &OpSys has value 'MVS'; if so, then the following statements generate an MVS Data Control Block. (Naturally, you will need to supply an appropriate macro library to the assembler at assembly time!)

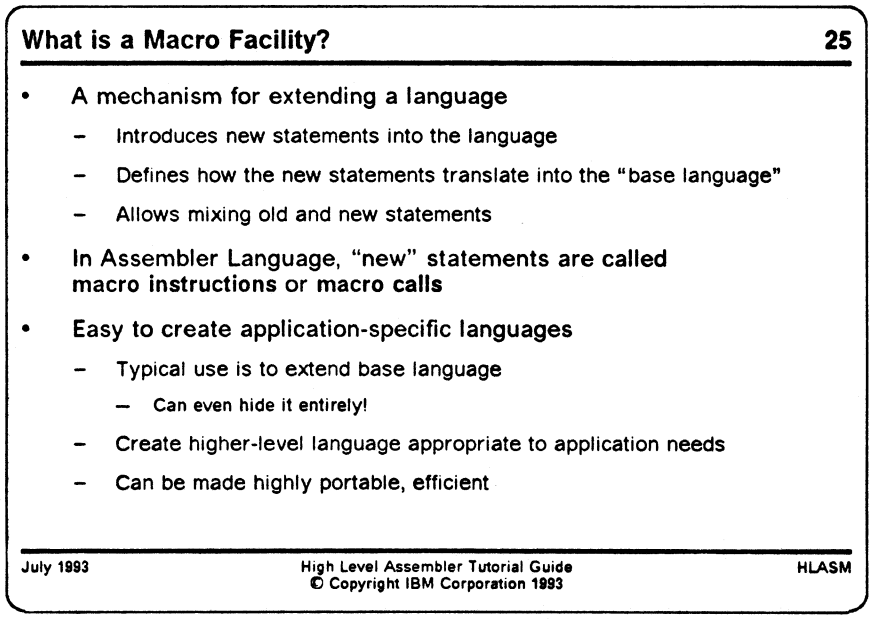
The technique illustrated here allows you to make your programs more portable across operating environments, without requiring major rewriting efforts or duplicated coding each time some new function is to be added.

Macros

Macros are a powerful mechanism for enhancing any language, and they are a very important part of the System/360/370/390 Assembler Language. Macros are widely used in many ways to simplify programming tasks.

We will begin our discussion with a conceptual overview of the basic concepts of macros, in a way that is not specific to the Assembler Language.[1] This will be followed by an investigation of the System/360/370/390 Assembler Language's implementation of macros, including the following topics:

- macro definition: how to define a macro
- macro encoding: how the assembler converts the definition into an internal format to simplify interpretation and expansion
- macro-instruction recognition: how the assembler identifies a macro call and its elements
- macro parameters and arguments
- macro expansion
- macro argument attributes and structures
- global variable symbols
- examples of macros.



[1] Some of the material in this chapter is based on an excellent overview article by William Kent, titled "Assembler-Language Macroprogramming: A Tutorial Oriented Toward the IBM 360" in the *ACM Computing Surveys*, Vol. 1, No. 4 (December 1969), pages 183-196.

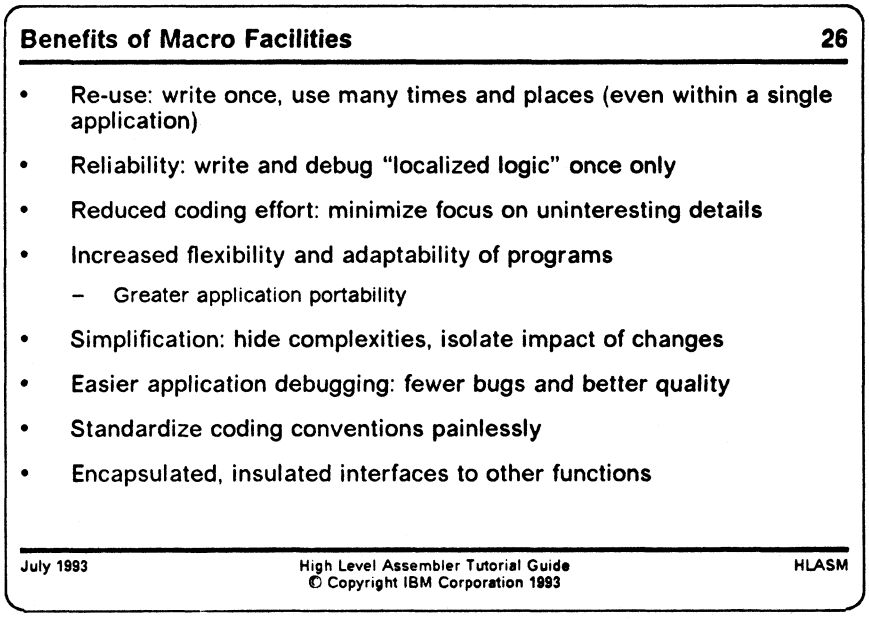
What is a Macro Facility?

Most simply, a macro facility is a mechanism for extending a language. It can be used not only to introduce new statements into the language, but also to define how the new statements should be translated into the "base language" on which they are built. One major advantage of macros is that they allow you to mix "old" (existing) and "new" statements, so that your language can grow incrementally to accommodate new functions, added requirements, and other benefits as and when you are able to take advantage of them.

In the Assembler Language, these new statements are called "macro instructions" or "macro calls". The use of the term "call" implies a useful analogy to subroutines; there are many parallels between (assembly-time) macro calls and (run-time) subroutine calls.

Macros and macro techniques make it very easy to create application-specific languages:

- you can create higher-level languages appropriate to the needs of particular application areas
- the language can be made highly portable and efficient
- typical uses are to extend the base language on which the extended language is built (in fact, it is possible to hide the base language entirely!).



Benefits of Macro Facilities

Macro facilities can provide you with many direct and immediate benefits:

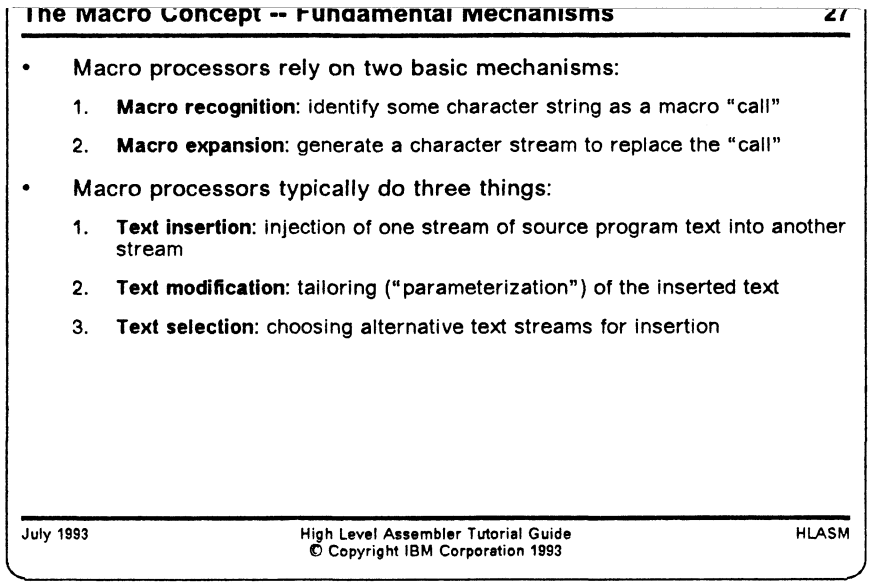
- Code re-use: once a macro is written, it becomes available to as many programmers and applications as are appropriate. A single definition can find multiple uses (even within a single application).
- Reliability: code and debug the logic in one place.
- Reduced coding effort: the coding in a macro needs to be written only once, and then can be used in many places.

ments of your programming language, relieving you of the need to be concerned with details that are typically only marginally relevant to your programming task.

- Increased flexibility and adaptability of programs: you can adapt your applications to different requirements by modifying only the macro definitions, without having to revise the fundamental logic of the program.
- Greater application portability: because almost every system supports a macro assembler, it is easy to port an application written in "macro language" to another host environment simply by writing an appropriate set of macros definitions on the new system.[2]
- Easier debugging, with fewer bugs and better quality: once you have debugged your macros, you can write your applications using their higher-level concepts and facilities, and then debug your programs at that higher level. Concerns with low-level details are minimized, because you are much less likely to make simple oversights among masses of uninteresting details.
- Standardize coding conventions painlessly: if your organization requires that certain coding conventions be followed, it is very simple to embody them in a set of macros that all programmers can use. Then, if the conventions need to change, only one set of objects – the macros – needs to be changed, not the entire application suite.
- Provide encapsulated interfaces to other functions, insulated from interface changes: using macros, you can support interfaces among different elements of your applications, and between applications and operating environments, in a controlled and defined way. This means that changes to those interfaces can be made in the macros, without affecting the coding of the applications themselves.
- Localized logic: specific and detailed (and often complex) code sequences can be implemented once in a macro, and used wherever needed, without the need for every user of the macro to understand the "inner workings" of the macro's logic.

---

[2] The SNOBOL4 language was implemented entirely in terms of a set of macros that defined a "string processing implementation language". The entire SNOBOL4 system could be "ported" to a new system with what the authors called "about a week of concentrated work by an experienced programmer". You may be interested in consulting *The Macro Implementation of SNOBOL4*, by Ralph Griswold.



The Basic Macro Concept

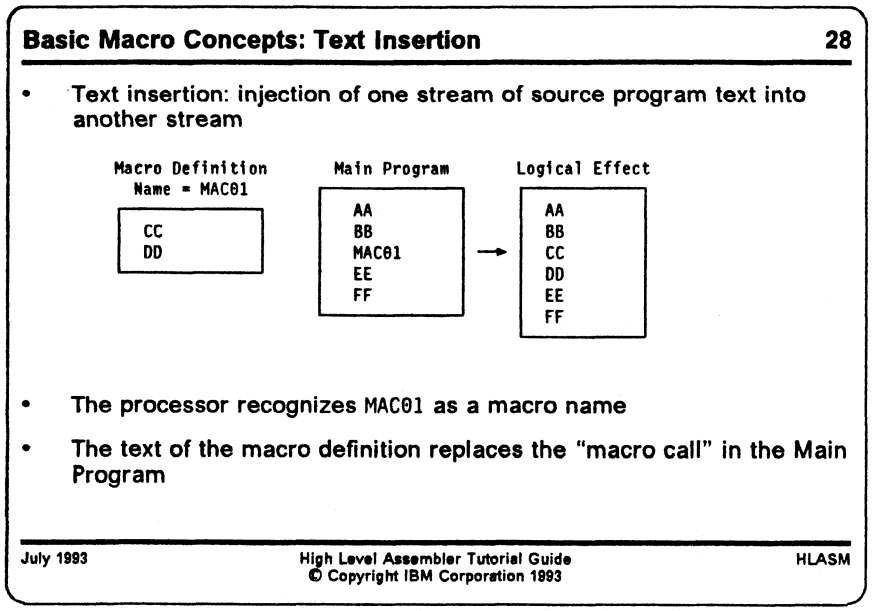
Macro processors typically rely on two basic processes:

- *Macro recognition* requires that the processor identify some string of characters as a macro invocation or macro call, indicating that the string is to be replaced.
- *Macro expansion* or *macro generation* causes the macro definition to be interpreted by the processor, with the usual result that the original string is replaced with a new (and presumably different) string.

In macro expansion, there are three fundamental mechanisms used by almost all macro processors:

- text insertion: the creation of a stream of characters to replace the string recognized in the macro "call"
- text parameterization: the tailoring and adaptation of the generated stream to the conditions of the particular call
- text selection: the ability to generate alternative streams of characters, depending on various conditions available during macro expansion.

These correspond to the mechanisms already described for the conditional assembly language: for example, text parameterization uses the process of substitution, and text selection uses that of statement selection.



Text Insertion

The simplest and most basic mechanism of macro processing is that of replacing a string of characters, or one or more statements, by other (often longer and more complex) strings or sets of statements.

In Figure 7, a set of statements has been defined to be a macro with the name MAC01. When the processor of the Main Program recognizes the string MAC01 as matching that of the macro, that string is *replaced* by the text within the macro definition.

This is called *text insertion*: the injection of one stream of source text into another stream.

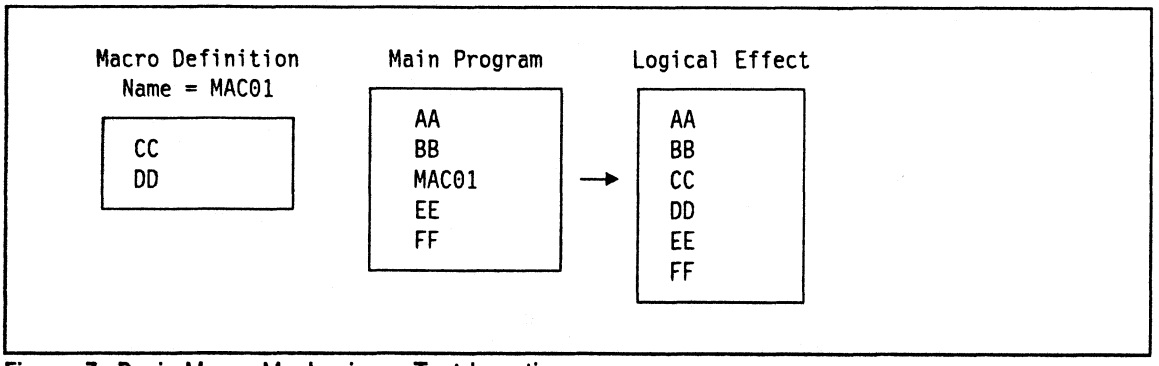
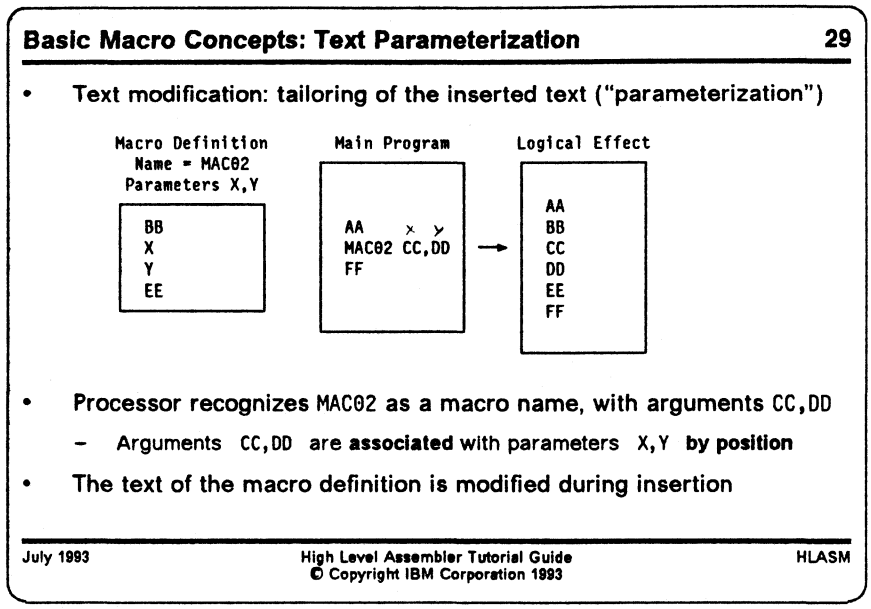


Figure 7. Basic Macro Mechanisms: Text Insertion



Text Parameterization and Argument Association

Simple text insertion has rather limited uses, because we usually want to tailor and adapt the inserted text to accommodate the various conditions and situations of each macro invocation. The simplest form of such adaptation is "text parameterization". In Figure 8, the macro with name MAC02 is defined with two *parameters* X and Y: that is, they are merely place-holders in the definition that indicate where other text strings are expected to be inserted when the macro is expanded.

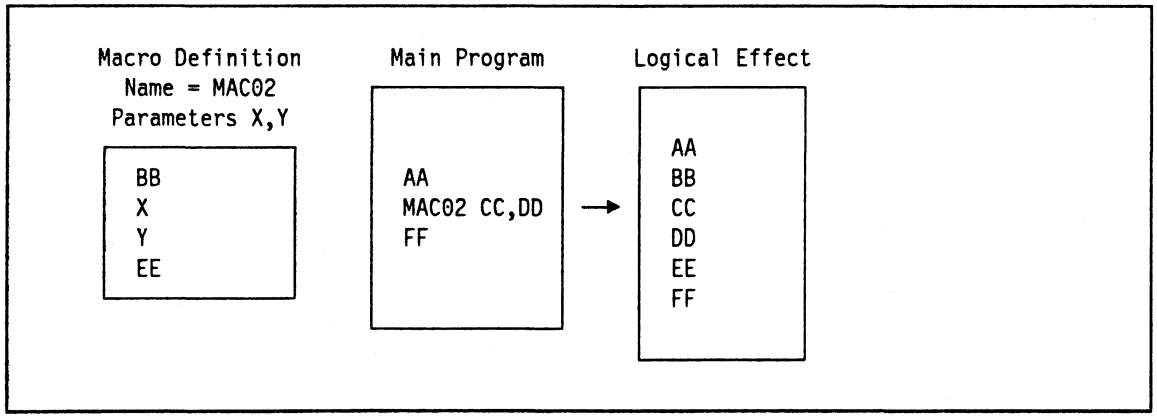


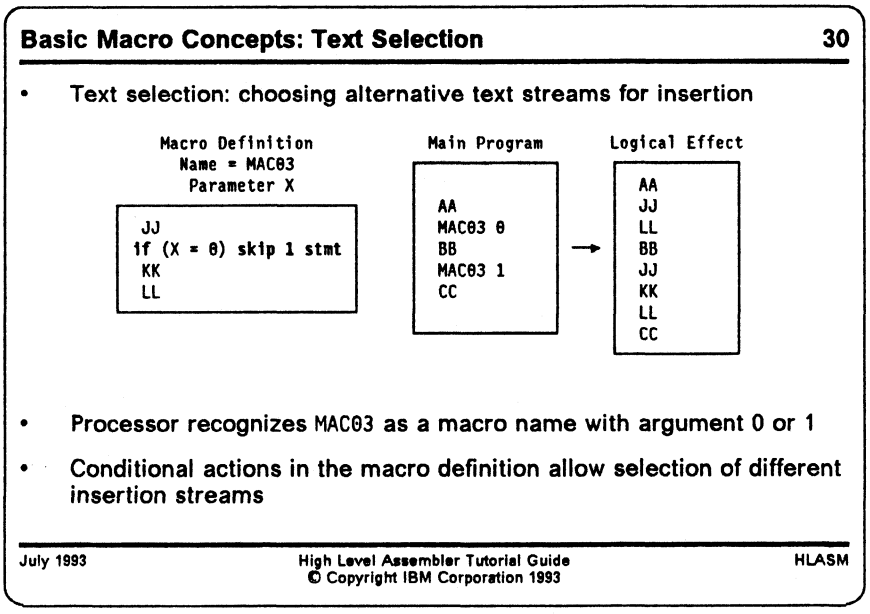
Figure 8. Basic Macro Mechanisms: Text Parameterization

This example illustrates text modification: tailoring of the inserted text ("parameterization") depending on locally-specified conditions.

When a macro call is recognized, it is normal for additional information (besides the simple act of activating the definition) to be passed to the macro expansion. Thus, when the processor of the Main Program recognizes MAC02 as a macro name, it also provides the two *arguments* CC and DD to the macro expander, which substitutes them for occurrences of the two *parameters* X and Y, respectively.

The argument CC is *associated* with parameter X, and DD is associated with Y. This simple example of parameter-argument association is typical of many macro processors: associ-

ation proceeds in left-to-right order, matching each positional parameter in turn with its corresponding positional argument. Other forms of association are possible.



Text Selection

Text selection is fundamental to most macro processors, because it allows choices among alternative sequences of generated text. In Figure 9, a simple form of text selection is modeled by the `if` statement: a simple test of the argument corresponding to the parameter X tells whether or not to generate the string KK. If the argument is 0, KK is not generated; otherwise it is.

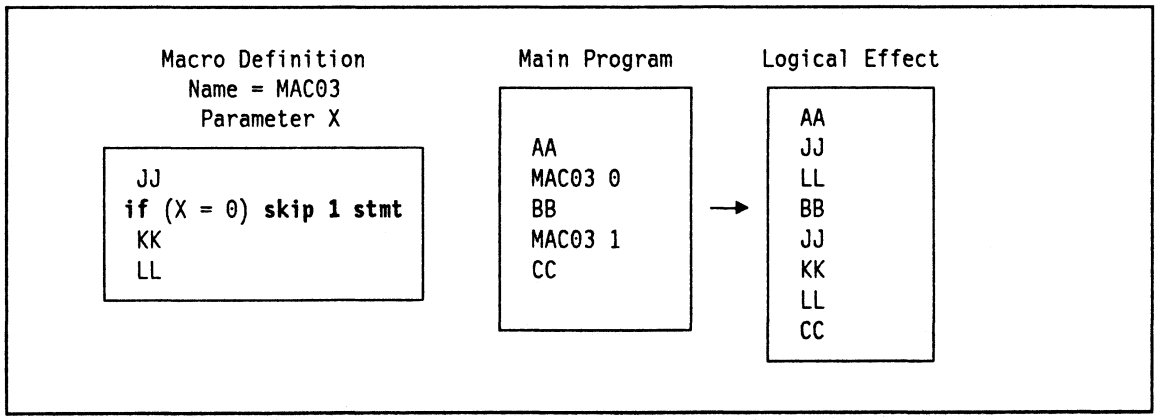
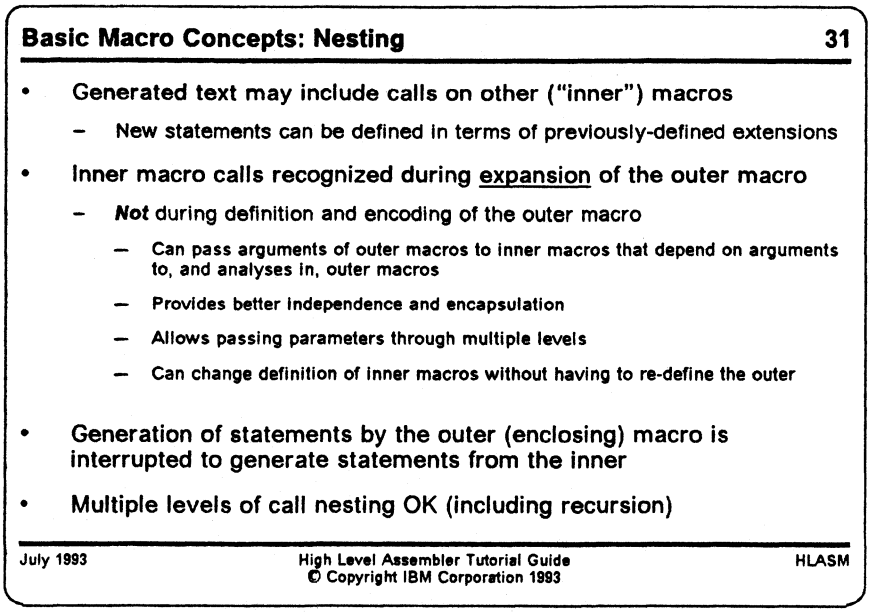


Figure 9. Basic Macro Mechanisms: Text Selection



Macro Nesting

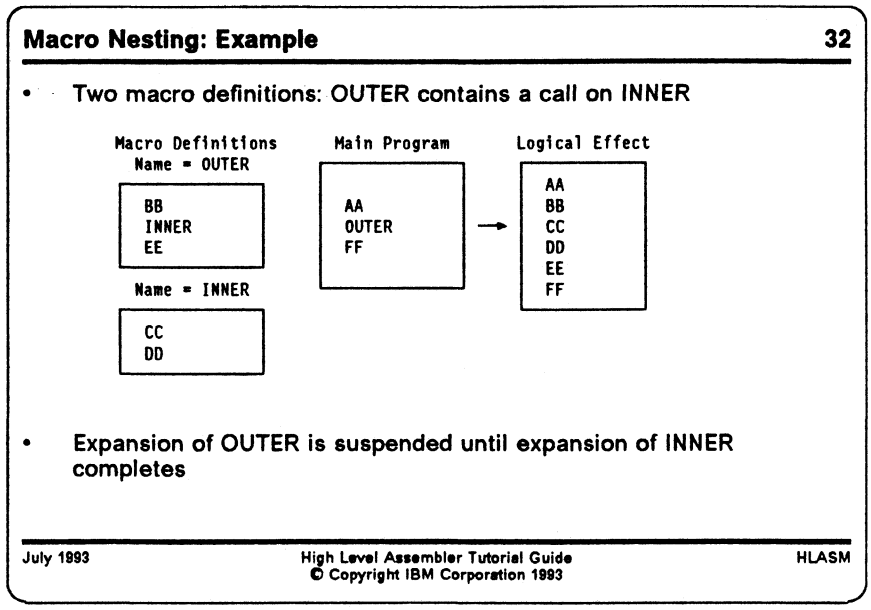
A key strength of the macro language is its ability to build new capabilities on existing facilities. The most common of these abilities is called "macro nesting": generated text may include (or create!) calls on other macros ("inner macro calls"). It is by this mechanism that new statements can be defined in terms of previously-defined extensions; it is fundamental to much of the power and "leverage" of macro languages.

The inner calls are recognized during *expansion* of the outer (enclosing) macro, *not* during macro definition and encoding. This may seem a very minor and obscure technical detail, but it turns out in practice to have wide-ranging implications.

- By deferring the recognition of inner macro calls until the enclosing macro is expanded, you can pass arguments to inner macros that depend on arguments to, and analyses in, outer macros.
- Recognition following expansion provides better independence and encapsulation: you can change the definition of the inner macro without having to re-define the outer.
- You will also save coding effort: if the definition of an inner macro needed to be changed, and its definition was already "embodied" in some way in other macros that called it, then all the "outer" macro definitions would have to be revised.

The generation process for inner macro calls requires that the macro processor maintain some kind of "push-down stack" for its activities.

- Generation of statements by the outer (enclosing) macro is suspended temporarily to generate statements from the inner.
- Multiple levels of call nesting are quite acceptable (including recursion: a macro may call itself directly or indirectly), and are often a source of added power and flexibility.



In the example in Figure 10, two macros named OUTER and INNER are known to the processor of the Main Program. When the name OUTER is recognized as a macro name, processing of the Main Program is suspended and expansion of the OUTER macro begins. When INNER is recognized as as macro name, processing of the OUTER macro is also suspended and expansion of the INNER macro begins. When the INNER macro expansion completes, the OUTER macro resumes expansion; when the expansion of the OUTER macro completes, processing resumes in the Main Program following the OUTER statement.

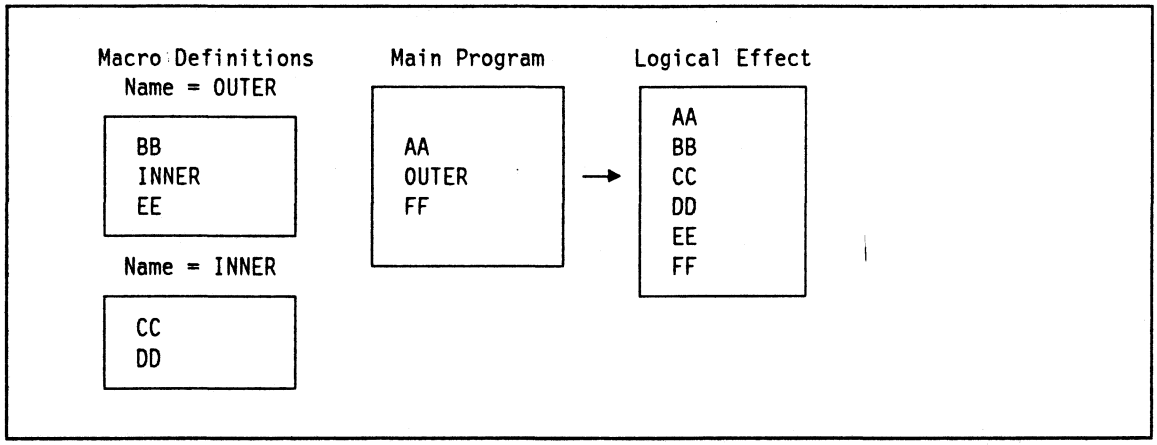


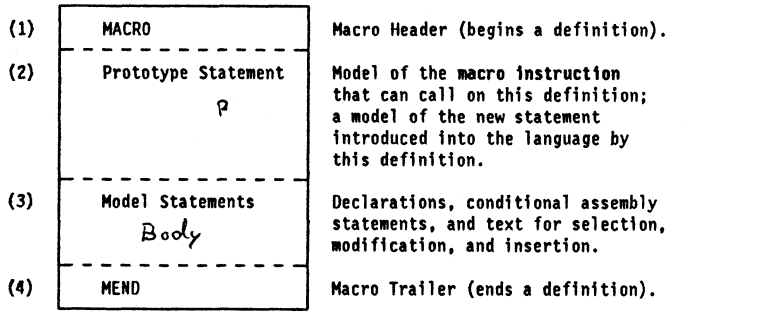
Figure 10. Basic Macro Mechanisms: Nesting

The power of a macro facility is enhanced by its ability to combine the basic functions of text insertion, text parameterization, text selection, and macro nesting.

Each of the features, concepts, and capabilities described above can be expressed in a way natural to the System/360/370/390 Assembler Language.

The Assembler Language Macro Definition 33

- A macro definition has four parts:



The Assembler Language Macro Definition ... 34

- Declares a macro name that represents a defined stream of program text
- Prototype statement declares parameter variable symbols
- Model statements ("macro body") provide logic and text
- Format of prototype dictated by desire not to introduce arbitrary forms of statement recognition for new statements
- Definition may be found
  - "in-line" (a "source macro definition")
  - in a library
  - or both
- Where the definition is found affects recognition rules

The Assembler Language Macro Definition

The definition of a macro declares the macro name that is to stand for (represent) a given stream of program text. The general form of an Assembler Language macro definition has four parts:

1. a macro header statement (MACRO: the start of the definition)
2. a prototype statement, which provides the macro name and a model of the macro-instruction "call" that must be recognized in order to activate this definition
3. the macro body, containing declarations of variable symbols, model statements to be parameterized and generated, and conditional assembly statements to assign values to variable symbols and to select alternative processing sequences

4. a macro trailer statement (MEND: the end of the definition).

These four parts are illustrated in Figure 11:

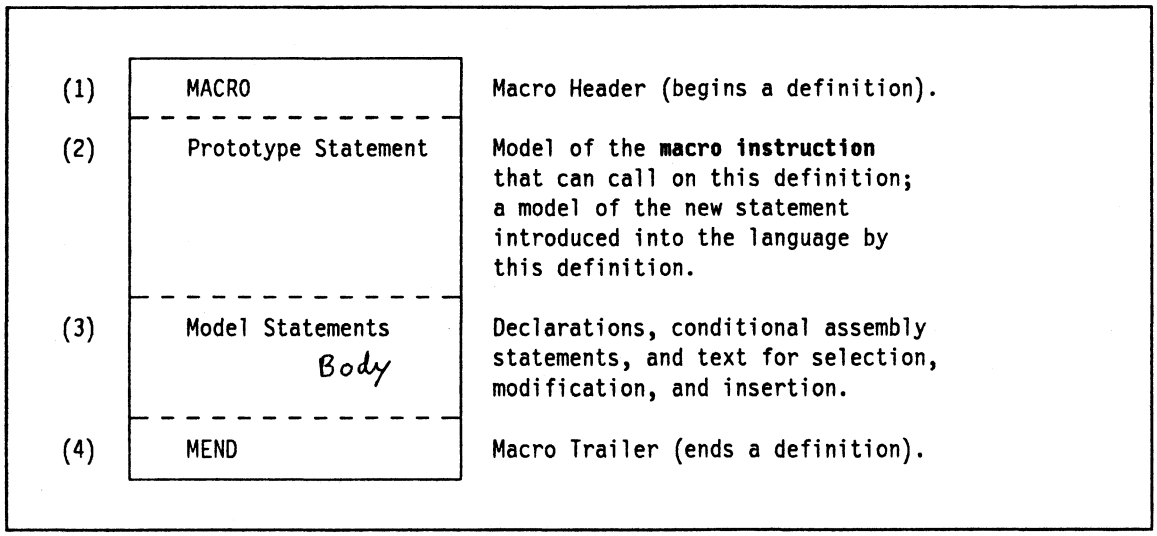


Figure 11. Assembler Language Macro Definition: Format

While many possible forms of macro definition and recognition are possible, the general format used in the System/360/370/390 Assembler Language is dictated by a desire not to introduce arbitrary forms of statement syntax and recognition rules for new statements. This has the advantage that there is no need to distinguish language extensions from the base language.

A macro definition may be "in-line" (also called a "source macro definition") or in a library. Where the definition is found by the assembler affects the recognition rules, as will be described in "Macro-Instruction Recognition" on page 45.

Macro-Instruction Definition Example

We can rewrite the example in Figure 7 on page 35 to look like a "real" macro, as follows:

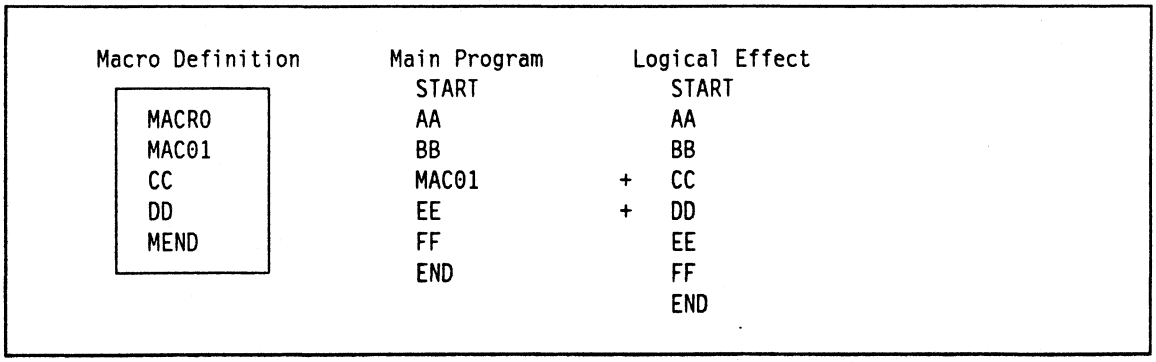
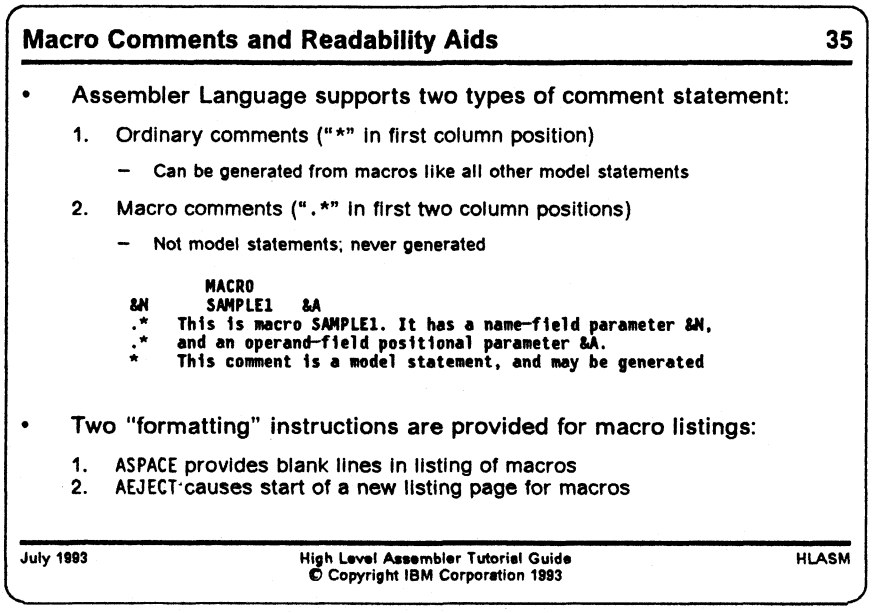


Figure 12. Assembler Language Macro Mechanisms: Text Insertion by a "Real" Macro

The "+" characters shown in the "Logical Effect" column correspond to the characters inserted by the assembler in its listing to indicate that the corresponding statements were generated from a macro.



Macro Comments and Readability

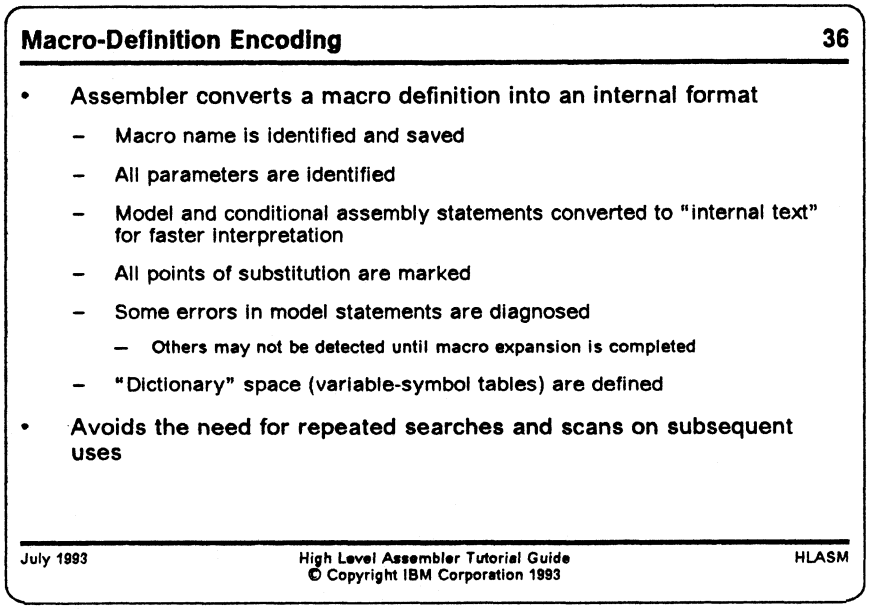
The macro facility provides a way to embed "macro comments" into the body of a macro definition. Because both ordinary comment statements (with an asterisk in the left margin) and blank lines (for spacing) are *model statements*, they may be part of the generated text from a macro expansion. Macro comments are never generated, and use the characters .* in the left margin, as illustrated below:

```
        MACRO
&N      SAMPLE1   &A
.*   This is macro SAMPLE1. It has a name-field parameter &N,
.*   and an operand-field positional parameter &A.
        - - -
*    This comment is a model statement, and may be generated
        - - -
        MEND
```

Figure 13. Example of Ordinary and Macro Comment Statements

It is good practice to comment macro definitions generously, because the conditional assembly language is sometimes difficult to read and understand.

The formatting and printing of macro definitions can be simplified by using the ASPACE and AEJECT statements. ASPACE provides blank lines in the assembler's listing of a macro definition, and AEJECT causes the assembler to start a new listing page when it is printing a macro definition. Both are not model statements, and are therefore never generated.

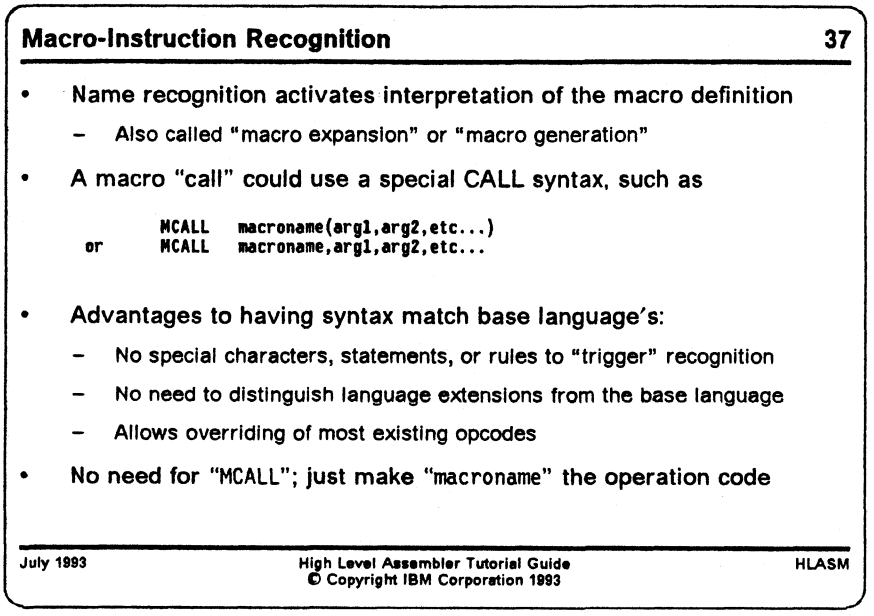


Macro-Definition Encoding

Because the System/360/370/390 Assemblers have been designed to support extensive use of macros, their implementation reflects a need to provide efficient processing. Thus, the assembler initially converts macro definitions into an encoded internal format for later use; this is sometimes called "macro editing".

- The macro's name is identified and saved (so that later references to the macro name can be recognized as macro calls).
- All parameters are identified, and entries are made in a "local macro dictionary" for them.
- Model and conditional assembly statements are converted to "internal text" for faster interpretation.
- All points of substitution are identified and marked. Because these are determined during macro encoding, it is perhaps more understandable why substituting strings like `'&A'` will not cause a further effort to re-scan the statement and substitute a new value represented by `'&A'`.
- Some errors in model statements are diagnosed, but others may not be detected until macro expansion is attempted.
- "Dictionary" space (variable-symbol tables) are defined for local variable symbols, and space is added to the global variable symbol dictionary for newly-encountered global names.

Encoding a macro definition in advance of any expansions avoids the need for repeated library searches and encoding scans on subsequent uses of the macro.



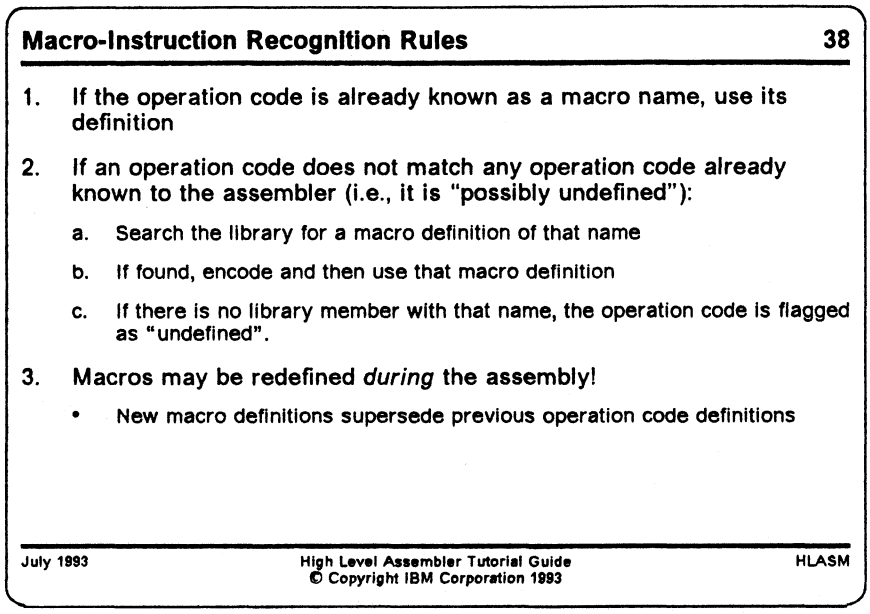
Macro-Instruction Recognition

When the assembler scans a statement, and identifies its operation code as a macro name, *recognition* of the name triggers an activation of an interpreter of the encoded form of the macro definition. This is called "macro expansion" or "macro generation", and typically results in insertion of program text into the assembler's input stream.

Both macro name declaration (definition) and recognition have specific rules that are closely tied to the base language syntax of the System/360/370/390 Assembler Language. A macro "call" could use or require a special CALL syntax, such as

```
        MCALL   macroname(arg1,arg2,etc...)
or      MCALL   macroname,arg1,arg2,etc...
```

However, there are advantages to having the syntax of macro calls match the base language's, and to allow overriding of existing opcodes; hence, we simply elide the MCALL and make the "macroname" become the operation code and the arguments become the operands of the macro instruction statement.



Macro-Instruction Recognition Rules

The assembler recognizes a macro instruction as follows:

1. If the macro name has already been defined in the program (as a "source" or "in-line" definition, either explicitly or because a COPY statement brought it in-line from a library, or because a previous macro instruction statement brought the definition from the library), use it in preference to any other definition of that operation.

   - You may use a macro definition to override the assembler's default definitions of all machine instruction statements, and of most "native" Assembler Instruction statements (generally, the conditional-assembly statements cannot be overridden).

2. If an operation code does not match any operation code "known" to the assembler (i.e., it is "possibly undefined"), the assembler will then:

   a. Search the library for a macro definition of that name.

   b. If the assembler finds a library member with that name, the macro name defined on the prototype statement must match the member name. The assembler will then encode and use this definition.

   c. If there is no library member with that name, then the operation code is flagged as "undefined".

While it is not a common practice to do so, macros may be redefined during the assembly by introducing a new macro definition for that name.

Define General Register Equates 39

- Easy to do this with a macro like

```
         MACRO
         GREGS
GR0      EQU   0
.*       - - -   etc.       Similarly for GR1 — GR14
GR15     EQU   15
         MEND
```

- A simple variation with a conditional-assembly loop:

```
         MACRO
         GREGS
         LCLA  &N             Define a counter variable
.X       ANOP
GR&N     EQU   &N
&N       SETA  &N+1           Increment &N by 1
         AIF   (&N LE 15).X   Repeat for all registers 1-15
         MEND
```

July 1993 High Level Assembler Tutorial Guide HLASM

Example: Defining Equated Symbols for Registers

To illustrate a basic form of macro, suppose you wish to generate a sequence of EQU statements to define symbolic names GR0, GR1, ..., GR15 for referring to the sixteen General Purpose Registers. A call to the GREGS macro will do this:

```
         MACRO
         GREGS
GR0      EQU   0
GR1      EQU   1
GR2      EQU   2
GR3      EQU   3
GR4      EQU   4
GR5      EQU   5
GR6      EQU   6
GR7      EQU   7
GR8      EQU   8
GR9      EQU   9
GR10     EQU   10
GR11     EQU   11
GR12     EQU   12
GR13     EQU   13
GR14     EQU   14
GR15     EQU   15
         MEND
```

Figure 14. Simple Macro to Generate Register Equates

Then, a call to the GREGS macro will define the desired equates, by inserting the sixteen model statements into the statement stream.

The macro definition can be made more compact by using conditional assembly statements to form a simple loop inside the macro:

```
         MACRO
         GREGS
         LCLA  &N            Define a counter variable
.X       ANOP
GR&N     EQU   &N
&N       SETA  &N+1          Increment &N by 1
         AIF   (&N LE 15).X  Repeat for all registers 1-15
         MEND
```

Figure 15. Macro to Generate Register Equates Differently

Macro Parameters and Arguments 40

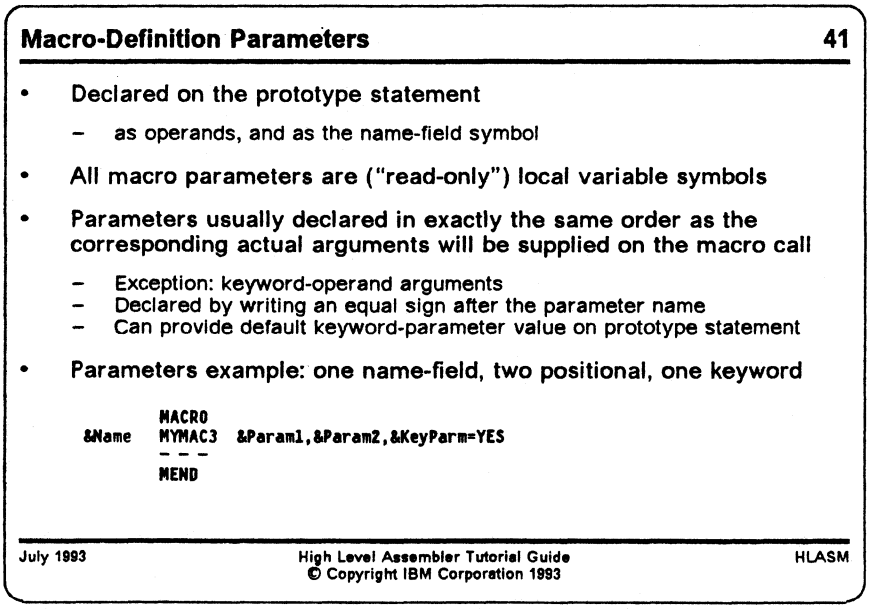
- Distinguish *parameters* from *arguments*:
- Parameters are
  - declared on macro definition prototype statements
  - always local variable symbols
  - assigned values by association with the arguments of macro calls
- Arguments are
  - supplied on a macro instruction (macro call)
  - almost any character string (typically, symbols)
  - providers of values to associated parameters

July 1993 High Level Assembler Tutorial Guide HLASM

Macro Parameters and Arguments

In the following discussion, we will distinguish *parameters* from *arguments*, as follows:

- Parameters are
  - declared on the prototype statements of macro definitions
  - always local variable symbols
  - assigned values by being associated with the arguments of a macro instruction
  - sometimes known as "dummy arguments" or "formal parameters".
- Arguments are
  - supplied on a macro instruction statement ("macro call")
  - almost any character string (typically, symbols)
  - the providers of values to the corresponding associated parameters
  - sometimes known as "actual arguments" or "actual parameters".



Macro-Definition Parameters

The parameters in a macro definition are declared by virtue of their appearing as operands (and the name-field symbol) on the prototype statement. These declared parameters *are* variable symbols! Usually, they are declared in exactly the same order as the corresponding actual arguments will be supplied on the macro call.

The exception is *keyword arguments*: they are declared by writing an equal sign after the parameter name. You can also provide a default value for a keyword parameter on the prototype statement, by placing that value after the equal sign. When the macro is called, the argument values for keyword parameters are supplied by writing the keyword parameter name, an equal sign, and the value, as an operand of the macro call.

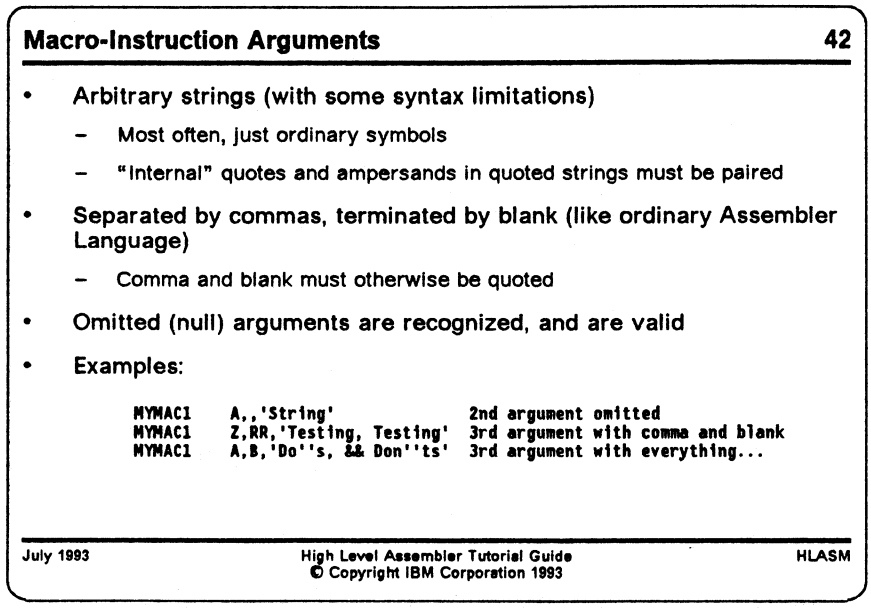
For example, suppose we write a macro prototype statements as shown in Figure 16:

```
        MACRO
&Name   MYMAC3  &Param1,&Param2,&KeyParm=YES
        - - -
        MEND
```

Figure 16. Sample Macro Prototype Statement

The prototype statement defines a name-field parameter (&Name), two positional parameters (&Param1,&Param2), and one keyword parameter (&KeyParm) with a default value YES.

Unlike positional arguments and parameters, keyword arguments and parameters may appear in any order, and may be mixed freely among the positional items on the prototype statement and the macro call.



Macro-Instruction Arguments

The arguments of a macro instruction are the name-field entry and the operands. They may be arbitrary strings of characters, with some syntax limitations such as requiring strings containing quotes and ampersands to contain pairs of each. Most often, the operands will be just symbols (literals are allowed in almost all circumstances).

The operands are separated by commas, and terminated by a blank (conforming to the normal Assembler Language syntax rules). If the argument string is intended to contain a comma or a blank, they must be quoted.

Omitted (null) arguments are perfectly acceptable.

To illustrate, suppose a macro named MYMAC1 expects three positional arguments. Then in the following example,

```
MYMAC1   A,,'String'              2nd argument omitted

MYMAC1   Z,RR,'Testing, Testing'  3rd argument with comma and blank

MYMAC1   A,B,'Do''s, && Don''ts'  3rd argument with everything...
```

the first call omits the second argument; the second call has a quoted character string containing an embedded comma and quote as its third argument; and the third call has a variety of special characters in its quoted-string third argument.

Pairs of quotes or ampersand characters are required within quoted strings used as macro arguments, for proper argument parsing and recognition. These characters are *not* condensed into a single character when the argument is associated ("passed") to the corresponding symbolic parameter.

Macro Parameter-Argument Association 43

- Three ways to associate (caller's) arguments with (definition's) parameters:
  1. by position, referenced by declared name (most common way)
  2. by position, by argument number (using &SYSLIST notation)
  3. by keyword: always referenced by name, arbitrary order
     - Argument *values* supplied by writing **keyname=value**
- Example 1: (Assume prototype statement as on foil 41)

```
&Name   MYMAC3  &Param1,&Param2,&KeyParm=YES    Prototype

Lab1    MYMAC3  X,Y,KeyParm=NO      Call: 2 positional, 1 keyword argument

*  Parameter values: &Name    = Lab1
*                    &KeyParm = NO
*                    &Param1  = X
*                    &Param2  = Y
```


Macro Parameter-Argument Association ... 44

- Example 2:

```
Lab2   MYMAC3   A                  Call: 1 positional argument

*  Parameter values: &Name    = Lab2
*                    &KeyParm = YES
*                    &Param1  = A
*                    &Param2  = (null)
```

- Example 3:

```
       MYMAC3   H,KeyParm=MAYBE,J   Call: 2 positional, 1 keyword argument

*  Parameter values: &Name    = (null)
*                    &KeyParm = MAYBE
*                    &Param1  = H
*                    &Param2  = J
```


Macro Parameter-Argument Association

There are three ways to associate arguments with parameters:

1. by position, referenced by the declared positional parameter name (this is the most usual way for macros to refer to their arguments)
2. by position and argument number (using the &SYSLIST system variable symbol, which will be discussed in "Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol" on page 63)

3. by keyword: keyword arguments are always referenced by name, and the order in which they appear is arbitrary.[3] Values provided for keyword arguments override default values declared on the prototype statement.

To illustrate, consider the examples in Figure 17. Assuming the same macro definition prototype statement shown in Figure 16 on page 49, the resulting values associated with the parameters are as shown:

```
Lab1   MYMAC3   X,Y,KeyParm=NO        2 positional, 1 keyword argument

*  Parameter values: &Name   = Lab1      &KeyParm = NO
*                    &Param1 = X         &Param2  = Y

Lab2   MYMAC3   A                      1 positional argument

*  Parameter values: &Name   = Lab2      &KeyParm = YES
*                    &Param1 = A         &Param2  = (null)

       MYMAC3   H,KeyParm=MAYBE,J     2 positional, 1 keyword argument

*  Parameter values: &Name   = (null)    &KeyParm = MAYBE
*                    &Param1 = H         &Param2  = J
```

Figure 17. Macro Parameter-Argument Association Examples

In the third example, observe that the keyword argument KeyParm=MAYBE appears *between* the first and second positional arguments.

[3] The *Ada*™ programming language is the first major high-level language to support keyword parameters and arguments. Assembler Language programmers have been using them for decades!

Generating a Byte Sequence: BYTESEQ1 Macro 45

- BYTESEQ1 generates a byte for each value

```
        MACRO
&L      BYTESEQ1 &N
.*  BYTESEQ1 — generate a sequence of byte values, one per statement.
.*  No checking or validation is done.
        LclA  &K
        AIF   ('&L' EQ '').Loop  Don't define the label if absent
&L      DS    0AL1               Define the label (if present)
.Loop   ANOP
&K      SetA  &K+1               Increment &K
        AIF   (&K GT &N).Done    Check for termination condition
        DC    A11(&K)
        AGO   .Loop              Continue
.Done   MEND

* Two test cases

BS1a    BYTESEQ1  5
        BYTESEQ1  1
```

July 1993 High Level Assembler Tutorial Guide HLASM

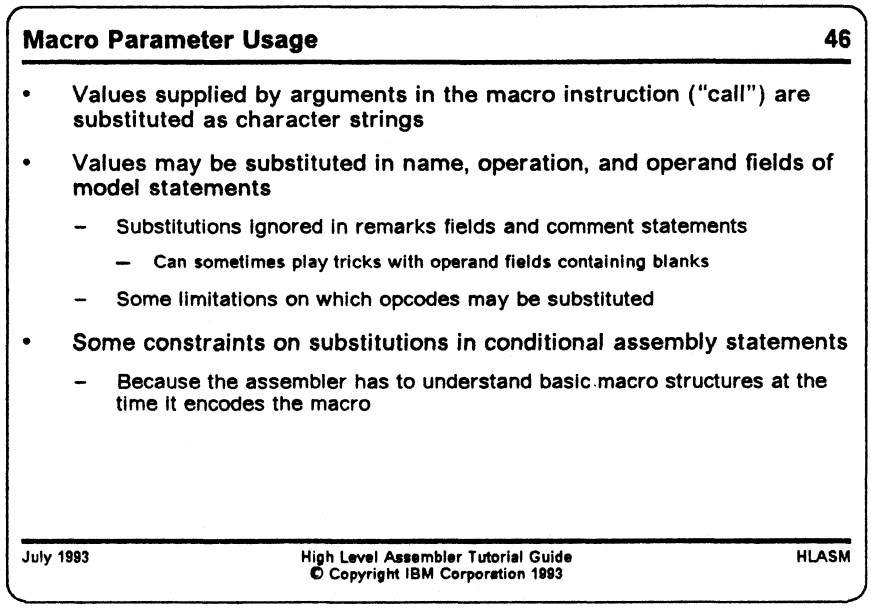
Example: Generating a Byte Sequence

We can write a macro with a single parameter to generate a sequence of bytes, using the same techniques as the conditional-assembly example given in Figure 4 on page 27.

```
        MACRO
&L      BYTESEQ1 &N
.*  BYTESEQ1 -- generate a sequence of byte values, one per statement.
.*  No checking or validation is done.
        LclA  &K
        AIF   ('&L' EQ '').Loop  Don't define the label if absent
&L      DS    0X                 Define the label
.Loop   ANOP
&K      SetA  &K+1               Increment &K
        AIF   (&K GT &N).Done    Check for termination condition
        DC    A11(&K)
        AGO   .Loop              Continue
.Done   MEND
* Two test cases
BS1a    BYTESEQ1  5
        BYTESEQ1  1
```

Figure 18. Macro to Define a Sequence of Byte Values

This macro generates a separate DC statement for each byte value. As we will see later, it has some limitations that are easy to fix.



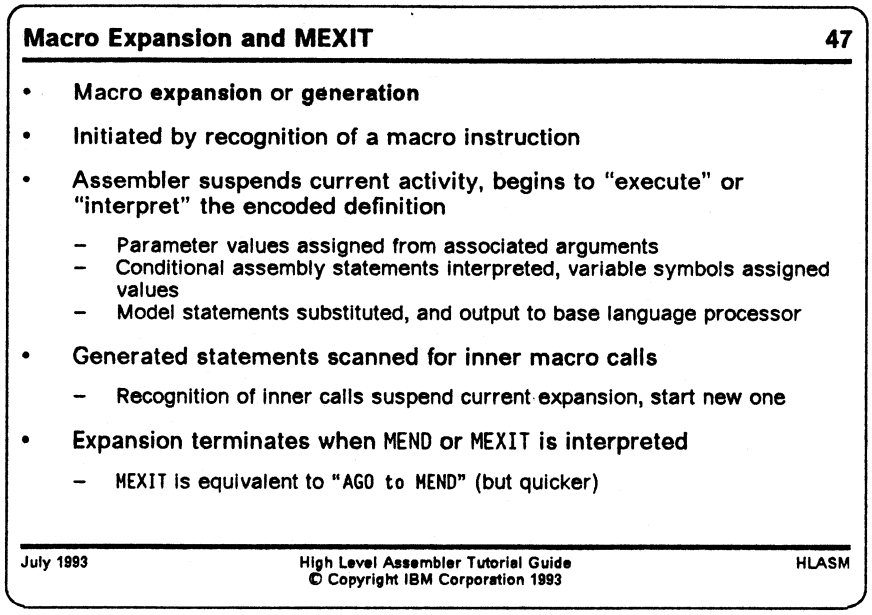
Macro Parameter Usage

Values are assigned to macro parameters from the corresponding arguments on the macro-instruction statement, either by position in left-to-right order (for positional arguments), or by name (for keyword arguments). These are then substituted as character strings into model statements (wherever points of substitution marked by the parameter variable symbols appear). The points of substitution in model statements may be in the

- name field
- operation field
- operand field

but not in the remarks field, nor in comment statements. (For some operations, it is possible to construct an operand string containing embedded blanks followed by "remarks" into which substitutions have been done. We will leave as an exercise for the reader the delights of discovering how to do this.)

Substitutions are not allowed in some places in conditional or ordinary assembly statements such as COPY, REPRO, MACRO, and MEND, because the assembler must know some information about the basic structure of the macro definition (and of the entire source program!) at the time it is encoded. For example, substituting the string `MEND` for an operation code in the middle of a macro definition could completely alter that definition!



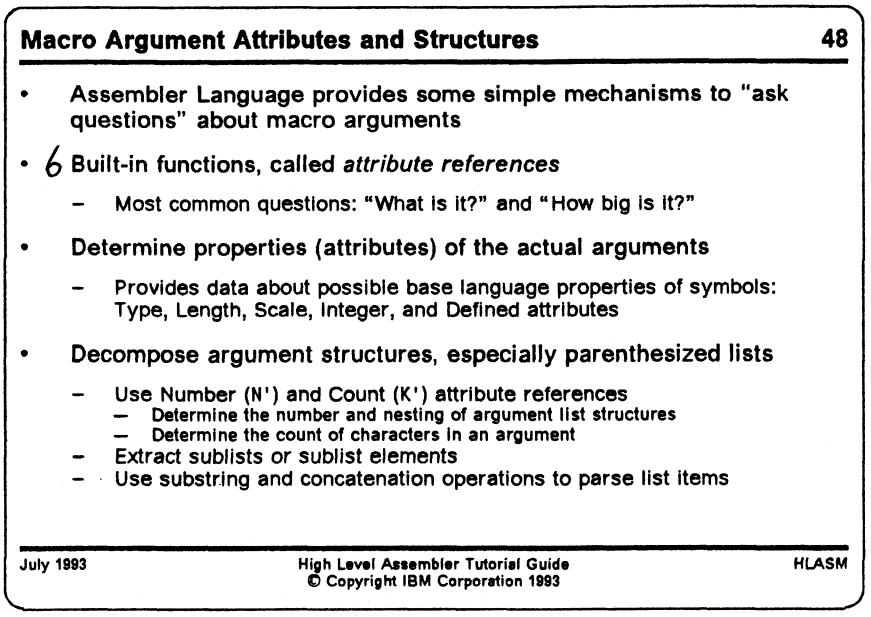
Macro Expansion and the MEXIT Statement

When the assembler recognizes a macro instruction, macro *expansion* or macro *generation* is initiated. The assembler suspends its current activity, and begins to "execute" or "interpret" the encoded definition of the called macro.

During expansion, the first step is to assign parameter values from the associated arguments on the macro call. Subsequently, conditional assembly statements are interpreted, variable symbols are assigned values, model statements are substituted, and text is output to the base language processor.

The generated statements are scanned for inner macro calls; recognition of an inner call suspends the current expansion, and starts a new one for the newly-recognized inner macro.

Expansion of a macro terminates when either the MEND statement is reached, or when an expansion-terminating macro-exit MEXIT statement is interpreted. (MEXIT is equivalent to an "AGO to MEND" statement, but is quicker to execute.)



Macro Argument Attributes and Structures

Among the elegant features of the Assembler Language are some simple mechanisms (built-in functions, called *attribute references*) that allow you to determine some properties (i.e., attributes) of the actual arguments. For example, attribute references provide information about possible base language (ordinary assembler language) use of the symbols: what kinds of objects they name, what is the length attribute of the named object, etc.

Three major classes of "inquiry facilities" are provided:

1. The type attribute reference (T') allows you to ask "What base-language meaning is attached to it?" about a macro argument. The value of the type attribute reference (a single character; only the type attribute reference has character values) can tell you whether the argument is

   - a symbol that names data, machine instructions, macro instructions, sections, etc.
   - a self-defining term (binary, character, decimal, or hexadeimal)
   - an "unknown" type.

2. The "mechanical" or "physical" characteristics of macro arguments can be determined by using

   - two attribute references: Count (K') supplies the actual count of characters in the argument, and Number (N') tells you how many elements appear in an argument list structure.
   - list-structure referencing and decomposition operations, involving subscripted references to parameter variable symbols.

     A rather sophisticated list scanning capability is provided to help you decompose argument structures, especially parenthesized lists. With this notation, you can

     – determine the number and nesting of all such list structures
     – extract any sublists or sublist elements

- use the usual substring and concatenation operations to manipulate portions of lists and list elements.

3. The base-language attributes of macro arguments can be determined by using any of four attribute references: Length (L'), Scale (S'), Integer (I'), and Defined (D'). All four have numeric values.

Macro Argument Attributes: Type 49

- Type attribute reference (T') answers
  - "What is it?"
  - "What meaning might it have in the base language?"
- Assume the following statements in a program:

```
A       DC    A(*)
B       DC    F'10'
C       DC    E'2.71828'
D       MVC   A,B
```

- And, assume the following prototype statement for MACTA:

```
MACTA &P1,&P2,...,etc.
```

July 1993 High Level Assembler Tutorial Guide HLASM

Macro Argument Attributes: Type ... 50

- Then a call to MACTA like

```
Z       MACTA A,B,C,D,C'A',,'?',Z        Call MACTA with various arguments
```

- would provide these type attributes:

```
T'&P1 = 'A'    aligned, implied-length address
T'&P2 = 'F'    aligned, implied-length fullword binary
T'&P3 = 'E'    aligned, implied-length short floating-point
T'&P4 = 'I'    machine instruction statement
T'&P5 = 'N'    self-defining term
T'&P6 = 'O'    omitted (null)
T'&P7 = 'U'    unknown
T'&P8 = 'M'    macro instruction statement
```

July 1993 High Level Assembler Tutorial Guide HLASM

Macro-Instruction Argument Properties: Type Attribute

The type attribute reference is often the first used in a macro, to help the macro determine "What is it?". More precisely, it tries to answer the question "What meaning might this argument string have in the base language?" It typically appears in conditional assembly statements like these:

```
        AIF   (T'&Param1 eq 'O').Omitted     Argument is null
        AIF   (T'&Param1 eq 'U').Unknown     Unknown argument type
```

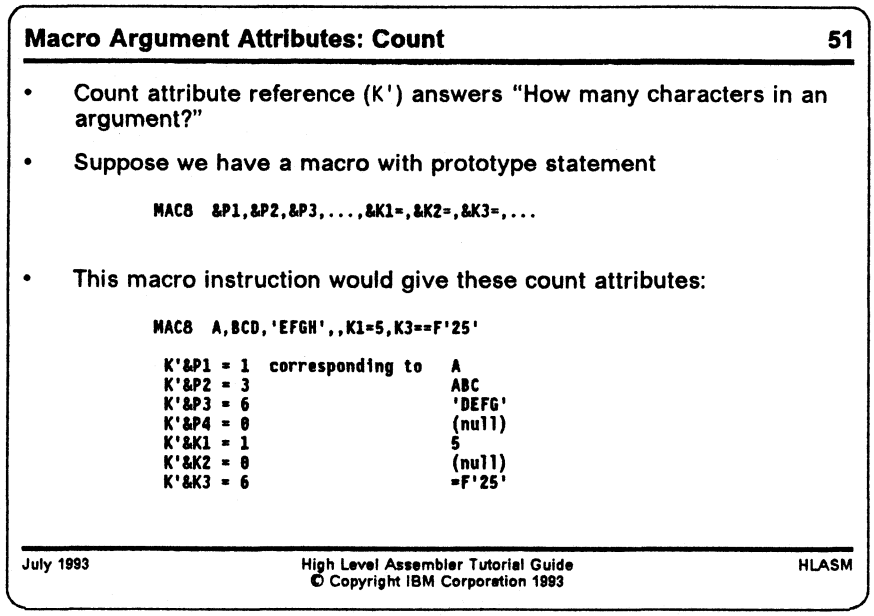
To illustrate some of the possible values returned by a type attribute reference, assume the following statements appear in a program:

```
A       DC    A(*)
B       DC    F'10'
C       DC    E'2.71828'
D       MVC   A,B
```

If the same program contains a macro named MACTA with positional arguments &P1,&P2,...,etc., and if MACTA is called with the following arguments, then a type attribute reference to each of the positional parameters would return the indicated values:

```
Z       MACTA A,B,C,D,C'A',,'?',Z        Call MACTA with various arguments

        T'&P1 = 'A'    aligned, implied-length address
        T'&P2 = 'F'    aligned, implied-length fullword binary
        T'&P3 = 'E'    aligned, implied-length short floating-point
        T'&P4 = 'I'    machine instruction statement
        T'&P5 = 'N'    self-defining term
        T'&P6 = 'O'    omitted (null)
        T'&P7 = 'U'    unknown
        T'&P8 = 'M'    macro instruction statement
```

There are 28 possible values that might be returned by a type attribute reference.



Macro-Instruction Argument Properties: Count Attribute

A macro argument has one irreducible, unavoidable property: the count of the number of characters it contains. These can be determined for any argument using the count attribute reference, K'. For example, if MAC8 has positional parameters &P1, &P2, ..., etc., and keyword parameters &K1, &K2, ..., etc., then for a macro instruction statement such as the following:

```
MAC8  A,BCD,'EFGH',,K1=5,K2=,K3==F'25'
```

we would find that

```
K'&P1 = 1  corresponding to   A
K'&P2 = 3                     ABC
K'&P3 = 6                     'DEFG'
K'&P4 = 0                     (null)
K'&K1 = 1                     5
K'&K2 = 0                     (null)
K'&K3 = 6                     =F'25'
```

Macro Argument Attributes: Number 52

- Number attribute reference (N') answers "How many items in a list?"
- List: a parenthesized sequence of items separated by commas

```
Examples:    (A)   (B,C)   (D,E,,F)
```

- List items may themselves be lists, to any nesting

```
Examples:    ((A))   (A,(B,C))   (A,(B,C,(D,E,,F),G),H)
```

- Subscripts on parameters refer to argument list (and sublist) items
  - Each added subscript references one nesting level deeper
- N' also determines maximum subscript used with a subscripted variable symbol

Macro Argument List Structure Examples 53

- Assume the same macro prototype as in foil 51:

```
MAC8  &P1,&P2,&P3,...,&K1=,&K2=,&K3=,...   Prototype

MAC8  (A),A,(B,C),(B,(C,(D,E)))     Sample macro call
```

- Then, the number attributes and sublists are:

```
&P1       = (A)           N'&P1       = 1   list of 1 item, A
&P1(1)    = A             N'&P1(1)    = 1   (A is not a list)
&P2       = A             N'&P2       = 1   (A is not a list)
&P3       = (B,C)         N'&P3       = 2   list of 2 items, B and C
&P3(1)    = B             N'&P3(1)    = 1   (B is not a list)
&P4       = (B,(C,(D,E))) N'&P4       = 2   list of 2 items, B and (C,(D,E))
&P4(2)    = (C,(D,E))     N'&P4(2)    = 2   list of 2 items, C and (D,E)
&P4(2,2)  = (D,E)         N'&P4(2,2)  = 2   list of 2 items, D and E
&P4(2,2,1) = D            N'&P4(2,2,1) = 1  (D is not a list)
&P4(2,2,2) = E            N'&P4(2,2,2) = 1  (E is not a list)
```


Macro-Instruction Argument Properties: Number Attribute

A list is a parenthesized sequence of items, separated by commas. The following are examples of lists:

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

Figure 19. Macro Argument List Structures

List items may themselves be lists (which may in turn contain lists, and so forth). Examples of lists containing sublists are:

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

Lists may have any number of items, and any level of nesting, subject only to the constraint that the size of the argument may not exceed 255 characters.

The number attribute reference (N') is used to determine the number of elements in a list or sublist, or the number of elements in a subscripted variable symbol. For example, if the three lists in Figure 19 on page 60 were arguments associated with parameters &P1, &P2, and &P3 respectively, then a number attribute reference to each parameter would return the following values:

```
N'&P1 = 1    (A)      is a list of 1 item
N'&P2 = 2    (B,C)    is a list of 2 items
N'&P3 = 4    (D,E,,F) is a list of 4 items; the third is null

&Z(17) = 42           Set an element of a subscripted variable symbol
 N'&Z  = 17           maximum subscript on &Z is 17
```

Sublists

It is sometimes useful to pass groups of related argument items as a single unit, by grouping them into a list. This can save the effort needed to name additional parameters on the macro prototype statement, can can simplify the documentation of the macro call.

To extract list items from argument lists and sublists within a macro, subscripts are attached to the parameter name. For example, if &P is a positional parameter, and N'&P is not zero (meaning that the argument associated with &P is indeed a list), then &P(1) is the first item in the list, &P(2) is the second, and &P(N'&P) is the last item.

To determine whether any list item is itself a list, we use another number attribute reference. For example, if &P(1) is the first item in the list argument associated with &P, then N'&P(1) is the number of items in the *sublist* associated with &P(1). For example, if argument ((X,Y),Z,T) is associated with &P, then

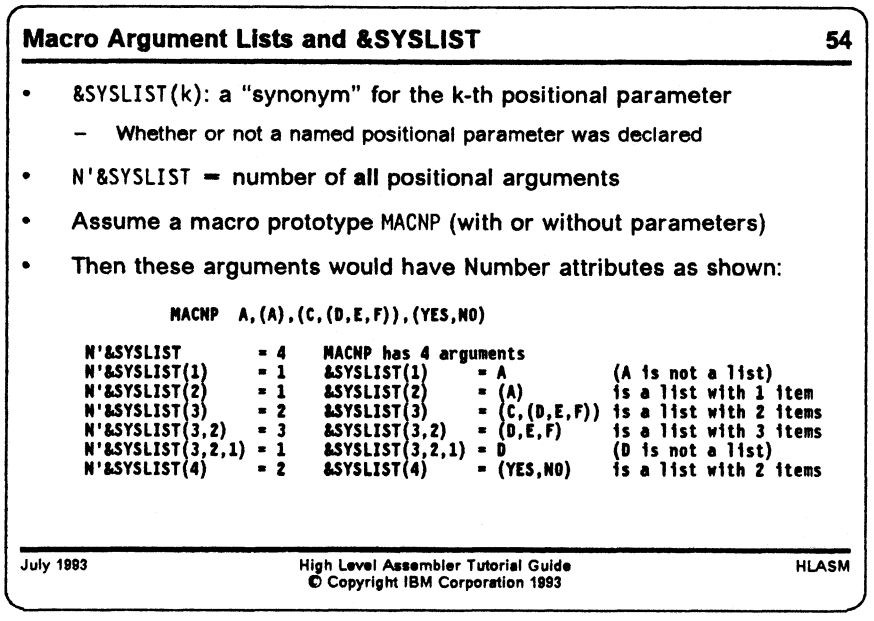
```
N'&P    = 3    items are (X,Y), Z, and T
N'&P(1) = 2    items are X and Y
```

As list arguments become more deeply nested, the number of subscripts used to refer to their list items also increases. For example, &P(1,2,3) refers to the third item in the sublist appearing as the second item in the sublist appearing as the first item in the list argument associated with &P. Suppose MAC8 has positional parameters &P1, &P2, ..., etc., then for a macro instruction statement such as the following:

```
        MAC8   (A),A,(B,C),(B,(C,(D,E)))     Sample macro call

&P1       = (A)           N'&P1       = 1   list of 1 item, A
&P1(1)    = A             N'&P1(1)    = 1   (A is not a list)
&P2       = A             N'&P2       = 1   (A is not a list)
&P3       = (B,C)         N'&P3       = 2   list of 2 items, B and C
&P3(1)    = B             N'&P3(1)    = 1   (B is not a list)
&P4       = (B,(C,(D,E))) N'&P4       = 2   list of 2 items, B and (C,(D,E))
&P4(2)    = (C,(D,E))     N'&P4(2)    = 2   list of 2 items, C and (D,E)
&P4(2,2)  = (D,E)         N'&P4(2,2)  = 2   list of 2 items, D and E
&P4(2,2,1) = D            N'&P4(2,2,1) = 1  (D is not a list)
&P4(2,2,2) = E            N'&P4(2,2,2) = 1  (E is not a list)
```

There is an oddity in the assembler's interpretation of the number attribute for items which are not themselves lists. As can be seen from the first two samples above, both '(A)' and 'A' return a number attribute of 1. The assembler will treat parameter references &P and &P(1) as the same string if the argument corresponding to &P is not a properly formed list.

This means that if it is important to know whether or not a list item is in fact a parenthesized list, you will need to test the first and last characters to verify that the list is properly enclosed in parentheses.



Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol

It is frequently useful to be able to call a macro with an indefinite number of arguments that we intend to process "identically" or "equivalently", so that no particular benefit is gained from naming and referring to each one individually.

The system variable symbol &SYSLIST can be used to refer to the positional elements of the argument list: &SYSLIST(**k**) refers to the **k**-th positional argument, whether or not a corresponding positional parameter was declared on the macro's prototype statement. The total number of positional arguments in the macro instruction's operand list can be determined using a Number attribute reference: N'&SYSLIST is the number of positional arguments.

No other reference to &SYSLIST can be made without subscripts. Thus, it is not possible to refer to all the arguments (or to all the positional parameters) as a group using a single unsubscripted reference to &SYSLIST.

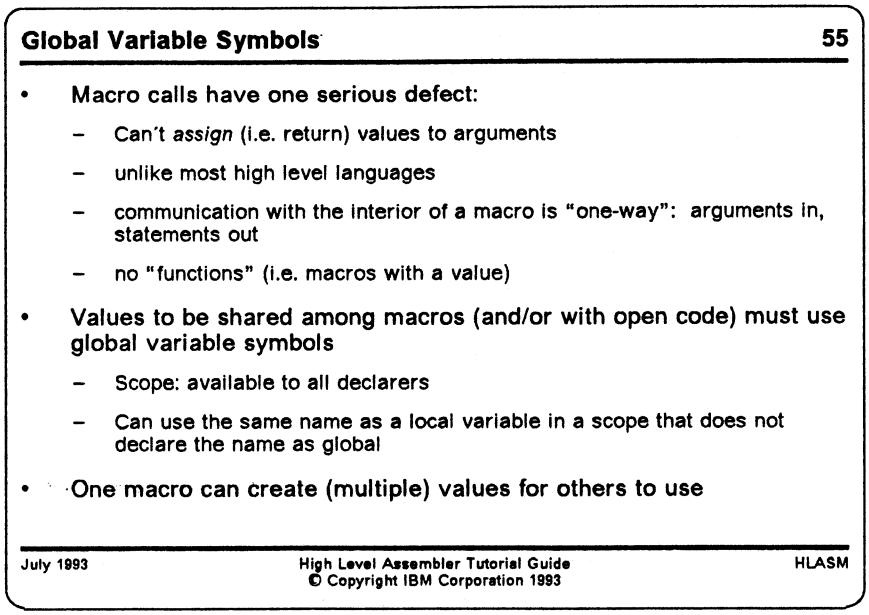
To illustrate the use of &SYSLIST references, suppose we have defined a macro named MACNP; whether or not any positional parameters are declared doesn't matter for this example. If we write the following macro call:

```
MACNP  A,(A),(C,(D,E,F)),(YES,NO)
```

then the number attributes of the &SYSLIST items, and their values, are the following:

```
N'&SYSLIST        = 4   MACNP has 4 arguments
N'&SYSLIST(1)     = 1   &SYSLIST(1)     = A           (A is not a list)
N'&SYSLIST(2)     = 1   &SYSLIST(2)     = (A)         is a list with 1 item
N'&SYSLIST(3)     = 2   &SYSLIST(3)     = (C,(D,E,F)) is a list with 2 items
N'&SYSLIST(3,2)   = 3   &SYSLIST(3,2)   = (D,E,F)     is a list with 3 items
N'&SYSLIST(3,2,1) = 1   &SYSLIST(3,2,1) = D           (D is not a list)
N'&SYSLIST(4)     = 2   &SYSLIST(4)     = (YES,NO)    is a list with 2 items
```

Observe that references to sublists are made in the same way as for named positional parameters. One additional (leftmost) subscript is needed for &SYSLIST references, because that parameter is being referenced by number rather than by name.



Global Variable Symbols

Thus far, our macro examples have been self-contained: all their communication with the "outside world" has been through their argument lists and the statements they generated.

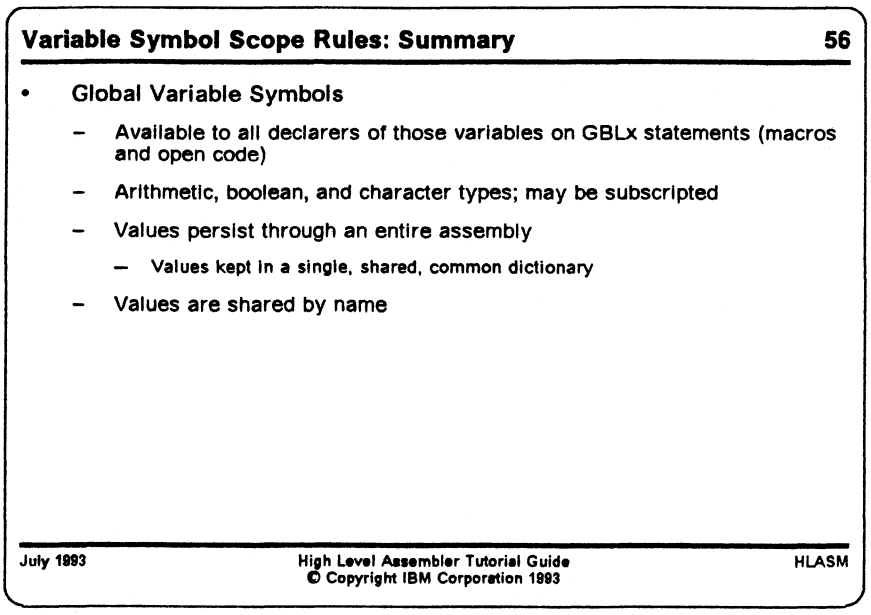
In the System/360/370/390 Assembler Language, macro calls have one serious omission: they can't *assign* (i.e: return) values to arguments, unlike most high level languages. That is, all macro arguments are "input only". Thus, communication with the interior of a macro by way of the argument list appears to be "one-way": arguments go in, but only statements come out.

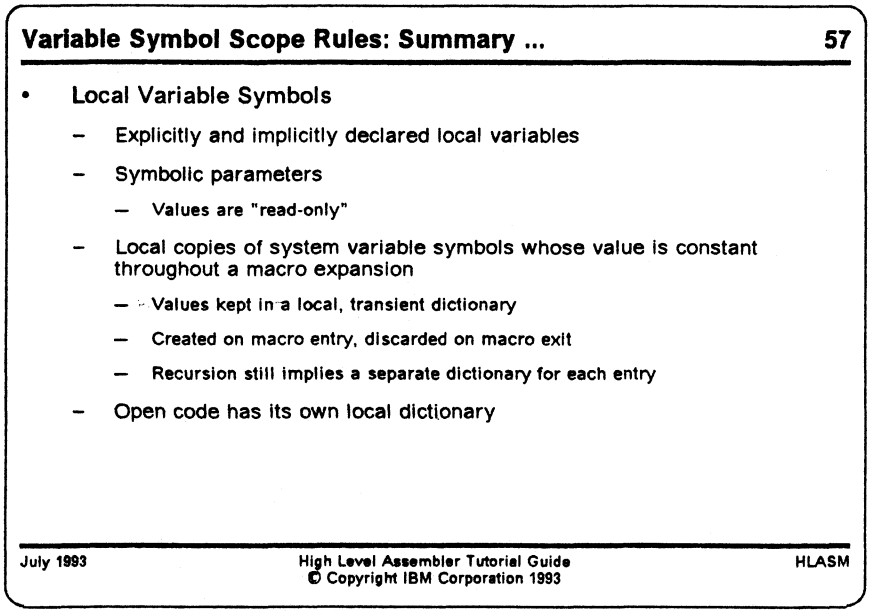
Furthermore, there is no provision for defining "functions" (that is, macros which return a value associated with the macro name itself).

Thus, values to be shared among macros (and/or with open code) must use a different mechanism, that of global variable symbols. The scope rule for global variable symbols is simple: they are shared by and are available to all declarers. (You may of course use the same name as a local variable in a scope that does not declare the name as global.)

With an appropriate choice of named global variable symbols, one macro can create single or multiple values for others to use.

The "dictionary" or "pool" of global symbols has similar behavior to certain kinds of external variables in high level languages, such as Fortran COMMON: all declarers of variables in COMMON may refer to them. However, the assembler imposes no conformance rules of ordering or size on declared global variable symbols; you simply declare the ones you need, and the assembler will figure out where to put them so they can be shared with other declarers. (Unlike most high-level languages, sharing of global variable symbols is purely by name!)



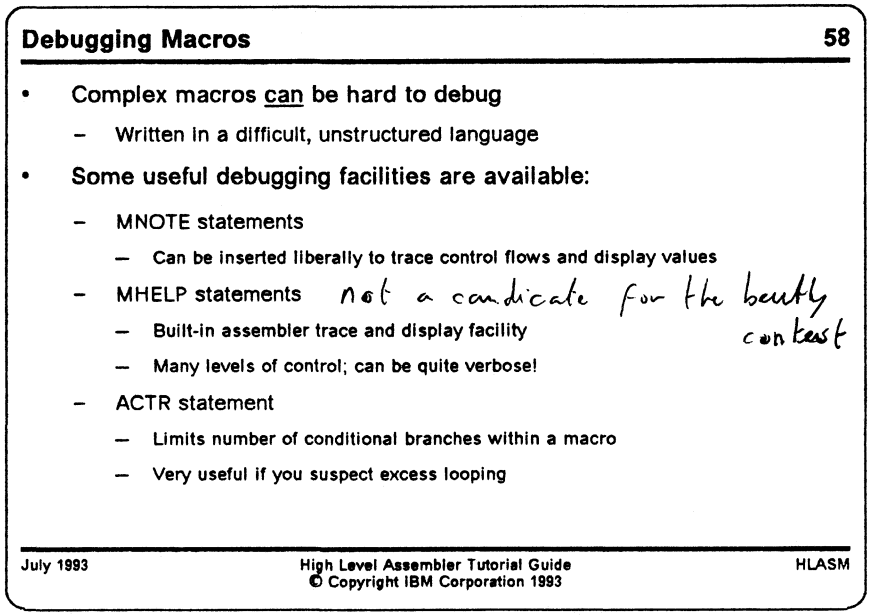


Variable Symbol Scope Rules: Summary

At this point, we will summarize the scope rules for variable symbols.

- Global Variable Symbols are available to all macros and open code that have declared the symbols as GBLx. The three types denoted by "x" are as for local variable symbols: Arithmetic, Boolean, and Character.
- The values of global variable symbols persist through an entire assembly, and their values are kept in a single, common dictionary.
- Local variable symbols include explicitly and implicitly declared variables, symbolic parameters, and local copies of system variable symbols whose value is constant throughout a macro expansion. They are not shared with other macros, or with open code (and vice versa). Open code has its own local dictionary, which is active throughout an assembly.

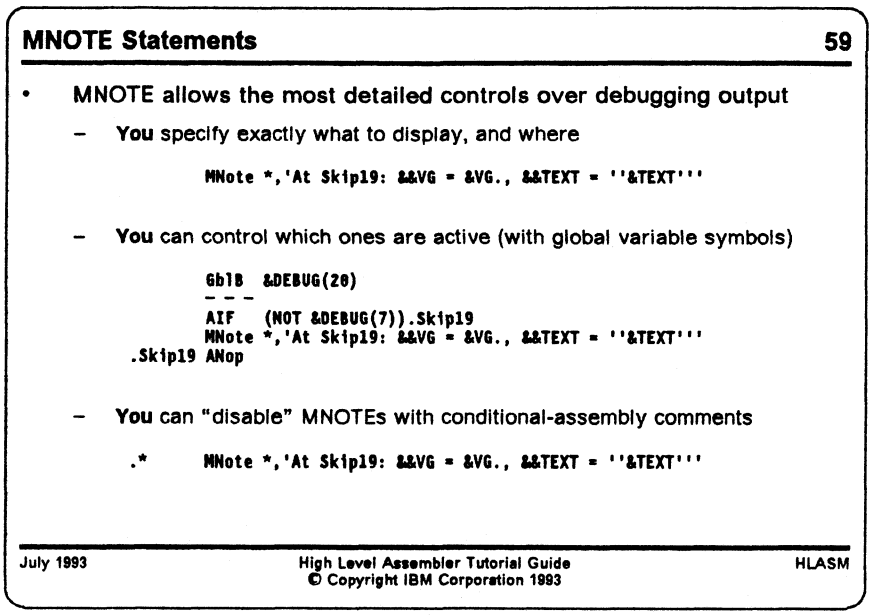
- Variable symbol values are kept in a local, transient dictionary that is created on macro entry, and discarded on macro exit. The symbols are treated as "read-only", meaning that their values are constant throughout a macro, and cannot be changed. Note that recursion still implies a separate dictionary for each entry to the macro; every invocation has its own non-shared dictionary.



Macro Debugging Techniques

No discussion of macros would be complete without some hints about debugging them. The macro language is complex and not well structured, and the "action" inside a macro is generally hidden because each statement is not "displayed" as it is interpreted by the conditional assembly logic of the assembler.

We will briefly describe three statements useful for macro debugging: MNOTE, MHELP, and ACTR.



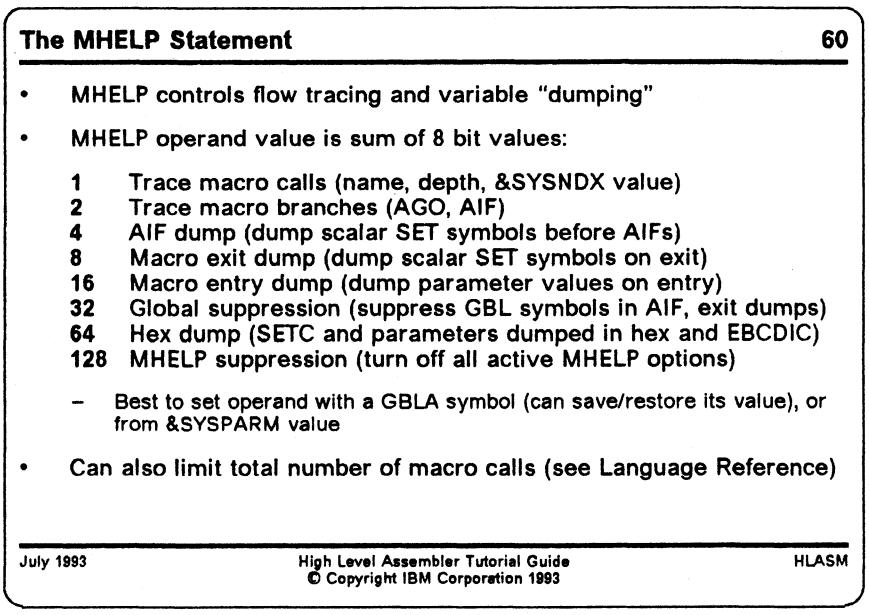
MNOTE Statements

We have already touched on the use of MNOTE statements in "The MNOTE Statement" on page 8. Their main benefits for debugging macros are:

- MNOTE statements may be placed at exactly those points where the programmer knows that internal information may be most useful, and exactly the needed items can be displayed.
- The MNOTE message text can provide specific indications of the internal state of the macro at that point, and why it is being provided.
- Though it requires additional programming effort to insert MNOTE statements in a macro, they can be left "in place", and enabled or disabled at will. Typical controls are as simple as "commenting out" the statement (with a ".*" conditional-assembly comment) to adding global debugging switches to control which statements will be executed, as illustrated here:

```
        GblB  &DEBUG(20)
        - - -
        AIF   (NOT &DEBUG(7)).Skip19
        MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
.Skip19 ANop
```

If the debug switch &DEBUG(7) is 1, then the MNOTE statement will produce the specified output.



The MHELP Statement

The MHELP statement is more general but less specific in its actions than the MNOTE statement. Once an MHELP option is enabled, it stays active until it is reset. The MHELP operand specifies which actions should be activated; the value of the operand is the sum of the "bit values" for each action:

1 Trace macro calls

MHELP 1 produces a single line of information, giving the name of the called macro, its nesting level, and its &SYSNDX number. This information can be used to trace the flow of control among a complex set of macros, because the &SYSNDX value indicates the exact sequence of calls.

2 Trace macro branches

The AIF and AGO branch trace provides a single line of information giving the name of the macro being traced, and the statement numbers of the model statements from which and to which the branch occurs. (Unfortunately, the target sequence symbol name is not provided, nor is branch tracing active for library macros. This latter restriction can be overcome by using the LIBMAC option: if you specify LIBMAC, tracing is active for library macros.

4 AIF dump

When MHELP 4 is active, all the scalar (undimensioned) SET symbols in the macro dictionary (i.e., explicitly or implicitly declared in the macro) are displayed before each AIF statment is interpreted.

8 Macro exit dump

MHELP 8 has the same effect as the preceding (MHELP 4), but the values are displayed at the time a macro expansion is terminated by either an MEXIT or MEND statement.

16 Macro entry dump

MHELP 16 displays the values of the symbolic parameters passed to a macro at the time it is invoked. This information can be very helpful when debugging macros that create or pass complex arguments to inner macros.

32 Global suppression

Sometimes you will use the MHELP 4 or MHELP 8 options to display variable symbols in a macro that has also declared a large number of scalar global symbols, and you are only interested in the local variable symbols. Setting MHELP 32 suppresses the display of the global variable symbols.

64 Hex dump

When used in conjunction with any of MHELP's "display" options (MHELP 4, 8, and 16), causes the value of displayed SETC symbols to be produced in both character (EBCDIC) and hexadecimal formats. If you are using character string data that contains non-printing characters, this option can help with understanding the values of those symbols.

128 MHELP suppression

Setting MHELP 128 will suppress all currently active MHELP options. (MHELP 0 will do the same.)

These values are additive: you may specify any combination. Thus,

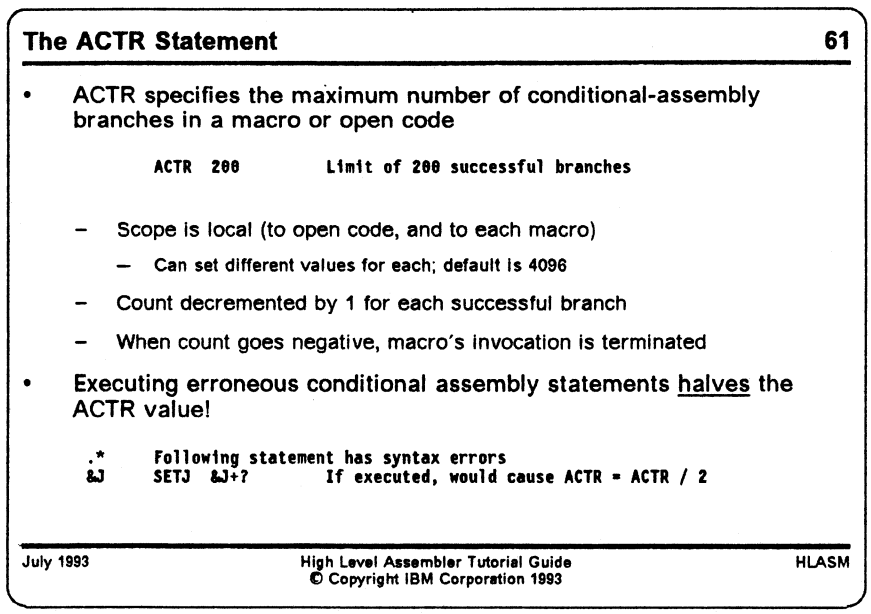
```
MHELP  1+2              Trace macro calls and AIFs
```

will request both macro call tracing and AIF branch tracing.

As you might infer from the values just described, these MHELP "switches" fit in a single byte. The actions of the MHELP facility are controlled by a fullword in the assembler, of which these values are the rightmost byte. The remaining three high-order bytes can be used to control the maximum number of macro calls allowed in an assembly; the details are described in the IBM High Level Assembler/MVS & VM & VSE *Language Reference* manual.

The output of the MHELP statement can sometimes be quite voluminous, especially if multiple traces and dumps are active. It is particularly useful in situations where the macro(s) you are debugging were ones you didn't write, and in which you cannot conveniently insert MNOTE statements. Also, if macro calls are nested deeply, the MHELP displays can help with understanding the actions taken by each inner macro.

To provide some level of dynamic control over the MHELP options in effect, it is useful to set a global arithmetic variable outside the macros to be traced, and then refer to that value inside any macro where the options might be modified; the MHELP operand can then be saved in a local arithmetic value, and restored to its "global" value on exit. Another useful technique is to derive the MHELP operand from the &SYSPARM string supplied to the assembler at invocation time; this lets you debug macros without making any changes to the program's source statements.



The ACTR Statement

The ACTR statement can be use the limit the number of conditional assembly branches (AIF and AGO) executed within a macro invocation (or in open code). It is written

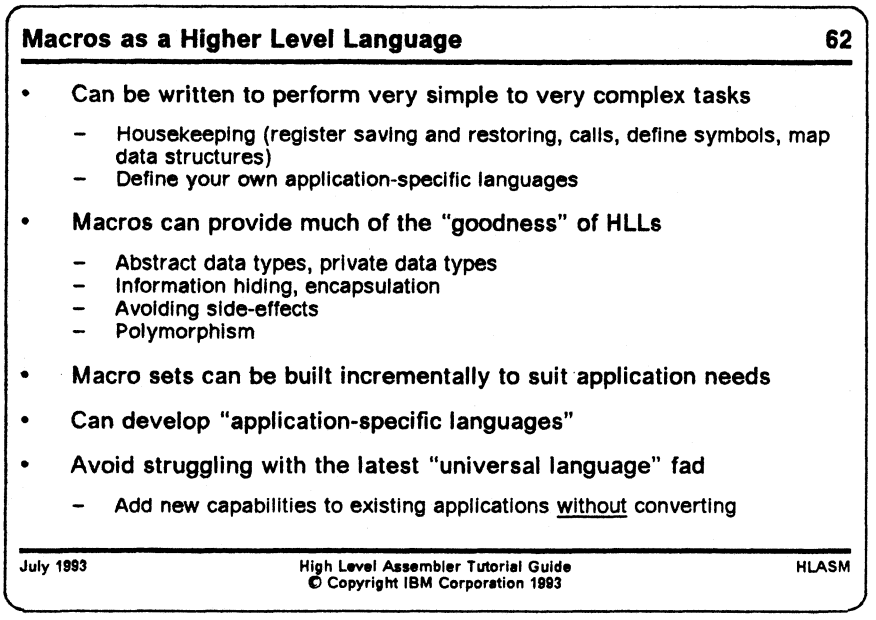
```
ACTR  arithmetic_expression
```

where the value of the "arithmetic_expression" will be used to set an upper limit on the number of branches executed by the assembler. In the absence of an ACTR statement, the default ACTR value is 4096, which is adequate for most macros.

ACTR is most useful in two situations:

1. If you suspect a macro may be looping or branching excessively, you can set a lower ACTR value to limit the number of branches.
2. If a very large or complex macro must make a large number of branches, you can set an ACTR value high enough that all normal expansions can be handled safely.

Macro Techniques



Macro instructions (or *macros* for short) provide the Assembler Language programmer with a wonderfully flexible set of possibilities. Macros share many of the properties of ordinary subroutines that can be called from many different applications: once written, they may be used for many other tasks. Their capabilities range from the very simple:

- perform "housekeeping" such as saving registers, making subroutine calls, and restoring the registers and returning (the operating system supplies the SAVE, CALL, and RETURN macros for these functions)
- define symbols for registers and fixed storage areas, and declare data structures to define or map certain system control blocks used by programs to communicate with the operating system (macros such as REGEQU, DCB, and DCBD)

to the very complex:

- macros have been written to define "artificial languages" in which entire applications are written (examples include the SNOBOL4 language, and certain banking and teleprocessing applications).

Part of our purpose here will be to show that you can write macros to simplify almost any part of the programming process, from the simplest and smallest to the very complex and powerful.

Higher-level languages are often deemed useful because they provide desirable "advanced" features. We will see that macros can also deliver most of these features:

- *Abstract Data Types:* — are user-specified types for data objects, and sets of procedures used to access and manipulate them. This "encapsulation" of data items and logic is one of the key benefits claimed for object-oriented programming.
- *Information Hiding:* — is an established technique for hiding the details of an implementation from the user. The concept of separating application logic from data representations is and old and well established programming principle.

- *Private Types:* — are user-defined data types for which the implementing procedures are hidden.
- *Avoiding Side-Effects:* — is achieved by having functions only return a value without altering either input values or setting of shared variables not declared in the invocation of an implementing procedure.
- *Polymorphism:* allows functions to accept arguments of different types. and enhances the possibility of reusing components in many contexts.

We will see that macros provide simple ways to implement any or all of these features. They provide some additional advantages:

- Macros may be written to perform as much or as little as is needed for a particular task.
- Macros can be built incrementally, so that simple functions can be used by more complex functions, as they are written.
- New "language" implemented by macros can be adapted to the needs of the application, giving an application-specific language that may well be better suited to its needs than a general-purpose "higher level" language designed to (nearly) fit (nearly) everything.

Examples

We will now examine some sample macros that illustrate various aspects of the macro language.

We will discuss several sets of example macros that illustrate different aspects of macro coding, and which provide various types of useful functions.

1. The example macros at "Defining Equated Symbols for Registers (Safely)" on page 74 generate a set of EQU statements to define symbols to be used for register references. They illustrate the use of a global variable symbol to set a "one-time" switch, text parameterization, use of the &SYSLIST system variable symbol, and created variable symbols.
2. Two example macros at "Generating a Byte Sequence" on page 77 generate a sequence of byte values. They illustrate conditional assembly statements, and some simple string-handling operations.
3. The "utility" macros at "Macro-Time Conversion Between Hex and Decimal" on page 79 might be used by other macros to perform conversions between decimal and hexadecimal representations. They illustrate construction of self-defining terms, global variables for communicating among macros, and substring operations.
4. The example macro at "Generate Lists of Named Integer Constants" on page 81 generates a list of constants from a varying-length list of arguments, using &SYSLIST to refer to each argument in turn, and constructs a name for each constant using its value.
5. The three example macros at "Creating a Prefixed Message Text" on page 83 generate a length-prefixed "message" string. The first and second examples illustrate some familiar techniques, while the third uses the AREAD statement and a full scan of a "human-format" message to generate an insertion-text character string for the final DC statement containing the message.
6. Three example macros at "Macro Recursion" on page 88 illustrate recursive macro calls. The first implements "indirect addressing", and the remaining two illustrate the use of global variable symbols and recursive macro calls to generate factorials and Fibonacci numbers.
7. Several macros at "Bit Handling" on page 94 illustrate techniques that can be used to define a private "bit" data type, with bit addressing by name and type checking within the bit handling macros themselves.

8. A set of macros illustrated at "Using and Defining Data Types" on page 117 illustrate some techniques that can be used to implement type-sensitive operations ("polymorphism"), and user-defined data types and user-defined operations on them, with type checking and information hiding.

Define General Register Equates (Simply) 63

- Easy to do this with a macro like

```
         MACRO
         GREGS
GR0      EQU   0
GR1      EQU   1
.*       - - -   etc.
GR15     EQU   15
         MEND
```

- Problem: what if two code segments are combined, and each calls GREGS?
  - How to preserve modularity?
- Answer: use a global variable symbol!

Define General Register Equates (Safely) 64

- Use a global variable symbol &GRegs to check for previous calls

```
         MACRO
         GREGS
         GBLB    &GRegs
         AIF     (&GRegs).Done
         LCLA    &N
.X       ANOP
GR&N     EQU     &N
&N       SETA    &N+1
         AIF     (&N LE 15).X
&GRegs   SetB    1     Indicate definitions have been done
         MEXIT
.Done    MNOTE   0,'GREGS previously called, this call ignored.'
         MEND
```

- If &GRegs is true, no statements are generated

```
         GREGS  This,Call,Is,Ignored
```

Defining Equated Symbols for Registers (Safely)

The technique illustrated in "Example: Defining Equated Symbols for Registers" on page 47 is quite acceptable unless we need at some point to combine multiple code segments, each of which may possible contain a call to GREGS (which was needed for its own modular development). How can we avoid generating multiple copies of the EQU statements, with their accompanying diagnostics for multiply-defined symbols?

The solution is simple: use a global variable symbol whose value will indicate that the GREGS macro has been called already. This is illustrated in Figure 20 below.

```
         MACRO
         GREGS
         GBLB    &GRegs
         AIF     (&GRegs).Done
         LCLA    &N
.X       ANOP
GR&N     EQU     &N
&N       SETA    &N+1
         AIF     (&N LE 15).X
&GRegs   SetB    1     Indicate definitions have been done
         MEXIT
.Done    MNOTE   0,'GREGS -- Previously called, this call ignored.'
         MEND

AAA      GREGS   What?
         GREGS   Again,Eh?
```

Figure 20. Macro to Define General Purpose Registers Once Only

Define All Register Equates (Safely) 65

- Use "created set symbols" for global variable symbol references

```
        MACRO
        REGS
.*  General macro to define General, Floating, or Access Registers.
.*  Just specify the argument A, F, or G for the appropriate set.
        GBLB    &GRegs_Done,&FRegs_Done,&ARegs_Done
        AIF     (N'&SysList eq 0).Exit
&J      SetA    1                   Initialize argument counter
.GetArg ANOP
&T      SetC    '&SysList(&J)'      Pick up an argument
        AIF     ('&T' ne 'G' and '&T' ne 'F' and '&T' ne 'A').Bad
        AIF     (&(&T.Regs_Done)).Done
&T.R0   EQU     0
&T.R2   EQU     2
&T.R4   EQU     4
&T.R6   EQU     6
        AIF     ('&T' eq 'F').Set  If type F, only 4 regs to define
&T.R1   EQU     1
.*      - - -   continued   - - -
```


Define All Register Equates (Safely) ... 66

```
&T.R3   EQU     3
.*      - - -   etc.
&T.R15  EQU     15
.Set    ANOP
&(&T.Regs_Done) SetB 1      Indicate definitions have been done
.Next   ANOP
&J      SetA    &J+1                Count to next argument
        AIF     (&J le N'&SysList).GetArg   Get next argument
        MEXIT
.Bad    MNOTE   8,'REGS — Unknown type ''&T.''.'
        MEXIT
.Done   MNOTE   0,'REGS — Previously called for type &T..'
        AGO     .Next
.Exit   MEND
```

- If &(&T.Regs_Done) is **true**, no statements are generated

```
        GREGS   G
        GREGS   G,F,A       G registers are not defined again
```


Encouraged by the success of this approach, we might wish to define a macro which will create equates for *all* the registers we might use in our program: General Purpose, Floating Point, and Access. Rather than write three separate macros (one for each type of register), we can write a single REGS macro whose operands specify the type of registers desired (e.g., "G" for GPRs, "F" for FPRs, and "A" for ARs).

The following example uses the technique illustrated in Figure 20 on page 74 above, but generalizes it by using a "created set symbol" to select the name of the proper global variable symbol.

```
        MACRO
        REGS
.*  General macro to define General, Floating, or Access Registers.
.*  Just specify the argument A, F, or G for the appropriate set.
        GBLB    &GRegs_Done,&FRegs_Done,&ARegs_Done
        AIF     (N'&SysList eq 0).Exit
&J      SetA    1                   Initialize argument counter
.GetArg ANOP
&T      SetC    '&SysList(&J)'      Pick up an argument
        AIF     ('&T' ne 'G' and '&T' ne 'F' and '&T' ne 'A').Bad
        AIF     (&(&T.Regs_Done)).Done
&T.R0   EQU     0
&T.R2   EQU     2
&T.R4   EQU     4
&T.R6   EQU     6
        AIF     ('&T' eq 'F').Set  If type F, only 4 regs to define
&T.R1   EQU     1
&T.R3   EQU     3
.*      - - -   etc.
&T.R15  EQU     15
.Set    ANOP
&(&T.Regs_Done) SetB 1      Indicate definitions have been done
.Next   ANOP
&J      SetA    &J+1                Count to next argument
        AIF     (&J le N'&SysList).GetArg   Get next argument
        MEXIT
.Bad    MNOTE   8,'REGS -- Unknown type ''&T.''.'
        MEXIT
.Done   MNOTE   0,'REGS -- Previously called for type &T..'
        AGO     .Next

        REGS    A
        REGS    F,G,A
.Exit   MEND
```

Figure 21. Macro to Define Any Sets of Registers Once Only

This REGS macro may be safely used any number of times (so long as no other definitions of the global variable symbols &ARegs_Done, &FRegs_Done, or &GRegs_Done, appear elsewhere in the program!).

Generating a Byte Sequence: BYTESEQ1 Macro 67

- BYTESEQ1 generates a byte for each value

```
        MACRO
&L      BYTESEQ1 &N
.*  BYTESEQ1 — generate a sequence of byte values, one per statement.
.*  No checking or validation is done.
        LclA  &K
        AIF   ('&L' EQ '').Loop  Don't define the label if absent
&L      DS    0AL1               Define the label
.Loop   ANOP
&K      SetA  &K+1               Increment &K
        AIF   (&K GT &N).Done    Check for termination condition
        DC    AL1(&K)
        AGO   .Loop              Continue
.Done   MEND

* Two test cases

BS1a    BYTESEQ1  5
        BYTESEQ1  1
```


Generating a Byte Sequence: BYTESEQ2 Macro 68

```
        MACRO
&L      BYTESEQ2 &N              Generates a single DC statement
&K      SetA     1               Initialize generated value counter
&S      SetC     '1'             Initialize output string
        AIF      (T'&N EQ 'N').Num   Validate type of argument
        MNOTE    8,'BYTESEQ2 — &&N=&N not self-defining.'
        MEXIT
.Num    AIF      (&N LE 255).NotBig  Check size of argument
        MNOTE    8,'BYTESEQ2 — &&N=&N is too large.'
        MEXIT
.NotBig AIF      (&N GT 0).OK        Check for small argument
        MNOTE    *,'BYTESEQ2 — &&N=&N too small, no data generated.'
        MEXIT
.OK     AIF      (&K GE &N).DoDC     If done, generate DC statement
&K      SetA     &K+1            Increment &K
&S      SetC     '&S.'.',&K'     Add comma and new value of &K to &S
        AGO      .OK             Continue
.DoDC   ANOP
&L      DC       AL1(&S)
        MEND
```


Generating a Byte Sequence

The sample BYTESEQ2 macro illustrated below in Figure 22 on page 78 uses the same techniques as the conditional-assembly examples given in Figure 4 on page 27 and Figure 5 on page 27. and the corresponding BYTESEQ1 macro illustrated in Figure 18 on page 53.

The BYTESEQ2 macro shown in Figure 22 on page 78 performs several validations of its argument, including a type attribute reference to verify that the argument is a self-defining term. As its output, the macro generates a single DC statement for the byte values, but it has a curious limitation: can you tell what it is, without reading the text following the next figure?

```
        MACRO
&L      BYTESEQ2 &N
.*  BYTESEQ2 -- generate a sequence of byte values, one per statement.
.*  The argument is checked and validated, and the entire constant is
.*  generated in a single DC statement.
        LclA    &K
        LclC    &S
&K      SetA    1             Initialize generated value counter
&S      SetC    '1'           Initialize output string
        AIF     (T'&N EQ 'N').Num   Validate type of argument
        MNOTE   8,'BYTESEQ2 -- &&N=&N not self-defining.'
        MEXIT
.Num    AIF     (&N LE 255).NotBig  Check size of argument
        MNOTE   8,'BYTESEQ2 -- &&N=&N is too large.'
        MEXIT
.NotBig AIF     (&N GT 0).OK        Check for small argument
        MNOTE   *,'BYTESEQ2 -- &&N=&N too small, no data generated.'
        MEXIT

.OK     AIF     (&K GE &N).DoDC     If done, generate DC statement
&K      SetA    &K+1          Increment &K
&S      SetC    '&S.',&K'     Add comma and new value of &K to &S
        AGO     .OK           Continue
.DoDC   ANOP
&L      DC      AL1(&S)
        MEND
* Test cases
BS2e    BYTESEQ2  0
BS2b    BYTESEQ2  1
BS2a    BYTESEQ2  5
BS2d    BYTESEQ2  X'58'
BS2c    BYTESEQ2  256
```

Figure 22. Macro to Define a Sequence of Byte Values As a Single String

This macro has a problem. Because no character variable symbol may contain more than 255 characters, the argument to BYTESEQ2 may not exceed 88; otherwise &S exceeds 255 characters.

It is easy to modify the AIF test (at sequence symbol .Num) to enforce an upper limit of 88 for &N. We leave as an exercise to the interested reader what steps could be taken to adapt this macro to accept arguments up to and including 255, and still generate a single DC statement.

Macro-Time Conversion from Hex to Decimal 69

- Convert hex digit strings to decimal values in GBLA variable &DEC

```
         Macro
         Dec   &Hex              Convert &Hex to decimal
         GblA  &Dec              Decimal value returned in &Dec
&X       SetC  'X''&Hex'''       Create hex self-defining term
&Dec     SetA  &X                Do the conversion
         MNote 0,'&Hex (hex) = &Dec (decimal)'   For debugging
         MEnd
*
         Dec   AA
  *** MNOTE ***  0,AA (hex) = 170 (decimal)
         Dec   FFF
  *** MNOTE ***  0,FFF (hex) = 4095 (decimal)
         Dec   FFFFFF
  *** MNOTE ***  0,FFFFFF (hex) = 16777215 (decimal)
         Dec   7FFFFFFF
  *** MNOTE ***  0,7FFFFFFF (hex) = 2147483647 (decimal)
```


Macro-Time Conversion from Decimal to Hex 70

- Convert decimal values to hex digit strings in GBLC variable &Hex

```
         Macro
         Hex   &Dec              Convert &Dec to hexadecimal
         GblC  &Hex              Hex value returned in &Hex
&Hex     SetC  ''                Initialize &Hex
&Q       SetA  &Dec              Local working variable
.Loop    ANop  ,                 Top of reduction loop
&R       SetA  &Q-&Q/16*16       &R = Mod ( &Q, 16 )
&Q       SetA  &Q/16             Quotient for next iteration
&Hex     SetC  '0123456789ABCDEF'(&R+1,1).'&Hex'   Build hex value
         Aif   (&Q gt 0).Loop    Repeat if &Q not zero
         MNote 0,'&Dec (decimal) = &Hex (hex)'   For debugging
         MEnd
*
         Hex   170
  *** MNOTE ***   0,170 (decimal) = AA (hex)
         Hex   16777215
  *** MNOTE ***   0,16777215 (decimal) = FFFFFF (hex)
```


Macro-Time Conversion Between Hex and Decimal

If you are writing macros, it is not uncommon to need from time to time to convert between two different representations of a data item. Some of these conversions are already available in the conditional assembly language; for example, arithmetic variables are automatically converted to character form by substituting them in SETC expressions.

To illustrate two "utility" macros, we will show how to convert between decimal and hexadecimal representations. The first macro, Dec, converts from hex to decimal, and places the result of its conversion into the global arithmetic variable &Dec for use by the calling macro (or open code statement). Because the assembler accepts hexadecimal self-defining terms in SETA expressions, the conversion merely needs to construct such a hexadecimal term.

```
        Macro
        Dec   &Hex              Convert &Hex to decimal
        GblA  &Dec              Decimal value returned in &Dec
&X      SetC  'X''&Hex'''       Create hex self-defining term
&Dec    SetA  &X                Do the conversion
        MNote 0,'&Hex (hex) = &Dec (decimal)'   For debugging
        MEnd
```

Figure 23. Macro-Time Conversion Between Hex and Decimal

Some examples of calls to the Dec macro are shown in the following figure, where the MNOTE statement has been used to display the results. (In production use, the MNOTE statement would probably be inactivated by placing a ".*" (conditional-assembly) comment indicator in the first two columns.)

```
        Dec   AA
*** MNOTE ***  0,AA (hex) = 170 (decimal)
        Dec   FFF
*** MNOTE ***  0,FFF (hex) = 4095 (decimal)
        Dec   FFFFFF
*** MNOTE ***  0,FFFFFF (hex) = 16777215 (decimal)
        Dec   7FFFFFFF
*** MNOTE ***  0,7FFFFFFF (hex) = 2147483647 (decimal)
```

Figure 24. Macro-Time Conversion Between Hex and Decimal: Examples

Note that this macro may appear to have a problem: any hex value exceeding X'7FFFFFFF' will not be displayed as a negative number. However, its decimal representation in the variable &Dec will be correct.

Conversion from decimal to hexadecimal requires reducing the decimal value one hex digit at a time, using successive divisions by sixteen.

```
        Macro
        Hex   &Dec              Convert &Dec to hexadecimal
        GblC  &Hex              Hex value returned in &Hex
&Hex    SetC  ''                Initialize &Hex
&Q      SetA  &Dec              Local working variable
.Loop   ANop  ,                 Top of reduction loop
&R      SetA  &Q-&Q/16*16       &R = Mod ( &Q, 16 )
&Q      SetA  &Q/16             Quotient for next iteration
&Hex    SetC  '0123456789ABCDEF'(&R+1,1).'&Hex'   Build hex value
        Aif   (&Q gt 0).Loop    Repeat if &Q not zero
        MNote 0,'&Dec (decimal) = &Hex (hex)'   For debugging
        MEnd
```

Figure 25. Macro-Time Conversion Between Decimal and Hex

Some examples of calls to the Hex macro to perform decimal-to-hex conversion are shown in the following figure.

```
      Hex   170
*** MNOTE ***   0,170 (decimal) = AA (hex)
      Hex   16777215
*** MNOTE ***   0,16777215 (decimal) = FFFFFF (hex)
      Hex   16777216
*** MNOTE ***   0,16777216 (decimal) = 1000000 (hex)
      Hex   2147483647
*** MNOTE ***   0,2147483647 (decimal) = 7FFFFFFF (hex)
```

Figure 26. Macro-Time Conversion Between Decimal and Hex: Examples

The technique shown in the Hex macro could be used to convert from decimal to any other base, simply by replacing occurrences of the value "16" in the macro with the desired base. (As an exercise, rewrite this macro to support a keyword parameter &BASE, with default value 16, and try it with various bases such as 2, 8, and 12.)

Generate a List of Named Integer Constants 71

- Macro to define a list of named integer constants (checking omitted)

```
         MACRO
&Lab     INTCONS &Type=F
.TypOK   AIF     ('&Lab' eq '').NoLab  Skip if no label
&Lab     DC      0&Type.'0'        Define the label
.ArgsOK  ANOP                          Argument-checking loop
&J       SetA    &J+1              Increment argument counter
         AIF     (&J GT N'&SysList).End  Exit if all done
&Name    SetC    '&Type.&SysList(&J)'  Assume non-negative arg
         AIF     ('&SysList(&J)'(1,1) ne '-').NotNeg  Check arg sign
&Name    SetC    '&Type.M'.'&SysList(&J)'(2,K'&SysList(&J)-1)  Neg Arg
.NotNeg  ANOP
&Name    DC      &Type.'&SysList(&J)'
         AGO     .ArgsOK           Repeat for further arguments
.End     MEND
*
C1b      INTCONS  0,-1                      Type F: names F0, FM1
C1c      INTCONS  99,-99,Type=H             Type H: names H99, HM99
```

July 1993 High Level Assembler Tutorial Guide HLASM

Generate Lists of Named Integer Constants

To illustrate a typical use of the &SYSLIST system variable symbol, we suppose we wish to write a macro named INTCONS that will generate integer-valued constants, giving them names by appending their value to a letter designating their type (F if the value is non-negative, or to FM if the value is negative). For good measure, we will provide a keyword parameter to specify their type, either F or H, with F as the default. (Negative halfword constants will then start with the letters HM.)

```
        MACRO
&Lab    INTCONS &Type=F
.*    INTCONS -- assumes a varying number of positional arguments
.*    to be generated as integer constants, with created names.
.*    Type will be F (default) or H if specified.
        LclA   &J              Count of arguments
        LclC   &Name           Name of the constant
.*    Validate the Type argument
        AIF    ('&Type' eq 'F' OR '&Type' eq 'H').TypOK  Check Type
        MNOTE  8,'INTCONS -- Invalid Type=''&Type''.'
        MEXIT
.*    Generate the name-field symbol &Lab if provided
.TypOK  AIF    ('&Lab' eq '').NoLab  Skip if no label
&Lab    DC     0&Type.'0'      Define the label
.*    Verify that arguments are present; no harm if none.
.NoLab  AIF    (N'&SysList gt 0).ArgsOK  Check presence of args
        MNOTE  *,'INTCONS -- No arguments provided.'
        MEXIT
.*    Argument-checking loop
.ArgsOK ANOP
&J      SetA   &J+1            Increment argument counter
        AIF    (&J GT N'&SysList).End  Exit if all done
        AIF    (K'&SysList(&J) gt 0).DoArg
        MNOTE  4,'INTCONS -- Argument No. &J. is empty.'
        AGO    .ArgsOK         Go for next argument
.DoArg  ANOP
&Name   SetC   '&Type.&SysList(&J)'  Assume non-negative arg
        AIF    ('&SysList(&J)'(1,1) ne '-').NotNeg  Check arg sign
&Name   SetC   '&Type.M'.'&SysList(&J)'(2,K'&SysList(&J)-1)  Neg Arg
.NotNeg ANOP
&Name   DC     &Type.'&SysList(&J)'
        AGO    .ArgsOK         Repeat for further arguments
.End    MEND
```

Figure 27. Macro Parameter-Argument Association Example: Create a List of Constants

Some test cases for the INTCONS macro are shown in the following figure:

```
*   Test cases -- first has no label, no args; second has no args.
        INTCONS
C1a     INTCONS
C1b     INTCONS  0,-1
C1c     INTCONS  99,-99,Type=H              Type H
C1d     INTCONS  -000000000,2147483647
C1e     INTCONS  1,2,3,4,Type=D             Invalid type
        INTCONS  1,2,3,4,,5,6,7,8,9,10E7    Null 5th argument
```

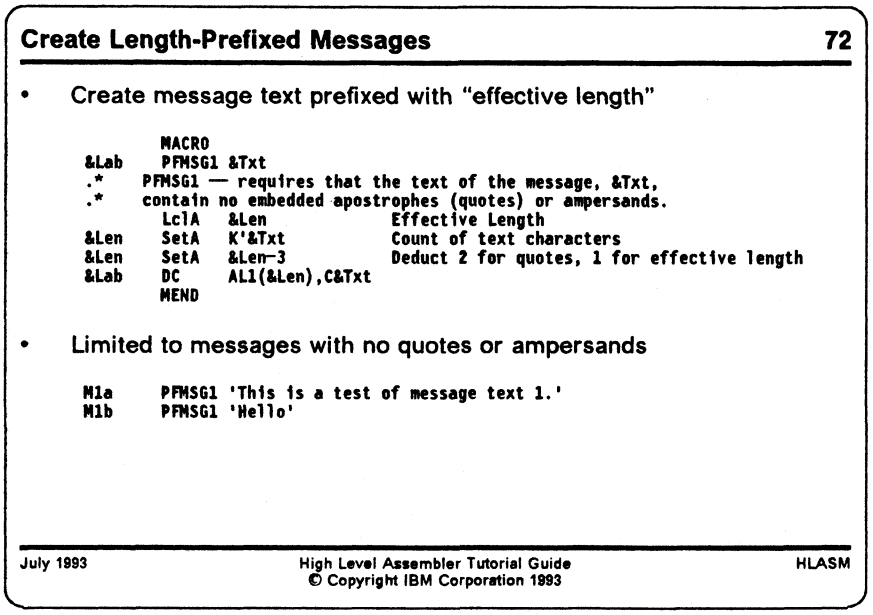
Figure 28. Macro Example: List-of-Constants Test Cases

The basic structure of this macro is in two parts: the first (through the second MEXIT statement, following the MNOTE statement for null arguments) checks the values and validity of the arguments, issuing various messages for cases that do not satisfy the constraints of the definition.

The second part (beginning at the sequence symbol .ArgsOK) uses the &SYSLIST system variable symbol to step through each of the positional arguments in turn, by applying a sub-

script (&J) to indicate which positional argument is desired. The argument is checked for being non-null, and then to see if its first character is a minus sign. If the minus sign is present, it is removed for constructing the constant's name; finally, the constant is generated with the required name.

As an interesting exercise: what would happen if you wished to add a test to verify that each argument is a valid self-defining term? Are negative arguments valid? Would the argument 10E7 be valid? (It's acceptable as a nominal value in an F-type constant.)



Creating a Prefixed Message Text

A common need in many applications is to produce messages. Often, the length of the message must be reduced by 1 prior to executing a move instruction, so it is helpful to store the message text and its "effective length" (i.e., its true length minus one).

We will illustrate three macros to create message texts with an effective-length prefix.

In this first example, the text of the message may not contain any "special" characters, namely apostrophes (quotes) or ampersands. A Count attribute reference is used to determine the number of characters in the message argument.

```
        MACRO
&Lab    PFMSG1 &Txt
.*    PFMSG1 -- requires that the text of the message, &Txt,
.*    contain no embedded apostrophes (quotes) or ampersands.
        LclA   &Len             Effective Length
&Len    SetA   K'&Txt           Count of text characters
&Len    SetA   &Len-3           Deduct 2 for quotes, 1 for effective length
&Lab    DC     AL1(&Len),C&Txt
        MEND

M1a     PFMSG1 'This is a test of message text 1.'
M1b     PFMSG1 'Hello'
```

Figure 29. Macro to Define a Length-Prefixed Message

Create General Length-Prefixed Messages 73

- Allow all characters in message text (pairing may be required)

```
        MACRO
&Lab    PFMSG2 &Txt
.*    PFMSG2 — the text of the message, &Txt, may contain embedded
.*    apostrophes (quotes) or ampersands, so long as they are
.*    properly paired. The macro expansion generates a symbol
.*    using the &SYSNDX system variable symbol, and uses a Length
.*    attribute reference for the effective length.
        LclC   &T               Local variable
&T      SetC   'TXT&SYSNDX.M'   Create symbol to name the text string
&Lab    DC     AL1(L'&T.-1)     Effective length
&T      DC     C&Txt
        MEND
```

- Quote and ampersand pairs are harder to write, read and translate

```
M2a     PFMSG2 'Test of ''This'' && ''That''.'
M2b     PFMSG2 'Hello'
```

July 1993 High Level Assembler Tutorial Guide HLASM

The requirement that no ampersands or quotes may be used in the message text defined by PFMSG1 may not be acceptable in some situations. Thus, in Figure 30 on page 85 we will define a second macro PFMSG2 that allows such characters in the message, but requires that they be properly paired in the argument string. It also generates an ordinary symbol so that a length attribute reference may be used.

```
        MACRO
&Lab    PFMSG2 &Txt
.*    PFMSG2 -- the text of the message, &Txt, may contain embedded
.*    apostrophes (quotes) or ampersands, so long as they are
.*    properly paired. The macro expansion generates a symbol
.*    using the &SYSNDX system variable symbol, and uses a Length
.*    attribute reference for the effective length.
        LclC   &T               Local variable
&T      SetC   'TXT&SYSNDX.M'   Create symbol to name the text string
&Lab    DC     AL1(L'&T.-1)     Effective length
&T      DC     C&Txt
        MEND

M2a     PFMSG2 'Test of ''This'' && ''That''.'
M2b     PFMSG2 'Hello'
```

Figure 30. Macro to Define a Length-Prefixed Message With Paired Characters

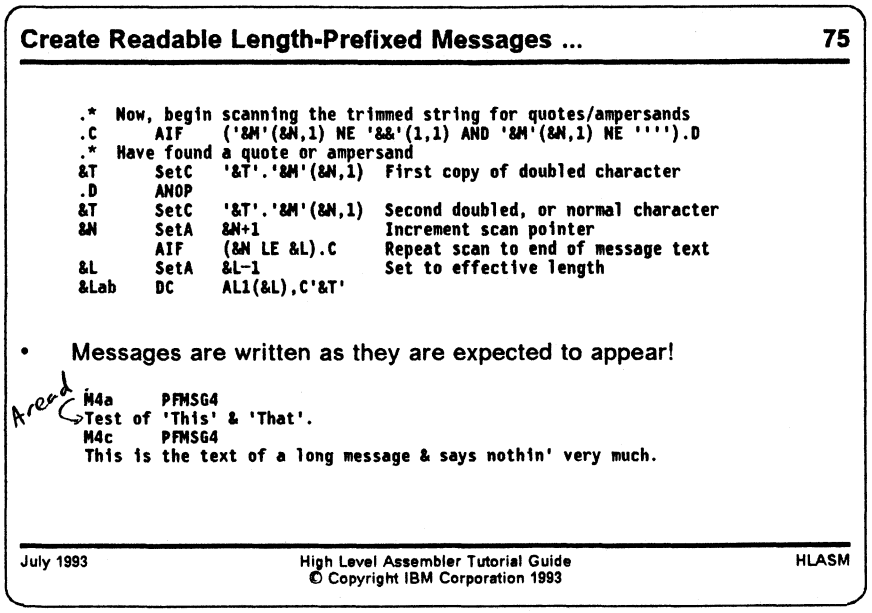
One interesting feature of this macro is its use of the &SYSNDX system variable symbol. The value of &SYSNDX is incremented by one for every macro call, and the value assigned to a given macro macro remains constant throughout its expansion. Thus, &SYSNDX may be used to generate symbols that are (much more likely to be) different for every macro expansion.

While the PFMSG2 macro defined in this example allows any characters in the message text, it is much more difficult to read and understand the macro argument. (Consider, for example, how to explain the odd rules about pairing quotes and ampersands to someone who wants to translate the message text into a different language!)

Create Readable Length-Prefixed Messages 74

- Allow all characters in message text without pairing, using AREAD

```
        MACRO
&Lab    PFMSG4
.*    PFMSG4 — the text of the message may contain any characters.
.*    The message is on a single line following the call to PFMSG4.
        LclA   &L,&N            Local arithmetic variables
        LclC   &T,&C,&M         Local character variables
        AIF    (N'&SYSLIST EQ 0).OK   No arguments allowed
        MNote  8,'PFMSG4 — no operands should be provided.'
        MEXIT                   Terminate macro processing
.OK     ANOP
&N      SetA   1                Initialize char-scan pointer to 1
.* Read the record following the PFMSG4 call into &M
&M      ARead  advance read (next stmt after macro call)
&M      SetC   '&M'(1,72)       Trim off any sequence field
&L      SetA   72               Point to end of initial text string
.* Trim off trailing blanks from message text
.Trim   AIF    ('&M'(&L,1) NE ' ').C  Check last character
&L      SetA   &L-1             Deduct blanks from length
        AGO    .Trim            Repeat trimming loop
```



This limitation can be removed by using an elegant and powerful feature of the macro language, the AREAD statement.

The AREAD Statement

The AREAD statement can be used in a macro to read lines from the program into a character variable symbol in the macro. If we write

```
&CVar  AREAD
```

then the first statement in the main program following the macro containing the AREAD statement (or the macro call that eventually resulted in interpreting the AREAD statement) will be "read" by the assembler, and the contents of that record will be assigned to the variable symbol &CVar.

We will exploit this capability in the PFMSG4 macro, which reads the text of a message written in its desired final form from the line following the macro call. The macro illustrated in Figure 31 on page 87 scans the text of the string, creating pairs of quotes and ampersands wherever needed; thus, the writer of the message need not be aware of the peculiar rules of the Assembler Language.

```
        MACRO
&Lab    PFMSG4
.*    PFMSG4 -- the text of the message may contain any characters.
.*    The message is on a single line following the call to PFMSG4,
.*    in the form it is expected to take when printed. The macro
.*    scans for quotes and ampersands, and creates a pair for each
.*    of them for the generated constant. The macro includes a test
.*    for the presence of an argument, which should not be present.
        LclA   &L,&N            Local arithmetic variables
        LclC   &T,&C,&M         Local character variables
        AIF    (N'&SYSLIST EQ 0).OK   No arguments allowed
        MNote  8,'PFMSG4 -- no operands should be provided.'
        MEXIT                   Terminate macro processing
.OK     ANOP
&N      SetA   1                Initialize char-scan pointer to 1
.*  Read the record following the PFMSG4 call into &M
&M      ARead
&M      SetC   '&M'(1,72)       Trim off any sequence field
&L      SetA   72               Point to end of initial text string
.*  Trim off trailing blanks from message text
.Trim   AIF    ('&M'(&L,1) NE ' ').C  Check last character
&L      SetA   &L-1             Deduct blanks from length
        AGO    .Trim            Repeat trimming loop
.*  Now, begin scanning the trimmed string for quotes/ampersands
.C      AIF    ('&M'(&N,1) NE '&&'(1,1) AND '&M'(&N,1) NE '''').D
.*  Have found a quote or ampersand
&T      SetC   '&T'.'&M'(&N,1)  First copy of doubled character
.D      ANOP
&T      SetC   '&T'.'&M'(&N,1)  Second doubled, or normal character
&N      SetA   &N+1             Increment scan pointer
        AIF    (&N LE &L).C     Repeat scan to end of message text
&L      SetA   &L-1             Set to effective length
&Lab    DC     AL1(&L),C'&T'
        MEND
```

Figure 31. Macro to Define a Length-Prefixed Message With "True Text"

Some test cases for the PFMSG4 macro are shown in the following figure:

```
M4a     PFMSG4
Test of 'This' & 'That'.
M4b     PFMSG4
Hello
M4c     PFMSG4
This is the text of a long message & says nothin' very much.
```

Figure 32. Test Cases for Macro With "True Text" Messages

Macro Recursion

Macros that call themselves either directly or indirectly are *recursive*. We will first illustrate a recursive call with a simple "Load Indirect" macro, which introduces a simple form of indirect addressing.

Indirect Addressing via Recursion 76

- LI macro calls itself for each level of indirection

```
         Macro
&Lab     LI    &Reg,&X           Load &Reg with indirection
         Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
&Lab     L     &Reg,&X
         MExit
.Ind     ANop
&XI      SetC  '&X'(2,K'&X-1)    Strip off leading asterisk
         LI    &Reg,&XI          Call ourselves recursively
         L     &Reg,0(,&Reg)
         MEnd
```

- Examples: each asterisk specifies a level of indirection

```
         LI    3,0(4)            Load from 0(4)
         LI    3,*0(,4)          Load from what 0(,4) points to
         LI    3,**0(,7)         Two levels of indirection
         LI    3,***X            Three levels of indirection
```

July 1993 High Level Assembler Tutorial Guide HLASM

Example 1: Indirect Addressing

In Figure 33 on page 89, the LI macro implements a form of "indirect" addressing: if the storage operand is preceded by an asterisk, the assembler interprets this as meaning that the operand to be loaded into the register is not at the operand, but is at the address specified by the operand without the asterisk.[4] Thus, if an instruction was written as

```
LI    8,*XXX              Indirect reference via XXX
```

then the item to be loaded into R8 is not at XXX, but is at the position "pointed to" by XXX.

This definition is recursive, in the sense that the "operand" preceded by an asterisk may itself be preceded by an asterisk, which thus provides multiple levels of indirection. A macro to implement this form of indirect addressing is shown in Figure 33 on page 89.

[4] Indirect addressing was a popular hardware feature in many second-generation computers, such as the IBM 709-7090-7094 series. The hardware supported only a single level of indirect addressing, and the instruction syntax was slightly different on those machines: a single asterisk could be appended to the mnemonic (as in TRA*), but the operand field was not modified.

```
        Macro
&Lab    LI    &Reg,&X           Load &Reg with indirection
        Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
&Lab    L     &Reg,&X
        MExit
.Ind    ANop
&XI     SetC  '&X'(2,K'&X-1)    Strip off leading asterisk
        LI    &Reg,&XI          Call ourselves recursively
        L     &Reg,0(,&Reg)
        MEnd
```

Figure 33. Recursive Macro to Implement Indirect Addressing

Some examples of calls to the LI macro are shown in Figure 34, where the "+" characters at the left margin are the assembler's indication of a macro-generated statement.

```
        LI    3,0(4)            Load from 0(4)
+       L     3,0(4)

        LI    3,*0(,4)          Load from what 0(,4) points to
+       L     3,0(,4)
+       L     3,0(,3)

        LI    3,**0(,7)         Two levels of indirection
+       L     3,0(,7)
+       L     3,0(,3)
+       L     3,0(,3)

        LI    3,***X            Three levels of indirection
+       L     3,X
+       L     3,0(,3)
+       L     3,0(,3)
+       L     3,0(,3)
```

Figure 34. Recursive Macro to Implement Indirect Addressing: Examples

```
        Macro
&Lab    FACTO1   &N
.* Factorials defined by Fac(N) = N * Fac(N-1), Fac(0) = Fac(1) = 0
        GBLA     &Ret             For returning values of inner calls
        AIF      (T'&N NE 'N').Error   N must be numeric
&L      SetA     &N               Convert from external form
        MNote    0,'Evaluating FACTO1(&L.)'
        AIF      (&L LT 0).Error  Can't handle N < 0
        AIF      (&L GE 2).Calc   Calculate via recursion if N > 1
&Ret    SetA     1                F(0) = F(1) = 1
        AGO      .Test            Return to caller
.Calc   ANOP
&K      SetA     &L-1
&Temp   SetA     &Ret
        FACTO1   &K
&Ret    SetA     &Ret*&L
.Test   AIF      (&SysNest GT 1).Cont
        MNote    0,'Factorial(&L.) = &Ret.'
&Lab    DC       F'&Ret'
.Cont   MExit                     Return to caller
.Error  MNote    11,'Invalid Factorial argument &N..'
        MEnd
```


Example 2: Factorial Function Values

Probably the best-known recursive function is the Factorial function. It can be defined and implemented iteratively (and more simply), but its familiarity makes it useful as an example.

In the macro in Figure 35 on page 91, the macro FACTO1 uses the global arithmetic variable symbol &Ret to return calculated values.

There are many ways to test for the end of a recursive calculation. In this example, the &SYSNEST variable symbol is used to check the "nesting" level at which the macro was called. Macros called from open code are at level 1.

```
FACT   Title  'Factorial Numbers by Macro Recursion'
*
*      Factorial Numbers are defined by
*           F(N) = N * F(N-1)
*           with F(0) = 1 and F(1) = 1
*
       Macro
&Lab   FACT01    &N
       GBLA      &Ret              For returning values of inner calls
       LCLA      &Temp,&K,&L       Local variables
       AIF       (T'&N NE 'N').Error   N must be numeric
&L     SetA      &N                Convert from external form
       MNote     0,'Evaluating FACT01(&L.)'
       AIF       (&L LT 0).Error   Can't handle N < 0
       AIF       (&L GE 2).Calc    Calculate via recursion if N > 1
&Ret   SetA      1                 F(0) = F(1) = 1
       AGO       .Test             Return to caller
.Calc  ANOP
&K     SetA      &L-1
&Temp  SetA      &Ret
       FACT01    &K
&Ret   SetA      &Ret*&L
.Test  AIF       (&SysNest GT 1).Cont
       MNote     0,'Factorial(&L.) = &Ret.'
&Lab   DC        F'&Ret'
.Cont  MExit                       Return to caller
.Error MNote     11,'Invalid Factorial argument &N..'
       MEnd
```

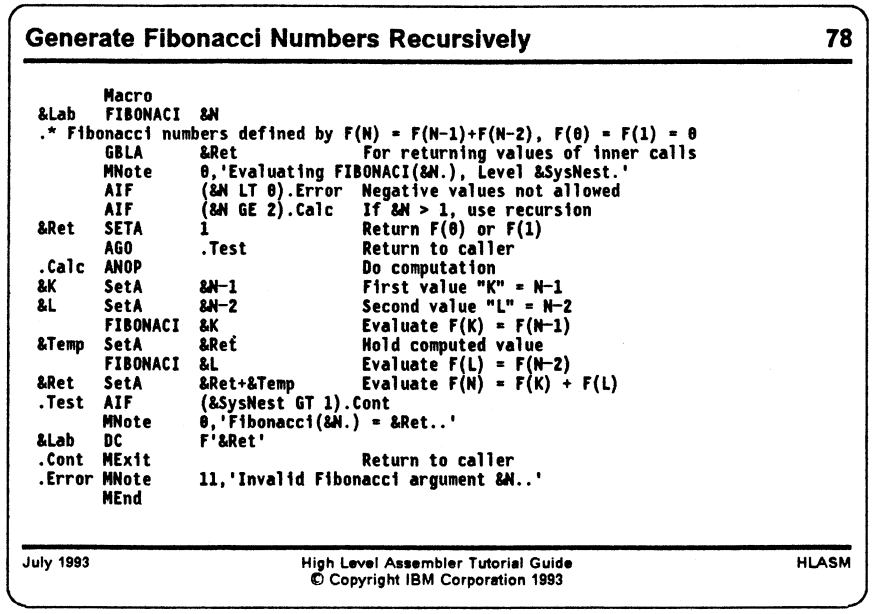
Figure 35. Macro to Calculate Factorials Recursively

Some test cases for the FACT01 macro are shown in the following figure:

```
*  Test cases
       FACT01    0
       FACT01    1
       FACT01    B'11'       Valid self-defining term
       FACT01    X'4'        Also
       FACT01    10
       FACT01    -2          Invalid argument
       FACT01    15          Should cause overflow
       End
```

Figure 36. Macro to Calculate Factorials Recursively: Examples

We leave to the reader the modifications needed to allow FACT01 to be called from other macros.



Generate Fibonacci Numbers Recursively 78

```
       Macro
&Lab   FIBONACI  &N
.* Fibonacci numbers defined by F(N) = F(N-1)+F(N-2), F(0) = F(1) = 0
       GBLA      &Ret            For returning values of inner calls
       MNote     0,'Evaluating FIBONACI(&N.), Level &SysNest.'
       AIF       (&N LT 0).Error  Negative values not allowed
       AIF       (&N GE 2).Calc   If &N > 1, use recursion
&Ret   SETA      1                Return F(0) or F(1)
       AGO       .Test            Return to caller
.Calc  ANOP                       Do computation
&K     SetA      &N-1             First value "K" = N-1
&L     SetA      &N-2             Second value "L" = N-2
       FIBONACI  &K               Evaluate F(K) = F(N-1)
&Temp  SetA      &Ret             Hold computed value
       FIBONACI  &L               Evaluate F(L) = F(N-2)
&Ret   SetA      &Ret+&Temp       Evaluate F(N) = F(K) + F(L)
.Test  AIF       (&SysNest GT 1).Cont
       MNote     0,'Fibonacci(&N.) = &Ret..'
&Lab   DC        F'&Ret'
.Cont  MExit                      Return to caller
.Error MNote     11,'Invalid Fibonacci argument &N..'
       MEnd
```

July 1993 High Level Assembler Tutorial Guide HLASM

Example 3: Fibonacci Numbers

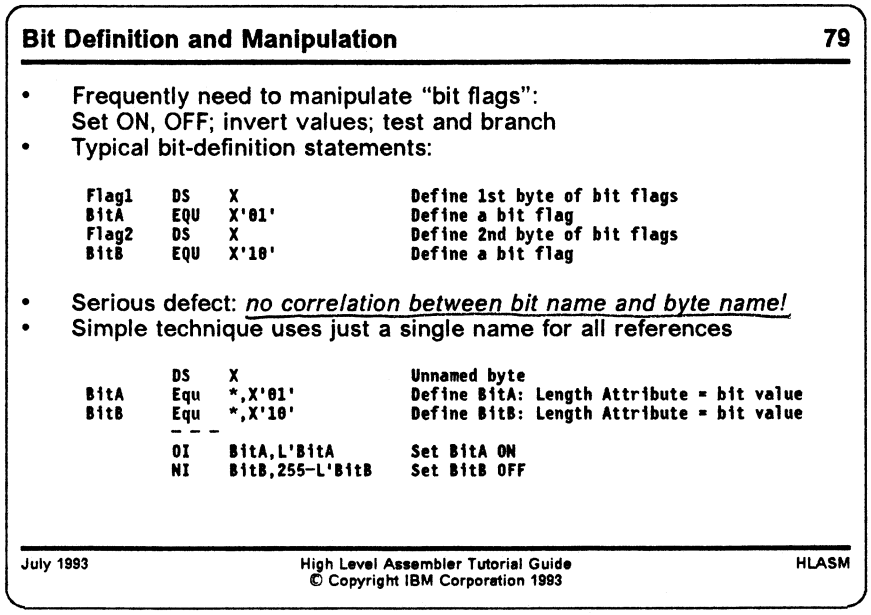
The Fibonacci numbers are defined by the recursion relations

$$F(N) = F(N-1) + F(N-2)$$
$$\text{with } F(0) = 1 \text{ and } F(1) = 1$$

Calculating them recursively is quite inefficient (though educational!) because many values are calculated more than once. The global arithmetic variable symbol &Ret is used to return values calculated at lower levels of the recursion.

```
FIB    Title  'Fibonacci Numbers by Macro Recursion'
*
       Macro
&Lab   FIBONACI  &N
       GBLA      &Ret              For returning values of inner calls
       LCLA      &Temp,&K,&L       Local variables
       MNote     0,'Evaluating FIBONACI(&N.), Level &SysNest.'
       AIF       (&N LT 0).Error   Negative values not allowed
       AIF       (&N GE 2).Calc    If &N > 1, use recursion
&Ret   SETA      1                 Return F(0) or F(1)
       AGO       .Test             Return to caller

.Calc  ANOP                        Do computation
&K     SetA      &N-1              First value "K" = N-1
&L     SetA      &N-2              Second value "L" = N-2
       FIBONACI  &K                Evaluate F(K) = F(N-1)
&Temp  SetA      &Ret              Hold computed value
       FIBONACI  &L                Evaluate F(L) = F(N-2)
&Ret   SetA      &Ret+&Temp        Evaluate F(N) = F(K) + F(L)
.Test  AIF       (&SysNest GT 1).Cont
       MNote     0,'Fibonacci(&N.) = &Ret..'
&Lab   DC        F'&Ret'
.Cont  MExit                       Return to caller
.Error MNote     11,'Invalid Fibonacci argument &N..'
       MEnd
*
       FIBONACI  4
       FIBONACI  5
       End
```

Figure 37. Macro to Calculate Fibonacci Numbers Recursively



Bit Handling

Applications frequently require status flags with binary values: ON or OFF, YES or NO, STARTED or NOT_STARTED, and the like. On a binary machine, such flags are represented by individual bits. However, few machines provide individually addressable bits; the bits are parts of larger data elements such as bytes or words. This means that special programming is needed to "address" and manipulate bits by name.

It is typical in the Assembler Language to define bits using statements like the following:

```
Flag1    DS    X               Define 1st byte of bit flags
BitA     EQU   X'01'           Define a bit flag
Flag2    DS    X               Define 2nd byte of bit flags
BitB     EQU   X'10'           Define a bit flag
```

and then doing bit operations like

```
         OI    Flag1,BitA      Set bit A "on"
```

There is implicitly a problem: the names of the bytes holding the flag bits, and the names given to the bits, are unrelated. This means that it is easy to make mistakes like the following:

```
         OI    Flag1,BitB      Set Bit B "on" ??
         TM    Flag2,BitA      Test Bit A ??
```

When there is no strict association between the byte and the bit it "contains", there is no way for the assembler (and often, the programmer) to detect such misuses.

One solution to the "association" problem is to use length attribute references to designate bit values. This allows us to "name" a bit, as follows:

```
.NoName1 DS    X
BitA     EQU   *,X'01'          Length Attribute = bit value
.NoName2 DS    X
BitB     EQU   *,X'10'          Length Attribute = bit value
```

That is, the bit name is the same as the name of the byte that contains it. Then, all bit references are made only with the bit "names":

```
         OI    BitA,L'BitA      Set Bit A "on"
         TM    BitB,L'BitB      Test Bit B
```

and (if one is careful) the bits will never be associated with the wrong byte! There is, of course, no guarantee that one might not write something like

```
         OI    BitA,L'BitB      ???
```

but a quick scan of the symbol cross-reference will show that there are unpaired references to the symbols BitA and BitB in this statement; correct references will occur in pairs.

Simple Bit-Handling Macros: Defining Bit Flags 80

```
         Macro ,                    Error checking omitted
         BitDef1
&L(I)    SetA  128,64,32,16,8,4,2,1 Define bit position values
&NN      SetA  N'&SysList           Number of bit names provided
&M       SetA  1                    Name counter
.NB      Aif   (&M gt &NN).Done     Check if names exhausted
&C       SetA  1                    Start new byte at leftmost bit
         DC    B'0'                 Allocate a bit-flag byte
.NewN    ANop  ,                    Get a new bit name
&B       SetC  '&SysList(&M)'       Get M-th name from argument list
&B       EQU   *,&L(&C)             Define bit via length attribute
&M       SetA  &M+1                 Step to next name
         Aif   (&M gt &NN).Done     Exit if names exhausted
&C       SetA  &C+1                 Count bits in a byte
         Aif   (&C le 8).NewN       Get new name if not done
         Ago   .NB                  Byte is filled, start a new byte
.Done    MEnd
*
         BitDef1  b1,b2,b3,b4,b5,b6,b7,b8    Eight bits in one byte
         BitDef1  c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v   Many bits+bytes
```


Bit-Handling Macros: Simple Forms

The simplest way to "encourage" correct matching of bit names and byte names is to make bit references with macros. Thus, we will illustrate a simple set of macros to do this.

First, suppose we want to "define" bit names. We will write a macro that accepts a list of bit names, and defines bit values in successive bytes, eight bits to a byte. The `BitDef1` macro in Figure 38 on page 96 takes the names in the argument list and allocates a single bit to each, eight bits to a byte. Each call to `BitDef1` starts a new byte.

```
        Macro
        BitDef1
&L(1)   SetA  128,64,32,16,8,4,2,1  Define bit position values
&NN     SetA  N'&SysList            Number of bit names provided
&M      SetA  1                     Name counter
.NB     Aif   (&M gt &NN).Done      Check if names exhausted
&C      SetA  1                     Start new byte at leftmost bit
        DC    B'0'                  Allocate a bit-flag byte
.NewN   ANop  ,                     Get a new bit name
&B      SetC  '&SysList(&M)'        Get M-th name from argument list
        Aif   ('&B' eq '').Null     Note null argument
&B      EQU   *,&L(&C)              Define bit via length attribute
&M      SetA  &M+1                  Step to next name
        Aif   (&M gt &NN).Done      Exit if names exhausted
&C      SetA  &C+1                  Count bits in a byte
        Aif   (&C le 8).NewN        Get new name if not done
        Ago   .NB                   Byte is filled, start a new byte
.Null   MNote 4,'BitDef1: Missing name at arglist position &M'
&M      SetA  &M+1                  Step to next name
        Aif   (&M le &NN).NewN      Go get new name if not done
.Done   MEnd
```

Figure 38. Bit-Handling Macros: Simple Bit Definition

Some examples of calls to the BitDef1 macro are shown in the following figure:

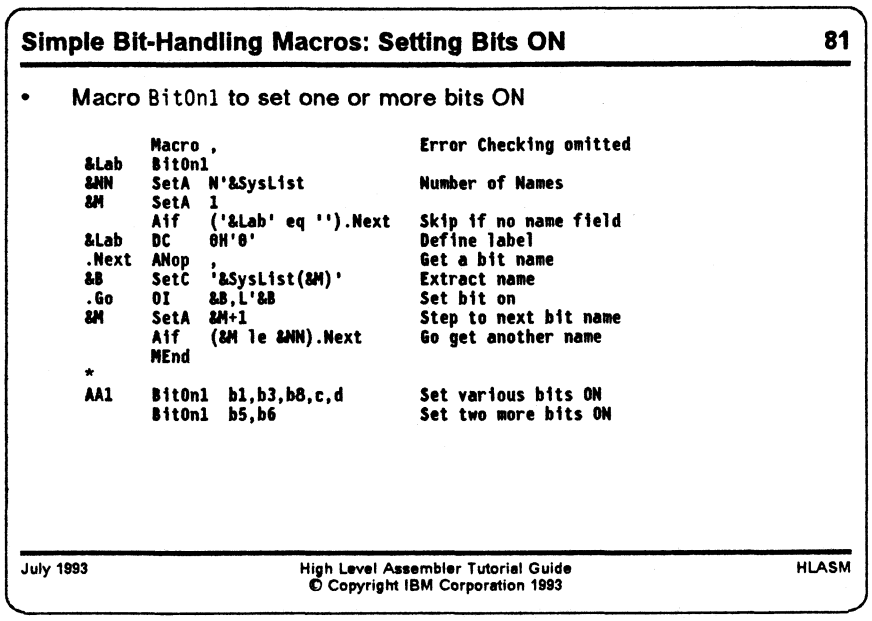
```
        BitDef1  b1,b2,b3,b4,b5,b6,b7,b8    Eight bits in one byte
        BitDef1  c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v   Many bits+bytes
        BitDef1
        BitDef1  F1,,F3             Omitted name
```

Figure 39. Bit-Handling Macros: Example of Bit Definitions

This simple macro has several limitations:

- Bits cannot be "grouped" so that related bits are certain to reside in the same byte, except by writing a statement with a new BitDef1 call.
- This means that we cannot plan to use the bit-manipulation instructions (which can handle up to 8 bits simultaneously) without manually arranging the assignments of bits and bytes.

We will explore some techniques that can be used to overcome these limitations in "Bit-Handling Macros: Advanced Forms" on page 102.



Simple Bit-Manipulation Macros

Having defined some bits with this BitDef1 macro, we will write some macros to manipulate them by setting them on and off, and by inverting ("flipping") their state. First, we will write a macro BitOn1 that will set a bit to an "on" state (i.e., to 1).

```
      Macro
&Lab  BitOn1
&NN   SetA  N'&SysList            Number of Names
&M    SetA  1
      Aif   (&NN gt 0).OK         Should not have empty name list
      MNote 4,'BitOn1: No bit names?'
      MExit
.OK   ANop  ,                     Names exist in the list
      Aif   ('&Lab' eq '').Next   Skip if no name field
&Lab  DC    0H'0'                 Define label
.Next ANop  ,                     Get a bit name
&B    SetC  '&SysList(&M)'        Extract name
      Aif   ('&B' ne '').Go       Check for missing argument
      MNote 4,'BitOn1: Missing argument at position &M'
      Ago   .Step                 Go look for more names
.Go   OI    &B,L'&B               Set bit on
.Step ANop  ,
&M    SetA  &M+1                  Step to next bit name
      Aif   (&M le &NN).Next      Go get another name
      MEnd
```

Figure 40. Bit-Handling Macros: Simple Bit Setting

In the following figure, we illustrate some calls to this macro to perform various bit settings; the generated statements are flagged with a "+" in the left margin:

```
 AA1    BitOn1  b1,b3,b8,c1,c2
+AA1    DC      0H'0'                 Define label
+       OI      b1,L'b1               Set bit on
+       OI      b3,L'b3               Set bit on
+       OI      b8,L'b8               Set bit on
+       OI      c1,L'c1               Set bit on
+       OI      c2,L'c2               Set bit on

        BitOn1  b1,b8
+       OI      b1,L'b1               Set bit on
+       OI      b8,L'b8               Set bit on
```

Figure 41. Bit-Handling Macros: Examples of Bit Setting

Each bit operation is performed by a separate instruction, even when two or more bits have been allocated in the same byte. We will see in "Bit-Handling Macros: Advanced Forms" on page 102 how we might remedy this defect.

Simple Bit-Handling Macros: Set OFF and Invert Bits 82

- Macros `BitOff1` and `BitInv` are defined just like `BitOn1`:

```
        Macro
&Lab    BitOff1
.*      - - -  etc., as for BitOn1
.Go     NI     &B,255-L'&B          Set bit off
.*      - - -  etc.
        MEnd
*
        Macro
&Lab    BitInv1
.*      - - -  etc., as for BitOn1
.Go     XI     &B,L'&B              Invert bit
.*      - - -  etc.
        MEnd
*
bb1     BitOff1   b1,b3,b8,c,d          Set various bits of
        BitOff1   b5,b6                 Set other bits off
cc1     BitInv1   b1,b3,b8,c,d          Invert various bits
        BitInv1   b5,b6                 Invert other bits
```


The BitOff1 macro is exactly like the BitOn1 macro, except that the generated statement to set the bit "off" (i.e., to 0) is changed:

```
        Macro
&Lab    BitOff1
.*      - - -  etc., as for BitOn1
.Go     NI     &B,255-L'&B          Set bit off
.*      - - -  etc., as for BitOn1
        MEnd
```

Figure 42. Bit-Handling Macros: Simple Bit Resetting

Some macro calls that illustrate the operation of the BitOff1 macro are shown in the following figure:

```
 bb1    BitOff1  b1,b3,b8,c1,c2
+bb1    DC    0H'0'                 Define label
+       NI    b1,255-L'b1           Set bit off
+       NI    b3,255-L'b3           Set bit off
+       NI    b8,255-L'b8           Set bit off
+       NI    c1,255-L'c1           Set bit off
+       NI    c2,255-L'c2           Set bit off

        BitOff1  b1,b8
+       NI    b1,255-L'b1           Set bit off
+       NI    b8,255-L'b8           Set bit off
```

Figure 43. Bit-Handling Macros: Examples of Bit Resetting

Similarly, the BitInv1 macro inverts the designated bits:

```
        Macro
&Lab    BitInv1
.*      - - -  etc., as for BitOn1
.Go     XI    &B,L'&B               Invert bit
.*      - - -  etc., as for BitOn1
        MEnd
```

Figure 44. Bit-Handling Macros: Simple Bit Inversion

Some calls to BitInv1 illustrate its operation:

```
 cc1    BitInv1  b1,b3,b8,c1,c2
+cc1    DC    0H'0'                 Define label
+       XI    b1,L'b1               Invert bit
+       XI    b3,L'b3               Invert bit
+       XI    b8,L'b8               Invert bit
+       XI    c1,L'c1               Invert bit
+       XI    c2,L'c2               Invert bit

        BitInv1  b1,b8
+       XI    b1,L'b1               Invert bit
+       XI    b8,L'b8               Invert bit
```

Figure 45. Bit-Handling Macros: Examples of Bit Inversion

Simple Bit-Handling Macros: Branch on Bit Values 83

- Simple bit-testing macros: branch to label L if bit B is on/off

```
        Macro
&Lab    BitBOn1 &B,&T              Bitname and branch label
&Lab    TM    &B,L'&B              Test specified bit
        BO    &T                   Branch if ON
        MEnd
*
        Macro
&Lab    BitBOff1 &B,&T             Bitname and branch label
&Lab    TM    &B,L'&B              Test specified bit
        BNO   &T                   Branch if OFF
        MEnd
*
        BitBOn1   b2,bb1           Branch to bb1 if b2 is on
        BitBOff1  b3,dd1           Branch to dd1 if b3 is off
```

July 1993 High Level Assembler Tutorial Guide HLASM

To complete our set of simple bit-handling macros, suppose we need macros to test the setting of a bit, and to branch to a designated label if the bit is on or off. We can write two macros named BitBOn1 and BitBOff1 to do this; each has two arguments, a bit name and a label name.

```
        Macro
&Lab    BitBOn1 &B,&T              Bitname and branch label
        Aif   (N'&SysList eq 2).OK  Should have exactly 2 arguments
        MNote 4,'BitBOn1: Incorrect argument list?'
        MExit
.OK     Aif   ('&B' eq '' or '&T' eq '').Bad
&Lab    TM    &B,L'&B              Test specified bit
        BO    &T                   Branch if ON
        MExit
.Bad    MNote 8,'BitBon1: Bit Name or Target Name missing'
        MEnd
```

Figure 46. Bit-Handling Macros: Branch if Bit is On

Some examples of calls to the BitBOn1 macro are shown in the following figure:

```
 dd1    BitBOn1  b1,aa1
+dd1    TM    b1,L'b1              Test specified bit
+       BO    aa1                  Branch if ON

        BitBOn1  b2,bb1
+       TM    b2,L'b2              Test specified bit
+       BO    bb1                  Branch if ON
```

Figure 47. Bit-Handling Macros: Examples of "Branch if Bit On"

A similar macro can be written to branch to a specified label if a bit is off:

```
        Macro
&Lab    BitBOff1 &B,&T              Bitname and branch label
.*      - - -    etc., as for BitBOn1 macro
&Lab    TM    &B,L'&B               Test specified bit
        BNO   &T                    Branch if OFF
.*      - - -    etc., as for BitBOn1 macro
        MEnd
```

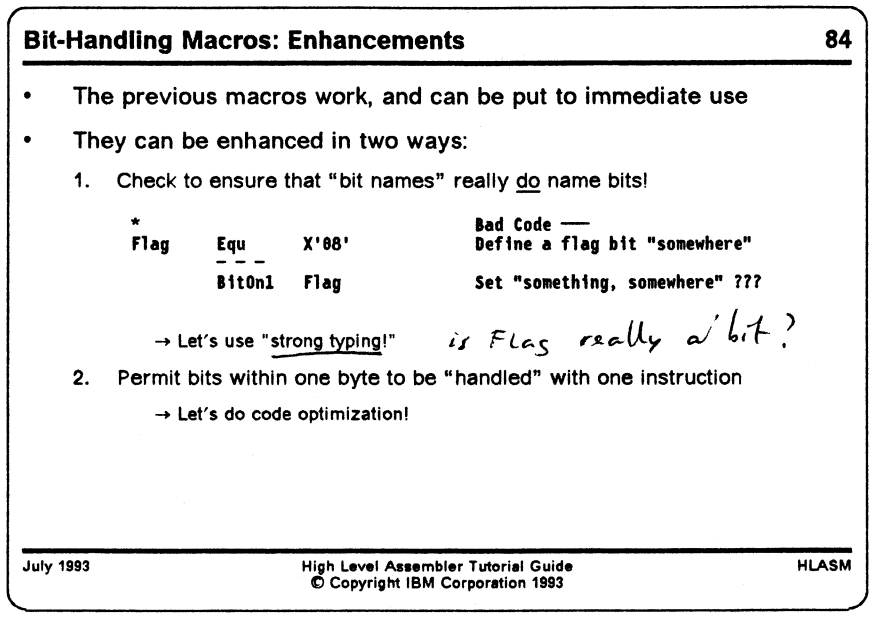
Figure 48. Bit-Handling Macros: Branch if Bit is Off

Calls to the BitBOf1 macro might appear as follows:

```
 ee1    BitBOff1 b1,dd1             Branch to dd1 if b1 is off
+ee1    TM    b1,L'b1               Test specified bit
+       BNO   dd1                   Branch if OFF

        BitBOff1 b2,dd1             Branch to dd1 if b2 is off
+       TM    b2,L'b2               Test specified bit
+       BNO   dd1                   Branch if OFF
```

Figure 49. Bit-Handling Macros: Examples of "Branch if Bit Off"

This completes our first set of bit-handling macros. It is evident that a fairly helpful set of capabilities can be written with a very small effort, and be put into immediate use.



Bit-Handling Macros: Advanced Forms

There are two problems with the preceding "simple set" of bit-handling macros:

1. it is common to want to operate on more than one bit within a given byte at the same time. For example, suppose two bits are defined within the same byte:

```
         DS    X
BitJ     EQU   *,X'40'
BitK     EQU   *,X'20'
```

   We would prefer to set both bits "on" with a single OI instruction. Two possibilities are evident:

```
         OI    BitJ,L'BitJ+L'BitK
         OI    BitK,L'BitJ+L'BitK
```

   Unfortunately, both of these schemes do not satisfy our intent to name just the bits we wish to manipulate, and not the bytes in which they are defined. Thus, we need some degree of "optimization" in our bit-handling macros.

2. There is no checking of the "bit names" presented as arguments in the bit-manipulation macros to verify that they were indeed *declared* as bits in a "bit definition" macro. For example, one might have written (through some oversight, probably not as drastic as this!)

```
Flag     Equ      X'08'          Define a flag bit
         - - -
         BitOn1   Flag           Set "something, somewhere" on ???
```

   and the result would not have been what was expected or desired. Thus, we need some degree of "strong typing" and "type checking" in our bit-handling macros.

We will start with a `BitDef` macro that declares bit flags, and keeps track of which ones have been declared. We will add an extra feature: if a group of bits should be kept in a single

byte, their names may be specified as a parenthesized operand sublist, and the macro will ensure that (if at most eight are specified) they will fit in a byte. Thus, in

```
BitDef  a,b,c,(d,e,f,g,h,i),j,k
```

the bits named a,b,c will be allocated in one byte, and bits d,e,f,g,h,i will be allocated in a new byte because there is not enough room left for all of them in the byte containing a,b,c. However, bits j,k will share the same byte as d,e,f,g,h,i because there are two bits remaining for them.

One of the decisions influencing the design of these macros is that we wish to optimize execution performance more than we wish to minimize storage utilization; because bits are small, wasting a few shouldn't be a major concern.

Bit-Handling Macros: Bit Definition 85

```
        Macro ,                     Some error checks omitted
        BitDef
        GblA  &BitDef_ByteNo        Used to count defined bytes
&L(1)   SetA  128,64,32,16,8,4,2,1  Define bit position values
&NN     SetA  N'&SysList            Number of bit names provided
&M      SetA  1                     Name counter
.NB     AIf   (&M gt &NN).Done      Check if names exhausted
&C      SetA  1                     Start new byte at leftmost bit
        DC    B'0'                  Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.NewN   ANop  ,                     Get a new bit name
&B      SetC  '&SysList(&M)'        Get M-th name from argument list
        AIf   ('&B'(1,1) ne '(').NoL   Branch if not a sublist
&NS     SetA  N'&SysList(&M)        Number of sublist elements
&CS     SetA  1                     Initialize count of sublist items
        AIf   (&C+&NS le 9).SubT    Skip if room left in current byte
&C      SetA  1                     Start a new byte
        DC    B'0'                  Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.*      - - -     (continued)
```


Bit name is stored as a GblA.

Bit-Handling Macros: Bit Definition ... 86

```
.SubT   ANop  ,                     Generate sublist equates
&B      SetC  '&SysList(&M,&CS)'    Extract sublist element
        GblA  &(BitDef_&B._ByteNo)  Declare byte number for this bit
&B      EQU   *,&L(&C)              Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
&CS     SetA  &CS+1                 Step to next sublist item
        AIf   (&CS gt &NS).NewA     Skip if end of sublist
&C      SetA  &C+1                  Count bits in a byte
        AGo   .Subt                 And go do more list elements
.NoL    ANop  ,                     Not a list
        GblA  &(BitDef_&B._ByteNo)  Declare byte number for this bit
&B      EQU   *,&L(&C)              Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
.NewA   ANop  ,                     Ready for next argument
&M      SetA  &M+1                  Step to next name
        AIf   (&M gt &NN).Done      Exit if names exhausted
&C      SetA  &C+1                  Count bits in a byte
        AIf   (&C le 8).NewN        Get new name if not done
        AGo   .NB                   Bit filled, start a new byte
.Done   MEnd
```

Declaring Bit Names

In the BitDef macro illustrated in Figure 50, several techniques are used. The global arithmetic variable &BitDef_ByteNo is used to keep track of a "byte number" in which the various bits are allocated; each time a new byte is allocated, this variable is incremented by 1. The first SETA statement initializes the local arithmetic array variables &L(1) through &L(8) to values corresponding to the binary weights of the bits in a byte, in left-to-right order.

After each bit name has been extracted from the argument list, a global arithmetic variable &(BitDef_&B._ByteNo) is constructed (and declared) using the supplied bit name as the value of &B, and is assigned the value of the byte number to which that bit will be assigned. This has two effects:

1. a unique global variable symbol is generated for every bit name;
2. the value of that symbol identifies the byte it "belongs to" (remember that the bytes have no names themselves; references in actual instructions will be made using bit names and length attribute references).

An additional benefit of this technique is that later references to a bit can be checked against this global variable: if its value is zero (meaning it was declared but not initialized) we will know that the bit was not declared, and therefore not allocated to a byte in storage.

The other new feature introduced in this macro definition is the ability to handle sublists of bit names that are to be allocated within the same byte.

```
        Macro
        BitDef
        GblA  &BitDef_ByteNo        Used to count defined bytes
&L(1)   SetA  128,64,32,16,8,4,2,1  Define bit position values
&NN     SetA  N'&SysList            Number of bit names provided
&M      SetA  1                     Name counter
.NB     Aif   (&M gt &NN).Done      Check if names exhausted
&C      SetA  1                     Start new byte at leftmost bit
        DC    B'0'                  Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.NewN   ANop  ,                     Get a new bit name
&B      SetC  '&SysList(&M)'        Get M-th name from argument list
        Aif   ('&B' eq '').Null     Note null argument
        Aif   ('&B'(1,1) ne '(').NoL   Branch if not a sublist
&NS     SetA  N'&SysList(&M)        Number of sublist elements
        Aif   (&NS gt 8).ErrS       Error if more than 8
```

Figure 50 (Part 1 of 2). Bit-Handling Macros: Define Bit Names

```
&CS     SetA  1                     Initialize count of sublist items
        Aif   (&C+&NS le 9).SubT    Skip if room left in current byte
&C      SetA  1                     Start a new byte
        DC    B'0'                  Define a bit-flag byte
&BitDef_ByteNo SetA &BitDef_ByteNo+1  Increment byte number
.SubT   ANop  ,                     Generate sublist equates
&B      SetC  '&SysList(&M,&CS)'    Extract sublist element
        GblA  &(BitDef_&B._ByteNo)  Declare byte number for this bit
        Aif   (&(BitDef_&B._ByteNo) gt 0).DupDef  Branch if declared
&B      EQU   *,&L(&C)              Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
&CS     SetA  &CS+1                 Step to next sublist item
        Aif   (&CS gt &NS).NewA     Skip if end of sublist
&C      SetA  &C+1                  Count bits in a byte
        Ago   .Subt                 And go do more list elements
.NoL    ANop  ,                     Not a list
        GblA  &(BitDef_&B._ByteNo)  Declare byte number for this bit
        Aif   (&(BitDef_&B._ByteNo) gt 0).DupDef  Branch if declared
&B      EQU   *,&L(&C)              Define bit via length attribute
&(BitDef_&B._ByteNo) SetA &BitDef_ByteNo  Byte no. for this bit
.NewA   ANop  ,                     Ready for next argument
&M      SetA  &M+1                  Step to next name
        Aif   (&M gt &NN).Done      Exit if names exhausted
&C      SetA  &C+1                  Count bits in a byte
        Aif   (&C le 8).NewN        Get new name if not done
        Ago   .NB                   Bit filled, start a new byte
.DupDef MNote 8,'BitDef: Bit name ''&B'' was previously declared.'
        MExit
.ErrS   MNote 8,'BitDef: Sublist Group has more than 8 members'
        MExit
.Null   MNote 8,'BitDef: Missing name at argument &M'
.Done   MEnd
```

Figure 50 (Part 2 of 2). Bit-Handling Macros: Define Bit Names

Some examples of calls to this BitDef macro are shown in the following figure; the generated instructions are displayed (with "+" characters in the left margin) for two of the calls:

```
 a4     BitDef   d1,d2,d3,(d4,d5,d6,d7,d8,d9),d10    d4 starts new byte
+        DC    B'0'                  Define a bit-flag byte
+d1      EQU   *,128                 Define bit via length attribute
+d2      EQU   *,64                  Define bit via length attribute
+d3      EQU   *,32                  Define bit via length attribute
+        DC    B'0'                  Define a bit-flag byte
+d4      EQU   *,128                 Define bit via length attribute
+d5      EQU   *,64                  Define bit via length attribute
+d6      EQU   *,32                  Define bit via length attribute
+d7      EQU   *,16                  Define bit via length attribute
+d8      EQU   *,8                   Define bit via length attribute
+d9      EQU   *,4                   Define bit via length attribute
+d10     EQU   *,2                   Define bit via length attribute

 a5     BitDef   e1,e2,e3,e4,e5,e6,e7,(e8,e9)        e8 starts new byte

 a6     BitDef   g1,(g2,g3,g4,g5,g6,g7,g8,g9)        g2 starts new byte

 a7     BitDef   (h2,h3,h4,h5,h6,h7,h8,h9,h10),h11   error, 9 in a byte

 a9     BitDef   (k1,k2,k3,k4),(k5,k6,k7,k8),k9,k10  two sublists
+        DC    B'0'                  Define a bit-flag byte
+k1      EQU   *,128                 Define bit via length attribute
+k2      EQU   *,64                  Define bit via length attribute
+k3      EQU   *,32                  Define bit via length attribute
+k4      EQU   *,16                  Define bit via length attribute
+k5      EQU   *,8                   Define bit via length attribute
+k6      EQU   *,4                   Define bit via length attribute
+k7      EQU   *,2                   Define bit via length attribute
+k8      EQU   *,1                   Define bit via length attribute
+        DC    B'0'                  Define a bit-flag byte
+k9      EQU   *,128                 Define bit via length attribute
+k10     EQU   *,64                  Define bit via length attribute

 a10    BitDef   l1,(l2,l3,l4),(l5,l6,l7,l8),l9,l10  two sublists

 a11    BitDef   m1,(m2,m3,m4),(m5,m6,m7,m8,m9),m10  two sublists
```

Figure 51. Bit-Handling Macros: Examples of Defining Bit Names

We will now see how we can utilize the information created by this BitDef macro to generate efficient instruction sequences to manipulate them.

Bit-Handling Macros: Set Bits ON 87

- Macro BitOn optimizes generated instructions (most error checks omitted)

```
         Macro
&Lab     BitOn
&L       SetC  '&Lab'                  Save label
&NBN     SetA  0                       No. of distinct Byte Nos.
&M       SetA  0                       Name counter
&NN      SetA  N'&SysList              Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2       Check if all names scanned
&M       SetA  &M+1                    Step to next name
&B       SetC  '&SysList(&M)'          Pick off a name
         GblA  &(BitDef_&B._ByteNo)    Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                       Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN      Not in list, a new Byte No
&K       SetA  &K+1                    Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
.*       - - -   continued
```


Bit-Handling Macros: Set Bits ON ... 88

```
&J       SetA  1                       Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK   Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                    Search next name in this byte
         Ago   .CkDup                  Check further for duplicates
.DupNm   MNote 8,'BitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                       No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1              Have matching BN, count up by 1
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                   Go get next name
.NewBN   ANop  ,                       New Byte No
&NBN     SetA  &NBN+1                  Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)    Save new Byte No
&IBN(&NBN) SetA 1                      Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'    Save 1st Bit Name, this byte
         Ago   .NMLp                   Go get next name
.*       - - -   continued
```

```
.Pass2   ANop  ,                        Pass 2: scan Byte No list
&M       SetA  0                        Byte No counter
.BLp     Aif   (&M ge &NBN).Done        Check if all Byte Nos done
&M       SetA  &M+1                     Increment outer-loop counter
&X       SetA  &BN(&M)                  Get M-th Byte No
&K       SetA  1                        Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' Operand
.OpLp    Aif   (&K ge &IBN(&M)).GenOI   Operand loop, check for done
&K       SetA  &K+1                     Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add next bit to operand
         Ago   .OpLp                    Loop (inner) for next operand
.GenOI   ANop  ,                        Generate instruction for Byte No
&L       OI    &Op                      Turn bits ON
&L       SetC  ''                       Nullify label string
         Ago   .BLp                     Loop (outer) for next Byte No
.UnDef   MNote 8,'BitOn: Name ''&B'' not defined by BitDef'
         MExit
.Done    MEnd
```


Using Declared Bit Names in a BitOn Macro

The BitOn macro accepts a list of bit names, and generates the minimum number of instructions needed to set them on (i.e., to 1), as illustrated in Figure 52 on page 109. The macro makes two "passes" over the supplied bit names:

- In the first pass, the bit names are read, and the global arithmetic variable &(BitDef_&B._&ByteNo) (where the value of &B is the bit name) is constructed and declared, and its value is checked. If the value is zero, we know that the name was not declared in a call to a BitDef macro (which would have assigned a non-zero byte number value to the variable).
- If the bit name was defined, the value of the constructed name is the byte number of the byte to which the bit was assigned. The array &BN() is searched to see if other bits with the same byte number have been supplied as arguments to this BitOn macro; if not, a new entry is made in the &BN() array.
- A second array &IBN() (paralleling the &BN() array) is used to count the number of **I**nstances of the **B**yte **N**umber that have occurred thus far.
- Finally, the bit name is saved in a created local character variable symbol &(BitDef_Nm_&bn._&in), where &bn is the byte number for this bit name, and &in is the "instance number" of this bit within this byte. (By checking the current bit name from the argument list against these names, the macro can also determine that a bit name has been "duplicated" in the argument list.)

Once all the names in the argument list have been handled, the macro uses the information in the two arrays and the created local character variable symbols:

- In the second pass, one instruction will be generated for each distinct byte number that was entered in the &BN() array during the first pass, using two nested loops; the outer loop is executed once per byte number.
- The inner loop is executed as many times as there are instances of names belonging to the current byte number (as determined from the elements of the &IBN() array), and constructs the operand field in the local character variable &Op, using the created local character variable symbols to retrieve the names of the bits.

- At the end of the inner loop, the OI instruction is generated using the created operand field string in &Op, and then the outer loop is repeated until the instructions for all the bytes containing named bit have been generated.

```
         Macro
&Lab     BitOn
&L       SetC  '&Lab'                    Save label
&NBN     SetA  0                         No. of distinct Byte Nos.
&M       SetA  0                         Name counter
&NN      SetA  N'&SysList                Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2         Check if all names scanned
&M       SetA  &M+1                      Step to next name
&B       SetC  '&SysList(&M)'            Pick off a name
         GblA  &(BitDef_&B._ByteNo)      Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                         Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN        Not in list, a new Byte No
&K       SetA  &K+1                      Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
&J       SetA  1                         Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK     Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                      Search next name in this byte
         Ago   .CkDup                    Check further for duplicates
.DupNm   MNote 8,'BitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                         No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1                Have matching BN, count up by 1
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                     Go get next name
.NewBN   ANop  ,                         New Byte No
&NBN     SetA  &NBN+1                    Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)      Save new Byte No
&IBN(&NBN) SetA 1                        Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'      Save 1st Bit Name, this byte
         Ago   .NMLp                     Go get next name
.Pass2   ANop  ,                         Pass 2: scan Byte No list
&M       SetA  0                         Byte No counter
.BLp     Aif   (&M ge &NBN).Done         Check if all Byte Nos done
&M       SetA  &M+1                      Increment outer-loop counter
&X       SetA  &BN(&M)                   Get M-th Byte No
&K       SetA  1                         Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' Operand
.OpLp    Aif   (&K ge &IBN(&M)).GenOI    Operand loop, check for done
&K       SetA  &K+1                      Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add next bit to operand
         Ago   .OpLp                     Loop (inner) for next operand
```

Figure 52 (Part 1 of 2). Bit-Handling Macros: Set Bits ON

```
.GenOI   ANop  ,                      Generate instruction for Byte No
&L       OI    &Op                     Turn bits ON
&L       SetC  ''                      Nullify label string
         Ago   .BLp                    Loop (outer) for next Byte No
.UnDef   MNote 8,'BitOn: Name ''&B'' not defined by BitDef'
         MExit
.Done    MEnd
```

Figure 52 (Part 2 of 2). Bit-Handling Macros: Set Bits ON

Some examples of calls to the BitOn macro are illustrated in the figure below. In each case, the minimum number of instructions necessary to set the specified bits will be generated. The instructions generated by the macro are shown for two of the calls.

```
 ABCD    BitOn b1,b2
+        OI    b1,L'b1+L'b2            Turn bits ON

 Fbc     BitOn b1,c2,b1         Duplicate bit name 'b1'

 Fbd     BitOn jj               Undeclared bit name 'jj'

     BitOn c1,c2,c3,c4,c5,c6,c7,c8,c9,c10,c11,c12,c13,c14,c15,c16,c17

 Fbg     BitOn b1,c1,d1,e1,b2,c2,d2,c3,b3,m2,c4,c5,m5,d6,c6,d7,b4,c7
+Fbg     OI    b1,L'b1+L'b2+L'b3+L'b4  Turn bits ON
+        OI    c1,L'c1+L'c2+L'c3+L'c4+L'c5+L'c6+L'c7   Turn bits ON
+        OI    d1,L'd1+L'd2            Turn bits ON
+        OI    e1,L'e1                 Turn bits ON
+        OI    m2,L'm2                 Turn bits ON
+        OI    m5,L'm5                 Turn bits ON
+        OI    d6,L'd6+L'd7            Turn bits ON

 DupB1   BitOn b1,c2,c3,c4,c5,c6,c7,c8,c9,c10,b1  Duplicated name 'b1'
```

Figure 53. Bit-Handling Macros: Examples of Setting Bits ON

Extending this macro to create BitOff and BitInv macros is straightforward (we can use the schemes illustrated in Figure 42 on page 98 and Figure 44 on page 99), and is left as the traditional "exercise for the reader".

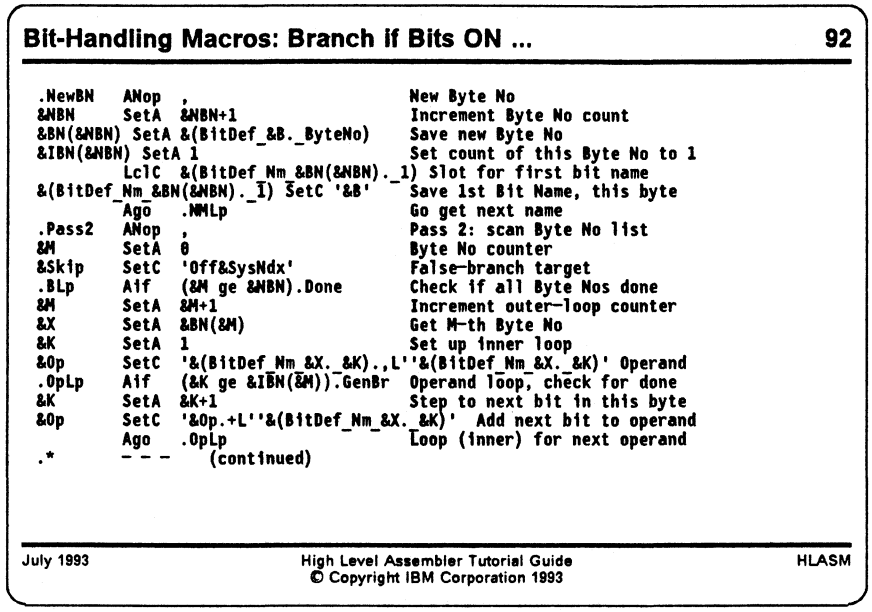
Bit-Handling Macros: Branch if Bits ON 90

- Macro BBitOn optimizes generated instructions (most error checks omitted)
- Two "passes" over bit name list:
  1. Scan and names, determine byte numbers
  2. If multiple bytes, generate "skip" tests/branches and label

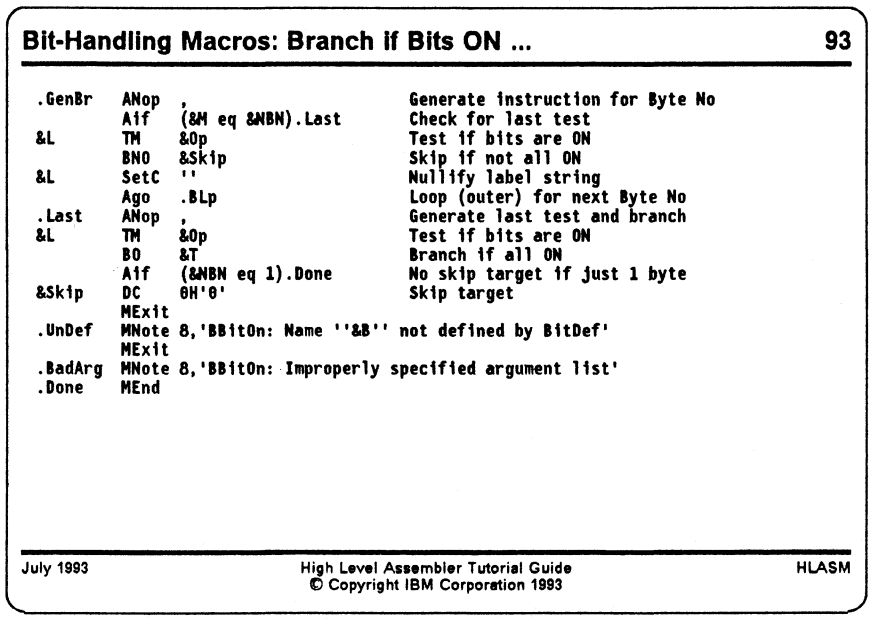
```
         Macro
&Lab     BBitOn &NL,&T                  Bit Name List, Branch Target
         Aif   (N'&SysList ne 2 or '&NL' eq '' or '&T' eq '').BadArg
&L       SetC  '&Lab'                   Save label
&NBN     SetA  0                        No. of distinct Byte Nos.
&M       SetA  0                        Name counter
&NN      SetA  N'&NL                    Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2        Check if all names scanned
.*       - - -    (continued)
```


Bit-Handling Macros: Branch if Bits ON ... 91

```
&M       SetA  &M+1                     Step to next name
&B       SetC  '&NL(&M)'                Pick off a name
         GblA  &(BitDef_&B._ByteNo)     Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                        Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN       Not in list, a new Byte No
&K       SetA  &K+1                     Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
&J       SetA  1                        Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK    Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                     Search next name in this byte
         Ago   .CkDup                   Check further for duplicates
.DupNm   MNote 8,'BBitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                        No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1               Have matching BN, count up by 1
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                    Go get next name
.*       - - -    (continued)
```



```
Bit-Handling Macros: Branch if Bits ON ...                    92

.NewBN   ANop  ,                        New Byte No
&NBN     SetA  &NBN+1                   Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)     Save new Byte No
&IBN(&NBN) SetA 1                       Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'     Save 1st Bit Name, this byte
         Ago   .NMLp                    Go get next name
.Pass2   ANop  ,                        Pass 2: scan Byte No list
&M       SetA  0                        Byte No counter
&Skip    SetC  'Off&SysNdx'             False-branch target
.BLp     Aif   (&M ge &NBN).Done        Check if all Byte Nos done
&M       SetA  &M+1                     Increment outer-loop counter
&X       SetA  &BN(&M)                  Get M-th Byte No
&K       SetA  1                        Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' Operand
.OpLp    Aif   (&K ge &IBN(&M)).GenBr   Operand loop, check for done
&K       SetA  &K+1                     Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add next bit to operand
         Ago   .OpLp                    Loop (inner) for next operand
.*       - - -   (continued)

July 1993          High Level Assembler Tutorial Guide          HLASM
                   © Copyright IBM Corporation 1993
```



```
Bit-Handling Macros: Branch if Bits ON ...                    93

.GenBr   ANop  ,                        Generate instruction for Byte No
         Aif   (&M eq &NBN).Last        Check for last test
&L       TM    &Op                      Test if bits are ON
         BNO   &Skip                    Skip if not all ON
&L       SetC  ''                       Nullify label string
         Ago   .BLp                     Loop (outer) for next Byte No
.Last    ANop  ,                        Generate last test and branch
&L       TM    &Op                      Test if bits are ON
         BO    &T                       Branch if all ON
         Aif   (&NBN eq 1).Done         No skip target if just 1 byte
&Skip    DC    0H'0'                    Skip target
         MExit
.UnDef   MNote 8,'BBitOn: Name ''&B'' not defined by BitDef'
         MExit
.BadArg  MNote 8,'BBitOn: Improperly specified argument list'
.Done    MEnd

July 1993          High Level Assembler Tutorial Guide          HLASM
                   © Copyright IBM Corporation 1993
```

Using Declared Bit Names in a BBitOn Macro

The `BBitOn` macro is intended to branch to a specified label if all the specified bit names are "on", and should use the minimum number of instructions; the calling syntax is the following:

```
BBitOn  (Bit_Name_List),Branch_Target
```

and we will accept a single non-parenthesized bit name for the first argument.

This macro will require a slightly different approach from the one used in the `BitOn` macro: if any of the bits have been allocated in different bytes, we must invert the "sense" of all generated branch instructions except the last. To see why this is so, suppose we wish to branch to XX if both BitA and BitB are "true", and the two bits have been allocated in the *same* byte:

```
         DC    B'0'
BitA     Equ   *,X'01'              Allocate BitA
BitB     Equ   *,X'20'              Allocate BitB
*
         TM    BitA,L'BitA+L'BitB   Test BitA and BitB
         BO    XX                   Branch if both are ON
```

and we see that only a single test instruction is needed. Now, suppose the two bits have been allocated to *distinct* bytes:

```
         DC    B'0'
BitA     Equ   *,X'01'              Allocate BitA
         DC    B'0'
BitB     Equ   *,X'20'              Allocate BitB
```

Then, to branch if both are true, we must use two test instructions:

```
         TM    BitA,L'BitA          Check BitA
         BNO   Not_True             Skip-Branch if not 1
         TM    BitB,L'BitB          BitA is 1; check BitB
         BO    XX                   Branch to XX if both are 1
Not_True DC    0H'0'                Label holder for "skip" target
```

The implementation of the `BBitOn` macro uses a scheme similar to that in the `BitOn` macro: the list of bit names in the first argument will be extracted, and the same list of variables will be constructed. The second "pass" will need some modifications:

- If more than one pair of test and branch instructions will be generated, a "not true" label must be used for all branches except the last, and the label must be defined following the final test and branch.
- The sense of all branches except the last must be "inverted" so that a branch will be taken to the target label only if all the bits tested have been determined to be "true".

```
         Macro
&Lab     BBitOn &NL,&T                 Bit Name List, Branch Target
         Aif   (N'&SysList ne 2 or '&NL' eq '' or '&T' eq '').BadArg
&L       SetC  '&Lab'                  Save label
&NBN     SetA  0                       No. of distinct Byte Nos.
&M       SetA  0                       Name counter
&NN      SetA  N'&NL                   Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2       Check if all names scanned
&M       SetA  &M+1                    Step to next name
&B       SetC  '&NL(&M)'               Pick off a name
         GblA  &(BitDef_&B._ByteNo)    Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                       Loop through known Byte Nos
.BNLp    Aif   (&K ge &NBN).NewBN      Not in list, a new Byte No
&K       SetA  &K+1                    Search next known Byte No
         Aif   (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
&J       SetA  1                       Check if name already specified
.CkDup   Aif   (&J gt &IBN(&K)).NmOK   Branch if name is unique
         Aif   ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)').DupNm  Duplicated
&J       SetA  &J+1                    Search next name in this byte
         Ago   .CkDup                  Check further for duplicates
.DupNm   MNote 8,'BBitOn: Name ''&B'' duplicated in operand list'
         MExit
.NmOK    ANop  ,                       No match, enter name in list
&IBN(&K) SetA  &IBN(&K)+1              Have matching BN, count up by 1
         LclC  &(BitDef_Nm_&BN(&K)._&IBN(&K))    Slot for bit name
&(BitDef_Nm_&BN(&K)._&IBN(&K)) SetC '&B'  Save K'th Bit Name, this byte
         Ago   .NMLp                   Go get next name
.NewBN   ANop  ,                       New Byte No
&NBN     SetA  &NBN+1                  Increment Byte No count
&BN(&NBN) SetA &(BitDef_&B._ByteNo)    Save new Byte No
&IBN(&NBN) SetA 1                      Set count of this Byte No to 1
         LclC  &(BitDef_Nm_&BN(&NBN)._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN)._1) SetC '&B'    Save 1st Bit Name, this byte
         Ago   .NMLp                   Go get next name
```

Figure 54 (Part 1 of 2). Bit-Handling Macros: Macro to Branch if Bits are ON

```
.Pass2   ANop  ,                        Pass 2: scan Byte No list
&M       SetA  0                        Byte No counter
&Skip    SetC  'Off&SysNdx'             False-branch target
.BLp     Aif   (&M ge &NBN).Done        Check if all Byte Nos done
&M       SetA  &M+1                     Increment outer-loop counter
&X       SetA  &BN(&M)                  Get M-th Byte No
&K       SetA  1                        Set up inner loop
&Op      SetC  '&(BitDef_Nm_&X._&K).,L''&(BitDef_Nm_&X._&K)' Operand
.OpLp    Aif   (&K ge &IBN(&M)).GenBr   Operand loop, check for done
&K       SetA  &K+1                     Step to next bit in this byte
&Op      SetC  '&Op.+L''&(BitDef_Nm_&X._&K)'  Add next bit to operand
         Ago   .OpLp                    Loop (inner) for next operand
.GenBr   ANop  ,                        Generate instruction for Byte No
         Aif   (&M eq &NBN).Last        Check for last test
&L       TM    &Op                      Test if bits are ON
         BNO   &Skip                    Skip if not all ON
&L       SetC  ''                       Nullify label string
         Ago   .BLp                     Loop (outer) for next Byte No
.Last    ANop  ,                        Generate last test and branch
&L       TM    &Op                      Test if bits are ON
         BO    &T                       Branch if all ON
         Aif   (&NBN eq 1).Done         No skip target if just 1 byte
&Skip    DC    0H'0'                    Skip target
         MExit
.UnDef   MNote 8,'BBitOn: Name ''&B'' not defined by BitDef'
         MExit
.BadArg  MNote 8,'BBitOn: Improperly specified argument list'
.Done    MEnd
```

Figure 54 (Part 2 of 2). Bit-Handling Macros: Macro to Branch if Bits are ON

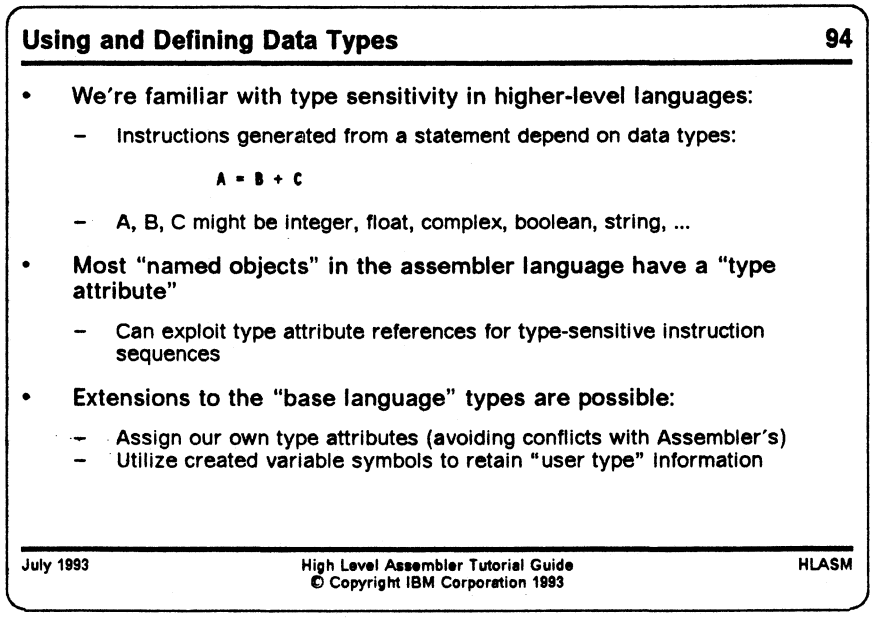
Some examples of calls to the BBitOn macro are shown in the following figure; the generated instructions are indicated by "+" characters in the left margin:

```
 TB4      BBitOn   b1,TB5
+TB4      TM    b1,L'b1                 Test if bits are ON
+         BO    TB5                     Branch if all ON

          BBitOn  (c5,c4,c3,c2),tb7
+         TM    c5,L'c5+L'c4+L'c3+L'c2  Test if bits are ON
+         BO    tb7                     Branch if all ON

 TB6      BBitOn  (b1,c2,b2,c3,b3,b4,c4,b5,c5),tb4
+TB6      TM    b1,L'b1+L'b2+L'b3+L'b4+L'b5    Test if bits are ON
+         BNO   Off0051                 Skip if not all ON
+         TM    c2,L'c2+L'c3+L'c4+L'c5  Test if bits are ON
+         BO    tb4                     Branch if all ON
+Off0051  DC    0H'0'                   Skip target

 TB7      BBitOn  (b1,b2,b3,b4,b5,b6,b7),tb7

          BBitOn  (b1,c2,b2,c3,d4,e2),tb7
+         TM    b1,L'b1+L'b2            Test if bits are ON
+         BNO   Off0054                 Skip if not all ON
+         TM    c2,L'c2+L'c3            Test if bits are ON
+         BNO   Off0054                 Skip if not all ON
+         TM    d4,L'd4                 Test if bits are ON
+         BNO   Off0054                 Skip if not all ON
+         TM    e2,L'e2                 Test if bits are ON
+         BO    tb7                     Branch if all ON
+Off0054  DC    0H'0'                   Skip target
```

Figure 55. Bit-Handling Macros: Examples of Calls to BBitON Macro

The extension of the `BBitOn` macro to a similar `BBitOff` macro is simple, and is also left as an exercise.

In summary, this final set of macros can be used to define, manipulate, and test bit flags with reliability and efficiency.



Using and Defining Data Types

One of the most useful features of the macro language is that it allows you to write macros whose behavior depends on the "types" of its arguments. A single macro definition can generate different instruction sequences, depending on what it can determine about its arguments. This behavior is common in most higher-level languages; for example, the statement

```
A = B + C
```

may generate very different instructions depending on whether the variables A, B, and C have been declared to be integer, floating, complex, boolean, or character string (or mixtures of those, as in PL/I). We will see that macros offer the same flexibility and power.

```
        Macro ,                     Increment &V by amount &A (default 1)
&Lab    INCR  &V,&A,&Reg=0          Default work register = 0
&T      SetC  T'&V                  Type attribute of 1st arg
&Op     SetC  '&T'                  Save type of &V for mnemonic suffix
&I      SetC  '1'                   Default increment
        Aif   ('&A' eq '').IncOK    Increment now set OK
&I      SetC  '&A'                  Supplied increment (N.B. Not SETA!)
.IncOK Aif    ('&T' eq 'F').F,('&T' eq 'P').P,                          X
               ('&T' eq 'H' or '&T' eq 'D' or '&T' eq 'E').T
        MNote 8,'INCR: Cannot use type ''&T'' of ''&V''.'
        MExit
.F      ANOP  ,                     Type of &V is F
&Op     SetC  ''                    Null operation suffix
.T      ANOP  ,                     Register-types D, E, H (and F)
&Lab    L&Op  &Reg,&V               Fetch variable to be incremented
        A&Op  &Reg,=&T.'&I'         Add requested increment
        ST&Op &Reg,&V               Store incremented value
        MExit
.P      ANOP  ,                     Type of &V is P
&Lab    AP    &V,=P'&I'             Increment packed variable
        MEnd
```


Base-Language Type Sensitivity

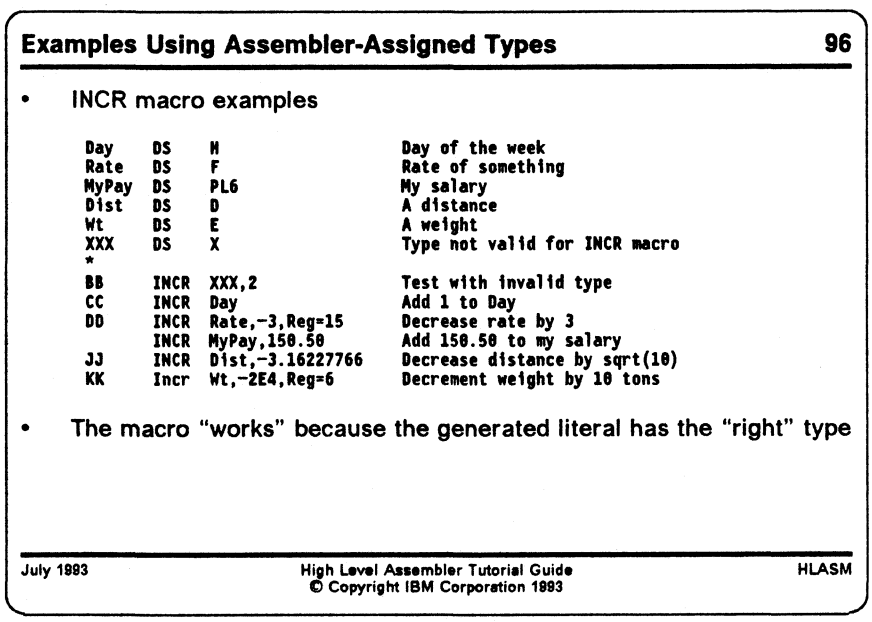
The assembler's assignment of type attributes to most forms of declared data lets us write macros that utilize the type information to make decisions about the instructions to be generated. For example, suppose we want to write a macro INCR to add a constant value to a variable, with default increment 1 if no value is specified in the macro call. Consider Figure 56:

```
        Macro
&Lab    INCR  &V,&A,&Reg=0
&T      SetC  T'&V                  Type attribute of 1st arg
&Op     SetC  '&T'                  Save type of &V for mnemonic suffix
&I      SetC  '1'                   Default increment
        Aif   ('&A' eq '').IncOK    Increment now set OK
&I      SetC  '&A'                  Supplied increment (N.B. Not SETA!)
.IncOK Aif    ('&T' eq 'F').F,('&T' eq 'P').P,                          X
               ('&T' eq 'H' or '&T' eq 'D' or '&T' eq 'E').T
        MNote 8,'INCR: Cannot use type ''&T'' of ''&V''.'
        MExit
.F      ANOP  ,                     Type of &V is F
&Op     SetC  ''                    Null operation suffix
.T      ANOP  ,                     Register-types D, E, H (and F)
&Lab    L&Op  &Reg,&V               Fetch variable to be incremented
        A&Op  &Reg,=&T.'&I'         Add requested increment
        ST&Op &Reg,&V               Store incremented value
        MExit
.P      ANOP  ,                     Type of &V is P
&Lab    AP    &V,=P'&I'             Increment variable
        MEnd
```

Figure 56. Macro Type Sensitivity to Base Language Types

The macro first determines the type attribute of the variable &V, and sets the increment value &I. The type attribute is checked for one of the five allowed types: D, E, F, H, and P. Finally, an instruction sequence appopriate to the variable's type is generated to perform the requested incrementation. This macro "works" because we can use the type attribute information about the variable &V to create a literal of the same type.

An observation: this macro represents a form of *polymorphism* in the sense that the operation it performs depends on the type(s) of its argument(s).

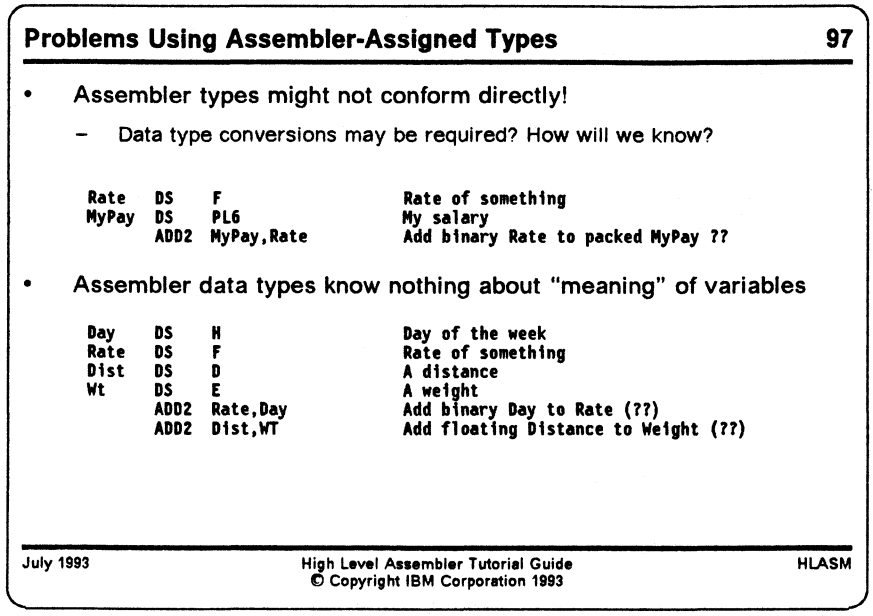


Some examples of calls to the INCR macro are shown in the following figure.

```
Day     DS    H                   Day of the week
Rate    DS    F                   Rate of something
MyPay   DS    PL6                 My salary
Dist    DS    D                   A distance
Wt      DS    E                   A weight
XXX     DS    X                   Type not valid for INCR macro
*
BB      INCR  XXX,2               Test with invalid type
CC      INCR  Day                 Add 1 to Day
DD      INCR  Rate,-3,Reg=15      Decrease rate by 3
        INCR  MyPay,150.50        Add 150.50 to my salary
JJ      INCR  Dist,-3.16227766    Decrease distance by sqrt(10)
KK      Incr  Wt,-2E4,Reg=6       Decrement weight by 10 tons
```

Figure 57. Examples: Macro Type Sensitivity to Base Language Types

Type sensitivity of this form can be used in many applications, and can help simplify program logic and structure.



Shortcomings of Assembler-Assigned Types

There are many benefits achievable from utilizing assembler type attributes. Suppose, however, that we wish to add two *variables* using a macro named ADD2 that works like the INCR macro just described. Two problems arise:

1. The types of the variables to be added may not "conform" by having the same assembler-assigned type attribute. For example, let some variables be defined as in Figure 56 on page 118:

```
Rate   DS    F                 Rate of something
MyPay  DS    PL6               My salary
```

   Then, if we can write a macro call like

```
       ADD2  MyPay,Rate        Add binary Rate to packed MyPay
```

   then some additional conversion work is needed because the types of the two variables do not allow direct addition. Such conversions are sometimes easy to program, either with inline code or with a call to a conversion subroutine. However, as the number of allowed types grows, the number of needed conversions may grow almost as the square of the number of types.

2. The more serious problem is that the assembler-assigned types may conform, but the programmer's "intended types" may have no sensible relationship to one another! Consider the same set of definitions:

```
Day    DS    H                 Day of the week
Rate   DS    F                 Rate of something
Dist   DS    D                 A distance
Wt     DS    E                 A weight
```

   Then, is is clear that we can write simple macros to implement these additions:

```
       ADD2  Rate,Day          Add binary Halfword to Fullword
       ADD2  Dist,WT           Add floating Distance to Weight
```

because the data types conform: halfword and fullword binary additions and short and long floating additions are supported by hardware instructions.

Consider, however, what is being added: in the first example, we are adding a "day" to a "rate" and in the second we are adding a "distance" to a "weight", and neither of these operations makes sense in the real world, even though a computer will blindly add the numbers representing these quantities.

Data Typing with User-Assigned Type Attributes 98

- Use third operand of EQU statement for type assignment:

```
symbol  EQU   expression,length,type
```

- Declaration of DATE types made by DclDate macro

```
         Macro
         DclDate
         GblC  &DateTyp                Type attr of "Date" variable
&DateTyp SetC  C'd'                    Type attr is lower case 'd'
&DateLen SetA  4                       Dates stored in 4 bytes
&NV      SetA  N'&SysList              Number of arguments to declare
&K       SetA  0                       Counter
.Test    Aif   (&K ge &NV).Done        Check for finished
&K       SetA  &K+1                    Increment argument counter
         DC    PL&DateLen.'0'          Define storage
&SysList(&K) EQU *-&DateLen.,&DateLen.,&DateTyp  Define name, length, type
         Ago   .Test
.Done    MEnd
```


Data Typing with User-Assigned Type Attributes ... 99

- Declaration of PERIOD types made by DclPerd macro

```
         Macro
         DclPerd  &Init=0              Optional initialization value
         GblC  &PerdTyp                Type attr of "Period" variable
         LclA  &PerdLen                Length of a "Period" variable
&PerdTyp SetC  C'p'                    Type attr is lower case 'p'
&PerdLen SetA  3                       Periods stored in 3 bytes
&NV      SetA  N'&SysList              Number of arguments to declare
&K       SetA  0                       Counter
.Test    Aif   (&K ge &NV).Done        Check for finished
&K       SetA  &K+1                    Increment argument counter
         DC    PL&PerdLen.'&Init.'     Define storage
&SysList(&K) EQU *-&PerdLen.,&PerdLen.,&PerdTyp  Define name, length, type
         Ago   .Test
.Done    MEnd
```

- Initial value can be specified with Init= keyword

User-Defined Type Attributes

One can obtain some (minor) relief from the limitations of the Assembler's assignment of type attributes by using the third operand of an EQU statement to assign user-defined type attributes to program objects. As a reminder, the full syntax of the EQU statement is

```
symbol  EQU   expression[,[length][,type_expression]]
```

The type_expression in the third operand must evaluate to an absolute quantity in the range from 0 to 255.

To overcome the limitations of using just assembler-assigned types, we will examine a set of macros that declare and operate on data items with just two specific types: calendar dates, and periods of elapsed time in days. With these two data types, we can perform certain kinds of arithmetic and comparisons:

- two dates may be subtracted to yield a period
- a period may be added or subtracted from a date to yield a date
- two periods may be added or subtracted
- dates may be compared with dates, and periods with periods

Any other operation involving dates and periods is invalid.

First, we will examine two macros that "declare" variables of type "date" and "period", (DclDate and DclPerd, respectively). Each macro will accept a list of names to be declared with that type, assign "private" type attributes `C'd'` and `C'p'`, and allocate storage for the variables.

First, we will illustrate a macro `DclDate` to declare variables of type "date".

```
         Macro
         DclDate
         GblC  &DateTyp                Type attr of "Date" variable
&DateTyp SetC  C'd'                    Type attr is lower case 'd'
&DateLen SetA  4                       Dates stored in 4 bytes
&NV      SetA  N'&SysList              Number of arguments to declare
&K       SetA  0                       Counter
.Test    Aif   (&K ge &NV).Done        Check for finished
&K       SetA  &K+1                    Increment argument counter
         DC    PL&DateLen.'0'          Define storage
&SysList(&K)  EQU   *-&DateLen.,&DateLen.,&DateTyp  Define name, length, type
         Ago   .Test
.Done    MEnd
*
         DclDate  Birth,Hire,Degree,Retire,Decease   Declare 5 date fields
         DclDate  LoanStart,LoanEnd                  Declare 2 date fields
```

Figure 58. Macro to Declare "DATE" Data Type

The `DclDate` macro accepts a list of names, and allocates a packed decimal storage of 4 bytes for each.

The `DclPerd` macro also accepts a list of names, and allocates a packed decimal field of 3 bytes for each; in addition, a keyword variable &Init can be used to supply an initial value for all the variables deciared on any one macro call.

```
         Macro
         DCLPERD  &Init=0          Declare a time period in days
         GblC  &PerdTyp            Type attr of "Period" variable
         LclA  &PerdLen            Storage length of elapsed periods
&PerdTyp SetC  C'p'                Type is lower case 'p'
&PerdLen SetA  3                   Length is 3 bytes
&NV      SetA  N'&SysList          Number of names to declare
&K       SetA  0                   Counter
.Test    Aif   (&K ge &NV).Done    Check for finish
&K       SetA  &K+1                Increment argument count
         DC    PL&PerdLen.'&Init'  Declare variable and initial value
&SysList(&K)  EQU  *-&PerdLen.,&PerdLen.,&PerdTyp  Declare name, length, type
         Ago   .Test               Check for more arguments
.Done    MEnd
*
Aaa      DclPerd  Vacation,Holidays
         DclPerd  LoanTime
         DclPerd  Year,Init=365
         DclPerd  Week,Init=7
```

Figure 59. Macro to Declare "PERIOD" Data Type

Calculating Date Variables: CalcDat Macro **100**

- Define user-called CalcDat macro to calculate dates:

```
&AnsDate CalcDat  &Arg1,Op,&Arg2      Calculate a Date variable
```

- Allowed forms are:

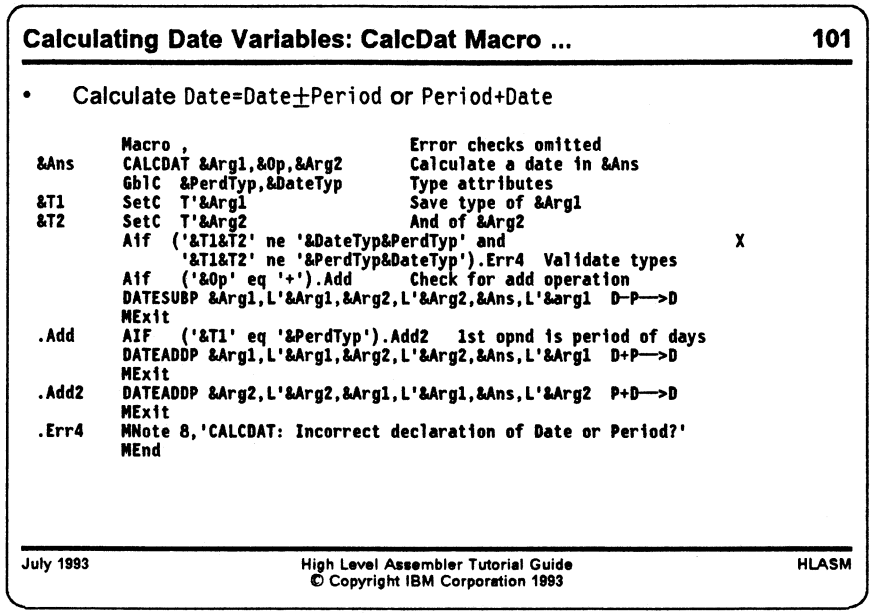
```
Date    CalcDat  Date,+,Period        Date = Date + Period
Date    CalcDat  Date,-,Period        Date = Date - Period
Date    CalcDat  Period,+,Date        Date = Period + Date
```

- CalcDat will validate types, and call two auxiliary macros:

```
        DATEADDP  Date1,Len1,Period,LPer,AnsDate,LenAns  Date+Per -> Date
        DATESUBP  Date1,Len1,Period,LPer,AnsDate,LenAns  Date-Per -> Date
```

  - Auxiliary service macros "understand" data representations

July 1993 High Level Assembler Tutorial Guide HLASM



Having written macros to declare the two data types, we can now consider macros for doing calculations with them. First, we will examine a date-calculation macro CALCDAT, with the following syntax:

```
&AnsDate CalcDat  &Arg1,Op,&Arg2      Calculate a Date variable
```

where &AnsDate must have been declared a "date" variable, and the allowed operand combinations are:

```
Date     CalcDat  Date,+,Period
Date     CalcDat  Period,+,Date
Date     CalcDat  Date,-,Period
```

We are now in a position to write a CalcDat macro that validates the types of all three operands before setting up the actual computations which will be done by two "service" macros called DATEADDP (to add a period to a date) and DATESUBP (to subtract a period from a date). These service macros will "understand" the actual representation of "date" and "period" variables, and can perform the operations accordingly.

```
         Macro
&Ans     CALCDAT &Arg1,&Op,&Arg2       Calculate a date in &Ans
&M       SetC  'CALCDAT: '             Macro name for messages
         GblC  &PerdTyp,&DateTyp       Type attributes
         Aif   (N'&SysList ne 3).Err1  Check for required arguments
         Aif   ('&Op' ne '+' and '&Op' ne '-').Err2
         Aif   (T'&Ans ne '&DateTyp').Err3
&T1      SetC  T'&Arg1                 Save type of &Arg1
&T2      SetC  T'&Arg2                 And of &Arg2
         Aif  ('&T1&T2' ne '&DateTyp&PerdTyp' and                        X
               '&T1&T2' ne '&PerdTyp&DateTyp').Err4  Validate types
         Aif   ('&Op' eq '+').Add      Check for add operation
         Aif   ('&T1&T2' ne '&DateTyp&Perdtyp').Err5 Bad operand seq?
         DATESUBP &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&arg1  D-P-->D
         MExit
.Add     AIF   ('&T1' eq '&PerdTyp').Add2  1st opnd a period of days
         DATEADDP &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Arg1  D+P-->D
         MExit
.Add2    DATEADDP &Arg2,L'&Arg2,&Arg1,L'&Arg1,&Ans,L'&Arg2  P+D-->D
         MExit
.Err1    MNote 8,'&M.Incorrect number of arguments'
         MExit
.Err2    MNote 8,'&M.Operator ''&Op'' not + or -'
         MExit
.Err3    Aif   (T'&Ans eq 'O').Err3a   Check for omitted target
         MNote 8,'&M.Target ''&Ans'' not declared by DCLDATE'
         MExit
.Err3A   MNote 8,'&M.Target Date variable omitted from name field'
         MExit
.Err4    MNote 8,'&M.Incorrect declaration of Date/Period arguments'
         MExit
.Err5    MNote 8,'&M.Subtraction operands in reversed order'
         MEnd
```

Figure 60. Macro to Calculate "DATE" Results

Some examples of calls to the CalcDat macro are shown in the following figure.

```
Hire     CalcDat  Degree,+,Year
Hire     CalcDat  Year,+,Degree
Hire     CalcDat  Degree,-,Year
```

Figure 61. Examples of Macro Calls to Calculate "DATE" Results

Calculating Period Variables: CalcPer Macro 102

- Define user-called CalcPer macro to calculate periods
- Allowed forms are:

```
Period   CalcPer  Date,-.Date         Difference of two date variables
Period   CalcDat  Period,+,Period     Sum of two period variables
Period   CalcDat  Period,-,Period     Difference of two period variables
Period   CalcDat  Period,*,Number     Product of a period and a number
Period   CalcDat  Period,/,Number     Quotient of a period and a number
```

- CalcPer will validate types, and call five auxiliary macros:

```
PERDADDP  Per1,Len1,Per2,Len2,AnsP,LenAns        Per +Per  -> Per
PERDSUBP  Per1,Len1,Per2,Len2,AnsP,LenAns        Per -Per  -> Per
PERDMULP  Per1,Len1,Per2,Len2,AnsP,LenAns        Per *Num  -> Per
PERDDIVP  Per1,Len1,Per2,Len2,AnsP,LenAns        Per /Num  -> Per
DATESUBD  Date1,Len1,Period,LPer,AnsDate,LenAns  Date-Date -> Per
```


Calculating Period Variables: CalcPer Macro ... 103

```
         Macro
&Ans     CALCPER  &Arg1,&Op,&Arg2
         GblC  &PerdTyp,&DateTyp          Type attributes
&X(C'+') SetC  'ADD'                      Name for Add routine
&X(C'-') SetC  'SUB'
&X(C'*') SetC  'MUL'
&X(C'/') SetC  'DIV'
&Z       SetC  'C''&Op'''                 Convert &Op char to SDT
&T1      SetC  T'&Arg1                    Type of Arg1
&T2      SetC  T'&Arg2                    Type of Arg2
         Aif   ('&T1&T2&Op' eq '&DateTyp&DateTyp.-').DD  Chk date-date
         Aif   ('&T2' ne 'N').PP          Second operand nonnumeric
         PERD&X(&Z).P Arg1,L'&Arg1,=PL3'&Arg2',3,&Ans,L'&Ans P op const
         MExit
.PP      PERD&X(&Z).P &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans P op P
         MExit
.DD      DATESUBD  &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans  date-date
         MEnd
```


A second macro CalcPer to calculate periods of time is similar in concept, but somewhat more complex because of a greater allowed set of operand combinations:

```
&AnsPerd CalcPer  &Arg1,Op,&Arg2      Calculate a Period variable
```

where &AnsPerd must have been declared a "period" variable, and the allowed operand combinations are:

```
Period   CalcPer  Date,-.Date         Difference of two date variables
Period   CalcDat  Period,+,Period     Sum of two period variables
Period   CalcDat  Period,-,Period     Difference of two period variables
Period   CalcDat  Period,*,Number     Product of a period and a number
Period   CalcDat  Period,/,Number     Quotient of a period and a number
```

The CalcPer macro validates its arguments before generating calls to the "operational" macros that do the actual arithmetic.

```
         Macro
&Ans     CALCPER  &Arg1,&Op,&Arg2
         GblC  &PerdTyp,&DateTyp          Type attributes
&M       SetC  'CALCPER: '                Macro name for messages
         Aif   (N'&SysList ne 3).Err1     Wrong number of arguments
         Aif   (T'&Ans ne 'p').Err2       Invalid target
         Aif   (T'&Op ne 'U' or K'&Op ne 1).Err5    Invalid operator
&X(C'+') SetC  'ADD'                      Name for Add routine
&X(C'-') SetC  'SUB'
&X(C'*') SetC  'MUL'
&X(C'/') SetC  'DIV'
&Z       SetC  'C''&Op'''                 Convert &Op char to SDT
&T1      SetC  T'&Arg1                    Type of Arg1
&T2      SetC  T'&Arg2                    Type of Arg2
         Aif   ('&T1&T2&Op' eq '&DateTyp&DateTyp.-').DD  Chk date-date
         Aif   ('&T1' ne '&PerdTyp').Err3        Invalid first operand
         Aif   ('&T2' eq '&PerdTyp' and                                X
               ('&Op' eq '+' or '&Op' eq '-')).PP
         Aif   ('&Op' eq '+' or '&Op' eq '-' or '&Op' eq '*').OpOK,    X
               ('&Op' ne '/').Err5
.OpOK    Aif   ('&T2' ne 'N').Err4        Second operand nonnumeric
.*       Third operand is a constant
         PERD&X(&Z).P Arg1,3,=PL3'&Arg2',3,&Ans,3  period op const
         MExit
.PP      PERD&X(&Z).P &Arg1,3,&Arg2,3,&Ans,3   period op period
         MExit
.DD      DATESUBD  &Arg1,4,&Arg2,4,&Ans,3 Difference of 2 dates
         MExit
.Err1    MNote 8,'&M.Incorrect number of arguments'
         MExit
.Err2    Aif   (T'&Ans ne 'O').Err2A      Check for omitted target
         MNote 8,'&M.Target variable omitted'
         MExit
.Err2A   MNote 8,'&M.Target ''&Ans'' not declared by DCLPERD'
         MExit
.Err3    MNote 8,'&M.First argument invalid or not declared by DCLPERD'
         MExit
.Err4    MNote 8,'&M.Third argument invalid or not declared by DCLPERD'
         MExit
.Err5    MNote 8,'&M.Invalid (or missing) operator ''&Op'''
         MEnd
```

Figure 62. Macro to Calculate "PERIOD" Results

```
Year     CALCPER  Year,+,Year        Period + Period
Year     CALCPER  Hire,-,Degree      Date - Date
Year     CALCPER  Hire,-,Hire        Date - Date
Year     CALCPER  Year,-,Year        Period - Period
Year     CALCPER  Year,+,10          Period + Number
Year     CALCPER  Year,-,10          Period - Number
Year     CALCPER  Year,*,10          Period * Number
Year     CALCPER  Year,/,10          Period / Number
```

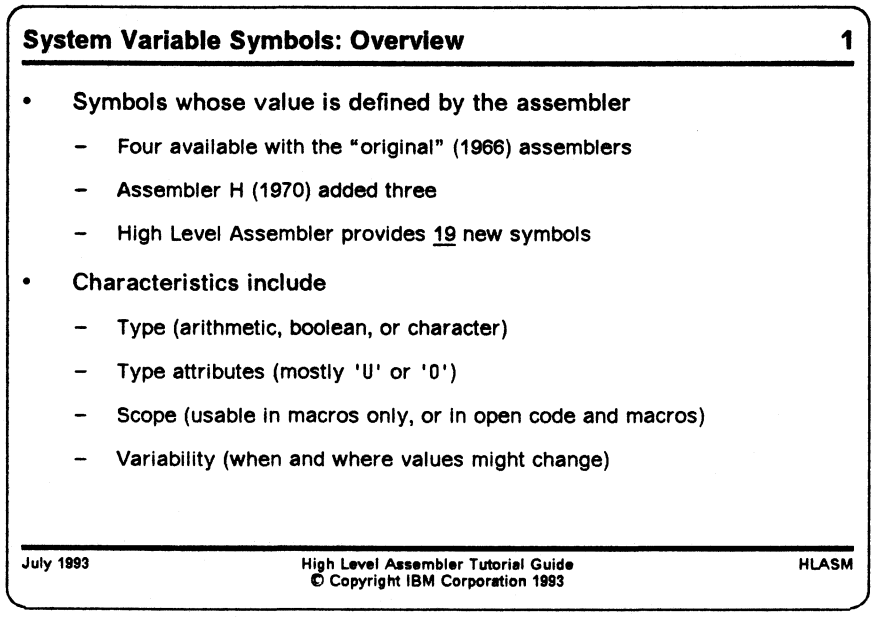
Figure 63. Examples of Macro Calls to Calculate "PERIOD" Results

As you can see, these macros provide a fairly strong degree of type checking of their arguments to ensure that they conform to the sets of operations appropriate to their types. If we had written only machine instructions, the opportunities for operand type conflicts, or operator-operand conflicts, would not only have been larger, but might have gone undetected. In addition, once a set of useful macros has been coded, you can think in terms of "higher level" operations, and avoid the many details necessary to deal with the actual machine instructions.

System (&SYS) Variable Symbols

System variable symbols are a special class of variable symbols, starting with the characters &SYS. They are "owned" by the assembler: they may not be declared in LCLx or GBLx statements, and may not be used as symbolic parameters. Their values are assigned by the assembler, and never by SETx statements.

High Level Assembler provides many new system variable symbols: nineteen will be new to users of the H-Level Assembler, and three additional symbols will be new to users of the DOS/VSE Assembler. Four symbols are available in all three assemblers: &SYSECT, &SYSLIST, &SYSNDX, and &SYSPARM. Figure 64 on page 131 summarizes their properties.



System Variable Symbols: Properties

The symbols have a variety of characterizations:

- Availability

  Four symbols are available in all (designated "All") assemblers for the System/360/370/390 family of processors. Three others are available in Assembler H (designated "AsmH"); High Level Assembler provides the richest set of twenty-six system variable symbols (designated "HLA").

- Type

  Most symbols have character values, and are therefore of type C: that is, they would normally be used in SETC statements or in similar contexts. A few, however, have arithmetic values (type A) or boolean values (type B). &SYSDATC and &SYSSTMT are nominally type C, but may also be used as type A.

- Type attributes

  Most system variable symbols have type attribute U ("undefined") or O ("omitted", usually indicating a null value); some numeric variables have type N. The exception is &SYSLIST: its type attribute is determined from the designated list item.

- Scope of usage

Some symbols are usable only within macros ("local" scope), while others are usable both within macros and in open code ("global" scope).

- Variability

  Some symbols have values that do not change as the assembly progresses. Normally, such values are established at the beginning of an assembly. These values are denoted "Fixed". Note that all have Global scope.

  Other symbols have values that may change during the assembly. These values might be established at the beginning of an assembly or at some point subsequent to the beginning, and may change depending on conditions either internal or external to the assembly process.

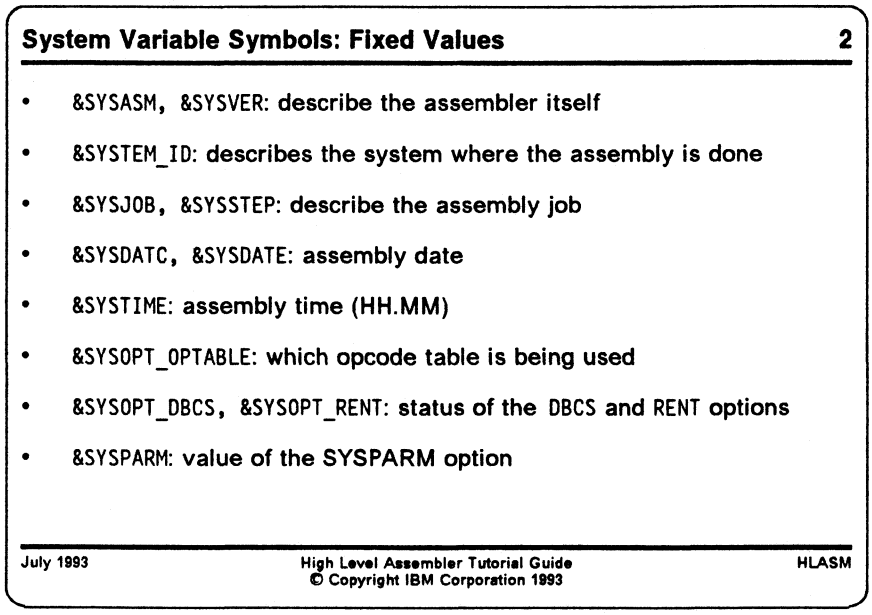
  - Variables whose values are established at the beginning of a macro expansion, and for this the values remain unchanged throughout the expansion, are designated "Constant", even though they may have different values in a later expansion of the same macro, or within "inner macros" invoked by another macro. Note that all have local scope.

  - Variables whose values may change within a single macro expansion are designated "Variable". Currently, this designation applies only to &SYSSTMT.

These symbols have many uses: helping to control conditional assemblies, capturing environmental data for inclusion in the generated object code, providing program debugging data, and more.

Variable Symbol	Availability	Type	Type Attr.	Scope	Variability	Content and Use
&SYSASM	HLA	C	U	Global	Fixed	Assembler name
&SYSDATC	HLA	C,A	N	Global	Fixed	Assembly date, including century, in YYYYMMDD format
&SYSDATE	AsmH	C	U	Global	Fixed	Assembly date in MM/DD/YY format
&SYSECT	All	C	U	Local	Constant	Current control section name
&SYSIN_DSN	HLA	C	U	Local	Constant	Current primary input data set name
&SYSIN_MEMBER	HLA	C	U,O	Local	Constant	Current primary input member name
&SYSIN_VOLUME	HLA	C	U,O	Local	Constant	Current primary input data set name volume identifier
&SYSJOB	HLA	C	U	Global	Fixed	Assembly job name
&SYSLIB_DSN	HLA	C	U	Local	Constant	Current library data set name
&SYSLIB_MEMBER	HLA	C	U,O	Local	Constant	Current library member name
&SYSLIB_VOLUME	HLA	C	U,O	Local	Constant	Current library data set volume identifier
&SYSLIST	All	C	any	Local	Constant	Macro argument list and individual list and sublist elements
&SYSLOC	AsmH	C	U	Local	Constant	Current location counter name
&SYSNDX	All	C	N	Local	Constant	Macro invocation count
&SYSNEST	HLA	A	N	Local	Constant	Nesting level of the macro call
&SYSOPT_DBCS	HLA	B	N	Global	Fixed	Setting of DBCS invocation parameter
&SYSOPT_OPTABLE	HLA	C	U	Global	Fixed	Setting of OPTABLE invocation parameter
&SYSOPT_RENT	HLA	B	N	Global	Fixed	Setting of RENT invocation parameter
&SYSPARM	All	C	U,O	Global	Fixed	Value provided by SYSPARM invocation parameter
&SYSSEQF	HLA	C	U,O	Local	Constant	Sequence field of current open code statement
&SYSSTEP	HLA	C	U	Global	Fixed	Assembly step name
&SYSSTMT	HLA	C,A	N	Global	Variable	Number of next statement to be processed
&SYSSTYP	HLA	C	U,O	Local	Constant	Current control section type
&SYSTEM_ID	HLA	C	U	Global	Fixed	System on which assembly is done
&SYSTIME	AsmH	C	U	Global	Fixed	Assembly start time
&SYSVER	HLA	C	U	Global	Fixed	Assembler version

Figure 64. Properties and Uses of System Variable Symbols



Variable Symbols With Fixed Values During an Assembly

&SYSASM and &SYSVER

The &SYSASM symbol provides the name of the assembler. For High Level Assembler, the value of this variable is

```
HIGH LEVEL ASSEMBLER
```

The &SYSVER variable symbol describes the version, release, and modification of the assembler. A typical value of this variable might be

```
1.1.0
```

This pair of variables could be used to provide identification within an assembled program of the assembler used to assemble it:

```
What_ASM  DC  C'Assembled by &SYSASM., Version &SYSVER..'
```

&SYSTEM_ID

The &SYSTEM_ID variable provides an identification of the operating system under which the current assembly is being performed. A typical value of this variable might be

```
MVS/ESA SP 4.2.0
```

This variable could be used to provide identification within an assembled program of the system on which it was assembled:

```
What_Sys  DC  C'Assembled on &SYSTEM_ID..'
```

&SYSJOB and &SYSSTEP

These two variables provides the name of the job and step under which the assembler is running.

When assembling under the CMS system, the value of the &SYSJOB variable is always (NOJOB); and when assembling under the CMS or VSE systems, the value of the &SYSSTEP variable is always (NOSTEP).

This pair of variables could be used to provide identification within an assembled program of the job and step used to assemble it:

```
Who_ASM   DC  C'Assembled in Job &SYSJOB., Step &SYSSTEP..'
```

&SYSDATC

This provides the current date, with century included, in the format YYYYMMDD. A typical value of this variable might be

```
19920626
```

Observe that the &SYSDATE variable provides only two digits of the year, in the form MM/DD/YY.

&SYSDATE

&SYSDATE provides the current date, in the form MM/DD/YY. A typical value of this variable might be

```
06/26/92
```

&SYSTIME

The &SYSTIME variable provides the time at which the assembly started, in the form HH.MM.

This variable, along with &SYSDATE or &SYSDATC, could be used to provide identification within an assembled program of the date and time of assembly:

```
When_ASM  DC  C'Assembled on &SYSDATC., at &SYSTIME..'
```

&SYSOPT_OPTABLE

This variable provides the name of the current operation code table being used for this assembly, as established by the OPTABLE option. A typical value of this variable might be

```
ESA
```

This variable is useful for creating programs that must execute on machines with limitations on the set of available instructions. For macro-generated code, this variable can be used to determine what instructions should be generated for various operations, e.g. BALR vs. BASR.

This variable could be used to provide identification within an assembled program of the operation code table used to assemble it:

```
What_Ops  DC  C'Opcode table for assembly was &SYSOPT_OPTABLE..'
```

&SYSOPT_DBCS and &SYSOPT_RENT

The &SYSOPT_DBCS and &SYSOPT_RENT binary variables provide the settings of the DBCS and RENT options, respectively. Their values can be used to control the generation of instructions or data, or to help control the scanning of macro arguments.

For example, character data to be included in constants can be generated with proper encodings if DBCS environments must be considered. Similarly, macros can use the setting of the RENT option to generate different instruction sequences for reentrant and nonreentrant situations.

The &SYSOPT_RENT variable could be used to provide conditional assembly support for different code sequences:

```
          AIF    (&SYSOPT_RENT).Do_Rent
          MYMAC  Parm1,Parm2,GENCODE=NORENT  Generate non-RENT code
          AGO    .Continue
.Do_Rent  MYMAC  Parm1,Parm2,GENCODE=RENT    Generate RENT code
.Continue ANOP
```

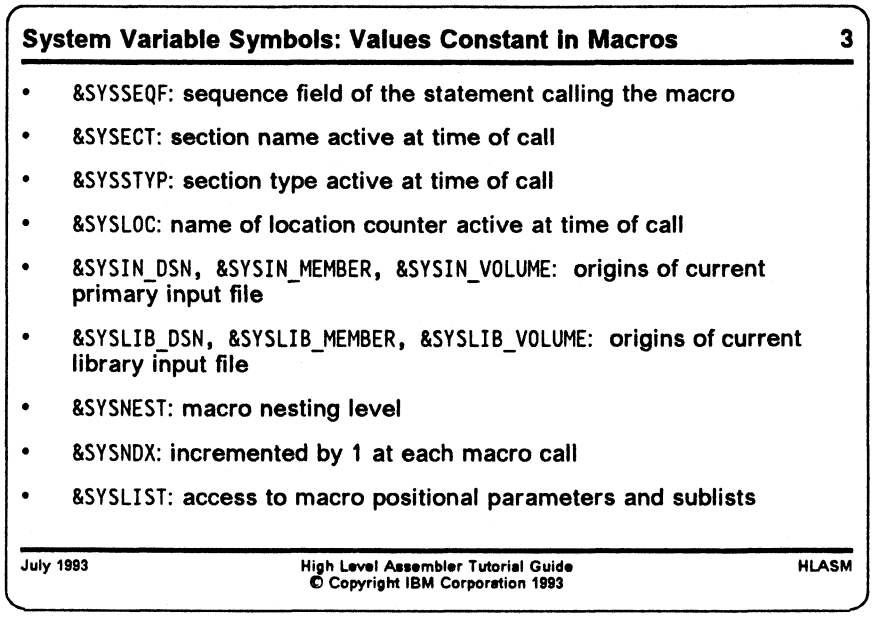
&SYSPARM

The &SYSPARM variable symbol provides the character string provided by the programmer in the invoking parameter string, in the SYSPARM option:

```
SYSPARM(string)
```

This variable could be used to provide identification within an assembled program of the &SYSPARM value used to assemble it, as well as to control conditional assembly activities:

```
What_PRM  DC    C'&&SYSPARM value was ''&SYSPARM.''.'

.X14      AIF   ('&SYSPARM' NE 'TRACE').Skip_Trace
          MNOTE 'Assembly reached Sequence Symbol .X14'
.Skip_Trace  ANOP
```



Variable Symbols With Constant Values Within a Macro

&SYSSEQF

The &SYSSEQF symbol provides the contents of the sequence field of the current input statement. This information can be used for debugging data. For example, suppose we have a macro which inserts information about the current sequence field into the object code of the program, and sets R0 to its address (so that a debugger can tell you which statement was identified in some debugging activity). A macro like the following might be used:

```
         Macro
&L       DebugPtA
&L       BAS   0,*+12            Addr of Sequence Field in R0
         DC    CL8'&SYSSEQF'     Sequence Field info
         MEnd
         - - -
B        DebugPtA
```

&SYSECT

The &SYSECT symbol provides the name of the control section (CSECT, DSECT, COM, or RSECT) into which statements are being grouped or assembled at the time the referencing macro was invoked. If a macro must generate code or data in a different control section, this variable permits the macro to restore the name of the previous environment before exiting. (Note also its relation to &SYSSTYP.) An example illustrating &SYSECT and &SYSSTYP is shown below.

&SYSSTYP

The &SYSSTYP symbol provides the type of the control section into which statements are being grouped or assembled (CSECT, DSECT, or RSECT) at the time the referencing macro was invoked. If a macro must generate code or data in a different control section, this variable permits the macro to restore the proper type of control section for the previous environment, before exiting.

For example, suppose we need to generate multiple copies of a small DSECT. The macro shown in the following example generates the DSECT so that each generated name is prefixed with the characters supplied in the macro argument. The environment in which the macro was invoked is then restored on exit from the macro.

```
         Macro
         DSectGen  &P
&P.Sect  DSect ,                 Generate tailored DSECT name
&P.F1    DS     D                DSECT Field No. 1
&P.F2    DS     18F              DSECT Field No. 2, a save area
&SYSECT  &SYSSTYP                Restore original section
         MEnd
```

&SYSLOC

&SYSLOC contains the name of the current location counter, as defined either by a control section definition or a LOCTR statement.

As in the example of &SYSSTYP, the &SYSLOC variable can be used to capture and restore the current location counter name. We again suppose in this example that we are interrupting the statement flow to generate a small DSECT:

```
         Macro
         DSectGen  &P
&P.Sect  DSect                   Generate the DSECT name
&P.F1    DS     D                DSECT Field No. 1
&P.F2    DS     18F              DSECT Save Area
&SYSLOC  LOCTR                   Restore previous location counter
         MEnd
```

&SYSIN_DSN, &SYSIN_MEMBER, and &SYSIN_VOLUME

These three symbols identify the origins of the current primary input file. Their values change across input-file concatenations. This information can be used to determine reassembly requirements.

The &SYSIN_DSN symbol provides the name of the current primary input (SYSIN) data set or file.

The &SYSIN_MEMBER symbol provides the name of the current primary input member, if any.

The &SYSIN_VOLUME symbol provides the name of the current primary input volume. For example, the following SYSINFO macro will capture the name of the current input file, its member name, and the volume identifier. (If the input does not come from a library member, the member name will be replaced by the characters "(None)".)

```
        Macro
&L      SYSINFO
&L      DC    C'Input: &SYSIN_DSN'
&Mem    SetC  '&SYSIN_MEMBER'
        AIF   ('&Mem' ne '').Do_Mem
&Mem    SetC  '(None)'
.Do_Mem DC    C'Member: &Mem'
        DC    C'Volume: &SYSIN_VOLUME'
        MEnd
My_Job  SYSINFO
```

&SYSLIB_DSN, &SYSLIB_MEMBER, and &SYSLIB_VOLUME

These three symbols identify the origins of the current library member. Their values change from member to member. This information can be used to determine reassembly requirements.

The &SYSLIB_DSN symbol provides the name of the library data set from which each macro and COPY file is retrieved.

The &SYSLIB_MEMBER symbol provides the name of the library member from which this macro and COPY file is retrieved.

The &SYSLIB_VOLUME symbol provides the volume identifier (VOLID) of the library data set from which this macro and COPY file is retrieved.

For example, suppose the LIBINFO macro below is stored in a macro library accessible to the assembler at assembly time. (The macro includes a test for a blank member name, which should never occur.)

```
        Macro
&L      LIBINFO
&L      DC    C'Library Input: &SYSLIB_DSN'
&Mem    SetC  '&SYSLIB_MEMBER'
        AIF   ('&Mem' ne '').Do_Mem
        MNote 4,'The library member name should not be null.'
.Do_Mem DC    C'Member: &Mem'
        DC    C'Volume: &SYSLIB_VOLUME'
        MEnd
```

Then the following small test assembly would capture information into the object text of the generated program about the macro library.

```
My_Job  LIBINFO
        End
```

&SYSNEST

The &SYSNEST arithmetic variable provides the nesting level at which the current macro was invoked (the outermost macro is at level 1).

For example, a macro might contain tests or MNOTE statements to indicate the nesting depth:

```
        AIF   (&SYSNEST LE 50).OK
        MNOTE 12,'Macro nesting depth exceeds 50. Possible recursion?'
        MEXIT
.OK     ANOP
```

&SYSNDX

The &SYSNDX variable provides a unique value for every macro invocation in the program. It may be used as a suffix for symbols generated in the macro, so that they will not "collide" with similar symbols generated in other invocations. It is incremented by 1 for every macro call in the program.

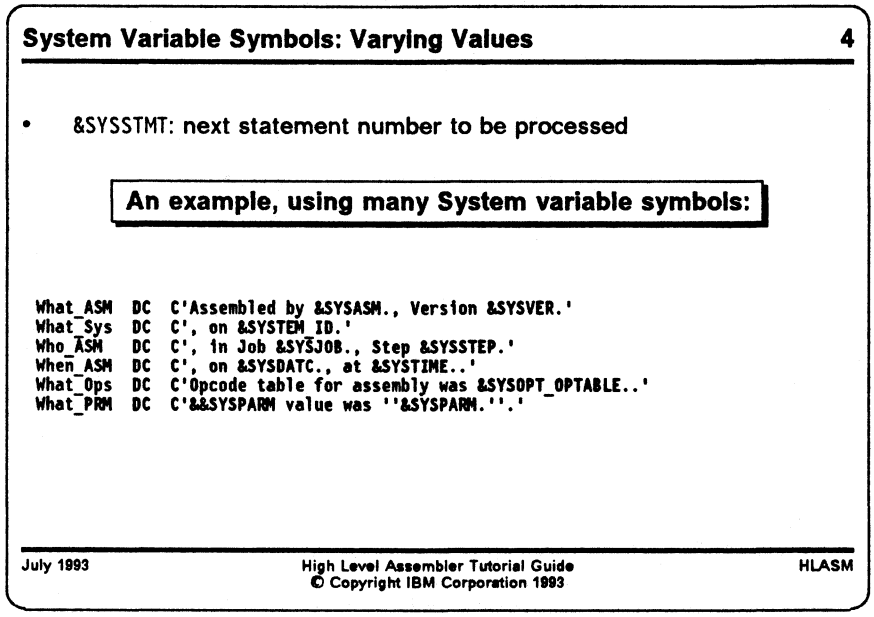
For values of &SYSNDX less than or equal to 9999, the value will always be four characters long (padded on the left with leading zeros, if necessary).

```
             Macro
&L           BDisp  &Target      Branch to non-addressable target
&L           BAS  1,Add&SYSNDX   Skip over constant
Off&SYSNDX   DC   Y(&Target-*)   Target offset
Add&SYSNDX   AH   1,Off&SYSNDX   Add offset
             BR   1              Branch to target
             MEnd
```

&SYSLIST

The &SYSLIST variable can be used to access positional parameters on a macro call (whether named or not). &SYSLIST supports a very rich set of sublist and attribute capabilities, and is therefore quite different from the other system variable symbols.

```
&NArgs   SETA   N'&SYSLIST       Number of arguments
&Arg_1   SETC   '&SYSLIST(1)'    Argument 1
&NArgs_1 SETA   N'&SYSLIST(1)    Number of sub-arguments
&Arg_2   SETC   '&SYSLIST(2)'    Argument 2
```



Variable Symbols Whose Values May Vary Anywhere

&SYSSTMT

The &SYSSTMT symbol provides the number of the next statement to be processed by the assembler. Debugger data that depends on the statement number can be generated with this variable. For example, suppose we have a macro which inserts information about the current statement number into the object code of the program, and sets R0 to its address (so that a debugger can tell you which statement was identified in some debugging activity). A macro like the following might be used:

```
        Macro
&L      DebugPtN
&L      BAS   0,*+8             Addr of Statement Number in R0
        DC    AL4(&SYSSTMT)     Statement number information
        MEnd

D       DebugPtN
```

System Variable Symbols Not Available in DOS/VSE

There are three system variable symbols supported by High Level Assembler which were previously available in Assembler H, but which were not available in the DOS/VSE Assembler: &SYSDATE, &SYSTIME, and &SYSLOC. (&SYSLOC relies on the availability of the LOCTR statement, which was not available in assemblers prior to Assembler H.)

Relationships to Previous System Variable Symbols

Some of the new system variable symbols introduced with High Level Assembler complement and supplement the data provided by system variable available in previous assemblers.

&SYSDATE and &SYSDATC

The variable symbol &SYSDATE is available in High Level Assembler and Assembler H, but not in any earlier assemblers. It provides a date in "American" format, without any century indication. As such, users in other countries sometimes had to extract and re-compose its fields to obtain a date conforming to local custom, convention, or standards. Further, the date could not be placed directly into fields as a sort key, because the year digits were in the lowest-order positions. Finally, no century was indicated.

High Level Assembler's introduction of the &SYSDATC variable solves all these problems very simply.

&SYSECT and &SYSSTYP

All previous assemblers have supported the &SYSECT variable to hold the name of the enclosing control section at the time a macro was invoked. This allows a macro which needs to change control sections (e.g., to declare a DSECT or to create code or data for a different CSECT) to resume the original control section on exit from the macro. There was, however, a sticky problem: there was no way for the macro to determine what *type* of control section to resume!

High Level Assembler provides the &SYSSTYP variable to rectify this omission: it provides the type of the control section named by &SYSECT. This permits a macro to restore the correct previous "control section environment" on exit.

&SYSNDX and &SYSNEST

All previous assemblers have supported the &SYSNDX variable symbol, which is incremented by one for every macro invocation in the program. This permits macros to generate unique ordinary symbols if they are needed as "local labels". Occasionally, in recursively nested macro calls, the value of the &SYSNDX variable was used to determine either the depth of nesting, or to determine when control had returned to a particular level.

Alternatively, the programmer could define a global variable symbol of his own, and in each macro insert statements to increment that variable on entry and decrement it on exit. This technique is both clumsy (because it requires extra coding in every macro) and insecure (because not every macro called in a program is likely to be under the programmer's control, particularly IBM-supplied macros).

High Level Assembler provides the &SYSNEST variable to keep track of the level of macro-call nesting in the program. The value of &SYSNEST is incremented globally on each macro entry, and decremented on each exit.

&SYSTIME and the AREAD Statement

The &SYSTIME variable symbol is provided by High Level Assembler and Assembler H, but not by eariler assemblers. It provides the local time of the start of the assembly in HH/MM format. This "time stamp" may not have sufficient accuracy or resolution for some applications.

High Level Assembler provides an extension to the AREAD statement that may be useful if a more accurate time stamp is required. The current time can be obtained either in decimal or binary format.

The macro in the following example captures the clock reading in both decimal and binary formats:

```
          Macro
&Lab      AREADCLK
          LCLC   &D,&B
&D        Aread  CLOCKD
&B        Aread  CLOCKB
&Lab      DC     C'&D'        Decimal Clock
          DC     C'&B'        Binary  Clock
          MEnd

A         AREADCLK
```