A++: Using the World's Most Powerful Macro Facility With High Level Assembler

Parts 1 and 2

SHARE 119, Anaheim CA, Summer 2012

Presented by Tom Wasik, IBM
wasik@us.ibm.com

Contents prepared by John R. Ehrman
ehrman@us.ibm.com

International Business Machines Corporation
Silicon Valley Laboratory
555 Bailey Avenue
San Jose, California 95141 USA

Synopsis:

This tutorial introduces the powerful concepts of conditional assembly supported by the High Level Assembler for z/OS, z/VM, & z/VSE, and provides examples of its use in both "open code" and in macro instructions.

The examples in this document are for purposes of illustration only, and no warranty of correctness or applicability is implied or expressed.

Copyright Notices and Trademarks

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

IBM	System/370	System/390	zSeries
z/OS	z/VM	z/VSE	OS/390
DFSMS	z/Architecture	ESA	

Publications and Web Site

The currently available product publications for High Level Assembler for z/OS, z/VM, and z/VSE are:

- High Level Assembler for z/OS, z/VM, and z/VSE *Language Reference*, SC26-4940
- High Level Assembler for z/OS, z/VM, and z/VSE *Programmer's Guide*, SC26-4941
- High Level Assembler for z/OS, z/VM, and z/VSE *Licensed Program Specifications*, GC26-4944
- High Level Assembler for z/OS, z/VM, and z/VSE *Installation and Customization Guide*, SC26-3494

The currently available product publications for High Level Assembler for z/OS, z/VM, and z/VSE Toolkit Feature are:

- High Level Assembler for z/OS, z/VM, and z/VSE *Toolkit Feature Interactive Debug Facility User's Guide*, GC26-8709
- High Level Assembler for z/OS, z/VM, and z/VSE *Toolkit Feature User's Guide*, GC26-8710
- High Level Assembler for z/OS, z/VM, and z/VSE *Toolkit Feature Installation and Customization Guide*, GC26-8711
- High Level Assembler for z/OS, z/VM, and z/VSE *Toolkit Feature Interactive Debug Facility Reference Summary*, GC26-8712

The currently available product publications for High Level Assembler for Linux on zSeries are:

- High Level Assembler for Linux on System z *User's Guide*, SC18-9611

HLASM publications are available online at the HLASM web site:

```
http://www.ibm.com/software/awdtools/hlasm/
```

Acknowledgments

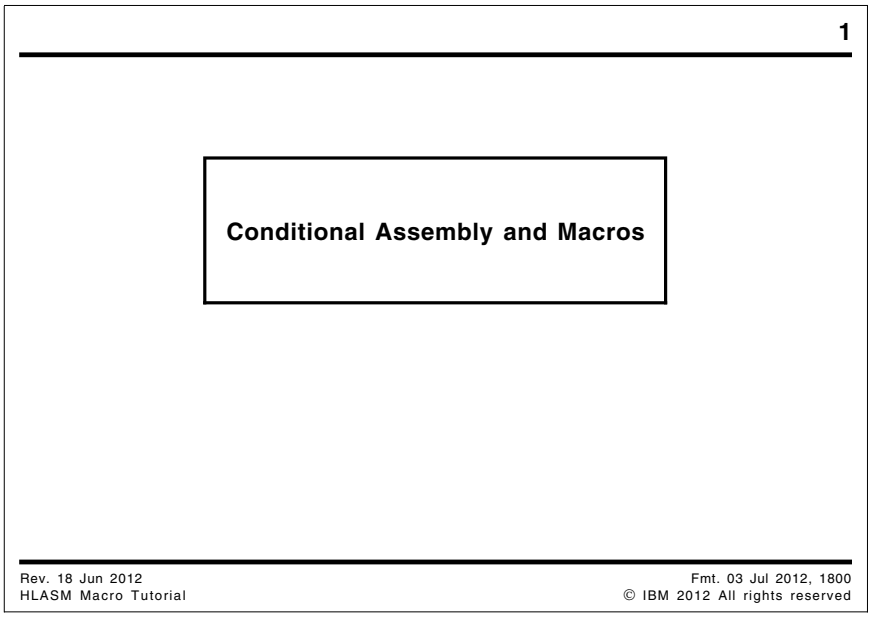
I am grateful to many people who have helped make this material more reliable and readable:

- Friends and colleagues at SHARE have offered advice (and corrections) during and after presentations.
- Abe Kornelis (abe@bixoft.nl) provided many helpful suggestions and caught many inaccuracies.

Contents

Figures

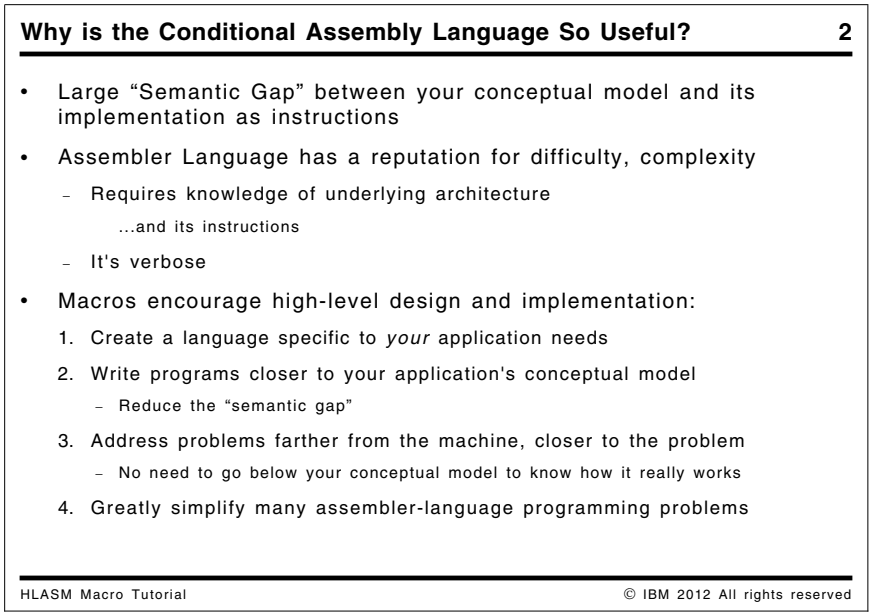
Conditional Assembly and Macros



We describe a powerful capability of High Level Assembler for z/OS, z/VM, & z/VSE that helps you tailor programs to your specific needs: the “Conditional Assembly and Macro Facility”.

The presentation will show why HLASM's conditional assembly and macro capabilities are superior in many ways to those of other programming languages, including “high-level” languages.

Why is the Conditional Assembly Language So Useful?



Assembler Language has a reputation for difficulty and complexity: it's verbose, and requires knowledge of the underlying architecture and relevant parts of its instruction set.

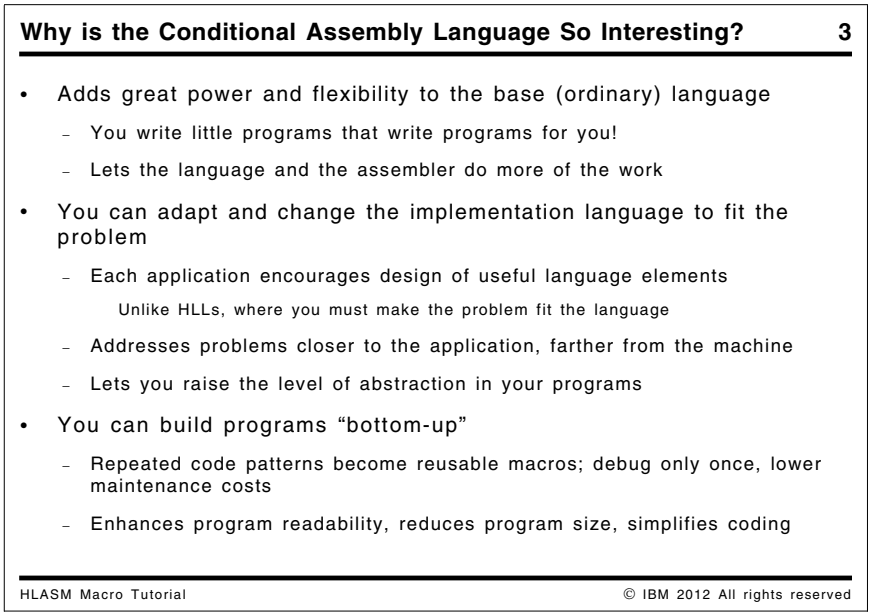
The level of detail normally required for Assembler Language programming normally implies a “Semantic Gap” between your conceptual model of the application and its implementation.

Good design expects that you should have no need to go “below” your conceptual model to know how the application really works. Thus, you will be able to address problems farther from the machine and closer to the problem.

Conditional assembly and macros help you reduce the “semantic gap” between your concept and its expression, and let you write programs much closer to the application's conceptual model.

We will see many examples showing how you can do this.

Why is the Conditional Assembly Language So Interesting?



Understanding the conditional assembly language not only adds to your knowledge of useful programming techniques, but also lets you think about application programming in new and different ways. You can design a language that best fits the application, rather than adapting the design of the application to fit the rules of a language.

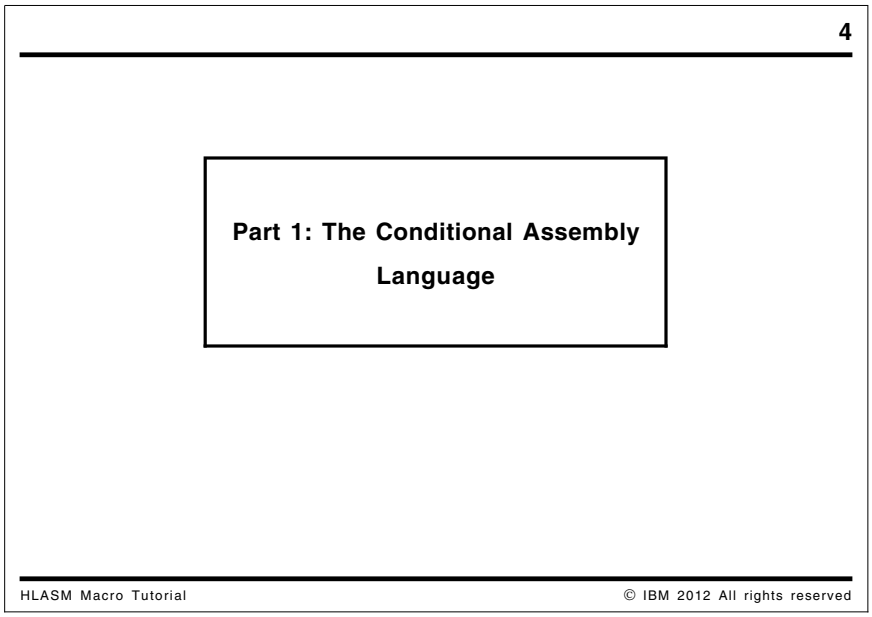
In writing macros, you effectively write small code-generating programs; this lets you do less work, in favor of the assembler doing more.

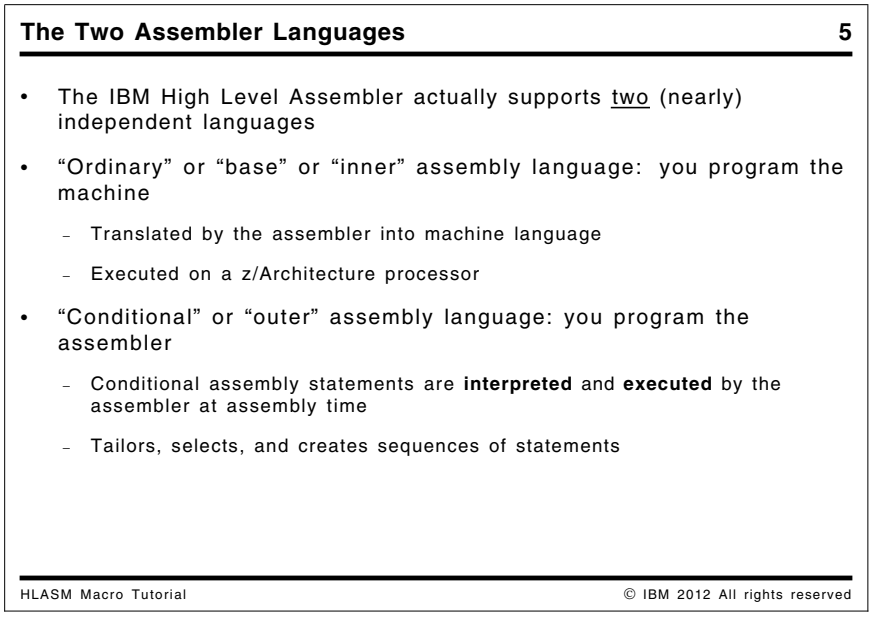
Thus, you can build an application from the “bottom up”. That is, by identifying the common, repeated elements of the application, you can create operations (macros) that reduce your concern with details, so your program and its language can evolve together. Those common elements can then be used throughout the application (code re-use!).

- Part 1 describes the conditional assembly language and its rich set of facilities and features.
- Part 2 explores basic concepts of macros and their definition and use in the Assembler Language.
- Part 3 provides “case study” examples of macro programming with High Level Assembler for z/OS, z/VM, & z/VSE.

Sometimes the conditional assembly language is called “macro language”, but since its use is not limited to macro instructions, we will use the more general term.

Part 1: The Conditional Assembly Language





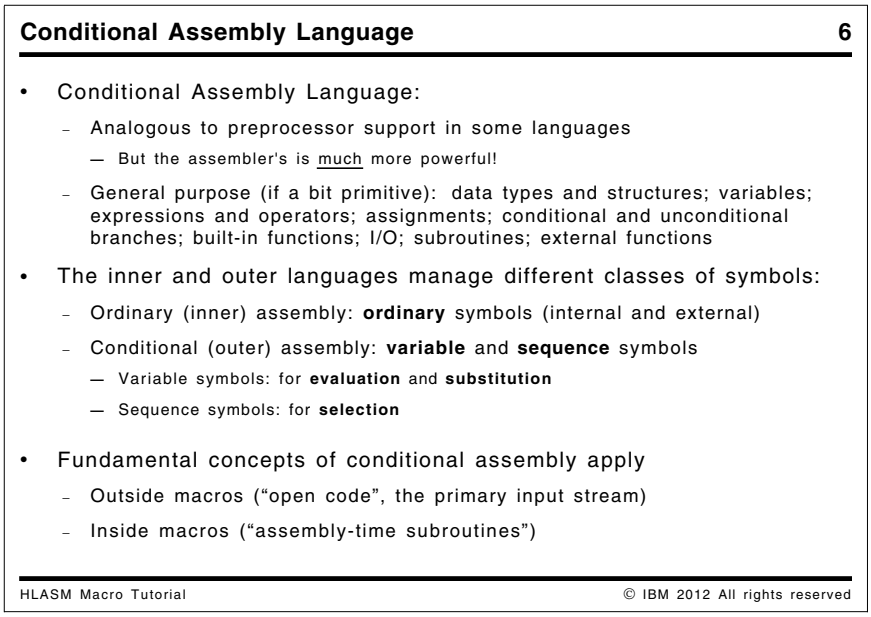
The Assembler Language is actually a mixture of two distinct languages:

- Ordinary assembly language is the familiar "base language" of machine and assembler instruction statements, translated by the assembler into machine language.
- Conditional assembly and macro language is the language of conditional statements, variable symbols, and macros. It is *interpreted* and *executed* by the assembler at assembly time to tailor, select, and create sequences of statements.

The conditional assembly and macro language has its own rules for declaring variables, assigning values, testing conditions, and generating values. The elements of the conditional language are statements, character strings and signed integers: the conditional assembly language is oriented towards those items.

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, global (shared) and local variables, expressions and operators, assignments, conditional and unconditional branches, built-in functions, simple forms of I/O, and internal and external subroutines.

The elements manipulated and controlled by the conditional assembly language include statements in the ordinary assembly language, so we sometimes refer to the conditional language as the "outer" language, in which the ordinary or "inner" or "base" language is enclosed.



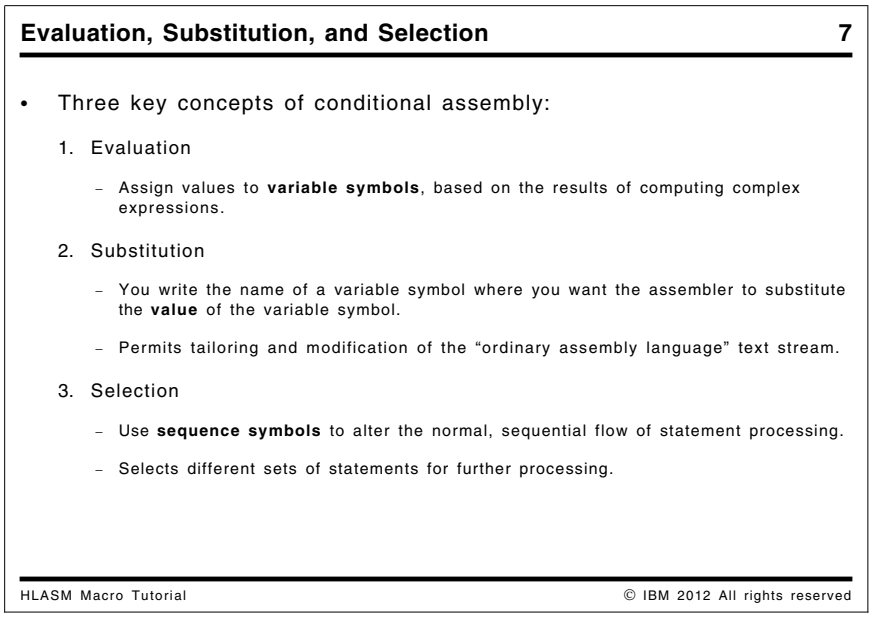
The conditional assembly language is used primarily in macro instructions (or "macros"), which may be thought of as "assembly-time subroutines" invoked during the assembly to perform useful functions. Most of the same techniques can also be used in ordinary assemblies ("open code", the primary input stream) without relying on macros.

Conditional assembly techniques are similar to those employed in some preprocessors for higher level languages such as C and PL/I, where compilers can interpret a special class of statements to perform substitutions, inclusion or exclusion of code fragments, and string replacement. As we will see, the assembler's support of these capabilities is more powerful: not only is the conditional assembly language complete, but – importantly – it provides extensive interactions with both the "ordinary" or "base" language and the external assembly environment.

A distinctive feature of the conditional assembly language is the introduction of two classes of symbols not used in the base language:

- **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. Figure 138 on page 263 compares the elements of the ordinary and conditional assembly languages.



Evaluation, Substitution, and Selection

There are three key concepts in the conditional assembly language:

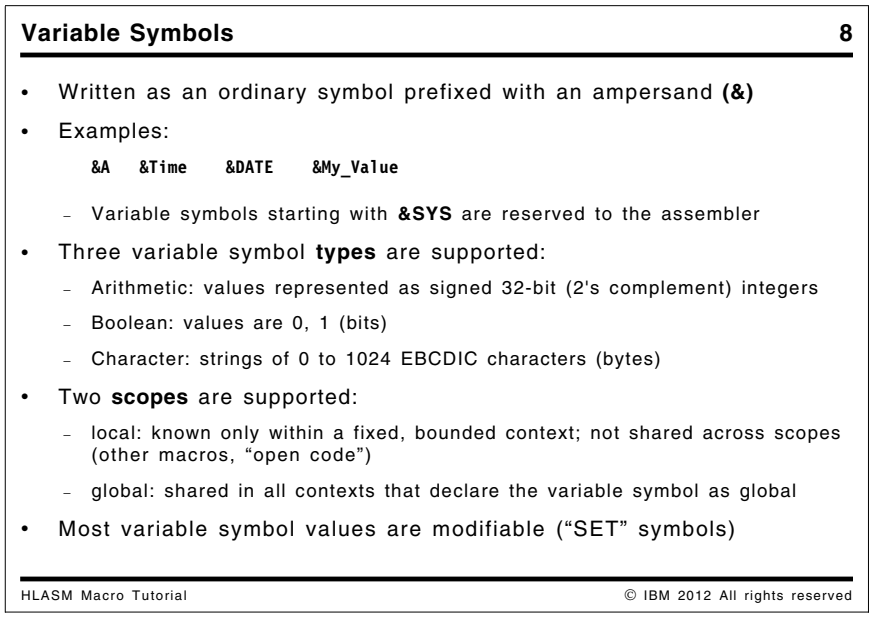
- evaluation
- substitution
- selection

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 *tailoring* and *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.

First, we examine variable symbols.



Variable Symbols

In addition to the familiar internal and external “ordinary” symbols managed by the assembler are the **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 like ordinary symbols, but with an 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 called *System Variable Symbols*, and are reserved to the assembler. They are described more fully in “System (&SYS) Variable Symbols” on page 249.

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 between -2^{31} and $+2^{31}-1$. (Be aware that in certain contexts, their values may be substituted as *unsigned* integers! This is discussed further at “Evaluating and Assigning Character Expressions: SETC” on page 19.)

Boolean
The allowed values of a Boolean variable symbol are 0 and 1, representing False and True respectively.

character
The value of a character variable symbol may contain from 0 to 1024 bytes, each being any EBCDIC character. (The bytes in a character variable symbol need not be characters, but usually are.) A character variable symbol containing no characters is sometimes called a *null string*.

The conditional assembly language supports two *scopes* for symbols: local and global.

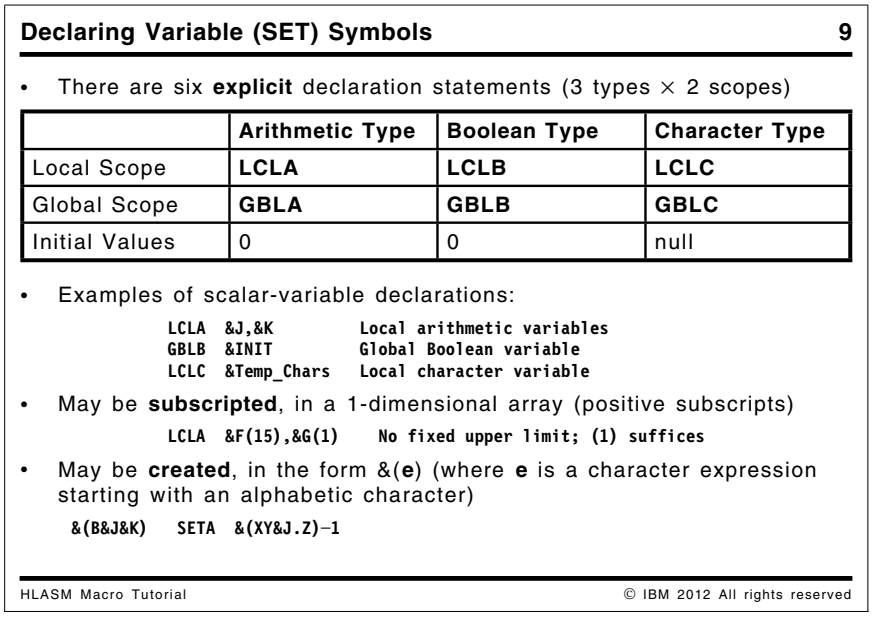
local
Local variable symbols have a limited, bounded scope, and are not known outside that scope. There are two types of local scope: within macros, and in "open code". "Open code" is the main sequence of Assembler Language statements read by the assembler, outside any macro invocations; it may contain a mixture of ordinary (base) language and conditional assembly statements.

global
Global variable symbols are shared *by name and type* by all scopes that declare them to be global. Thus, they can be shared between macros and open code. All declarations of global variables must have the same type, and be uniformly declared as either scalars or arrays.

It helps to distinguish two types of variable symbol:

1. Symbols whose values you can change: these are called SET symbols, because you use a "SET" statement to assign their values. We'll see a brief overview of how SET symbols work at "Assigning Values to Variable Symbols: SET Statements" on page 12.
2. Symbols whose values you can use, but not change: these are system variable symbols and symbolic parameters.

Each of these will be discussed in detail.


Declaring Variable (SET) Symbols 9

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

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

- Examples of scalar-variable declarations:

```
LCLA  &J,&K         Local arithmetic variables
GBLB  &INIT         Global Boolean variable
LCLC  &Temp_Chars   Local character variable
```

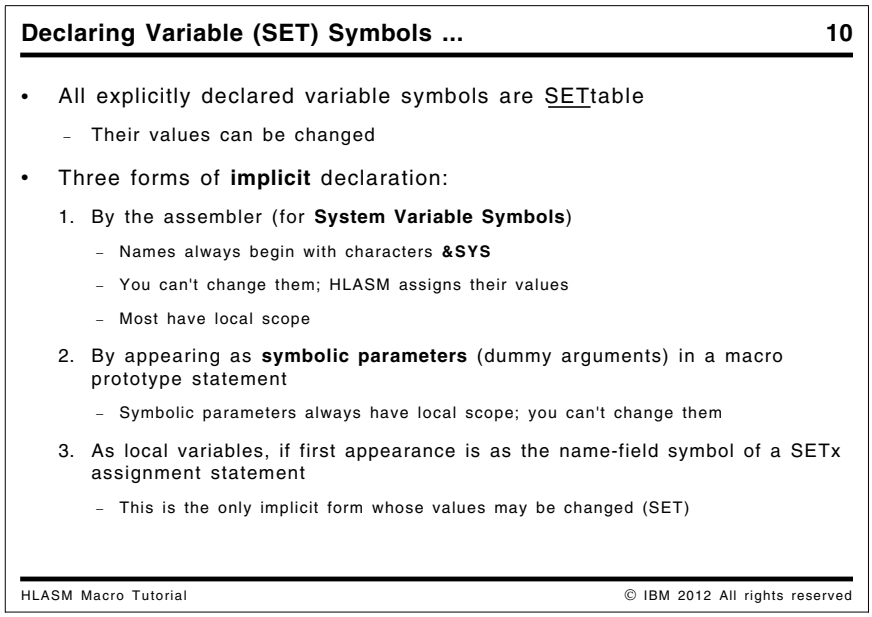
- May be **subscripted**, in a 1-dimensional array (positive subscripts)

```
LCLA  &F(15),&G(1)    No fixed upper limit; (1) suffices
```

- May be **created**, in the form &(**e**) (where **e** is a character expression starting with an alphabetic character)

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

HLASM Macro Tutorial



Declaring Variable (SET) Symbols

Variable symbols are declared in several ways:

1. Explicitly, through the use of declaration statements (global variable symbols must always be declared explicitly); all explicitly declared symbols are SET symbols, so their values may be changed.
2. Implicitly by the assembler (the *System Variable Symbols*, which may not be declared explicitly).
3. Implicitly, by their appearance as dummy arguments in a macro prototype statement (these are known as *symbolic parameters*; they always have character type, and are local in scope).
4. Implicitly as local variables, by being encountered for the first time in the name field of a SET statement as the recipient of an assignment. Their values may then be modified in other SET statements.

System variable symbols provide access to information the assembler "knows" about the state of the assembly and its environment. The symbols and examples of their use are given in "System (&SYS) Variable Symbols" on page 249; we will use some of them in later examples.

To explicitly declare variable symbols, two sets of statements specify their type and scope:

	Arithmetic Type	Boolean Type	Character Type
Local scope	LCLA	LCLB	LCLC
Global scope	GBLA	GBLB	GBLC
Initial Values	0	0	null

Figure 1. Explicit Variable Symbol Declarations and Initial Values

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 89. For now, 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.

Subscripted Variable Symbols

Variable symbols may also be *dimensioned* or *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 each having 15 elements. In practice, 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 actually *used* for a subscripted variable symbol with a Number attribute reference (to be discussed later, at "Macro-Instruction Argument Properties: Lists and Number Attributes" on page 83). Undimensioned (scalar) variable symbols have number attribute reference value zero to indicate they are not dimensioned.

Subscripts on variable symbols need not be assigned sequentially starting at 1. For example, you could assign values to &F(1) and &F(98765431); but be careful, because the assembler will allocate space for all the intervening elements, so you will likely use up all available storage!

Subscripted variable symbols may appear anywhere a scalar (unsubscripted) variable symbol appears.

Created Variable Symbols

You can create variable symbols "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 and result in a valid variable symbol name that is *not* the name of a macro parameter or a system variable symbol. Created variable symbols may also be subscripted; like other variable symbols they may be declared explicitly or implicitly.

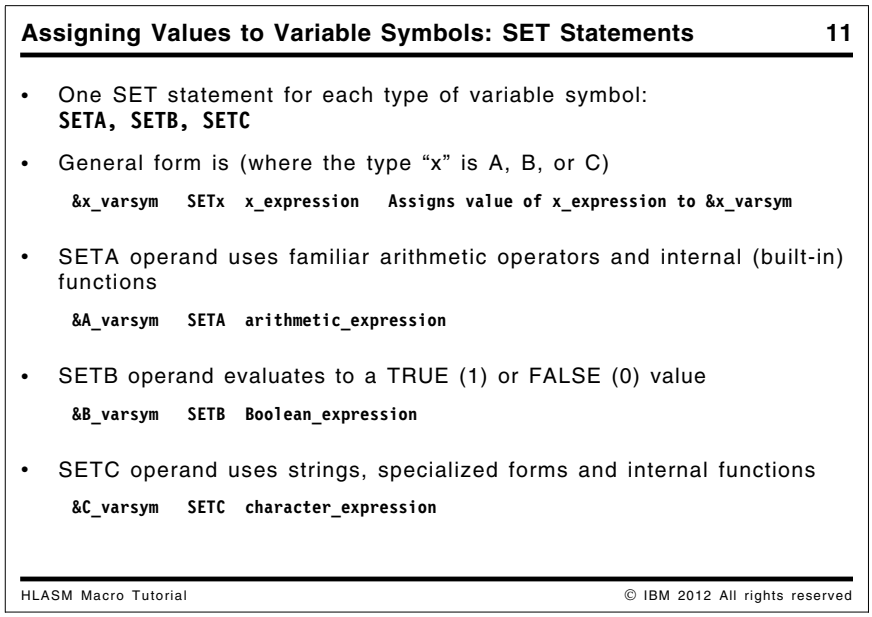
Created variable symbols may be formed from other created or uncreated variable symbols, to many levels. For example (using some SET statements to be discussed shortly):

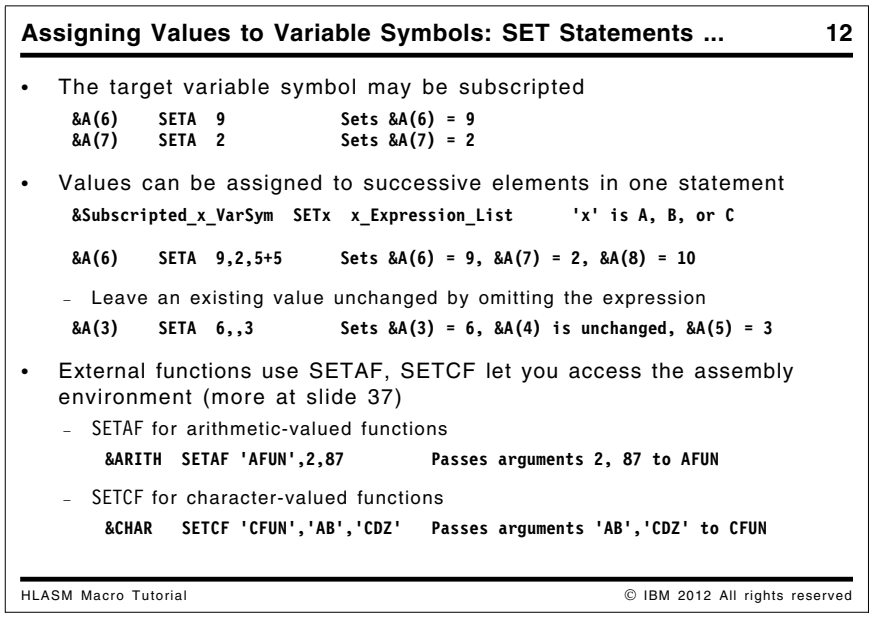
```
&C     SetC  'X'        Variable &C contains the character X
&BX    SetC  'PQ'       Variable &BX contains the characters PQ
.* &(A&(B&C))    is the same as &APQ

&APQ   SetA  42         Variable &APQ contains the integer 42
```

Then, the variable symbol &(A&(B&C)) is the same as the variable &APQ: first B&C is evaluated to form BX; then &(BX) is evaluated to form PQ; then A&(BX) is evaluated to form APQ; and finally &(APQ) is evaluated to form the symbol &APQ.

This form of "associative addressing" is very powerful, and we will use it in several case studies.

In the examples that follow, we will enclose character 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.





Assigning Values to Variable Symbols: SET Statements

New values are assigned to variable symbols in three ways, corresponding to the three 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.)

  Details of the three SET statements will be given shortly.

- 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.

For now, we will discuss only assignments to *declared* variable symbols.

External arithmetic-valued and character-valued functions are invoked by the SETAF and SETCF instructions respectively, and are described at "External Conditional-Assembly Functions" on page 40.

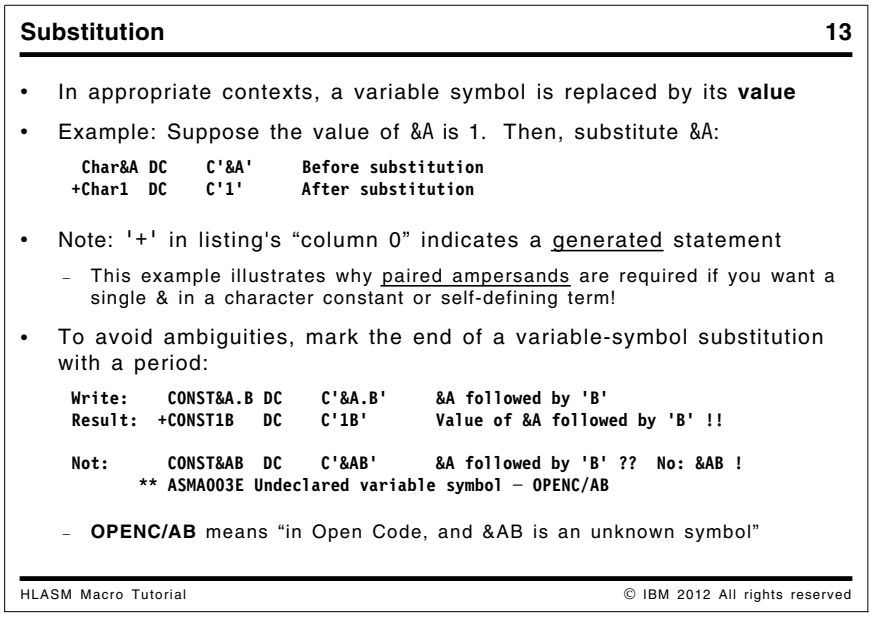
Multiple array elements may have values assigned in a single SET statement: specify a list of operand-field expressions of the proper type, separated by commas. For example,

```
&A(6)    SETA  9,2,5+5       Sets &A(6) = 9, &A(7) = 2, &A(8) = 10
&C(3)    SETC  '','A','2'    Sets &C(3) = '', &C(4) = 'A', &C(5) = '2'
```

assigns 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:

```
&A(3)    SETA  6,,3          Sets &A(3) = 6, &A(4) unchanged, &A(5) = 3
```

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.



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:

```
   Char&A DC    C'&A'
```

then the resulting statement would appear as

```
  +Char1  DC    C'1'
```

where the '+' character to the left of "column 1" is the assembler's indication in the listing that the statement was generated internally, and was not part of the original source program. (Such statements may be suppressed in the listing by specifying a PRINT NOGEN statement.)

At each appearance of the variable symbol &A, *its value is substituted in place of the symbol*. This behavior explains why you must write a pair of ampersands in character constants and self-defining terms where you want 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 appears in that position. The results of a substitution are almost always straightforward, but there are a few special cases we will discuss shortly.

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' ??
     ** ASMA003E Undeclared variable symbol - OPENC/AB
```

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! As you can see, it chose to recognize &AB, which is undefined to the assembler, as indicated in the diagnostic: the **OPENC/** indicator means "in Open Code", and **AB** is the unknown symbol.

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

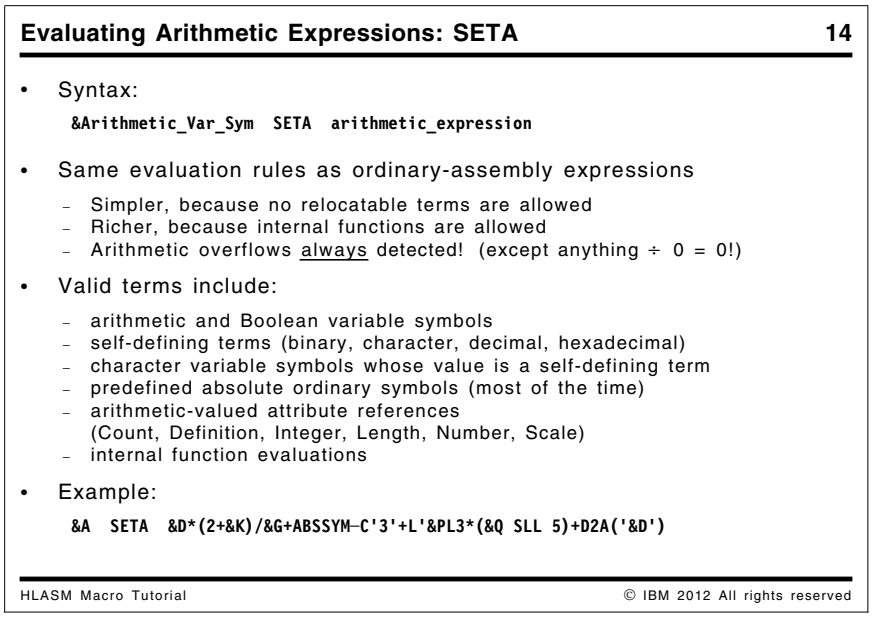
```
   CONST&A.B DC    C'&A.B'     &A followed by 'B'
```

giving

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

Variable symbols are not substituted in remarks fields or in comments statements. (We'll see that the AINSERT statement can help.)

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



Evaluating Arithmetic Expressions: SETA

As in any programming language, the assembler evaluates expressions involving variable symbols and other terms, and assigns the results to variable symbols.

The syntax of arithmetic and Boolean expressions is similar to that of common higher-level languages, but the syntax of character expressions is apparently unique to the Assembler Language.

The rules for evaluating conditional-assembly arithmetic expressions are very similar to those for ordinary expressions, with the added simplification that none of the terms in a conditional-assembly expression may be relocatable. The unary operators are + and −, which may precede any term. The binary operators are * and /, which must be preceded and followed by a term (itself possibly preceded by a unary operator). In addition to self-defining terms, predefined absolute ordinary symbols may usually be used as terms, as may evaluations of "internal functions" and 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.

Overflows are detected and diagnosed in almost all cases:

- addition and subtraction overflow is diagnosed and returns 0
- multiplication overflow is diagnosed and returns 1
- division overflow (−2147483648/−1) is diagnosed and returns 0
- division by zero (including 0÷0) returns 0 with *no* diagnostic!

```
&A    SetA  2*750        Value of &A is 1500
&B    SetA  3+7/2        Value of &B is 6
&C    SetA  (3+7)/2      Value of &C is 5
&D    SetA  0000005      Value of &D is 5
&E    SetA  5*6/3        Value of &E is 10
&F    SetA  5/3*6        Value of &D is 6  Note order of evaluation!
```

Arithmetic-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 arithmetic-valued attribute references are:

- Count (K')
- Defined (D')
- Integer (I')

- Length (L')
- Number (N')
- Scale (S')

A simple example of an attribute reference:

```
&A    SetA  K'&SYSVER       Count of characters in &SYSVER
```

We will illustrate applications of attribute references later, particularly when we discuss macros in "Macro Argument Attributes and Structures" on page 79. Attribute references may, of course, be used in "open code".

Operands of SETA statements may contain complex expressions. For example, we might execute the following statement:

```
&A  SETA  &D*(2+&K)/&G+ABSSYM-C'3'+L'&PL3*(&Q SLL 5)+D2A('&D')
```

The value assigned to &A is evaluated as follows:

1. multiply the value of &D by the value of (2+&K)
2. divide the result by the value of &G
3. to that result, add the value of the symbol ABSSYM and subtract the value of the character self-defining term C'3'
4. evaluate the product of the length attribute of the symbol that is the value of &PL3 and the value of &Q shifted left logically 5 bit positions; then, add this result to the result from the previous step.
5. Add the value of the function D2A('&D').

Internal functions will almost always be evaluated more rapidly than an equivalent but more complex expression. For example, suppose you must "extract" the value of bit 16 (having numeric weight 2^{15}) from the arithmetic variable &A. Previously, you might have written

```
&Bit16   SetA  (&A/16384)-(&A/32768)*2
```

which involves four arithmetic operations. Using shifting and masking, the same result can be obtained by writing

```
&Bit16   SetA  ((&A SRL 15) AND 1)
```

involving only two operations.

SETA Statements vs. EQU Statements 15

- Differences between SETA and EQU statements:

SETA Statements	EQU Statements
Active only at conditional assembly time	Active at ordinary assembly time; predefined absolute values usable at conditional assembly time
May assign values to a given *variable* symbol *many* times	A value is assigned to a given *ordinary* symbol *only once*
Expressions yield a 32-bit binary signed absolute value	Expressions may yield absolute, simply relocatable, or complexly relocatable values
No base-language attributes are assigned to variable symbols	Attribute values (length, type, assembler, program) may be assigned with an EQU statement

HLASM Macro Tutorial

SETA Statements vs. EQU Statements

It may help to identify some of the differences between the simpler results of SETA statements and the possibly more complicated results of EQU statements. The following table compares some key factors:

SETA Statements	EQU Statements
Active only at conditional assembly time	Active at ordinary assembly time; predefined absolute values usable at conditional assembly time
May assign values to a given *variable* symbol *many* times	A value is assigned to a given *ordinary* symbol *only once*
Expressions yield a 32-bit binary signed absolute value	Expressions may yield absolute, simply relocatable, or complexly relocatable values
No base-language attributes are assigned to variable symbols	Attribute values (length, type, assembler, program) may be assigned with an EQU statement

Figure 2. Differences between SETA and EQU Statements

Some earlier assemblers used ordinary symbols for both types of functions: conditional assembly and ordinary assembly. While this can be made to work in simple situations, the rules become much more complex and limiting when "interesting" things are tried.

Further comparisons of ordinary and conditional assembly are shown in "Comparing Ordinary and Conditional Assembly" on page 263.

High Level Assembler supports a very useful interaction between the "worlds" of ordinary and variable symbols: if an ordinary symbol is assigned an absolute value in an EQU statement prior to any reference in a conditional assembly expression, that "predefined absolute ordinary symbol" may be used wherever an arithmetic term is allowed.

Evaluating and Assigning Boolean Expressions: SETB **16**

- Syntax:

```
&Boolean_Var_Sym  SETB  (Boolean_expression)
```

- Boolean constants: 0 (false), 1 (true)
- Boolean operators:
  - **NOT** (highest priority), **AND, OR, XOR** (lowest)

```
&A     SetB  (&V gt 0 AND &V le 7)        &A=1 if &V between 1 and 7
&B     SetB  ('&C' lt '0' OR '&C' gt '9') &B=1 if &C is a decimal character
&S     SetB  (&B XOR (&G OR &D))
&T     SetB  (&X ge 5 XOR (&Y*2 lt &X OR &D))
```

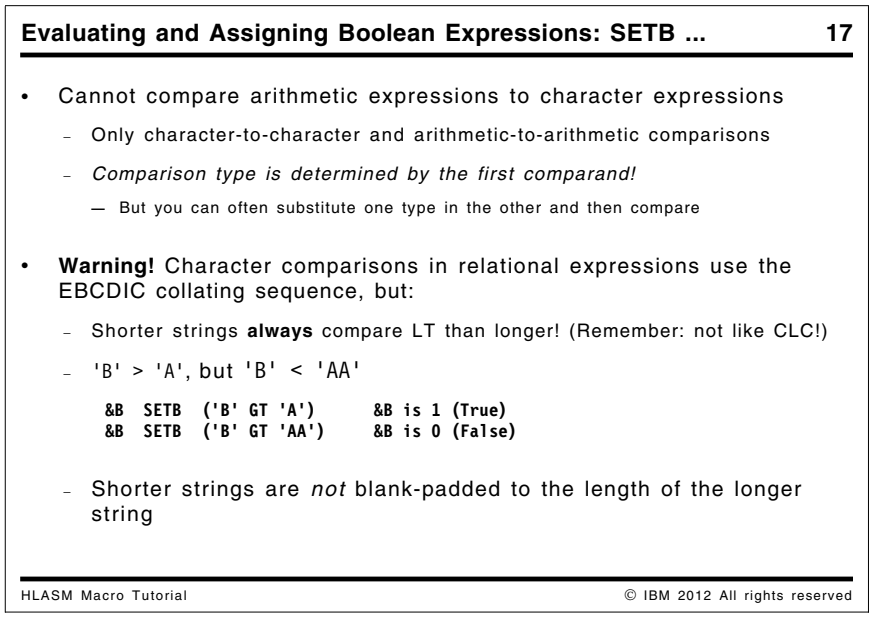
  - Unary **NOT** also allowed in **AND NOT, OR NOT, XOR NOT**

```
&Z     SetB  (&A AND NOT &B)
```

- Relational operators (for arithmetic and character comparisons):
  - **EQ, NE, GT, GE, LT, LE**

```
&A  SETB  (&N LE 2)          Arithmetic comparison
&B  SETB  ('&CVAR' NE '*')   Character comparison
&C  SETB  ((&A GT 10) AND NOT ('&X' GE 'Z') OR &R) Both
```

HLASM Macro Tutorial



Evaluating and Assigning Boolean Expressions: SETB

Boolean expressions provide much of the selection capability of the conditional assembly language. In practice, many Boolean expressions are not assigned to Boolean variable symbols; 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. Boolean constants may also be assigned from self-defining terms, previously defined absolute symbols, and SETA variables, in the forms

```
&Bool_Var  SetB  (self-defining term)
&Bool_Var  SetB  (previously defined absolute symbol)
&Bool_Var  SetB  (SETA variable)
```

The value assigned to the variable &Bool_Var is 0 if the value of the operand is zero, and is 1 otherwise.

The set of Boolean connectives includes the OR, AND, XOR, and NOT operators. For example, you can write statements such as

```
&A     SetB  (&V gt 0 AND &V le 7)          &A=1 if &V between 1 and 7
&B     SetB  ('&C' lt '0' OR '&C' gt '9')   &B=1 if &C is a decimal character
&Z     SetB  (&A AND NOT &B)
&S     SetB  (&B XOR (&G OR &D))
&T     SetB  (&X ge 5 XOR (&Y*2 lt &X OR &D))
```

The Boolean operators are the usual logical operators NOT, AND, OR, and XOR. For example:

```
&B  SETB  ((&A GT 10) AND NOT ('&X' GE 'Z') OR &R)
```

NOT is 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; OR the next; and XOR has lowest priority. Thus, the expression

```
(&A AND &B OR NOT &C XOR &D)
```

is evaluated as

```
((&A AND &B) OR ((NOT &C))) XOR &D
```

where the nesting depth of the parentheses indicates the priority of evaluation.

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 19 below). Comparisons between arithmetic and character expressions is not allowed, but you can often substitute one type in a variable of the other type and then compare.

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 shorter string is not padded, nor is the comparison done using the shorter string's length.) For example:

```
&B  SETB  ('B' GT 'A')    &B is 1 (True)
&B  SETB  ('B' GT 'AA')   &B is 0 (False), 'B' is shorter than 'AA'.
```

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 ('BB' GT 'AAA')
```

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
```

It is important to remember that the type of comparison is determined by the first comparand. These are invalid comparisons:

```
(5 LT '8')           invalid arithmetic comparison
('8' GT 5)           invalid character comparison
```

Evaluating and Assigning Character Expressions: SETC 18

- Syntax:

```
&Character_Var_Sym  SETC  character_expression
```

- A character constant is a 'quoted string' 0 to 1024 characters long

```
&CVar1   SETC  'AaBbCcDdEeFf'
&CVar2   SETC  'This is the Beginning of the End'
&Decimal SETC  '0123456789'
&Hex     SETC  '0123456789ABCDEF'
&Empty   SETC  ''      Null (zero-length) string
```

- Strings may be preceded by a parenthesized duplication factor

```
&X    SETC  (3)'ST'         &X has value STSTST
&J    SETA  2
&Y    SETC  (2*&J)'*'       &Y has value ****
```

- Strings are quoted; type-attribute and opcode-attribute references, and internal functions are not

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

  - Type/opcode attribute references: no duplication, quoting, or combining

HLASM Macro Tutorial

Evaluating and Assigning Character Expressions: SETC ... 19

- Apostrophes and ampersands in ordinary-assembly strings must be paired; but...
  - Apostrophes **are** paired internally for assignments and relationals!

```
&QT   SetC  ''''        Value of &QT is a single apostrophe
&Yes  SetB  ('&QT' eq '''')     &Yes is TRUE
```

  - Ampersands **are not** paired internally for assignments and relationals!

```
&Amp  SetC  '&&'        &Amp has value &&
&Yes  SetB  ('&Amp' eq '&&')  &Yes is TRUE
&D    SetC  (2)'A&&B'   &D has value A&&BA&&B
```

  - Use the BYTE function (slide 27) or substring notation (slide 21) to create a single &
- Warning! SETA variable values are substituted **without** sign!

```
&A  SETA  -5
    DC    F'&A'  Generates X'00000005'
&C  SETC  '&A'   &C has value 5 (not -5!)
```

  - The SIGNED and A2D functions avoid this problem

```
&C SETC (SIGNED &A) or &C SETC A2D(&A)   &C has value '-5'
```

HLASM Macro Tutorial

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'
&Decimal SETC  '0123456789'
&Hex     SETC  '0123456789ABCDEF'
&Empty   SETC  ''      Null (zero-length) string
```

Repeated sets of characters may be written very easily using a parenthesized integer expression preceding a string as a duplication factor:

```
&X    SETC  (3)'ST'         &X has value STSTST
&J    SETA  2
&Y    SETC  (2*&J)'*'       &Y has value ****
```

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. (Note that predefined absolute symbols may be used in character expressions only in contexts where an arithmetic term is allowed.)

Type attribute and opcode 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      Type attribute
&OCVar1  SETC  O'&CVar1      Opcode attribute
```

The opcode attribute will not be discussed further here.

When the assembler determines the *value* of a character-string term in a SETC expression, there is one key difference from character terms and constants, where *both* ampersands and apostrophes are paired. In *character-string* constants apostrophes *are* are paired but two ampersands are *not* paired to yield a single internal ampersand! 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
&Yes  SetB  ('&QT' eq BYTE(X'7D'))    &Yes is 1 (TRUE)

&Amp  SETC  '&&'        &Amp has value &&
&Yes  SetB  ('&Amp' eq '&&')  &Yes is 1 (TRUE)

&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 use the BYTE or A2C functions, or a substring (described below) of a pair of ampersands.

One reason for this behavior is that it avoids 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 make the language more awkward. The existing rules represent a trade-off between inconvenience and inconsistency, in favor of greater convenience.

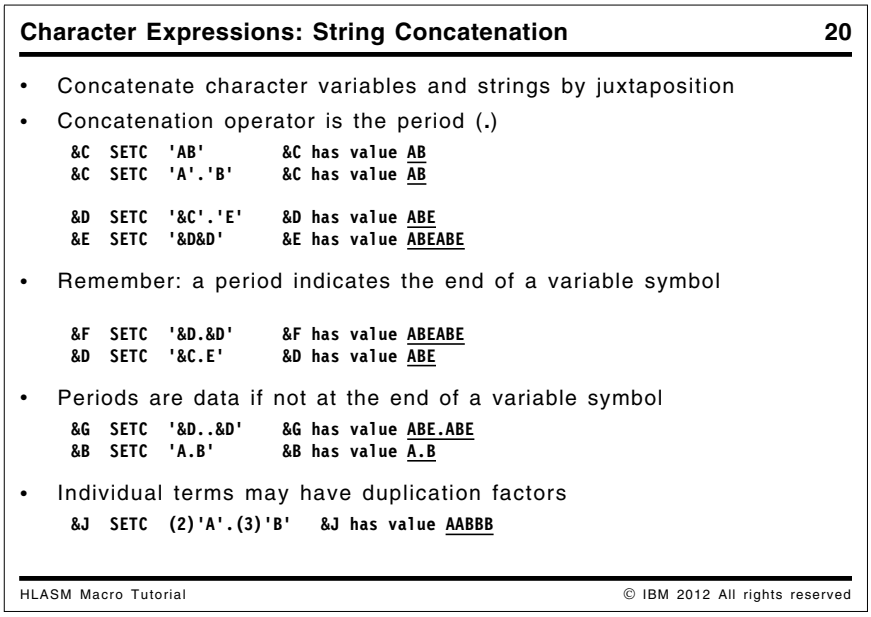
Be aware that substitution of arithmetic-valued variable symbols into character (SETC) expressions does *not* preserve the sign of the arithmetic value! For example:

```
&A  SETA  –5
    DC    F'&A'  Generates X'00000005'
&C  SETC  '&A'   &C has value '5' (not '-5'!)

&B  SETA  X'80000000'     (maximum negative number)
&D  SETC  '&B'   &D has value '2147483648' (!)
&E  SETA  &D     Error: too large! (“not a self-defining term”)
```

If signed arithmetic is important, use arithmetic expressions and variable symbols. If signed values must be substituted into character variables or ordinary statements, then use the SIGNED func-

tion (see “Character-Valued Functions Using Logical-Expression Format” on page 29) or A2D function (see “Converting from Arithmetic Data to Character Value Types” on page 32).

Character expressions support two useful capabilities: *string concatenation*, and *substring operations*.



Character Expressions: String Concatenation

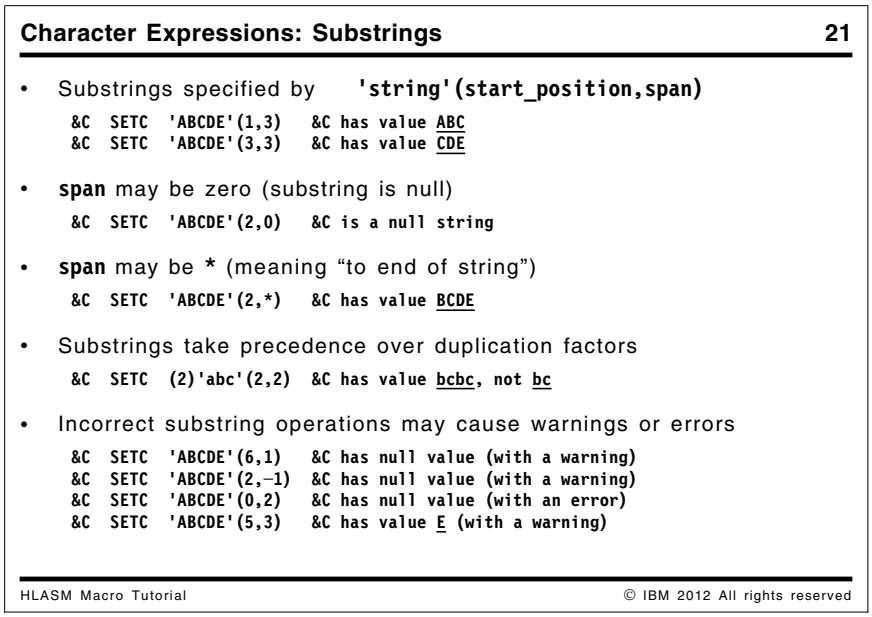
We are familiar with the notion of string concatenation from the earlier examples of substitution, where a substituted value is concatenated with the adjoining characters to create the completed string of characters. As before, the end of a variable symbol may be denoted with a period. The period is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

```
&C  SETC  'AB'        &C has value AB
&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'      &E has value ABEABE
&F  SETC  '&D.&D'     &F has value ABEABE

&G  SETC  '&D..&D'    &G has value ABE.ABE
&B  SETC  'A.B'       &B has value A.B
```

Each term in a concatenation may have a duplication factor:

```
&J  SETC  (2)'A'.(3)'B'   &J has value AABBB
```

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



Character Expressions: Substrings

Substrings are defined by a somewhat unusual notation:

```
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 (with 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 1024.
2. The *span* of the *substring* must be between 0 and 1024.
3. If $1 \le start\_position \le length$, and $1 \le span \le length$, and $start\_position + span \le length+1$, then the substring is completely contained in the source string, and a normal substring will be extracted.
4. 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.
5. If $span=0$, then the substring will be set to null. No error message will be issued.
6. If $start\_position \le 0$, then the assembler will issue an error message, and the substring will be set to null.
7. If $start\_position > length$, then the assembler will issue a warning message, and the substring will be set to null.
8. If $span<0$, then the assembler will issue a warning message, and the substring will be set to null.

The assembler provides a simple substring notation meaning “from here to the end of the string”: simply write the second operand of a substring specification as an asterisk. For example:

```
&C  SETC  'ABCDE'(2,*)   &C has value BCDE
```

will select the substring starting at the second character of 'ABCDE' through the last character, setting &C to 'BCDE'.

Substrings take precedence over duplication factors, as shown in the following example:

```
&C  SETC  (2)'abc'(2,2)  &C has value bcbc, not bc
```

The duplication factor repeats the substring 'bc' twice, rather than first creating the string 'abcabc' and taking the two characters starting at position 2.

String expressions are constructed using the operations of substitution, concatenation, and sub-stringing. You can also use type and opcode attribute references as character terms, but they are limited to “single-term” expressions with no duplication factors.

Substring operations apply to the string term they follow, and not to string expressions involving concatenation or character-valued internal functions (discussed in “Character-Valued Functions Using Logical-Expression Format” on page 29). For example:

```
&A    SetC  'abcde'
&B    SetC  'qrstu'
&C    SetC  '&A.&B'(4,4)       &C contains deqr
&D    SetC  '&A.'&B'(4,4)      &D contains abcdetu
```

Note: Don't confuse substring notation with subscripted variable symbols: for substrings, the parenthesized *start_position* and *span* appear following the quoted string:

```
&SubStr  SetC  'string'(start_position,span)
```

whereas subscripts appear inside the quotes:

```
&StrVal  SetC  '&ArrayVar(&Subscript)'
```

They may of course appear together, to extract a substring of a subscripted character variable symbol:

```
&StrVal  SetC  '&ArrayVar(&Subscript)'(start_position,span)
```

Character Expressions: String Lengths 22

- Use a Count Attribute Reference (**K'**) to determine the number of characters in a variable symbol's value

```
&N    SETA  K'&C       Sets &N 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  '''&&'''   &C has value '&&'
&N    SETA  K'&C       &N has value 4

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

- Arithmetic and Boolean variables converted to strings first

```
&A    SETA  -999       K'&A has value 3    (Remember: no sign!)
```

HLASM Macro Tutorial

Character Expressions: String Lengths

The number of characters in a character variable symbol's value 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  '''&&'''    &C has value '&&'
&N    SETA  K'&C        &N has value 4

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

Note that the pairing rules for apostrophes and ampersands apply only to character strings, not to the contents of SETC variables:

```
&C    SETC  '''&&'''    &C has value '&&'
&D    SETC  '&C'        &D still has value '&&'
&M    SETA  K'&D        &M has value 4
```

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 of &C to &X.

The value of a count attribute reference applied to an arithmetic or Boolean variable symbol is determined by first converting the value of the symbol to a character string (remember that arithmetic values are converted without sign!). The length of the resulting string is the attribute's value. For example, if &A has value −999, its count attribute is 3.

```
&A    SETA  -999        K'&A has value 3
```

Conditional Expressions with Mixed Operand Types 23

- Expressions sometimes simplified with mixed operand types
  - Some limitations on substituted values and converted results
- Let &A, &B, &C be arithmetic, Boolean, character:

Variable Type	SETA Statement	SETB Statement	SETC Statement
Arithmetic	no conversion	zero &A becomes 0; nonzero &A becomes 1	'&A' is decimal representation of magnitude(&A)
Boolean	extend &B to 32-bit 0 or 1	no conversion	'&B' is '0' or '1'
Character	&C must be a self-defining term	&C must be a self-defining term; convert to 0 or 1 as above	no conversion

HLASM Macro Tutorial

Conditional Expressions with Mixed Operand Types

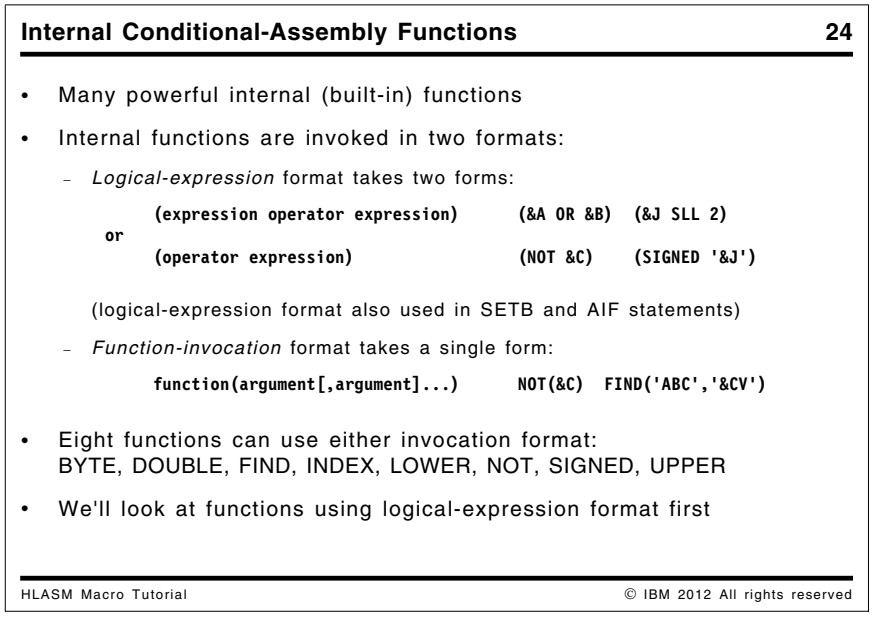
Conditional assembly expressions are often simpler if mixed operand types are used, avoiding a need for additional statements for converting to the desired type. Figure 3 indicates the allowed combinations of SETx statement types and operands; the variables &A, &B, and &C respectively represent arithmetic, Boolean, and character variable symbols.

Variable Type	SETA Statement	SETB Statement	SETC Statement
Arithmetic	no conversion	zero &A becomes 0; nonzero &A becomes 1	'&A' is decimal representation of magnitude of &A
Boolean	extend &B to 32-bit 0 or 1	no conversion	'&B' is '0' or '1'
Character	&C must represent a self-defining term	&C must represent a self-defining term; converted to 0 or 1 as for arithmetic variables	no conversion

Figure 3. Conditional Assembly SET Statement Operand Types

In most cases, the result of a substitution is as expected. However, there are a few cases to note:

- Arithmetic values substituted into Boolean expressions are converted using a simple rule: zero values are converted to 0, and nonzero values are converted to 1.
- Arithmetic values substituted into character expressions are converted to their *unsigned* decimal representation. This means that the *magnitude* of the arithmetic term is converted to decimal!
- Character values substituted into arithmetic expressions must be self-defining decimal, hexadecimal, binary, or character self-defining terms.
- Character values substituted into Boolean expressions must be self-defining decimal, hexadecimal, binary, or character self-defining terms, which are then converted to 0 or 1 following the first case above.



Internal Conditional-Assembly Functions

Internal, built-in functions can save effort for common operations such as type conversion, testing operands, etc. They are invoked in either of two formats:

1. Logical-expression format, which may take either of two forms:

```
     (expression operator expression)      (&A OR &B)  (&J SLL 2)
or
     (operator expression)                 (NOT &C)    (SIGNED '&J')
```

   The logical-expression format is also used in SETB and AIF statements, where the result of the expression is a Boolean (SETB) value.[1]

2. Function-invocation format takes a single form:

```
     function(argument[,argument]...)      FIND('&S1','&S2')
```

   where the square brackets indicate that you may specify an optional argument, and the ellipsis (...) indicates that additional optional arguments are allowed.

Built-in functions return either arithmetic (SETA) or character (SETC) results, and their arguments may be arithmetic expressions or character expressions.

There are no Boolean-valued internal functions that return bit values.

The internal functions are:

- Arithmetic-valued functions with arithmetic arguments:
  - Logical operations: AND, OR, XOR, NOT
  - Shift operations: SLA, SLL, SRA, SRL
- Arithmetic-valued functions with character arguments:
  - Representation-conversion functions: B2A, C2A, D2A, X2A
  - String-scanning functions: DCLEN, FIND, INDEX
  - Validity-checking functions: ISBIN, ISDEC, ISHEX, ISSYM
- Character-valued functions with arithmetic arguments:
  - Representation-conversion functions: A2B, A2C, A2D, A2X, BYTE, SIGNED
- Character-valued functions with character arguments:
  - Representation-conversion functions involving decimal characters: B2D, C2D, X2D, D2B, D2C, D2X
  - Representation-conversion functions for other forms: B2C, B2X, C2B, C2X, X2B, X2C
  - String-manipulation functions: DCVAL, DEQUOTE, DOUBLE, LOWER, UPPER
  - Attribute retrieval functions: SYSATTRA, SYSATTRP

[1] One nice thing about logical-expression format is that spaces can be used within the parentheses to make statement formatting more readable.

Functions Using Logical-Expression Format

These internal functions use logical-expression format, as shown in Tables 4 and 5.

- For arithmetic operations: AND, OR, NOT, XOR, SLA, SLL, SRA, SRL
- For character operations: UPPER, LOWER, DOUBLE, INDEX, FIND, BYTE, SIGNED.

The functions SLA, SLL, SRA, SRL, AND, OR, and XOR are available *only* in logical-expression format.

Arithmetic Arguments	Character Arguments
NOT, SLA, SLL, SRA, SRL, AND, OR, XOR	FIND, INDEX
Note: NOT has a single argument; the others have two arguments	

Figure 4. Arithmetic-Valued Functions Using Logical-Expression Format

Arithmetic Arguments	Character Arguments
BYTE, SIGNED	LOWER, UPPER, DOUBLE
Note: These functions have a single argument	

Figure 5. Character-Valued Functions Using Logical-Expression Format

Arithmetic-Valued Functions, Logical-Expression Format **25**

- Logical-expression format:

```
(operand  operator  operand)   or   (operator  operand)
```

- Masking/logical operations: AND, OR, NOT, XOR

```
((&A1 AND &A2) AND X'FF')
(&A1 OR (&A2 OR &A3))
(&A1 XOR (&A3 XOR 7))
(NOT &A1)+&A2
(7 XOR (7 OR (&A+7)))    Round &A to next multiple of 8
```

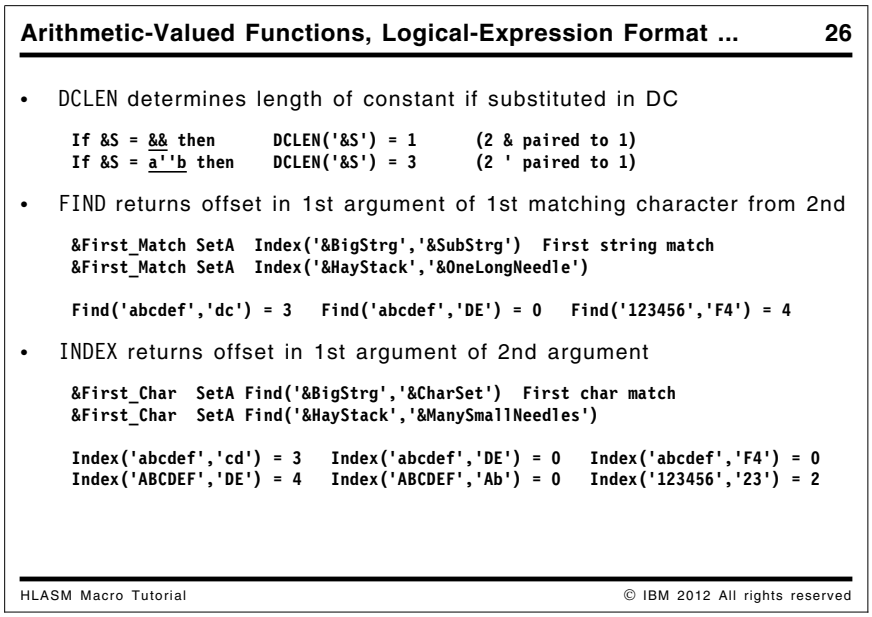
- Shifting operations: SLL, SRL, SLA, SRA

```
(&A1 SLL 3)          Shift left 3 bits, unsigned
(&A1 SRL &A2)        Shift right &A2 bits, unsigned
(&A1 SLA 1)          Shift left 1 bit, signed (can overflow!)
(&A1 SRA &A2)        Shift right &A2 bits, signed
```

- Any combination...

```
(3+(NOT &A) SLL &B)/((&C-1 OR 31)*5)
```

HLASM Macro Tutorial



Arithmetic-Valued Functions Using Logical-Expression Format

The eight arithmetic functions are in two groups: logical (or masking) operations, and shifting operations. The logical/masking operations include AND, OR, NOT, and XOR. For example:

```
((&A1 AND &A2) AND X'FF')   Low-order 8 bits of &A1 AND &A2
(&A1 OR (&A2 OR &A3))       OR of 3 variables
(&A1 XOR (&A3 XOR 7))       XOR of 7 and variables &A1 and &A3
(NOT &A1)+&A2               Complement and add
```

These logical operations act on the 32-bit binary values of arithmetic operands, in exactly the same way as the corresponding machine instructions N, O, and X. The NOT operator produces the bit-wise or "ones' complement" of its operand, which has the same effect as XORing the operand with a word of all one-bits (−1).

To "round up" the value of &A to a multiple of 8 (if it is not already a multiple) you might write:

```
((&A+7)/8)*8                Round &A to next multiple of 8
```

Using the masking operations OR and XOR, you can write instead:

```
(&A+7 AND -8)               Round &A to next multiple of 8
```

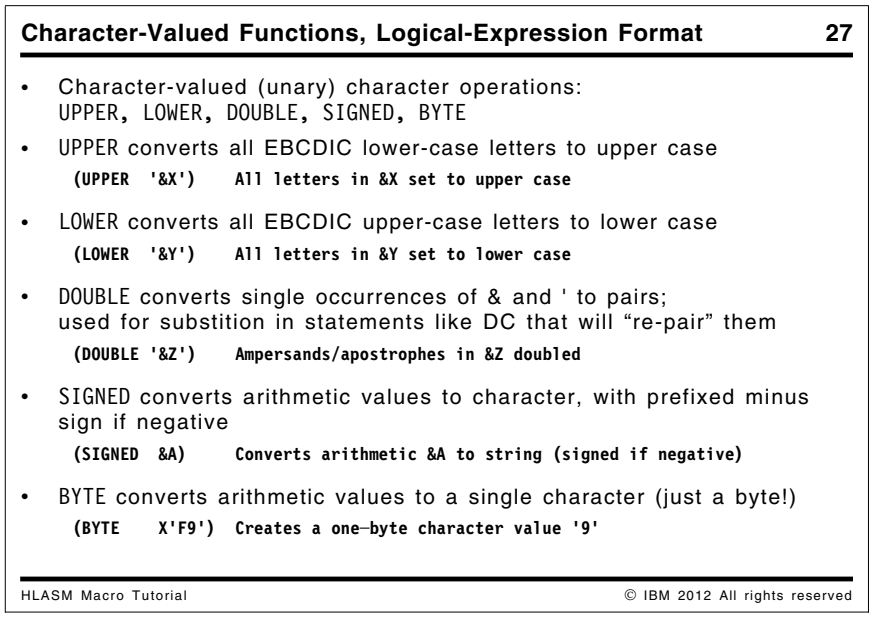
The shifting operators for arithmetic operands correspond to the shift instructions provided by the processor instructions: left or right, and arithmetic (signed) or logical (unsigned).

```
(&A1 SLL 3)                 Shift left 3 bits, unsigned
(&A1 SRL &A2)               Shift right &A2 bits, unsigned
(&A1 SLA 1)                 Shift left 1 bit, signed
(&A1 SRA &A2)               Shift right &A2 bits, signed
```

These operators may be used in any combination:

```
(3+(NOT &A) SLL &B)/((&C-1 OR 31)*5)
```

Just as with the SLA instruction, the SLA function can generate a fixed-point overflow.



Character-Valued Functions Using Logical-Expression Format

The assembler supports five character-valued character functions invoked using logical-expression format: UPPER, LOWER, DOUBLE, SIGNED, and BYTE. Each is a unary operator.

- The UPPER function operates on a string of EBCDIC characters and produces a string in which all lower-case letters are converted to their upper-case equivalents.

```
(Upper '&X')   All letters in &X set to upper case
```

- The LOWER function does the inverse of the UPPER function, converting all upper-case letters to lower case.

```
(Lower '&Y')   All letters in &Y set to lower case
```

- The DOUBLE function replaces each occurrence of an ampersand and apostrophe (single quote) with a pair. This allows the result to be directly substituted into a character self-defining term or into DC-statement character constant (or literal). For example, if the value of &Z is &&', then

```
(DOUBLE '&Z')  Ampersands/apostrophes in &Z doubled (giving &&&&'')
```

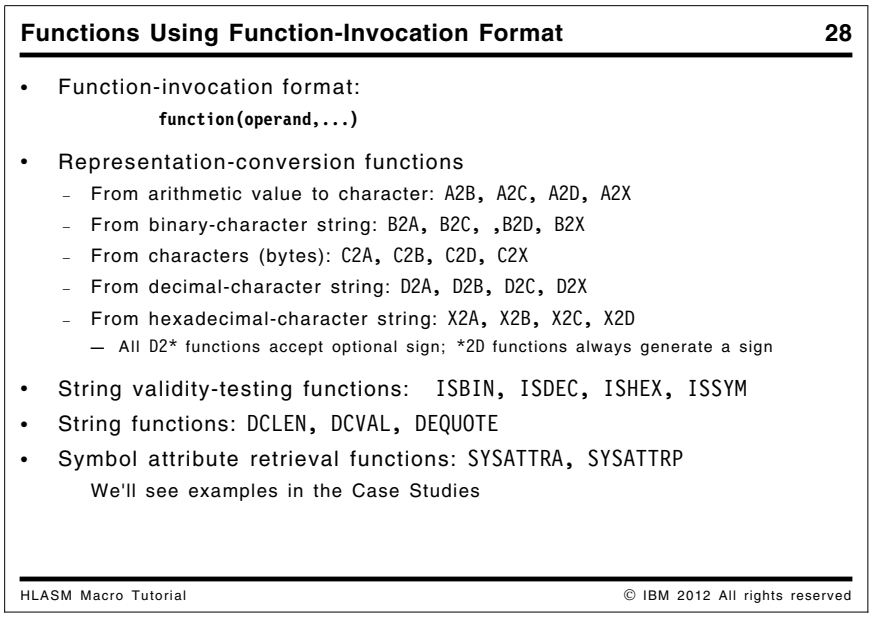
  DOUBLE is very useful when you issue conditional-assembly messages.

- The SIGNED function creates a character-string representation of arithmetic values, with a prefixed minus sign if the arithmetic value is negative. (Assigning an arithmetic variable in a SetC statement to a character variable produces only the *unsigned magnitude* of the arithmetic value!)

```
(SIGNED  10)   has value  '10'
(SIGNED -10)   has value '-10'
```

- The BYTE function allows you to assign any pattern of eight bits to a character variable containing a single byte.

```
(BYTE 0)       returns a character with bit pattern X'00'
(BYTE 64)      returns a character with bit pattern X'40'
```

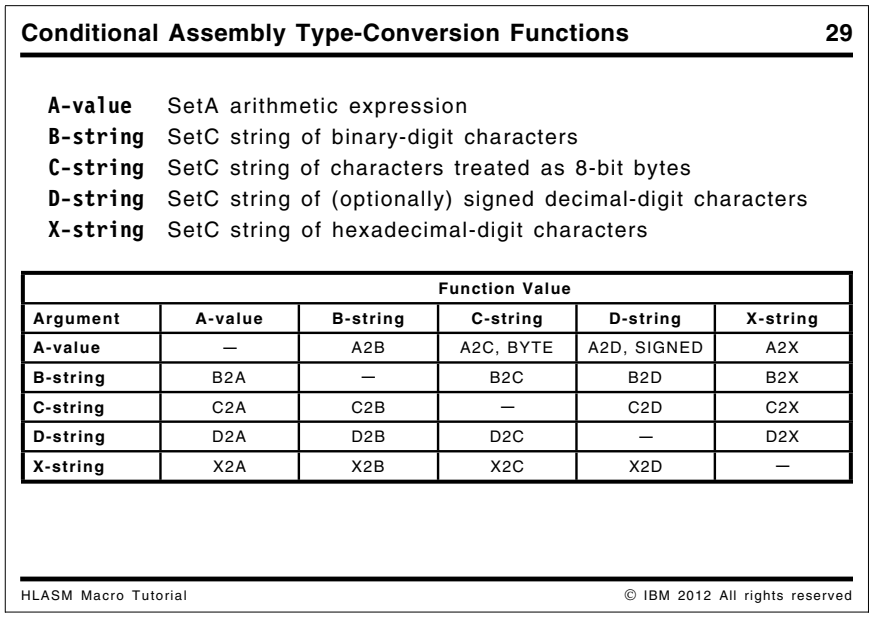


Functions Using Function-Invocation Format

Functions using function-invocation format provide a range of capabilities.

- Some functions convert data from one representation to another, as shown in Figure 6 on page 31. There are three basic conditional-assembly data types: arithmetic, Boolean, and character. In most situations, Boolean data is treated as a special case of arithmetic data. Character data can be considered to have four value types:
  - bit strings, containing only the characters 0 and 1
  - byte or character strings, where each byte may contain any bit pattern
  - decimal strings representing (optionally signed) decimal numbers
  - hexadecimal strings, where each character is a hexadecimal digit.
- The functions ISBIN, ISDEC, ISHEX, and ISSYM test strings for validity, to verify that they contain only binary, decimal, or hexadecimal characters, or that they form a valid Assembler Language symbol.
- DCLEN, DCVAL, and DEQUOTE respectively determine the length of a string after pairs of ampersands and apostrophes have been reduced to single occurrences and return the string after such pairing (as though it had been substituted in a DC statement), and remove leading or trailing quotes from a string.
- SYSATTRA and SYSATTRP return the values of the Assembler and Program attributes of an ordinary symbol.

These functions are described below.



Internal Type-Conversion Functions

Each of the four value types of character data, and arithmetic data, may be converted among one another using the representation conversion functions. The notation used to describe the arguments is:

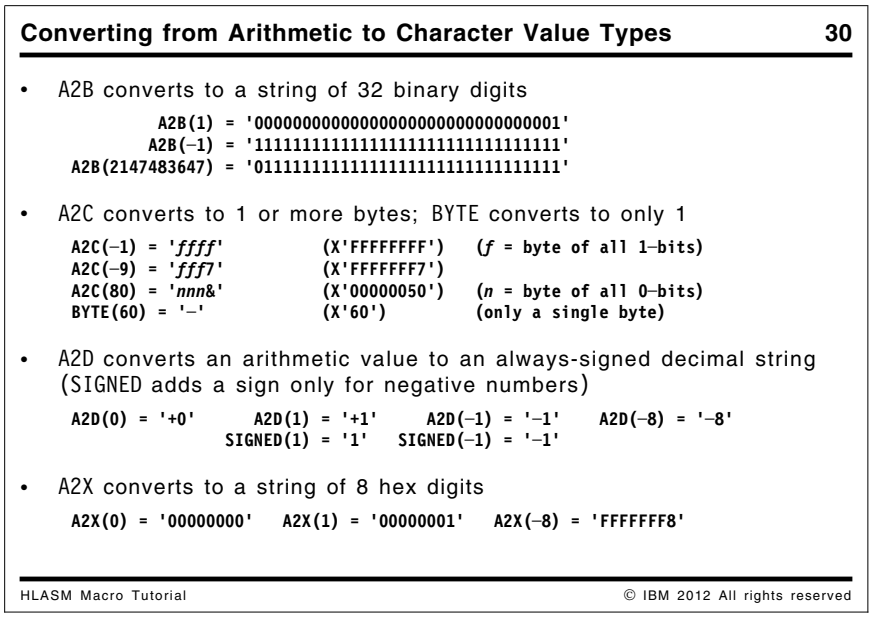
`A-value` arithmetic expression; a SETA value such as `-5`
`B-string` string of binary digits (characters `'0'` and `'1'`) such as `'1010'`
`C-string` string of characters (bytes) such as `'aB?7&Y'`
`D-string` string of (optionally) signed decimal digits such as `'+42'`, `'42'`, and `'-42'`
`X-string` string of hexadecimal digits (characters `'0'` through `'F'`) such as `'c0ffee'`

	Function Value				
Argument	**A-value**	**B-string**	**C-string**	**D-string**	**X-string**
A-value	—	A2B	A2C, BYTE	A2D, SIGNED	A2X
B-string	B2A	—	B2C	B2D	B2X
C-string	C2A	C2B	—	C2D	C2X
D-string	D2A	D2B	D2C	—	D2X
X-string	X2A	X2B	X2C	X2D	—

Figure 6. Representation-Conversion Functions

Some examples of these representation-conversion functions are:

```
C2A('0000')      has arithmetic value -252645136  (-252645136 is an A-value)
B2X('11')        has character value '3'          ('11'  is a  B-string)
C2B('a')         has character value '10000001'   ('a'   is a  C-string)
D2X('+25')       has character value '19'         ('+25' is a  D-string)
X2B('E')         has character value '1110'       ('E'   is an X-string)
```



Converting from Arithmetic Data to Character Value Types

Arithmetic (SETA) values are converted to the four other character value types: binary, bytes ("characters"), signed decimal strings, and hexadecimal by four corresponding functions.

- A2B converts an arithmetic value to a string of 32 '0' and '1' characters representing the 32 bits of its argument:

```
A2B(1) = '00000000000000000000000000000001'
A2B(-1) = '11111111111111111111111111111111'
A2B(-8) = '11111111111111111111111111111000'
A2B(99) = '00000000000000000000000001100011'
A2B(16777217) = '00000001000000000000000000000001'
A2B(2147483647) = '01111111111111111111111111111111'
A2B(-2147483648) = '10000000000000000000000000000000'
```

- A2C returns the binary value of its argument as a string of four bytes (the four bytes of the binary value); BYTE converts to only 1 byte.

```
A2C(0) = 'nnnn'  (X'00000000')
A2C(96) = 'nnn-'  (X'00000050')   (n = byte of all 0-bits)
BYTE(96) = '-'    (X'60')         (only a single byte)
A2C(-1) = 'ffff'  (X'FFFFFFFF')   (f = byte of all 1-bits)
A2C(-9) = 'fff7'  (X'FFFFFFF7')
A2C(2147483647) = '"fff'  (X'7FFFFFFF')
A2C(-385875968) = 'Znnn'  (X'E9000000')
A2C(125) = 'nnn''  (X'0000007D')
A2C(-235736076) = '1234'  (X'F1F2F3F4')
```

- A2D converts the binary value of its argument to an always- signed decimal string; SIGNED adds a sign only for negative values:

```
A2D(0) = '+0'              SIGNED(0) = '0'
A2D(1) = '+1'              SIGNED(1) = '1'
A2D(-1) = '-1'             SIGNED(-1) = '-1'
A2D(-8) = '-8'             A2D(99) = '+99'
A2D(16777217) = '+16777217'
A2D(2147483647) = '+2147483647'
A2D(-2147483648) = '-2147483648'
```

- A2X converts an arithmetic value to a string of 8 characters representing the hexadecimal digits of its argument:

```
          A2X(0) = '00000000'          A2X(1) = '00000001'
         A2X(-1) = 'FFFFFFFF'          A2X(9) = '00000009'
         A2X(-8) = 'FFFFFFF8'         A2X(99) = '00000063'
   A2X(16777217) = '01000001'
 A2X(2147483647) = '7FFFFFFF'
A2X(-2147483648) = '80000000'
```

Converting from Character Values to Arithmetic Values 31

- B2A converts from bit-strings to arithmetic:

```
B2A('1001') = 9   B2A('11111111111111111111111111111111') = -1
                  B2A('11111111111111111111111111111000') = -8
```

- C2A converts from bytes to arithmetic:

```
     C2A('-') = 96       (Argument is a single byte, X'60')
     C2A('a') = 129
  C2A('nnn2') = 242      (n = byte of all 0-bits)
```

- D2A converts from decimal strings to arithmetic: (+ signs optional)

```
D2A('') = 0   D2A('-001') = -1   D2A('+1') = 1   D2A('1') = 1
```

- X2A converts from hex-strings to arithmetic:

```
X2A('fffffff8') = -8   X2A('63') = 99   X2A('7fffffff') = 2147483647
```

HLASM Macro Tutorial

Converting from Character Values to Arithmetic Values

Four functions convert character value types to a binary arithmetic value.

- B2A converts strings of 0 to 32 '0' and '1' characters to the equivalent binary value. Strings shorter than 32 characters are padded on the left with zeros if necessary.

```
                                B2A('') = 0
                              B2A('001') = 1
                             B2A('1001') = 9
B2A('11111111111111111111111111111111') = -1
B2A('11111111111111111111111111111000') = -8
```

- C2A converts 0 to 4 characters (left-padded with "null" bytes of eight 0-bits if necessary) to a 4-byte binary value:

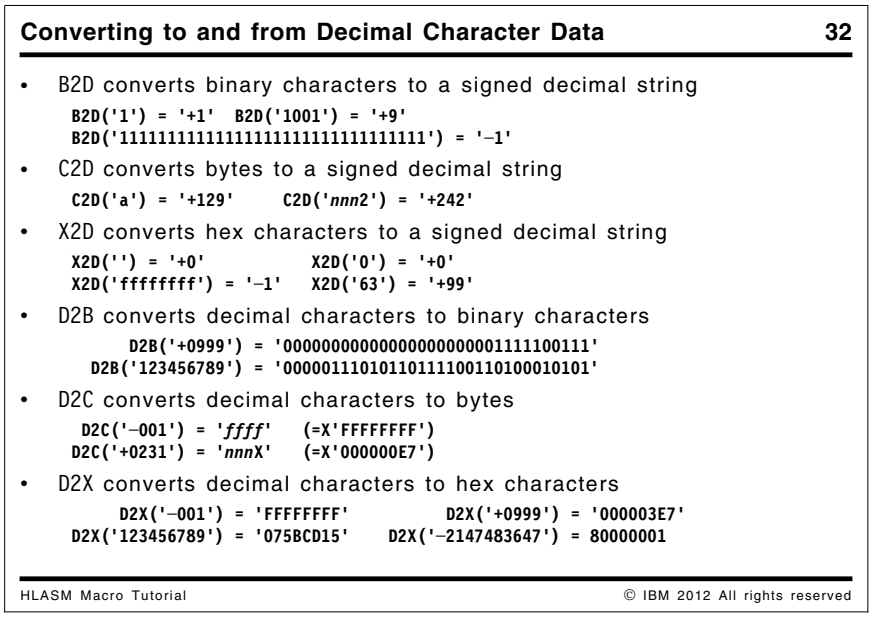
```
   C2A('') = 0
  C2A('-') = 96
  C2A('a') = 129
C2A('nnn2') = 242       (n = byte of all 0-bits)
```

- D2A converts a string of optionally signed decimal digits to binary:

```
          D2A('') = 0                 D2A('000') = 0
      D2A('-001') = -1              D2A('+0999') = 999
 D2A('2147483647') = 2147483647
D2A('-2147483648') = -2147483648
```

- X2A converts 0 to 8 characters (left-padded with zero characters if necessary) to the equivalent binary value:

```
       X2A('0') = 0                  X2A('1') = 1
X2A('ffffffff') = -1          X2A('fffffff8') = -8
      X2A('63') = 99          X2A('7fffffff') = 2147483647
```



Converting to and from Decimal Character Data

Six functions convert between strings of decimal characters and the other three character value types. Three functions convert *to* decimal strings: they always generate a signed result; and three functions converting *from* decimal strings accept optionally signed arguments.

These three functions convert character value types to signed decimal strings.

- B2D converts 0 to 32 '0' and '1' characters (left-padded with zero characters if necessary) to their equivalent signed decimal value:

```
                                  B2D('') = '+0'
                                 B2D('0') = '+0'
                                 B2D('1') = '+1'
B2D('11111111111111111111111111111111') = '-1'
                               B2D('1001') = '+9'
B2D('11111111111111111111111111111000') = '-8'
                             B2D('1100011') = '+99'
 B2D('11111111111111111111111111111111') = '+2147483647'
B2D('10000000000000000000000000000000') = '-2147483648'
```

- C2D converts 0 to 4 bytes (left-padded with "null" bytes of eight 0-bits if necessary) to a signed decimal value:

```
     C2D('') = '+0'                    C2D('-') = '+96'
    C2D('a') = '+129'                C2D('nnn2') = '+242'
C2D('   2') = '+1077952754'          C2D('aaaa') = '-2122219135'
 C2D('999') = '+16382457'
```

- X2D converts 0 to 8 hexadecimal characters (left-padded with zero characters if necessary) to a signed decimal value:

```
        X2D('') = '+0'                    X2D('0') = '+0'
       X2D('1') = '+1'                    X2D('9') = '+9'
      X2D('63') = '+99'            X2D('FFFFFFF8') = '-8'
X2D('7FFFFFFF') = '+2147483647'
X2D('80000000') = '-2147483648'
```

These three functions convert optionally signed decimal strings to character value types:

- D2B converts a string of decimal characters to a string of 32 '0' or '1' characters representing the binary value of the argument.

```
          D2B('0') = '00000000000000000000000000000000'
        D2B('000') = '00000000000000000000000000000000'
       D2B('-001') = '11111111111111111111111111111111'
      D2B('+0999') = '00000000000000000000001111100111'
  D2B('123456789') = '00000111010110111100110100010101'
 D2B('2147483647') = '01111111111111111111111111111111'
D2B('-2147483647') = '10000000000000000000000000000001'
D2B('-2147483648') = '10000000000000000000000000000000'
```

- D2C converts a string of decimal characters to a string of 4 bytes having the same binary value as the argument string:

```
          D2C('0') = 'nnnn' (=X'00000000')
       D2C('-001') = 'ffff' (=X'FFFFFFFF')
      D2C('+0231') = 'nnnX' (=X'000000E7')
D2C('2130706431') = '=fff' (=X'7EFFFFFF')
```

- D2X converts a string of decimal characters to 8 hexadecimal characters having the same binary representation as the argument string:

```
          D2X('0') = '00000000'          D2X('000') = '00000000'
       D2X('-001') = 'FFFFFFFF'        D2X('+0999') = '000003E7'
  D2X('123456789') = '075BCD15'   D2X('2147483647') = '7FFFFFFF'
D2X('-2147483647') = '80000001'  D2X('-2147483648') = '80000000'
```

Converting Among Binary, Bytes, and Hex Strings 33

- B2C converts binary digits to bytes

```
B2C('') = '' (null string)     B2C('10000001') = 'a' (X'81')
B2C('0000') = 'n' (X'00')  B2C('100000001011100') = ' *'    (X'405C')
```

- B2X converts binary digits to hex digits

```
B2X('') = ''   B2X('1000000') = '40'    B2X('001111101') = '07D'
B2X('0000000000001') = '0001'   B2X('1011100010111000101110001011100') = '5C5C5C'
```

- C2B converts bytes to binary digits

```
C2B('a') = '10000001'   C2B('****') = '01011100010111000101110001011100'
 C2B('') = ''           C2B(' *') = '0100000001011100'
```

- C2X converts bytes to hex characters

```
C2X('') = ''   C2X(' ') = '40'   C2X('a') = '81'   C2X('9999') = 'F9F9F9F9'
```

- X2B converts hex digits to binary digits

```
X2B('') = ''   X2B('7D') = '01111101'   X2B('040') = '000001000000'
```

- X2C converts hex digits to bytes

```
X2C('') = ''   X2C('81') = 'a'   X2C('405c') = ' *'   X2C('5c5c5c') = '***'
```

HLASM Macro Tutorial

Converting Among Bit, Byte, and Hexadecimal String Data

The following functions convert character value data representing binary, byte, and hexadecimal data. Because they do not involve data that represents (or has been converted to or from) SETA values, their arguments and results may be longer than conversion to a 32-bit arithmetic value would allow.

- B2C converts strings of '0' and '1' characters to bytes. If the argument string length is not a multiple of 8, it is padded on the left with zero characters to form an integral number of result bytes.

```
                                B2C('') = ''      (null string)
                              B2C('0000') = 'n'     (X'00')
                    B2C('0000000000000000') = 'nnn'   (X'000000')
                             B2C('1000000') = ' '     (X'40')
                            B2C('10000001') = 'a'     (X'81')
                          B2C('100000001011100') = ' *'    (X'405C')
        B2C('10111000101110001011100001011100') = '****'  (X'5C5C5C5C')
       B2C('11111001111110011111100111111001') = '9999'  (X'F9F9F9F9')
  B2C('10000001010100110101001101010100110101001') = ' zzzz' (X'40A9A9A9A9')
```

- B2X converts strings of '0' and '1' characters in groups of 4 to equivalent characters representing hexadecimal digits. If the argument string length is not a multiple of 4, it is padded on the left with zero characters.

```
                          B2X('') = ''        (null string)
                       B2X('1000000') = '40'
                     B2X('001111101') = '07D'
                    B2X('0010000001') = '081'
                  B2X('0000000000001') = '0001'
                 B2X('100000001011100') = '405C'
   B2X('10111000101110001011100001011100') = '5C5C5C5C'
  B2X('11111001111110011111100111111001') = 'F9F9F9F9'
```

- C2B converts each byte in the argument string to groups of eight '0' and '1' characters:

```
     C2B('') = ''                    C2B(' ') = '01000000'
    C2B('a') = '10000001'            C2B(''') = '01111101'
   C2B(' *') = '0100000001011100'
  C2B('****') = '01011100010111000101110001011100'
  C2B('9999') = '11111001111110011111100111111001'
 C2B(' zzzz') = '01000000101010011010100110101010011010100110101001'
    C2B('n') = '00000000'            C2B('n*') = '0000000001011100'
```

- C2X converts each byte in the argument string to 2 hexadecimal characters:

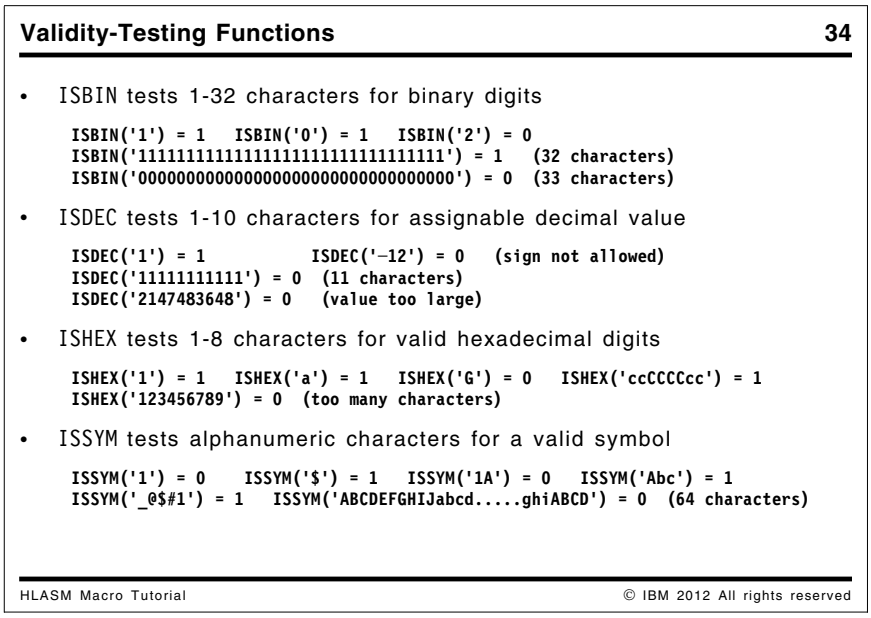
```
      C2X('') = ''                    C2X(' ') = '40'
     C2X('a') = '81'                 C2X(' *') = '405C'
    C2X('n*') = '005C'              C2X('9999') = 'F9F9F9F9'
 C2X('123456') = 'F1F2F3F4F5F6'
```

- X2B converts each hexadecimal character in the argument string to four '0' and '1' characters:

```
           X2B('') = ''
          X2B('7D') = '01111101'
         X2B('040') = '000001000000'
        X2B('0081') = '0000000010000001'
      X2B('000405C') = '00000000000001000000001011100'
     X2B('5C5C5C5C') = '01011100010111000101110001011100'
  X2B('F5F6F9F9F9F9') = '111101011111011011111001111110011111110011111101'
```

- X2C converts each pair of hexadecimal characters in the argument string to a byte having the equivalent value. If the argument string has an odd number of characters, it is padded on the left with a zero character.

```
          X2C('') = ''                  X2C('81') = 'a'
        X2C('040') = 'n '             X2C('405c') = ' *'
     X2C('000040') = 'nn '          X2C('5c5c5c5c') = '****'
  X2C('f9f9f9f9') = '9999'     X2C('f1f2f3f4f5f6') = '123456'
```



Validity-Testing Functions

Four functions help you determine the validity of strings whose contents might be used in other contexts. Each function returns a 0 (false) or 1 (true) value.

- ISBIN tests whether all the characters in the string are either '0' or '1'. At most 32 characters are allowed.

```
                              ISBIN('0') = 1
                              ISBIN('1') = 1
                              ISBIN('2') = 0  (not binary)
 ISBIN('11111111111111111111111111111111') = 1  (32 characters)
ISBIN('00000000000000000000000000000000') = 0  (33 characters)
```

- ISDEC tests whether all the characters in the string are decimal digits, and may validly be converted to 32-bit binary. Signs are not allowed; at most 10 characters are allowed.

```
       ISDEC('001') = 1           ISDEC('-12') = 0   (sign not allowed)
 ISDEC('1111111111') = 1    ISDEC('11111111111') = 0   (11 characters)
 ISDEC('2147483647') = 1     ISDEC('2147483648') = 0   (value too large)
```

- ISHEX tests whether all the characters in the string are valid hexadecimal digits. Mixed-case letters are allowed; the argument string may contain at most 8 characters.

```
       ISHEX('1') = 1             ISHEX('a') = 1
       ISHEX('G') = 0          ISHEX('12ef') = 1
 ISHEX('ccCCCCcc') = 1      ISHEX('123456789') = 0  (too many characters)
```

- ISSYM tests whether the characters in the string represent a valid Assembler Language symbol.

```
                              ISSYM('1') = 0         ISSYM('$') = 1
                              ISSYM('_') = 1        ISSYM('1A') = 0
                            ISSYM('Abc') = 1     ISSYM('_@$#1') = 1
                           ISSYM('aaaa') = 1
    ISSYM('abcdefghijklmnopqrstuvwxyz') = 1
 ISSYM('YYYYYYYYYYYYY.....YYYYYYYY') = 1 (63 characters)
ISSYM('ABCDEFGHIJabcd.....ghiABCD') = 0 (64 characters)
```

Character-Valued String Functions 35

- DCVAL pairs quotes and ampersands, and returns the string generated as if substituted in DC (compare to DCLEN, slide 36)

  `If &S = a''b&&c then DCVAL('&S') = a'b&c (single quote and ampersand)`

- DEQUOTE removes a single leading and/or trailing quote

  ```
  If &S = '' then DEQUOTE('&S') = null string (both quotes removed)
  If &S = '1F1B' then DEQUOTE('&S') = 1F1B (quotes removed at both ends)
  ```

- DOUBLE pairs quotes and ampersands (inverse of DCVAL)

  ```
  Let &A have value a'b and &B have value a&b; then

  (DOUBLE '&A') = DOUBLE('&A') = a''b   (two quotes)
  (DOUBLE '&B') = DOUBLE('&B') = a&&b   (two ampersands)
  ```

- LOWER converts all letters to lower case

  `(LOWER 'aBcDeF') = LOWER('aBcDeF') = 'abcdef'`

- UPPER converts all letters to upper case

  `(UPPER 'aBcDeF') = UPPER('aBcDeF') = 'ABCDEF'`

HLASM Macro Tutorial

Character-Valued String Functions

These functions help you manage character strings.

- DCVAL returns its argument string with all paired apostrophes and ampersands collapsed to single occurrences, as would be the case if the argument string were used as the nominal value of a C-type constant. (The function does *not* perform substitutions of variable symbols!)

  ```
  If &S = null  then     DCVAL('&S') = null string
  If &S = ''  then       DCVAL('&S') = '     (single quote)
  If &S = &&  then       DCVAL('&S') = &     (single ampersand)
  If &S = a''b  then     DCVAL('&S') = a'b   (single quote)
  If &S = a''b&&c  then  DCVAL('&S') = a'b&c (single quote and ampersand)
  ```

- DEQUOTE removes any single quotation marks (apostrophes) from the first and/or last characters of the argument string.

  ```
  If &S = null  then    DEQUOTE('&S') = null string
  If &S = '  then       DEQUOTE('&S') = '    (single quote)
  If &S = '1F  then     DEQUOTE('&S') = 1F   (leading quote removed)
  If &S = 1B'  then     DEQUOTE('&S') = 1B   (trailing quote removed)
  If &S = ''  then      DEQUOTE('&S') = null string   (both quotes removed)
  If &S = '1F1B'  then  DEQUOTE('&S') = 1F1B (quotes removed at both ends)
  ```

- DOUBLE pairs quotes and ampersands (the inverse of DCVAL)

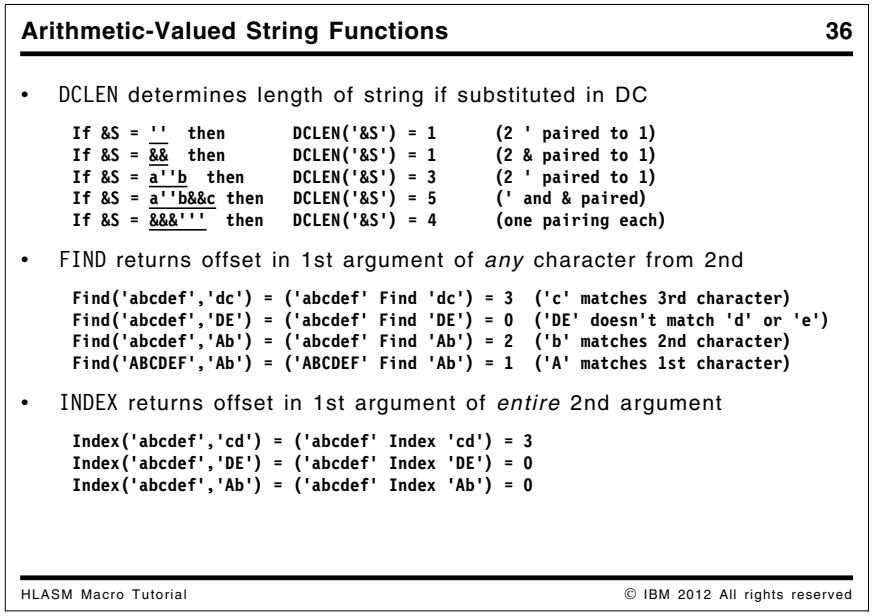
  ```
  DOUBLE(a'b) = a''b   (two quotes)
  DOUBLE(a&b) = a&&b   (two ampersands)
  ```

- LOWER converts all letters to lower case

  LOWER('aBcDeF') = 'abcdef'

- UPPER converts all letters to upper case

  UPPER('aBcDeF') = 'ABCDEF'



Arithmetic-Valued String Functions

- DCLEN examines a string for the presence of paired apostrophes and ampersands to determine how many characters would be generated if the string was used as the nominal value of a C-type constant. (The function does *not* perform substitutions of variable symbols, as a DC statement would!)

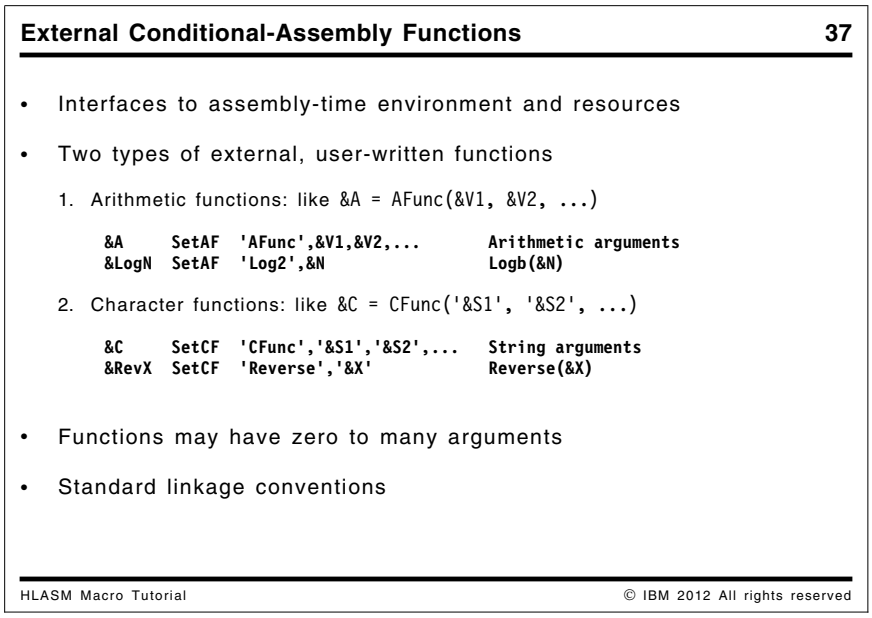
  If &S = '' then DCLEN('&S') = 1 (2 ' paired to 1)
  If &S = && then DCLEN('&S') = 1 (2 & paired to 1)
  If &S = a''b then DCLEN('&S') = 3 (2 ' paired to 1)
  If &S = a''b&&c then DCLEN('&S') = 5 (' and & paired)
  If &S = &&&''' then DCLEN('&S') = 4 (one pairing each)
  If &S = ''''' then DCLEN('&S') = 3 (5 quotes, only two pairings)

- FIND returns the offset in the first argument string of *any* character in the second argument string.

  Find('abcdef','dc') = ('abcdef' Find 'dc') = 3 ('c' matches 3rd character)
  Find('abcdef','DE') = ('abcdef' Find 'DE') = 0 ('DE' doesn't match 'd' or 'e')
  Find('abcdef','Ab') = ('abcdef' Find 'Ab') = 2 ('b' matches 2nd character)
  Find('ABCDEF','Ab') = ('ABCDEF' Find 'Ab') = 1 ('A' matches 1st character)
  Find('123456','F4') = ('123456' Find 'F4') = 4 ('4' matches 4th character)

- INDEX returns the offset in the first argument of the first occurrence of the *entire* second argument string.

```
Index('abcdef','cd') = ('abcdef' Index 'cd') = 3
Index('ABCDEF','cd') = ('ABCDEF' Index 'cd') = 0
Index('abcdef','DE') = ('abcdef' Index 'DE') = 0
Index('abcdef','Ab') = ('abcdef' Index 'Ab') = 0
Index('ABCDEF','DE') = 4
```



External Conditional-Assembly Functions

High Level Assembler for z/OS, z/VM, & z/VSE supports a powerful and flexible capability for invoking externally-defined functions during the assembly. These functions can perform any desired action, and provide easy access to the environment in which the assembler is operating. They are invoked using the SETAF and SETCF statements, by analogy with SETA and SETC.

The syntax of the statements is similar to that of SETA and SETC: a local or global variable symbol appears in the name field; it receives the value returned by the function. The operation mnemonic indicates the type of function to be called, and the type of value assigned to the name-field variable. The first operand in each case is a character expression giving the name of the function to be called. The remaining operands are optional, and their presence depends on the function: some functions could require no arguments, others could require several. The type of each argument is the same as that of the receiving "target" variable: arithmetic arguments for SETAF functions, and character arguments for SETCF functions.

A compact notational representation of this description is

```
&Arith_Var   SETAF  'Arith_function'[,arith_val]...
&Char_Var    SETCF  'Char_function'[,'character_val']...
```

For example, we might invoke the LOG2 and REVERSE functions with these two statements:

```
&LogN  SetAF  'Log2',&N              Logb(&N)
&RevX  SetCF  'Reverse','&X'         Reverse(&X)
```

Interface descriptions and code samples for these two functions are described in "External Conditional Assembly Functions" on page 234. Details of external function interfaces are described in the *High Level Assembler for z/OS, z/VM, & z/VSE Programmer's Guide*.

Statement Selection: Conditional Assembly Control Flow 38

- Lets the assembler select different sequences of statements for further processing
- Key elements are:
  1. Sequence symbols
     - Used to “mark” positions in the statement stream
     - A conditional assembly “label”
  2. Two statements that reference sequence symbols:

     AGO conditional-assembly “unconditional branch”

     AIF conditional-assembly “conditional branch”
  3. One statement that defines a sequence symbol:

     ANOP conditional-assembly “No-Operation”

HLASM Macro Tutorial

Statement Selection: Conditional Assembly Control Flow

Much of the power of the conditional assembly language lies in its ability to direct the assembler to *select* different sequences of statements for processing. 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. The ANOP statement provides a “place holder” for defining a sequence symbol.

Sequence Symbols 39

- Sequence symbol: an ordinary symbol preceded by a period (**.**)

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

- Used to **mark** a statement
  - **Defined** by appearing in the name field of a statement

```
.A     LR   R0,R9
.B     ANOP ,
```

  - **Referenced** as the target of AIF, AGO statements
- Not assigned any value (absolute, relocatable, or other)
- Purely local scope; no sharing of sequence symbols across scopes
- Cannot be created or substituted (unlike variable symbols)
  - Cannot even be created by substitution in a macro-generated macro (!)
    (AINSERT provides a way around this)
- Never passed as the value of any symbolic parameter

HLASM Macro Tutorial

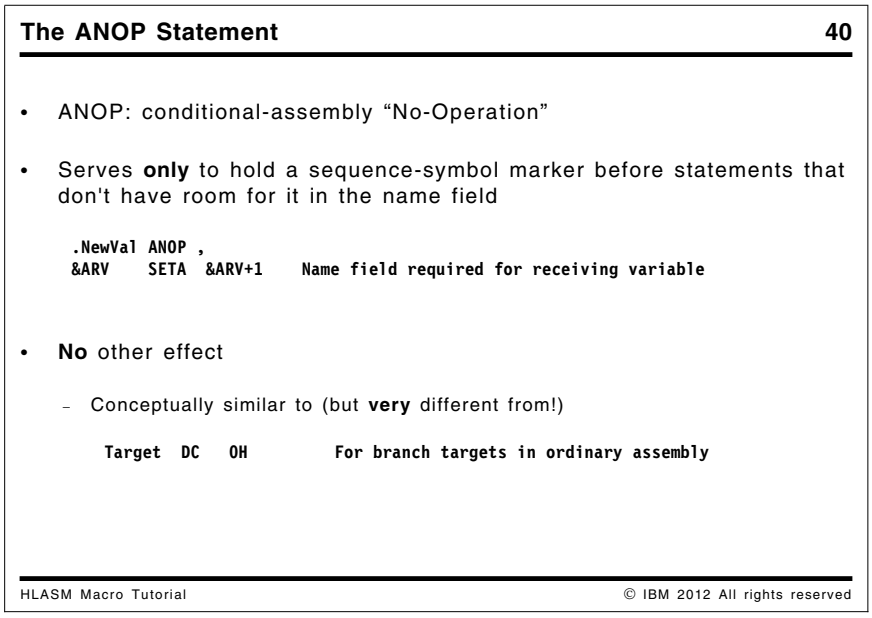
Sequence Symbols

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 any expression.
- 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 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.
- Sequence symbols may be defined before or after references to them. This means that both forward and backward "branches" are possible (implying possible endless loops).[2]



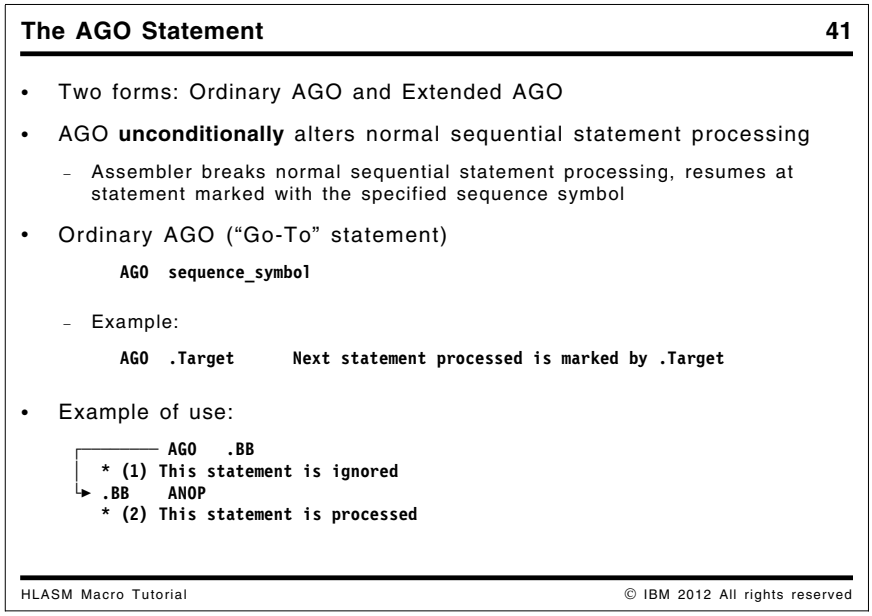
[2] The ability of conditional assembly branching to go "backward" to an earlier point in the statement stream means that great care must be taken when defining sequence symbols in COPY segments, because the same symbol might be defined in open code or in another COPYed instance of the same segment. Typically, the assembler will not be able to complete enough processing to create a listing with an error message.

ANOP Statement

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

```
.NewVal ANOP  ,         Mark the following statement
&ARV    SETA  &ARV+1    Name field required for target variable
```

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



The AGO Statement

The AGO statement unconditionally alters 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 is marked by .Target
```

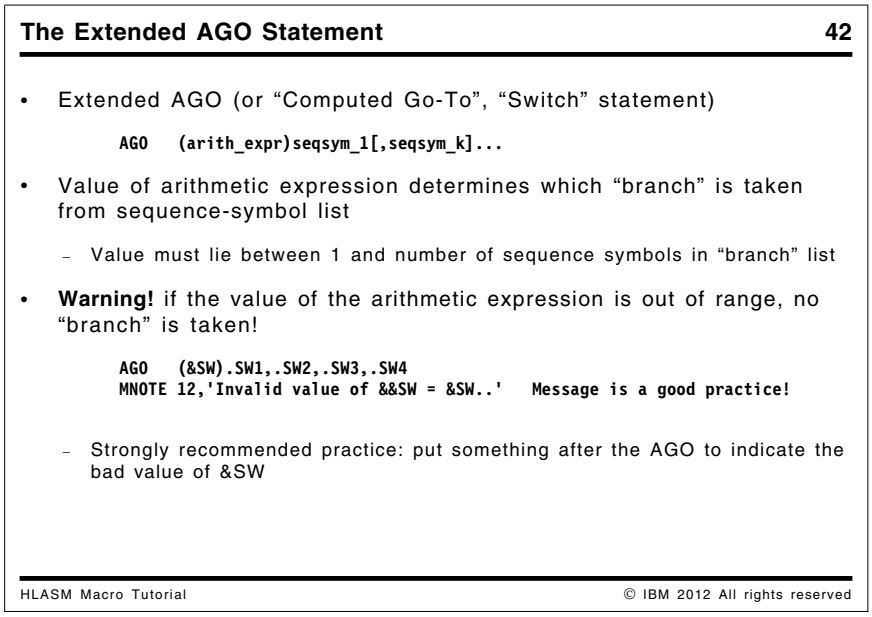
The assembler breaks its normal sequential statement processing, and resumes processing at the statement "marked" with the specified sequence symbol. For example,

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

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

forward/backward branch in the statement stream

The Extended AGO Statement



The assembler provides a convenient extension to the simple AGO statement, in the form of the “Extended AGO” statement. It is analogous to “switch” or “case” statements in other languages. The operand field contains a parenthesized arithmetic expression, followed by a list of sequence symbols, as shown in Figure 7.

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

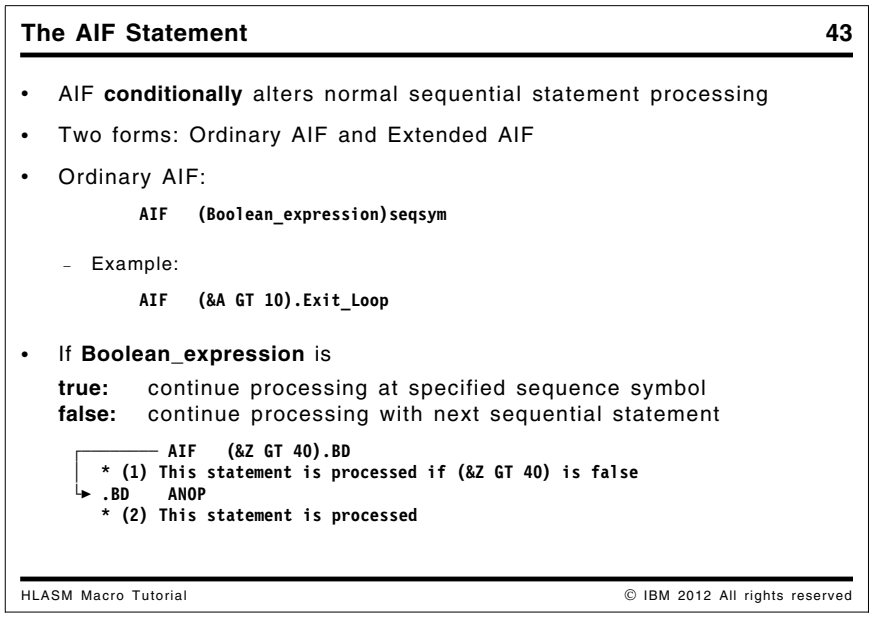
Figure 7. General Form of the Extended AGO Statement

The extended AGO statement tests the value of the *arithmetic_expression* 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. If 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.

A recommended technique to catch invalid values is:

```
AGO   (&SW).SW1,.SW2,.SW3,.SW4
MNOTE 12,'Invalid value of &&SW = &SW..'   Always a good practice!
```

where a message indication is placed after the AGO to trap cases where the arithmetic variable's value is invalid. (We'll see more about MNOTE on page 47.)



The AIF Statement

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

```
AIF  (Boolean_expression)seqsym
```

For 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:

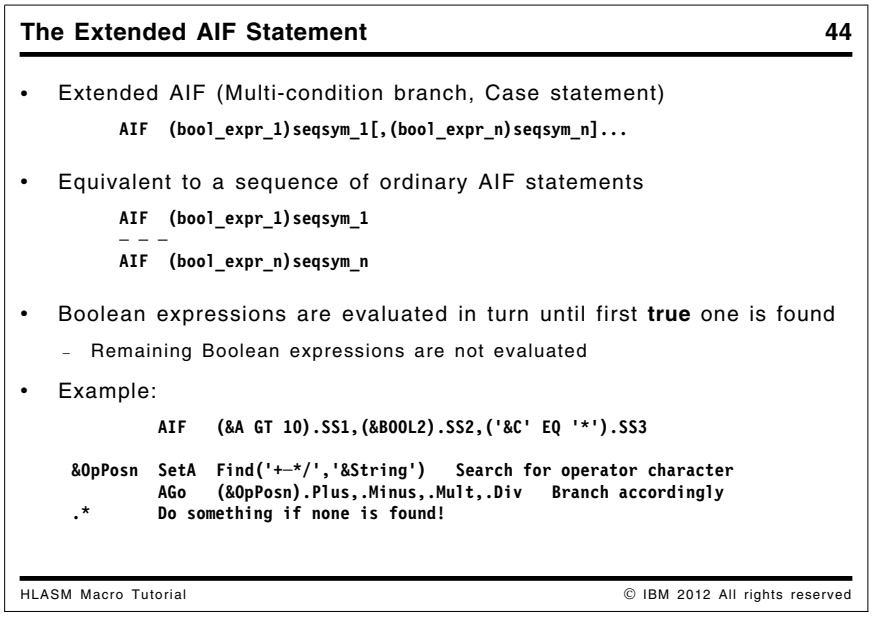
```
       AIF   (&A GT 10).BD
* (1) This statement is processed if (&A GT 10) is false
.BD    ANOP
* (2) This statement is processed
```

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 operation of the extended AGO statement illustrated in Figure 7 on page 44 is precisely equivalent to the following set of AIF statements:

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

This helps to illustrate why it is possible for no "branch" to be taken if the value of `arith_expr` isn't between 1 and k.



The extended form of the AIF statement lets you write multiple conditions and "branch targets" on a single statement, as shown Figure 8.

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

Figure 8. General Form of the Extended AIF Statement

The Boolean expressions are evaluated in turn until the first **true** expression is found; the next statement to be 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. If none is true, processing continues with the next sequential statement.

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 8 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.

Here are two examples of the Extended AGO statement in combination with the INDEX and FIND functions.

1. Suppose the character variable symbol &Response might contain one of four values: YES, NO, MAYBE, and NONE, and we wish to branch to some processing statements. The search can be done in a single statement:

```
&OK      SetC  'YES  NO   MAYBENONE '       5 positions per term
&RVal    SetA  4+Index('&OK','&Response')   Search for match
         AGO   (&RVal./5).Yes,.No.,Maybe,.None  Process response
         - - -                              No match found
```

2. Suppose you want to search an "expression string" for the presence of the arithmetic operators +, −, *, and /. The FIND function lets you locate an operator easily:

```
&OpPosn  SetA  Find('+-*/','&String')   Search for operator character
         AGo   (&OpPosn).Plus,.Minus,.Mult,.Div   Branch accordingly
         - - -  etc.                    No operator found
```

Displaying Symbol Values and Messages: MNOTE 45

- Useful for diagnostics, tracing, information, error messages
  - See macro debugging discussion (slide 85)
- Syntax:

```
MNOTE  severity,'message text'
```

- **severity** may be
  - any arithmetic expression with value between 0 and 255
    - value of **severity** is used to determine assembly completion code
  - an asterisk; the message is treated as a comment
  - omitted
- Displayable quotes and ampersands must be paired
- Examples:

```
.Msg_1B MNOTE  8,'Missing Required Operand'                  (severity 8)
.X14    MNOTE   ,'Conditional Assembly has reached .X14'     (severity 1)
.Trace4 MNOTE  *,'Value of &&A = &A., value of &&C = ''&C.''' (no severity)
        MNOTE    'Hello World (How Original!)'               (no severity)
```

HLASM Macro Tutorial

Displaying Symbol Values and Messages: The MNOTE Statement

The MNOTE statement provides a way for the conditional assembly language to "communicate" to the programmer. It 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 .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; 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 a simple message.

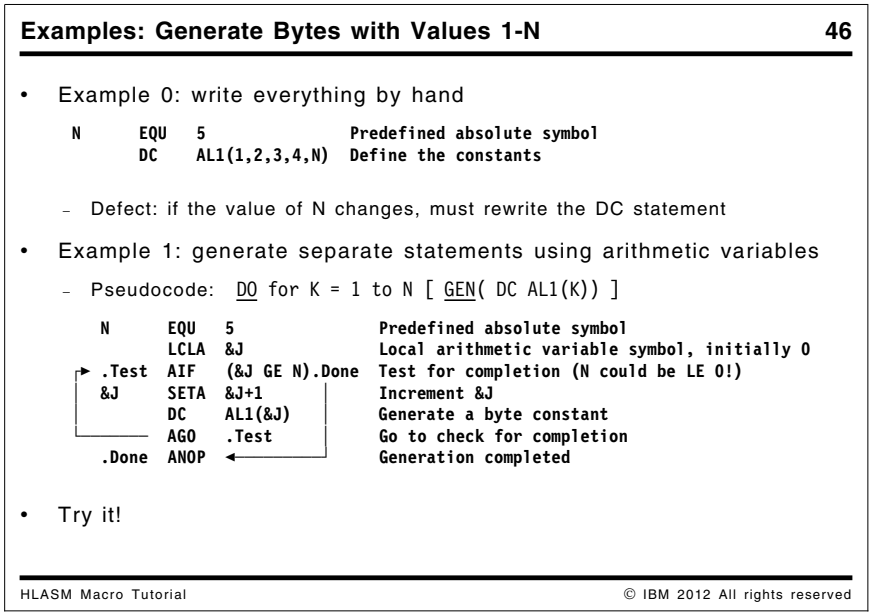
The first two MNOTEs are treated as "error" messages, which means that they will be flagged in the error summary in the listing and will appear in the SYSTERM output if the TERM option was specified. A setting of an assembly severity code is also performed. The latter two MNOTEs will be treated as comments, and will appear only in the listing.

Any quotation marks and ampersands intended to be part of the message must be *paired*, as illustrated in the example above. (The DOUBLE function (on page 38) will do this for you.)

The High Level Assembler provides two system variable symbols (&SYSM_SEV and &SYSM_HSEV) that allow you to determine the current values of MNOTE statement severities. These two variables will be discussed in "&SYSM_HSEV and &SYSM_SEV" on page 260.

Examples of Conditional Assembly

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



Example 1: Generate Bytes with Values 1-N

Suppose we wish to generate DC statements defining a sequence of byte values from 1 to N, where N is a predefined value. This can 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. Pseudo-code for such a sequence might look like this:

```
DO for K = 1 to N [ GEN( DC AL1(K)) ]
```

Conditional-assembly statements to generate the DC statements are:

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

Figure 9. Generating a Sequence of Bytes, Individually Defined

The LCLA declaration (2) of &J also initializes it to zero; we cannot omit the declaration in this example, because the first appearance of &J is in the AIF statement (3), not in the SETA statement (4). The AIF statement (3) compares &J to N (a predefined (1) absolute symbol), and if &J exceeds N, a "branch" is taken to the label .Done (7). If the AIF test does not change the flow of statement processing, the next statement (4) increments &J by one, and its new value is then substituted in the DC statement (5). The following AGO (6) then returns control to the test in the AIF statement at sequence symbol .Test (3).

This example is of course a hard way to create such a sequence; most programmers would write something like this:

```
Seq      DC    (N)AL1(*-Seq+1)        Generate bytes with 1,2,...,N
```

Example 2: Generate a Character String **47**

- Generate a *string* with the byte values (like '1,2,3,4,5')
  - Pseudocode:
    Set S='1'; DO for K = 2 to N [S = S || ',K'); GEN(DC AL1(S)]

```
   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: add comma and next value
└──────── 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', 'N EQU 90', 'N EQU 300'**

Another way that generates only a single DC statement constructs the nominal value string for the DC statement. A pseudo-code sketch of this method is:

```
Set S='1'; DO for K = 2 to N [ S = S || ',K'); GEN( DC AL1(S) ]
```

A conditional-assembly code sequence might be:

```
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: add comma and next value
        AGO   .Loop           Branch back to check for completed
.Done1  DC    AL1(&S.)        Generate the byte string
.Done2  ANOP                  Generation completed
```

Figure 10. Generating a Sequence of Bytes, as a Single Operand String

In this program fragment, a single character string is constructed with the sequence of integer 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 value field of the AL1 operand.

Test this example with values of N large enough to cause the string &S to become longer than (say) 60 characters: assign a value of 30 to N, and observe what the assembler does with the generated DC statement. (It creates a continuation continuation automatically!) Then try larger values of N.

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 with a new and different set of sequence symbols. We will see in “Case Study 2: Generating a Sequence of Byte Values” on page 115 that a simple macro definition can make this task easier to solve.

Example 3: System-Dependent I/O Statements **48**

- Suppose a system-interface module declares I/O control 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
  - Then, assemble the module with a system-specific macro library

HLASM Macro Tutorial

Example 3: Generating System-Dependent I/O Statements

Suppose you are writing a module that provides operating system services to an application. For 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 MVS, CMS, and VSE operating systems.

We use 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 11:

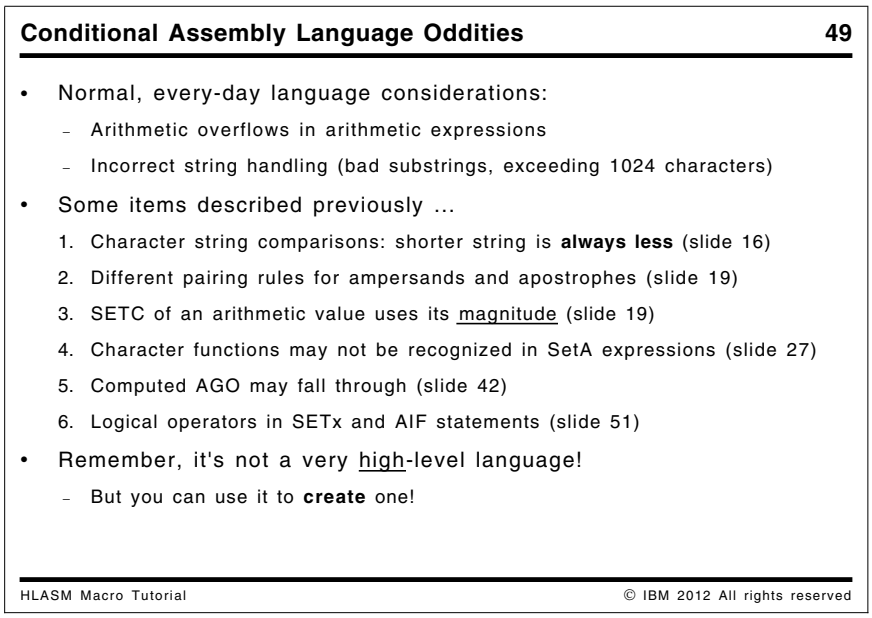
```
&OpSys   SETC  'MVS'              Set desired operating system
         - - -
         AIF   ('&OpSys' NE 'MVS').T1   Skip if not MVS
Input    DCB   DDNAME=SYSIN,...etc...   Generate MVS DCB
         - - -
         AGO   .T4
.T1      AIF   ('&OpSys' NE 'CMS').T2   Skip if not CMS
Input    FSCB  ,LRECL=80,...etc...      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 11. Conditional Assembly of I/O Module for Multiple OS Environments

In this example, different blocks of code contain the necessary statements for particular operating environments. In any portion of the program that contains statements for an environment, conditional assembly statements direct the assembler to select the correct statements for processing. By setting single variable symbol &OpSys, 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, 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. (You must of course supply an appropriate macro library to the assembler.)

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



Conditional Assembly Language Eccentricities

The previous text has described several potential pitfalls in the conditional assembly language; they are summarized here.

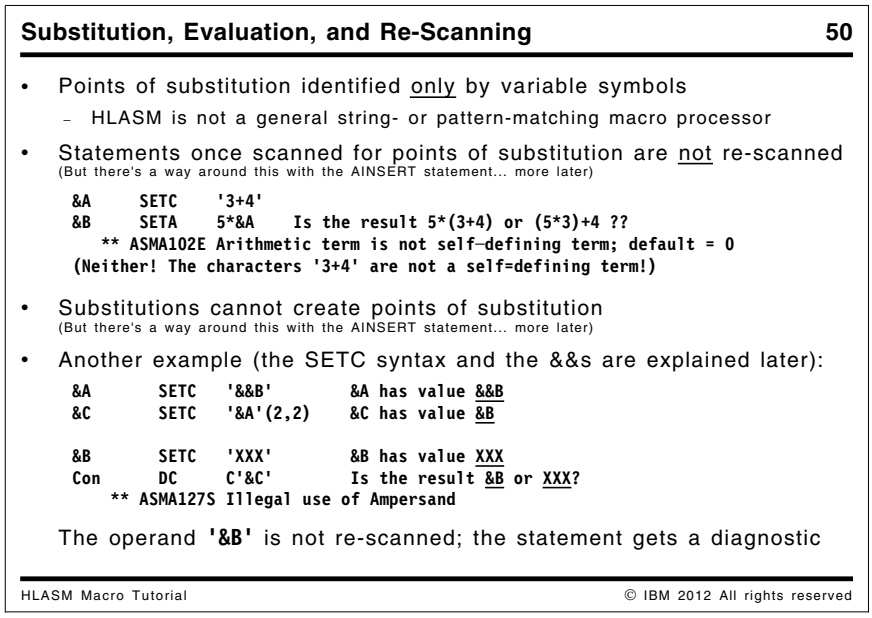
1. When character strings of unequal lengths are compared, the shorter string is always treated as being less than the longer string, even though a comparison of their first characters might indicate otherwise. (See “Evaluating and Assigning Boolean Expressions: SETB” on page 17.)
2. The pairing rules for ampersands and apostrophes are different from those in the ordinary assembler Language (apostrophes are, but ampersands are not). (See “Evaluating and Assigning Character Expressions: SETC” on page 19.)
3. Conversion of an arithmetic variable to a character string returns the magnitude of the variable; no minus sign is provided for negative values. The SIGNED internal function provides a minus sign. (See “Evaluating and Assigning Character Expressions: SETC” on page 19.)
4. Internal function evaluations involving string functions cannot always be “nested” in arithmetic expressions. (See “Character-Valued Functions Using Logical-Expression Format” on page 29.)
5. If the number of sequence symbols listed on an extended AGO does not match the value of the supplied variable, no branch is taken. (See “The Extended AGO Statement” on page 44.)
6. The rules for evaluating expressions involving logical operators such as AND and OR are different for SetA (arithmetic) and SetB (Boolean) expressions. AIF expressions are evaluated using the SetB rules. (See “Logical Operators in SETA, SETB, and AIF Statements” on page 55.)

In addition, *all* arithmetic overflow conditions are flagged; they cannot be suppressed. Most forms of incorrect string handling are also diagnosed.

Conditional Assembly Language Special Topics

Here we mention two special topics that sometimes arise when using the conditional assembly language:

- Substitution, evaluation, and re-scanning of substituted values.
- The interpretation of logical operators in SETA, SETB, and AIF statements.



Comments on Substitution, Evaluation, and Re-Scanning

The assembler uses a method of identifying points of substitution that may differ 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 recognized as candidates for substitution.
2. Statements are scanned only once to identify points of substitution. 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.
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 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 familiar rules for arithmetic expressions?

In fact, a third situation occurs: because the expression 5*&A is *not* re-scanned in any way. To evaluate the expression 5*&A, the assembler requires that the value of &A must be a self-defining term. That is, the assembler expects to evaluate a numeric constant. Because it is not — the '+' character is not part of a decimal self-defining term — the assembler produces this error message:

```
** ASMA102E 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 is an ordinary symbol, value 7
&B   SetA  5*N          &B has value 35

&T   SetC  'N'          Set &T to the character N
&C   SetA  5*&T         Error message for invalid term!
```

As another example, you might ask what happens in this situation: will the substituted value of &B in the DC statement be substituted again? (The pairing rules in SETC statements for ampersands are different from the pairing rules in DC statements, and are explained in "Evaluating and Assigning Character Expressions: SETC" on page 19.)

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

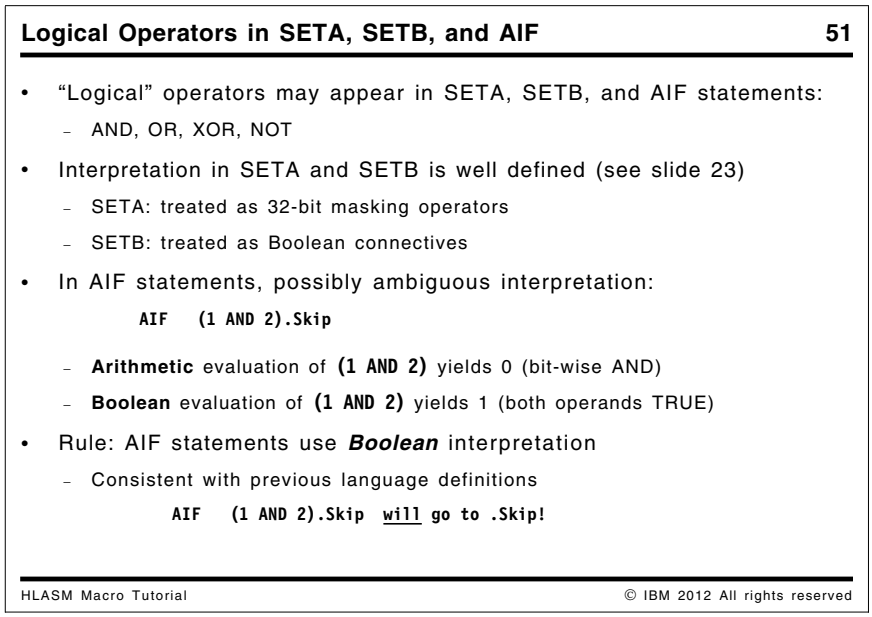
The answer is "no". In fact, this DC statement results in an error message:

```
** ASMA127S 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. (We will see in "The AINSERT Statement" on page 183 that there are some ways around this problem.)

As a final 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 notation previously described.



Logical Operators in SETA, SETB, and AIF Statements

The AND, OR, XOR, and NOT logical operators may appear in SETA, SETB, and AIF statements. In AIF statements there may be an ambiguous interpretation, while their interpretation in SETA and SETB statements is well defined:

- in SETA statements, they are treated as 32-bit masking operators;
- in SETB statements, they are treated as Boolean connectives (see "Conditional Expressions with Mixed Operand Types" on page 25).

In AIF statements, the following example illustrates is a possible ambiguity:

```
AIF   (1 AND 2).Skip
```

If the expression `(1 AND 2)` is evaluated using "SETA rules", its value is zero, because the arithmetic representations of 1 and 2 have no one-bits in common, so their logical AND is zero.

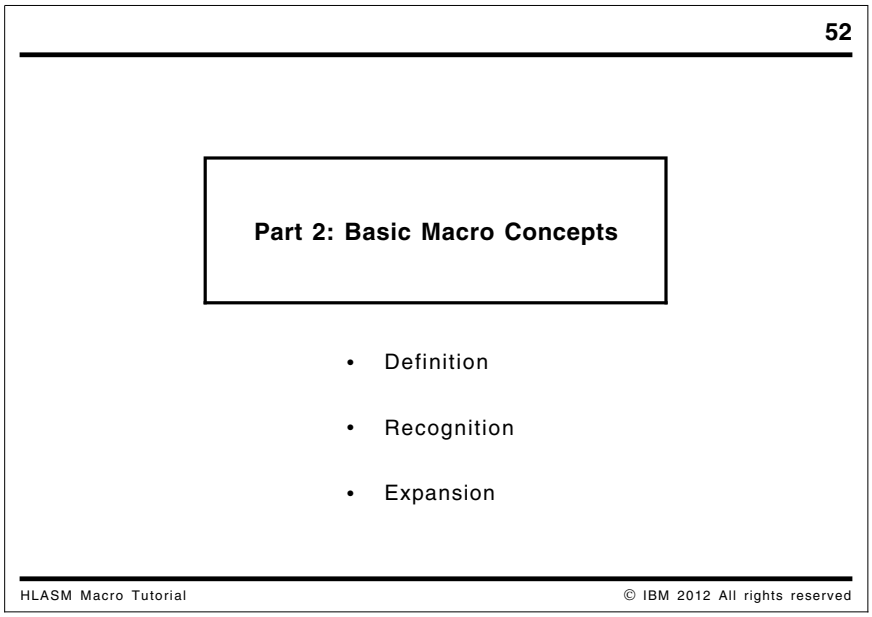
However, if the expression is evaluated using "SETB rules", then according to the conversion rules described in "Conditional Expressions with Mixed Operand Types" on page 25, the result must be 1 (both 1 and 2 are nonzero, so they are first converted to Boolean terms having value 1).

To avoid any possibility of ambiguity, High Level Assembler uses the **Boolean** interpretation in AIF statements. Thus,

```
AIF   (1 AND 2).Skip
```

will cause a conditional-assembly branch to `.Skip`.

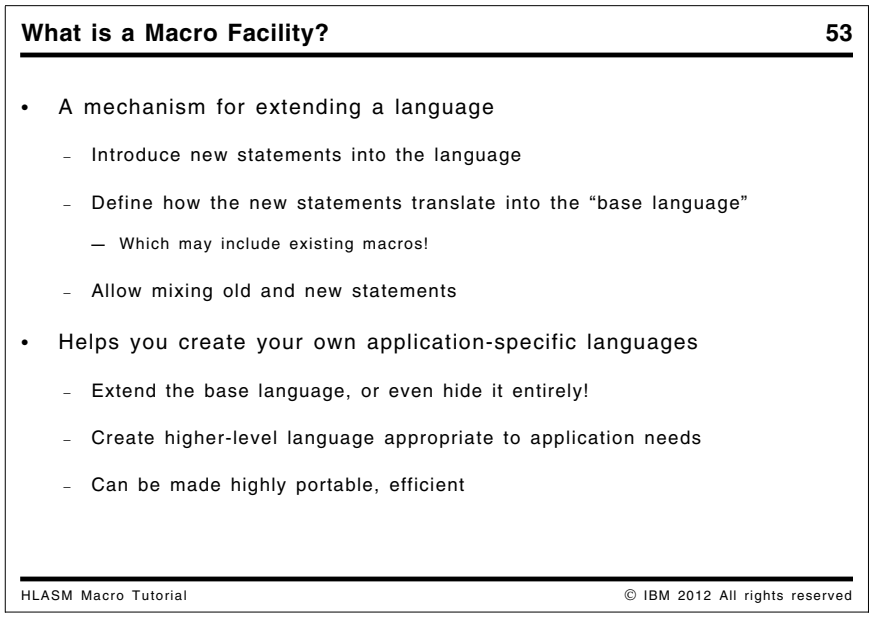
Part 2: Basic Macro Concepts



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

We begin with a conceptual overview of macros that is not specific to the Assembler Language. Then we investigate the Assembler Language's implementation of macros, including the following topics:

- macro definition: defining your 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 and text generation
- macro argument attributes and structures
- global variable symbols
- macro debugging



What is a Macro Facility?

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 when you are able to take advantage of them. The “old” statements may include existing macros, providing added leverage with each increment of growth.

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 Subroutines

You can think of a macro as an “assembly-time subroutine”. Macros and subroutines have many properties in common:

- They are “named” collections of statements invoked by that name, and to which various arguments are passed
- Their arguments are processed according to the logic of the internal statements
- Once written, they can be used in many programs.

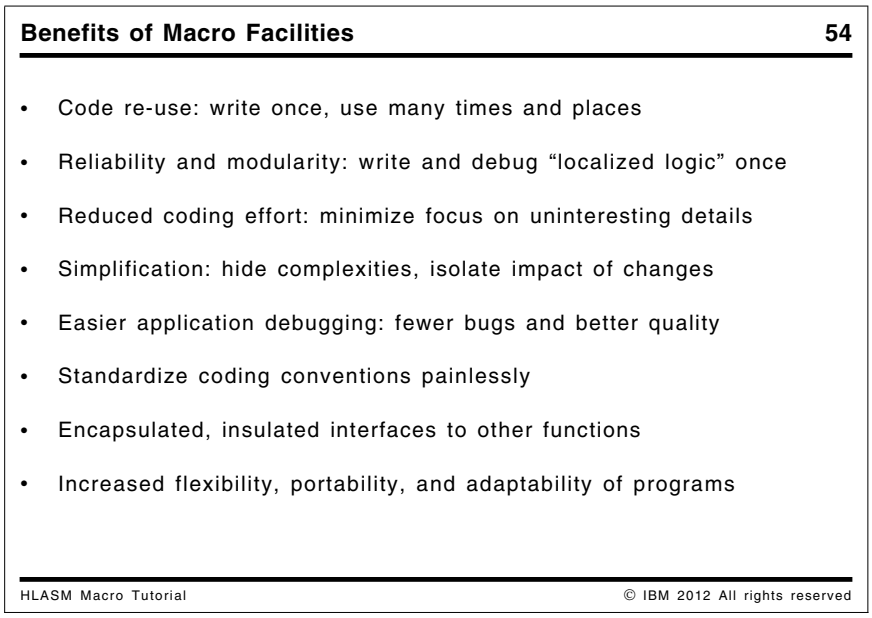
The major difference is that subroutines are called at the time a program is executed by a “hardware” processor (after having been translated to machine code), but a macro is executed by “software” *during* the translation (assembly) process, prior to the generation of machine code.

Macros have several advantages over most high-level language subroutines and functions (“procedures”):

- access to attribute information about arguments
- great flexibility in specifying and processing arguments
- simple methods for managing complex argument list structures.

However, macros also have several limitations:

- computed values returnable only via global variable symbols
- limited string- and statement-rescan capabilities.



Benefits of Macro Facilities

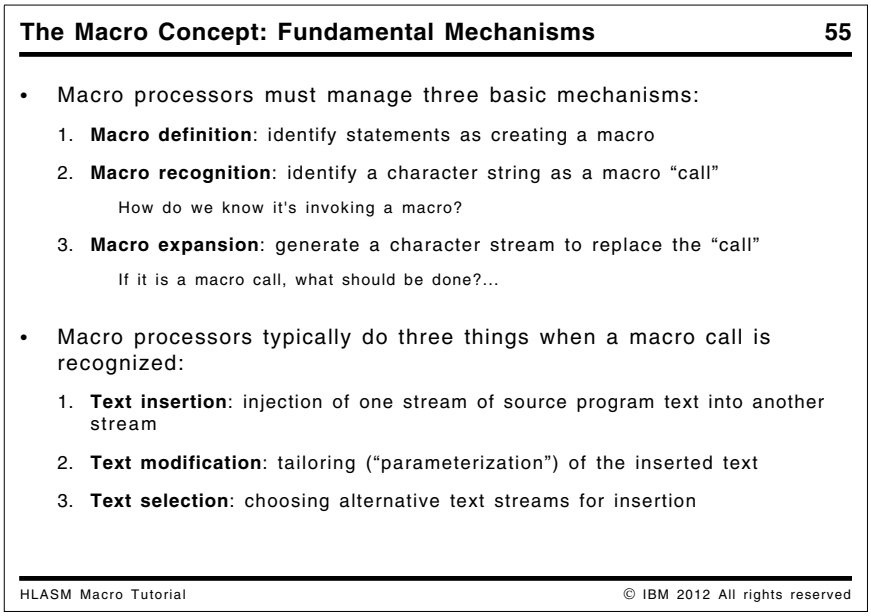
Macro facilities can provide you with many direct and immediate benefits:

- Code re-use: once a macro is written, it becomes available to many programmers and applications. A single definition can find multiple uses, even within a single application.
- Reliability and modularity: code and debug the logic in one place.
- Reduced coding effort: macro needs to be written only once, and then can be used in many places.
- Reduced focus on uninteresting details: macros allow you to create “higher-level” elements of your programming language, relieving you of concerns with details that are only marginally relevant to your programming task.
- 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.[3]
- 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.

---

[3] 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. Another example is the IBM Fortran G compiler, which was written entirely in a portable macro-based language.

- 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 code sequences can be implemented once in a macro, and used wherever needed, without every user of the macro having to understand the "inner workings" of the macro's logic.
- 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.



The Macro Concept: Fundamental Mechanisms

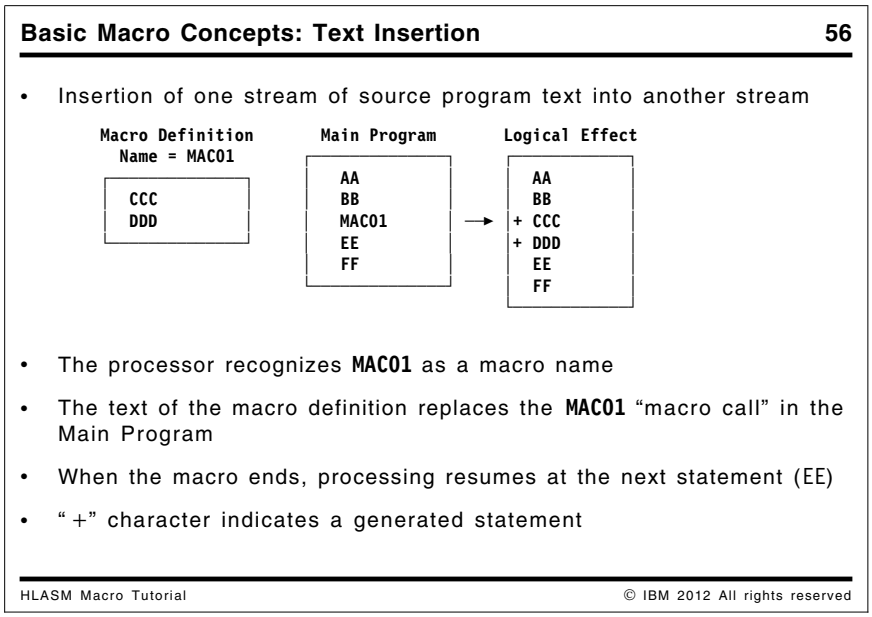
Macro processors typically rely on two basic processes:

- *Macro recognition* requires that the processor identify a 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. Some macro processors let you re-scan the replacement string to see if it offers new macro recognition opportunities.

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 12, a set of statements has been defined to be a macro named 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. Finally, when the macro ends, statement processing resumes at the next statement (EE) following the macro call.

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

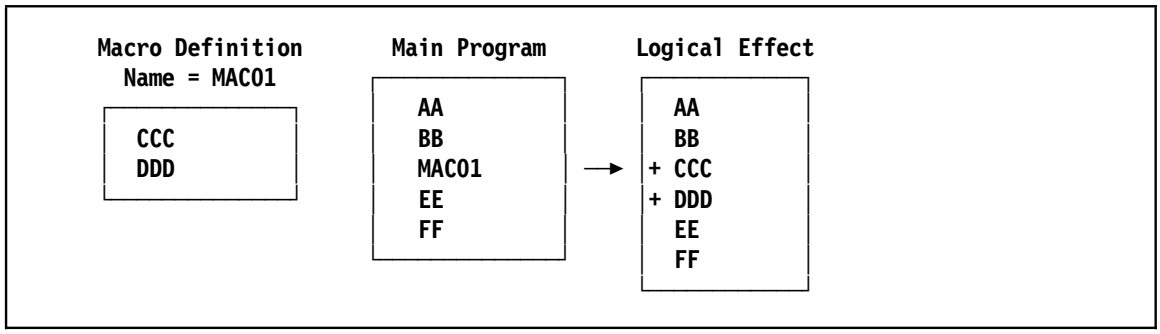
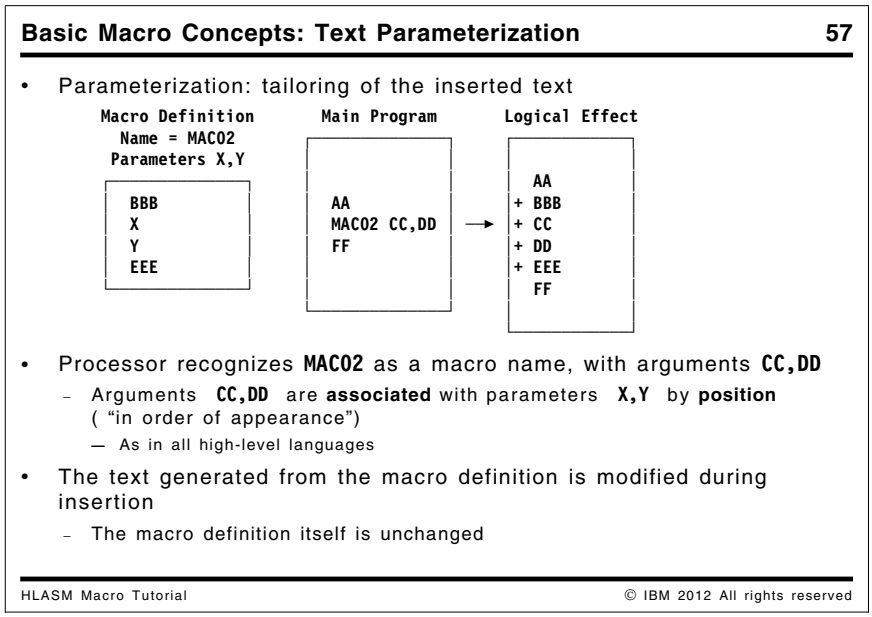


Figure 12. Basic Macro Mechanisms: Text Insertion

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



Text Parameterization and Argument Association

Simple text insertion has rather limited uses, because we usually want to adapt the inserted text to accommodate the conditions of each macro invocation. The simplest form of such adaptation is “text parameterization”. In Figure 13, a macro named MACO2 is defined with two *parameters* X and Y: that is, they are place-holders in the definition that indicate where other text strings are expected to be inserted when the macro is expanded.

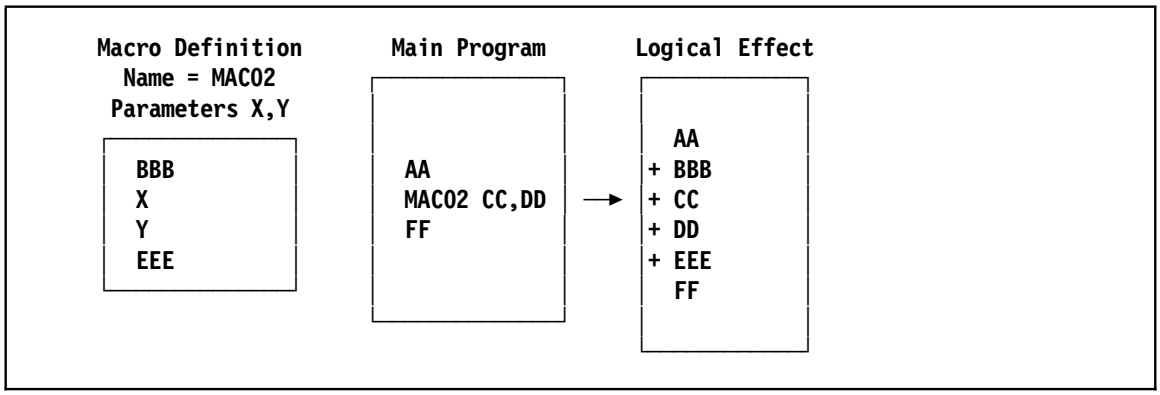
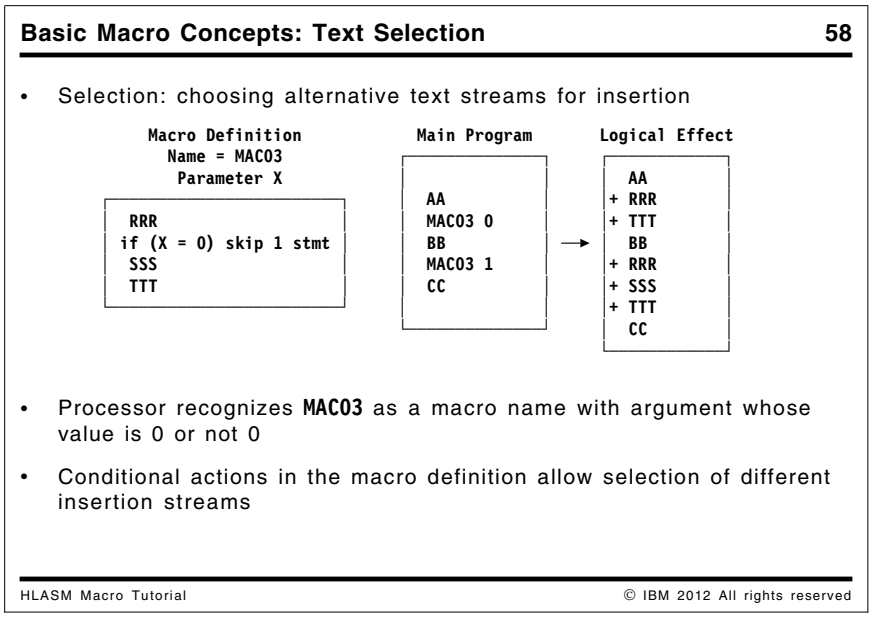


Figure 13. Basic Macro Mechanisms: Text Parameterization

When a macro call is recognized, additional information besides the simple act of activating the definition can be passed to the macro expansion. Thus, when the processor of the Main Program recognizes MACO2 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 example of parameter-argument association is typical of many macro processors: association proceeds in left-to-right order, matching each positional parameter in turn with its corresponding positional argument. This is the form of association used in almost all high-level programming languages; other forms of association (such as keyword association) are possible, as we will see shortly.



Text Selection

Text selection is fundamental to most macro processors, because it allows choices among alternative sequences of generated text. In Figure 14, a simple form of text selection is modeled by the **if** statement: the parameter X is associated with the argument of the two calls to MAC03. A simple test of the argument corresponding to X tells whether or not to generate the string SSS. If the argument is 0, SSS is not generated; otherwise it is.

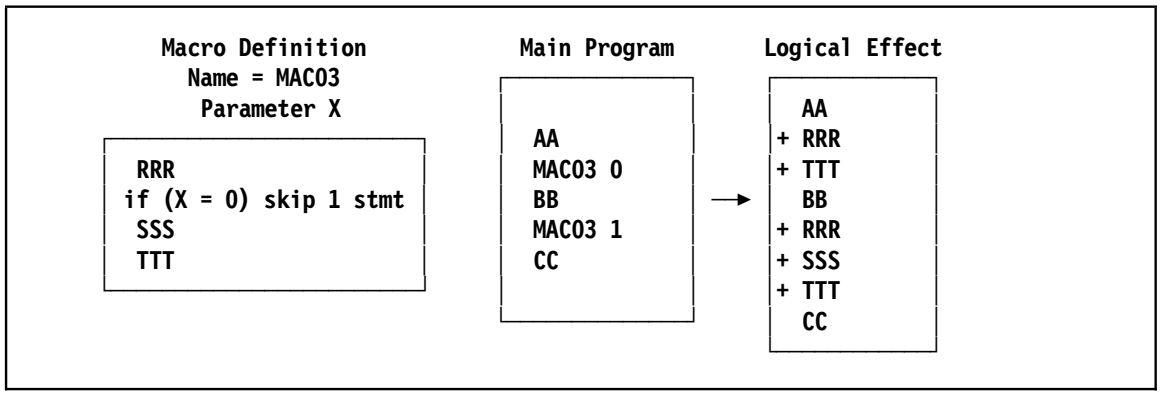
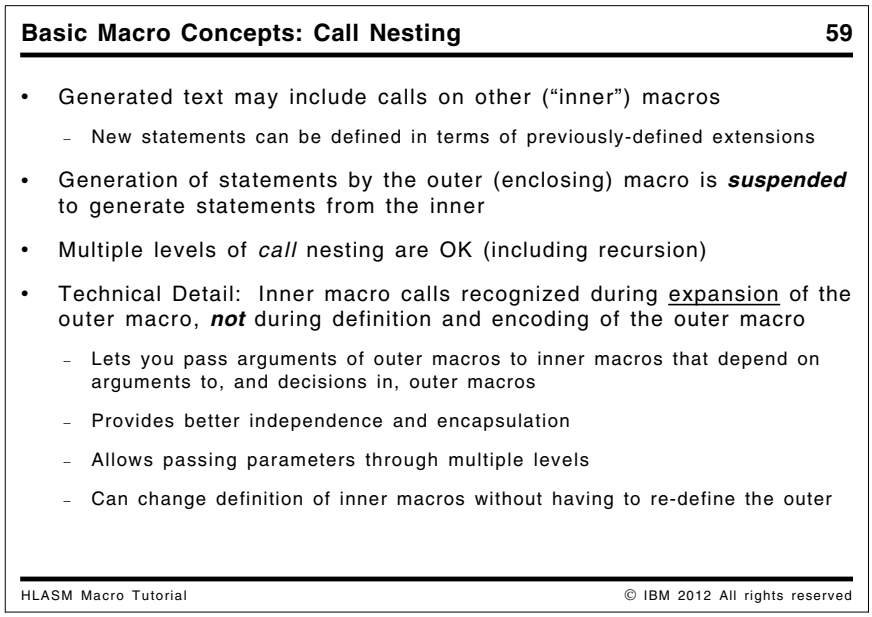


Figure 14. Basic Macro Mechanisms: Text Selection



Macro Call 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 “macro call nesting” or “macro nesting”: generated text may include (or create!) calls on other macros (“inner macro calls”). This mechanism lets you define new statements in terms of previously-defined extensions; it is fundamental to much of the power and “leverage” of macro languages.[4]

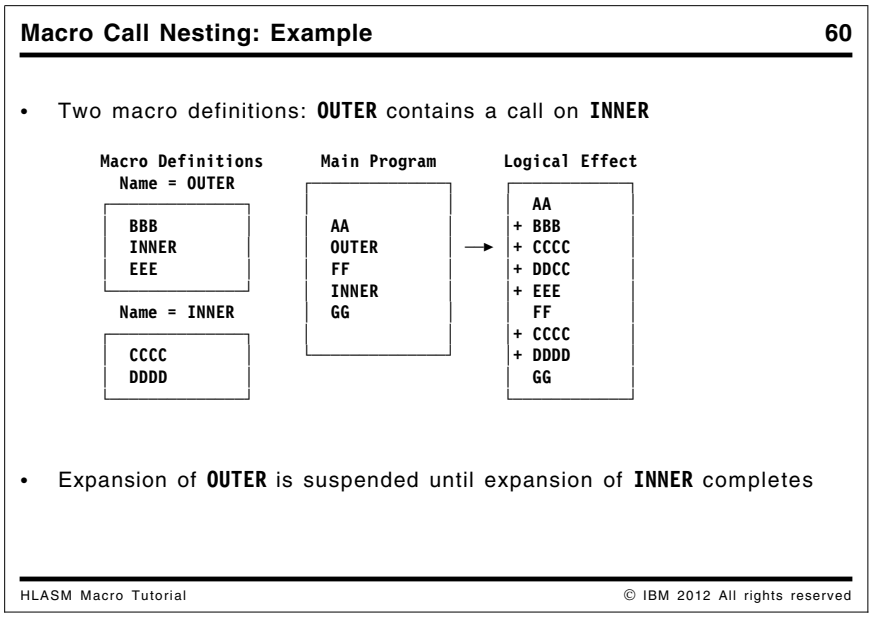
The generation process for inner macro calls requires that the macro processor maintain a “push-down stack” for its activities.

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

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

- By deferring the recognition of inner macro calls until an 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.

[4] Some call this process “bootstrapping” because each macro can be used to build new ones, increasing the power and expressiveness of your macros at each step.



In the example in Figure 15, the OUTER and INNER macros are known to the processor of the Main Program. When 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 a 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 at the next sequential statement (EE) following the call on INNER; when the expansion of the OUTER macro completes, processing resumes in the Main Program following the OUTER statement, at FF.

Note also that the INNER macro can be called from the Main Program, because it is known to the processor at the time its call is recognized.

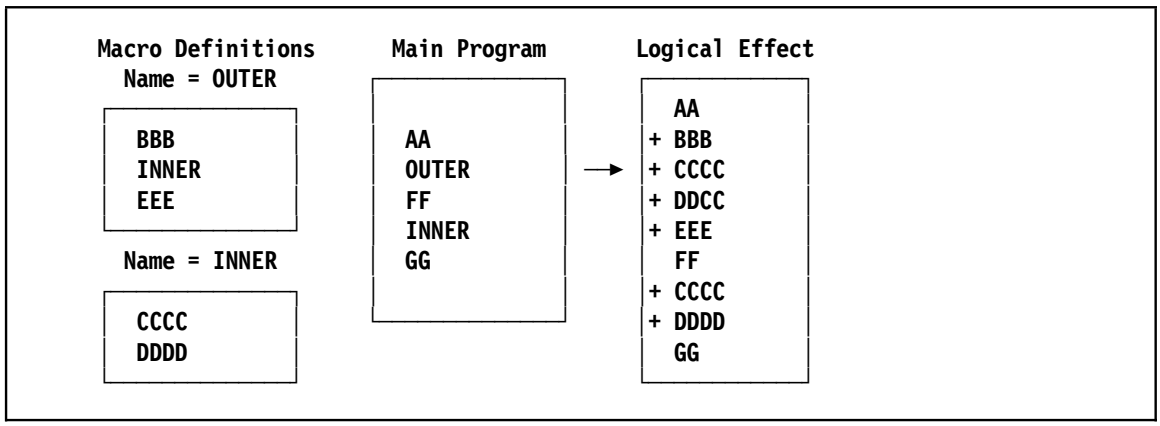
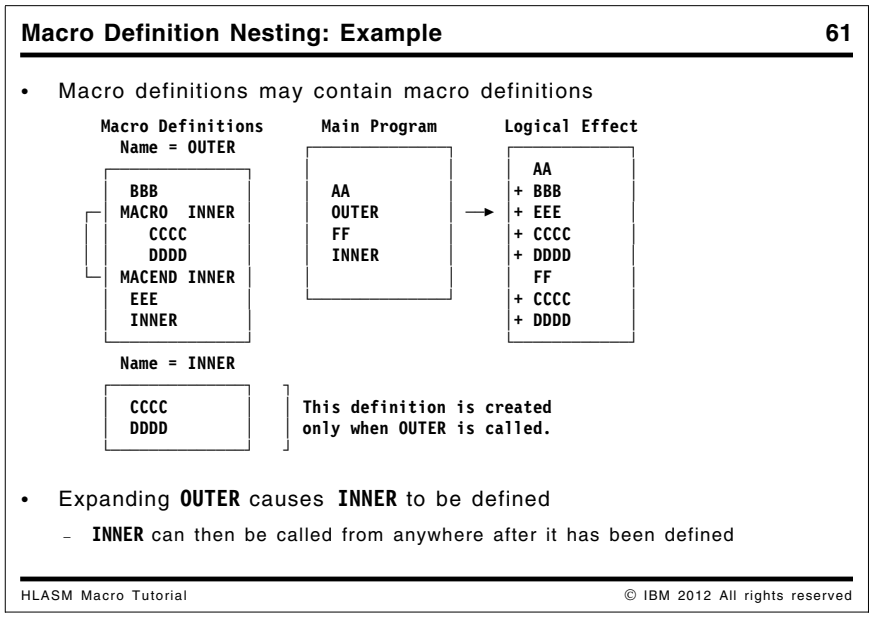


Figure 15. Basic Macro Mechanisms: Call Nesting

Each of the capabilities described above can be expressed in a natural way in the Assembler Language.



Macro Definition Nesting

While macro *call* nesting is widely used, macro *definition* nesting is relatively rare. The idea of macro definition nesting is illustrated in Figure 16, where we suppose that the definition of the macro named INNER is enclosed within the MACRO and MACEND statements.

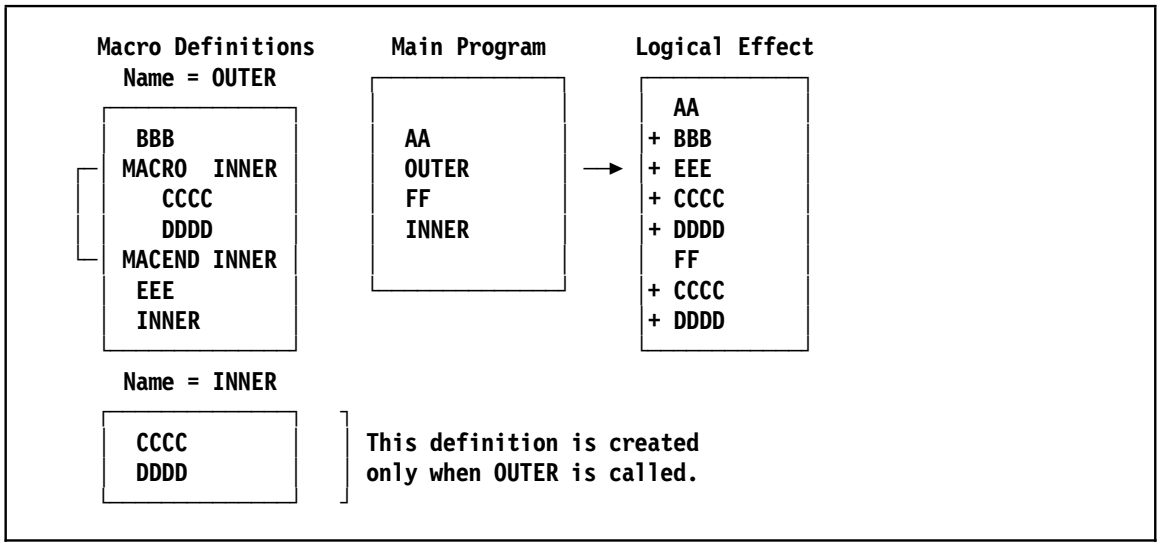


Figure 16. Basic Macro Mechanisms: Nested Macro Definitions

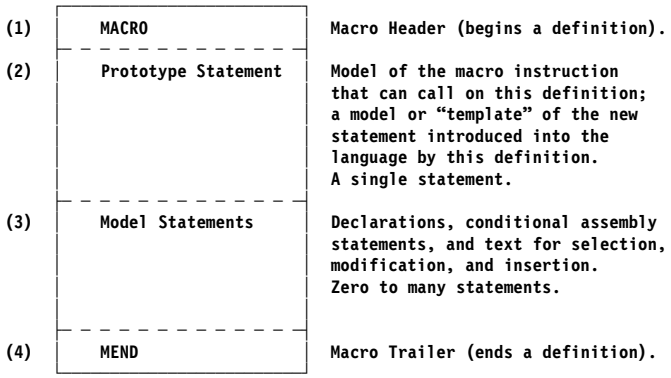
In this example, only the OUTER macro is known initially to the processor of the Main Program. When OUTER is recognized as a macro call, processing of the Main Program is suspended and expansion of OUTER begins. When the OUTER-generated statement MACRO INNER is recognized, the processor begins to create a *new* macro definition for INNER, saving the following statements until the MACEND INNER statement is recognized.

Later in the expansion of the OUTER macro, the nested call on the INNER macro is recognized, and the previously described mechanisms are used to generate the statements of the INNER macro. When the expansion of OUTER completes and processing of the Main Program resumes, the INNER macro call is now recognized and expanded.

Note that the INNER macro definition is known to the macro processor only *after* it has been generated during expansion of the OUTER macro. If INNER had been called from the Main Program *prior* to a call on OUTER, the processor would have to treat it as an unknown operation. After OUTER has been called, INNER can be called from anywhere in the main program or from other macros.

The Assembler Language Macro Definition **62**

- A macro definition has four parts:



HLASM Macro Tutorial

The Assembler Language Macro Definition ... **63**

1. **MACRO** and **MEND** statements delimit start and end of the definition
2. Declares the macro name on a *prototype statement*
   - Prototype statement also declares parameter variable symbols
3. Model statements ("macro body") provide logic and text

- Definitions may be found
  - "in-line" (a "source macro definition")
  - in a library (COPY can bring definitions "in-line")
  - or both
- Macro call recognition rules affected by where the definition is found

HLASM Macro Tutorial

The Assembler Language Macro Definition

The definition of a macro declares the macro name that represents a set of statements. An Assembler Language macro definition has four parts:

1. a macro header statement (MACRO: the start of the definition)
2. a prototype statement provides the macro name and a model or "template" 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 17:

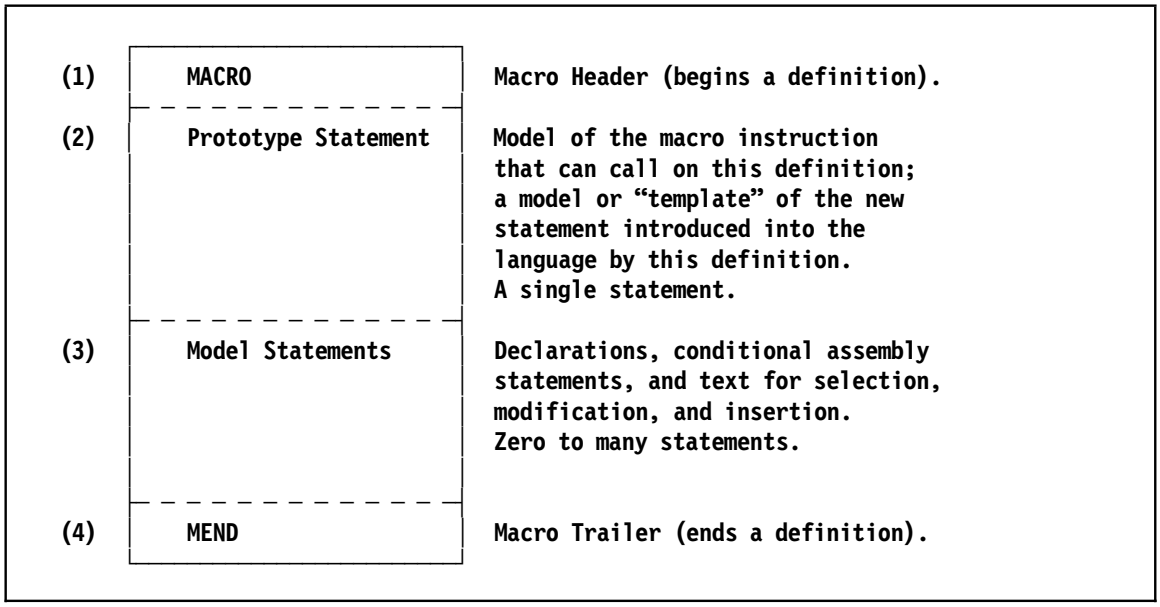


Figure 17. Assembler Language Macro Definition: Format

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 we'll see in "Macro-Instruction Recognition: Details" on page 98.

Macro-Instruction Definition Example

We can rewrite the example in Figure 12 on page 60 to look more like a "real" macro, as follows:

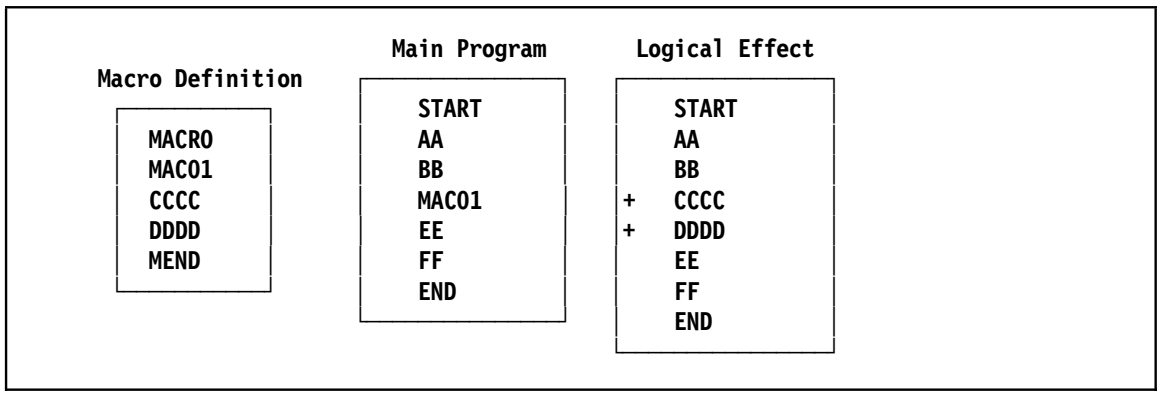
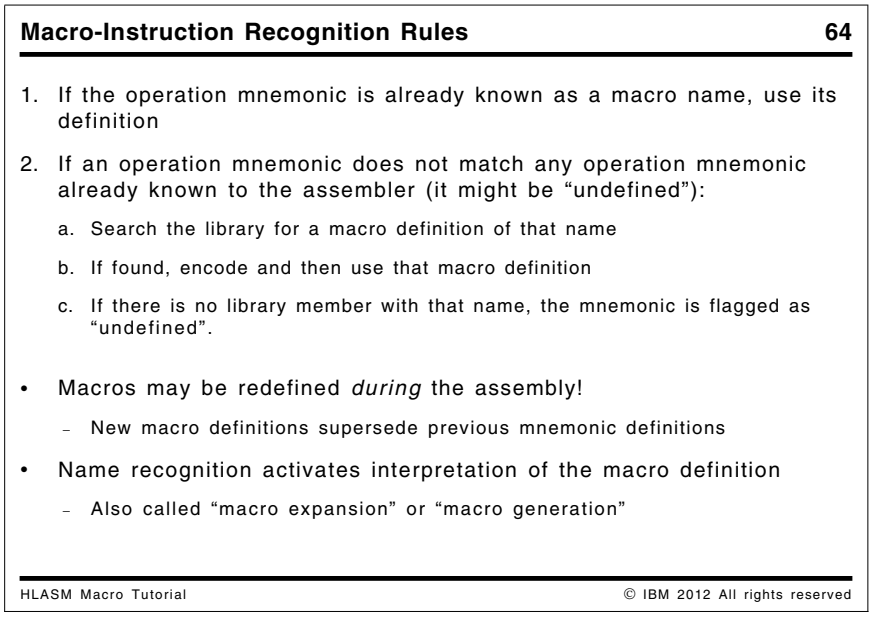


Figure 18. Assembler Language Macro Mechanisms: Text Insertion by a “Real” Macro



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 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. Some of the conditional-assembly statements cannot be overridden.

2. If an operation mnemonic does not match any operation mnemonic 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.

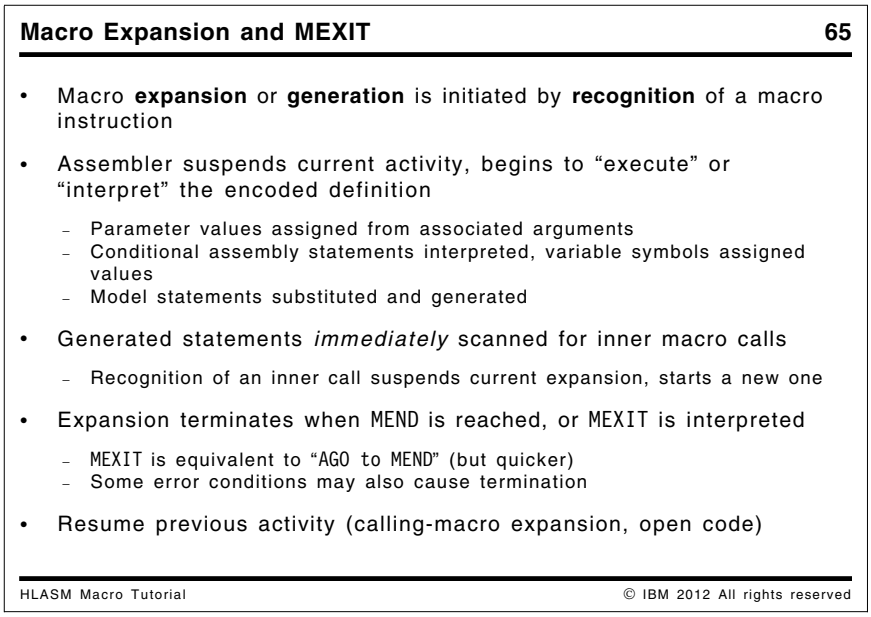
c. If there is no library member with that name, then the operation code is flagged as "undefined".

While it is not a common practice, macros may be redefined during the assembly by introducing a new macro definition with the same name.

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

Source or "in-line" macros are usable only in the program that contains them, whereas library macros can be used in any program.

The `O'` attribute reference can be used to determine the status of a macro or instruction name. Its uses are specialized, and will not be discussed here.



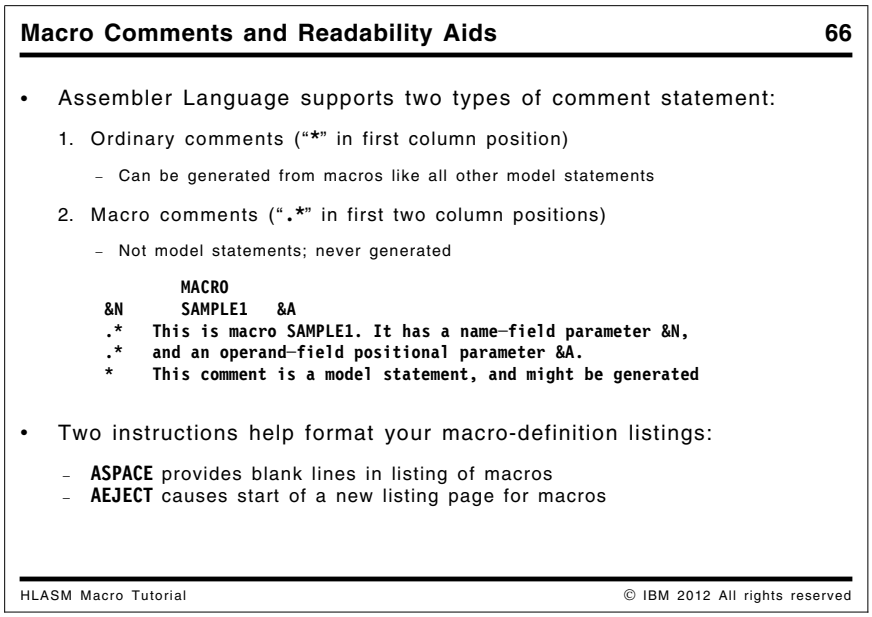
Macro Expansion, Generated Statements, and the MEXIT Statement

When the assembler recognizes a macro instruction, macro *expansion* or macro *generation* begins. 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, substitutions are made in model statements as needed, and generated statements are output.

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

Expansion of a macro terminates either when the `MEND` statement is reached, or an expansion-terminating `MEXIT` macro-exit statement is interpreted. `MEXIT` is equivalent to an "`AGO to MEND`" statement, but is quicker to execute, because the assembler need not search for the target of the AGO statement.



Macro Comments and Readability Aids

You can 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 can be part of the generated text from a macro expansion. Macro comments are never generated, and are defined by the characters `.*` in the first two columns, as illustrated in Figure 19:

```
        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 might be generated
        - - -
        MEND
```

Figure 19. Example of Ordinary and Macro Comment Statements

You should comment your macro definitions generously, because the conditional assembly language is sometimes difficult to read and understand.

Formatting and printing 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 not generated.

Example 1: Define General Register Equates 67

- Generate EQUates for general register names (GR0, ..., GR15)

```
         MACRO                    (Macro Header Statement)
         GREGS                    (Macro Prototype Statement)
GR0      EQU   0                  (First Model Statement)

.*       - - -   etc.             Similarly for GR1 — GR14

GR15     EQU   15                 (Last Model Statement)
         MEND                     (Macro Trailer Statement)
```

- A more interesting variation with a conditional-assembly loop:

```
            MACRO
            GREGS
            LCLA  &N             Define a counter variable, initially 0
.X          ANOP  ,              Next statement can't be labeled
.* 2 points of substitution in EQU statement
GR&N        EQU   &N
&N          SETA  &N+1           Increment &N by 1
            AIF   (&N LE 15).X   Repeat for all registers 1-15
            MEND
```

HLASM Macro Tutorial

Example 1: Define Equated Symbols for Registers

Suppose you wish to define a macro named GREGS that generates a sequence of EQU statements to define symbolic names GR0, GR1, ..., GR15 for the sixteen General Purpose Registers. Calling the GREGS macro will define them:

```
         MACRO                    (Macro Header Statement)
         GREGS                    (Macro Prototype Statement)
GR0      EQU   0                 ┌(First Model Statement)
GR1      EQU   1                 │
GR2      EQU   2                 │
GR3      EQU   3                 │
GR4      EQU   4                 │
GR5      EQU   5                 │
GR6      EQU   6                 │
GR7      EQU   7                 │     Model Statements
GR8      EQU   8                 │
GR9      EQU   9                 │
GR10     EQU   10                │
GR11     EQU   11                │
GR12     EQU   12                │
GR13     EQU   13                │
GR14     EQU   14                │
GR15     EQU   15                └(Last Model Statement)
         MEND                     (Macro Trailer Statement)
```

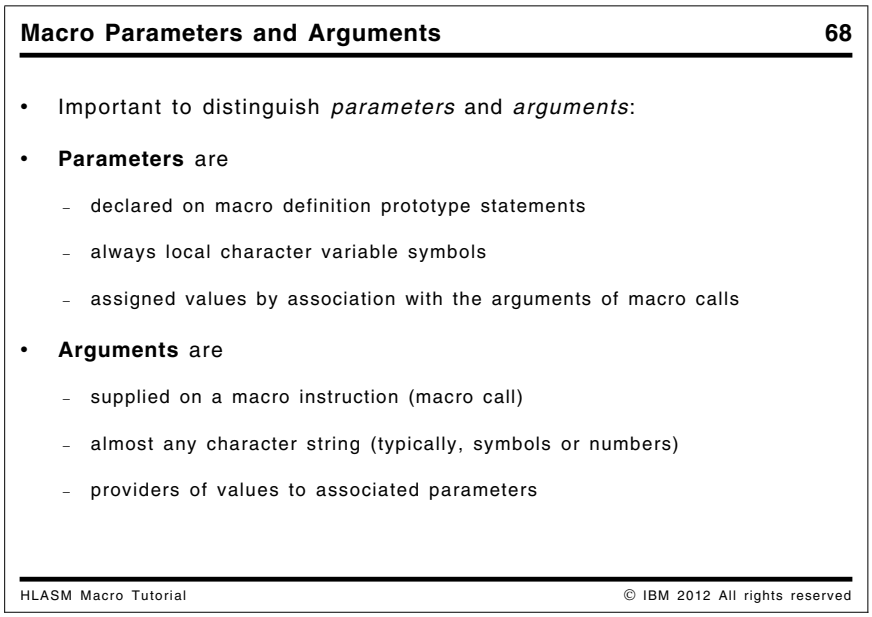
Figure 20. Simple Macro to Generate Register Equates

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

```
         MACRO
         GREGS
         LCLA  &N            Define a counter variable, initially 0
.X       ANOP  ,
.* 2 points of substitution in EQU statement
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 21. Macro to Generate Register Equates Differently

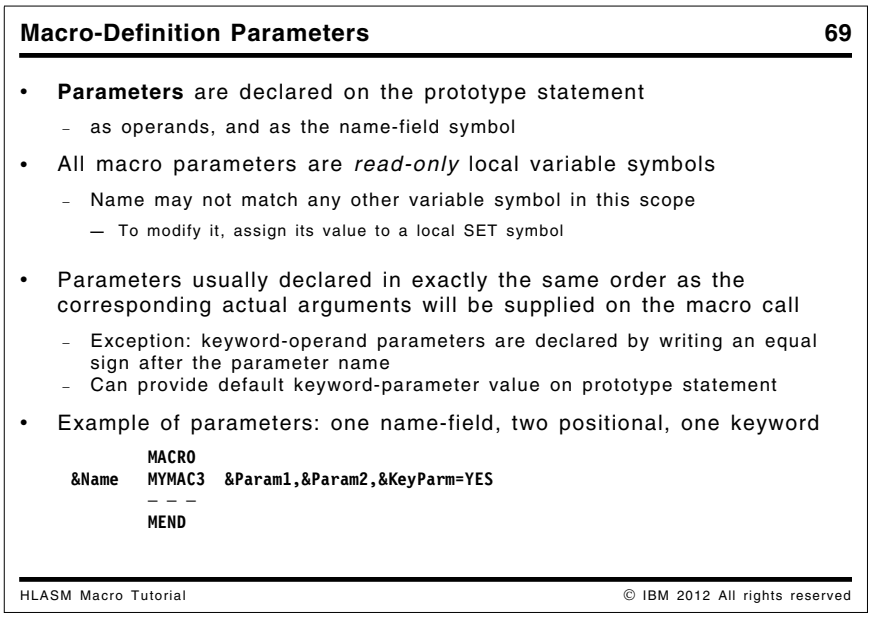
We'll revisit this example when we discuss “Case Study 1: Defining Equated Symbols for Registers” on page 110.



Macro Parameters and Arguments

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

- Parameters are
  - declared on the prototype statements of macro definitions
  - always read-only local character 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

Parameters in a macro definition are implicitly declared by appearing as operands (and the name-field symbol) on the prototype statement. These declared parameters *are* variable symbols, but they cannot be assigned a value in the body of the macro because the value is assigned by *association* when the macro is called, as described in "Macro Parameter-Argument Association" on page 76. Usually, parameters are declared in the same order as the corresponding actual arguments will be supplied on the macro call.

If you need to modify the value of a symbolic parameter, assign it first to a local SET symbol.

The exceptions are *keyword parameters*: 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, argument values for keyword parameters are supplied by writing the keyword parameter name, an equal sign, and the argument's value, as an operand of the macro call.

The name of a parameter may not be the same as that of any other variable symbol known in the macro's scope.

For example, suppose we write a macro prototype statement as shown in Figure 22:

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

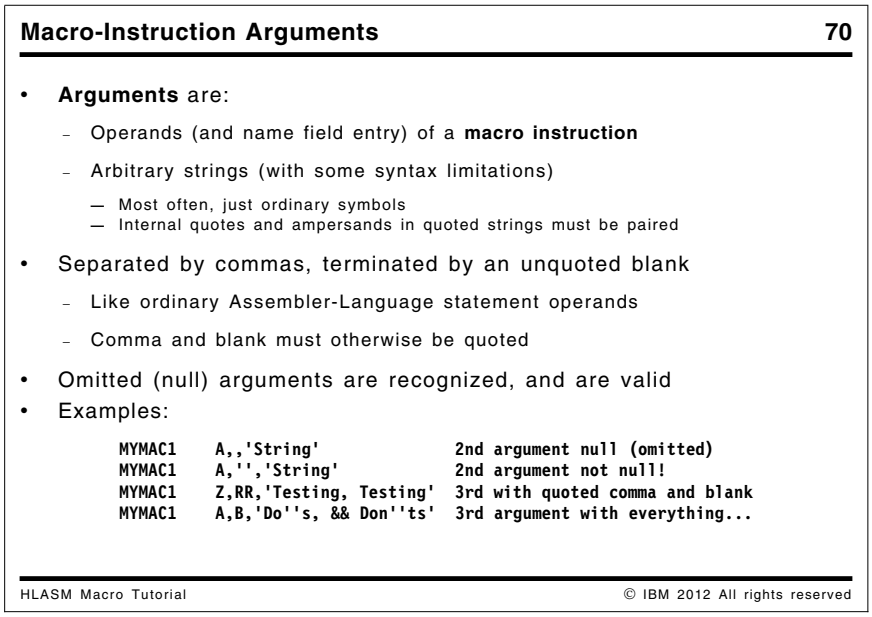
Figure 22. Sample Macro Prototype Statement

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, as in the following figure:

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

Figure 23. The Same Macro Prototype Statement



Macro-Instruction Arguments

The arguments of a macro instruction are its name-field entry and its 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 symbols; literals are allowed in almost all circumstances.

The operands are separated by commas and terminated by an unquoted blank, conforming to the normal Assembler Language syntax rules. If an argument is intended to contain a character normally used to delimit operands (blank, comma, parentheses, and sometimes apostrophes and periods), they must be quoted with apostrophes. Remember that the enclosing apostrophes are passed as part of the associated parameter's value, so you may need to test for (and maybe remove) them before processing the enclosed characters.

Positional arguments are written in the order required for correspondence with their associated positional parameters in the macro definition. Keyword arguments may be intermixed freely in any order among the positional arguments, without affecting the ordering of the positional arguments (but doing so usually makes the macro call harder to read).

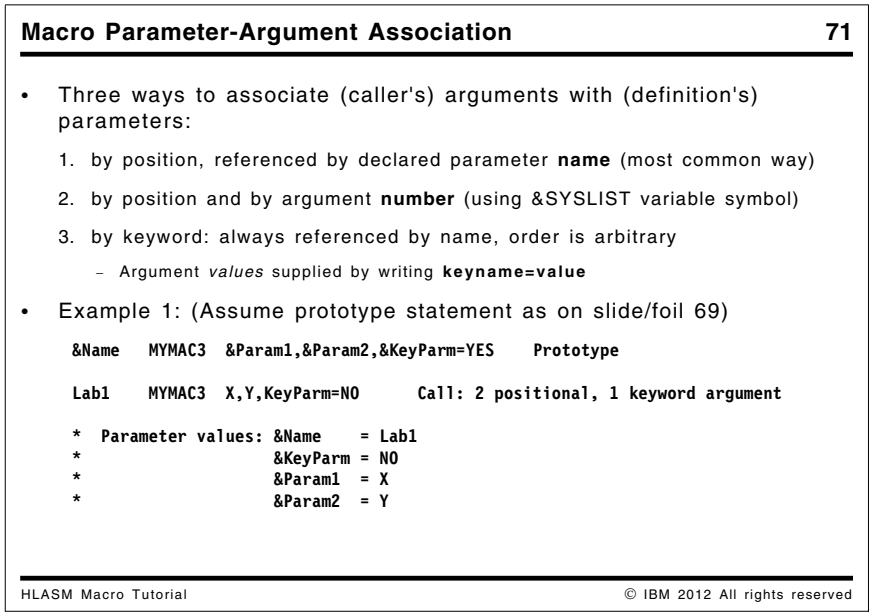
Omitted (null) arguments are valid. To illustrate, suppose a macro named MYMAC1 expects three positional arguments. Then in the following example,

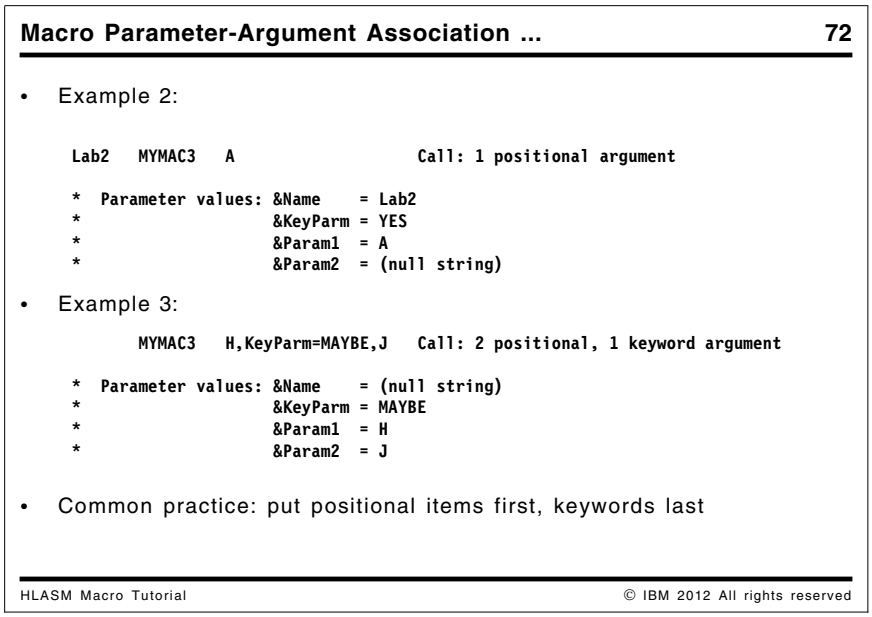
```
MYMAC1    A,,'String'              2nd argument null (omitted)

MYMAC1    A,'','String'            2nd argument not null!

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

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

the first call omits the second argument (it's a null string); the second call has a non-null quoted (but empty) character string; the third call has a quoted character string containing an embedded comma and space in its third argument; and the fourth call has a variety of special characters in its quoted-string third argument.

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

An argument consisting of a single ampersand will be diagnosed by the assembler as an invalid variable symbol. An argument consisting of a single apostrophe will appear to initiate a quoted string, and the assembler's reactions are unpredictable; one possibility is an error message indicating "no ending apostrophe".





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 we will discuss in “Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol” on page 86)
3. by keyword: keyword arguments are always referenced by name, and the order in which they appear is arbitrary.[5] Values provided for keyword arguments override default values declared on the prototype statement.

Note that the name-field parameter can be treated as a positional argument in a macro by referencing it as &SYSLIST(0).

To illustrate, consider the examples in Figure 24 on page 77. Assuming the same macro definition prototype statement shown in Figure 22 on page 73, the resulting values associated with the parameters are as shown:

[5] 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!

```
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 string)

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

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

Figure 24. Macro Parameter-Argument Association Examples

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

It is best not to mix positional and keyword parameters and arguments, because it may be difficult to count the positional items correctly.

Example 2: Generate a Byte Sequence (BYTESEQ1) **73**

- Rewrite previous example (slide 46) as a macro
- BYTESEQ1 generates a separate DC statement for each byte value

```
        MACRO
&L      BYTESEQ1 &N            Prototype stmt: positional parameters &L, &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
```

- Examples

```
BSeq1   BYTESEQ1  5
        BYTESEQ1  1
```


Example 2: Generate a Sequence of Byte Values (BYTESEQ1)

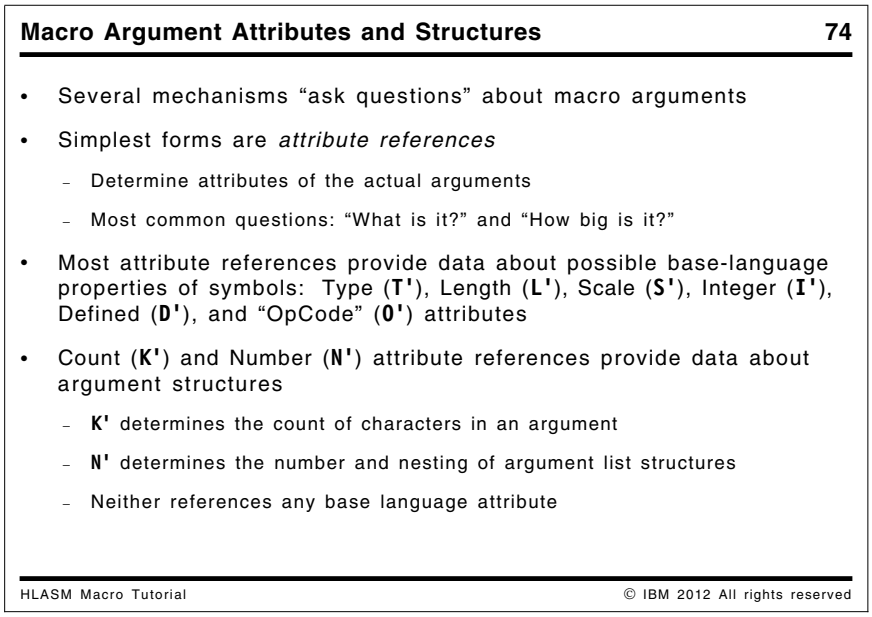
We can write a macro named BYTESEQ1 with a single parameter that will generate a sequence of bytes, using the same techniques as the conditional-assembly example given in Figure 9 on page 49. The pseudo-code for the BYTESEQ1 macro is quite simple:

```
IF (name-field label is present) GEN(label DS 0AL1)
DO for K = 1 to N [ GEN(  DC  AL1(K)) ]
```

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
&L      BYTESEQ1 &N             Prototype statement: 2 positional parameters
.*  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
BSeq1   BYTESEQ1  5
        BYTESEQ1  1
```

Figure 25. Macro to Define a Sequence of Byte Values



Macro Argument Attributes and Structures

Attribute references let you determine properties (attributes) of actual arguments. They also provide information about possible base language use of the symbols: what kinds of objects they name, the length attribute of the named object, etc.

Three major classes of attribute inquiry facilities are provided:

1. The "mechanical" or "physical" characteristics of macro arguments, independent of any meaning they might have, can be determined by

   - 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. (It also provides the largest subscript assigned to a dimensioned variable symbol, as we saw at "Declaring Variable (SET) Symbols" on page 9)

   - list-structure referencing and decomposition operations, using subscripted references to parameter variable symbols.

     A powerful list scanning capability helps you decompose argument structures, especially parenthesized lists. With this notation, you can

     – determine the number and nesting of list structures

     – extract sublists or sublist elements

     – use substring and concatenation operations to manipulate portions of lists and list elements.

     List structures and techniques for scanning them are described at "Macro-Instruction Argument Lists and Sublists" on page 84.

2. The type attribute reference (T') allows you to ask "What base-language type might be attached to it?" about a macro argument. The value of the type attribute reference can tell you whether the argument is

   - a base-language symbol that names data, machine instructions, macro instructions, sections, etc.

   - a self-defining term (binary, character, decimal, or hexadecimal)

   - an "unknown" type.

3. The "Opcode" attribute (O') can be used to test a symbol for possible use as an instruction. Its value tells you whether the symbol represents an assembler instruction, a machine instruction mnemonic, an already-encoded macro name, or a library macro name. Its uses will not be described further here.

   Only the opcode (O') and type (T') attribute references have character values.

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

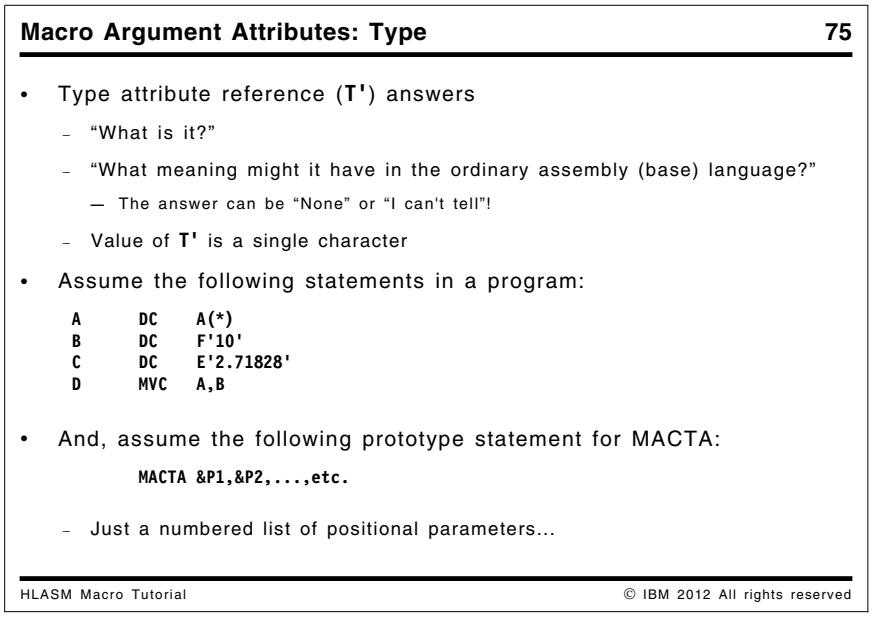
There is an important difference between the number (N') and count (K') attributes and all the others: N' and K' treat their operands only as strings of characters, independent of any meaning that might be associated with the strings.

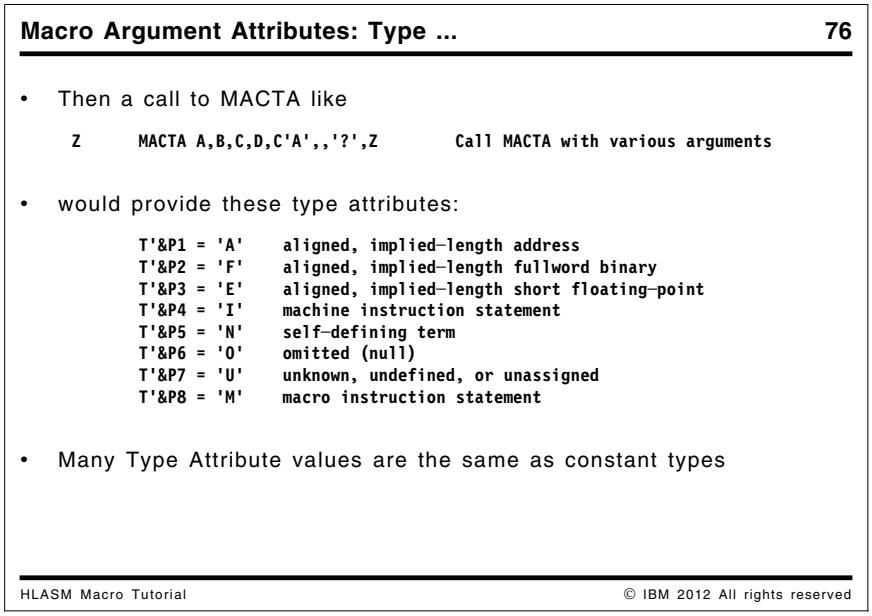
Thus, if the argument associated with the parameter &X is the five characters (A,9), then K'&X is 5 and N'&X is 2. Other attribute references probe more deeply into the possible meanings of a parameter. Thus, T'&X(1) would test what the character A might designate: if A is a label on a constant, T'&X(1) would return information about the type of the constant. If the type attribute is indeed that of a constant, then L'&X(1) would provide its length attribute. Similarly, T'&X(2)

would be N, indicating that it is a self-defining term that may be used wherever such terms are valid.

Attribute references of attribute references (such as K'L'&X) are not allowed, but it's easy to use intermediate variable symbols to determine the result.

We will now look at several attribute references in more detail. Summary information about attributes is described at "Summary of Attribute References" on page 87.





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?". It tries to answer the question "What meaning might this argument string have in the base language?" Typical uses are 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 values returned by a type attribute reference, assume these 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 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, undefined, or unassigned
         T'&P8 = 'M'    macro instruction statement
```

A type attribute reference can return any of 28 possible values.

Length, Integer, and Scale Attributes

The Length (L') attribute of a symbol refers to the implicit or explicit length of the area of storage named by the symbol. Symbols not naming areas of storage are assigned a default length attribute 1 if an explicit length value was not provided in its definition.

```
Addr     DS    A        L'Addr = 4
R5       EQU   5        L'R5 = 1
Go       BR    14       L'Go = 2
```

For symbols naming constants defined by DC and DS statements, the Integer (I'), and Scale (S') attributes can be used to test properties related to numeric scaling.

- For decimal constants of types P and Z, the Integer attribute is the number of digits to the left of an explicit or implied decimal point, and the Scale attribute is the number of digits to the right of the decimal point.
- For binary constants of types F and H, the Scale attribute is the number of bits to the right of the radix point, and the Integer attribute is the number of non-sign bits in the constant minus the Scale attribute.

  ```
  HalfW    DC    HS7'98.765432'   S'HalfW = 7, I'HalfW = 8
  FullW    DC    FS6'1.3'         S'FullW = 6, I'FullW = 25
  DblW     DC    FDS5'1.23'       S'DblW  = 5, I'DblW  = 58
  ```

- For hexadecimal floating-point constants of types E, D, and L, the Scale attribute is the number of de-normalizing zero hexadecimal digits in the fraction, and the Integer attribute is the number of hexadecimal digits in the fraction (6, 14, and 28 respectively) minus the value of the Scale attribute.

```
EConst  DC    ES2'0.8'          S'EConst = 2, I'EConst = 4
DConst  DC    DS3'0.9'          S'DConst = 3, I'DConst = 11
LConst  DC    LS5'123.4567'     S'LConst = 5, I'LConst = 23
```

- For binary floating-point constants, the Scale attribute is always zero, and the Integer attribute is the same as for a hexadecimal floating-point constant of the same length.
- For decimal floating-point constants, the Scale attribute is always zero, and the Integer attribute is the decimal precision of the operand.

The Integer and Scale attributes are used much less often in conditional than in ordinary assembly statements.

Defined Attribute

Sometimes a macro will need to define a symbol that might or might not have been previously defined, perhaps by another macro. The D' ("Defined") attribute reference returns 1 if the symbol is known to the assembler, and 0 otherwise. A typical test might be:

```
         AIF   (D'&Symbol eq 1).Known  Don't re-define if known
&Symbol  DS    A
.Known   ANOP  ,
```

If a symbol has not been encountered in the text of the program at the point where an attribute reference *other than* a Defined attribute reference is tested, the assembler will enter Lookahead Mode to search for the symbol and enter it into the symbol table if found. However, a Defined attribute reference to a symbol will not cause this Lookahead Mode scan.

Macro Argument Attributes: Count 77

- Count attribute reference (**K'**) answers:
  - "How many characters in a SETC variable symbol's value (or in its character representation, if not SETC)?"
- Suppose macro MAC8 has many positional and keyword parameters:

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

- This macro instruction would give these count attributes:

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

 K'&P1 = 1 corresponding to  A
 K'&P2 = 3                   ABC
 K'&P3 = 6                   'DEFG'
 K'&P4 = 0                   (omitted; explicitly null)
 K'&P5 = 0                   (implicitly null; no argument)
 K'&K1 = 1                   5
 K'&K2 = 0                   (null default value)
 K'&K3 = 6                   =F'25'
```

HLASM Macro Tutorial

Macro-Instruction Argument Properties: Count Attribute

A macro argument has one inherent property: the number of characters it contains. These can be determined for any argument using a 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                     (omitted; explicitly null)
K'&P5 = 0                     (implicitly null; no argument)
K'&K1 = 1                     5
K'&K2 = 0                     (null default value)
K'&K3 = 6                     =F'25'
```

When the value of a parameter is assigned to a character variable, the content of the parameter string is unchanged; the pairing rules for ampersands and apostrophes apply only to character *strings*, not to argument strings.

Macro Argument Attributes: Lists and Number Attribute **78**

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

```
Examples:        (A)   (B,C)    (D,E,,F)
No. list items:   1      2          4
```

- List items may themselves be lists, to any (reasonable) nesting depth

```
Examples:        ((A))   (A,(B,C))    (A,(B,C,(D,E,,F),G),H)
No. list items:    1      1    2       1           4  3  2
```

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

HLASM Macro Tutorial

Macro-Instruction Argument Properties: Lists and Number Attributes

It is sometimes useful to pass groups of related arguments 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, and can simplify the documentation of the macro call.

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

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

Figure 26. Sample Macro Argument List Structures

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

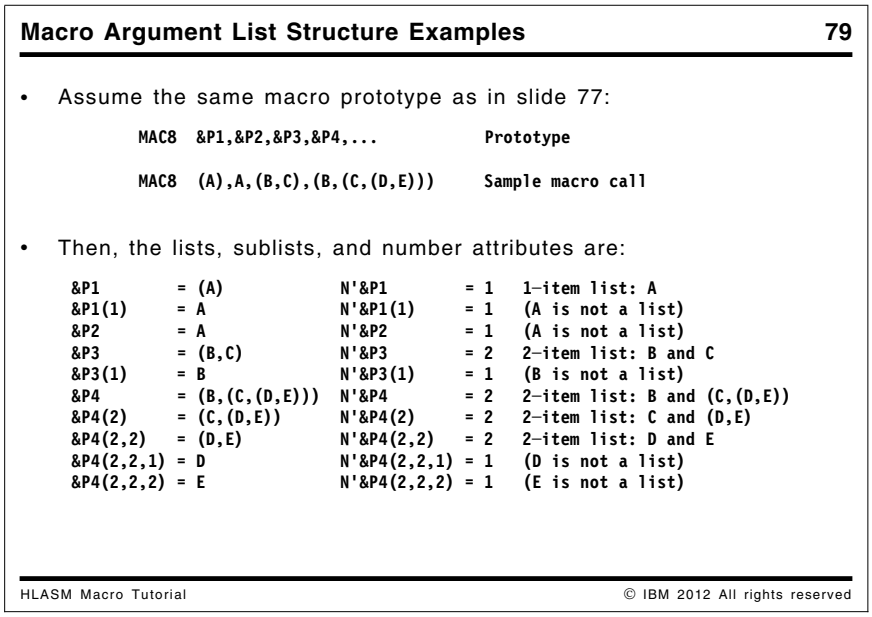
```
((A))    (A,(B,C))     (A,(B,C,(D,E,,F),G),H)
```

Figure 27. Sample Macro Argument Nested List Structures

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 1024 characters.

The number attribute reference (N') determines 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 26 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 of &Z is now 17
```



Macro-Instruction Argument Lists and Sublists

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 not null or badly formed), 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 contain more deeply nested sublists, 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)
```

A possibly confusing situation occurs when an argument is not parenthesized. In the macro call

```
MAC8  (A),A
```

the first argument is a parenthesized list with one item. Then, the number attributes and sublists of the two arguments are:

```
&P1     = (A)      N'&P1     = 1   (a 1-item list)
&P1(1)  = A        N'&P1(1)  = 1   (A is not a list)
&P2     = A        N'&P2     = 1   (A is not a list)
```

which may be unexpected. These “rules of thumb” may help you understand number attribute references to variable symbols:

1. If the variable symbol is dimensioned, its number attribute is the subscript of the highest-numbered element of the array to which a value has been assigned.
2. If the variable symbol is not dimensioned and is not a macro parameter (either explicitly named, or implicitly named as &SYSLIST(n)), its number attribute is zero.
3. If the first character of a macro argument is a left parenthesis, count the number of unquoted and un-nested commas between it and the next matching right parenthesis. That number plus 1 is the number attribute of the “list”.
4. If there is no initial left parenthesis, the number attribute is 1.

It can be important to know whether or not a list item is parenthesized, so you should test the first *and* last characters to verify that the list is enclosed in parentheses. (Some macros test only for the opening left parenthesis, assuming that the assembler will automatically enforce correct nesting of parentheses. This is not always a safe assumption.)

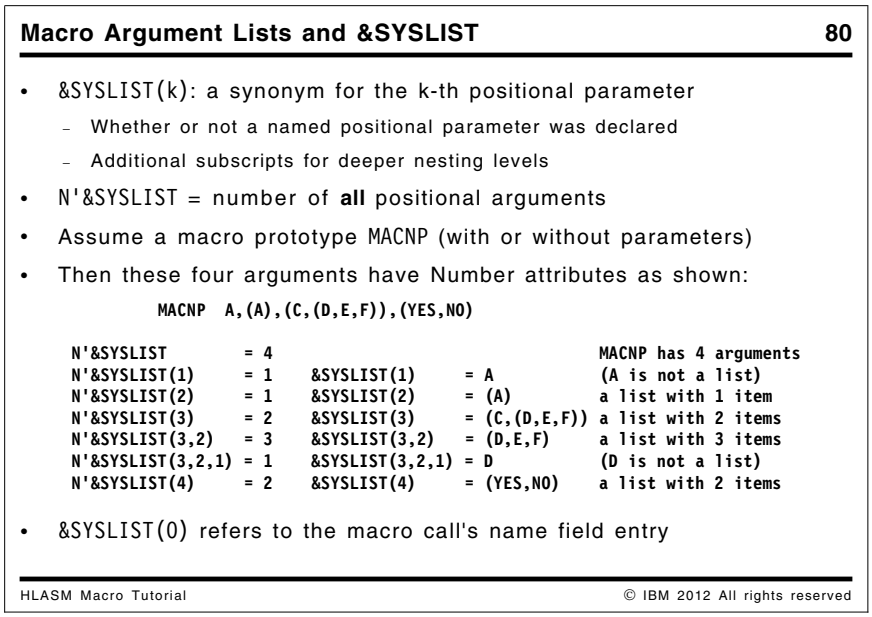
There may not be a problem if a single item is or is not enclosed in parentheses (depending on how the argument is tested and substituted). For example,

```
         LR   0,(R9)
```

and

```
         LR   0,R9
```

will be processed the same way by the assembler.



Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol

It is often useful to be able to call a macro with an indefinite number of arguments that we will process "identically" or "equivalently"; there may be no particular benefit from naming and then referring to each parameter by name.

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. Each additional subscript refers to a nested sublist; &SYSLIST(2,1,3) refers to the third element in the first element in the second positional argument. &SYSLIST(0) refers to the entry in the name field of the macro call (which need not be present). The total number of *positional* arguments in the macro instruction's operand list can be determined using N'&SYSLIST.

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 &SYSLIST references, suppose we have defined a macro named MACNP; whether or not any positional parameters are declared on the macro prototype statement 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)         a list with 1 item
N'&SYSLIST(3)     = 2    &SYSLIST(3)     = (C,(D,E,F)) a list with 2 items
N'&SYSLIST(3,2)   = 3    &SYSLIST(3,2)   = (D,E,F)     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)    a list with 2 items
```

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.

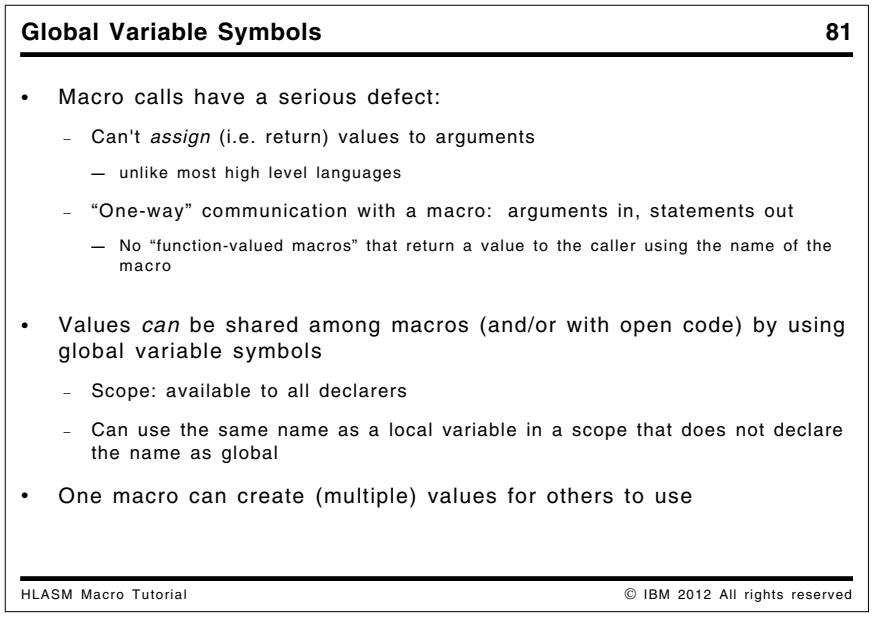
Summary of Attribute References

All eight types of attribute reference are valid in macros and conditional assembly statements; only `L'`, `I'`, and `S'` are valid in ordinary assembly statements. Other usage limitations depend on the type of the value of the reference.

Figure 28. Attribute Usage in the Base and Conditional Languages

Attr.	Val. (*)	Base Language	Conditional Language	
			Open Code	Macros
L	A	OK	OK	OK
I	A	OK for symbol types D, E, F, G, H, K, L, P, Z	OK	OK
S	A	OK for symbol types D, E, F, G, H, K, L, P, Z	OK	OK
T	C	OK in most contexts	SET symbols, symbolic parameters, system variable symbols, and ordinary symbols	
D	A	Not allowed	For ordinary symbols, SETC variables whose values are ordinary symbols, and predefined literals	
O	C	Not allowed	Note (1)	OK
N	A	Not allowed	Note (2)	Note (3)
K	A	Not allowed	OK	OK

Notes:
(*) Attribute values are arithmetic (A) or character(C)
(1) Valid only if the operand resolves to a valid symbol; not valid if the type attribute of the operand is N or O, or U (if the operand is not a valid symbol)
(2) For dimensioned variable symbols only (0 if not dimensioned).
(3) For &SYSLIST, symbolic parameters, and dimensioned variable symbols only (0 if not dimensioned).



Global Variable Symbols

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

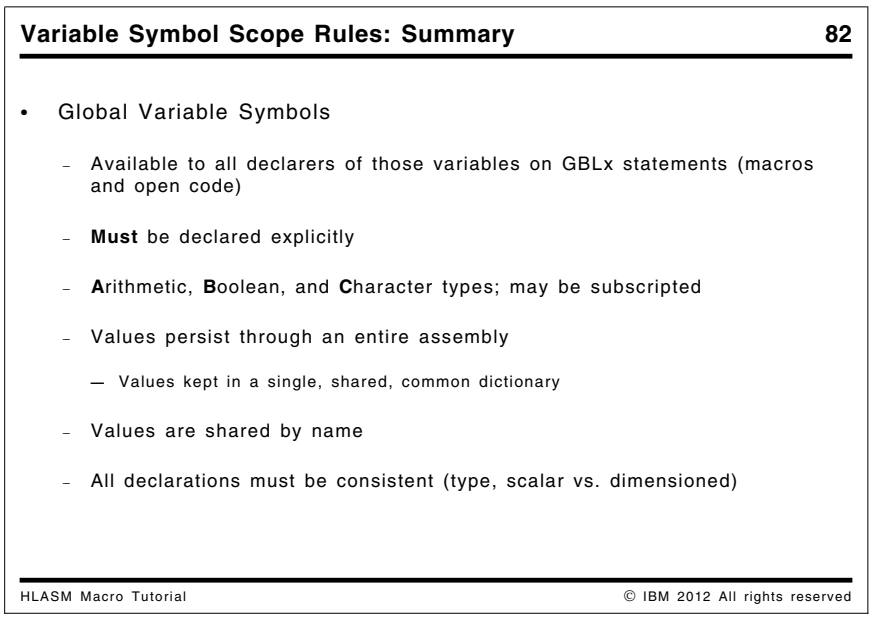
In the Assembler Language, macro calls have a serious omission: they can't *assign* (i.e. return) values to arguments, unlike most high level languages. That is, all macro arguments are "input only". Thus, communication with the interior of a macro by way of the argument list appears to be "one-way":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

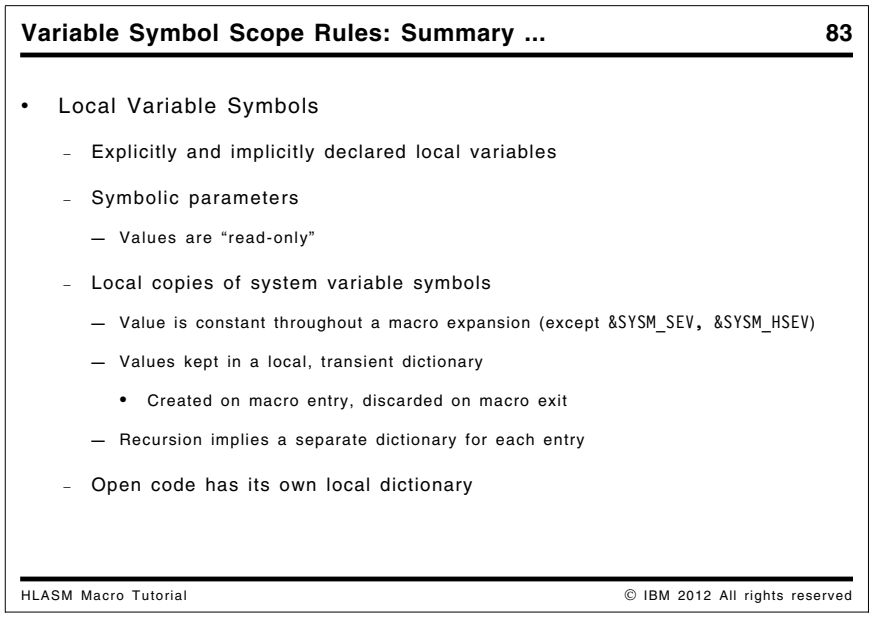
Furthermore, there is no provision for defining macros which act as "functions", macros which return a value associated with the macro name itself. This capability *is* available with external functions, but their access to global variables is severely limited (they must be passed as arguments, and their values cannot be updated by the external function).

Thus, values to be shared among macros and with open code use a different mechanism, global variable symbols. Global variable symbols are shared by and are available to all declarers. (You may 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: all declarers of external variables 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 know where to find them so they can be shared with other declarers. (Unlike some high-level languages, sharing of the assembler's global variable symbols is purely by name!)





Variable Symbol Scope Rules: Summary

Let us review and 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: **A**rithmetic, **B**oolean, and **C**haracter.
- The values of global variable symbols persist through an entire assembly, and their values are kept in a single, common dictionary. They may be referenced and set by any declarer.
- Local variable symbols include explicitly and implicitly declared variables, symbolic parameters, and system variable symbols. They are not shared with other macros, or with open code. Open code has its own local dictionary, which is active throughout an assembly. Local variable symbols may be referenced or set only in their local scope.
- System variable symbols and parameters are “read-only”, meaning that their values are constant throughout a macro invocation, and cannot be changed. The only system variable symbol whose value may change are `&SYSM_SEV` and `&SYSM_HSEV`, which depend on severity settings of MNOTE statements. (System variable symbols are summarized in “System (&SYS) Variable Symbols” on page 249.)
- Variable symbol values for macros are kept in a transient local dictionary that is created on macro entry and discarded on macro exit. Note that recursion implies a separate dictionary for each entry to the macro; every invocation has its own local, non-shared dictionary.

The following figure illustrates the use of local and global variable symbol dictionaries for local and global symbols, and for macros.

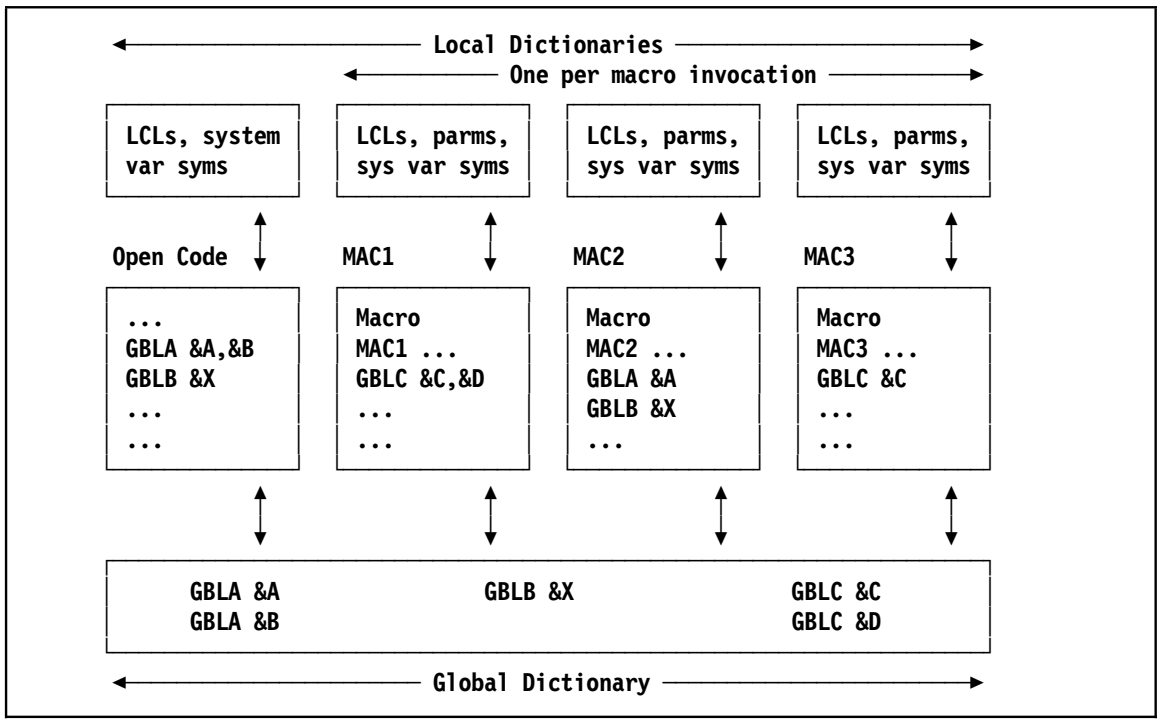


Figure 29. Example of Variable Symbol Dictionaries

The open code dictionary contains system variable symbols applicable to open code, and any local variable symbols declared in open code. Each of the macro dictionaries contains local variable symbols, parameter values, and the values of system variable symbols local to the macro, such as &SYSNDX. Finally, the global variable symbol dictionary contains all global symbols declared in open code and in any macro. Only declarers of a global variable symbol may refer to it; in Figure 29, only open code and macro MAC2 may refer to the GLBL symbol &X.

Macro Debugging Techniques **84**

- Complex macros can be hard to debug
  - Written in an poorly structured language
- Some useful debugging facilities are available:
  1. MNOTE statement
     - Can be inserted liberally to trace control flows and display values
  2. MHELP statement
     - Built-in assembler trace and display facility
     - Many levels of control; output can be quite verbose!
  3. ACTR statement
     - Limits number of conditional branches within a macro
     - Very useful if you suspect excess looping
  4. LIBMAC Option
     - Library macros appear to be defined in-line
  5. PRINT MCALL statement, PCONTROL(MCALL) option
     - Displays inner-macro calls

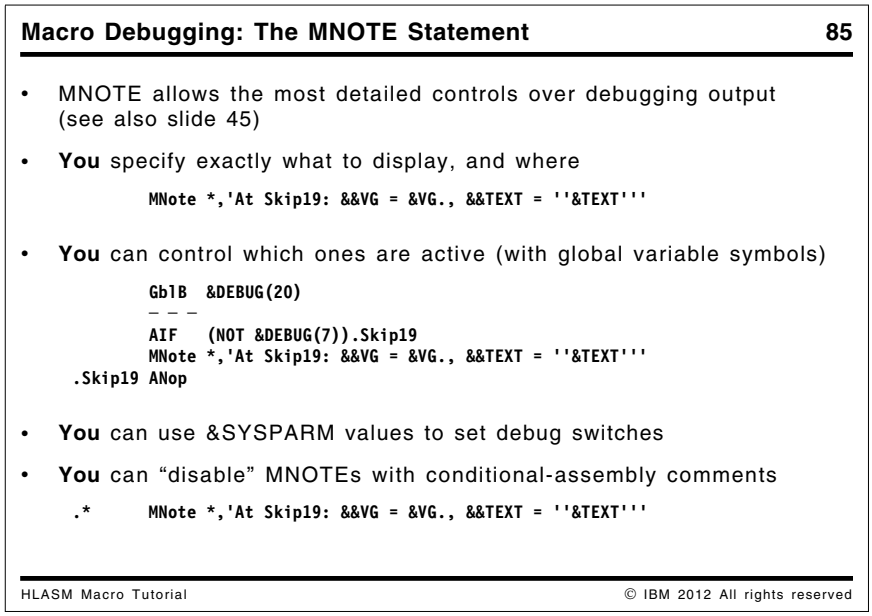
HLASM Macro Tutorial

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.

We briefly describe four statements and two options useful for macro debugging:

- the MNOTE statement
- the MHELP statement
- the ACTR statement
- the LIBMAC option
- the PRINT MCALL statement and the PCONTROL(MCALL) option.



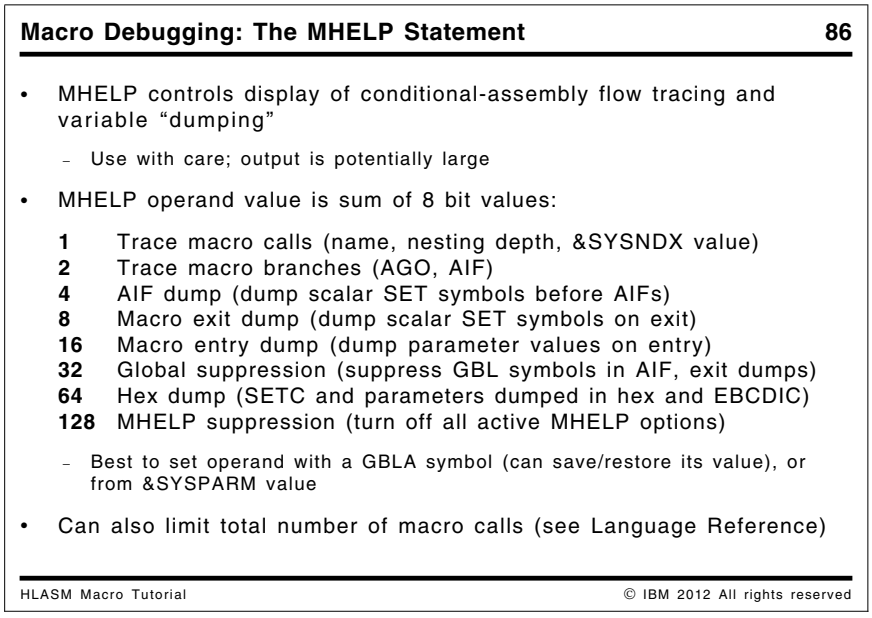
Macro Debugging: The MNOTE Statement

We have already touched on the use of MNOTE statements in "Displaying Symbol Values and Messages: The MNOTE Statement" on page 47. Their main benefits for debugging macros are:

- MNOTE statements may be placed at exactly those points where you know 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 include "commenting out" the statement (with a "**.***" conditional-assembly comment), or adding global debugging switches to control which statements will be executed:

```
        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 desired output.



Macro Debugging: 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 if you specify the LIBMAC option; tracing is then active for library macros.

4 AIF dump

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

8 Macro exit dump

MHELP 8 has the same effect as the preceding (MHELP 4), but the values are also 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,

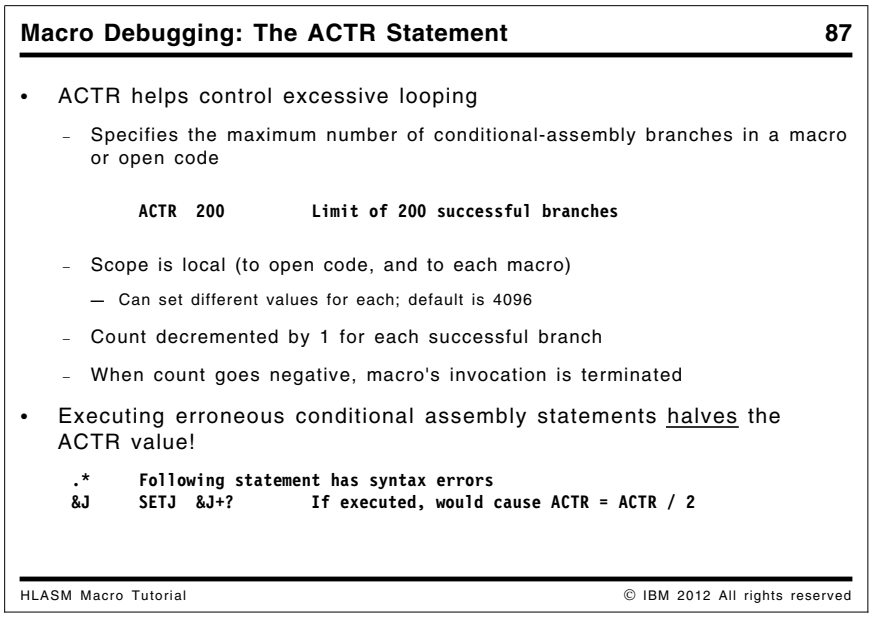
```
MHELP  1+2              Trace macro calls and AIFs
```

will enable 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 High Level Assembler for z/OS, z/VM, & z/VSE *Language Reference* manual.

The output of the MHELP statement can sometimes be large, 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, 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.



Macro Debugging: The ACTR Statement

The ACTR statement is used to 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
```

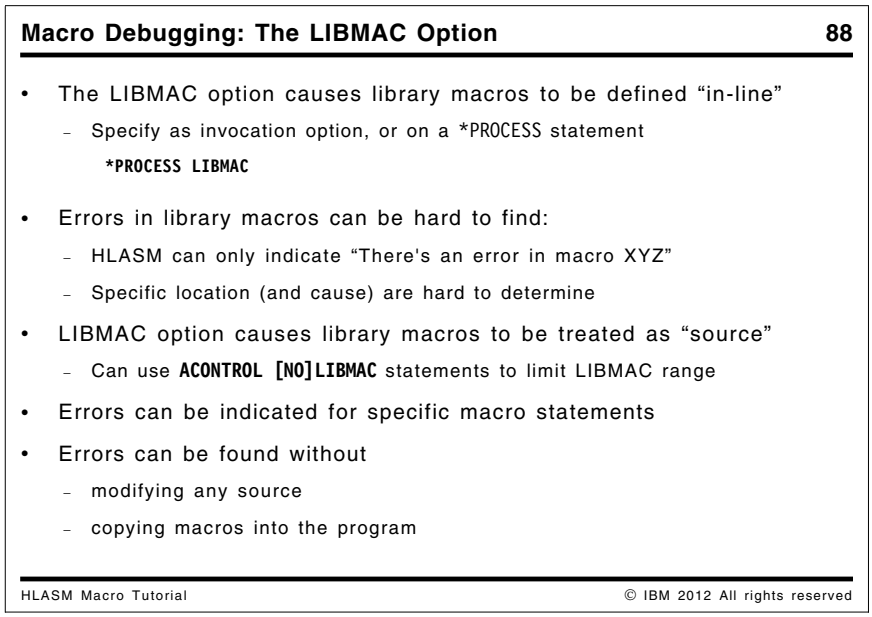
where the value of the “arithmetic_expression” will be used to set an upper limit on the number of branches executed by the assembler. In the absence of an ACTR statement, the default ACTR value is 4096, which is adequate for most macros.

ACTR is most useful in two situations:

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

If the macro definition contains errors detected during encoding, the ACTR value may be divided by 2 each time such a statement is interpreted. This helps avoid wasting resources on what will undoubtedly be a failed assembly.

The ACTR value is local to each scope, including open code. If exceeded in a macro, the expansion is terminated; if exceeded in open code, the rest of the source program is treated as comments, and is not processed. An ACTR value can be changed within its scope by executing other ACTR statements.



Macro Debugging: The LIBMAC Option

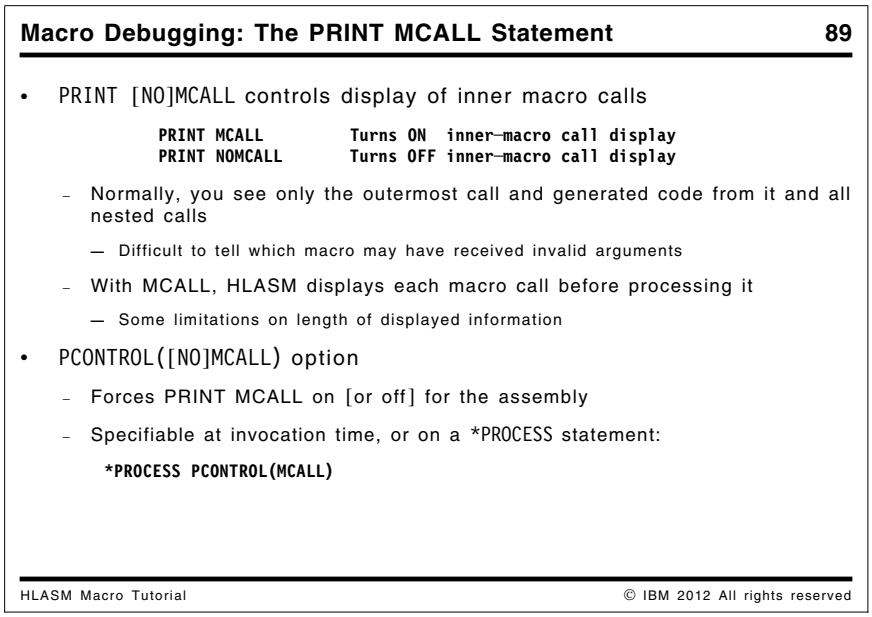
The LIBMAC assembler option is very helpful in locating errors in macros whose definitions have been placed in a macro library. Because library macros are edited as they are read, they do not have statement numbers associated with each statement of the definition. If the assembler detects errors during encoding or expansion of a library macro, it provides less precise information about the problem's causes.

The LIBMAC option causes the assembler to treat library macro definitions as though they were found in the primary source stream. When an error condition is detected, the assembler is able (in most cases) to supply the number of the relevant statement and provide more precise diagnostics. This makes locating and correcting errors much easier.

If your program calls many macros, but only one or two macros need this form of analysis, you can “bracket” the calls with ACONTROL statements to limit the range of statements during which the LIBMAC option will be in effect:

```
ACONTROL  LIBMAC     Turn LIBMAC option on
OddMacro  ...        The macro to be analyzed
ACONTROL  NOLIBMAC   Turn LIBMAC option off
GoodMac   ...        Trusted macro, analysis not needed
```

This facility would not be needed, of course, if macros could be perfectly debugged before they were placed into a macro library. Unfortunately, creators and testers of macro definitions cannot always anticipate all possible uses, so errors sometimes occur long after a macro was written and “certified”.



Macro Debugging: The PRINT MCALL Statement

The PRINT MCALL statement and the PCONTROL(MCALL) assembler option can be very helpful in locating errors in nested macro calls. Under normal circumstances, the assembler displays only the outermost macro call, and (if PRINT GEN is in effect) the code generated from that call and all nested calls.

If a complex nest of macro calls generates incorrect code, it can sometimes be difficult to isolate the problem to a specific macro, or to the interfaces among the macros. The PRINT MCALL statement causes the assembler to display inner macro calls before they are processed; this can help in ensuring that correct arguments are passed to each macro. For example, suppose you have defined two macros, OUTER and INNER:

```
         Macro
&L       OUTER  &P,&Q,&R
&T       SetC   '&P._O
&L.X     INNER  &Q,ZZ,&T
         MEnd

         Macro
&L       INNER  &F,&G,&H
         DC     C'F=&F., G=&G., H=&H'
         MEnd
```

Then, if you call the OUTER macro with the statement

```
 K       OUTER  A,B,C
```

the listing will show only the call to OUTER and (if PRINT GEN is in effect) the generated DC statement. If PRINT MCALL is in effect, the listing will also show the call to INNER:

```
 K       OUTER  A,B,C
+KX      INNER  B,ZZ,A_C
+        DC     C'F=B, G=ZZ, H=A_C'
         End
```

If macro arguments are modified when passed to inner macros (as in this example), debugging can be made much simpler if the actual arguments of the inner macro calls are visible.

The PRINT MCALL statement is subject to a "global" override through the use of the PCONTROL option, which specifies that PRINT MCALL should be active or not for the entire assembly, no matter what PRINT MCALL or PRINT NOMCALL statements may be present in the source program.

IBM Macro Libraries

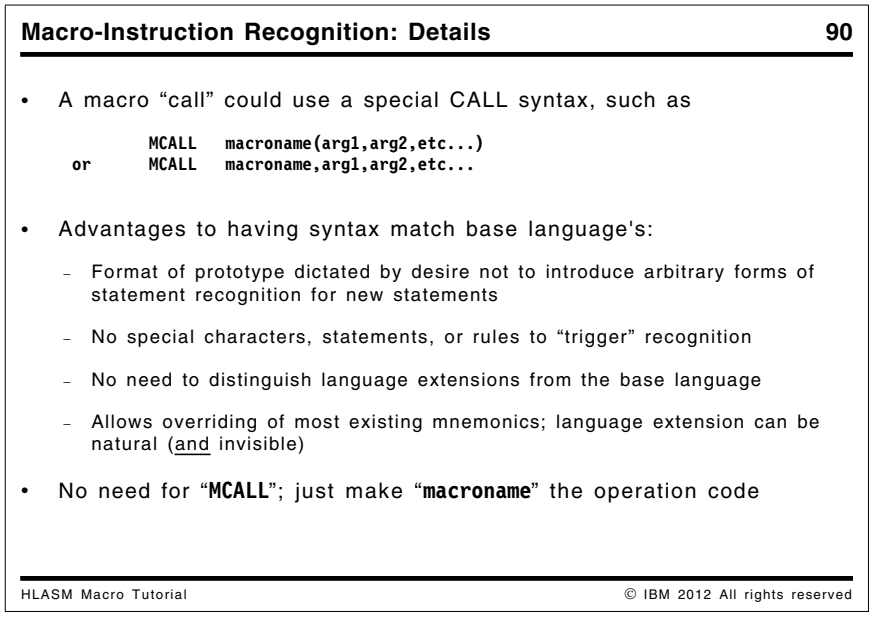
Every IBM operating system provides several macro libraries that can provide helpful examples of macro coding techniques.[6] Some macros simply set up parameters lists for calls to a system service; these tend to be less instructive than macros that generate sequences of instructions for other uses. You will probably want to defer study of very large macros until you are comfortable with reading and writing your own macro definitions.

But do remember that many IBM macros were written in the early days of System/360; the assemblers of those times were far less powerful than today's High Level Assembler, so the coding techniques may appear unnecessarily complicated by today's standards.

Macro Special Topics

Here we mention topics that sometimes arise when defining and calling macros:

- Macro-instruction recognition.
- Macro-definition encoding
- Nested macro definitions
- Constructed keyword arguments
- Macro parameter usage in model statements
- Macro argument lists and sublists



[6] Not that the coding techniques are necessarily the best; as mentioned earlier, the conditional assembly language is awkward and unfamiliar to many programmers, and was especially so in the early days when many macros were written.

Macro-Instruction Recognition: Details

Both macro name declaration (definition) and recognition have specific rules that are closely tied to the base language syntax of the Assembler Language. Some macro languages and pre-processors require special characters or syntactic forms to "trigger" the invocation of a macro. For example, an Assembler Language macro "call" could use or require a special CALL syntax, such as

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

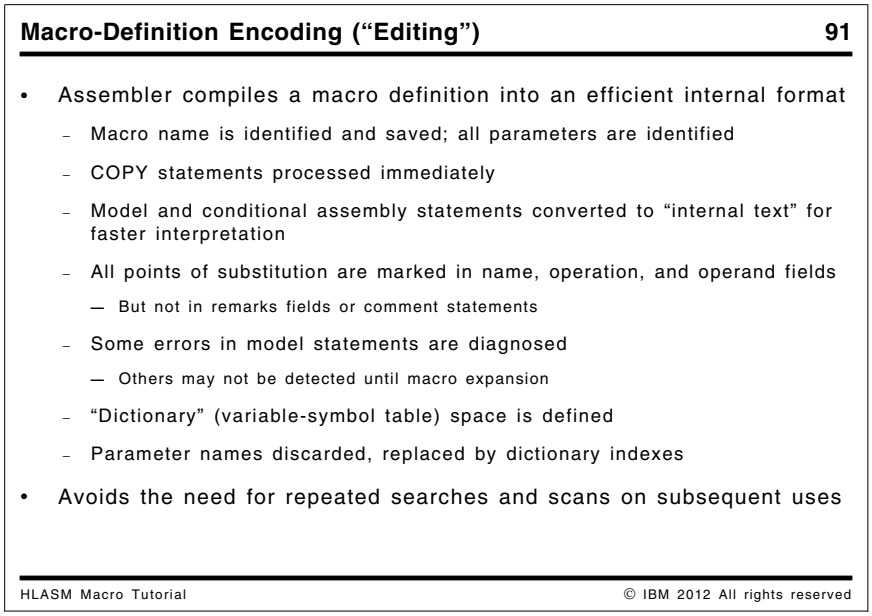
However, there are advantages to having the syntax of macro calls match the base language's, and to allow overriding of existing mnemonics; hence, we simply elide the MCALL and make the "macroname" become the operation mnemonic and the arguments become the operands of the macro instruction statement.

While many possible forms of macro definition and recognition are possible, the general format used in the Assembler Language is dictated by a desire not to introduce arbitrary forms of statement syntax and recognition rules for new statements.

The SETAF and SETCF instructions use explicit invocation:

```
         SETxF   function_name,arg1,arg2,etc...
```

in order to avoid conflicts between instruction names and external function names.



Macro-Definition Encoding

Because the assembler is designed to support extensive use of macros, their implementations reflect a need to provide efficient processing. Thus, the assembler initially converts macro definitions into an efficient encoded internal format for later use; this is sometimes called "macro editing".

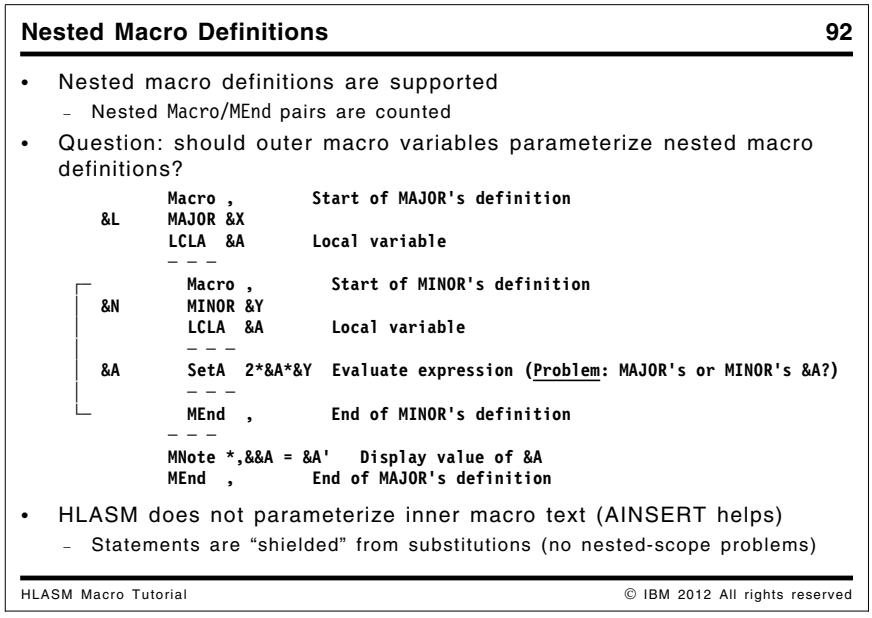
- The macro's name is identified and saved (so that later references to the macro name can be recognized as macro calls).
- All parameters are identified, and entries for them are made in a "local macro dictionary".

- Parameter and system variable symbol names are discarded, and references to them are replaced by indexes into the local macro dictionary.
- COPY statements are recognized and processed immediately. This allows common sets of declarations to be shared among macros.
- Model and conditional assembly statements are converted to "internal text" for faster interpretation.
- All points of substitution in the name, operation, and operand fields are identified and marked. (Substitutions are *not* supported in the remarks field, nor in comment statements.) Because these points of substitution 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 the value represented by `&A`.

  Note: Because generated machine instruction statements are scanned differently from generated macro instructions, you can create substitutions in remarks fields by creating an "operand" that contains the true operands, one or more blanks, and the characters of the remarks field. This technique is laborious and rarely worth the effort; using AINSERT may be easier.
- Some errors in model statements are diagnosed, but others may not be detected until macro expansion is attempted.
- "Dictionary" (symbol table) space is 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.

Some macro processors re-interpret macro definitions each time the macro is invoked. This provides greater flexibility (which is not often needed) at the expense of much slower interpretation and expansion. The design choice made in the assembler was to encode the macro for fast interpretation and expansion.



Nested Macro Definitions

Nested macro definitions are supported in the Assembler Language, but is more complicated than illustrated in the simple example in Figure 16 on page 65. To illustrate one complication, consider the following example:

```
         Macro ,        Start of MAJOR's definition
&L       MAJOR &X
         LCLA  &A       Local variable
         - - -
┌          Macro ,        Start of MINOR's definition
│ &N       MINOR &Y
│          LCLA  &A       Local variable
│          - - -
│ &A       SetA  2*&A*&Y  Evaluate expression (Problem: MAJOR's or MINOR's &A?)
│          - - -
└          MEnd  ,        End of MINOR's definition
         - - -
         MNote *,&&A = &A'   Display value of &A
         MEnd  ,        End of MAJOR's definition
```

Figure 30. Macro Definition Nesting in High Level Assembler

The variable symbol &A appears in both the MAJOR outer macro and the MINOR inner macro. Thus, the macro encoder must decide how to process occurrences of &A in the nested definition of MINOR: should they be marked as points of substitution, and by which macro should the substitutions be done, using its value of &A?

To avoid complex syntax and rules of interpretation, the assembler simply treats all statements between the `Macro` and `MEnd` statements of nested macro definitions as uninterpreted strings of text into which *no* substitutions are performed. In effect, all nested macro definitions are "shielded" from enclosing definitions. This means that a macro definition can generate a macro definition, but cannot parameterize, modify, or "tailor" it in any way. The nested definition is simply generated as a sequence of statements.

Some of the limitations imposed by this choice can be overcome by using the AINSERT statement, described in "The AINSERT Statement" on page 183.

You would get the same results if the two macros were defined independently; but this method of "packaging" one macro definition inside another can help avoid accidental invocation of a secondary, inner macro before the primary, outer macro has been called.

Constructed Keyword Arguments Do Not Work 93

- Keyword arguments *cannot* be created by substitution
- Suppose a macro prototype statement is

```
&X       TestMac  &K=KeyVal,&P1          Keyword and Positional Parameters
```

- If you construct an "apparent" keyword argument and call the macro:

```
&C       SetC     'K=What'               Create an apparent keyword
         TestMac  &C,Maybe               Call with "keyword" 'K=What'?
```

- This looks like a keyword and a positional argument:

```
 +       TestMac  K=What,Maybe           Call with "keyword"?
```

- In fact, the argument is *positional*, with value **'K=What'** !
- Macro calls are not re-scanned after substitutions!
  - The loss of generality is traded for gains in efficiency

HLASM Macro Tutorial

Constructed Keyword Arguments

You may need to construct argument lists for macro calls, and be tempted to create keyword arguments. Suppose you have written a macro with a prototype statement like this:

```
&X       TestMac  &K=KeyVal,&P1          Keyword and Positional Parameters
```

and you want to construct a keyword argument:

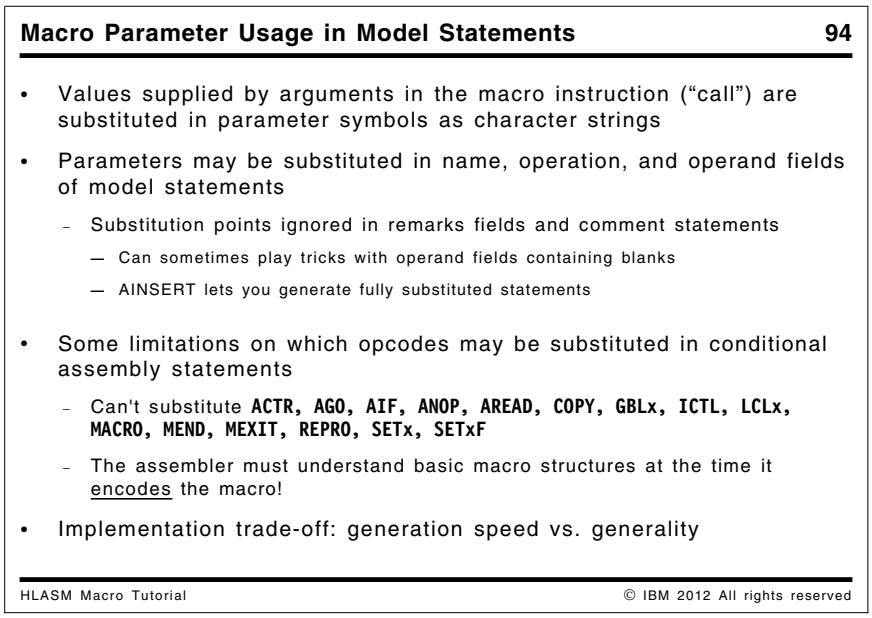
```
&C       SetC     'K=What'               Create an apparent keyword
         TestMac  &C,Maybe               Call with "keyword" 'K=What'?
```

While this appears to be a properly formed keyword argument `K=What`, it is actually treated as a *positional* argument, because the statement is not re-scanned after the value of `&C` has been substituted. The little test program shown in Figure 31 shows what happens: the substituted string is treated as a positional argument.

```
         Macro
 &X      TestMac  &K=KeyVal,&P1          Keyword and Positional Parms
         MNote    *,'P1=''&P1.'', K=''&K.''' Display values of each
         MEnd

         TestMac  Yes,K=No               Test with positional first
+*,P1='Yes', K='No'

         TestMac  K=No,Yes               Test with keyword first
+*,P1='Yes', K='No'

 &C      SetC     'K=What'               Create an apparent keyword

         TestMac  &C,Maybe               Call with 'keyword' first?
+*,P1='K=What', K='KeyVal'
```

Figure 31. Example of a Substituted (Apparent) Keyword Argument

The original design of the System/360 assembler and its successors focused on efficient macro expansion, so macro calls containing substitutions were not re-scanned.



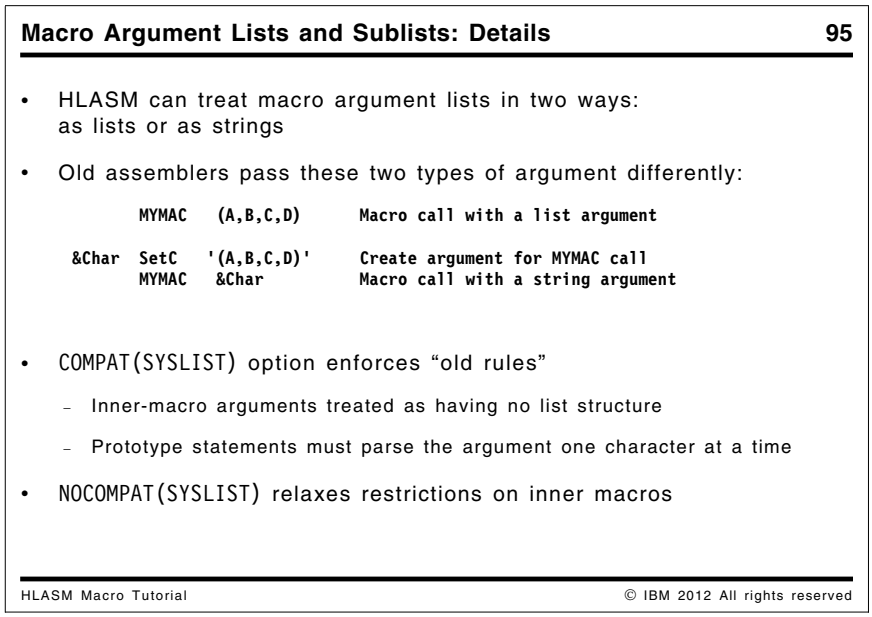
Macro Parameter Usage in Model Statements

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, operation, and operand field fields, 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 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 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!

The original implementation of the conditional assembly language assumed that macros will be used frequently, so that speed of generation was more important than complete generality. Since this conditional assembly language is more powerful than that of most macro-processors or pre-processors, the choice seemed reasonable.



Macro Argument Lists and Sublists: Details

In older assemblers, all inner-macro arguments passed as strings were treated as having no structure; that is, the operand scanner for the inner macro call recognized no list structure, even if it is present (as in case 3 on page 104). Thus, for example, a reference inside the INNER macro to (say) the length attribute of the argument would be diagnosed as invalid, because the argument would not be recognized either as a symbol or as a list. The most serious defect of this treatment is that the powerful facilities in the conditional assembly language, such as number attribute references (N') and subscripted &SYSLIST notation, could not be used be used to “parse” the operand to extract individual list elements.

The COMPAT(SYSLIST) option can affect how you handle lists of arguments passed to macros. While this is rarely a concern, there are situations where your macros can be written more simply if you can utilize the High Level Assembler's enhanced ability to handle lists.

There are two types of lists passed as arguments to macros:

1. a positional argument list, and
2. a parenthesized list of terms passed as a single argument.

For example, a list of *four* positional arguments appears in the call

```
MYMAC   A,B,C,D        Macro call with four arguments
```

and may be treated as a list using the &SYSLIST system variable symbol. A list of items passed as a *single* argument appears in the call

```
MYMAC   (A,B,C,D)      Macro call with one argument (a list)
```

where the argument (A,B,C,D) is a list of four items. We will now examine the second of these forms, where an argument is a list.

Inner-Macro Sublists

There are several ways to create and then pass arguments from an outer macro to an inner:

1. by direct substitution of an enclosing-macro's entire argument:

```
        MACRO
&L      OUTER   &A,&B,&C      Three positional parameters
        - - -
&L      INNER   &B            Pass second parameter as an argument to INNER
        - - -
        MEND
        - - -
        OUTER   R,(S,T,U),V   Passes  (S,T,U)  to INNER
```

   The second argument of OUTER is passed unchanged as the argument of INNER.

2. by substitution of parts:

```
        MACRO
&L      OUTER   &A,&B,&C
        - - -
&L      INNER   &B(1)         Pass first element of &B
        - - -
        MEND
        - - -
        OUTER   R,(S,T,U),V   Passes  S  to INNER
        OUTER   R,S,T         Passes  S  to INNER
```

   In this case, the first list element of the second argument of OUTER is passed unchanged as the argument of INNER. If the argument of the call to OUTER corresponding to the parameter &B is not a list, then the entire argument will be passed.

3. by construction as a string, in part or as a whole:

```
        MACRO
&L      OUTER   &A,&B,&C
        - - -
&T      SETC    '('.'&B'(2,K'&B-2).')'
        - - -
&L      INNER   &T            Pass parenthesized string of &B
        - - -
        MEND
        - - -
        OUTER   R,(S,T,U),V   Passes  (S,T,U)  to INNER
```

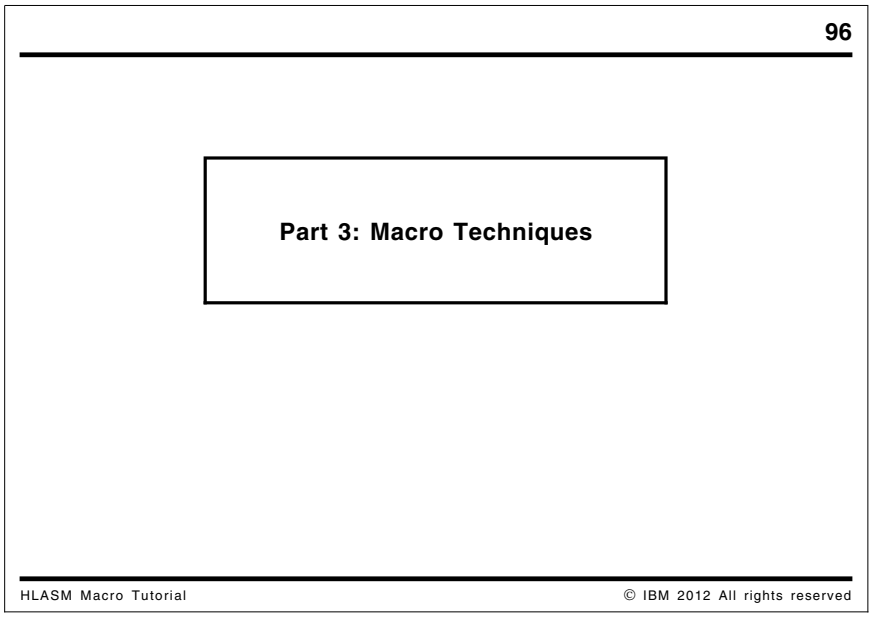
   In this case, a string variable &T is constructed, and its contents (S,T,U) is passed as the argument to INNER.

The method used can effect the recognition and treatment of arguments by inner macros. It might appear that the third example should give the same results as the first, because they both pass the argument (S,T,U) to INNER. They can be treated quite differently: is (S,T,U) a list or a string?

If you specify the COMPAT(SYSLIST) option, you must scan the argument string &Arg one character at a time to extract the needed pieces of information. Thus, macros called as inner macros may have to be much more complex than outer-level macros, because they analyze arguments one character at a time; substituted arguments to inner macros are treated as having no structure.

However, if you specify NOCOMPAT(SYSLIST), all macro arguments will be treated the same way, independent of the level of macro invocation; no distinction is made between outermost and inner macro calls. Thus, you can use the full power of the &SYSLIST notation, sublist notations, and number attributes.

Part 3: Macro Techniques



Macro instructions (or *macros*) give you a wonderfully flexible set of possibilities. Macros share many of the properties of ordinary subroutines (you can think of a macro as an assembly-time subroutine!) that can be called from many different applications: once created, they may be used to simplify 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 (for example, 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)
- generate short code sequences to convert among data types, justify numeric fields, search tables, validate data values, and other helpful tasks.

to the very complex:

- macros have been created to define "artificial languages" in which entire applications are written. Examples include
  - the SNOBOL4[7] programming language,
  - specialized compiler-writing operations[8],

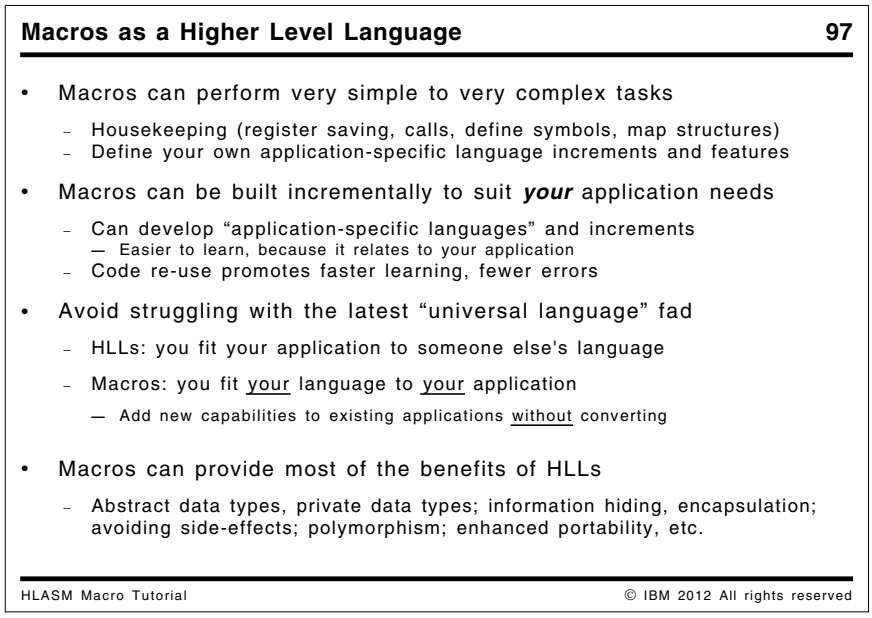
[7] See *The Macro Implementation of SNOBOL4*, by Ralph E. Griswold (Freeman & Co., San Francisco, 1972). Chapter 10 describes the macro techniques used.

[8] The IBM Fortran G-Level compiler was written in an assembler macro language that allowed it to be quickly and easily ported to other systems.

- insurance[9],
- teleprocessing[10]

and banking, marketing, and other applications.

Our purpose here is to show that you can write macros to simplify almost any programming task, from the simple and small to the very complex and powerful.



Macros many advantages:

- Macros can perform as much or as little as is needed for a particular task.
- Macros can be built incrementally; 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 be better suited to its needs than any general-purpose "higher level" language designed to (nearly) fit (nearly) everything. When completed, a macro can be used by everyone, giving immediate benefits of code re-use.
- Macro-based implementations are often more efficient than compiled code. A compiler must accept arbitrary combinations of statements, and then attempt to optimize them; a macro-based language can concentrate on just those parts of the application for which optimization efforts are justified.
- A macro-based language is *your* language! You need not adapt the expression of your application to fit the peculiarities and rigidities of a particular language or compiler (or of a language designer's pet theories). You can select whatever language features are appropriate and useful for your application.

---

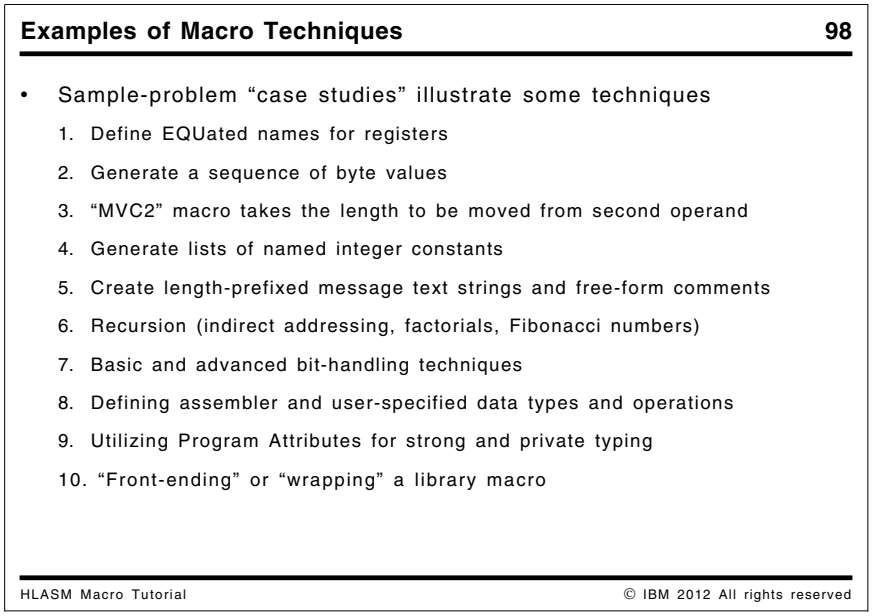
[9] Allstate Insurance implemented many of their applications in an easy-to-learn macro language that required much less training than traditional HLLs.

[10] The SPRINT system was originally developed by the Southern Pacific railroad for communicating within its rail system. It is described in *Total Operations Processing System for the Southern Pacific Company*, IBM Data Processing Application, E20-0310.

- Macro-based languages are usually easier to learn, because they relate better to your business needs than general-purpose languages.[11]
- Macros can also provide an excellent introduction to language and compiler concepts, in a controllable way. You can create and analyze generated code immediately, and can build useful language fragments without having to worry about extensive side-effects. Macros also allow you to investigate tradeoffs involved in compile-time vs. run-time issues such as a choice between generating in-line code or calls to a run-time library.

Higher-level languages are often deemed useful because they provide desirable "advanced" features. Macros can also deliver most of these features, and many more:

- *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; it is a natural consequence of macro programming.
- *Information Hiding* helps hide the details of an implementation from the user. The concept of separating application logic from data representations is an old and well established programming principle. This also is a natural and normal benefit of macro programming.
- *Private Types* are user-defined data types for which details of the implementing procedures are hidden.
- *Avoiding Side-Effects* is achieved by having functions only return a value without altering input values or setting shared variables not declared in the invocation of a procedure.
- *Polymorphism* allows functions to accept arguments of different types, and enhances component reuse in other contexts.



[11] Many studies have shown that the value of an employee depends more on knowledge of the business than on knowledge of a programming language.

Macro Techniques Case Studies

We now examine some sample case studies that illustrate aspects of the macro language, and provide various types of useful functions.

1. "Case Study 1: Defining Equated Symbols for Registers" on page 110 generates 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. (This case study is a generalization of the macro discussed at "Example 1: Define Equated Symbols for Registers" on page 71.)

2. Two example macros at "Case Study 2: Generating a Sequence of Byte Values" on page 115 generate a sequence of byte values. They illustrate conditional assembly statements, and some simple string-handling operations. (This case study is a generalization of the macro discussed at "Example 2: Generate a Sequence of Byte Values (BYTESEQ1)" on page 78.)

3. The standard MVC instruction derives its implied length from the length attribute of the first (receiving) operand. The "MVC2" macro in "Case Study 3: 'MVC2' Macro Uses Source Operand Length" on page 118 takes its implied length from the length attribute of its *second* (source) operand.

4. The INTCONS macro at "Case Study 4: Generate a List of Named Integer Constants" on page 120 generates a list of constants from a varying-length list of arguments, using &SYSLIST to refer to each argument and construct a name for each constant using its value.

5. "Case Study 5: Using the AREAD Statement" on page 123 illustrates two uses of the AREAD statement:

   a. Three macros at "Case Study 5a: Creating Length-Prefixed Message Texts" on page 124 show how to generate a length-prefixed string. The first and second examples illustrate 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.

   b. "Case Study 5b: Block Comments" on page 130 shows how to use the AREAD statement to write free-form or "block" comments in your program.

6. Three macros at "Case Study 6: Macro Recursion" on page 132 illustrate recursive macro calls. The first two illustrate the use of global variable symbols and recursive macro calls to generate factorials and Fibonacci numbers. The third implements a form of indirect addressing.

7. Two styles of macros illustrate techniques that define a private "bit" data type that let you address bits by name, and do type checking of the bit names. After describing some basic bit-handling techniques, simple and optimized macros are created:

   a. "Case Study 7a: Bit-Handling Macros -- Simple Forms" on page 142 describes basic forms of declaring and using a bit data type.

   b. "Case Study 7b: Bit-Handling Macros -- Advanced Forms" on page 151 shows how to improve the basic forms for safety and efficiency, and to generate optimized code.

8. The macros illustrated at "Case Study 8: Utilizing The Assembler's Data Types" on page 173 describe techniques that can be used to implement type-sensitive operations ("polymorphism"), and to declare user-defined data types and user-defined operations on them, with type checking and information hiding.

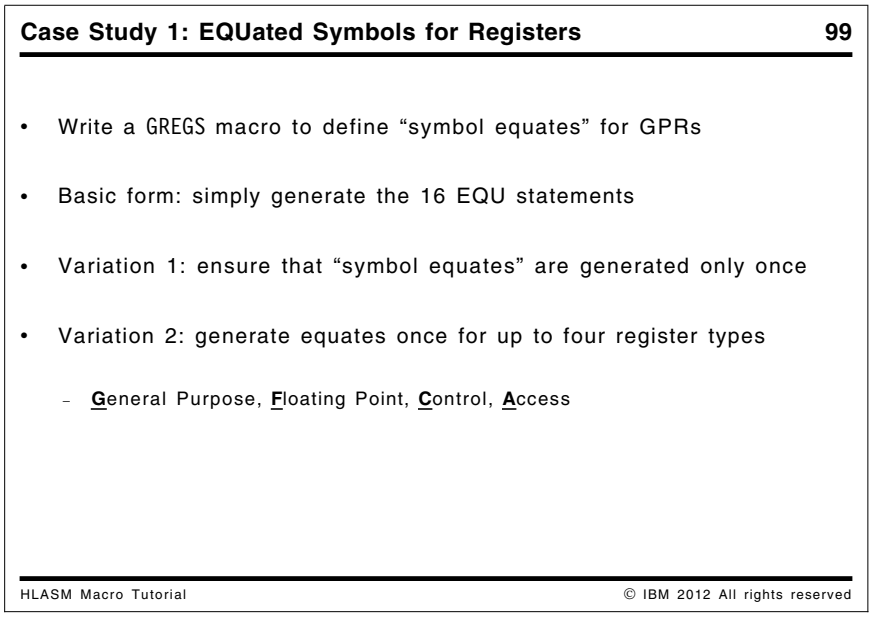
   a. "Case Study 8a: Type Sensitivity with Simple Polymorphism" on page 174 shows how the assembler's type attributes can be used to generate tailored code sequences to the types of operands.

   b. "Case Study 8b: Instruction-Operand Type Checking" on page 179 shows how user-assigned type attributes can be used to perform type checking "conformance" between instructions, operands, and registers.

c. "Case Study 8c: Encapsulated Abstract Data Types" on page 191 describes a simple application-specific language to show how user-defined data types and operations can "encapsulate" the details of data definitions and low-level operations on the data objects.

9. This case study shows how you can use Program Attributes to assign application-specific *extended* attributes to symbols, and how to use those attributes to check for consistent use and to generate code sequences with greater flexibility than high-level language provide.

10. Sometimes it is useful to be able to capture and analyze the arguments passed to another macro and post-process its results, while still using the original macro definition for its intended purposes. This is called "front-ending" or "wrapping" a macro. "Case Study 10: "Front-Ending" a Library Macro" on page 231 illustrates a way to do this.



Case Study 1: Defining Equated Symbols for Registers

The technique illustrated in "Example 1: Define Equated Symbols for Registers" on page 71 is quite acceptable unless we need to combine multiple code segments, each of which may contain a call to GREGS (as 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?

Define General Register Equates (Simply) 100

- Define “symbol equates” for GPRs with this macro (see slide 67)

```
         MACRO
         GREGS
GR0      Equ   0
GR1      Equ   1
.*       — — —  etc.
GR15     Equ   15
         MEND
```

- Problem: what if two code segments using the GREGS macro are combined?
  - If each calls GREGS, could have duplicate definitions
  - How can we preserve modularity, and define symbols only once?
- Answer: use a global variable symbol **&GRegs**
  - Value is available across all macro calls

Define General Register Equates (Safely) 101

- Initialize **&GRegs** to “false”; set to “true” when EQUs are generated

```
         MACRO
         GREGS
         GBLB    &GRegs          &GRegs initially 0 (false)
         AIF     (&GRegs).Done   Check if &GRegs already true
         LCLA    &N              &N initially 0
.X       ANOP    ,
GR&N     Equ     &N
&N       SETA    &N+1            Increment &N by 1
         AIF     (&N LE 15).X    Test for completion
&GRegs   SetB    (1)             &GRegs true (definitions have been done)
         MEXIT
.Done    MNOTE   0,'GREGS previously called, this call ignored.'
         MEND
```

- If **&GRegs** is **true**, no statements are generated

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


The solution is simple: use a global variable symbol whose value will indicate that the GREGS macro has been called already, as illustrated in Figure 32 on page 112.

```
        MACRO
        GREGS
        GBLB    &GRegs          &GRegs initially 0 (false)
        AIF     (&GRegs).Done   Check if &GRegs already true
        LCLA    &N              &N initially 0
.X      ANOP    ,
GR&N    Equ     &N
&N      SETA    &N+1            Increment &N by 1
        AIF     (&N LE 15).X    Test for completion
&GRegs  SetB    (1)             Indicate definitions have been done
        MEXIT
.Done   MNOTE   0,'GREGS previously called, this call ignored.'
        MEND

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

Figure 32. Macro to define general purpose registers once only

Defining Register Equates Safely: Pseudo-Code 102

- Allow declaration of multiple register types on one call:
  Example: REGS $type_1$[,$type_2$]... as in REGS G,F
- Pseudo-code:

```
IF (number of arguments is zero) EXIT
FOR each argument:
   Verify valid register type &T (values A, C, F, or G):
      IF invalid, ERROR EXIT with message
   IF (that type already done) Give message and ITERATE
   Generate equates
   Set appropriate 'Type_Done' flag and ITERATE
```

- 'Type_Done' flags are global Boolean variable symbols
  - Use *created variable symbols* **&(&T.Regs_Done)**
- If **&(&T.Regs_Done)** is **true**, no statements are generated

```
        REGS   G,F,A,G     G registers will not be defined twice!
```

HLASM Macro Tutorial

Now we define a macro which will create equates for *all* register types we might use in our program: General Purpose, Floating Point, Control, and Access. Rather than write separate macros (one for each type of register), we can write a single REGS macro whose operands specify the types of register desired: “G” for GPRs, “F” for FPRs, “C” for CRs, and “A” for ARs. Its syntax is:

REGS $type_1$[,$type_2$]... One or more register types

as in

```
REGS  G,F
```

A pseudo-code sketch of the techniques used is the following:

```
IF (number of arguments is zero) EXIT
FOR each argument:
   Verify valid register type &T (values A, C, F, or G):
      IF invalid, ERROR EXIT with message
   IF (that type already done) Give message and ITERATE
   Generate equates
   Set appropriate 'Type_Done' flag and ITERATE
```

Define All Register Equates Safely 103

- Improved REGS macro, to generate EQUates for four register types
  - Basic form; error checking not shown in this slide

```
            MACRO
            REGS
   &J       SetA    1                Initialize argument counter
.GetArg     ANOP
   &T       SetC    (Upper '&SysList(&J)')     Pick up an argument
            GBLB     &(&T.Regs_Done)  Declare global variable symbol
            AIF     (&(&T.Regs_Done)).NewArg  Test if true already
   &N       SetA    0                Initialize register number
.Gen        ANop    ,                Generate Equ statements
   &T.R&N   Equ     &N
   &N       SetA    &N+1
            AIf     (&N le 15).Gen
&(&T.Regs_Done) SetB (1)   Indicate definitions have been done
.NewArg     ANop  ,                   Check next argument
   &J       SetA    &J+1              Count to next argument
            AIF     (&J le N'&SysList).GetArg   Get next argument
   .Exit    MEND
```

HLASM Macro Tutorial

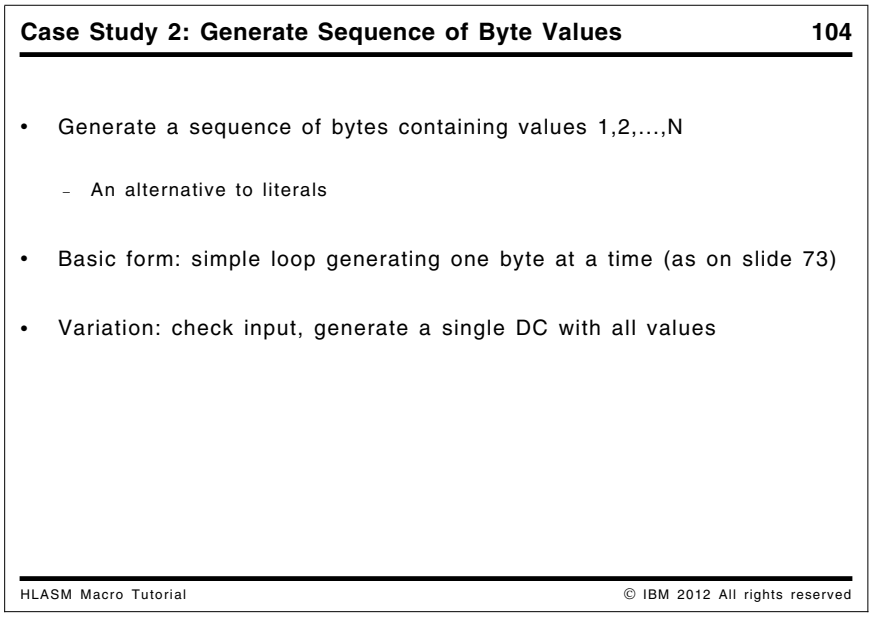
The following example uses the technique illustrated in Figure 32 on page 112 above, and generalizes it by using a “created variable symbol” to select the name of the proper global variable symbol. It also uses two “built-in” internal functions: UPPER and INDEX.

```
        MACRO
        REGS
        AIF     (N'&SysList eq 0).Exit   Quit if no arguments
&J      SetA    1                 Initialize argument counter
.GetArg ANOP
&T      SetC    (Upper '&SysList(&J)')     Pick up an argument
&N      SetA    Index('&T','ACFG')         Check type
        AIF     (&N eq 0).Bad     Error if not a supported type
        GBLB     &(&T.Regs_Done)  Declare global variable symbol
        AIF     (&(&T.Regs_Done)).Done Test if true already
&N      SetA    0                 Initialize register number
.Gen    ANop    ,                 Generate Equ statements
&T.R&N  Equ     &N
&N      SetA    &N+1
        AIf     (&N le 15).Gen
&(&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,'&SysMac.: Unknown type ''&T.''.'
        MEXIT
.Done   MNOTE   0,'&SysMac.: Previously called for type &T..'
        AGO     .Next
.Exit   MEND
```

Figure 33. 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, &CRegs_Done, or &GRegs_Done appear elsewhere in the program.

You may be interested in modifying the REGS macro to use the Defined Attribute described on page 82 to determine whether a given type of symbol has been generated.



Case Study 2: Generating a Sequence of Byte Values

Generating a Byte Sequence: BYTESEQ1 Macro **105**

- BYTESEQ1 generates a separate DC statement for each value (compare slides 46 and 73)

```
         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

 BSeq1a  BYTESEQ1  3
+BSeq1a  DC    AL1(1)
+        DC    AL1(2)
+        DC    AL1(3)
```

HLASM Macro Tutorial

The sample BYTESEQ2 macro illustrated in Figure 34 on page 117 uses the same techniques as the conditional-assembly examples given in Figure 9 on page 49 and Figure 10 on page 50. and the corresponding BYTESEQ1 macro illustrated in Figure 25 on page 78.

Generating a Byte Sequence: BYTESEQ2 Pseudo-Code **106**

- Generate a single DC statement, creating a string of bytes with binary values from 1 to N
  - N has been previously defined as an absolute symbol

```
IF (N not self-defining) ERROR EXIT with message

IF (N > 255) ERROR EXIT with too-big message

IF (N ≤ 0) EXIT with notification

Set local string variable S = '1'
DO for K = 2 to N
   S = S || ',K'          (append comma and next value)
GEN (label  DC  AL1(S) )
```

- Compare to slide 47

HLASM Macro Tutorial

A pseudo-code sketch of the macro implementation is:

```
IF (N not self-defining) ERROR EXIT with message

IF (N > 255) ERROR EXIT with too-big message

IF (N ≤ 0) EXIT with notification

Set local string variable S = '1'
DO for K = 2 to N
   S = S || ','K         (append comma and next value)
GEN (label  DC  AL1(S) )
```

Generating a Byte Sequence (BYTESEQ2) **107**

```
         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

BSeq2b   BYTESEQ2  1
BSeq2c   BYTESEQ2  5
```

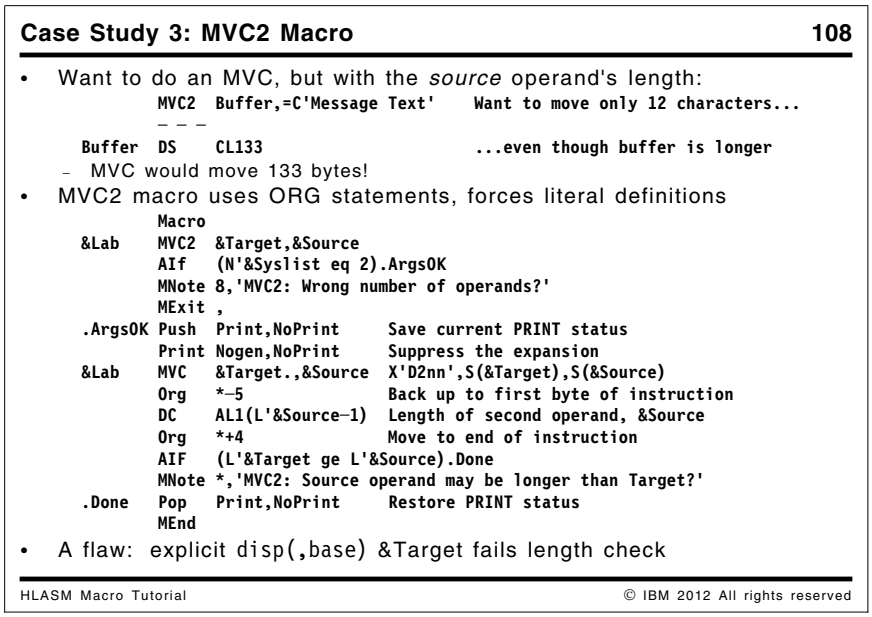
HLASM Macro Tutorial

The BYTESEQ2 macro shown in Figure 34 on page 117 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 limitation: can you tell what it is?

```
        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
BSeq2a  BYTESEQ2  0
BSeq2b  BYTESEQ2  1
BSeq2c  BYTESEQ2  5
BSeq2d  BYTESEQ2  X'58'
BSeq2e  BYTESEQ2  256
```

Figure 34. Macro to define a sequence of byte values as a single string

The argument to BYTESEQ2 may not exceed 255; otherwise the DC values are truncated at 8 bits and start at 0 again! (In earlier assemblers, the macro would fail if &S exceeds 255 characters, which limited the argument to a maximum value 88.)



Case Study 3: 'MVC2' Macro Uses Source Operand Length

Sometimes it is useful to determine the length byte of an MVC instruction from the length attribute of the *second* operand, rather than of the first. That is, rather than write clumsy and error-prone statements like

```
MVC   Buffer(L'=C'Message Text'),=C'Message Text'
```

you would rather write something like

```
MVC2  Buffer,=C'Message Text'
```

and get the same result. This can be done with an MVC2 macro with prototype

```
MVC2  &Target,&Source
```

where the macro effectively generates

```
MVC   &Target(L'&Source),&Source
```

This might not work as simply as it is written: the most difficult situation (and often the most useful!) occurs when a literal is used as the source operand.

```
        Macro
&Lab    MVC2  &Target,&Source
        AIf   (N'&Syslist eq 2).ArgsOK
        MNote 8,'MVC2: Wrong number of operands?'
        MExit ,
.ArgsOK Push  Print,NoPrint     Save current PRINT status
        Print Nogen,NoPrint     Suppress the expansion
&Lab    MVC   &Target.,&Source  X'D2nn',S(&Target),S(&Source)
        Org   *-5               Back up to first byte of instruction
        DC    AL1(L'&Source-1)  Length of second operand, &Source
        Org   *+4               Move to end of instruction
        AIF   (L'&Target ge L'&Source).Done
        MNote *,'MVC2: Source operand may be longer than Target?'
.Done   Pop   Print,NoPrint     Restore PRINT status
        MEnd
```

Figure 35. MVC2 macro definition

This macro is useful in many situations, but it has one flaw: if you write the first (target) operand as an explicit address, such as `displacement(,base)` the second AIF test for possibly moving too much data will cause an error, because the `displacement` operand is unlikely to provide a usable length attribute.

An example of the instruction generated by the MVC2 macro is:

```
0004C8 D278 F031 F0C6 ...       MVC2  Buff,=C'Error: '
                      ...
000765                ...  Buff DS   CL133
                      ...
0008AD 60C5999996997A40             =C'-Error: '
```

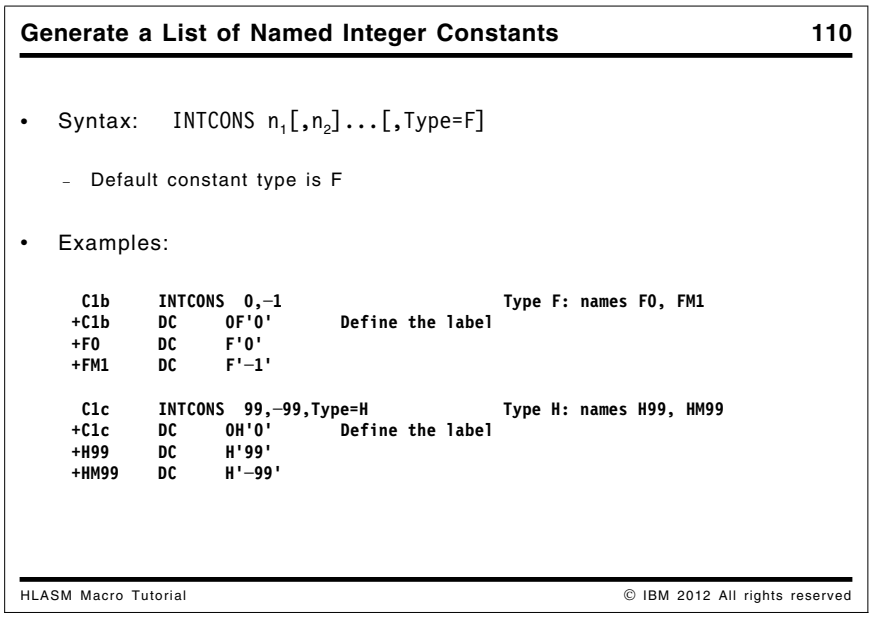
Case Study 4: Generate Named Integer Constants 109

- Intent: generate a list of “intuitively” named halfword or fullword integer constants
  - An alternative to using literals
- For example:
  - Fullword constant “1” is named **F1**
  - Halfword constant “−1” is named **HM1**

HLASM Macro Tutorial

Case Study 4: Generate a List 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 named halfword or fullword integer-valued constants that might be preferable to using literals. Their names are formed by prefixing their value to a letter designating their type: F if the value is nonnegative, or FM if the value is negative. We also provide a keyword parameter to specify their type, either F or H, with default F. Negative halfword constants will then start with the letters HM.



The macro syntax might look like this:

```
INTCONS n1[,n2]...[,Type=F]
```

If we write

```
INTCONS 1,-1
```

the macro generates these statements:

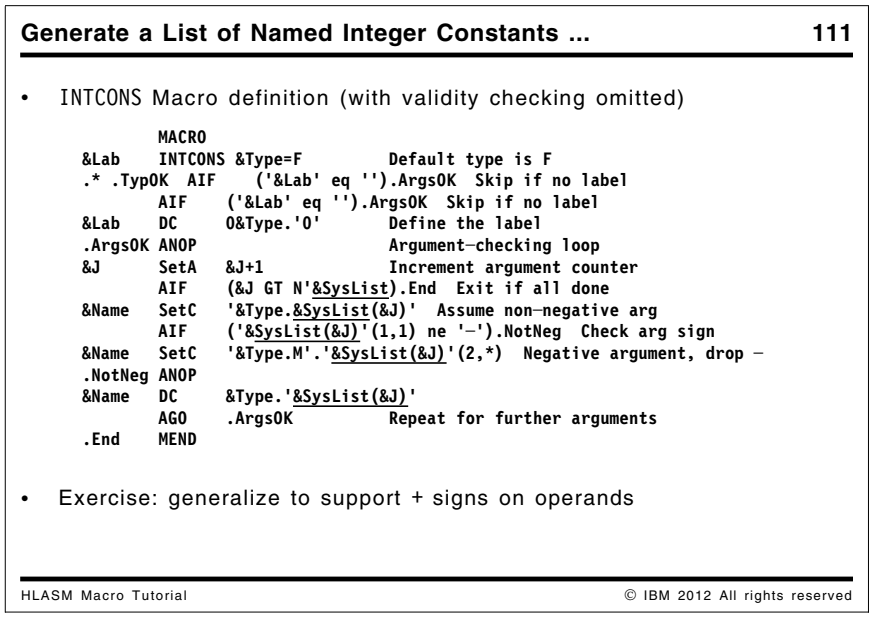
```
F1      DC    F'1'
FM1     DC    F'-1'
```

Similarly, if we write

```
INTCONS 2,-2,Type=H
```

then the macro generates

```
H2      DC    H'2'
HM2     DC    H'-2'
```



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

Then, beginning at sequence symbol .ArgsOK, the &SYSLIST system variable symbol steps through the positional arguments in turn, using subscript &J to indicate which positional argument is selected. The argument is checked for being not 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.

```
        MACRO
&Lab    INTCONS &Type=F         Default type is 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     O&Type.'O'       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
.*    end of first part... (continued...)
```

Figure 36 (Part 1 of 2). Macro parameter-argument association example: create a list of constants

```
.*    Second part: argument-checking loop, code generation
.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 nonnegative arg
        AIF    ('&SysList(&J)'(1,1) ne '-').NotNeg  Check arg sign
&Name   SetC   '&Type.M'.'&SysList(&J)'(2,*)  Negative argument, drop -
.NotNeg ANOP
&Name   DC     &Type.'&SysList(&J)'
        AGO    .ArgsOK          Repeat for further arguments
.End    MEND
```

Figure 36 (Part 2 of 2). Macro parameter-argument association example: create a list of constants

Some test cases for the INTCONS macro are shown in the following figure. The last two sets test unusual conditions, and the others show statements generated by the macro.

```
 C1b     INTCONS  0,-1                       Type F: names F0, FM1
+C1b     DC     0F'0'       Define the label
+F0      DC     F'0'
+FM1     DC     F'-1'

 C1c     INTCONS  99,-99,Type=H              Type H: names H99, HM99
+C1c     DC     0H'0'       Define the label
+H99     DC     H'99'
+HM99    DC     H'-99'

 * Test cases -- first has no label, no arguments; second has no arguments
         INTCONS
 C1a     INTCONS

 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 37. Macro example: List-of-constants test cases

As an exercise: how would you add a test to verify that each argument is a valid self-defining term? Are negative arguments then valid? Would the argument 10E7 be valid? (It's acceptable as a nominal value in an F-type constant.) Another exercise is to modify the macro to handle plus (+) signs on the numeric values.

Note: on modern processors, it's much better to use instructions with immediate operands than to use constants in storage with the same values.

Case Study 5: Using the AREAD Statement 112

1. Case Study 5a: Generate strings of message text
   - Prefix the string with an “effective length” byte (length-1)
   - Basic form: count characters
   - Variation 1: create an extra symbol, use its length attribute
   - Variation 2: use the AREAD statement and conditional-assembly functions to support “readable” input
2. Case Study 5b: Block comments
   - Write free-form text comments (without * in column 1)
   - Lets you include the documentation with the source program, so you can update both at the same time

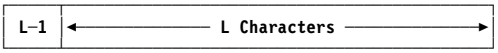
HLASM Macro Tutorial

Case Study 5: Using the AREAD Statement

This case study illustrates the power of the AREAD statement. The first set of examples shows how to create strings (such as messages) prefixed by an effective-length field, and the second example illustrates a technique for entering “block comments” into your source program.

Case Study 5a: Create Length-Prefixed Message Texts 113

- Problem: want messages prefixed with “effective length” field



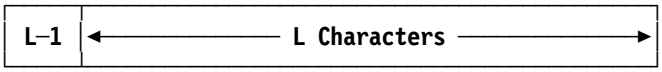
- Nobody should *ever* have to count characters!
  - Let the assembler do it for you!
- How such prefixed strings might be used:

```
  HWMsg   PFMSG 'Hello World'       Define a sample message text
 +HWMsg   DC    AL1(10),C'Hello World'   Length-prefixed message text
          - - -
          - - -
          LA    2,HWMsg             Prepare to move message to buffer
          IC    1,0(,2)             Effective length of message text
          EX    1,MsgMove           Move message to output buffer
          - - -
  MsgMove MVC   Buffer(*-*),1(2)    Executed to move message texts
```

HLASM Macro Tutorial

Case Study 5a: Creating Length-Prefixed Message Texts

A common need in applications is to produce messages. 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" (its true length minus one), as shown:



Such a length-prefixed message text could be used in code sequences like the following. The message is declared using a PFMSG macro, which generates a length byte followed by the message text:

```
 HWMsg   PFMSG 'Hello World'       Define a sample message text
+HWMsg   DC    AL1(10),C'Hello World'   Length-prefixed message text
```

Then, this small "data structure" could be used by instructions like these to move the message text to a buffer:

```
         LA    2,HWMsg             Prepare to move message to buffer
         - - -
         IC    1,0(,2)             Effective length of message text
         EX    1,MsgMove           Move message to output buffer
         - - -
MsgMove MVC   Buffer(*-*),1(2)    Executed to move message texts
```

We will illustrate three macros to create message texts with an effective-length prefix, each using progressively more powerful techniques.

Create Length-Prefixed Messages (1) 114

- PFMSG1: length-prefixed message texts

```
        MACRO
 &Lab    PFMSG1 &Txt
 .*    PFMSG1 — requires that the quoted text of the message, &Txt,
 .*    contain no embedded apostrophes (quotes) or ampersands.
         LclA   &Len             Effective Length
 &Len    SetA   K'&Txt-3         (# text chars)-3 (2 quotes, eff. length)
 &Lab    DC     AL1(&Len),C&Txt
         MEND
```

- Limited to messages with no quotes or ampersands

```
 M1a     PFMSG1 'This is a test of message text 1.'
+M1a     DC    AL1(32),C'This is a test of message text 1.'

 M1b     PFMSG1 'Hello'
+M1b     DC    AL1(4),C'Hello'
```

HLASM Macro Tutorial

Simplest Prefixed Message Text

In this first example, the text of the message may not contain any 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 quoted text of the message, &Txt,
 .*    contain no embedded apostrophes (quotes) or ampersands.
        LclA   &Len             Effective Length
 &Len   SetA   K'&Txt-3         (# text chars)-3 (2 quotes, eff. length)
 &Lab   DC     AL1(&Len),C&Txt
        MEND

 M1a    PFMSG1 'This is a test of message text 1.'
+M1a    DC    AL1(32),C'This is a test of message text 1.'

 M1b    PFMSG1 'Hello'
+M1b    DC    AL1(4),C'Hello'
```

Figure 38. Macro to define a length-prefixed message

Create General Length-Prefixed Messages (2) **115**

- PFMSG2: Allow all characters in text (but: may require pairing)

```
         MACRO
 &Lab    PFMSG2 &Txt
 .*    PFMSG2 — the text of the message, &Txt, may contain embedded
 .*    apostrophes (quotes) or ampersands, if they are properly paired.
 &T      SetC   'TXT&SYSNDX.M'   Create TXTnnnM symbol to name the text
 &Lab    DC     AL1(L'&T.-1)     Effective length
 &T      DC     C&Txt
         MEND

 M2a     PFMSG2 'Test of ''This'' && ''That''.'
+M2a     DC     AL1(L'TXT0001M-1)      Effective length
+TXT0001M DC    C'Test of ''This'' && ''That''.'

 M2b     PFMSG2 'Hello, World'
+M2b     DC     AL1(L'TXT0002M-1)      Effective length
+TXT0002M DC    C'Hello, World'
```

- Quotes/ampersands in message are harder to write, read, translate
- Extra (uninteresting) labels are generated

HLASM Macro Tutorial

More General Prefixed Message Text

Requiring no ampersands or quotes in the message text defined by PFMSG1 may not be acceptable in some situations. Thus, in Figure 39 on page 126 we define a second macro PFMSG2 that allows such characters in the message, but requires that they be properly paired in the argument string. By generating an ordinary symbol, a length attribute reference can be used.

```
        MACRO
&Lab    PFMSG2 &Txt
.*    PFMSG2 -- the text of the message, &Txt, may contain embedded
.*    apostrophes (quotes) or ampersands, if 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.
&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
```

Figure 39. Macro to define a length-prefixed message with paired characters

Some sample calls to the PFMSG2 macro are shown in the following figure:

```
 M2a      PFMSG2 'Test of ''This'' && ''That''.'
+M2a      DC     AL1(L'TXT0001M-1)
+TXT0001M DC     C'Test of ''This'' && ''That''.'

 M2b      PFMSG2 'Hello, World'
+M2b      DC     AL1(L'TXT0002M-1)
+TXT0002M DC     C'Hello, World'
```

The generated symbol is of the form TXTnnnnM, where the characters nnnn are the value of the system variable symbol &SYSNDX. The assembler increments &SYSNDX by one each time a macro expansion begins, and its value is constant within that macro. (Inner macro calls have their own, different value of &SYSNDX.) Thus, &SYSNDX can be used to generate unique symbols or other values for every macro expansion. (See the description of &SYSNDX on page 259 in System (&SYS) Variable Symbols.)

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!) Also, the generated TXTnnnnM symbols are used only for a length attribute reference, and are otherwise uninteresting.

This limitation can be removed by using the powerful AREAD statement.

Readable Length-Prefixed Messages (3): Pseudo-Code 116

- User writes “plain text” messages (single line, ≤ 72 characters)
- PFMSG3: AREAD statement within the macro “reads” the next source record in the input stream (following the macro call, usually) into a character variable symbol
- Allow all characters in message text without pairing
- Pseudo-code:

```
IF (any positional arguments) ERROR EXIT with message

[1] AREAD a message from the following source record

[2] Trim off sequence field (73–80) and trailing blanks, note length

[3] Create paired quotes and ampersands (for nominal value in DC)

[4] GEN (label  DC  AL1(Text_Length–1),C'MessageText')
```


Create Readable Length-Prefixed Messages 117

```
        MACRO
&Lab    PFMSG3  &Null           Comments OK after comma
.*    PFMSG3 — the text of the message may contain any characters.
.*    The message is on a single record following the call to PFMSG3.
        LclA   &L               Local arithmetic variables
        LclC   &T,&C,&M         Local character variables
        AIF    ('&Null' eq '').OK     Null argument OK
        AIF    (N'&SYSLIST EQ 0).OK   No arguments allowed
        MNote  4,'PFMSG3 — no operands should be provided.'
        MEXIT                  Terminate macro processing
.OK     ANOP
.*  Read the record following the PFMSG3 call into &M
&M      ARead  ,               Read the message text [1]
&M      SetC   '&M'(1,72)      Trim off sequence field [2...]
&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
.*      – – –  (continued)
```


Create Readable Length-Prefixed Messages ... 118

```
.*      - - -  (continuation)
.C      ANOP
&T      SetC   DOUBLE('&M'(1,&L))  Pair-up quotes, ampersands [3]
&L      SetA   &L-1             Set to effective length
&Lab    DC     AL1(&L),C'&T' [4]
        MEnd
```

- Messages are written as they are expected to appear!
- Easier to read and translate to other national languages

```
 M4a     PFMSG3  ,     Test with mixed apostrophes/ampersands
-Test of 'This' & 'That'.
+M4a     DC     AL1(27),C'Test of ''This'' && ''That''.'

 M4c     PFMSG3
-This is the text of a long message & says nothin' very much.
+M4c     DC     AL1(63),C'This is the text of a long message && saysX
+                nothin'' very much.'
```

- '+' prefix in listing for generated statements, '-' for records digested by AREAD

Prefixed Message Text with the AREAD Statement

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

```
&CVar  AREAD ,
```

then the first statement in the main program following the macro containing the AREAD statement (or the outermost 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 use AREAD in the PFMSG3 macro to read the text of a message written in its natural final form from the line following the macro call. The operation of the PFMSG3 macro is summarized in this pseudo-code:

```
IF (any positional arguments) ERROR EXIT with message
AREAD a message from following record
Trim off sequence field (73-80) and trailing blanks
Create paired quoted and ampersands (for nominal value in DC)
GEN (label  DC  AL1(Text_Length-1),C'MessageText')
```

The macro illustrated in Figure 40 on page 129 uses the DOUBLE internal function to create pairs of quotes and ampersands wherever needed; thus, the of the message writer need not be aware of the peculiar rules of the Assembler Language.

To allow users of the PFMSG3 macro to add comments on the macro-call line, the &Null parameter is provided on the prototype statement. If the corresponding argument is null (that is, any comments are preceded by a stand-alone comma), the rest of the statement — the comments — are ignored.

```
        MACRO
&Lab    PFMSG3  &Null           Comments OK after comma
.*    PFMSG3 -- the text of the message may contain any characters.
.*    The message is on a single line following the call to PFMSG3.
        LclA   &L               Local arithmetic variables
        LclC   &T,&C,&M         Local character variables
        AIF    ('&Null' eq '').OK     Null argument OK
        AIF    (N'&SYSLIST EQ 0).OK   No arguments allowed
        MNote  4,'PFMSG3 -- no operands should be provided.'
        MEXIT                  Terminate macro processing
.OK     ANOP
.*  Read the record following the PFMSG3 call into &M
&M      ARead  ,                Read the message text
&M      SetC   '&M'(1,72)       Trim off 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
.C      ANOP
&T      SetC   DOUBLE('&M'(1,&L))  Pair-up quotes, ampersands
&L      SetA   &L-1             Set to effective length
&Lab    DC     AL1(&L),C'&T'
        MEnd
```

Figure 40. Macro to define a length-prefixed message with “true text”

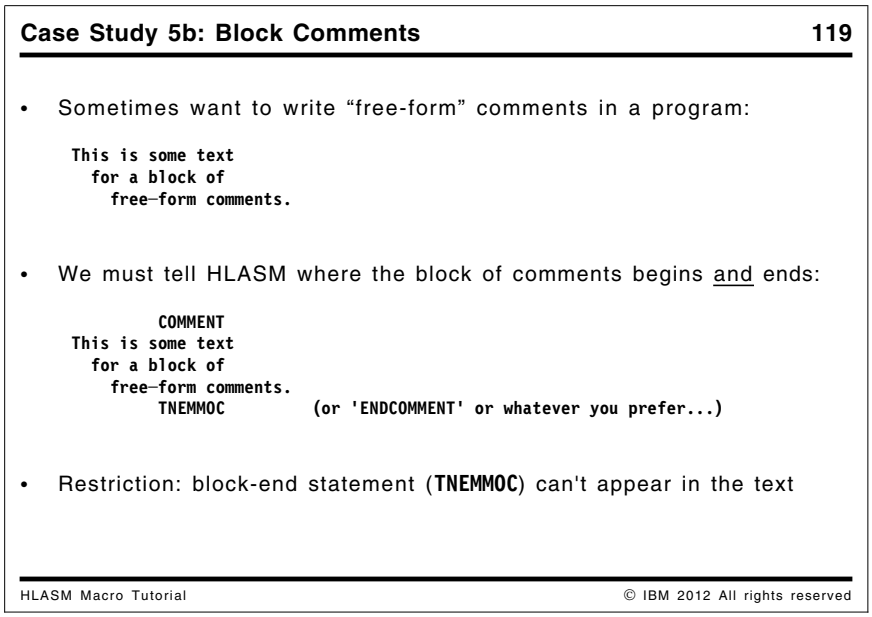
Some tests of the PFMSG3 macro are shown in the following figure.

```
 M4a     PFMSG3  ,     Test with mixed apostrophes/ampersands
-Test of 'This' & 'That'.
+M4a     DC     AL1(27),C'Test of ''This'' && ''That''.'

 M4c     PFMSG3
-This is the text of a long message & says nothin' very much.
+M4c     DC     AL1(63),C'This is the text of a long message && saysX
+                nothin'' very much.'
```

Figure 41. Test cases for macro with “true text” messages

The '+' characters in the left margin denote statements generated by the assembler; the '-' characters denote records read from the source stream by AREAD instructions.

Note also that the loop that removes trailing blanks from the string accepted by the AREAD statement could be replaced by a call to an external conditional-assembly function TRIM; again, writing such a character-valued function is a useful exercise. Another exercise: generalize the above macro to accept multi-line messages, with no limitations on total length.



Case Study 5b: Block Comments

Occasionally it is helpful to be able to insert “blocks” of comments into a program, but without having to put an asterisk in the first position of each line. For example, you might want to write something like

```
This is some text
 for a block of
  comments.
```

Naturally, we will need some way to tell the assembler that the comment “lines” should not be scanned as normal input statements. Thus, we need something that indicates the start (and end) of a “block comment”.

Suppose we create a macro named `COMMENT` that indicates the start of a block comment, and that the end of the block is indicated by a `TNEMMOC` (“`COMMENT`” spelled backward) statement. You could of course choose any terminator you like, such as `ENDCOMMENT`.

In the above example, the lines would be entered as follows:

```
      Comment
This is some text
 for a block of
  comments.
      Tnemmoc
```

You can include program documentation with your source code, and keep them “in sync”.

Block Comments Macro **120**

- The COMMENT macro initiates block comments:

```
         Macro
 &L      Comment &Arg
         LclC    &C
         AIf     ('&L' eq '' and '&Arg' eq '').Read
         MNote   *,'Comment macro: Label and/or argument ignored.'
.Read    ANop
&C       ARead   ,
&C       SetC    Upper('&C')                    Force upper case
&A       SetA    Index('&C'(1,72),' TNEMMOC ')  Note blanks!
         AIf     (&A eq 0).Read
         MEnd
```

- Lets you include your documentation with the source code!

HLASM Macro Tutorial

A simple macro using the AREAD statement can do the job:

```
         Macro
&L       Comment &Arg
         LclC    &C
         AIf     ('&L' eq '' and '&Arg' eq '').Read
         MNote   *,'Comment macro: Label and/or argument ignored.'
.Read    ANop
&C       ARead   ,
&C       SetC    Upper('&C')                    Force upper case
&A       SetA    Index('&C'(1,72),' TNEMMOC ')  Note blanks!
         AIf     (&A eq 0).Read
         MEnd
```

Figure 42. Macro for block comments

The macro first checks for the presence of a label or operand on the COMMENT statement, and if either is present, it emits an MNOTE comment saying they were ignored. The macro then reads each line of the comment block (using the AREAD statement) until a line containing the end-of-comment marker (in any mixture of upper and lower case, with preceding and following blanks and without quotes) is encountered. The UPPER function internally converts each line of the block comment text to upper case to simplify the INDEX function's checking for the presence of the TNEMMOC terminator.

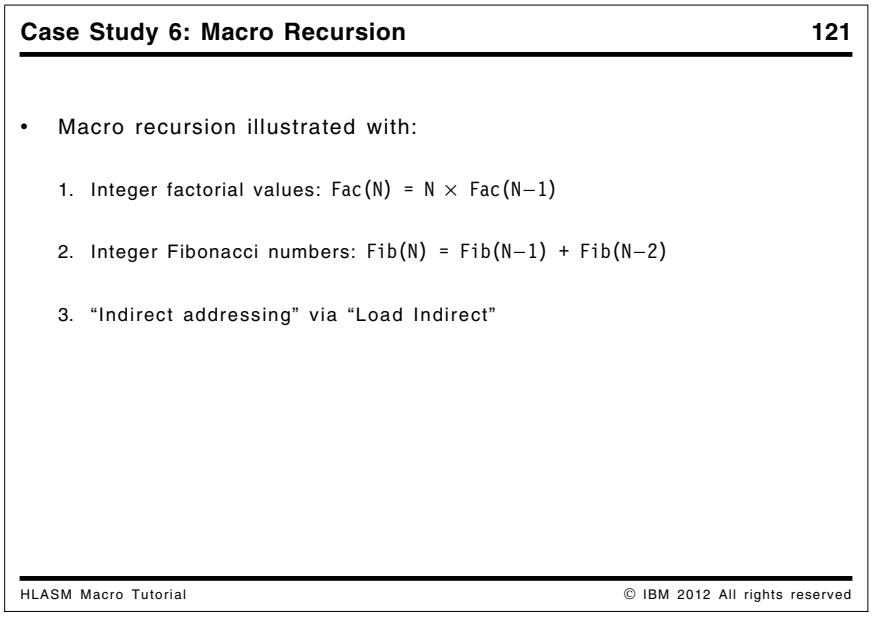
The only restriction on this technique is that the end-of-block terminator cannot appear in the text of the comments with blanks on either side. A test case with the comment terminator embedded in the text is:

```
         Comment
Note that the block-comment terminator can't appear in the
comments! That's because the embedded terminator string on
this line causes an error when this TNEMMOC is found:
         Tnemmoc
```

The presence of the string ' TNEMMOC ' in an input line (in any mix of upper and lower case) causes the macro to terminate its AREAD loop too early, leaving one or more statements to be scanned by the assembler as normal input:

```
      27          Comment
      28-Note that the block-comment terminator can't appear in the
      29-comments! That's because the embedded terminator string on
      30-this line causes an error when this TNEMMOC is encountered:
      31          Tnemmoc
** ASMA057E Undefined operation code - TNEMMOC
```

Suppose you want to use the terminating string TNEMMOC in a record, as in this example. As an exercise, you can modify the COMMENT macro to remove leading and trailing blanks before checking that the terminator record contains *only* the string 'TNEMMOC'.



Case Study 6: Macro Recursion

Macros that call themselves either directly or indirectly are *recursive*. Here are three examples:

- a “Factorial” macro FACTORAL (see “Recursion Example 1: Binary Factorial-Function Values” on page 133)
- a FIBONACI macro to calculate Fibonacci numbers (see “Recursion Example 2: Fibonacci Numbers” on page 135)
- a “Load Indirect” macro LI (see “Recursion Example 3: Indirect Addressing” on page 137)

Generate Binary Factorial Values Recursively 122

- Factorial: defined by Fac(N) = N × Fac(N−1), Fac(0) = Fac(1) = 1

```
        Macro
&Lab    FACTORAL  &N
        GBLA      &Ret            For returning values of inner calls
&L      SetA      &N              Convert from external form
        AIF       (&L GE 2).Calc  Calculate via recursion if N > 1
&Ret    SetA      1               Fac(0) = Fac(1) = 1
        AGO       .Test           Return to caller
.Calc   ANOP
&K      SetA      &L-1
        FACTORAL  &K              Recursive call!
&Ret    SetA      &Ret*&L
.Test   AIF       (&SysNest GT 1).Exit   Check nesting level
.*      MNote     0,'Factorial(&L.) = &Ret.'  Intermediate result
&Lab    DC        F'&Ret'
.Exit   MEnd
```

- Error checking omitted...

HLASM Macro Tutorial

Recursion Example 1: Binary Factorial-Function Values

Probably the best-known recursive function is the Factorial function, defined by the relations

Factorial(0) = Factorial(1) = 1
Factorial(N) = N × Factorial(N-1)

In a macro, it can be defined and implemented iteratively (and more simply), but its familiarity makes it useful as a recursion example.

In the macro in Figure 43 on page 134, the macro FACTORAL 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. The assembler increments &SYSNEST by one each time a macro expansion begins, and decreases it by one each time a macro expansion terminates. Thus, for nested macro calls, &SYSNEST indicates the current nesting level or "depth" of the call. A macro called from open code is at level 1.

```
       Macro
&Lab   FACTORAL  &N
.* Factorials defined by Fac(N) = N * Fac(N-1), Fac(0) = Fac(1) = 1
       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 FACTORAL(&L.)'  For debugging
       AIF       (&L LT 0).Error  Can't handle N < 0
       AIF       (&L GE 2).Calc   Calculate via recursion if N > 1
&Ret   SetA      1                Fac(0) = Fac(1) = 1
       AGO       .Test            Return to caller
.Calc  ANOP
&K     SetA      &L-1
       FACTORAL  &K               Recursive call!
&Ret   SetA      &Ret*&L
.Test  AIF       (&SysNest GT 1).Exit   Check nesting level
.*     MNote     0,'Factorial(&L.) = &Ret.'  Intermediate result
&Lab   DC        F'&Ret'
.Cont  MExit                      Return to caller
.Error MNote     11,'Invalid Factorial argument &N..'
       MEnd
```

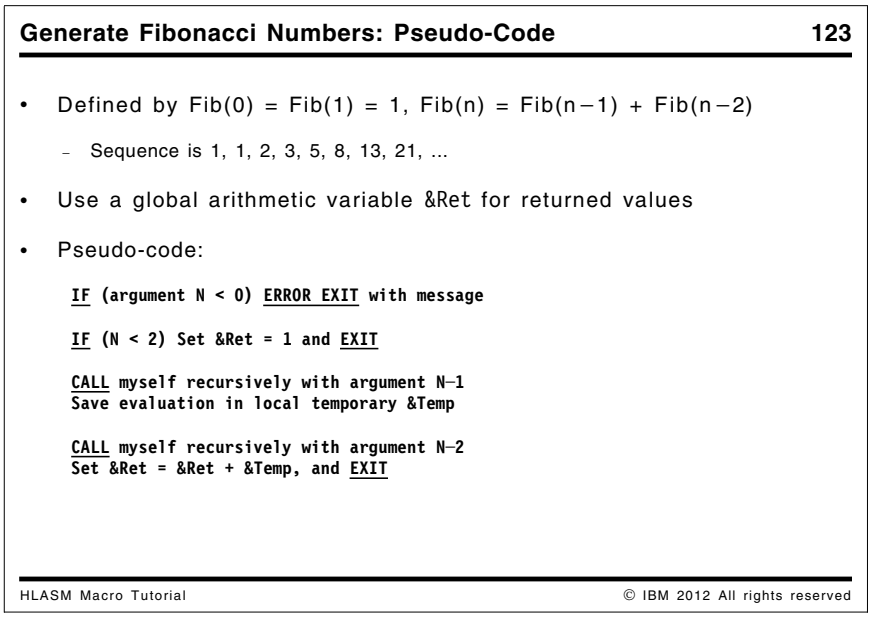
Figure 43. Macro to calculate factorials recursively

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

```
        FACTORAL  0
+       DC        F'1'

        FACTORAL  1
+       DC        F'1'

        FACTORAL  B'11'       Valid self-defining term
+       DC        F'6'

        FACTORAL  X'4'        Also valid
+       DC        F'24'

        FACTORAL  10
+       DC        F'3628800'
```

Figure 44. Macro to calculate factorials recursively: examples

As an exercise, make modifications to allow FACTORAL to be called from other macros. A more challenging exercise is a macro to generate 64-bit integer factorial values. Also, you could revise the macro to generate a table of factorials from 0 to &N.



Recursion Example 2: Fibonacci Numbers

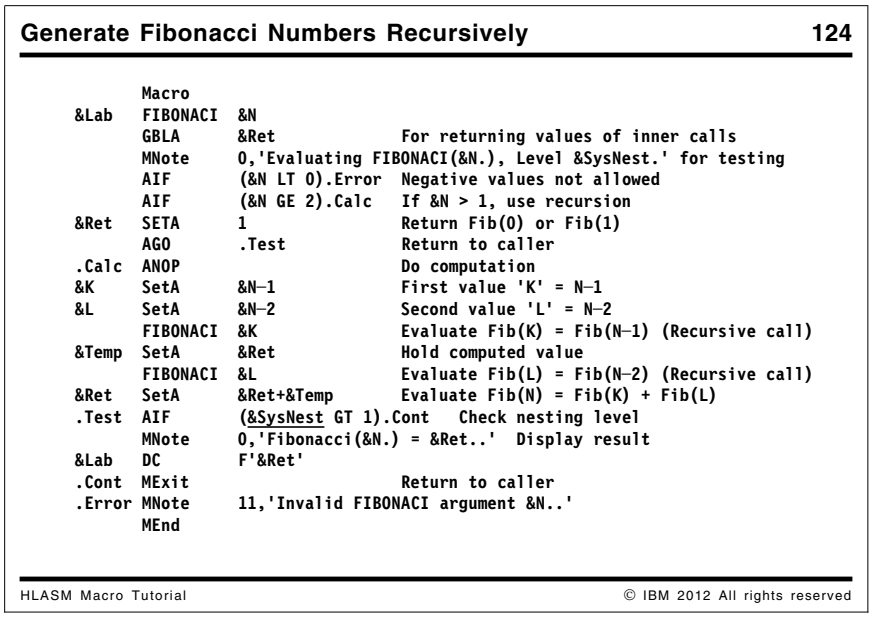
The Fibonacci numbers are generated by starting with 1 and 1, and generating the next number in the sequence by adding the previous two. The recursion relations are

Fib(N) = Fib(N-1) + Fib(N-2)
with Fib(0) = 1 and Fib(1) = 1

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

A pseudo-code description follows:

```
IF (argument N < 0) ERROR EXIT with message
IF (N < 2) Set &Ret = 1 and EXIT
CALL myself recursively with argument N-1
Save evaluation in local temporary &Temp
CALL myself recursively with argument N-2
Set &Ret = &Ret + &Temp, and EXIT
```



The FIBONACI macro is illustrated in Figure 45. The global variable &Ret is used to return the value of a call to FIBONACI, because macros do not have any other method to return function values. The local variable &Temp holds the value returned by the first recursive call, so that the second can be made without destroying the value returned by the first.

```
       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.' for testing
       AIF       (&N LT 0).Error  Negative values not allowed
       AIF       (&N GE 2).Calc   If &N > 1, use recursion
&Ret   SETA      1                Return Fib(0) or Fib(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 Fib(K) = Fib(N-1) (Recursive call)
&Temp  SetA      &Ret             Hold computed value
       FIBONACI  &L               Evaluate Fib(L) = Fib(N-2) (Recursive call)
&Ret   SetA      &Ret+&Temp       Evaluate Fib(N) = Fib(K) + Fib(L)
.Test  AIF       (&SysNest GT 1).Cont   Check nesting level
       MNote     0,'Fibonacci(&N.) = &Ret..'  Display result
&Lab   DC        F'&Ret'
.Cont  MExit                      Return to caller
.Error MNote     11,'Invalid FIBONACI argument &N..'
       MEnd
```

Figure 45. Macro to calculate Fibonacci numbers recursively

An example of executing this macro is:

```
                                24        FIBONACI  4
** ASMA254I *** MNOTE ***      25+     0,Evaluating FIBONACI(4), Level 1
** ASMA254I *** MNOTE ***      27+     0,Evaluating FIBONACI(3), Level 2
** ASMA254I *** MNOTE ***      29+     0,Evaluating FIBONACI(2), Level 3
** ASMA254I *** MNOTE ***      31+     0,Evaluating FIBONACI(1), Level 4
** ASMA254I *** MNOTE ***      33+     0,Evaluating FIBONACI(0), Level 4
** ASMA254I *** MNOTE ***      35+     0,Evaluating FIBONACI(1), Level 3
** ASMA254I *** MNOTE ***      37+     0,Evaluating FIBONACI(2), Level 2
** ASMA254I *** MNOTE ***      39+     0,Evaluating FIBONACI(1), Level 3
** ASMA254I *** MNOTE ***      41+     0,Evaluating FIBONACI(0), Level 3
** ASMA254I *** MNOTE ***      42+     0,Fibonacci(4) = 5.
```

Figure 46. Example showing evaluation of Fibonacci numbers

If you want to have some fun, write a macro to calculate the Ackermann function, defined by

```
Ack(0,y) = y + 1
Ack(x,0) = Ack(x-1,1)
Ack(x,y) = Ack(x-1,Ack(x,y-1))
```

But be careful: going beyond Ack(3,3) can consume very large amounts of time and storage! (As a check: Ack(1,1)=3 and Ack(2,2)=7.)

Indirect Addressing via Recursion 125

- "Load Indirect" macro for multi-level "pointer following"
- Syntax: each prefixed asterisk adds one 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
```

- 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
 .*      Generate top-level (direct) reference
 &Lab    L     &Reg,&X
         MExit                   Exit from bottom level of recursion
 .Ind    ANop
 &XI     SetC  '&X'(2,*)         Remove leading asterisk
 .*      Generate indirect reference
         LI    &Reg,&XI          Call myself recursively
         L     &Reg,0(,&Reg)
         MEnd
```

HLASM Macro Tutorial

Recursion Example 3: Indirect Addressing

In Figure 47 on page 138, 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.[12] Thus, if an instruction was written as

[12] 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*), and the statement's operand field was not modified.

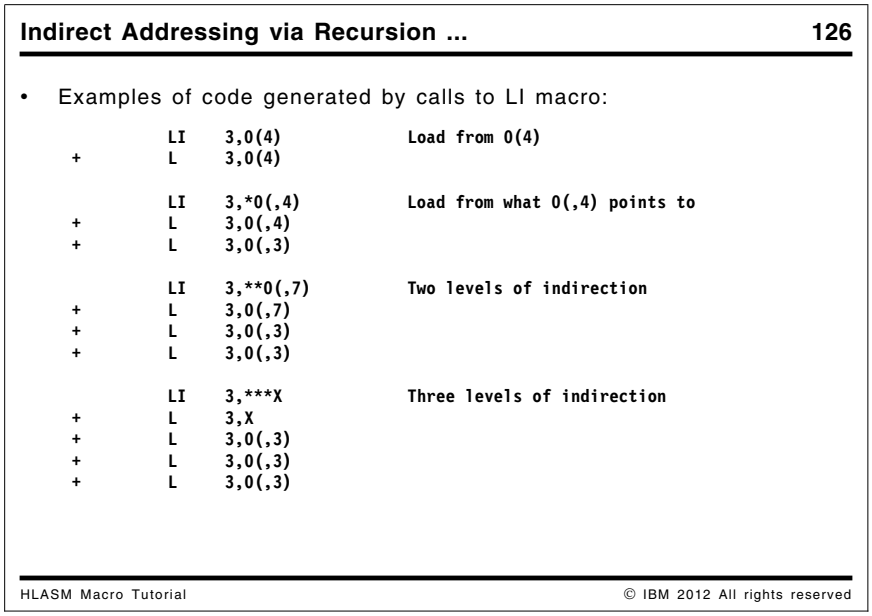
```
         LI    8,*XXX              Indirect reference via XXX
```

then the item to be loaded into R8 is not at XXX, but at the location whose address is found at XXX. Thus, the asterisk can be thought of as a "de-referencing" operator.

A macro to implement this form of indirect addressing is shown in Figure 47.

```
         Macro
&Lab     LI    &Reg,&X             Load &Reg with indirection
         Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
.*       Generate top-level (direct) reference
&Lab     L     &Reg,&X
         MExit                     Exit from bottom level of recursion
.Ind     ANop
&XI      SetC  '&X'(2,K'&X-1)      Remove leading asterisk
.*       Generate indirect reference
         LI    &Reg,&XI            Call myself recursively
         L     &Reg,0(,&Reg)
         MEnd
```

Figure 47. Recursive macro to implement indirect addressing



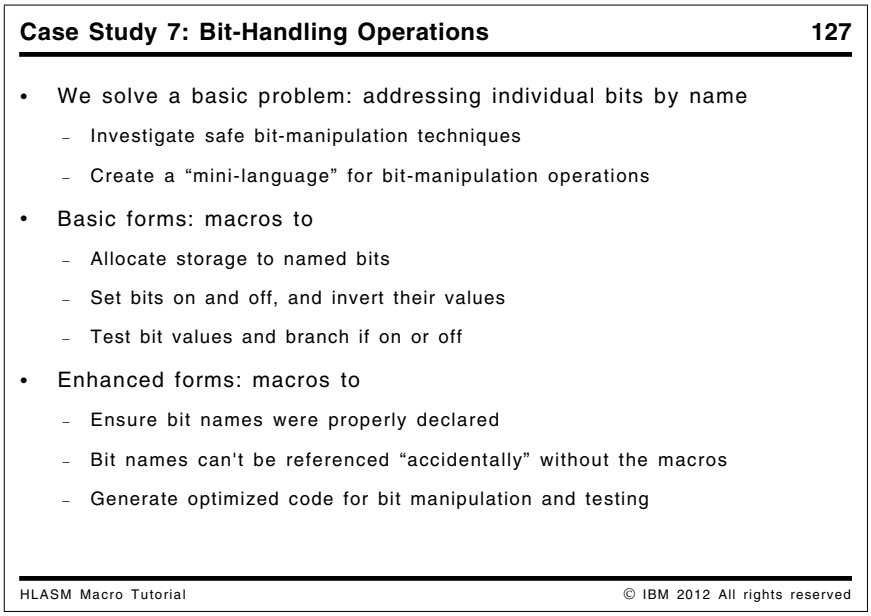
Examples of calls to the LI macro are shown in Figure 48 on page 139.

```
         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 48. Recursive macro to implement indirect addressing: examples

This definition is recursive, because an operand preceded by an asterisk may itself be preceded by an asterisk: this provides multiple levels of indirection.

Note that R0 cannot be used for &Reg if any levels of indirection are indicated. As an exercise, modify the macro to check for the register operand being 0 if the second operand indicates indirection.

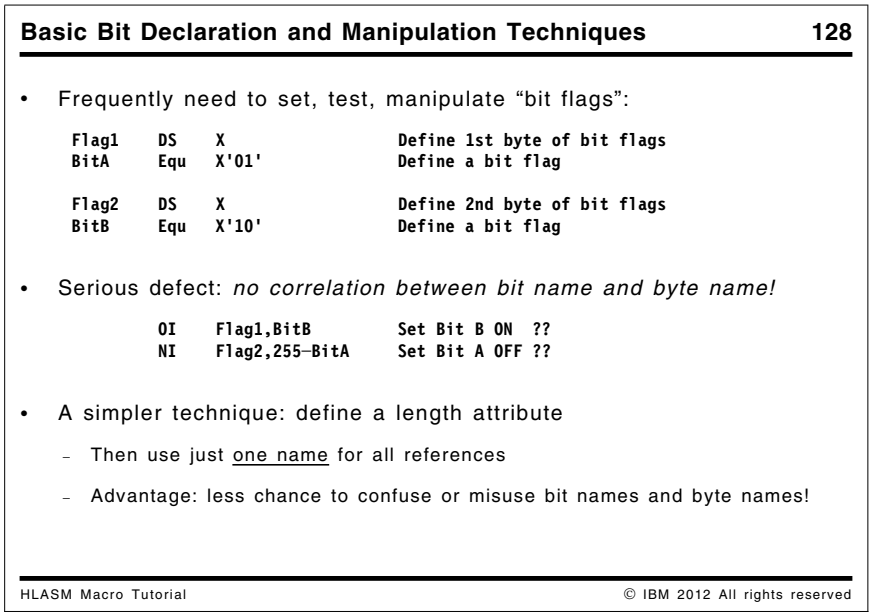


Case Study 7: Macros for Bit-Handling Operations

We now examine some macros that show how you can build a language to suit your needs. Our examples will be based on a typical Assembler Language requirement to manipulate bit values, and we will illustrate two levels of possible implementation:

1. The first set of macros (Case Study 7a) illustrates simple techniques for declaring bit names, assigning them to storage, performing operations on them, and testing bit values and making conditional branches.
2. The second set (Case Study 7b) does the same functions, but in addition validates that only declared names are used, and generates optimized code for storage allocation, bit operations, and bit testing.

One purpose of these examples is to show how macros can be made as simple or as complex as are needed for a specific application: if bit operations need not be efficient, the simple macros can be used; if safety is important (so that named bits can't be manipulated without macro validation), and storage requirements and/or execution time must be minimized, the second set can be used.



Basic Bit Handling Techniques

Applications frequently require status flags with binary values: ON or OFF, YES or NO, STARTED or NOT_STARTED, etc. Such flags are usually 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.

A common Assembler Language technique defines bits using statements like these:

```
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
```

Then we do bit operations like

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

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

```
OI    Flag1,BitB         Set Bit B ON  ??
NI    Flag2,255-BitA     Set Bit A OFF ??
```

Because 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.

Simple Bit-Declaring Macro: Design Considerations 129

- Several ways to generate bit-name definitions
  1. Allocate storage byte and bit name together:

```
Flag1  DC   X'0',X'80'      Byte with bit value = length attribute
```

  2. Allocate *unnamed* storage byte first, define bits following:

```
       DC   X'0'            Unnamed byte
Bit_A  Equ  *-1,X'80'       Bit_A defined as bit 0
```

  3. Define bits first, allocate *unnamed* storage byte following:

```
Bit_B  DS   0XL(X'40')      Bit_B defined as bit 1
       DC   X'0'            Unnamed byte
```

- Length Attribute used for named bits and *unnamed* bytes

```
TM   Bit_Name,L'Bit_Name Refer to byte and bit using bit name

TM    Flag1,L'Flag1      Test setting of Flag1 bit
NI    BitA,255-L'BitA    Set BitA OFF (uses name 'BitA' only)
OI    BitB,L'BitB        Set BitB ON  (uses name 'BitB' only)
```

HLASM Macro Tutorial

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

```
Flag1    DS    X,X'80'          Named byte and associated bit
Flag2    Equ   Flag1,X'40       Same byte, another bit
         DS    X                Unnamed byte
BitA     Equ   *-1,X'01'        Length Attribute = bit value
         DS    X                Unnamed byte
BitB     Equ   *-1,X'10'        Length Attribute = bit value
```

Another way to achieve the same result is to associate the length attribute with the storage location:

```
BitA     DS    0XL(X'01')       Length Attribute = bit value
         DS    X                Unnamed byte
BitB     DS    0XL(X'10')       Length Attribute = bit value
         DS    X                Unnamed byte
```

In each case, the bit name is the same as the name of the byte that contains it, and the byte itself has no name. Then, bit references are made only with the bit names:

```
XI    Flag1,L'Flag1      Invert Flag1
OI    BitA,L'BitA        Set Bit A 'on'
TM    BitB,L'BitB        Test Bit B
```

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

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

There is something peculiar about this statement. 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-Declaring Macro: Pseudo-Code **130**

- Generates a bit-name EQUate for each argument, allocates storage
- Syntax: SBitDef bitname[,bitname]...
- Examples:

```
SBitDef  b1,b2,b3,b4,b5,b6,b7,b8    Eight bits in one byte

SBitDef  c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v   Many bits+bytes
```

- Pseudo-code:

```
Set Lengths to bit-position weights (128,64,32,16,8,4,2,1)

DO for M = 1 to Number_of_Arguments
   IF (Mod(M,8)=1) GEN (  DC  B'0' )     (Generate unnamed byte)
   GEN (Arg(M)  EQU  *-1,Lengths(Mod(M-1,8)+1) )  (Define bit name)
```

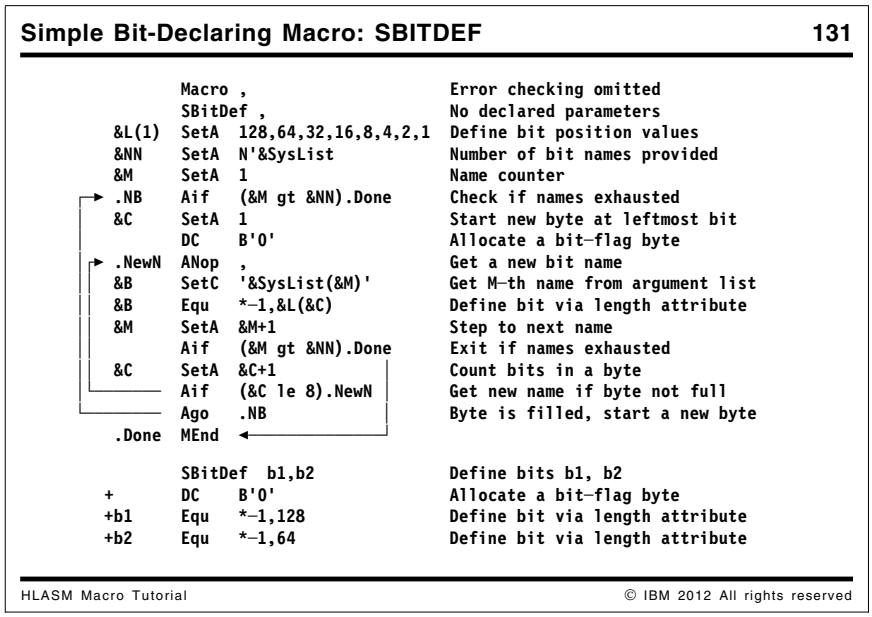
HLASM Macro Tutorial

Case Study 7a: Bit-Handling Macros -- Simple Forms

The simplest way to ensure correct matching of bit names and byte names is to make all bit references with macros. Here is a simple set of macros to do this.

First, we define bit names and allocate storage for them, with a macro that accepts a list of bit names and defines bit values in successive bytes, up to eight bits per byte. A pseudo-code description of the macro's operation is:

```
Set Lengths to bit-position weights (128,64,32,16,8,4,2,1)

DO for M = 1 to Number_of_Arguments
   IF (Mod(M,8)=1) GEN (  DC  B'0')    (Generate unnamed byte)
   GEN (Arg(M)  EQU  *-1,Lengths(Mod(M-1,8)+1) )  (Define bit name)
```



The SBitDef macro in Figure 49 takes the names in the argument list and allocates a single bit to each. Each call to the SBitDef macro starts a new byte. The &SYSLIST system variable symbol accesses the arguments, and a Number attribute reference, N'&SYSLIST, determines the number of arguments.

```
       Macro
       SBitDef ,                    No declared parameters
&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
       Aif   (&NN eq 0).Null        Check for null argument list
.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   *-1,&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,'SBitDef: 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 49. Simple bit-handling macros: bit declarations

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

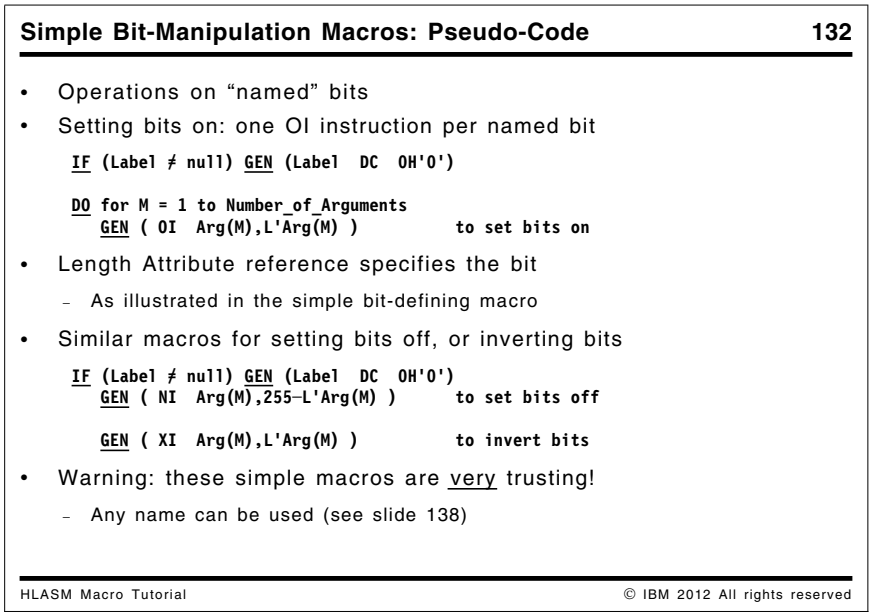
```
         SBitDef  c,d,e,f,g,h,i,j,k,l,m   Many bits and bytes
+        DC    B'0'                  Allocate a bit-flag byte
+c       Equ   *-1,128               Define bit via length attribute
+d       Equ   *-1,64                Define bit via length attribute
+e       Equ   *-1,32                Define bit via length attribute
+f       Equ   *-1,16                Define bit via length attribute
+g       Equ   *-1,8                 Define bit via length attribute
+h       Equ   *-1,4                 Define bit via length attribute
+i       Equ   *-1,2                 Define bit via length attribute
+j       Equ   *-1,1                 Define bit via length attribute
+        DC    B'0'                  Allocate a bit-flag byte
+k       Equ   *-1,128               Define bit via length attribute
+l       Equ   *-1,64                Define bit via length attribute
+m       Equ   *-1,32                Define bit via length attribute
```

Figure 50. Simple bit-handling macros: example of bit declarations

This simple macro has several limitations:

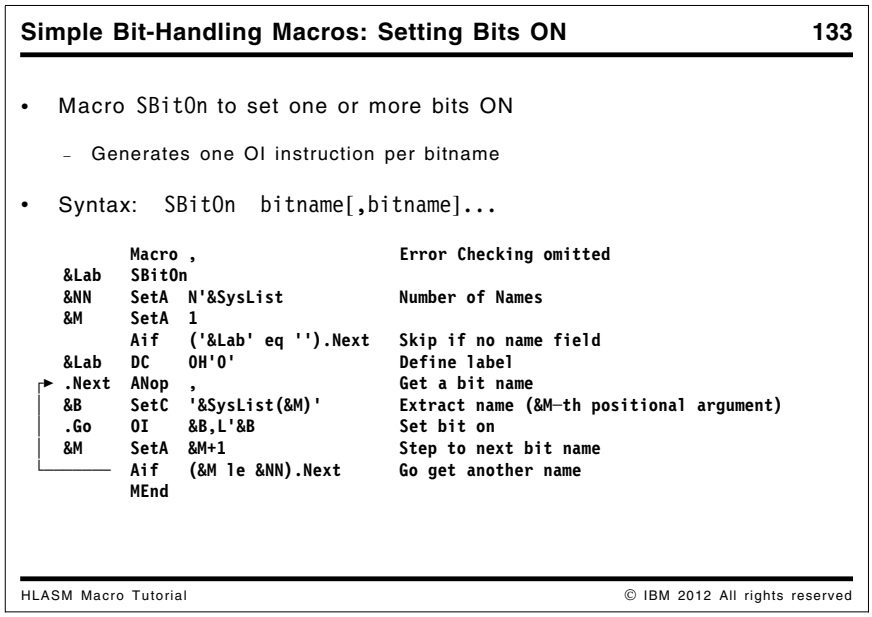
- Bits cannot be “grouped” so that related bits are certain to reside in the same byte, except by writing a statement with a new `SBitDef` macro call.
- This means that we cannot plan to use the machine's bit-manipulation instructions (which can handle up to 8 bits simultaneously) without manually arranging the assignments of bits and bytes.
- If a bit name is declared twice, the macro cannot detect the error; HLASM will issue ASMA043E message for a previously defined symbol.

In Case Study 7b we will explore some techniques that overcome these limitations.



Simple Bit-Manipulation Macros

Now, we illustrate some simple macros that use the bit declarations just described. While the macros are useful, they do very little checking; improvements will be discussed later, at "Case Study 7b: Bit-Handling Macros -- Advanced Forms" on page 151.



Simple Bit-Manipulation Macros: Setting Bits ON

Having created the SBitDef macro to define bit names, we now write macros to manipulate them by setting bits on and off, and by inverting their state. First, we write a macro SBitOn that will set a bit to 1.

A pseudo-code description of the SBitOn macro:

```
IF (Label ≠ null) GEN (Label  DC  OH'0')

DO for M = 1 to Number_of_Arguments
   GEN (  OI EQU  Arg(M),L'Arg(M) )
```

The SBitOn macro is defined in Figure 51 on page 146.

```
        Macro
&Lab    SBitOn
&NN     SetA  N'&SysList            Number of Names
&M      SetA  1
        Aif   (&NN gt 0).OK         Should not have empty name list
        MNote 4,'SBitOn: 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 (&M'th positional arg)
        Aif   ('&B' ne '').Go       Check for missing argument
        MNote 4,'SBitOn: 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 51. Simple bit-handling macros: bit setting

Simple Bit-Handling Macros: Setting Bits ON ... 134

- Examples:

```
 AA1    SBitOn  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

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

- Observe: one **OI** instruction per bit!
  - We will consider optimizations in Case Study 7b

HLASM Macro Tutorial

The following figure illustrates some calls to this macro:

```
 AA1    SBitOn  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
```

Figure 52. Simple 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 “Case Study 7b: Bit-Handling Macros -- Advanced Forms” on page 151 how we might remedy this defect.

Simple Bit-Handling Macros: Set OFF and Invert Bits 135

- Macros SBitOff and SBitInv are defined like SBitOn:
  - SBitOff uses **NI** to set bits off

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

  - SBitInv uses **XI** to invert bits

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

HLASM Macro Tutorial

Simple Bit-Manipulation Macros: Inverting and Setting Bits OFF

The SBitOff macro is exactly like the SBitOn macro, except that the generated statement to set the bit “off” (i.e., to 0) is changed from OI to NI, and the bit-testing mask field is inverted:

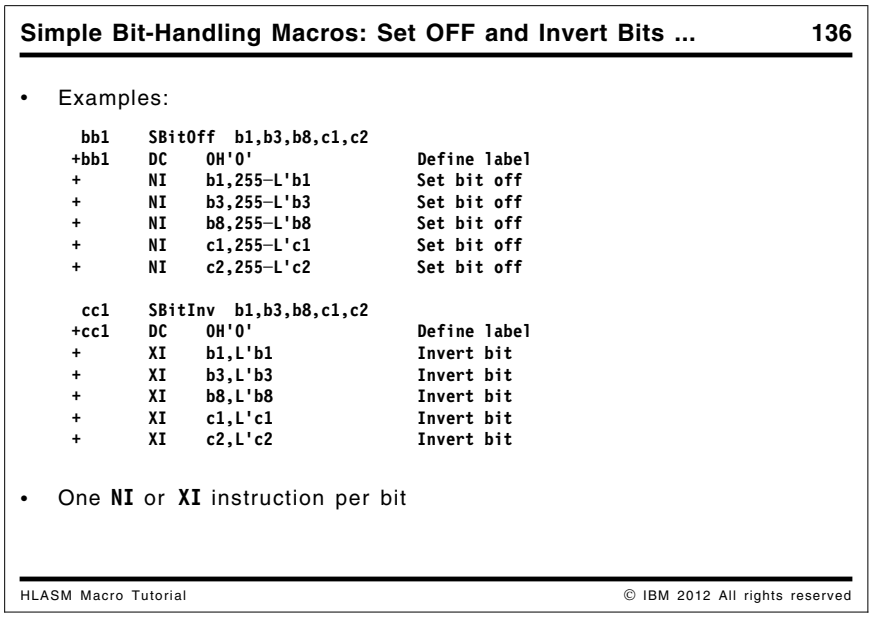
```
       Macro
&Lab   SBitOff
.*     - - -  etc., as for SBitOn
.Go    NI    &B,255-L'&B          Set bit off
.*     - - -  etc., as for SBitOn
       MEnd
```

Figure 53. Simple bit-handling macros: bit resetting

Similarly, the SBitInv macro inverts the designated bits, using XI instructions:

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

Figure 54. Simple bit-handling macros: bit inversion



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

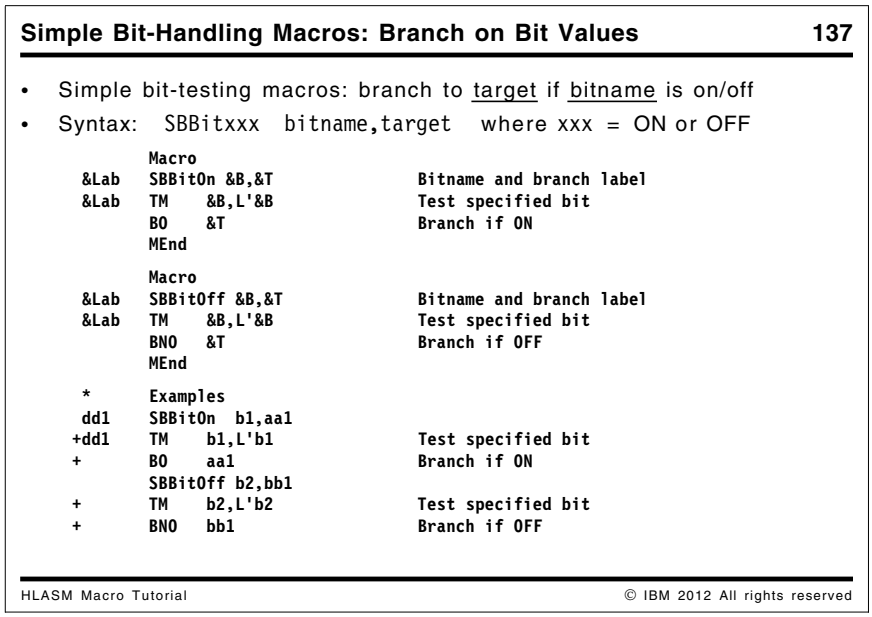
```
 bb1    SBitOff  b1,b3,b8,c1,c2
+bb1    DC    OH'0'                Define label
+       NI    b1,255-L'b1          Set bit off
+       NI    b3,255-L'b3          Set bit off
+       NI    b8,255-L'b8          Set bit off
+       NI    c1,255-L'c1          Set bit off
+       NI    c2,255-L'c2   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

Figure 55. Simple bit-handling macros: examples of bit resetting

Some calls to `SBitInv` illustrate its operation:

```
 cc1    SBitInv  b1,b3,b8,c1,c2
+cc1    DC    OH'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
```

Figure 56. Simple bit-handling macros: examples of bit inversion



Simple Bit-Testing Macros

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 specified by &T if the bit named by &B is on or off. We can write two macros named SBBitOn and SBBitOff to do this; each has two arguments, a bit name and a label name.

The syntax of the two macros is the same:

```
SBBitOn   bitname,target
SBBitOff  bitname,target
```

tests the bit named *bitname*, and if on or off as specified by the name of the macro, branches to the statement with label *target*.

```
       Macro
&Lab   SBBitOn &B,&T               Bitname and branch label
       Aif   (N'&SysList eq 2).OK  Should have exactly 2 arguments
       MNote 4,'SBBitOn: 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,'SBBitOn: Bit Name or Target Name missing'
       MEnd
```

Figure 57. Simple bit-testing macros: branch if bit is on

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

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

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

Figure 58. Simple 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    SBBitOff &B,&T             Bitname and branch label
.*      - - -    etc., as for SBBitOn macro
&Lab    TM    &B,L'&B              Test specified bit
        BNO   &T                   Branch if OFF
.*      - - -    etc., as for SBBitOn macro
        MEnd
```

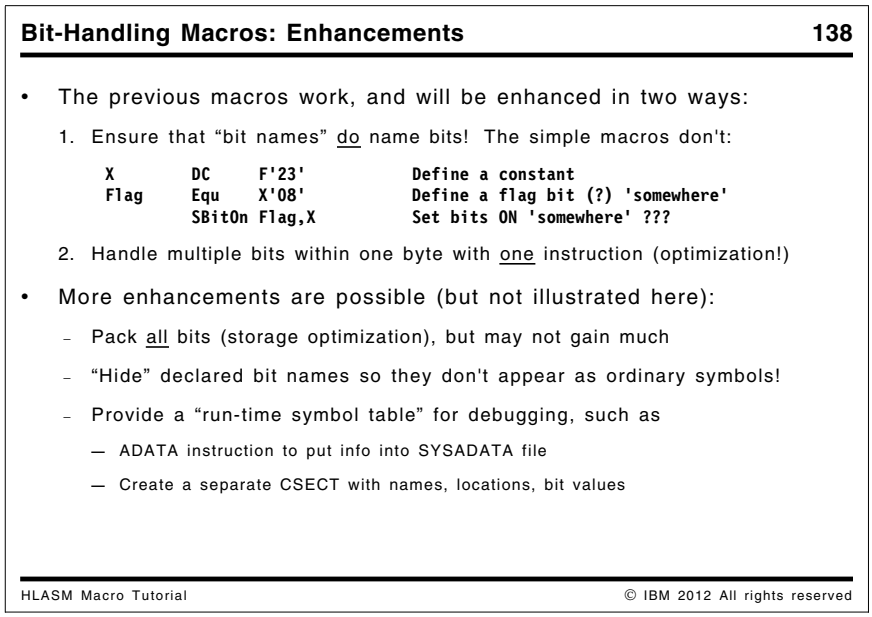
Figure 59. Simple bit-handling macros: branch if bit is off

Calls to the SBBitOff macro might appear as follows:

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

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

Figure 60. Simple bit-handling macros: examples of "branch if bit off"

This completes our first, simple set of bit-handling macros. We see that we can write a fairly helpful set of capabilities with a very small effort, and put them to immediate use.



Case Study 7b: Bit-Handling Macros -- Advanced Forms

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

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

```
         DS    X
BitJ     Equ   *-1,X'40'
BitK     Equ   *-1,X'20'
```

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

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

   While these generated instructions are correct, they do not completely satisfy our intent to name only the bits we wish to manipulate, and not the bytes in which they are defined. Thus, we need some optimization in our bit-handling macros.

2. The previous simple macros are very trusting (and therefore you must be very careful). There is no checking that the bit names presented as arguments in the bit-manipulation macros 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
         - - -
         SBitOn   Flag             Set 'something, somewhere' on ???
```

   and the result would not have been what was expected or desired.

   Similarly, if you had defined a variable X as the name of a fullword integer:

```
X        DC   F'23'
```

   then you could use X as a "bit name" with no warnings:

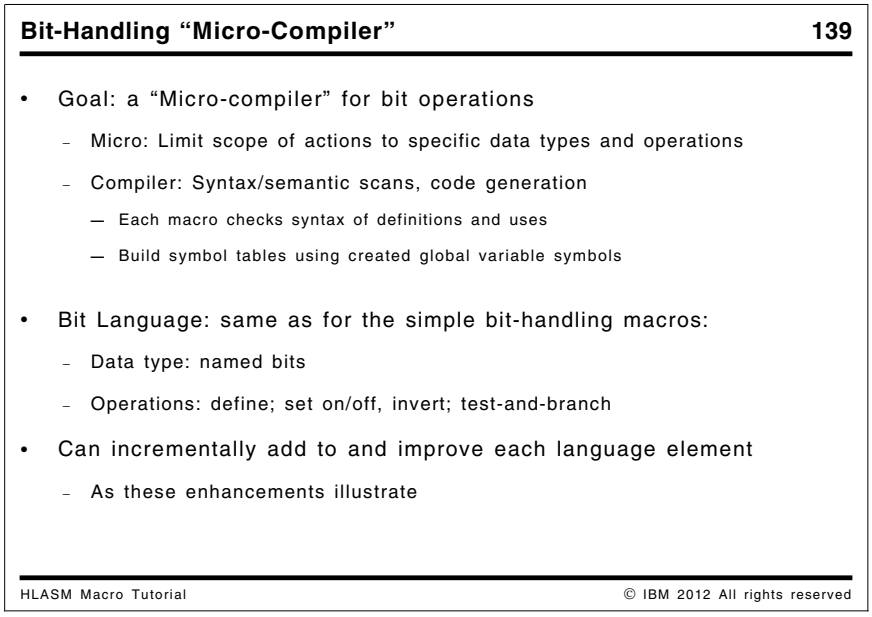
```
         SBitOn  X
```

   This would generate the instruction

```
         OI    X,L'X
```

which is unlikely to give the result you intended!

Thus, we need stronger types and type checking in our bit-handling macros.



Bit-Handling "Micro Language" and "Micro-Compiler"

Solving these problems lets us create a "micro-compiler" for bit declarations and operations. Because we have limited our concerns to bit operations, the macros can be fairly simple, while illustrating some of the functions needed in a typical compiler for a high-level language.

We start with a `BitDef` macro that declares bit flags and keeps track of which ones have been declared. We add an extra feature to help improve program efficiency: if a group of bits should be kept in a single byte, so that they can be set and tested in combinations, then their names may be specified as a parenthesized operand sublist. The macro will ensure that (if at most eight are specified) they will fit in a single 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 decision 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. Each instruction saved represents many bits! (Storage optimization is left as an exercise for the reader.)

General Bit-Handling Macros: Data Structures 140

- Declaring a bitname requires three “global” items:
  1. A **Byte_Number** to count bytes in which bits are declared
  2. A **BitCount** for the next unallocated bit in the current byte
  3. An *associatively addressed* Symbol Table using *created variable symbols*
     - Each declared bit name creates a global arithmetic variable
     - Its name &(BitDef_bitname_ByteNo) is constructed from
       - a prefix **BitDef_** (whatever you like, to avoid global-name collisions)
       - the declared bitname (the “associative” feature)
       - a suffix **_ByteNo** (whatever you like, to avoid global-name collisions)
     - Its value is the **Byte_Number** in which this bit was allocated
- Remember: the storage bytes themselves are unnamed!

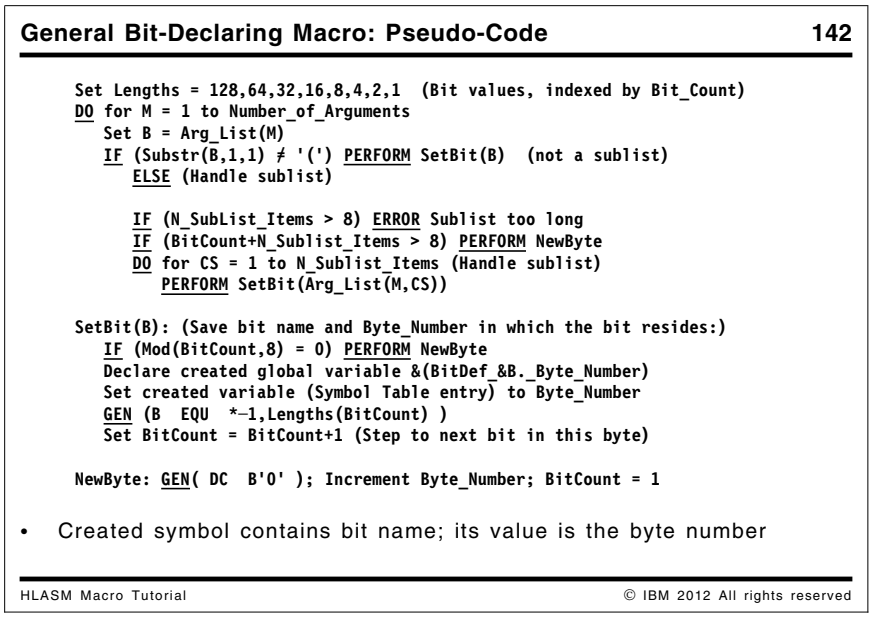
The data structures (which may be thought of as our “micro-compiler's” symbol table) used for these macros include a Byte_Number to enumerate the bytes in which the named bits have been allocated, a Bit_Count to count how many bits have been allocated in the current byte, and a created global arithmetic variable symbol for each bit. The value of the created variable symbol is the Byte_Number in which the named bit resides. We use the fact that a declared arithmetic variable symbol is initialized to zero to detect undeclared bit names.

The created variable symbol's name is arbitrary, and need only contain the bit name somewhere; we construct the name from a prefix BitDef_, the bit name, and a suffix _ByteNo. If such names collide with global variable symbol names used by other macros, it is easy to change the prefix or suffix.

General Bit-Declaring Macro: Design 141

- Bits may be “packed”; sublisted names are kept in one byte
- Example: `BitDef a,(b,c),d` keeps b and c together
- High-level pseudo-code:

```
DO for all arguments
IF argument is not a sublist
   THEN assign the named bit to a byte (start another byte if needed)
   ELSE IF sublist has more than 8 items, ERROR STOP, can't assign
        ELSE if not enough room in current byte, start another
        Assign sublist bit names to a byte
```

This bit-defining macro starts a new byte in storage for each macro call. It would be easy to "pack" all bits (not just those in sublists) to improve storage utilization by providing a global arithmetic variable to remember the current unallocated bit position across calls to the BitDef macro.

A pseudo-code description of the macro is shown in the following figure:

```
Set Lengths = 128,64,32,16,8,4,2,1  (Bit values, indexed by Bit_Count)
DO for M = 1 to Number_of_Arguments
   Set B = Arg_List(M)
   IF (Substr(B,1,1) ≠ '(') PERFORM SetBit(B)  (not a sublist)
      ELSE (Handle sublist)

      IF (N_SubList_Items > 8) ERROR Sublist too long
      IF (BitCount+N_Sublist_Items > 8) PERFORM NewByte
      DO for CS = 1 to N_Sublist_Items (Handle sublist)
         PERFORM SetBit(Arg_List(M,CS))

SetBit(B): (Save bit name and Byte_Number in which the bit resides:)
   IF (Mod(BitCount,8) = 0) PERFORM NewByte
   Declare created global variable &(BitDef_&B._Byte_Number)
   Set created variable (Symbol Table entry) to Byte_Number
   GEN (B  EQU  *-1,Lengths(BitCount) )
   Set BitCount = BitCount+1 (Step to next bit in this byte)

NewByte: GEN( DC  B'0' ); Increment Byte_Number; BitCount = 1
```

Figure 61. Bit-handling macros: define bit names: pseudo-code

In the SetBit(B): portion of the pseudo-code, we use created variable symbols as entries in the "BitDef" symbol table. Each such entry is set to the nonzero value of the Byte_Number in which the bit was allocated. (This "Byte_Number" is simply a counter of the number of bytes allocated by the BitDef macro.)

General Bit-Handling Macros: Bit Declaration 143

```
        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)
```


General Bit-Handling Macros: Bit Declaration ... 144

```
.*      - - -     (continuation)   Name is in a sublist
.SubT   ANop  ,                    Generate sublist equates
&B      SetC  '&SysList(&M,&CS)'   Extract sublist element
        GblA  &(BitDef_&B._ByteNo) Created var sym with ByteNo for this bit
&B      Equ   *-1,&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 sublist
        GblA  &(BitDef_&B._ByteNo) Declare byte number for this bit
&B      Equ   *-1,&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 62 on page 156, several techniques are used. The global arithmetic variable &BitDef_ByteNo is incremented by 1 each time a new byte is allocated. 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.

After each bit name &B has been extracted from the argument list, a global arithmetic variable &(BitDef_&B._ByteNo) is constructed (and declared) using the supplied bit name, 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 is that later references to a bit can be checked against this global variable: if its value is zero (meaning the *variable* was declared but not initialized) we will know that the *bit* was not declared, and therefore not allocated to a byte in storage.

Another 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. The pseudo-code doesn't test for missing or duplicate bit names, but the full macro definition checks them, as in the following figure.

```
        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
        Aif   (&NN eq 0).Null       Check for null argument list
.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
&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
        Aif   ('&B' eq '').Null     Check for null item
        GblA  &(BitDef_&B._ByteNo)  Created var sym with ByteNo for this bit
        Aif   (&(BitDef_&B._ByteNo) gt 0).DupDef  Branch if declared
&B      Equ   *-1,&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
```

Figure 62 (Part 1 of 2). Bit-handling macros: define bit names

```
.NoL    ANop  ,                      Not a sublist
        GblA  &(BitDef_&B._ByteNo)   Declare byte number for this bit
        Aif   (&(BitDef_&B._ByteNo) gt 0).DupDef  Branch if declared
&B      Equ   *-1,&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 62 (Part 2 of 2). Bit-handling macros: define bit names

Examples of Bit Declaration 145

- Example: Define ten bit names (with macro-generated code)
- Bits named d4-d9 are allocated in a single byte
  - Causes some bits to remain unused in the first byte

```
 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   *-1,128              Define bit via length attribute
+d2      Equ   *-1,64               Define bit via length attribute
+d3      Equ   *-1,32               Define bit via length attribute
+        DC    B'0'                 Define a bit-flag byte
+d4      Equ   *-1,128              Define bit via length attribute
+d5      Equ   *-1,64               Define bit via length attribute
+d6      Equ   *-1,32               Define bit via length attribute
+d7      Equ   *-1,16               Define bit via length attribute
+d8      Equ   *-1,8                Define bit via length attribute
+d9      Equ   *-1,4                Define bit via length attribute
+d10     Equ   *-1,2                Define bit via length attribute
```

HLASM Macro Tutorial

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

```
 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   *-1,128               Define bit via length attribute
+d2       Equ   *-1,64                Define bit via length attribute
+d3       Equ   *-1,32                Define bit via length attribute
+         DC    B'0'                  Define a bit-flag byte
+d4       Equ   *-1,128               Define bit via length attribute
+d5       Equ   *-1,64                Define bit via length attribute
+d6       Equ   *-1,32                Define bit via length attribute
+d7       Equ   *-1,16                Define bit via length attribute
+d8       Equ   *-1,8                 Define bit via length attribute
+d9       Equ   *-1,4                 Define bit via length attribute
+d10      Equ   *-1,2                 Define bit via length attribute

 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   *-1,128               Define bit via length attribute
+k2       Equ   *-1,64                Define bit via length attribute
+k3       Equ   *-1,32                Define bit via length attribute
+k4       Equ   *-1,16                Define bit via length attribute
+k5       Equ   *-1,8                 Define bit via length attribute
+k6       Equ   *-1,4                 Define bit via length attribute
+k7       Equ   *-1,2                 Define bit via length attribute
+k8       Equ   *-1,1                 Define bit via length attribute
+         DC    B'0'                  Define a bit-flag byte
+k9       Equ   *-1,128               Define bit via length attribute
+k10      Equ   *-1,64                Define bit via length attribute
```

Figure 63. Bit-handling macros: examples of defining bit names

We will now utilize the information created by this `BitDef` macro to generate efficient instruction sequences.

General Bit-Setting Macro: Data Structure Details 146

Two “phases” used to generate bit-operation instructions:

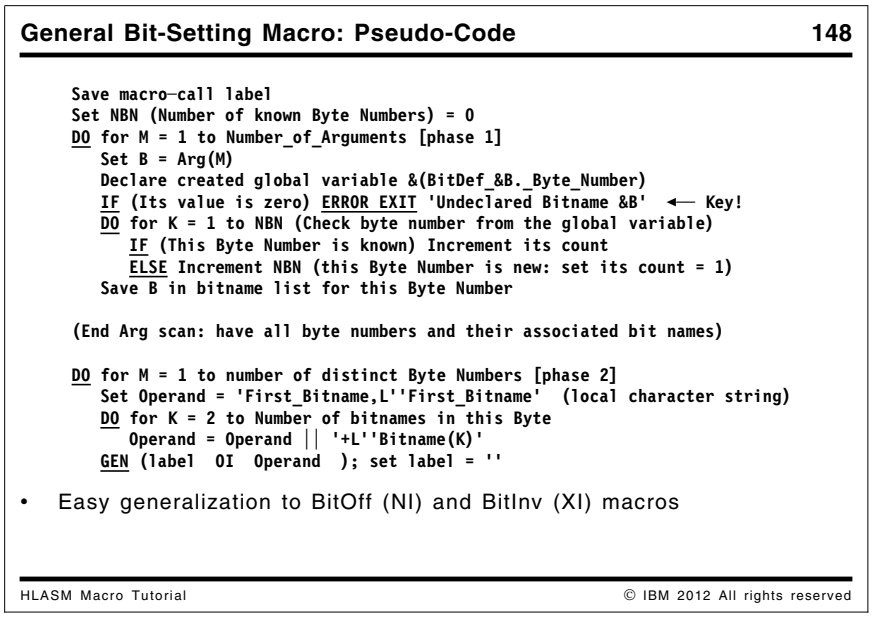
1. Check that bit names are declared (“strong typing”), and collect information about bits to be set:
   a. Number of distinct Byte_Numbers (which bytes “own” the bit names?)
   b. For each byte, the number of instances of bit names in that byte
   c. An associatively addressed variable-symbol “name table”
      - Name prefix is **BitDef_Nm_** (whatever, to avoid global-name collisions)
      - Suffix is a “double subscript”, **&ByteNumber._&InstanceNumber**
      - Value of the symbol is the bit name
2. Use the information to generate optimal instructions
   - Names and number of name instances needed to build operands

General Bit-Setting Macro: Design Details 147

- Optimize generated code for setting bits on
- Syntax: `BitOn bitname[,bitname]...`
  Example: `BitOn a,b,c,d`
- High-level pseudo-code:

```
DO for all arguments (Pass 1)

   Verify that the argument bit name was declared (check global symbol)
   IF not declared, STOP with error message for undeclared bit name

   Save argument bit names and their associated byte numbers

DO for all saved distinct byte numbers (Pass 2)

   GEN Instructions to handle argument bits belonging to each byte
```

- Pass 1 captures bit names & byte numbers, pass 2 generates code



Improved Bit-Manipulation Macros

We now explore some improved techniques for managing bit variables, including verifying that they were declared properly, and optimizing the instructions that manipulate and test them.

The macros use created variable symbols as an associatively-addressed symbol table, reducing the effort needed for table searches.

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, as illustrated in Figure 65 on page 163. The macro makes two "passes" over the supplied bit names:

- In the first pass, the bit names are read, and a global arithmetic variable `&(BitDef_&B._&ByteNo)` (where the value of `&B` is the bit name) is 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 nonzero byte number value to the variable).
- If the bit name was defined, the value of that 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 is generated for each distinct byte number that was entered in the `&BN()` array during the first pass.

- The outer loop is executed once per byte number, and 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. It 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 all instructions have been generated.

A pseudo-code description of the macro's operation is illustrated in Figure 64.

```
Save macro-call label
Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_Arguments [phase 1]
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT 'Undeclared bit name &B'
   DO for K = 1 to NBN (Check byte number from the global variable)
      IF (This Byte Number is known) Increment its count
      ELSE Increment NBN (this Byte Number is new: set its count = 1)
   Save B in bit name list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)

DO for M = 1 to NBN [phase 2]
   Set Operand = 'First_Bitname,L''First_Bitname'
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   GEN (label  OI  Operand  ); set label = ''
```

Figure 64. Bit-handling macros: set bits on: pseudo-code

General Bit-Setting Macros: Set Bits ON **149**

- 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
          Aif   ('&B' eq '').Null       Check for null item
          GblA  &(BitDef_&B._ByteNo)    Declare GBLA for 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
```


General Bit-Setting Macros: Set Bits ON ... 150

```
.*      - - -     (continuation)
&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  Matching BN, bump count of bits in this byte
         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
```

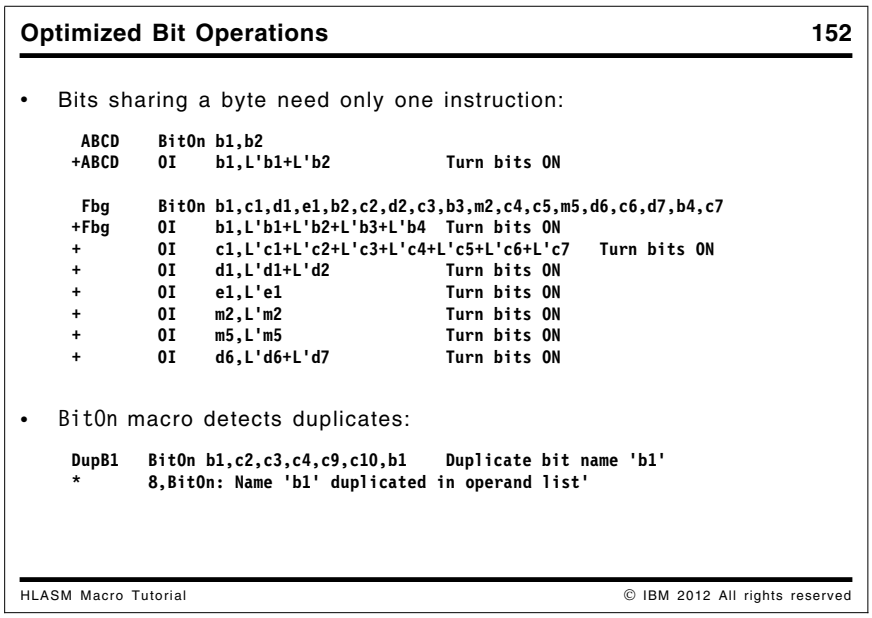

General Bit-Setting Macros: Set Bits ON ... 151

```
   .*      - - -     (continuation)
   .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)' 1st 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 "+L'bitname" 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
   .Null    MNote 8,'BitOn: Null argument at position &M.'
   .Done    MEnd
```


The definition of the BitOn macro is shown in Figure 65 on page 163.

```
        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
        Aif   ('&B' eq '').Null       Check for null item
        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  Matching BN, bump count of bits in this byte
        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)' 1st 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 +L'bitname 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
.Null   MNote 8,'BitOn: Null argument at position &M.'
.Done   MEnd
```

Figure 65. Bit-handling macros: set bits on



Some examples of calls to the BitOn macro are illustrated in the figure below. In each case, the macro generates the minimum number of instructions necessary.

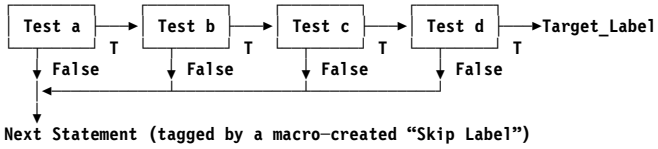
```
 ABCD    BitOn b1,b2
+ABCD    OI    b1,L'b1+L'b2            Turn bits ON

 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
```

Figure 66. 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 53 on page 147 and Figure 54 on page 147), and is left as an exercise for the reader.

General "Branch if Bits On" Macro: Design 153

- Function: branch to target if all named bits are on
- Syntax: BBitOn (bitlist),target
  Example: BBitOn (a,b,c,d),Label
- Optimize generated code using data created by BitDef
- If more than one byte is involved, need "skip-if-false" branches



- Need only one test instruction for multiple bits in a byte!

HLASM Macro Tutorial

Using Declared Bit Names in a BBitOn Macro

The BBitOn macro branches to a target label if *all* the specified bit names are on, using the minimum number of instructions. The calling syntax is:

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

It also accepts a single non-parenthesized bit name for the first argument.

This macro requires 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   *-1,X'01'           Allocate BitA
BitB      Equ   *-1,X'20'           Allocate BitB
*
          TM    BitA,L'BitA+L'BitB  Test BitA and BitB
          BO    XX                  Branch if both are ON
```

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

```
          DC    B'0'
BitA      Equ   *-1,X'01'           Allocate BitA
          DC    B'0'
BitB      Equ   *-1,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 true
          TM    BitB,L'BitB         BitA is 1; check BitB
          BO    XX                  Branch to XX if both are true
Not_True  DC    OH'0'               Label holder for 'skip target'
BitB      Equ   *-1,X'20'           Allocate BitB
```

This situation is illustrated in the following figure:

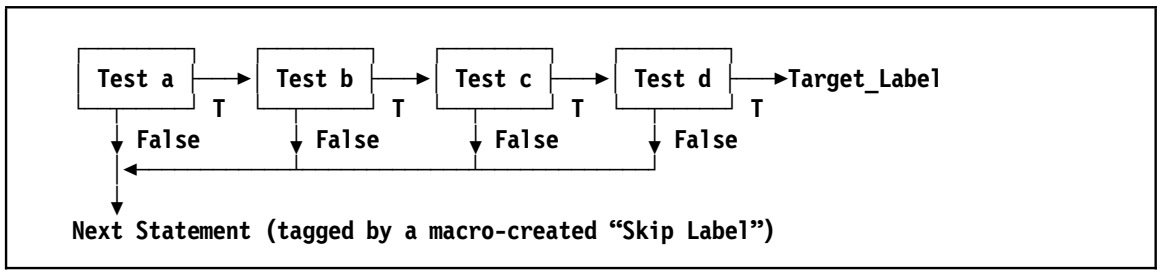


Figure 67. Bit-handling macros: branch if bits are on (flow diagram)

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 is extracted, and the same list of variables is constructed. The second “pass” is different:

- If more than one pair of test and branch instructions will be generated, a “not true” or “skip” label must be used for all branches except the last, and that 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.

General “Branch if Bits On” Macro: Pseudo-Code 154

```
Save macro-call label; Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_1st-Arg_Items [phase 1]
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT, undeclared bitname
   DO for K = 1 to NBN (Check byte number from the global variable)
      IF (This Byte Number is known) Increment its count
      ELSE Increment NBN (this Byte Number is new: set its count = 1)
   Save B in bit name list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)

Create Skip_Label (using &SYSNDX)
DO for M = 1 to NBN [phase 2]
   Set Operand = 'First_Bitname,L''First_Bitname'  (first operand)
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   IF (M < NBN) GEN (label TM Operand ; BNO Skip_Label); set label = ''
   ELSE         GEN (label TM Operand ; BO  Target_label)
IF (NBN > 1) GEN (Skip_Label DS 0H)
```

HLASM Macro Tutorial

A pseudo-code description of the BBitOn macro is shown in Figure 68 on page 167.

```
Save macro-call label; Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_1st-Arg_Items
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT, undeclared bit name
   DO for K = 1 to NBN (Check byte number from the global variable)
      IF (This Byte Number is known) Increment its count
      ELSE Increment NBN (this Byte Number is new: set its count = 1)
   Save B in bit name list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)

Create Skip_Label (using &SYSNDX)
DO for M = 1 to NBN
   Set Operand = 'First_Bitname,L''First_Bitname'
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   IF (M < NBN) GEN (label TM Operand ; BNO Skip_Label); set label = ''
   ELSE         GEN (label TM Operand ; BO  Target_label;Skip_Label DS 0H)
IF (NBN > 1) GEN (Skip_Label DS 0H)
```

Figure 68. Bit-handling macros: branch if bits are on: pseudo-code

General Bit-Handling Macros: Branch if Bits On **155**

- BBitOn macro optimizes generated instructions (most error checks omitted)
- Two “passes” over bit name list:
  1. Scan, check, and save names, determine byte numbers (as in BitOn)
  2. Generate optimized tests and branches;
     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)
```

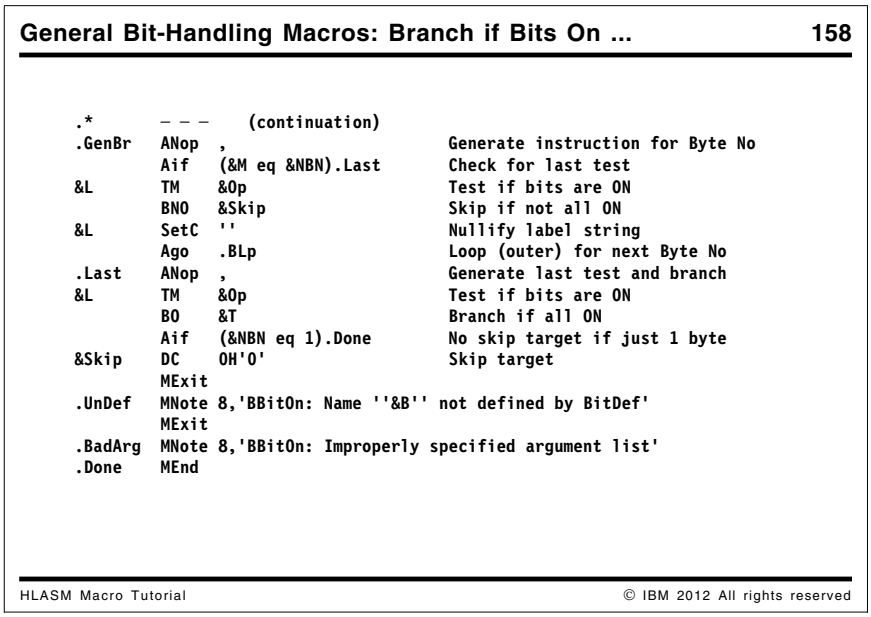
HLASM Macro Tutorial

General Bit-Handling Macros: Branch if Bits On ... 156

```
.*       - - -    (continuation)
&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)
```


General Bit-Handling Macros: Branch if Bits On ... 157

```
.*       - - -    (continuation)
.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)
```

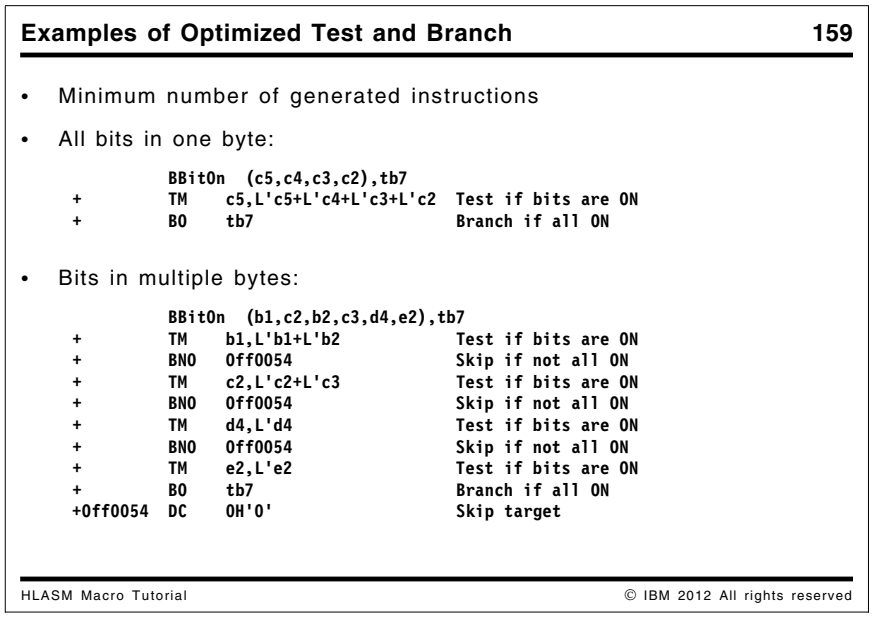
The actual BBitOn macro definition is shown in Figure 69.

```
        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
```

Figure 69 (Part 1 of 2). Bit-handling macros: macro to branch if bits are on

```
.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
.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 69 (Part 2 of 2). Bit-handling macros: macro to branch if bits are on



Some examples of calls to the BBitOn macro showing the minimum number of generated instructions are in the following figure:

```
 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

          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 70. Bit-handling macros: examples of calls to BBitOn macro

The extension of the BBitOn macro to a similar BBitOff macro is simple, and is left as an exercise. This full set of macros can be used to define, manipulate, and test bit flags with reliability and efficiency.

An interesting generalization of the BBitOn macro might be a modification causing a branch to the Target_Label if *any* bit in the first-argument list is "on". (Remember that the macro in Figure 69

on page 169 branches to the target only if *all* bits are on.) Try adding a `Type=` keyword parameter to the macro definition, specifying which type of branch is desired. For example, the new keyword parameter might look like this:

```
BBitOn  (a,b,c,d),Target,Type=All          (default)
BBitOn  (a,b,c,d),Target,Type=Any
```

where the default value (`Type=All`) causes the macro to work as described above. If `Type=Any` is specified, the logic of the bit tests in the `BBitOn` macro must be modified slightly to cause a branch to the `Target_Label` if *any* of the tested bits is on. This situation is illustrated in the following figure:

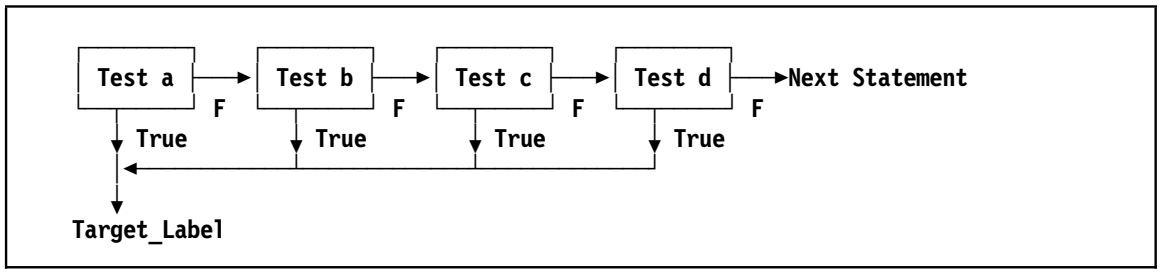


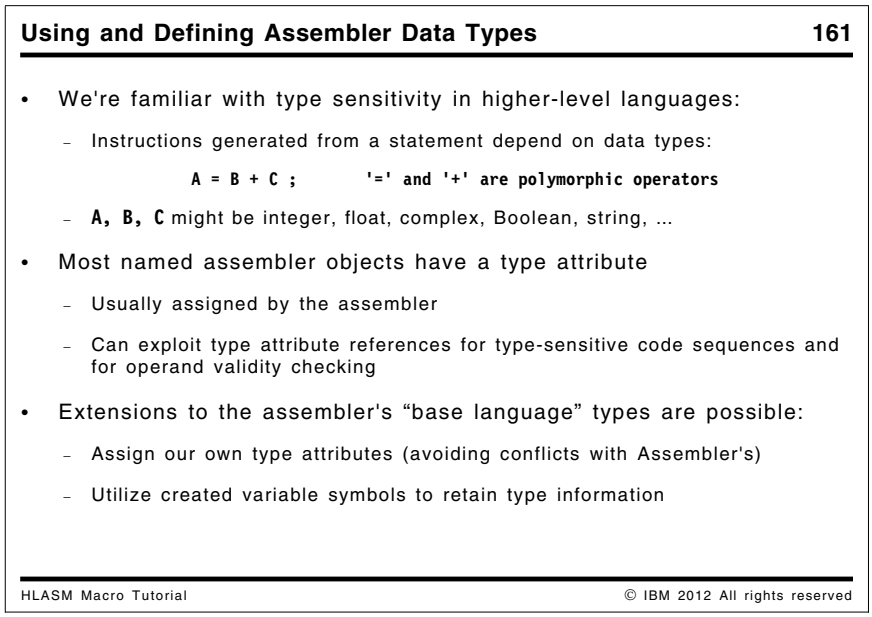
Figure 71. Bit-handling macros: branch if any bits are on (flow diagram)

In this "`Type=Any`" case, no `Skip_Label` is needed!

Case Study 8: Utilizing the Assembler's Data Types **160**

- Overview of data typing
- Case Study 8a: use operand type attribute to generate correct literal types
  - Use base-language assembler-generated type attributes
- Case Study 8b: create macros to check conformance of instructions and operand types
  - Shortcomings of assembler-assigned type attributes
  - Extension: instruction vs. operand vs. register consistency checking
- Case Study 8c: declare user data types and "operators" on them
  - User-assigned (and assembler-maintained) data types

HLASM Macro Tutorial



Case Study 8: Utilizing The Assembler's Data Types

One of the most useful features of the macro language is that you can write macros whose behavior depends on the types of its arguments. A 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 these, as in PL/I), each possibly having different lengths or precisions. We will see that macros offer the same flexibility.

These case studies show how macros can be used to provide increasingly powerful levels of control over generated code.

- Case study 8a uses the assembler's native type attributes to determine the type of literal to use in an instruction.
- Case study 8b creates macros that check for consistency between instructions and their operands, utilizing the AINSERT statement to simplify macro creation.
- Case study 8c uses user-defined type attributes for declaring "abstract types" for variables, and illustrates how to use such abstract types to generate instructions with encapsulation of the types for use by private methods.

Base-Language Type Sensitivity: Simple Polymorphism **162**

- Intent: INCR macro increments a numeric **var** by a constant **amt** (or 1)
  Syntax: INCR var[,amt] (default amt=1)
- Usage examples:

```
Day    DS    H          Type H: Day of the week
Rate   DS    F          Type F: Rate of speed
MyPay  DS    PL6        Type P: My salary
Dist   DS    D          Type D: A distance
Wt     DS    E          Type E: A weight
WXY    DS    X          Type X: Type not valid for INCR macro
*
CC     Incr  Day              Add 1 to Day
DD     Incr  Rate,-3          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
       Incr  WXY,2            Test with unsupported type
```

- Use **var**'s assembler type attribute to create compatible literals
  - type of **amt** guaranteed to match type of **var**

Base-Language Type Sensitivity: Simple Polymorphism ... **163**

- Supported types are numeric: H, F, E, D, P

```
         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,  (check base language types)  X
               ('&T' eq 'H' or '&T' eq 'D' or '&T' eq 'E').T   Valid types
         MNote 8,'&SysMac.: Cannot use type ''&T'' with ''&V''.'
         MExit
.F       ANOP  ,                  Type of &V is F
&Op      SetC  ''                 Null opcode suffix for F (no LF opcode)
.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 as typed literal
         ST&Op &Reg,&V            Store incremented value
         MExit
.P       ANOP  ,                  Type of &V is P
&Lab     AP    &V,=P'&I'          Incr packed variable with P-type literal
         MEnd
```


reference the TYPECHEK macros?

Case Study 8a: Type Sensitivity with Simple Polymorphism

The assembler's assignment of type attributes to most forms of declared data lets us write macros that utilize that type information to make decisions about instructions to be generated.

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. Because we know the assembler type assigned to the variable, we can use that same type for the constant increment.

```
       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,  (check base language types)  X
               ('&T' eq 'H' or '&T' eq 'D' or '&T' eq 'E').T   Valid types
       MNote 8,'&SysMac.: Cannot use type ''&T'' with ''&V''.'
       MExit
.F     ANOP  ,                   Type of &V is F
&Op    SetC  ''                  Null opcode suffix for F (no LF opcode)
.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 72. 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 appropriate 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.

If the symbol naming the data to be incremented, a MNOTE message is issued, using the &SysMac system variable symbol to capture the name of the issuing macro.

This macro illustrates a form of *polymorphism*: 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           Type H: Day of the week
Rate   DS    F           Type F: Rate of speed
MyPay  DS    PL6         Type P: My salary
Dist   DS    D           Type D: A distance
Wt     DS    E           Type E: A weight
WXY    DS    X           Type X: Type not valid for INCR macro
*
CC     Incr  Day               Add 1 to Day
DD     Incr  Rate,-3           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
       Incr  WXY,2             Test with unsupported type
+ *** MNOTE ***  8,INCR: Cannot use type 'X' with 'WXY'.
       MEnd
```

Figure 73. Examples: macro type sensitivity to base language types

Base-Language Type Sensitivity: Generated Code 164

- Code generated by INCR macro (see slide 162)

```
 CC     Incr  Day                Add 1 to Day
+CC     LH    0,Day              Fetch variable to be increment
+       AH    0,=H'1'            Add requested increment
+       STH   0,Day              Store incremented value
 DD     Incr  Rate,-3,Reg=15       Decrease rate by 3
+DD     L     15,Rate            Fetch variable to be increment
+       A     15,=F'-3'          Add requested increment
+       ST    15,Rate            Store incremented value
        Incr  MyPay,150.50         Add 150.50 to my salary
+       AP    MyPay,=P'150.50'   Increment variable
 JJ     Incr  Dist,-3.16227766     Decrease distance by SQRT(10)
+JJ     LD    0,Dist             Fetch variable to be increment
+       AD    0,=D'-3.16227766'  Add requested increment
+       STD   0,Dist             Store incremented value
 KK     Incr  Wt,-2E4,Reg=6        Decrement weight by 10 tons
+KK     LE    6,Wt               Fetch variable to be increment
+       AE    6,=E'-2E4'         Add requested increment
+       STE   6,Wt               Store incremented value

        Incr  WXY,2                Test with unsupported type
+ *** MNOTE ***  8,INCR: Cannot use type 'X' with 'WXY'.
```

HLASM Macro Tutorial

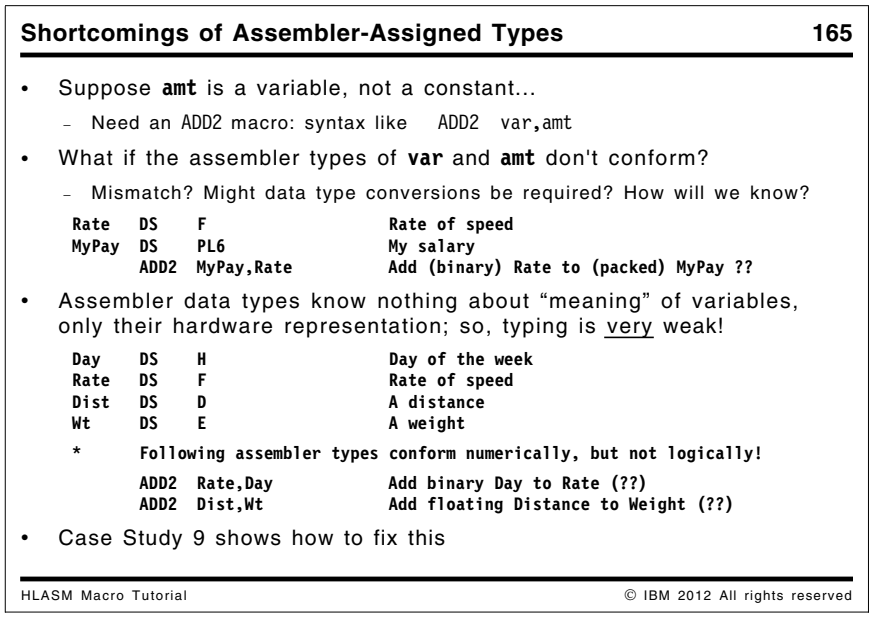
Examples of the code generated by the INCR macro are shown in Figure 74.

```
 CC     Incr  Day                Add 1 to Day
+CC     LH    0,Day              Fetch variable to be increment
+       AH    0,=H'1'            Add requested increment
+       STH   0,Day              Store incremented value

 DD     Incr  Rate,-3,Reg=15       Decrease rate by 3
+DD     L     15,Rate            Fetch variable to be increment
+       A     15,=F'-3'          Add requested increment
+       ST    15,Rate            Store incremented value

        Incr  MyPay,150.50         Add 150.50 to my salary
+       AP    MyPay,=P'150.50'   Increment variable

 JJ     Incr  Dist,-3.16227766     Decrease distance by sqrt(10)
+JJ     LD    0,Dist             Fetch variable to be increment
+       AD    0,=D'-3.16227766'  Add requested increment
+       STD   0,Dist             Store incremented value

 KK     Incr  Wt,-2E4,Reg=6        Decrement weight by 10 tons
+KK     LE    6,Wt               Fetch variable to be increment
+       AE    6,=E'-2E4'         Add requested increment
+       STE   6,Wt               Store incremented value

        Incr  WXY,2                Test with unsupported type
+ *** MNOTE ***  8,INCR: Cannot use type 'X' with 'WXY'.
```

Figure 74. Examples: macro type sensitivity: incr macro generated code

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

A useful exercise is to modify the INCR macro to use instructions with immediate operands wherever possible.



Shortcomings of Assembler-Assigned Types

While many benefits are achievable from utilizing assembler type attributes, they do not provide as reliable a checking mechanism as we might need. If 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: they don't have the same assembler-assigned type attribute. For example, let some variables be defined as in Figure 72 on page 175:

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

   Then, if we write a macro call like

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

   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 in-line code or with a call to a conversion subroutine. However, as the number of allowed types grows, the number of needed conversions can 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" or "logical" 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 speed
   Dist   DS    D                 A distance
   Wt     DS    E                 A weight
   ```

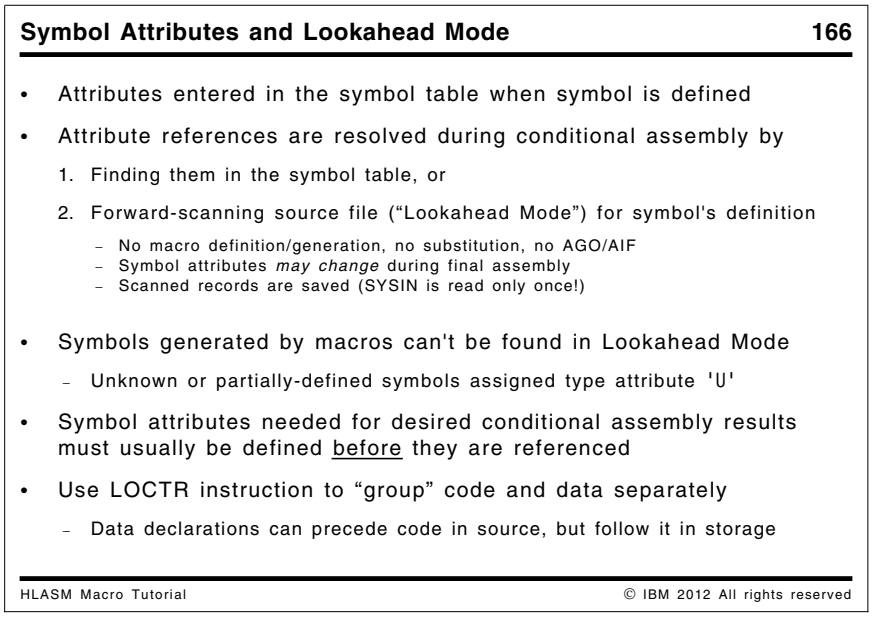
   Then, it is clear that we can write simple macros to implement these additions:

   ```
          ADD2  Rate,Day          Add binary Halfword to Fullword
          ADD2  Dist,WT           Add floating Distance to Weight
   Wt     DS    E                 A weight
   ```

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

But 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”. Neither of these operations makes sense in the real world, even though a computer will blindly add the numbers representing these quantities.

This lack of programmer-defined meaning (sometimes called “strong typing”) can be a serious shortcoming of the Assembler Language, but it is easily overcome by defining and using macros, and by use of Program Attributes, as described in “Case Study 9: Using Program Attributes” on page 203.



Symbol Attributes and Lookahead Mode

There is a potential problem in utilizing attribute information in conditional assembly: the attributes might not be available at the time they are needed. A statement defining a symbol might occur later in the source file than a macro that references the symbol's attributes. When the assembler notes that the symbol's attributes are currently unknown, it begins a forward scan of the primary source file called “Lookahead Mode”.

In Lookahead mode, all scanned statements are saved (so that the primary input file is read only once). No macros are encoded or expanded, and no AIF or AGO statements are obeyed. Symbol definitions are entered in the symbol table with a flag indicating that their attributes are “partially defined”. When the assembly completes, the attributes of a symbol might be different from the attributes assigned during Lookahead mode.

The usual solution is to expand all macros that generate necessary symbol definitions *before* any other macros that reference their attributes. While this might seem to force data to be generated in a module ahead of (or mixed with) the code, the assembler's LOCTR statement helps you group and rearrange related segments of the object code.

The LOCTR Statement

The LOCTR statement lets you define named groups of statements in such a way that all object code generated from statements in each named group will eventually be combined with other statements from groups with the same name, even though the various groups with other names are scattered among one another in the source file. The following figure illustrates how LOCTR works:

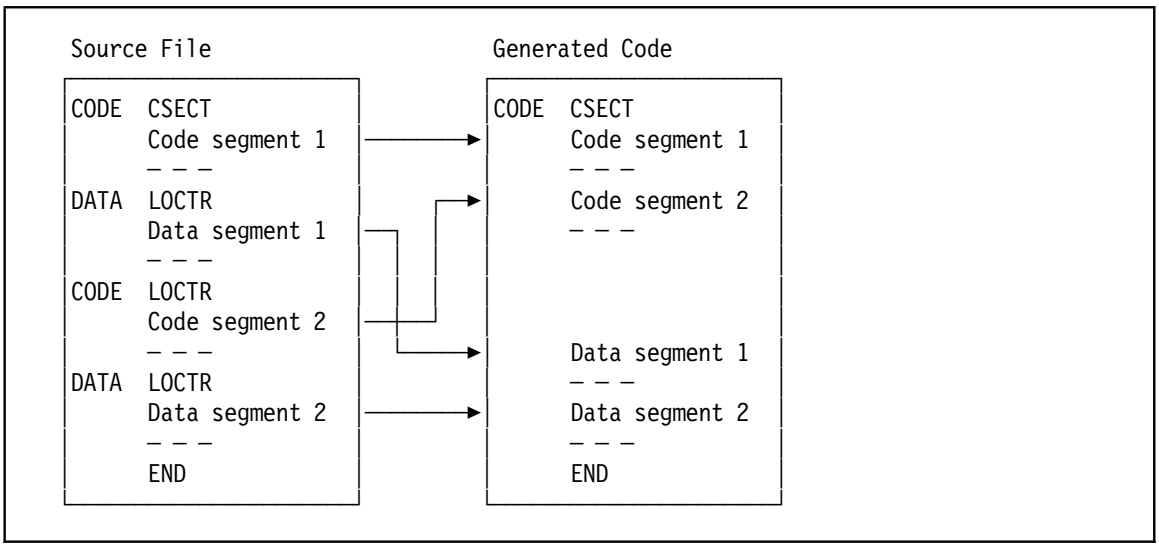


Figure 75. Using the LOCTR statement to “group” code and data

LOCTR groups can help your macros derive correct type attribute information when needed.

Case Study 8b: Simple Instruction-Operand Type Checking **167**

- Check the second operand of the A instruction
  - Accept type attributes type F, A, or Q; note others and proceed
- First, save the assembler's definition of instruction “A”
  - OPSYN copies or nullifies a mnemonic definition

```
My_A     OpSyn  A          Save definition of A as My_A
A        OpSyn  ,          Nullify assembler's definition of A
```

- Second, define a macro named “A” that eventually calls My_A
- Macro “A” checks the second operand for type F, A, or Q

```
         Macro
&L       A      &R,&X
         AIF    (T'&X eq 'F' or T'&X eq 'A' or T'&X eq 'Q').OK
         MNote  1,'Note! Second operand type not F, A, or Q.'
.OK      ANop
&L       My_A   &R,&X
         MEnd
```

- Allowed types are “hard coded” in this macro

HLASM Macro Tutorial

Case Study 8b: Instruction-Operand Type Checking

The Assembler's type attribute values can be used to check for consistency between data types and instruction types, as the following example will show. You may want to ensure that an instruction in your application references only operands that are likely to be natural for that instruction.

Suppose we wish to check the second operand of the Add (A) instruction to verify that its type is only F, A, or Q. First, we use OPSYN to preserve the original definition of the A opcode as My_A:

```
My_A     OpSyn  A          Save assembler's definition of A
A        OpSyn  ,          Nullify assembler's definition of A
```

Then, we can write a macro like this:

```
         Macro
&L       A      &R,&X
         AIF    (T'&X eq 'F' or T'&X eq 'A' or T'&X eq 'Q').OK
         MNote  1,'Note! Second operand type not F, A, or Q.'
.OK      ANop
&L       My_A   &R,&X
         MEnd
```

We used the OPSYN statement to preserve the assembler's definition of the A instruction, and then to remove A from the assembler's opcode table, where it will later be replaced by the macro definition of A. The generated machine language instruction will be the same as it would be for the assembler's "native" A instruction. The result of using this macro might look like the following:

```
          A     1,D2
  *** MNOTE *** 1,Note! Second operand type not F, A, or Q.
+         My_A   1,D2
 *
 D2       DC    D'2'
```

To extend this example, we might choose to permit type attributes F and D (fullword and doubleword constants), A, Q, and V (fullword address constants), and X ("almost anything"), and flag uses of other types with a low-level message.

We will examine some generalizations of this simple example to show how the assembler can provide very useful forms of consistency checking of instructions, operands, and registers.

Base-Language Type Sensitivity: General Type Checking **168**

- Intent: compatibility checking between instruction and operand types
- Define TypeChek macro to request type checking
  Syntax: `TypeChek mnemonic,valid_types`
- Call TypeChek with mnemonic to check, and its allowed types

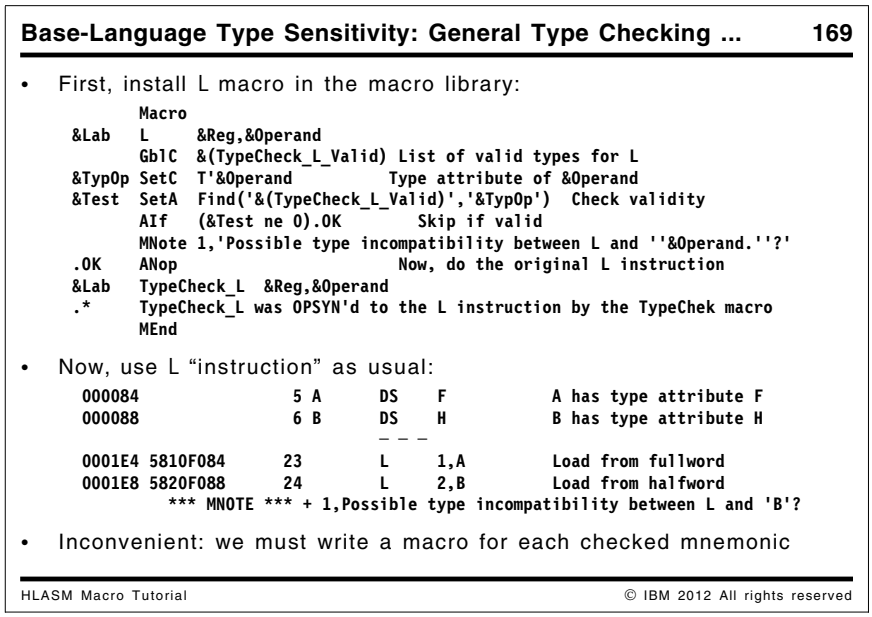
```
TypeChek  L,'ADFQVX'    Allowed types: AQV (adcons), D, F, X
```

- Sketch of a general macro to initiate type checking for one mnemonic:

```
         Macro
         TypeChek  &Op,&Valid    Mnemonic, set of valid types
         GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op)
 &(TypeCheck_&Op._Valid)  SetC  '&Valid'   Save valid types for this opcode
 TypeCheck_&Op.           OpSyn &Op.    Save original opcode definition
 &Op     OpSyn ,                 Disable previous definition of &Op
 .*      MNote *,'Mnemonic ''&Op.'' valid types are ''&(TypeCheck_&Op._Valid).''.'
         MEnd
```

- Can be generalized to multiple mnemonics
  - But: requires creating macros for each mnemonic...

HLASM Macro Tutorial



Instruction-Operand Type Checking

First, we will define a TypeChek macro whose arguments are an instruction mnemonic and a set of allowed types. (This approach is more general than strictly needed, but it will allow easy generalization to multiple mnemonics with the same set of permitted operand types.) This macro will define two created variable symbols, &(TypeCheck_&Op._Valid) with the types, and &(TypeCheck_&Op) with a substituted name TypeCheck_&Op for saving the meaning of the mnemonic to be checked.

```
       Macro
       TypeChek  &Op,&Valid      Mnemonic, set of valid types
       GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op)
&(TypeCheck_&Op._Valid)  SetC  '&Valid'   Save valid types for this opcode
TypeCheck_&Op.           OpSyn &Op.    Save original opcode definition
&Op    OpSyn ,                  Disable previous definition of &Op
.*     MNote *,'Mnemonic ''&Op.'' valid types are ''&(TypeCheck_&Op._Valid).'''
       MEnd
```

Figure 76. Instruction-operand type checking: typechek macro

This definition of the TypeChek macro may be called to define checked types for other mnemonics, also. When the TypeChek macro is called:

```
TypeChek  L,'ADFQVX'    Allowed types: AQV (adcons), DF, X
```

it will nullify the Assembler's definition of the L mnemonic.

Thus, the second step is to define an L macro which will be added to the macro library used before the type-checked application is assembled.

```
        Macro
&Lab    L     &Reg,&Operand
        GblC  &(TypeCheck_L_Valid) List of valid types for L
&TypOp  SetC  T'&Operand          Type attribute of &Operand
&Test   SetA  Find('&(TypeCheck_L_Valid)','&TypOp')  Check validity
        AIf   (&Test ne 0).OK       Skip if valid
        MNote 1,'Possible type incompatibility between L and ''&Operand.''?'
.OK     ANop                       Now, do the original L instruction
&Lab    TypeCheck_L  &Reg,&Operand
        MEnd
```

Figure 77. Instruction-operand type checking: "instruction" macro

Now, when the L "instruction" is used, it will actually invoke the L *macro*, which then checks the type of the operand and issues an MNOTE message in case of a mismatch. Finally, the correct instruction (whose true definition was saved by the TypeChek macro as TypeCheck_L) is generated, with the same operands as the call to the L macro.

```
000084                5 A      DS    F           A has type attribute F
000088                6 B      DS    H           B has type attribute H
                               - - -
0001E4 5810F084      23        L     1,A         Load from fullword
0001E8 5820F088      24        L     2,B         Load from halfword
         *** MNOTE *** +    1,Possible type incompatibility between L and 'B'?
```

Figure 78. Instruction-operand type checking: examples

As the above example illustrates, using an operand of a "non-approved" type will be flagged.

While useful, this scheme requires writing a separate macro for each instruction to be type-checked. Installing the macros in a library needs to be done only once, but their presence could cause problems if other users accidentally reference the macros when no type checking was intended. These difficulties can be overcome by generalizing the TypeChek macro, and by finding a way for the instruction-replacement macros to be generated automatically.

Base-Language Type Checking: Extensions 170

- Previous technique requires a macro for each checked instruction
  - Not difficult to write, just a lot of repetitive work
  - Macros must be available in a library
    - If not using **TypeChek**, don't use the instruction-replacement macros!
- Better:
  - Specify a list of instructions to be checked and their allowed operand types:

```
TypeChek  (L,ST,A,AL,S,SL,N,X,O),'ADFQVX'
```

  - The **TypeChek** macro generates the replacement macros as needed
  - Using AINSERT!

HLASM Macro Tutorial

Instruction-Operand Type Checking (Generalized)

Obviously, we could define the list of allowed types in the L macro itself, and eliminate the TypeChek macro; but we will still need statements like

```
TypeChek_L Opsyn L          Save original definition of L
L          OpSyn ,          Null operand eliminates 'L' mnemonic
```

to "nullify" the assembler's built-in definition, for each mnemonic to be checked.

The scheme illustrated here can be generalized in many ways. For example, the TypeChek macro could accept a list of mnemonics that share the same set of valid types:

```
TypeChek  (L,ST,A,AL,S,SL,N,X,O),'ADFQVX'
```

which allows handling mnemonics in related groups.

One possibility is to have the TypeChek macro generate a macro for each instruction to be checked; each macro has the same pattern for a given class of mnemonics. Unfortunately, one key capability of the original macro and conditional assembly language was missing: when a macro is defined inside another macro (so that expanding the first causes the second to become defined), values cannot be substituted from the scope of the enclosing "outer" macro definition into the statements of the enclosed "inner" macro definition. (See "Nested Macro Definitions" on page 100.) The ability to parameterize generated macros would make it much easier to create the "mnemonic" macros directly.

This shortcoming is eliminated by the AINSERT statement.

The AINSERT Statement **171**

- AINSERT allows generation of fully parameterized records

```
AINSERT 'string',[FRONT|BACK]
```

- Placed at front or back of assembler's internal buffer queue
  - String padded or truncated to form 80-byte record
- HLASM reads from the FRONT of the buffer before reading from SYSIN
  - Input from SYSIN resumes when the buffer is empty
- Operand string may contain "almost anything"

```
          AInsert  '* comment about &SysAsm. &SysVer.',BACK
 >* comment about HIGH LEVEL ASSEMBLER 9.7.72
```

  - The '>' character in "column 0" indicates an AINSERTed statement
- Use AINSERT to generate tailored, parameterized macro definitions

HLASM Macro Tutorial

The AINSERT Statement

Sometimes it is useful to exercise greater control over the order in which generated statements will be processed. The AINSERT statement lets you generate complete statements in almost *any* order, and removes many of the restrictions associated with encoding.

The syntax of AINSERT is

```
AINSERT 'string',[FRONT|BACK]
```

The first operand may contain points of substitution.

The assembler maintains an internal buffer queue into which AINSERT strings are placed, padded or truncated to an 80-byte record. Each record is placed either at the front or back end of the buffer, depending on the second AINSERT operand. When the assembler is ready to read records from the primary input (SYSIN) file, it first checks the AINSERT buffer: if it's not empty, records are taken from the buffer until it is empty, and input then resumes from the primary input stream.

AINSERT removes many limitations on substitutable fields:

```
          AInsert  '* comment about &SysAsm. &SysVer.',BACK
>* comment about HIGH LEVEL ASSEMBLER 9.7.72
          AInsert  '* Assembled &SYSDatC.',BACK
+         AInsert  '* Assembled 20350705',BACK
>* Assembled 20350705
```

where the ">" character in the listing is the assembler's indication of a statement inserted into the statement stream via AINSERT. (Remember that AINSERTed statements are treated as part of the *primary* input stream, and are not within the body of any existing macro.)

We will now use AINSERT to generate the desired instruction-replacement macros.

Base-Language Type Checking: Generated Macros 172

- Generate each type-checking macro using AINSERT

```
TypeChek  (L,ST,A,AL,S,SL,N,X,0),'ADFQVX'   Desired style
```

- Sketch of revised inner loop of TypeChek macro:

```
 &Op    SetC  '&Ops(&K)'          Pick off K-th mnemonic
 &Op    OpSyn ,                   Disable previous definition of &Op
 .*   Generate macro to redefine &Op for type checking (note paired ', &)
  AInsert ' Macro ',BACK
  AInsert '&&Lab &Op. &&Reg,&&Opd',BACK
  AInsert ' GblC &&(TypeCheck_&Op._Valid)',BACK
  AInsert '&&T0 SetC T''&&Opd ',BACK
  AInsert '&&T SetA Find(''&&(TypeCheck_&Op._Valid)'',''&&T0'')',BACK
  AInsert ' AIf (&&T ne 0).OK ',BACK
  AInsert ' MNote 1,''Possible type conflict between &Op and &&Opd?'''',B*
               ACK
  AInsert '.OK ANop ',BACK
  AInsert '&&Lab TypeCheck_&Op &&Reg,&&Opd ',BACK  Assumes previous OPSYN
  AInsert ' MEnd ',BACK
 .*    End of macro generation
```

- Compare to "hand-coded" L macro (slide 169)

HLASM Macro Tutorial

In Figure 77 on page 182 we saw how the "instruction" macro was created for a single mnemonic (L). We can use the AINSERT statement in the TypeChek macro to create macros for *each* mnemonic.

These examples have used RX-type instructions to show how to set up a type-checking macro. Assuming that we may want to generalize to other instruction types, we first write a TypChkRX macro (based on the TypeChek macro illustrated above). The same techniques are used, and generate the needed macros for each mnemonic:

```
       Macro
       TypChkRX  &Ops,&Valid
&K     SetA  1                  Count of mnemonics
.Prcss ANop                     Process each opcode in &Ops
&Op    SetC  '&Ops(&K)'         Pick off K-th mnemonic
       GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op.)
&(TypeCheck_&Op._Valid) SetC  '&Valid'         Save valid types
&(TypeCheck_&Op.)       SetC  'TypeCheck_&Op.' Create new mnemonic
&(TypeCheck_&Op.)       OpSyn &Op              Save original mnemonic
&Op    OpSyn ,                  Disable previous definition of &Op
  MNote *,'Mnemonic &Op. valid types are &(TypeCheck_&Op._Valid)'

.*   Generate macro to redefine &Op for type checking
 AInsert ' Macro ',BACK
 AInsert '&&Lab &Op. &&Reg,&&Opd',BACK
 AInsert ' GblC &&(TypeCheck_&Op._Valid)',BACK
 AInsert '&&TO SetC T''&&Opd ',BACK
 AInsert '&&T SetA Find(''&&(TypeCheck_&Op._Valid)'',''&&TO'')',BACK
 AInsert ' AIf (&&T ne 0).OK ',BACK
 AInsert ' MNote 1,''Possible type conflict between &Op and &&Opd?''',B*
               ACK
 AInsert '.OK ANop ',BACK
 AInsert '&&Lab TypeCheck_&Op &&Reg,&&Opd ',BACK  Assumes previous OPSYN
 AInsert ' MEnd ',BACK
.*    End of macro generation
&K     SetA  &K+1               Increment &K
       AIf   (&K le N'&Ops).Prcss  If not finished get next opcode
       MEnd
```

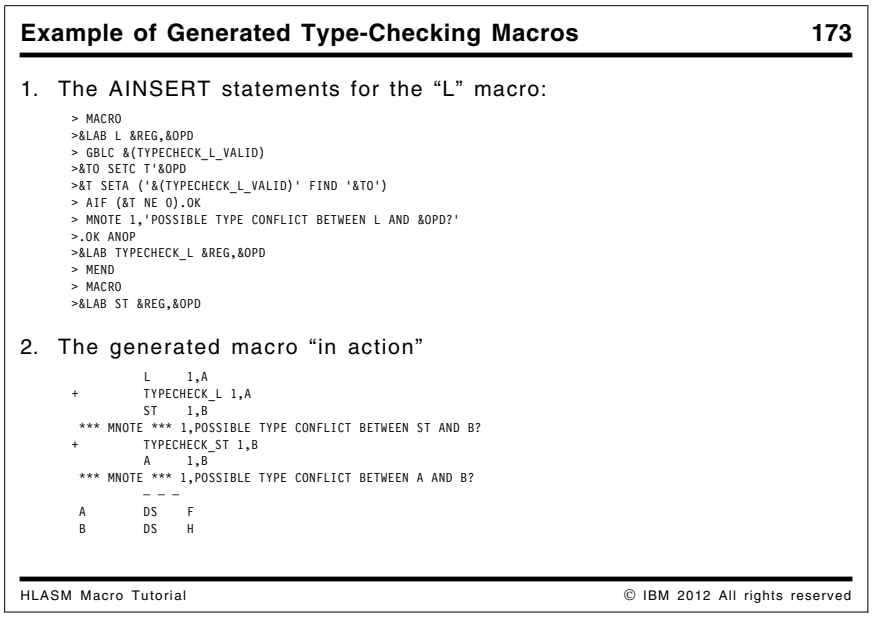
Figure 79 (Part 1 of 2). Instruction-operand type checking: generated macro definitions

A call to the TypChkRX macro causes a “mnemonic” macro to be created for each mnemonic in the first operand:

```
TypChkRX  (L,A,ST),'ADFQVX'  Allowed types: AQV (adcons), D, F, X
```

will generate macros for mnemonics L, A, and ST, each of which validates that its operand type is one of the allowed types.

A minor detail worth noting: the second operand of the macro is enclosed in apostrophes, in case we may want to include user-defined (lower-case or special-character) types in the &Valid operand. If the user specifies the COMPAT(MACROCASE) option, unquoted lower-case letters would be converted internally to upper case before being made available to the expansion of the macro.



The following figure illustrates the operation of the TypChkRX macro. (Many repetitive lines were removed; if you don't want all the AINSERT statements and AINSERTed records to appear in your listing, you could modify the macro to generate PRINT OFF and PRINT ON statements in appropriate places.)

```
         TypChkRX  (L,ST,A,AL,S,SL,N,X,O),ADFQVX
+TypeCheck_L             OpSyn L                Save original opcode
+L      OpSyn ,                   Disable previous definition of &Op
+*,Mnemonic L valid types are ADFQVX
+ AInsert ' Macro ',BACK
+ AInsert '&&Lab L &&Reg,&&Opd',BACK
+ AInsert ' GblC &&(TypeCheck_L_Valid)',BACK
+ AInsert '&&TO SetC T''&&Opd ',BACK
+ AInsert '&&T SetA (''&&(TypeCheck_L_Valid)'' Find ''&&TO'')',BACK
+ AInsert ' AIf (&&T ne 0).OK ',BACK
+ AInsert ' MNote 1,''Possible type conflict between L and &&Opd?''',BACX
+               K
+ AInsert '.OK ANop ',BACK
+ AInsert '&&Lab TypeCheck_L &&Reg,&&Opd ',BACK
+ AInsert ' MEnd ',BACK
+TypeCheck_ST            OpSyn ST               Save original opcode
+ST     OpSyn ,                   Disable previous definition of &Op
+*,Mnemonic ST valid types are ADFQVX

 ... etc. etc.
 ... many AINSERT statements later, the assembler reads the buffer:
```

Figure 80. Generated statements from TYPCHKRX macro

```
 ... etc. etc.
 ... many macro definitions later, the assembler reads the input file:

          L     1,A
+         TypeCheck_L 1,A
          ST    1,B
 *** MNOTE *** 1,Possible type conflict between ST and B?
+         TypeCheck_ST 1,B
          A     1,B
 *** MNOTE *** 1,Possible type conflict between A and B?
          - - -
 A        DS    F
 B        DS    H
```

Figure 80. Generated statements from TYPCHKRX macro

User-Assigned Assembler Type Attributes 174

- We can utilize third operand of EQU for special types:

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

  - Assembler's "native" types are upper case letters, $, and '@'
  - We can use lower case letters for user-assigned types (actually, any term < 256)
- Example (extend the REGS macro, slide 103) to create a TYPEREGS macro:

  ```
  GR&N   EQU   &N,,C'g'      Assign value and type attribute 'g' for GPR
  FR&N   EQU   &N,,C'f'      Assign value and type attribute 'f' for FPR
  ```

- GRnn symbols have type attribute **'g'**, FRnn have **'f'**
- Can use type attribute to check symbols used in register operands
- Extension of instruction/operand type checking

HLASM Macro Tutorial

User-Defined Assembler Type Attributes

One can obtain some 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. The "native" type attributes assigned by the assembler are all upper-case letters, the '$' character, or the '@' character. Thus, the other values can be used for user-assigned attributes.

We will see in "Case Study 9: Using Program Attributes" on page 203 that Program Attributes provide a more powerful, flexible, and generalized symbol-attribute facility than the assembler's "native" type attribute.

A simple generalization of two previous examples will show how we could do further assembly-time checking of instruction usage. First, consider the previously defined REGS macro (see Figure 33 on page 114) that generates symbolic names to refer to various types of registers. If we

modify the EQU statements in those macros to include a user-assigned type attribute, we could (for example) assign type 'g' to general purpose registers, 'f' to floating point registers, and so forth. Then, a simple extension of the TypeChek macro (or the L macro) can be used to verify that a symbolic name used to designate a register is of the correct type.

First, in the TYPEREGS macro, the EQU statements are modified:

```
GR&N   EQU   &N,,C'g'      Assign value and type attribute 'g' for GPRs
FR&N   EQU   &N,,C'f'      Assign value and type attribute 'f' for FPRs
       - - -   etc.
```

Suppose we extend the REGS macro described in "Case Study 1: Defining Equated Symbols for Registers" on page 110 to create a TYPEREGS macro that assigns a special type attribute to the symbols naming each register. Figure 81 shows how to do this.

```
         MACRO
         TypeRegs
         AIF     (N'&SysList eq 0).Exit
&J       SetA    1                  Initialize argument counter
.GetArg  ANOP
&T       SetC    Upper('&SysList(&J)')     Pick up an argument
&N       SetA    ('ACFG' Index '&T')        Check type
         AIF     (&N eq 0).Bad     Error if not a supported type
         GBLB     &(&T.Regs_Done)  Declare global variable symbol
         AIF     (&(&T.Regs_Done)).Done Test if true already
&L       SetC    Lower('&T')       Lower case for type attribute
&N       SetA    0
.Gen     ANop    ,                 Generate Equ statements
&T.R&N   Equ     &N,,C'&L'
&N       SetA    &N+1
         AIf     (&N le 15).Gen
&(&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,'&SysMac. -- Unknown type ''&T.''.'
         MEXIT
.Done    MNOTE   0,'&sysMac. -- Previously called for type &T..'
         AGO     .Next
.Exit    MEND
```

Figure 81. Instruction-operand type checking: assigning register types

This macro assigns the same symbolic names to register symbols as before, but also assigns special type attributes that specify the type of register. These types can be used in the macros generated for each instruction type to verify correct usage.

A sample of TypeRegs generated statements is shown in the following figure.

```
         TYPEREGS   F,G
+FR0     Equ     0,,C'f'
+FR1     Equ     1,,C'f'
+FR2     Equ     2,,C'f'
    ... etc.
+FR15    Equ     15,,C'f'

+GR0     Equ     0,,C'g'
+GR1     Equ     1,,C'g'
+GR2     Equ     2,,C'g'
    ... etc.
+GR15    Equ     15,,C'g'
```

Instruction-Operand-Register Type Checking **175**

- Intent: check "typed" register names in type-checking macros
- Example: extend L macro (see slides 168 and 169)

```
        Macro
 &Lab   L     &Reg,&Operand
        GblC  &(TypeCheck_L_Valid),&(TypeCheck_L_RegType)
 &TypOp SetC  T'&Operand          Type attribute of &Operand
 &Test  SetA  Find('&(TypeCheck_L_Valid)','&TypOp')  Check validity
        AIf   (&Test ne 0).OK_Op      Skip if valid
   MNote 1,'Possible type incompatibility between L and ''&Operand.''?'
 .OK_Op ANop                       Now, do the original L instruction
 .*     Added checking for register type:
 &TypRg SetC  T'&Reg             Type attribute of &Reg
 &Test  SetA  Find('&(TypeCheck_L_RegType)','&TypRg')  Check validity
        AIf   (&Test ne 0).OKReg     Skip if valid
   MNote 1,'Possible register incompatibility between L and ''&Reg.''?'
 .OKReg ANop                       Now, do the original L instruction
 &Lab   TypeCheck_L  &Reg,&Operand
        MEnd
```

- Typical output (assuming F names a floating-point constant):

```
          L     FR4,F
 *** MNOTE *** 1,Possible type incompatibility between L and 'F'?
 *** MNOTE *** 1,Possible register incompatibility between L and 'FR4'?
```

HLASM Macro Tutorial

Instruction-Operand-Register Type Checking

After assigning user-defined type attributes to the register symbols generated by the TYPEREGS macro, the TypeChek macro (see Figure 76 on page 181) can be modified by adding a keyword parameter &RegType, with a default value that includes 'g':

```
        Macro
        TypeChek  &Op,&Valid,&RegType='gN'   Mnemonic, set of types, RegType
        GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op)
        GblC  &(TypeCheck_&Op._RegType)
&(TypeCheck_&Op._Valid) SetC '&Valid'   Save valid operand types
&(TypeCheck_&Op._RegType) SetC '&RegType'(2,K'&RegType-2) Save valid reg types
        - - - etc.
```

The default &RegType values allow self-defining terms with type attribute 'N' (that is, self-defining constants) and declared register types ('g') as register operands. As mentioned before, the &RegType operand is a quoted string, to avoid the possibility that the COMPAT(MACROCASE) option might internally convert the argument value to upper case. (Note: if you want to use the apostrophe character as the value of a user-assigned type attribute, you will need to add statements to remove the quotes from each end of the &Valid and

&RegType operands before assigning the strings to the global variables &(TypeCheck_&Op._Valid) and &(TypeCheck_&Op._RegType) respectively.)

An enhanced L macro (see Figure 77 on page 182) can then be used to validate both the register type and the operand type:

```
       Macro
&Lab   L     &Reg,&Operand
       GblC  &(TypeCheck_L_Valid),&(TypeCheck_L_RegType)
&TypOp SetC  T'&Operand          Type attribute of &Operand
&Test  SetA  Find('&(TypeCheck_L_Valid)','&TypOp')  Check validity
       AIf   (&Test ne 0).OK_Op      Skip if valid
       MNote 1,'Possible type incompatibility between L and ''&Operand.''?'
.OK_Op ANop  ,                    Now, check register validity
&TypRg SetC  T'&Reg             Type attribute of &Reg
&Test  SetA  Find('&(TypeCheck_L_RegType)','&TypRg')  Check validity
       AIf   (&Test ne 0).OKReg     Skip if valid
       MNote 1,'Possible register incompatibility between L and ''&Reg.''?'
.OKReg ANop  ,                    Now, do the original L instruction
&Lab   TypeCheck_L  &Reg,&Operand
       MEnd
```

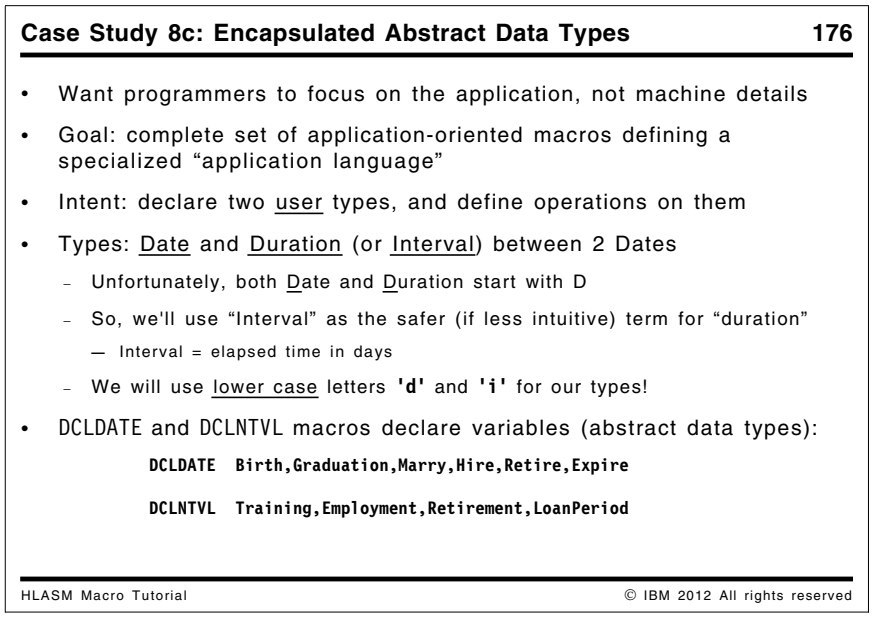
Figure 82. Instruction-operand-register type checking: “instruction” macro

This modification checks that all values provided as register operands for the L instruction are properly defined.

An example of the output of these macros is shown in the following figure:

```
        TYPEREGS   F,G          Create typed names for registers
        TYPCHKRX   L,FDEAVQX,RegType='gN'  L instruction valid types

        L     1,A               Register operand self-defining
+       TypeCheck_L  1,A
        L     GR1,C
 *** MNOTE *** 1,Possible type incompatibility between L and 'C'?

+       TypeCheck_L  GR1,C
        L     FR2,D             Floating-point register
 *** MNOTE *** 1,Possible register incompatibility between L and 'FR2'?
+       TypeCheck_L  FR2,D

        L     FR4,F             Float register and invalid operand
 *** MNOTE *** 1,Possible type incompatibility between L and 'F'?
 *** MNOTE *** 1,Possible register incompatibility between L and 'FR4'?
+       TypeCheck_L  FR4,F

 A        DS    F
 C        DS    CL3
 D        DS    D
 F        DS    S
```

These type-checking examples are incomplete, and are intended more as a detailed sketch than a completed macro package. Feel free to extend and adapt them to suit your needs.



Case Study 8c: Encapsulated Abstract Data Types

To overcome the limitations of using just assembler-assigned types, we will now examine a set of macros that declare and operate on data items with just two specific types: calendar *dates*, and *durations* or *intervals* or *periods* of elapsed time in days. (Because both “date” and “duration” begin with the letter “D”, we'll use “interval” as the preferred term. Other application-specific choices are possible, of course.)

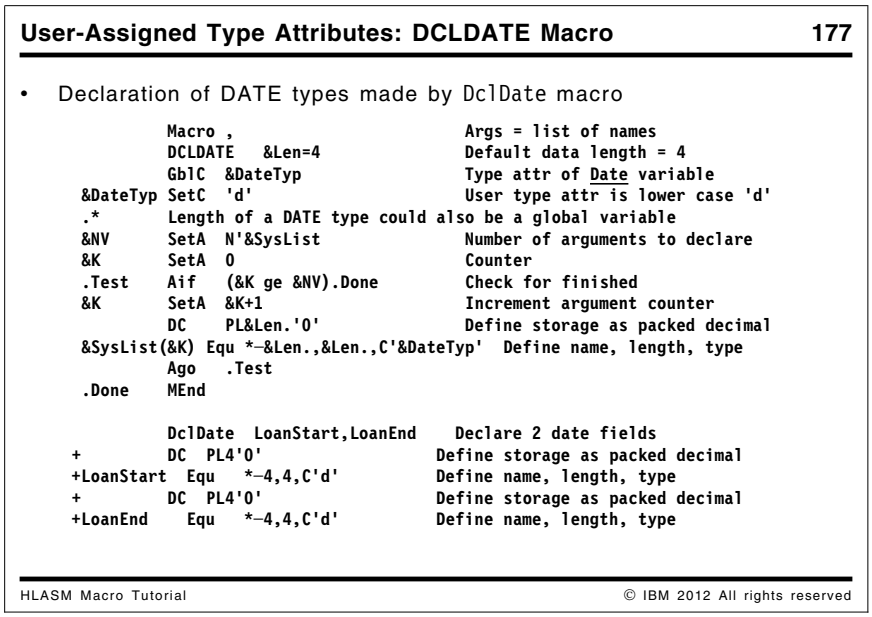
Even though dates and intervals of days can be represented as numbers, some operations on their representations make no “logical” sense — we don't want, for example, to multiply two dates.

With these two data types, we can limit the allowed operations to perform only certain kinds of arithmetic and comparisons:

- two dates may be subtracted to yield an interval
- an interval may be added or subtracted from a date to yield a date
- two intervals may be added or subtracted to yield a new interval
- an interval may be multiplied or divided by an integer constant
- dates may be compared with dates, and intervals with intervals

Any other operation involving dates and intervals is invalid.

First, we examine two macros that declare variables of type “date” and “interval”, (`DCLDATE` and `DCLNTVL`, respectively.) Each macro accepts a list of names to be declared with that type, assigns type attributes `'d'` and `'i'`, and allocates storage for the variables.



First, we illustrate a macro DclDate to declare variables of type "date". The DclDate macro accepts a list of names, and allocates a packed decimal variable of 4 bytes for each, which we assume are represented as Julian dates in the form PL4'yyyyddd'

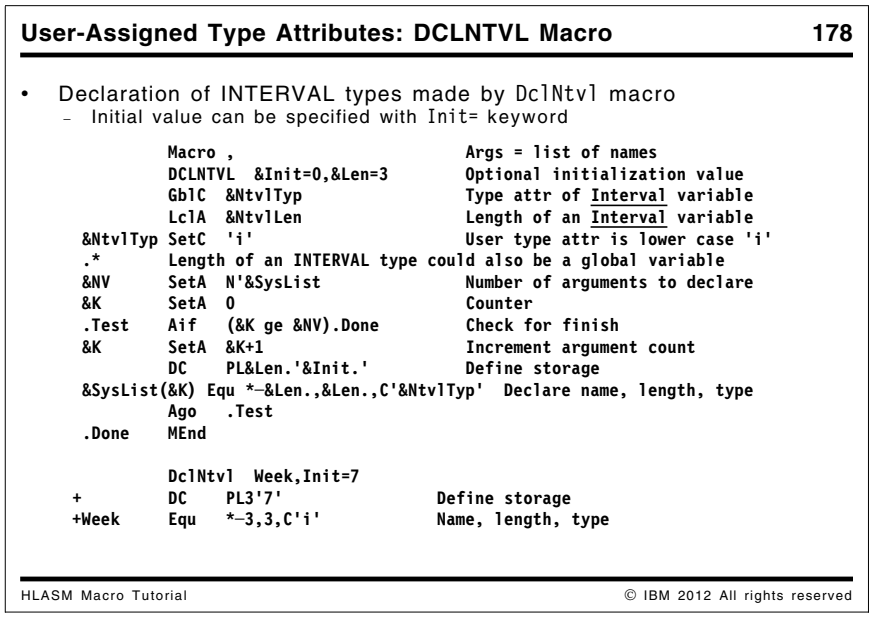
```
         Macro
         DCLDATE   &Len=4               Default data length = 4
         GblC  &DateTyp                 Type attr of Date variable
&DateTyp SetC  'd'                      User type attr is lower case 'd'
.*       Length of a DATE type could also be a global variable
&NV      SetA  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&Len.'0'               Define storage as packed decimal
&SysList(&K) Equ *-&Len.,&Len.,C'&DateTyp'  Define name, length, type
         Ago   .Test
.Done    MEnd
```

Figure 83. Macro to declare "date" data type

Sample calls to the DCLDATE macro are shown in Figure 84 below:

```
         Print   NoGen
         DclDate  Birth,Hire,Degree,Retire,Decease   Declare 5 date fields
         Print   Gen
         DclDate  LoanStart,LoanEnd    Declare 2 date fields
+        DC  PL4'0'                    Define storage as packed decimal
+LoanStart  Equ   *-4,4,C'd'           Define name, length, type
+        DC  PL4'0'                    Define storage as packed decimal
+LoanEnd    Equ   *-4,4,C'd'           Define name, length, type
```

Figure 84. Examples of declaring variables with "date" data type



The DCLNTVL macro also accepts a list of names, and allocates a packed decimal field of 3 bytes for each, which we assume represents an interval of up to 99999 days in the form PL3'ddddd'. In addition, a keyword variable &Init can be used to supply an initial value for all variables declared on a macro call.

```
         Macro
         DCLNTVL  &Init=0,&Len=3        Optional initialization value
         GblC  &NtvlTyp              Type attr of Interval variable
         LclA  &NtvlLen              Storage length of interval variable
&NtvlTyp SetC  'i'                   User type attr is lower case 'i'
.*       Length of an INTERVAL type could also be a global variable
&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&Len.'&Init'        Declare variable and initial value
&SysList(&K) Equ *-&Len.,&Len.,C'&NtvlTyp'  Declare name, length, type
         Ago   .Test                 Check for more arguments
.Done    MEnd
```

Figure 85. Macro to declare “interval” data type

Sample calls to the DCLNTVL macro are illustrated in Figure 86 on page 194:

```
 Aaa      DclNtvl  Vacation,Holidays
+         DC    PL3'0'                Define storage
+Vacation Equ   *-3,3,C'i'            Name, length, type
+         DC    PL3'0'                Define storage
+Holidays Equ   *-3,3,C'i'            Name, length, type

          DclNtvl  LoanTime
+         DC    PL3'0'                Define storage
+LoanTime Equ   *-3,3,C'i'            Name, length, type

          DclNtvl  Year,Init=365
+         DC    PL3'365'              Define storage
+Year     Equ   *-3,3,C'i'            Name, length, type

          DclNtvl  LeapYear,Init=366
+         DC    PL3'366'              Define storage
+LeapYear Equ   *-3,3,C'i'            Name, length, type

          DclNtvl  Week,Init=7
+         DC    PL3'7'                Define storage
+Week     Equ   *-3,3,C'i'            Name, length, type
```

Figure 86. Examples of declaring variables with “interval” data type

Calculating With Date Variables: CalcDat Macro **179**

- Now, define operations on DATEs and INTERVALs
- Callable CalcDat macro calculates dates without user having to know actual data representations

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

- Allowed forms are:

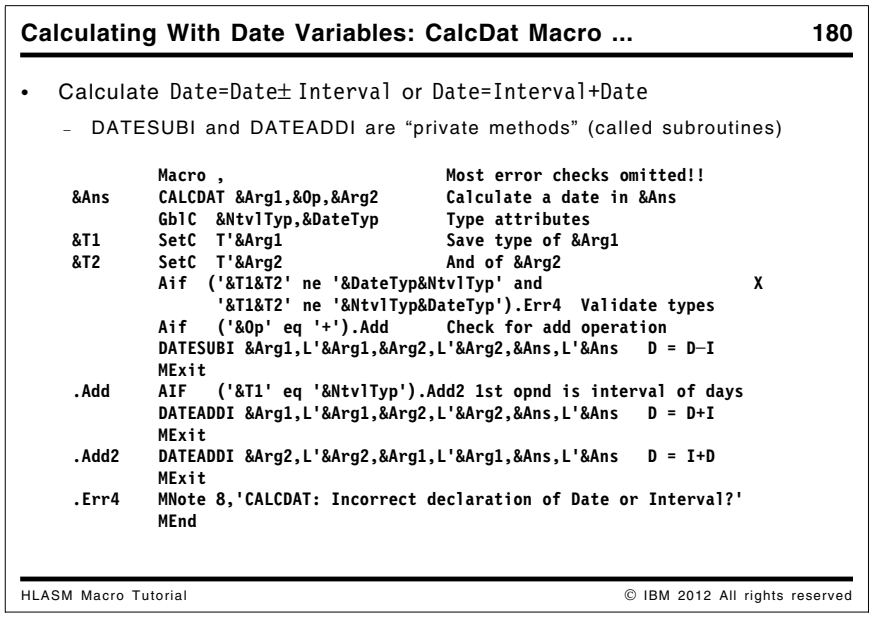
```
ResultDate    CalcDat  Date,+,Interval        Date = Date + Interval
ResultDate    CalcDat  Date,-,Interval        Date = Date - Interval
ResultDate    CalcDat  Interval,+,Date        Date = Interval + Date
```

- CalcDat validates arguments, calls auxiliary macros

```
       DATEADDI  Date1,LDat,Interval,LNvl,AnsDate,AnsLen  Date = Date+Interval
       DATESUBI  Date1,LDat,Interval,LNvl,AnsDate,AnsLen  Date = Date-Interval
```

  - Auxiliary service macros (“private methods”) understand actual data representations (“encapsulation”)

HLASM Macro Tutorial



Calculating with Date Variables

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:

```
ResultDate    CalcDat  Date,+,Interval       Date = Date + Interval
ResultDate    CalcDat  Date,-,Interval       Date = Date - Interval
ResultDate    CalcDat  Interval,+,Date       Date = Interval + Date
```

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 DATEADDI (to add an interval to a date) and DATESUBI (to subtract an interval from a date). These service macros will “understand” the actual representation of “date” and “interval” 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  &NtvlTyp,&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&NtvlTyp' and                    X
              '&T1&T2' ne '&NtvlTyp&DateTyp').Err4  Validate types
        Aif   ('&Op' eq '+').Add      Check for add operation
        Aif   ('&T1&T2' ne '&DateTyp&Ntvltyp').Err5 Bad operand seq?
        DATESUBI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans   D = D-I
        MExit
.Add    AIF   ('&T1' eq '&NtvlTyp').Add2  1st opnd an interval of days
        DATEADDI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans   D = D+I
        MExit
.Add2   DATEADDI &Arg2,L'&Arg2,&Arg1,L'&Arg1,&Ans,L'&Ans   D = I+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 variable
        MNote 8,'&M.Target variable ''&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/Interval arguments'
        MExit
.Err5   MNote 8,'&M.Subtraction operands in reversed order'
        MEnd
```

Figure 87. Macro to calculate “date” results

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

```
 Hire     CalcDat  Degree,+,Year
+         DATEADDI Degree,L'Degree,Year,L'Year,Hire,L'Degree D = D+I

 Hire     CalcDat  Year,+,Degree
+         DATEADDI Degree,L'Degree,Year,L'Year,Hire,L'Degree D = I+D

 Hire     CalcDat  Degree,-,Year
+         DATESUBI Degree,L'Degree,Year,L'Year,Hire,L'Degree D = D-I
```

Figure 88. Examples of macro calls to calculate “date” results

The service macros DATEADDI and DATESUBI do the real work: they must handle whatever representation is chosen for dates (e.g. YYYYDDD for Julian dates, or YYYYMMDD for readable dates), accounting for things like month lengths and leap years. These two macros would most likely invoke a general-purpose service subroutine that handles all such details, rather than generating complex in-line code to handle all possible cases.

Calculating Interval Variables: CalcNvl Macro **181**

- Define user-called CalcNvl macro to calculate intervals
- Allowed forms are:

```
ResultInterval  CalcNvl  Date,–,Date         Difference of two date variables
ResultInterval  CalcNvl  Interval,+,Interval Sum of two interval variables
ResultInterval  CalcNvl  Interval,–,Interval Difference of two intervals
ResultInterval  CalcNvl  Interval,*,Number   Product of interval, number
ResultInterval  CalcNvl  Interval,/,Number   Quotient of interval, number
```

- CalcNvl validates declared types of arguments, and calls one of five auxiliary macros (more “private methods”):

```
DATESUBD  Date1,LDat1,Date2,LDat2,AnsI,AnsLen    Nvl = Date–Date
NTVLADDI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl + Nvl
NTVLSUBI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl – Nvl
NTVLMULI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl * Num
NTVLDIVI  Nvl1,Len1,Nvl2,Len2,AnsI,AnsLen        Nvl = Nvl / Num
```

- Date arithmetic done by called service subroutines

Calculating Interval Variables: CalcNvl Macro ... **182**

```
          Macro ,                    Most error checks omitted!
&Ans      CALCNVL  &Arg1,&Op,&Arg2
          GblC  &NtvlTyp,&DateTyp    Type attributes
&X(C'+')  SetC  'ADD'                Name for ADD routine
&X(C'–')  SetC  'SUB'                Name for SUB routine
&X(C'*')  SetC  'MUL'                Name for MUL routine
&X(C'/')  SetC  'DIV'                Name for DIV routine
&Z        SetC  'C''&Op'''           Convert &Op char to self–def term
&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').II          Second operand nonnumeric
          NTVL&X(&Z).I Arg1,L'&Arg1,=PL3'&Arg2',3,&Ans,L'&Ans I op const
          MExit
.II       NTVL&X(&Z).I &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans I op I
          MExit
.DD       DATESUBD  &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Ans  date–date
          MEnd

 Days     CALCNVL  Days,+,Days        Interval + Interval
+         NTVLADDI     Days,L'Days,Days,L'Days,Days,L'Days     I op I
 Days     CALCNVL  Hire,–,Degree      Date – Date
+         DATESUBD  Hire,L'Hire,Degree,L'Degree,Days,L'Days  date–date
```


Calculating with Interval Variables

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

```
&AnsNtvl   CalcNvl  &Arg1,Op,&Arg2       Calculate an Interval variable
```

where &AnsNtvl must have been declared a “interval” variable, and the allowed operand combinations are:

```
ResultInterval  CalcNvl  Date,-,Date         Difference of two date variables
ResultInterval  CalcNvl  Interval,+,Interval Sum of two interval variables
ResultInterval  CalcNvl  Interval,-,Interval Difference of two intervals
ResultInterval  CalcNvl  Interval,*,Number   Product of interval, number
ResultInterval  CalcNvl  Interval,/,Number   Quotient of interval, number
```

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

```
         Macro ,                    Most error checks omitted!
&Ans     CALCNVL  &Arg1,&Op,&Arg2
         GblC  &NtvlTyp,&DateTyp          Type attributes
&M       SetC  'CALCNVL: '                Macro name for messages
         Aif   (N'&SysList ne 3).Err1     Wrong number of arguments
         Aif   (T'&Ans ne '&NtvlTyp').Err2         Invalid target type
         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'                      Name for SUB routine
&X(C'*') SetC  'MUL'                      Name for MUL routine
&X(C'/') SetC  'DIV'                      Name for DIV routine
&Z       SetC  'C''&Op'''                 Convert &Op to self-def term
.*       &Z used as an index into the &X array
&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 '&NtvlTyp').Err3        Invalid first operand
         Aif   ('&T2' eq '&NtvlTyp' and                                X
               ('&Op' eq '+' or '&Op' eq '-')).II
         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
         NTVL&X(&Z).I Arg1,3,=PL3'&Arg2',3,&Ans,3  interval op const
         MExit
.II      NTVL&X(&Z).I &Arg1,3,&Arg2,3,&Ans,3   interval op interval
         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 variable
         MNote 8,'&M.Target variable omitted'
         MExit
.Err2A   MNote 8,'&M.Target variable ''&Ans'' not declared by DCLNTVL'
         MExit
.Err3    MNote 8,'&M.First argument invalid or not declared by DCLNTVL'
         MExit
.Err4    MNote 8,'&M.Third argument invalid or not declared by DCLNTVL'
         MExit
.Err5    MNote 8,'&M.Invalid (or missing) operator ''&Op'''
         MEnd
```

Figure 89. Macro to calculate “interval” results

This macro also provides a form of encapsulation: the operators (or “methods”) are hidden internally, and are not expected to be visible to the programmer. Thus, the macro names NTVLADDI, NTVLSUBI, NTVLMULI, NTVLDIVI, and DATESUBD perform the actual operations, and need not be visible directly to the user of the CALCNVL macro.

The calls to the “private” NTVLxxxI macros are generated with a form of associative indirect addressing by using the single-character operator (such as + or −) as an index into a four-entry array of strings specifying which macro name will be generated.

Examples of calls to CALCNVL are shown in the following figure:

```
 Days     CALCNVL  Days,+,Days        Interval + Interval
+         NTVLADDI     Days,L'Days,Days,L'Days,Days,L'Days     I op I

 Days     CALCNVL  Hire,-,Degree      Date - Date
+         DATESUBD  Hire,L'Hire,Degree,L'Degree,Days,L'Days  date-date

 Days     CALCNVL  Hire,-,Hire        Date - Date
+         DATESUBD  Hire,L'Hire,Hire,L'Hire,Days,L'Days      date-date

 Days     CALCNVL  Days,-,Days        Interval - Interval
+         NTVLSUBI     Days,L'Days,Days,L'Days,Days,L'Days     I op I

 Days     CALCNVL  Days,+,10          Interval + Number
+         NTVLADDI     Arg1,L'Days,=PL3'10',3,Days,L'Days     I op const

 Days     CALCNVL  Days,-,10          Interval - Number
+         NTVLSUBI     Arg1,L'Days,=PL3'10',3,Days,L'Days     I op const

 Days     CALCNVL  Days,*,10          Interval * Number
+         NTVLMULI     Arg1,L'Days,=PL3'10',3,Days,L'Days     I op const

 Days     CALCNVL  Days,/,10          Interval / Number
+         NTVLDIVI     Arg1,L'Days,=PL3'10',3,Days,L'Days     I op const
```

Figure 90. Examples of macro calls to calculate “interval” results

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 or operator-operand conflicts would not only have been larger, but might have gone undetected. 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.

These macros can be extended to avoid using the Assembler's (rather limited) type-attribute mechanism by using Program Attributes (see “Case Study 9: Using Program Attributes” on page 203) or by maintaining global data structures containing information such as a programmer-declared type, length, and so forth.

Example of an Interval-Calculation Macro 183

- Macro NTVLADDI adds intervals to intervals

```
         Macro
 &L      NTVLADDI  &Arg1,&L1,&Arg2,&L2,&Ans,&LAns
         AIf   ('&Arg1' ne '&Ans').T1    Check for Ans being Arg1
         AIf   (&L1 ne &LAns).Error      Same field, different lengths
 &L      AP    &Ans.(&Lans),&Arg2.(&L2)  Add Arg2 to Answer
         MExit
 .T1     AIf   ('&Arg2' ne '&Ans').T2    Check for Ans being Arg2
         AIf   (&L2 ne &LAns).Error      Same field, different lengths
 &L      AP    &Ans.(&Lans),&Arg1.(&L1)  Add Arg1 to Answer
         MEXit
 .T2     ANop  ,
 &L      ZAP   &Ans.(&Lans),&Arg1.(&L1)  Move Arg1 to Answer
         AP    &Ans.(&Lans),&Arg2.(&L2)  Add Arg2 to Arg1
         MExit
 .Error  MNote 8,'NTVLADDI: Target variable ''&Ans'' has same name as,*
               but different length than, a source operand'
         MEnd

  A      NTVLADDI  X,3,=P'5',1,X,3
 +A      AP    X(3),=P'5'(1)             Add Arg2 to Answer
```

HLASM Macro Tutorial

The "service" macros for handling intervals will probably be simpler than those for dates (except for DATESUBD, which subtracts two dates to yield an interval, and must also account for the choice of date representation, leap years, and the like). As an example of an interval-handling macro, consider an implementation of NTVLADDI shown below.

```
         Macro
&L       NTVLADDI  &Arg1,&L1,&Arg2,&L2,&Ans,&LAns
         AIf   ('&Arg1' ne '&Ans').T1    Check for Ans being Arg1
         AIf   (&L1 ne &LAns).Error      Same field, different lengths
&L       AP    &Ans.(&LAns),&Arg2.(&L2)  Add Arg2 to Answer
         MExit
.T1      AIf   ('&Arg2' ne '&Ans').T2    Check for Ans being Arg2
         AIf   (&L2 ne &LAns).Error      Same field, different lengths
&L       AP    &Ans.(&LAns),&Arg1.(&L1)  Add Arg1 to Answer
         MEXit
.T2      ANop  ,
&L       ZAP   &Ans.(&LAns),&Arg1.(&L1)  Move Arg1 to Answer
         AP    &Ans.(&LAns),&Arg2.(&L2)  Add Arg2 to Arg1
         MExit
.Error   MNote 8,'NTVLADDI: Target variable ''&Ans'' has same name as,*
               but different length than, a source operand'
         MEnd
```

Figure 91. Macro to add an interval to an interval

The macro checks first to see if the "answer" or "target" operand &Ans is the same as one of the "source" operands &Arg1 and &Arg2. If one of them matches, the macro then checks to ensure that the lengths specified are the same, and issues an error message if not. If neither source operand matches the target, then the first operand is copied to the target field, and the second operand is then added to it.

Examples of code generated by the macro are shown in the following figure:

```
 A        NTVLADDI  X,3,=P'5',1,X,3
+A        AP    X(3),=P'5'(1)             Add Arg2 to Answer

 B        NTVLADDI  X,3,Year,2,Y,3
+B        ZAP   Y(3),X(3)                 Move Arg1 to Answer
+         AP    Y(3),Year(2)              Add Arg2 to Arg1

 C        NTVLADDI  X,3,Year,2,X,4
 *** MNOTE *** 8,NTVLADDI: Target variable 'X' has same name as, but
                           different length than, a source operand

 X        DS    PL3
 Y        DS    PL3
 Year     DC    P'365'
```

The NTVLADDI (and related) macros could be generalized to allow length attribute references to be used for length operands, by inserting some additional SETA statements before the AIF tests of the lengths. This is left as an exercise for the reader.

Comparison Operators for Dates and Intervals 184

- Define comparison macros CompDate and CompNtvl

```
&Label  CompDate  &Date1,&Op,&Date2,&True  Compare two dates
&Label  CompNtvl  &Ntvl1,&Op,&Ntvl2,&True  Compare two intervals
```

  - &Op is any useful comparison operator (EQ, NEQ, GT, LE, etc.)
  - &True is the branch target for true compares

```
         Macro
&Label   CompDate  &Date1,&Op,&Date2,&True
         GblA  &DateLen          Length of Date variables
&Mask(1) SetA  8,7,2,13,4,11,10,5,12,3  BC Masks
&T       SetC  ' EQ  NEQ GT  NGT LT  NLT GE  NGE LE  NLE ' Operators
&C       SetC  Upper('&Op')      Convert to Upper Case
&N       SetA  Index('&T','&C')  Find operator
         AIf   (&N eq 0).BadOp
&N       SetA  (&N+3)/4          Calculate mask index
&Label   CP    &Date1.(&DateLen),&Date2.(&DateLen)
         BC    &Mask(&N),&True   Branch to 'True Target'
         MExit
.BadOp   MNote 8,'&SysMac: Bad Comparison Operator ''&Op.'''
         MEnd
```

HLASM Macro Tutorial

Comparison Operators for Dates and Intervals

One further set of functions is needed to complete this set of macros, the comparison operators. Suppose we define two macros CompDate and CompNtvl:

```
&Label  CompDate  &Date1,&Op,&Date2,&True  Compare two dates
&Label  CompNtvl  &Ntvl1,&Op,&Ntvl2,&True  Compare two intervals
```

where the allowed operators could include mnemonic terms such as EQ, NE, GT, NGT, LT, NLT, GE, NGE, LE, NLE, or "graphics" such as =, <, <=, >, >=, <>, and the like. The fourth operand &True is the name of an instruction to which control should branch if the comparison relation is true. The CompDate macro could be written as follows:

```
         Macro
&Label   CompDate  &Date1,&Op,&Date2,&True
         GblA  &DateLen          Length of Date variables
&Mask(1) SetA  8,7,2,13,4,11,10,5,12,3  BC Masks
&T       SetC  ' EQ  NEQ GT  NGT LT  NLT GE  NGE LE  NLE ' Operators
&C       SetC  Upper('&Op')      Convert to Upper Case
&N       SetA  Index('&T','&C')  Find operator
         AIf   (&N eq 0).BadOp
&N       SetA  (&N+3)/4          Calculate mask index
&Label   CP    &Date1.(&DateLen),&Date2.(&DateLen)
         BC    &Mask(&N),&True   Branch to 'True Target'
         MExit
.BadOp   MNote 8,'&SysMac: Bad Comparison Operator ''&Op.'''
         MEnd
```

Figure 92. Comparison macro for “date” data types

The only unusual consideration in this macro is the ordering of the allowed operators in the character variable &T: EQ must appear before NEQ (and similarly for the other combinations) so that if the specified operator is EQ, the INDEX function does not match the EQ in NEQ before finding the correct match at EQ.

Instructions generated by the macro are shown in the following figure:

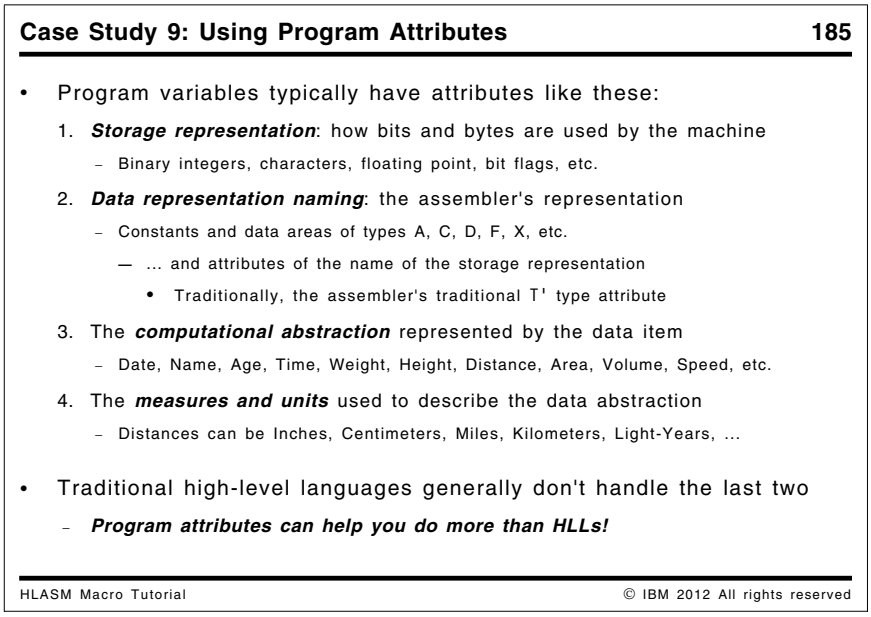
```
 XXX      Compdate A,eq,B,ABEqual
+XXX      CP    A(4),B(4)
+         BC    8,ABEqual         Branch to 'True Target'

 YY       Compdate A,ne,B,ABNeq
+YY       CP    A(4),B(4)
+         BC    7,ABneq           Branch to 'True Target'

 A        DS    PL4
 B        DS    PL4
```

Case Study 9: Using Program Attributes

We have seen in previous case studies how the assembler's "native" type attribute can help you tailor generated code to specific conditions. The type attribute, however, is limited to a single 8-bit byte, and some of its values are reserved by the assembler.

Program attributes provide a way to assign attributes to symbols that are completely independent of the type attribute, and which can take arbitrary freely-chosen 32-bit values.

They are assigned to symbols as an operand of an EQU statement, or as a modifier in a DC or DS statement. You can assign any meaningful 32-bit or 4-byte value to a program attribute: numerics, bit patterns, characters, etc., whatever is most useful for your needs.



Program Attributes

Generally, program variables have attributes like these:

1. Storage *representation*: how bits and bytes are used by instructions executed by the machine. These may be things like:

   - Numeric data, such as binary integers, floating point (hex, binary), packed, zoned, etc.
   - Character: EBCDIC, ASCII, Unicode, etc.
   - Other: pointers, flags, etc.

   Each of these data items will also have properties such as storage length, precision, etc.

   No attributes are attached by the machine to such data — for example, you can do integer arithmetic with characters, floating point data, etc. — "it's just bits" to the hardware.

2. Data representation naming: how you describe the storage representation; that is, what you believe the bits represent. In the assembler, you might specify constants with types like A, C, D, F, X, etc., so that the assembler converts your data to the desired storage representation. The assembler also assigns attributes such as type, length, scale, and integer to the symbolic name of the storage representation.

   - The assembler's traditional T' type attribute can be used to retrieve one such attribute; it is used widely in existing macros, as we've seen.

3. The computational abstraction represented by the data *item*: the meaning you attach to the data. These may be things like Date, Name, Age, Time, Weight, Height, Distance, Area, Volume, Speed, etc.

4. Data typically has *units* (e.g. English or Metric) and may even be "purely numeric", such as dates and mathematical constants like pi, SQRT(2), ...

   - Dates can have many different formats, and may come from many different cultural calendars.

5. Finally, Data may have *measures* (e.g. Feet or Meters) appropriate to the units chosen to describe the data abstraction:

   - Distances and lengths can be Inches, Centimeters, Miles, Kilometers, Light-Years, ...
   - Weights can be Pounds, Ounces, Tons, Grams, Kilograms, ...
   - Times can be microseconds, seconds, minutes, days, weeks, years, ...
   - Combinations of units: Area (distance×distance), speed (distance/time), momentum (weight×speed), ...

Few high-level languages can handle the last two: the computational abstraction and the units of measure. It is not unusual for programs to perform computations on variables whose computational abstractions or units of measure are incompatible. (What is 2×today?)

We will illustrate how the assembler can help you handle these, using program attributes. Our examples will be variations on a theme, exploiting increasingly detailed checking of operations and operands, generating code tailored to the each abstraction and its units of measure.

We will show some examples that assign character values to program attributes, primarily for easier readability. Other encodings could be used if more (or other) information is needed to specify the properties of each symbol.

Suppose we have a macro that assigns one variable to another:

```
EVAL   var1,=,var2        Assigns var2 to var1
```

with the idea that it could be extended some day to handle other types of evaluations and expressions, such as

```
EVAL   var1,=,expression  Evaluates expression, assigns result to var1
```

We will show how you can use concepts like representations, items, units, and measures with five variations on this "EVAL" theme:

1. A simple `EVAL1` macro uses the `T'` type attribute, with only a single data representation.

2. The `EVAL2` macro shows another use of `T'`, with two data representations and lengths (integer and floating point) and some conversions among them.

3. The `DCL3` macro declares variables for two items (Weight, Distance) having either of two representations. It uses two one-byte fields in the program attribute. The `EVAL3` macro uses the declared variables and supports data type conversion.

4. The `EVAL4` macro uses a single storage representation for the same two items (Weight, Distance) now having two units (English, Metric), with generated instructions for unit conversion.

5. The `DCL5` and `EVAL5` macros support two storage representations, two items, two units, and measures appropriate to the units (Kilometers, Miles; Kilograms, Pounds, etc.)

We start with two examples using the `T'` assembler type attribute.

Program Attributes, Variation 1: Simple Assignment **186**

- Suppose macros EVAL1-EVAL5 assign one variable to another:

```
         EVALn  var1,=,var2       Assigns var2 to var1
```

- A simple form (EVAL1) with no checking; types F, H, E, D are OK

```
         Macro ,             Some checking omitted
 &L      EVAL1 &T,&Op,&S     Target variable, operator, source
         LclC  &V            Operation variant, initially null
 &TS     SetC  T'&S          Type of source variable
 &TT     SetC  T'&T          Type of target variable
         AIF   ('&TS' eq '&TT').OKType
         MNote 8,'&SysMac. — Incompatible argument types'
         MExit
 .OKType AIF   ('&TS' eq 'F').NoVar
 &V      SetC  '&TS'         Type variant
 .NoVar  ANop  ,
 &L      L&V   0,&S
         ST&V  0,&T
         MEnd
```

- Traditional T' type attribute selects instructions

HLASM Macro Tutorial

Program Attributes, Variation 1: Simple Assignment

This example uses the traditional assembler type attribute of the data representation to select generated instructions. In this simplest case, the type attributes of the source and target operands must be the same.

```
         Macro
&L       EVAL1 &T,&Op,&S     Target variable , operator, source variable
         LclC  &V            Operation variant, initially null
         AIF   (N'&SysList eq 3).OKNarg
         MNote 8,'&SysMac. -- Invalid argument list'
         MExit
.OKNarg  AIF   ('&Op' eq '=').OKOp
         MNote 8,'&SysMac. -- ''&Op'' unsupported operation'
         MExit
.OKOp    ANop  ,
&TS      SetC  T'&S          Type of source variable
&TT      SetC  T'&T          Type of target variable
         AIF   ('&TS' eq '&TT').OKType
         MNote 8,'&SysMac. -- Incompatible argument types'
         MExit
.OKType  AIF   ('&TS' eq 'F').NoVar
&V       SetC  '&TS'         Type variant
.NoVar   ANop  ,
&L       L&V   0,&S
         ST&V  0,&T
         MEnd
```

Figure 93. Simple EVAL1 macro using type attribute

Typical instruction sequences generated by this EVAL1 macro are:

```
000000 000014A0              33 A1      DC    F'5280'
000004                       34 B1      DS    F
                             35         EVAL1 B1,=,A1
000008 5800 F000             36+        L     0,A1
00000C 5000 F004             37+        ST    0,B1

000010 441C7E0000000000      39 X1      DC    D'7294'
000018                       40 Y1      DS    D
                             41         EVAL1 Y1,=,X1
000020 6800 F010             42+        LD    0,X1
000024 6000 F018             43+        STD   0,Y1

000028 07CF                  45 K1      DC    H'1999'
00002A                       46 L1      DS    H
                             47         EVAL1 L1,=,K1
00002C 4800 F028             48+        LH    0,K1
000030 4000 F02A             49+        STH   0,L1
```

Figure 94. Simple EVAL1 macro code beneration

The above figure shows typical type-sensitive generation of load and store instructions appropriate to the type of data. Examples of error checking by the EVAL1 macro are:

```
                             51         EVAL1 L1,+,K1
** ASMA254I *** MNOTE ***    52+     8,EVAL1 -- '+' unsupported operation

                             54         EVAL1 L1=K1
** ASMA017W Undefined keyword parameter; default to positional,
           including keyword - EVAL1/L1
** ASMA254I *** MNOTE ***    55+     8,EVAL1 -- Invalid argument list

                             57         EVAL1 L1,=,A1
** ASMA254I *** MNOTE ***    58+     8,EVAL1 -- Incompatible argument types
```

Next, we examine a more complex use of the assembler's type attributes; this example will be a model for later use of program attributes.

Program Attributes, Variation 2: Mixed Representations **187**

- Let EVAL2 support mixed integer/floating point operands.
  - Symbol names chosen to show type and **S**ource vs. **T**arget

```
DS    DC   D'0.2'       Doubleword hex float source
DT    DS   D            Doubleword target variable
ES    DC   E'0.3'       Short hex float source
ET    DS   E            Short hex float target variable
FS    DC   F'45678901'  Fullword binary integer source
FT    DS   F            Fullword target variable
HS    DC   H'23456'     Halfword binary integer source
HT    DS   H            Halfword target variable
```

- Allow mixed-type assignments like

```
EVAL2 FT,=,HS     Halfword binary assigned to fullword
EVAL2 DT,=,ES     Short hex float assigned to long
EVAL2 DT,=,FS     Fullword binary assigned to long hex float
EVAL2 ET,=,HS     Halfword binary assigned to short hex float
```

  - The third and fourth assignments require type conversion!
- Some assignments are flagged (e.g. float hex to halfword binary)

HLASM Macro Tutorial

Program Attributes, Variation 2: Mixed Representations

The EVAL2 macro in this example supports four storage representations: halfword and fullword binary, and short and long hexadecimal floating point. It also allows some mixed type assignments between source and target operands.

In the macro definition, the SETC variables &COK, &CCVT, &CWARN, &CSWR, and &CERR contain target-source type pairs that respectively

- can be assigned needing only appropriate load and store instructions
- require conversion from fixed-point binary to floating point
- extend a short floating point operand to long
- convert compatible types with possible truncation and give a warning message
- cannot safely be converted from floating point to binary without a possibly serious loss of data.

The five strings are concatenated into &C, which is then scanned using the Index function to determine how to handle each possible pair of type attributes.

```
        Macro
&L      EVAL2 &T,&Op,&S     Target, operator, source
        LclC  &VT,&VS       Operation variants, target/source
        AIF   (N'&SysList eq 3).OKNarg
        MNote 8,'&SysMac. -- Invalid argument list'
        MExit
.OKNarg AIF   ('&Op' eq '=').OKOp
        MNote 8,'&SysMac. -- ''&Op'' unsupported operation'
        MExit
.OKOp   ANop  ,
&COK    SetC  'DD EE FF HH FH ' No problem
&CCVT   SetC  'DF DH EH '       Conversion
&CSWR   SetC  'DE '             Short/long conversion
&CWarn  SetC  'ED EF HF '       Possible data loss
&CErr   SetC  'FD FE HD HE '    No conversion
&C      SetC  '&COK.&CSWR.&CCVT.&CWarn.&Cerr'
&TS     SetC  T'&S          Type of source variable
&TT     SetC  T'&T          Type of target variable
        AIF   ('&L' eq '').NoLab
&L      DC    0H
.NoLab  ANop  ,
```

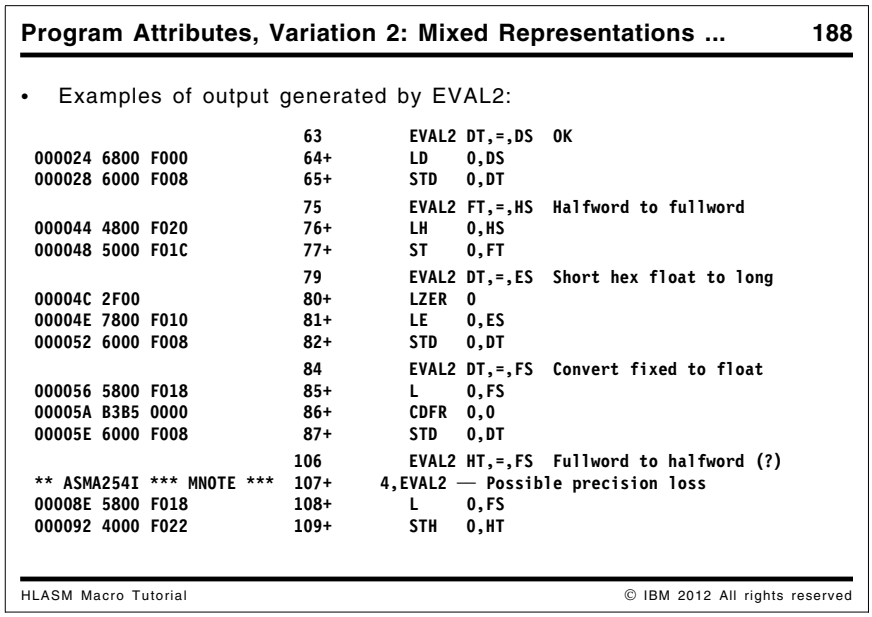
Figure 95 (Part 1 of 2). EVAL2 macro with some type conversions

```
&N      SetA  (Index('&C','&TT.&TS')+2)/3
        AIF   (&N gt 0).Valid1  Known combination
        MNote 8,'&SysMac. -- Unsupported type mix &TT&TS'
        MExit
.Valid1 AIF   (&N lt 13).Valid2
        MNote 8,'&SysMac. -- Types not convertible'
        MExit
.Valid2 AIF   (&N lt 10).Valid3
        MNote 4,'&SysMac. -- Possible precision loss'
.Valid3 AIF   ('&TS' eq 'F').V2A
&VS     SetC  '&TS'
.V2A    AIF   ('&TT' eq 'F').V2B
&VT     SetC  '&TT'
.V2B    AIF   (&N ne 6).DoLdSt   Clear FPR for short to long?
        LZER  0
.DoLdSt L&VS  0,&S
        AIF   (&N eq 11).DoCDFR
        AIF   (&N lt 7 or &N gt 9).DoSt
.DoCDFR CDFR  0,0
.DoSt   ST&VT 0,&T
        MEnd
```

Figure 95 (Part 2 of 2). EVAL2 macro with some type conversions

The following figure shows examples of defining the four data types supported by the EVAL2 macro. The first letter of the name is the Assembler type, and the second letter indicates that the field is a **S**ource or a **T**arget.

```
000000 4033333333333333      54 DS      DC    D'0.2'
000008                       55 DT      DS    D

000010 404CCCCD              56 ES      DC    E'0.3'
000014                       57 ET      DS    E

000018 02B90135              58 FS      DC    F'45678901'
00001C                       59 FT      DS    F

000020 5BA0                  60 HS      DC    H'23456'
000022                       61 HT      DS    H
```



Next, we show some examples of code generated by the EVAL2 macro:

```
                          63         EVAL2 DT,=,DS  OK
000024 6800 F000          64+        LD    0,DS
000028 6000 F008          65+        STD   0,DT

                          75         EVAL2 FT,=,HS  Halfword to fullword
000044 4800 F020          76+        LH    0,HS
000048 5000 F01C          77+        ST    0,FT

                          79         EVAL2 DT,=,ES  Short float to long
00004C 2F00               80+        LZER  0
00004E 7800 F010          81+        LE    0,ES
000052 6000 F008          82+        STD   0,DT

                          84         EVAL2 DT,=,FS  Convert fixed to float
000056 5800 F018          85+        L     0,FS
00005A B3B5 0000          86+        CDFR  0,0
00005E 6000 F008          87+        STD   0,DT
```

Figure 96. Examples of EVAL2 code generation

The following examples illustrate what happens when a longer data item is assigned to a shorter data item:

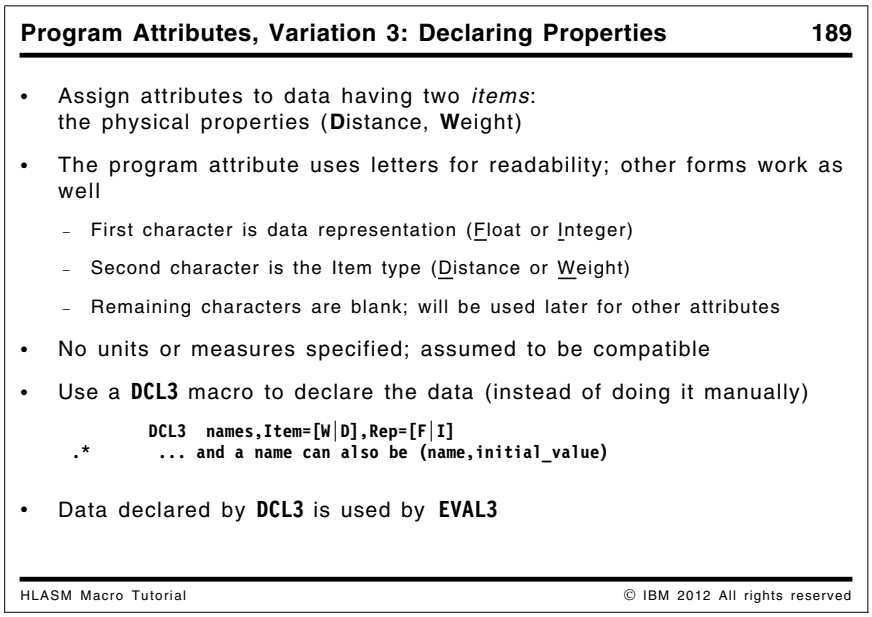
```
                          97         EVAL2 ET,=,DS  Long float to short
** ASMA254I *** MNOTE ***  98+     4,EVAL2 -- Possible precision loss
00007A 6800 F000          99+        LD    0,DS
00007E 7000 F014         100+        STE   0,ET

                         106         EVAL2 HT,=,FS  Fullword to halfword
** ASMA254I *** MNOTE ***  107+     4,EVAL2 -- Possible precision loss
00008E 5800 F018          108+       L     0,FS
000092 4000 F022          109+       STH   0,HT
```

The following figure illustrates some error checking by EVAL2:

```
                          117         EVAL2 HT,=,ES  Short float to halfword
** ASMA254I *** MNOTE ***  118+     8,EVAL2 -- Types not convertible

                          120 T       EVAL2 T,=,T    Bad Type
000096                    121+T       DC    0H
** ASMA254I *** MNOTE ***  122+     8,EVAL2 -- Unsupported type mix MM

                          127         EVAL2 T,+,T    Bad operator
** ASMA254I *** MNOTE ***  128+     8,EVAL2 -- '+' unsupported operation
```

An interesting exercise is to generalize this macro in some of these ways:

- Modify the MNote severities to suit your tastes.
- Add a keyword parameter &Reg=0 allowing the user to select the register to be used as a work register (but remember that long floating point operands may require an even-numbered register)
- Modify the handling of error cases to still generate code.
- Support binary or decimal floating point data.



Program Attributes, Variation 3: Declaring Properties

We assign program attributes to data having two *items* that convey the physical properties **D**istance and **W**eight:

- We use letters in the program attribute for readability. Any absolute expression such as bit groups, hexadecimal digits, etc. would work as well; and the needed bits or digits can be extracted from the value of the program attribute using functions like C2B and C2X.
- The program attribute's first character is the data representation (**F**loat or **I**nteger), and the second character is the item type (D or W). The remaining two characters are blanks and will be used for other attributes in later examples.
- No units or measures are assigned, as they are assumed to be compatible.

We now use a **DCL3** macro to declare the data (rather than doing it manually). The macro can declare either an unitialized storage area, or an initialized data item. The syntax is

```
DCL3  names,Item=[W|D],Rep=[F|I]
```

where the “names” operand is a list of variable names. If an initial value is desired, specify (name,value) instead.

First, the macro validates its arguments:

```
        Macro
&L      DCL3  &Item=,&Rep=
        AIF   (N'&SysList gt 0).OKList
        MNote 8,'&SysMac. -- No positional arguments'
        MExit
.OKList AIF   ('&Item' ne '').HavItm
        MNote 8,'&SysMac. -- No item type specified'
        MExit
.HavItm AIF   ('&Rep' ne '').HavRep
        MNote 8,'&SysMac. -- No representation specified'
        MExit
.HavRep ANop  ,
&I      SetA  Find('&Item','WD')
        AIF   (&I gt 0).ItmOK
        MNote 8,'&SysMac. -- Item type not D or W'
        MExit
.ItmOK  ANop  ,
&I      SetA  Find('&Rep','FI')
        AIF   (&I gt 0).RepOK
        MNote 8,'&SysMac. -- Representation not I or F'
        MExit
.RepOK  ANop  ,
.*      - - -   continued
```

Figure 97. DCL3 macro to declare data items (Part 1 of 2)

Then, the macro scans its argument list and generates DS and DC statements for each declared variable:

```
&NI     SetA  N'&SysList     Count of positionals
&I      SetA  1              First positional
.Loop   ANop  ,
&Name   SetC  '&SysList(&I,1)' Get name
        AIF   (N'&SysList(&I) eq 1).GenDS
        AIF   (N'&SysList(&I) eq 2).GenDC
        MNote 8,'&SysMac. -- Excess or missing suboperands, item &I'
        MExit
.GenDS  ANop  ,
&T      SetC  'D'
        AIF   ('&Rep' eq 'F').GenDS1
&T      SetC  'F'
.GenDS1 ANop  ,
&Name   DS    &T.P(C'&Rep.&Item.  ')
        AGO   .Next
.GenDC  ANop  ,
&T      SetC  'D'
        AIF   ('&Rep' eq 'F').GenDC1
&T      SetC  'F'
.GenDC1 ANop  ,
&Name   DC    &T.P(C'&Rep.&Item.  ')'&SysList(&I,2)'
.Next   ANop  ,
&I      SetA  &I+1
        AIF   (&I le &NI).Loop
        MEnd
```

Figure 98. DCL3 macro to declare data items (Part 2 of 2)

As an exercise, you might generalize the DCL3 macro to allow multiple variables to be declared with the same initial value, and to allow default values for one or both keyword variables.

Some examples of declarations using DCL3 are:

```
        DCL3  w,Item=W,Rep=I
        DCL3  x,Item=W,Rep=F
        DCL3  y,Item=D,Rep=I
        DCL3  z,Item=D,Rep=F
        DCL3  a,(b,99.99),Item=D,Rep=F
        DCL3  m,(n,66),p,(q,444),Item=W,Rep=I
```

The statements generated by these declarations are shown in the following figure:

```
                          51          DCL3  w,Item=W,Rep=I
000000                    52+w        DS    FP(C'IW  ')
                          53          DCL3  x,Item=W,Rep=F
000008                    54+x        DS    DP(C'FW  ')
                          55          DCL3  y,Item=D,Rep=I
000010                    56+y        DS    FP(C'ID  ')
                          57          DCL3  z,Item=D,Rep=F
000018                    58+z        DS    DP(C'FD  ')

                          59          DCL3  a,(b,99.99),c,(d,17.77),Item=D,Rep=F
000020                    60+a        DS    DP(C'FD  ')
000028 4263FD70A3D70A3D   61+b        DC    DP(C'FD  ')'99.99'

                          74          DCL3  m,(n,66),p,(q,444),Item=W,Rep=I
000070                    75+m        DS    FP(C'IW  ')
000074 00000042           76+n        DC    FP(C'IW  ')'66'
000078                    77+p        DS    FP(C'IW  ')
00007C 000001BC           78+q        DC    FP(C'IW  ')'444'
```

Figure 99. Examples of DCL3 code generation

Now, we will see how to use these declarations in the **EVAL3** macro.

Program Attributes, Variation 3: Evaluating Conversions **190**

- **EVAL3** macro checks types and representations, generate instructions:

```
                          147 T1      EVAL3 w,=,m   IW ← IW
000090 5800 F070          148+T1      L     0,m
000094 5000 F000          149+        ST    0,w
                          150 T2      EVAL3 x,=,j   FW ← FW
000098 6800 F058          151+T2      LD    0,j
00009C 6000 F008          152+        STD   0,x
                          160 T3      EVAL3 x,=,q   FW ← IW
0000B0 5800 F07C          161+T3      L     0,q
0000C0 B3B5 0000          162+        CDFR  0,0
0000B8 6000 F008          163+        STD   0,x
                          164 T4      EVAL3 w,=,j   IW ← FW
0000BC 6800 F058          165+T4      LD    0,j
0000B4 B3B9 4000          166+        CFDR  0,4,0
** ASMA254I *** MNOTE *** 167+  4,EVAL3 — Possible int←float precision loss
0000C4 5000 F000          168+        ST    0,w
```


Program Attributes, Variation 3: Evaluating Conversions

The EVAL3 macro uses the program attributes created by the DCL3 macro to check types and generate instructions. This first portion of the macro validates the structure of the argument list, and the presence of program attributes.

```
        Macro
&Lab    EVAL3 &T,&Op,&S
        AIf   (N'&SysList eq 3).OKList
        MNote 8,'&SysMac. -- Invalid argument list'
        MExit
.OKList AIf   ('&Op' eq '=').OKOp
        MNote 8,'&SysMac. -- Unsupported ''&Op'' operator'
        MExit
.OKOp   ANop  ,
&PAT    SetC  SysAttrP('&T')      Target symbol attribute
&PAS    SetC  SysAttrP('&S')      Source symbol attribute
        AIf   ('&PAT' ne '' and 'PAS' ne '').OpPat
        MNote 8,'&SysMac. -- An operand attribute not specified'
        MExit
.*      - - -   continued
```

The next segment of the macro checks the data representations and data items in the program attributes for validity, and that the data items of source and target variables are the same:

```
.OpPat  ANop  ,
&N      SetA  Index('FFIIF','&PAT'(1,1).'&PAS'(1,1))
        AIF   (&N ne 0).OKRep
        MNote 8,'&SysMac. -- An operand has unknown representation'
        MExit
.OKRep  ANop  ,
&N      SetA  Index('WWDDW','&PAT'(2,1).'&PAS'(2,1))
        AIF   (&N ne 0).OKItem
        MNote 8,'&SysMac. -- An operand has unknown item'
        MExit
.OKItem AIF   ('&PAT'(2,1) eq '&PAS'(2,1)).OKUMat
        MNote 8,'&SysMac. -- Operand items mismatch'
        MExit
.OKUMat ANop  ,
.*      - - -   continued
```

The final segment of the macro generates the code:

```
&SV     SetC  ''                  Source variant
&TV     SetC  ''                  Target variant
        AIf   ('&PAS'(1,1) eq 'I').OKLdT
&SV     SetC  'D'
.OKLdT  AIf   ('&PAT'(1,1) eq 'I').OKStT
&TV     SetC  'D'
.OKStT  ANop  ,
&Lab    L&SV  0,&S
        AIf   ('&PAT'(1,1) eq '&PAS'(1,1)).DoSt
        AIf   ('&PAT'(1,1) ne 'I').CDFR
        MNote 4,'&SysMac. -- Possible float->int precision loss'
        CFDR  0,4,0
        AGo   .DoSt
.CDFR   CDFR  0,0
.DoSt   ST&TV 0,&T
        MEnd
```

Some typical uses of the EVAL3 macro are shown in the following figure; the data items were declared by the DCL macro, as shown in Figure 99 on page 213:

```
                          147 T1      EVAL3 w,=,m   IW <- IW
000090 5800 F070          148+T1      L     0,m
000094 5000 F000          149+        ST     0,w

                          150 T2      EVAL3 x,=,j   FW <- FW
000098 6800 F058          151+T2      LD    0,j
00009C 6000 F008          152+        STD    0,x

                          160 T5      EVAL3 x,=,q   FW <- IW
0000B0 5800 F07C          161+T5      L     0,q
** ASMA254I *** MNOTE *** 162+     4,EVAL3, -- Possible float->int precision loss
0000B4 B3B9 4000          163+        CFDR  0,4,0
0000B8 6000 F008          164+        STD    0,x

                          165 T6      EVAL3 w,=,j   IW <- FW
0000BC 6800 F058          166+T6      LD    0,j
0000C0 B3B5 0000          167+        CDFR  0,0
0000C4 5000 F000          168+        ST     0,w
```

Program Attributes, Variation 4: Assigning Measures **191**

- Program attributes can assign “application meaning” to data items
  - Weights: Metric kilograms and English pounds:

```
000000 42A5000000000000     44 WtE    DC    DP(C'WE  ')'165'  English: Pounds
000008 424ACCCCCCCCCCD      45 WtM    DC    DP(C'WM  ')'74.8' Metric: Kilograms
000010                      46 NewE   DS    DP(C'WE  ')       English units
000018                      47 NewM   DS    DP(C'WM  ')       Metric units
```

  - Single storage representation: long hex floating point
- Final segment of the EVAL4 macro converts measures, e.g.

```
        – – –
&L      LD    0,&S
        AIF   ('&PAS'(2,1) eq '&PAT'(2,1)).DoSt
        AIF   ('&PAS'(2,1) eq 'E').ESrc  Source units English
.*            Source units metric:
        MD    0,=D'2.20462'       ←– pounds per kilogram
        AGO   .DoSt
.ESrc   DD    0,=D'2.20462'       ←– kilograms per pound
.DoSt   STD   0,&T
        – – –
```

HLASM Macro Tutorial

Program Attributes, Variation 4: Assigning Measures

In this example, we use program attributes to assign properties that have meaning in an application dealing with weights. We assume that the weight values in the application are measured in Metric kilograms and English pounds, and that a single storage representation (hexadecimal floating point) is used.

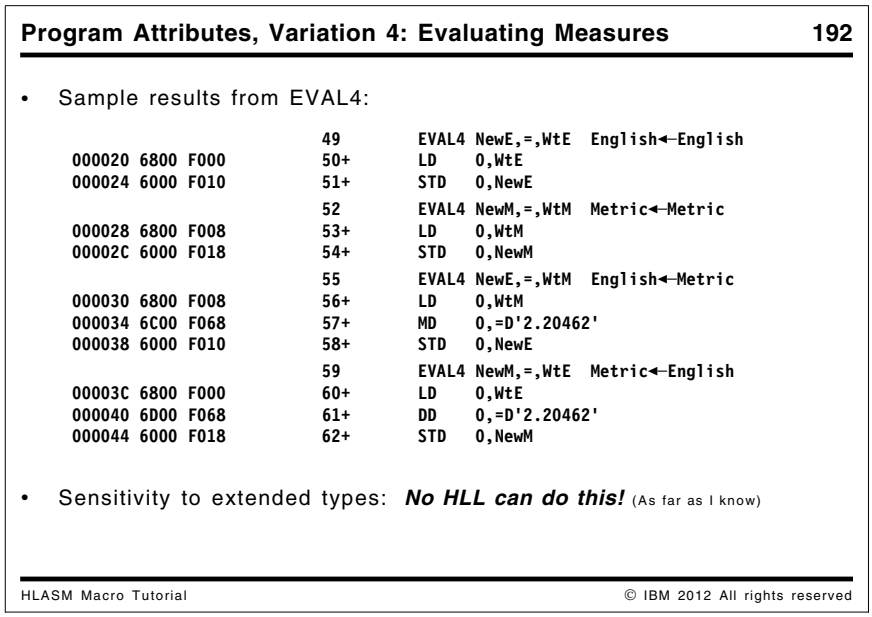
Assuming that the objects represented by program variables are shipped internationally, the EVAL4 macro must convert weight values between kilograms and pounds. To illustrate, we declare four variables representing the two possible values of measures. Because there are just two options (Kg, Lb), the declarations can be done manually rather than by a macro:

```
000000 42A5000000000000      44 WtE     DC    DP(C'FWE ')'165'  Metric: Kilograms
000008 424ACCCCCCCCCCD       45 WtM     DC    DP(C'FWM ')'74.8' English: Pounds
000010                       46 NewE    DS    DP(C'FWE ')       English units
000018                       47 NewM    DS    DP(C'FWM ')       Metric units
```

Most of the code in the EVAL4 macro is devoted to error checks; the last eight statements do the "real work".

```
        Macro
&L      EVAL4 &T,&Op,&S     Target, operator, source
        AIF   (N'&SysList eq 3).OKNarg
        MNote 8,'&SysMac. -- Invalid argument list'
        MExit
.OKNarg AIF   ('&Op' eq '=').OKOp
        MNote 8,'&SysMac. -- ''&Op'' unsupported operation'
        MExit
.OKOp   ANop  ,
&PAS    SetC  SysAttrP('&S')  Source program attribute
&PAT    SetC  SysAttrP('&T')  Target program attribute
        AIF   ('&PAS' ne '' and '&PAT' ne '').OKPats
        MNote 8,'&SysMac. -- Program attribute(s) missing'
        MExit
.OKPats ANop  ,
        AIF   ('&PAS'(1,1) eq '&PAT'(1,1)).UMatch
        MNote 8,'&SysMac. -- Target/source unit mismatch'
        MExit
.UMatch AIF   ('&PAS'(1,1) eq 'W').TMatch
        MNote 8,'&SysMac. -- Item unit(s) not ''W'''
        MExit
.TMatch AIF   ('&PAS'(2,1) eq 'M' or '&PAS'(2,1) eq 'E').OKSM
        MNote 8,'&SysMac. -- Source measure not E or M'
        MExit
.OKSM   AIF   ('&PAT'(2,1) eq 'M' or '&PAT'(2,1) eq 'E').OKTM
        MNote 8,'&SysMac. -- Target measure not E or M'
        MExit
.OKTM   ANop  ,               Source operand
&L      LD    0,&S
        AIF   ('&PAS'(2,1) eq '&PAT'(2,1)).DoSt
        AIF   ('&PAS'(2,1) eq 'E').ESrc  Source units English
.*            Source units metric:
        MD    0,=D'2.20462'
        AGO   .DoSt
.ESrc   DD    0,=D'2.20462'
.DoSt   STD   0,&T
        MEnd
```

Figure 100. EVAL4 macro converts measures



Program Attributes, Variation 4: Evaluating Measures

When variables using different measures are assigned, the EVAL4 macro converts between metric and English measures, as shown in the following listing extract:

```
                         49       EVAL4 NewE,=,WtE  English<-English
000020 6800 F000         50+      LD    0,WtE
000024 6000 F010         51+      STD   0,NewE
                         52       EVAL4 NewM,=,WtM  Metric<-Metric
000028 6800 F008         53+      LD    0,WtM
00002C 6000 F018         54+      STD   0,NewM
                         55       EVAL4 NewE,=,WtM  English<-Metric
000030 6800 F008         56+      LD    0,WtM
000034 6C00 F068         57+      MD    0,=D'2.20462'
000038 6000 F010         58+      STD   0,NewE
                         59       EVAL4 NewM,=,WtE  Metric<-English
00003C 6800 F000         60+      LD    0,WtE
000040 6D00 F068         61+      DD    0,=D'2.20462'
000044 6000 F018         62+      STD   0,NewM
```

Some examples of error checking by the EVAL4 macro are shown in the following figure:

```
                          64 * test error cases
000048                    66 Bad1    DS    D
000050                    67 Bad2    DS    DP(42)
000058                    68 Bad3    DS    DP(C'AA  ')
000060                    69 Bad4    DS    DP(C'WZ  ')

                          72         EVAL4 NewE,=,Bad1
** ASMA254I *** MNOTE ***  73+     8,EVAL4 -- Program attribute(s) missing
                          74         EVAL4 NewE,=,Bad2
** ASMA254I *** MNOTE ***  75+     8,EVAL4 -- Target/source unit mismatch
                          80         EVAL4 Bad3,=,Bad3
** ASMA254I *** MNOTE ***  81+     8,EVAL4 -- Item unit(s) not 'W'
                          84         EVAL4 NewE,=,Bad4
** ASMA254I *** MNOTE ***  85+     8,EVAL4 -- Source measure not E or M
```

This macro illustrates an important benefit of program attributes: you can assign application-specific information to each program variable used in the application, in a way that avoids errors with mismatched variable properties. It also lets you automatically handle any needed conversions among variables. Such sensitivity to extended types is not generally available in most "high level" languages.

Program Attributes, Variation 5: Declaring Units **193**

- First, we use a **DCL5** macro to declare variables with *all* properties
  - Representations: Float, Integer (F, I)
  - Items: Weight, Distance (W, D)
  - Measures: English, Metric (E, M)
  - Units: Kilogram(Kg,K), Pound(P), Mile(Mi,M), Foot(Ft,F), Meter(M), Kilometer(Km,K)
- Note the dual use of M (Mile, Meter) and K (Kilogram, Kilometer)
- Some variables declared as Weights:

```
DCL5  iwmk,Rep=I,Item=W,Unit=M,Meas=kg
DCL5  fwmk,Rep=F,Item=W,Unit=m,meas=k
```

- Some variables declared as Distances:

```
DCL5  idef,Rep=I,Item=D,unit=e,meas=ft
DCL5  idem,Rep=I,Item=D,unit=e,meas=mi
DCL5  fdem,Rep=F,Item=D,unit=e,meas=m
DCL5  idmm,Rep=I,Item=D,unit=m,meas=m
DCL5  fdmk,Rep=F,Item=D,unit=m,meas=k
```

- Choice of letters used for Program Attribute values is arbitrary

HLASM Macro Tutorial

Program Attributes, Variation 5: Declaring Units

First, we use a **DCL5** macro to declare variables having four properties:

- Representation: Float, Integer (indicated by the letters F and I)
- Item: Weight, Distance (letters W and D)
- Measure: English, Metric (letters E and M)
- Unit: Kilogram, Pound (characters Kg, K, and P); Mile, Foot (characters Mi, M, Ft, and F); and Meter, Kilometer (characters M, Km, and K)

Some letters like M and K are used for multiple purposes, and their interpretation depends on the choice of item and unit types.

To simplify declaring variables with this many possible combinations of representation, item, units, and measures, we first provide a DCL5 macro to do the declarations for us. This first segment of the macro checks the arguments to see if all necessary items have been specified. It then checks the specified representation, item type, and unit type.

```
        Macro
&L      DCL5  &Item=,&Rep=,&Unit=,&Meas=
        AIF   (N'&SysList gt 0).OKList
        MNote 8,'&SysMac. -- No positional arguments'
        MExit
.OKList AIF   ('&Item' ne '').HavItm
        MNote 8,'&SysMac. -- No item type specified'
        MExit
.HavItm AIF   ('&Rep' ne '').HavRep
        MNote 8,'&SysMac. -- No representation specified'
        MExit
.HavRep AIF   ('&Unit' ne '').HavUnt
        MNote 8,'&SysMac. -- No unit type specified'
        MExit
.HavUnt AIF   ('&Meas' ne '').HavMes
        MNote 8,'&SysMac. -- No measure specified'
        MExit
.HavMes ANop  ,
&RE     SetC  Upper('&Rep')
&IT     SetC  Upper('&Item')
&UN     SetC  Upper('&Unit')
&ME     SetC  Upper('&Meas')
&I      SetA  Find('&IT','WD')
        AIF   (&I gt 0).ItmOK
        MNote 8,'&SysMac. -- Item type not D or W'
        MExit
.ItmOK  ANop  ,
&I      SetA  Find('&RE','FI')
        AIF   (&I gt 0).RepOK
        MNote 8,'&SysMac. -- Representation not I or F'
        MExit
.RepOK  ANop  ,
&I      SetA  Find('&UN','EM')
        AIF   (&I gt 0).UntOK
        MNote 8,'&SysMac. -- Unit type not E or M'
        MExit
.*      - - -   continued
```

Figure 101. DCL5 macro to declare four program pttributes (Part 1 of 3)

The next segment of the macro checks that the measures are correct for the specified choice of item and unit:

```
.UntOK  AIF   ('&UN' eq 'E').EngUnt
.*      Metric Units
        AIF   ('&IT' eq 'W').MetWt  Check for metric weight
&I      SetA  Find('&ME','KM K M')
        AIF   (&I gt 0).Valid
        MNote 8,'&SysMac. -- Metric D measure not Km, K,or M'
        MExit
.MetWt  ANop  ,             Check metric weight
&I      SetA  Find('&ME','Kg K')
        AIF   (&I gt 0).Valid
        MNote 8,'&SysMac. -- Metric W measure not Kg or K'
        MExit
.EngUnt AIF   ('&IT' eq 'W').EngWt
&I      SetA  Find('&ME','Ft Mi F M')
        AIF   (&I gt 0).Valid
        MNote 8,'&SysMac. -- English D measure not Ft, Mi, F or M'
        MExit
.EngWt  ANop  ,             Check English weight
&I      SetA  Find('&ME','P')
        AIF   (&I gt 0).Valid
        MNote 8,'&SysMac. -- English W measure not P'
        MExit
.Valid  ANop  ,             All keywords validated
&M      SetC  '&ME'(1,1)    Pick off measure letter
&NI     SetA  N'&SysList    Count of positional args
&I      SetA  1             First positional
.*      - - -   continued
```

Figure 102. DCL5 macro to declare four program attributes (Part 2 of 3)

The final macro segment generates the DS and DC statements:

```
.Loop   ANop  ,
&Name   SetC  '&SysList(&I,1)' Get name
        AIF   (N'&SysList(&I) eq 1).GenDS
        AIF   (N'&SysList(&I) eq 2).GenDC
        MNote 8,'&SysMac. -- Excess or missing operands, item &I'
        MExit
.GenDS  ANop  ,
&T      SetC  'D'
        AIF   ('&RE' eq 'F').GenDS1
&T      SetC  'F'
.GenDS1 ANop  ,
&Name   DS    &T.P(C'&RE.&IT.&UN.&M.')
        AGO   .Next
.GenDC  ANop  ,
&T      SetC  'D'
        AIF   ('&RE' eq 'F').GenDC1
&T      SetC  'F'
.GenDC1 ANop  ,
&Name   DC    &T.P(C'&RE.&IT.&UN.&M.')'&SysList(&I,2)'
.Next   ANop  ,
&I      SetA  &I+1
        AIF   (&I le &NI).Loop
        MEnd
```

Figure 103. DCL5 macro to declare four program attributes (Part 3 of 3)

To show some uses of the DCL5 macro, we can declare “Weight” variables with each possible valid value of the four attribute types. The names of the variables are chosen for readability: their names match the assigned letters in the program attribute.

```
         DCL5  iwmk,Rep=I,Item=W,Unit=M,Meas=kg
         DCL5  iwep,Rep=I,Item=W,Unit=E,Meas=p
         DCL5  fwmk,Rep=F,Item=W,Unit=m,meas=k
         DCL5  fwep,Rep=F,Item=W,Unit=e,meas=p
```

The statements generated by these declarations are shown below:

```
                  91         DCL5  iwmk,Rep=I,Item=W,Unit=M,Meas=kg
000000            92+iwmk    DS    FP(C'IWMK')
                  93         DCL5  iwep,Rep=I,Item=W,Unit=E,Meas=p
000004            94+iwep    DS    FP(C'IWEP')
                  95         DCL5  fwmk,Rep=F,Item=W,Unit=m,meas=k
000008            96+fwmk    DS    DP(C'FWMK')
                  97         DCL5  fwep,Rep=F,Item=W,Unit=e,meas=p
000010            98+fwep    DS    DP(C'FWEP')
```

Similarly, we can declare “Distance” variables with each possible valid value of the four attribute types, using the same naming convention as was used for weight variables:

```
         DCL5  idef,Rep=I,Item=D,unit=e,meas=ft
         DCL5  fdef,Rep=F,Item=D,unit=e,meas=f
         DCL5  idem,Rep=I,Item=D,unit=e,meas=mi
         DCL5  fdem,Rep=F,Item=D,unit=e,meas=m
         DCL5  idmm,Rep=I,Item=D,unit=m,meas=m
         DCL5  fdmm,Rep=F,Item=D,unit=m,meas=m
         DCL5  idmk,Rep=I,Item=D,unit=m,meas=km
         DCL5  fdmk,Rep=F,Item=D,unit=m,meas=k
```

The generated statements for these declarations are shown below:

```
                 100         DCL5  idef,Rep=I,Item=D,unit=e,meas=ft
000018           101+idef    DS    FP(C'IDEF')
                 102         DCL5  fdef,Rep=F,Item=D,unit=e,meas=f
000020           103+fdef    DS    DP(C'FDEF')
                 104         DCL5  idem,Rep=I,Item=D,unit=e,meas=mi
000028           105+idem    DS    FP(C'IDEM')
                 106         DCL5  fdem,Rep=F,Item=D,unit=e,meas=m
000030           107+fdem    DS    DP(C'FDEM')
                 108         DCL5  idmm,Rep=I,Item=D,unit=m,meas=m
000038           109+idmm    DS    FP(C'IDMM')
                 110         DCL5  fdmm,Rep=F,Item=D,unit=m,meas=m
000040           111+fdmm    DS    DP(C'FDMM')
                 112         DCL5  idmk,Rep=I,Item=D,unit=m,meas=km
000048           113+idmk    DS    FP(C'IDMK')
                 114         DCL5  fdmk,Rep=F,Item=D,unit=m,meas=k
000050           115+fdmk    DS    DP(C'FDMK')
```

Finally, some examples of invalid declarations illustrate diagnostic checking by the DCL5 macro:

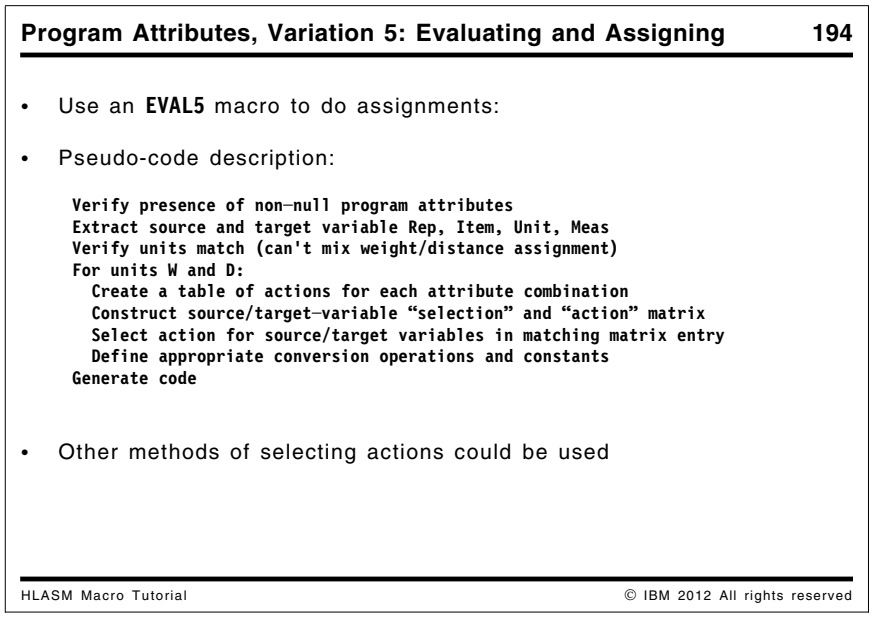
```
         DCL5  z1
         DCL5  z2,Item=X
         DCL5  z2a,Item=X,rep=f,unit=e,meas=f
         DCL5  z2b,Item=w,rep=x,unit=e,meas=f
         DCL5  z2d,Item=d,rep=f,unit=e,meas=x
         DCL5  z3,Rep=X,Item=X,Unit=X
         DCL5  z5b,Item=d,Rep=F,unit=x,meas=k
```

The output from these test cases is shown below:

```
                            139         DCL5  z1
** ASMA254I *** MNOTE ***   140+     8,DCL5 -- No item type specified
                            141         DCL5  z2,Item=X
** ASMA254I *** MNOTE ***   142+     8,DCL5 -- No representation specified
                            143         DCL5  z2a,Item=X,rep=f,unit=e,meas=f
** ASMA254I *** MNOTE ***   144+     8,DCL5 -- Item type not D or W
                            145         DCL5  z2b,Item=w,rep=x,unit=e,meas=f
** ASMA254I *** MNOTE ***   146+     8,DCL5 -- Representation not I or F
                            149         DCL5  z2d,Item=d,rep=f,unit=e,meas=x
** ASMA254I *** MNOTE ***   150+     8,DCL5 -- English D measure not Ft, Mi, F or M
                            157         DCL5  z3,Rep=X,Item=X,Unit=X
** ASMA254I *** MNOTE ***   158+     8,DCL5 -- No measure specified
                            163         DCL5  z5b,Item=d,Rep=F,unit=x,meas=k
** ASMA254I *** MNOTE ***   164+     8,DCL5 -- Unit type not E or M
```

The choice of letters for Program Attribute values is arbitrary, and is intended to make the examples more readable. We could just as well have used bit patterns or other values; for example, since each of the four properties has fewer than four values, we can encode them in two bits, keeping the '00' combination to mean “unknown”. For example, this encoding requires only 8 bits, rather than DCL5's 32:

```
&Rep   SetC  '01'    Floating point
&Item  SetC  '10'    Distance
&Unit  SetC  '01'    English
&Meas  SetC  '10'    Mile
&Patt  SetC  '&Rep.&Item.&Unit.&Meas'
Var    DS    DP(B'&Patt')    Similar to 'FDEM' in DCL5
```



Program Attributes, Variation 5: Evaluating and Assigning

The EVAL5 macro is detailed, but fairly straightforward in its operation, as outlined in this pseudocode:

```
Verify presence of non-null program attributes
Extract source and target variable Rep, Item, Unit, Meas
Verify units match (can't mix weight/distance assignment)
For units W and D:
  Create a table of actions for each attribute combination
  Construct source/target-variable "selection" and "action" matrix
  Select action for source/target variables in matching matrix entry
  Define appropriate conversion operations and constants
Generate code
```

In the entries in each action matrix, the low-order decimal digit is the action code, as follows:

1 assign with no further action
2 convert integer source to floating point and assign with no further action
3 convert integer source to floating point, multiply by the appropriate constant, and assign
4 convert integer source to floating point, divide by the appropriate constant, and assign
5 multiply by the appropriate constant, and assign
6 divide by the appropriate constant, and assign
7 floating point to integer conversions cannot be done without loss of precision; diagnose and generate no code
8 integer to integer conversions that mix units or measures cannot be done without loss of accuracy; diagnose and generate no code

The tens digit of each action code is an index into a table of conversion constants, as follows:

1 English pounds per metric kilogram: 2.20422
2 Metric meters per kilometer: 1000
3 English feet per mile: 5280
4 English feet per metric meter: 3.28083
5 Metric kilometers per English mile: 1.60935
6 Metric meters per English mile: 1609.35
7 English feet per metric kilometer: 3280.83

The row and column headings in the action matrix for weights are the three letters of the program attribute for representation, unit, and measure, in that order. The "W" for "Weight" is omitted because it is common to all items in the matrix. For example, **IMK** means "weights represented as **I**ntegers, in **M**etric units of **K**ilograms".

Figure 104. EVAL5 Macro: Action Matrix for Weight Items

To → From ↓	**IMK**	**FMK**	**IEP**	**FEP**
IMK	1	2	8	13
FMK	7	1	7	15
IEP	8	14	1	2
FEP	7	16	7	1

Similarly, in the action matrix for distances, we omit the second letter "D" in the row and column headings.

Figure 105. EVAL5 macro: Action Matrix for Distance Items

To → From ↓	**IEF**	**FEF**	**IEM**	**FEM**	**IMM**	**FMM**	**IMK**	**FMK**
IEF	1	2	8	34	8	44	8	74
FEF	7	1	7	36	7	46	7	76
IEM	135	33	1	2	8	63	8	53
FEM	7	35	7	1	7	65	7	55
IMM	8	43	8	64	1	2	8	24
FMM	7	45	7	66	7	1	7	26
IMK	8	73	8	54	125	23	1	2
FMK	7	75	7	56	7	25	7	1

For two special cases involving conversion of integer miles to feet and integer kilometers to meters, an additional hundreds unit is used to select an `MHI` instruction to perform integer rather than floating point multiplication.

The first segment of the EVAL5 macro checks for correct format of the argument list, and for the presence of program attributes for the source and target variables.

```
        Macro
&Lab    EVAL5 &T,&Op,&S
        AIf   (N'&SysList eq 3).OKList
        MNote 8,'&SysMac. -- Invalid argument list'
        MExit
.OKList AIf   ('&Op' eq '=').OKOp
        MNote 8,'&SysMac. -- Unsupported ''&Op'' operator'
        MExit
.OKOp   ANop  ,
&PAT    SetC  SysAttrP('C      Target symbol attribute
&PAS    SetC  SysAttrP('&S')      Source symbol attribute
        AIF   ('&PAS' ne '' and '&PAT' ne '').HavPA
.DErr   MNote 8,'&SysMac. -- missing or incorrect attributes --'
        MNote 0,'Source or target not declared by DCL5?'
        MExit
.*      - - -   continued
```

Figure 106. EVAL5 macro (Part 1 of 7)

The next segment extracts four “characters” from the program attribute of the source and target variables, and verifies that the assignment is for matching items (so that the user can't, for example, try to assign a distance to a weight). It then sets the type suffix for the load and store instructions, assuming that the representation values in the program attributes are valid. (Their validity is checked below.)

```
.HavPA  ANop  ,
&SR     SetC  '&PAS'(1,1)         Source representation
&TR     SetC  '&PAT'(1,1)         Target representation
&SI     SetC  '&PAS'(2,1)         Source item
&TI     SetC  '&PAT'(2,1)         Target item
&SU     SetC  '&PAS'(3,1)         Source unit
&TU     SetC  '&PAT'(3,1)         Target unit
&SM     SetC  '&PAS'(4,1)         Source measure
&TM     SetC  '&PAT'(4,1)         Target measure
        AIf   ('&PAT' ne '' and 'PAS' ne '').OpPat
        MNote 8,'&SysMac. -- An operand attribute not specified'
        MExit
.OpPat  AIf   ('&SI' eq '&TI').OKItm
        MNote 8,'&SysMac. -- Cannot mix weight/distance items'
        MExit
.OKItm  Anop  ,
&SV     SetC  ''                  Source variant
&TV     SetC  ''                  Target variant
        AIf   ('&SR' eq 'I').OKLdT
&SV     SetC  'D'                 Float source
.OKLdT  AIf   ('&TR' eq 'I').OKStT
&TV     SetC  'D'                 Float target
.OKStT  ANop  ,
.*      - - -   continued
```

Figure 107. EVAL5 macro (Part 2 of 7)

This segment initializes the list of conversion constants, and (if the source and target items are weights) initializes the selection matrix &W and the action matrix &WA. Then, it searches for the matching entries in the selection matrix, and if found, extracts the action code. It also initializes the conversion constant if it will be needed. If no matching entry in the selection matrix is

found, the program attributes for one or both variables is incorrect, and a message is issued, terminating the macro expansion.

```
&CVT    SetC  'MD'                Assume multiply
&ScTg   SetC  '&SR.&SU.&SM.&TR.&TU.&TM'
&Con(1) SetC  '2.20422','1000','5280','3.28083','1.60935'
.*              Lb/Kg    M/Km   Ft/Mi  Ft/Me     Km/Mi
&Con(6) SetC  '1.60935E3','3.28083E3'
.*             Me/Mi       Ft/Km
        AIf   ('&SI' eq 'D').Dist
.*      Do Weights
&WA(1)  SetA  1,2,8,13,7,1,7,15,8,14,1,2,7,16,7,1   Weight Actions
&W1     SetC  ' IMKIMK IMKFMK IMKIEP IMKFEP'   Row 1
&W2     SetC  ' FMKIMK FMKFMK FMKIEP FMKFEP'   Row 2
&W3     SetC  ' IEPIMK IEPFMK IEPIEP IEPFEP'   Row 3
&W4     SetC  ' FEPIMK FEPFMK FEPIEP FEPFEP'   Row 4
&W      SetC  '&W1.&W2.&W3.&W4'   Weights "matrix"
&N      SetA  Index('&W','&ScTg') Search for match
        AIf   (&N eq 0).DErr      Error if no match
&Act    SetA  &WA((&N/7)+1)       Get action code
        AIf   (&Act lt 10).DoCode No Float required?
&C      SetA  &Act/10             Get constant index
&Const  SetC  '=D''&Con(&C)'''    Construct literal
&Act    SetA  &Act-10*&C          Only the action code
        Ago   .DoCode
.*      - - -   continued
```

Figure 108. EVAL5 macro (Part 3 of 7)

If the two variables are both distance items, this segment of the macro initializes the selection matrix &D.

```
.Dist   ANop  ,
&D1a    SetC  ' IEFIEF IEFFEF IEFIEM IEFFEM'  Row 1
&D1b    SetC   ' IEFIMM IEFFMM IEFIMK IEFFMK'
&D2a    SetC  ' FEFIEF FEFFEF FEFIEM FEFFEM'  Row 2
&D2b    SetC   ' FEFIMM FEFFMM FEFIMK FEFFMK'
&D3a    SetC  ' IEMIEF IEMFEF IEMIEM IEMFEM'  Row 3
&D3b    SetC   ' IEMIMM IEMFMM IEMIMK IEMFMK'
&D4a    SetC  ' FEMIEF FEMFEF FEMIEM FEMFEM'  Row 4
&D4b    SetC   ' FEMIMM FEMFMM FEMIMK FEMFMK'
&D5a    SetC  ' IMMIEF IMMFEF IMMIEM IMMFEM'  Row 5
&D5b    SetC   ' IMMIMM IMMFMM IMMIMK IMMFMK'
&D6a    SetC  ' FMMIEF FMMFEF FMMIEM FMMFEM'  Row 6
&D6b    SetC   ' FMMIMM FMMFMM FMMIMK FMMFMK'
&D7a    SetC  ' IMKIEF IMKFEF IMKIEM IMKFEM'  Row 7
&D7b    SetC   ' IMKIMM IMKFMM IMKIMK IMKFMK'
&D8a    SetC  ' FMKIEF FMKFEF FMKIEM FMKFEM'  Row 8
&D8b    SetC   ' FMKIMM FMKFMM FMKIMK FMKFMK'
&D      SetC  '&D1a.&D1b.&D2a.&D2b.&D3a.&D3b.&D4a.&D4b'
&D      SetC  '&D.&D5a.&D5b.&D6a.&D6b.&D7a.&D7b.&D8a.&D8b'
.*      - - -   continued
```

Figure 109. EVAL5 macro (Part 4 of 7)

This segment initializes the action matrix &DA, whose entries correspond to the entries in the selection matrix &D.

```
.*      Distance Actions
&DA(01) SetA  001,002,008,034,008,044,008,074   Row 1
&DA(09) SetA  007,001,007,036,007,046,007,076   Row 2
&DA(17) SetA  135,033,001,002,008,063,008,053   Row 3
&DA(25) SetA  007,035,007,001,007,065,007,055   Row 4
&DA(33) SetA  008,043,008,064,001,002,008,024   Row 5
&DA(41) SetA  007,045,007,066,007,001,007,026   Row 6
&DA(49) SetA  008,073,008,054,125,023,001,002   Row 7
&DA(57) SetA  007,075,007,056,007,025,007,001   Row 8
.*      - - -   continued
```

Figure 110. EVAL5 macro (Part 5 of 7)

This segment of the macro analyzes the action code, and creates the operand for the conversion instruction if it will be needed.

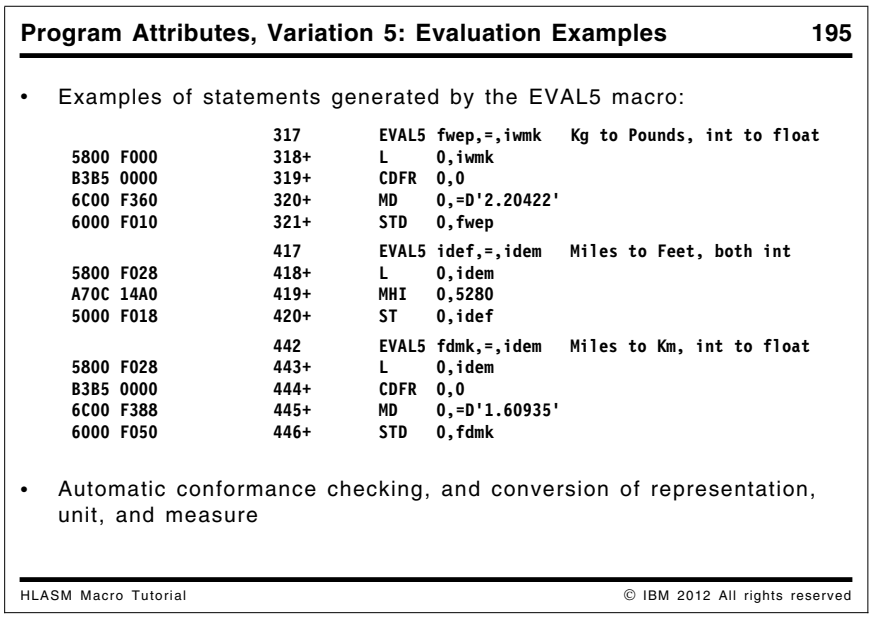
```
&N      SetA  Index('&D','&ScTg')
        AIf   (&N eq 0).DErr
&Act    SetA  &DA((&N/7)+1)
        AIf   (&Act lt 10).DoCode
&C      SetA  &Act/10
        AIf   (&C gt 10).MHI
&Const  SetC  '=D''&Con(&C)'''
        AGo   .DoW1
.MHI    ANop  ,
&CVT    SetC  'MHI'              Integer multiple
&Const  SetC  '&Con(&C-10)'      Integer constant
.DoW1   ANop  ,
&Act    SetA  &Act-10*(&Act/10)  Correct action code (Mod 10)
.*      - - -   continued
```

Figure 111. EVAL5 macro (Part 6 of 7)

The final segment of the macro generates either an appropriate message, or the instructions to perform the assignment, with conversions if needed.

```
.DoCode ANop  ,                  Ready to generate code
        AIf   (&Act le 7).CodeA
        MNote 8,'&SysMac -- cannot safely convert integer unit/measure'
        MExit
.CodeA  AIf   (&Act le 6).CodeB
        MNote 8,'&SysMac. -- cannot convert without precision loss'
        MExit
.CodeB  ANop  ,
&Lab    L&SV  0,&S
        AIf   (&Act eq 1).DoSt
        AIf   (&Act lt 2 or &Act gt 4).CodeC
        CDFR  0,0
        AIF   (&Act eq 2).DoSt  Done
.CodeC  AIf   (&Act eq 3 or &Act eq 5).CodeD
&CVT    SetC  'DD'               Divide
.CodeD  &CVT  0,&Const
.DoSt   ST&TV 0,&T
        MEnd
```

Figure 112. EVAL5 macro (Part 7 of 7)



We now illustrate some examples of EVAL5 calls. Some simply generate code to copy the source variable to the target, while others require conversion of representation, unit, or measures, in any combination. First, some examples of weight assignments:

```
EVAL5 iwmk,=,iwmk    integer kilos ◄— integer kilos
EVAL5 fwmk,=,iwmk      float kilos ◄— integer kilos
EVAL5 iwep,=,iwmk   integer pounds ◄— integer kilos?
EVAL5 fwep,=,iwmk     float pounds ◄— integer kilos

EVAL5 iwep,=,fwmk   integer pounds ◄— float kilos?
EVAL5 fwep,=,fwmk     float pounds ◄— float kilos

EVAL5 fwmk,=,iwep      float kilos ◄— integer pounds
```

The statements generated by these calls are shown in the following figure. Note that the fourth and seventh EVAL5 calls require all three conversions.

```
                    308         EVAL5 iwmk,=,iwmk
5800 F000           309+        L     0,iwmk
5000 F000           310+        ST    0,iwmk

                    311         EVAL5 fwmk,=,iwmk
5800 F000           312+        L     0,iwmk
B3B5 0000           313+        CDFR  0,0
6000 F008           314+        STD   0,fwmk

                    315         EVAL5 iwep,=,iwmk
     *** MNOTE ***  316+     8,EVAL5 -- cannot safely convert integer unit/measure

                    317         EVAL5 fwep,=,iwmk
5800 F000           318+        L     0,iwmk
B3B5 0000           319+        CDFR  0,0
6C00 F360           320+        MD    0,=D'2.20422'
6000 F010           321+        STD   0,fwep

                    328         EVAL5 iwep,=,fwmk
     *** MNOTE ***  329+     8,EVAL5 -- cannot convert without precision loss

                    330         EVAL5 fwep,=,fwmk
6800 F008           331+        LD    0,fwmk
6C00 F360           332+        MD    0,=D'2.20422'
6000 F010           333+        STD   0,fwep

                    337         EVAL5 fwmk,=,iwep
5800 F004           338+        L     0,iwep
B3B5 0000           339+        CDFR  0,0
6D00 F360           340+        DD    0,=D'2.20422'
6000 F008           341+        STD   0,fwmk
```

Next, we see some examples of EVAL5 calls involving distance variables:

```
        EVAL5 fdem,=,idef         float miles ◄── integer feet
        EVAL5 idmm,=,idef      integer meters ◄── integer feet?
        EVAL5 fdmm,=,idef        float meters ◄── integer feet
        EVAL5 fdmk,=,idef    float kilometers ◄── integer feet
        EVAL5 fdef,=,fdef          float feet ◄── float feet
        EVAL5 idef,=,idem        integer feet ◄── integer miles
        EVAL5 fdmk,=,idem    float kilometers ◄── integer miles
```

The statements generated by these calls are shown in the following figure. As with the examples of weight variables above, the fourth and seventh EVAL5 calls require all three conversions.

```
                    373        EVAL5 fdem,=,idef
5800 F018           374+       L     0,idef
B3B5 0000           375+       CDFR  0,0
6D00 F368           376+       DD    0,=D'5280'
6000 F030           377+       STD   0,fdem

                    378        EVAL5 idmm,=,idef
     *** MNOTE ***  379+    8,EVAL5 -- cannot safely convert integer unit/measure

                    380        EVAL5 fdmm,=,idef
5800 F018           381+       L     0,idef
B3B5 0000           382+       CDFR  0,0
6D00 F370           383+       DD    0,=D'3.28083'
6000 F040           384+       STD   0,fdmm

                    387        EVAL5 fdmk,=,idef
5800 F018           388+       L     0,idef
B3B5 0000           389+       CDFR  0,0
6D00 F378           390+       DD    0,=D'3.28083E3'
6000 F050           391+       STD   0,fdmk

                    395        EVAL5 fdef,=,fdef
6800 F020           396+       LD    0,fdef
6000 F020           397+       STD   0,fdef

                    417        EVAL5 idef,=,idem
5800 F028           418+       L     0,idem
A70C 14A0           419+       MHI   0,5280
5000 F018           420+       ST    0,idef

                    442        EVAL5 fdmk,=,idem
5800 F028           443+       L     0,idem
B3B5 0000           444+       CDFR  0,0
6C00 F388           445+       MD    0,=D'1.60935'
6000 F050           446+       STD   0,fdmk
```

Assembler and Program Attribute Summary **196**

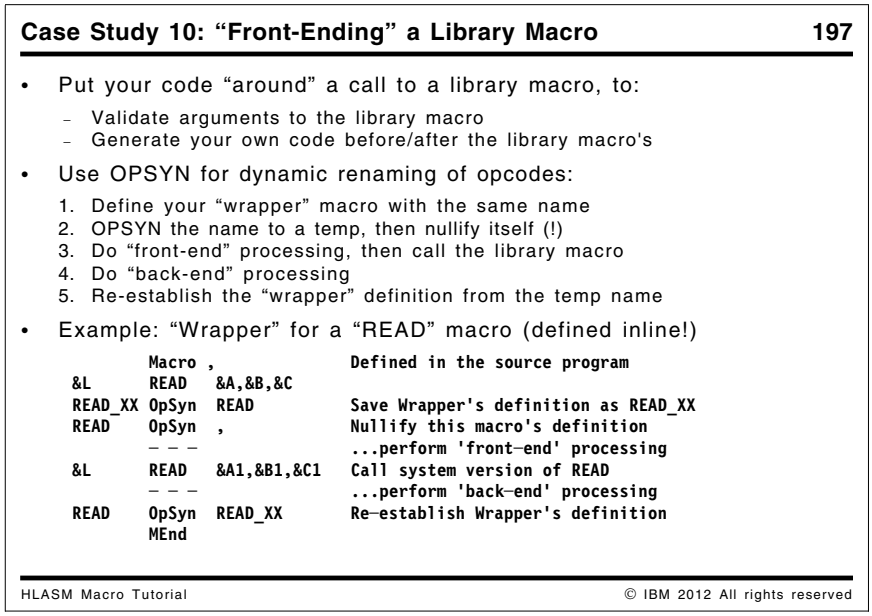
- Assembler attributes help ensure correct register usage
- Program attributes provide tremendous flexibility
  - Detailed descriptions of individual data items
  - Verifiable interactions between instructions and data
  - Enables conversions, diagnostics, etc.
- Adds more power to the macro language
- ***No HLL can do such powerful and useful things as this!***
  (As far as I know)
  - What HLLs call “strong typing” is much weaker

HLASM Macro Tutorial

Assembler and Program Attribute Summary

The examples in this case study have shown only a few possible uses of program attributes. The ability of the Assembler Language to assign a greater range of properties to program variables lets you write macros that can validate and perform operations that are difficult or impossible in most high-level languages. For example, you can check that assigning a value to a "velocity" or "speed" variable has been derived from other velocity variables, or are calculated by dividing a distance variable by a time variable, multiplying an acceleration variable by a time variable, and so on.

The vast number of possibilities provided by assembler and program attributes is nearly unlimited. It's up to you to take advantage of the added power provided by these two attributes; and, you can still retain the high levels of control and efficiency always associated with Assembler Language programs.



Case Study 10: "Front-Ending" a Library Macro

Sometimes it is useful to modify slightly the behavior of a "system" or other established macro. Making changes to the macro itself can lead to maintenance problems if service or updates are provided to the original definition. If your needs can be met by "front-ending" or "wrapping" the original macro definition, it can be called by the "wrapper" macro using the *same* name!

This may seem strange, because the assembler knows of only one definition of each operation code at a given time. The technique used is this:

1. Define a "wrapper" macro with the same name as the original macro.
2. When the "wrapper" is expanded, it uses OPSYN to save its name under a different name, and then nullifies its own definition!
3. The "wrapper" macro does whatever "front-end" processing it likes, and then calls the original macro; if any modifications were made to the original operands, those new operands are used instead. Because the "wrapper" definition has been nullified, the assembler will search the macro library for the intended "official" definition of the original macro; once found, it will be encoded and the call will cause normal macro expansion.
4. When expansion of the original macro is finished, the "wrapper" macro can do any further "back-end" processing needed.

5. Finally, the “wrapper” macro re-establishes its *own* definition, and exits.

To illustrate, suppose we want to “front-end” a READ macro:

```
        Macro ,              Defined in the source program
&L      READ   &A,&B,&C
READ_XX OpSyn  READ          Save Wrapper's definition as READ_XX
READ    OpSyn  ,             Nullify this macro's definition
        - - -                ...perform 'front-end' processing
&L      READ   &A1,&B1,&C1   Call system version of READ
        - - -                ...perform 'back-end' processing
READ    OpSyn  READ_XX       Re-establish Wrapper's definition
        MEnd
```

Figure 113. Example of a macro “wrapper”

The “wrapper” macro cannot be placed in the macro library, because it would then replace the original macro it is intended to “wrap”! Similarly, the wrapping macro cannot be placed in a separate library concatenated before or after the wrapped macro, because the assembler will always find that definition first in the search order, and never the other. If the “wrapper” macro is not part of the source file, it can easily be inserted either via COPY or as part of a PROFILE-option member (with a different name, of course!).

This technique can be useful in setting different default keyword values in a macro. Rather than rewrite the macro (which may belong to some other organization), you can “wrap” the macro to pass modified arguments of your choice.

Summary **198**

- Easy to implement “High-Level Language” features in **your** Assembler Language
- Start with simple, concrete, useful forms
- Build new “language” elements incrementally
- Useful results directly proportional to implementation effort
  - Create as few or as many capabilities as needed
  - Checking and diagnostics as simple or elaborate as desired
- New language can precisely match application requirements
- Best of all: it's fun!

HLASM Macro Tutorial

Summary

This overview has conveyed how concepts of typical high-level languages can be implemented in Assembler Language in a controlled, incremental, and comprehensible way. Nothing unusual has been done here: all macro actions and designs are straightforward, with simple goals and results.

These macro techniques are also useful for teaching:

- You can start with very simple, concrete examples before attempting complex or abstract designs.
- From a simple base, you can elaborate and extend the macros in many directions, to enhance whatever features are interesting.
- You can create a "language of choice" with as few or as many features as desired. For example, it is easy to design a "mini-language" with at least two different data types, conversions between them, operations on each (possibly involving mixing of types), and input-output operations (possibly involving conversions to and from "external" representations).[13]

The best aspect of using macros to build your own language is that you can watch what is happening at each stage, and elaborate or tailor the results as desired.

A humorous example of dynamic language modification appeared many years ago in the *Reader's Digest*.[14]

> In a letter to *The Economist*, M. J. Shields, of Jarrow, England, points out that George Bernard Shaw, among others, urged spelling reform, suggesting that one letter be altered or deleted each year, thus giving the populace time to absorb the change. Shields writes:
>
> > For example, in Year 1 that useless letter "c" would be dropped to be replased by either "k" or "s," and likewise "x" would no longer be part of the alphabet. The only kase in which "c" would be retained would be the "ch" formation, which will be dealt with later. Year 2 might well reform "w" spelling, so that "which" and "one" would take the same konsonant, wile Year 3 might well abolish "y" replasing it with "i," and Iear 4 might fiks the "g-j" anomali wonse and for all.
> >
> > Jenerally, then, the improvement would kontinue iear bai iear, with Iear 5 doing awai with useless double konsonants, and Iears 6-12 or so modifaiing vowlz and the rimeining voist and unvoist konsonants. Bai Ier 15 or sou, it wud fainali bi posibl tu meik ius ov thi ridandant letez "c," "y" and "x" -- bai now jast a memori in the maindz ov ould doderez -- tu riplais "ch," "sh" and "th" rispektivli.
> >
> > Fainali, xen, aafte sam 20 iers of orxogrefkl riform, wi wud hev a lojikl, kohirnt speling in ius xrewawt xe Ingliy-spiking werld. Haweve, sins xe Wely, xe Airiy, and xe Skots du not spik Ingliy, xei wud hev to hev a speling siutd tu xer oun lengwij. Xei kud, haweve, orlweiz lern Ingliy az a sekond lengwij et skuul!
> >
> > -- Iorz feixfuli, M. J. Yilz.

---

[13] The author has seen examples of macro sets to perform recursive-descent parsing of expressions; to generate in-line code for Format-statement conversion expansions; and even a single macro named "FORTRAN" followed by a Fortran program all of whose statements were read by AREAD statements!

[14] Reprinted by permission.

External Conditional Assembly Functions

ExtFunc-199

External Functions

External Conditional Assembly Functions

ExtFunc-200

- Two types of external, user-written functions
  1. Arithmetic functions: like &A = AFunc(&A1, &A2, ...)

```
&A     SetAF  'AFunc',&A1,&A2,...      Arithmetic arguments
&LogN  SetAF  'Log2',&N                Logb(&N)
```

  2. Character functions: like &C = CFunc('&S1', '&S2', ...)

```
&C     SetCF  'CFunc','&S1','&S2',...  String arguments
&RevX  SetCF  'Reverse','&X'           Reverse(&X)
```

- Functions can access the assembly environment
- Link from HLASM uses standard linkage conventions
  - Assembler provides a save area and a 4-doubleword work area
  - Zero to many arguments
- Functions may provide messages with severity codes for the listing
- Return code indicates success or failure
  - Failure return terminates the assembly

External Conditional Assembly Functions

High Level Assembler supports a powerful capability for invoking externally-defined functions *during the assembly* that can perform almost any desired action. They are invoked using the conditional assembly statements SETAF and SETCF, by analogy with the familiar SETA and SETC statements.

The syntax of the statements is similar to that of SETA and SETC: a local or global variable symbol appears in the name field; it will receive the value returned from the function. The operation mnemonic indicates the type of function to be called, and the type of value to be assigned to the "target" variable. The first operand in each case is a quoted character expression (typically a character string) giving the name of the function to be called. The remaining operands are optional, and their presence depends on the function: some functions require no arguments, others may require several. The type of each of these arguments is the same as that of the target variable: arithmetic arguments for SETAF, and character arguments for SETCF.

A compact notational representation of this description is

```
&Arith_Var   SETAF  'Arith_function'[,arith_val]...
&Char_Var    SETCF  'Char_function'[,character_val]...
```

For example, we might invoke the LOG2 and REVERSE functions (to be discussed in detail below) with these two statements:

```
&LogN  SetAF  'Log2',&N                Logb(&N)
&RevX  SetCF  'Reverse','&X'           Reverse(&X)
```

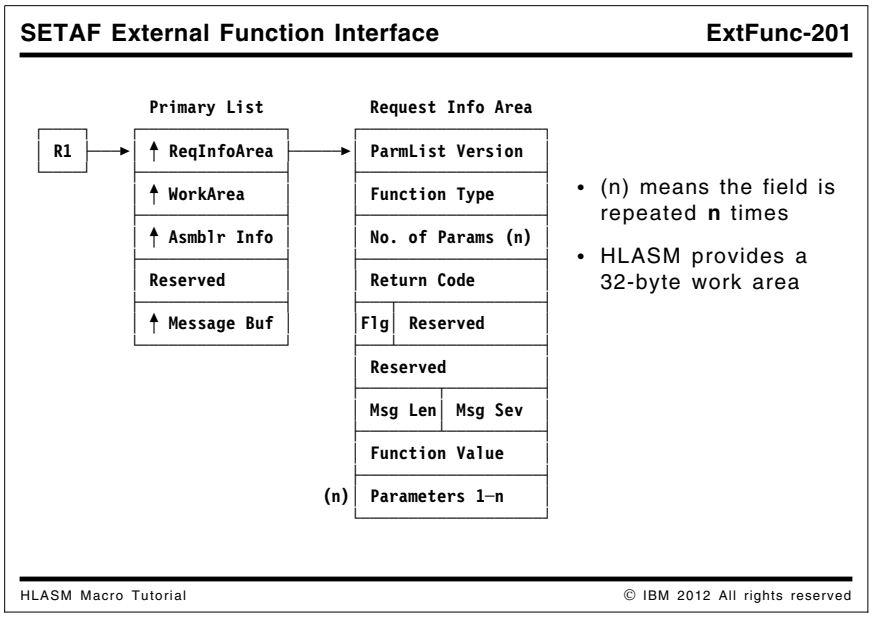
When a function is first invoked, the assembler dynamically loads the module containing the function and prepares control structures for calling the function. The call uses standard operating system calling conventions; the assembler creates the calling sequence using the parameters and the function name supplied in the SETxF statement.

Using standard parameter-passing conventions, the assembler sets R1 to point to a list of addresses. The first address in this primary list points to a "Request Information Area", a list of integer values describing the type of function (arithmetic or character), the version of the interface, the number of arguments, the return code, and either the returned value and the integer arguments (for SETAF), or the lengths of the respective argument strings (for SETCF). The remaining items in the primary list pointed to by R1 are pointers to a 32-byte work area, and (for SETCF) pointers to the result string and each of the argument strings.

HLASM provides ways for external function to return messages and severity codes; this allows functions to detect and signal error conditions in a way similar to the facility provided by I/O exits.

At the end of the assembly, HLASM checks to see if each called external function wants a final "closing" call so it can free any resources it may have acquired. Finally, the assembler's summary page lists for each function the number of SETAF and SETCF calls, the number of messages issued, and the highest severity code returned by the function.

We will illustrate the capabilities of these functions with two simple examples: an arithmetic function LOG2 to evaluate the binary logarithm of an integer argument, and a string function REVERSE to reverse the characters in a character-string argument. These examples don't really require an external function; they can be programmed easily (if inelegantly) using familiar conditional assembly statements. However, because they also can access the environment in which HLASM is executing, external functions have considerably greater power and flexibility than the conditional language alone can provide.



SETAF External Function Interface

The interface used by High Level Assembler to invoke external arithmetic-valued functions is a standard calling sequence, with an argument list composed of two structures: the formats of the Primary Address List and the Request Information Area are shown in Figure 114. (Symbolic mappings of the Primary List and the Request Information Area are provided by the ASMAEFNP macro.)

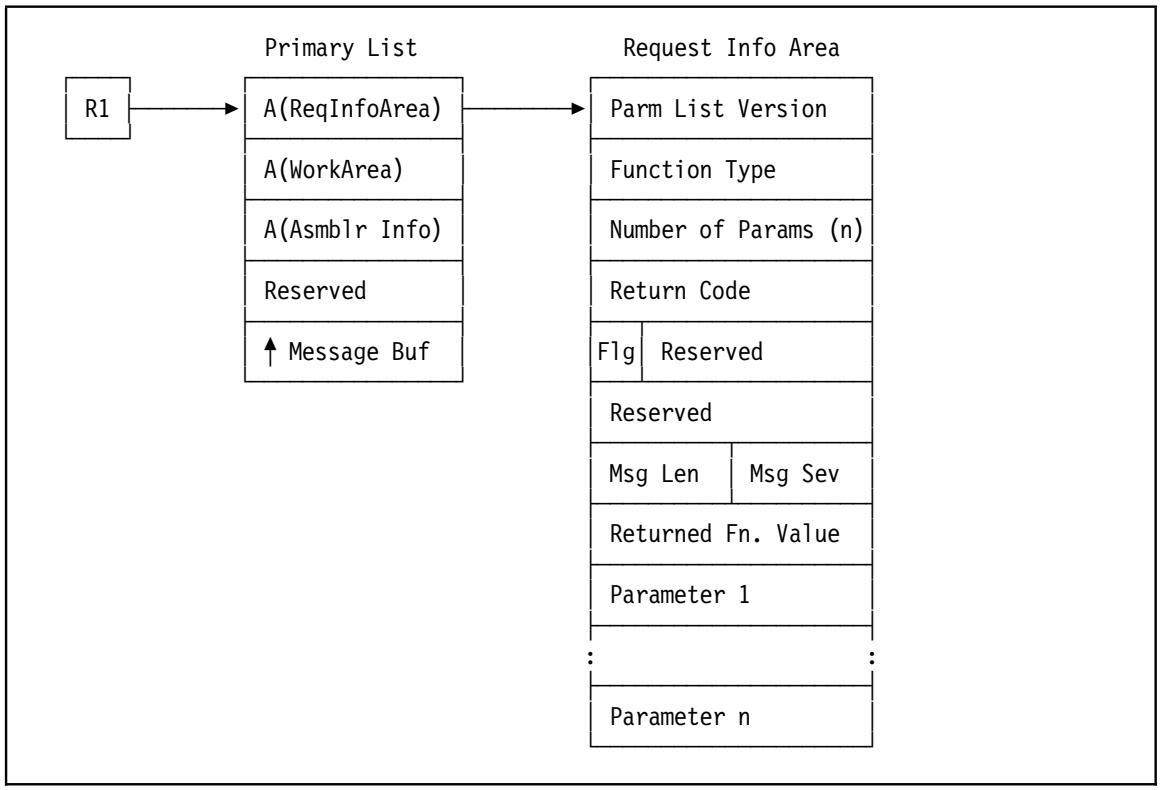


Figure 114. Interface for Arithmetic (SETAF) External Functions

Arithmetic-Valued Function Example: LOG2

This LOG2 function evaluates the binary logarithm of its single argument, and returns the exponent of the largest power of two not exceeding the value of the argument. Mathematically, the result of calling LOG2 with argument x can be expressed as

$$result = floor(\log_2(x))$$

This can be used to calculate the largest power of two less than or equal to **x**. For example, if &Exponent is an arithmetic variable symbol whose value will be returned by calling LOG2, the power of two can be found using statements such as

```
  &Exponent    SETAF  'Log2',&Arith_Var
```

Special treatment is provided for non-positive arguments, for which the binary logarithm is undefined. Invalid calls to LOG2 cause either an error message or a nonzero return code to be returned to the assembler (which will then terminate the assembly).

We will now describe an implementation of the LOG2 function. It uses no local storage, and may reside anywhere below or above 16MB.

```
LOG2     Title 'HLASM Conditional-Assembly Function LOG2'
*************************************************************************
*                                                                       *
*        Call from High Level Assembler:                                *
*                                                                       *
* &Int_Ans  SETAF  'LOG2',&Int_Arg                                      *
*                                                                       *
*        If &Int_Arg > 0, &Int_Ans is set to floor(log2(&Int_Arg))      *
*                         That is, to the largest N such that           *
*                         2**N <= &Int_Arg.                             *
*                                                                       *
*        If the function is invoked incorrectly, the return code        *
*        will indicate the reason, and the assembler will terminate     *
*        the assembly. An appropriate message is provided, except       *
*        when the wrong parameter list version is detected, in which    *
*        the function causes assembly termination (the interface for    *
*        returning a message may not be available).                     *
*                                                                       *
*************************************************************************
```

Figure 115. Conditional-Assembly Function LOG2: Initial Commentary

The block of comments in Figure 115 describes the operation of the function, the returned function values, and error handling.

```
LOG2     RSect ,                  Code is reentrant, read-only
LOG2     AMode Any                No dependence on addressing mode
LOG2     RMode Any                No dependence on residence mode
*********************************************************************
*        Primary Entry Point                                        *
*********************************************************************
         Using LOG2,R15           Addressability for code
         STM   R14,R4,D12(R13)    Save caller's registers
         Using AEFNPARM,R1        Map the Primary List
         L     R2,AEFNRIP         Load address of Request Info Area
         Using AEFNRIL,R2         Map Request Info Area
         XC    AEFNRETC,AEFNRETC  Set Return Code area to zero
         XC    AEFN_VALUE,AEFN_VALUE   Set answer to zero also
```

Figure 116. Conditional-Assembly Function LOG2: Entry

The entry point instructions illustrated in Figure 116 save appropriate general registers and establish mappings for the Primary List and the Request Information Area. The return code field is set to zero, indicating that the assembler can continue. (This field will be changed if the parameter list version is invalid.) In case the assembly might continue in spite of errors, the result field is set to zero.

```
*********************************************************************
*        Validate Calling Sequence                                  *
*********************************************************************
         CLC   AEFNVER,=A(AEFNVER2)   Check for expected version
         JL    Err_LVer               Branch if wrong PList version
         CLC   AEFNVER,=A(AEFNVER3)   Check for expected version
         JH    Err_LVer               Branch if wrong PList version
         CLC   AEFNTYPE,=A(AEFNSETAF) Check for SETAF function call
         JNE   Err_FTyp               Branch if wrong function type
         CLC   AEFNUMBR,=A(OurNArgs)  Check for single argument
         JNE   Err_NArg               Branch if wrong # of arguments

*********************************************************************
*        Calling sequence is valid, check value of argument         *
*********************************************************************
         L     R3,AEFN_PARM1          Get function argument in R1
         LTR   R3,R3                  Check for nonnegative argument
         JZ    Zero_Arg               Branch if zero argument
         JM    Neg_Arg                Branch if negative argument
```

Figure 117. Conditional-Assembly Function LOG2: Validation

The instructions illustrated in Figure 117 first validate that the function is being invoked with the expected calling sequence. The function type, parameter list version, and number of arguments are checked, and error messages for the assembler will be used to indicate improper invocations. Once the interface has been checked, the argument itself is tested.

```
*********************************************************************
*        Calculate Floor(Log2(argument)) in R0                      *
*********************************************************************
         LA    R4,31               Set answer to 1 past max possible
TestLoop DC    OH'0'               Check magnitude of the argument
         BCTR  R4,Null             Count answer down by 1
         JXH   R3,R3,TestLoop      Double arg, branch if no overflow

*********************************************************************
*        Store result and return to High Level Assembler            *
*********************************************************************
         ST    R4,AEFN_VALUE       Store result in Request Info List
         LM    R2,R4,D28(R13)      Restore registers
         BR    R14                 Return to Assembler
```

Figure 118. Conditional-Assembly Function LOG2: Computation

The "computation" of the logarithm itself is quite simple, as shown in Figure 118. The JXH instruction effectively doubles the value in R3 each time it is executed, and compares the doubled result to the previous (un-doubled) value. When a bit overflows into the sign position, the JXH branch-test condition will fail and control will pass to the sequence that stores the result and returns control to the assembler.

```
*********************************************************************
*        Handle zero and negative arguments                         *
*********************************************************************
Zero_Arg DC    OH'0'               Return for negative argument
         LA    R4,BadArgZ          Point to error message
         J     Err_Exit            And return with a message

Neg_Arg  DC    OH'0'               Return for negative argument
         LA    R4,BadArgN          Point to error message
         J     Err_Exit            And return with a message

*********************************************************************
*        Handle invalid calling sequences                           *
*********************************************************************
Err_LVer DC    OH'0'               Wrong interface version
         MVC   AEFNRETC,=A(AEFNBAD) Can't count on doing a message
         J     Return              Return to Assembler immediately

Err_FTyp DC    OH'0'               Wrong function type
         LA    R4,BadFun           Point to error message
         J     Err_Exit            Return to Assembler

Err_NArg DC    OH'0'               Wrong number of arguments
         LA    R4,BadNum           Point to error message
```

Figure 119. Conditional-Assembly Function LOG2: Error Handling

The error-handling code in Figure 119 provides either an immediate termination return to the assembler (at Err_LVer) in case the parameter list format is unacceptable, or points to an error message and its preceding length byte.

```
*********************************************************************
*        Issue Error Messages and Return to HLASM                   *
*********************************************************************
Err_Exit DC    OH'O'
         MVC   AEFNMSGS,=Y(ErrSev)  Set error message severity
         L     R1,AEFNMSGA          Get pointer to message buffer
         Drop  R1
         XR    R3,R3                Clear for message length
         IC    R3,DO(,R4)           Get message length
         STH   R3,AEFNMSGL          Store for assembler's use
         BCTR  R3,Null              Decrement for MVC instruction
         EX    R3,Move_Msg          Move message to buffer

Return   DC    OH'O'                Return to HLASM
         LM    R2,R4,28(R13)        Restore R2-R4
         Drop  R2,R15               Release addressability
         BR    R14                  Return to assembler

Move_Msg MVC   DO(*-*,R1),D1(R4)    Executed
```

Figure 120. Conditional-Assembly Function LOG2: Error Message Handling

The error-handling code in Figure 120 moves messages to the assembler's message buffer, and sets the message severity code to 12 (as defined by the symbol ErrSev).

```
*********************************************************************
*        Error Messages                                             *
*********************************************************************
BadFun   DC    AL1(L'BadFunM)      Length of message
BadFunM  DC    C'Wrong function type (not SETAF)'

BadNum   DC    AL1(L'BadNumM)      Length of message
BadNumM  DC    C'Wrong number of arguments (not 1)'

BadArgZ  DC    AL1(L'BadArgZM)     Length of message
BadArgZM DC    C'Zero argument'

BadArgN  DC    AL1(L'BadArgNM)     Length of message
BadArgNM DC    C'Negative argument'
```

Figure 121. Conditional-Assembly Function LOG2: Error Message Handling

Each message text shown in Figure 121 is defined with a preceding byte containing its length.

```
*********************************************************************
*        Equates for Registers and Displacements                    *
*********************************************************************
Null     Equ   0                  Null Register for BCTR
R1       Equ   1                  A(Parm list), A(msg buffer)
R2       Equ   2                  A(Req info list)
R3       Equ   3                  Arg test, msg length
R4       Equ   4                  Result value, msg address
R13      Equ   13                 Save area
R14      Equ   14                 Return address
R15      Equ   15                 Code base

D0       Equ   0                  Displacement 0
D1       Equ   1                  Displacement 1
D12      Equ   12                 Displacement 12
D28      Equ   28                 Displacement 28
```

Figure 122. Conditional-Assembly Function LOG2: Symbol Equates

The equates shown in Figure 122 are typical, except that symbols are defined for use wherever an absolute displacement is to be used in an instruction. This technique helps in locating (and, if necessary, modifying) what would otherwise be non-symbolic references in instructions.

```
*********************************************************************
*        Equates for values used in argument and call validations   *
*********************************************************************
OurNArgs Equ   1                  Expected number of arguments
ErrSev   Equ   12                 Severity code for all messages

AEFN_PARM1 Equ AEFN_PARMN         First argument in list
```

Figure 123. Conditional-Assembly Function LOG2: Validation Equates

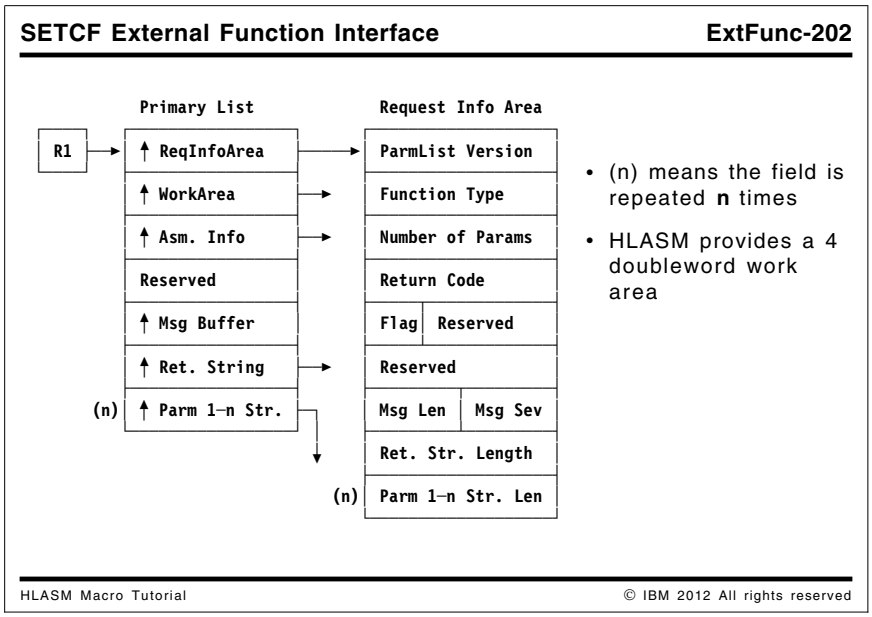
The symbols defined in Figure 123 define the expected value of the number of arguments in the Request Information Area provided by the assembler, and the severity code used for messages. The symbol AEFN_PARM1 is equated to the first item in the argument list; it is used only for its symbolic value.

```
*********************************************************************
*        Dummy Control Sections for SETAF Interface                 *
*********************************************************************
         ASMAEFNP  PRINT=GEN
         End
```

Figure 124. Conditional-Assembly Function LOG2: Dummy Sections

Finally, the Request Information Area is mapped by the expansion of the ASMAEFNP macro supplied with HLASM, as shown in Figure 124.

Installing the LOG2 function is described on page 248.



SETCF External Function Interface

The assembler interface for character functions is illustrated in Figure 125, where the formats of the Primary Address List and the Request Information Area are shown.

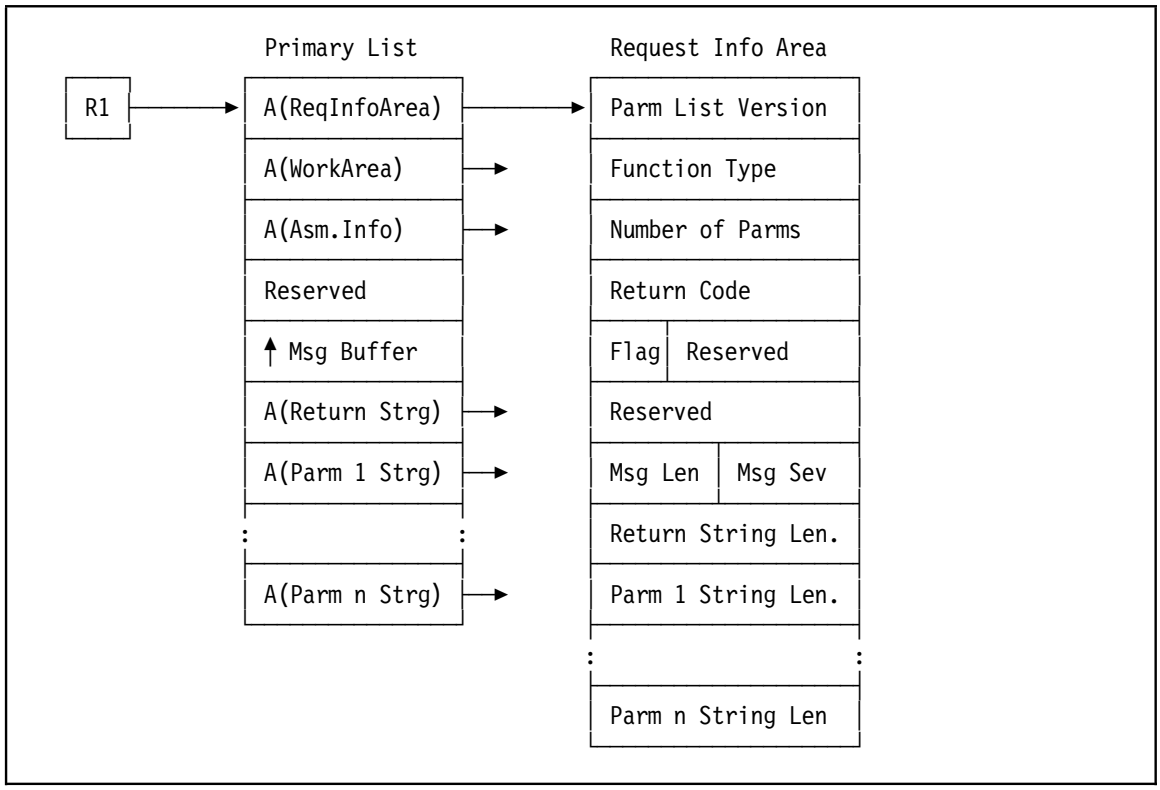


Figure 125. Interface for Character (SETCF) External Functions

String-Valued Function Example: REVERSE

The REVERSE function accepts a single string argument of zero to 256 characters, and returns a string of the same length with the characters in reverse order.

The implementation described here uses no local or working storage, and may reside anywhere above or below 16MB.

```
REV      Title 'Macro-Time Function REVERSE: Reverse Character Strings'
*********************************************************************
*                                                                   *
*        This external function reverses a character string. Null   *
*        strings are acceptable.                                    *
*                                                                   *
*        If the function is invoked with an unsupported parameter   *
*        list version, the assembly will be terminated. Other error *
*        conditions will be indicated by an error message, and a    *
*        null string will be returned. Errors detected are:         *
*                                                                   *
*           Function was not invoked by SETCF                       *
*           Number of arguments was not 1                           *
*           Argument string length was not 0-256                    *
*                                                                   *
*********************************************************************
```

Figure 126. Conditional-Assembly Function REVERSE: Prologue Text

The prologue text for the REVERSE function shown in Figure 126 describes the operation of the function, and the error conditions diagnosed. If the parameter list version is not supported, the assembler will be requested to terminate the assembly, as there is no guarantee that a message can be provided by the function.

```
REVERSE  Rsect ,                  Code is reentrant, read-only
REVERSE  AMode Any                No dependence on addressing mode
REVERSE  RMode Any                No dependence on residence mode
         Using Reverse,R15        Establish code base register
         STM   R14,R5,D12(R13)    Save caller's registers
         Using AEFNPARM,R1        Map primary argument-address list
         L     R2,AEFNRIP         Get address of Request Info Area
         Using AEFNRIL,R2         Map Request Info Area
         XC    AEFNRETC,AEFNRETC  Set return code to zero
         XC    AEFN_STRL,AEFN_STRL Set return string to null
         L     R5,AEFNMSGA        Address of message buffer
```

Figure 127. Conditional-Assembly Function REVERSE: Entry Point

The entry point instructions in Figure 127 first save the caller's registers; no save area linkage is required, as the REVERSE function makes no further calls, and uses no system services. Then, the Primary Address List and the Request Information Area are mapped using fields defined by the ASMAEFNP macro. The return code and returned string length are set to zero, and R5 is set to point to the message buffer in case a message is to be produced. (Note that the Primary Address List contains more fields than were referenced in the LOG2 example.)

```
*********************************************************************
*        Validate calling sequence                                  *
*********************************************************************
         CLC   AEFNVER,=A(AEFNVER2)   Check for interface version
         JL    Err_LVer               Branch if bad PList version
         CLC   AEFNVER,=A(AEFNVER3)   Check for interface version
         JH    Err_LVer               Branch if bad PList version
         CLC   AEFNTYPE,=A(AEFNSETCF) Check for SETCF function call
         JNE   Err_FTyp               Branch if bad function type
         CLC   AEFNUMBR,=A(OurNArgs)  Check for single argument
         JNE   Err_NArg               Branch if bad number of arguments

         L     R3,AEFNCF_PARM1     Point R3 to argument string
         L     R1,AEFNCF_SA        Point R1 to returned string
         Drop  R1                  R1 no longer addresses primary list
```

Figure 128. Conditional-Assembly Function REVERSE: Call Validation

The instructions shown in Figure 128 validate that the version of the parameter list is correct, that the REVERSE function was invoked as a character function, and that there is a single argument. Then, pointers to the argument and result strings are established.

```
*********************************************************************
*        Check for invalid argument string length                   *
*********************************************************************
         L     R4,AEFN_PARM1_L     Get length of argument string
         LTR   R4,R4               Validate length of input string
         JM    Err_Arg             Branch if invalid argument
         JZ    Return              Branch if input string is null
         C     R4,=A(OurStMax)     Check for excess length
         JH    Err_Arg             Branch if invalid argument
```

Figure 129. Conditional-Assembly Function REVERSE: Argument Validation

In Figure 129, the length of the argument string is validated. If efficiency is a major concern, these validation checks could be omitted.

```
*********************************************************************
*        Argument is valid; set up reversing translate string       *
*********************************************************************
         ST    R4,AEFN_STRL        Set return string length
         LA    R5,EndTrans         Point 1 past end of translate table
         SR    R5,R4               Subtract argument length
         BCTR  R4,Null             Decrement count by 1 for move
         EX    R4,Move_TR          Move translation string to answer
         EX    R4,Tran_Ans         Reverse bytes of arg into answer

Return   DC    OH'0'
         LM    R2,R5,D28(R13)      Restore R2-R5
         BR    R14                 Return to Assembler

Move_TR  DC    OH'0'               Executed, length in R4
         MVC   D0(*-*,R1),D0(R5)   Move table 'tail' to result string

Tran_Ans DC    OH'0'               Executed, length in R4
         TR    D0(*-*,R1),D0(R3)   Translate with reversal into answer
```

Figure 130. Conditional-Assembly Function REVERSE: String Reversal

The instructions in Figure 130 perform the actual “work” of the REVERSE function. The length of the argument string is used to extract the proper number of bytes from the end of the translate table (which contains the byte values from X'FF' to X'00' in descending order), and place them in the output string area. Then, the output string is “translated” using the argument string as the “table”, yielding the reversed argument string as a result. The caller's register contents are then restored, and control is returned to the assembler.

The function could of course use an MVCIN instruction, but we can't guarantee it is available on the system doing the assembly.

```
*********************************************************************
*        Error Returns and Message Handling                         *
*********************************************************************
Err_LVer DC    OH'0'               Unsupported parameter list version
         MVC   AEFNRETC,=A(AEFNBAD) Termination return code
         J     Err_Exit             Return to Assembler

Err_Arg  DC    OH'0'               Return for invalid argument
         LA    R3,InvArg           Point to error message
         J     Err_Msg             Return message to Assembler

Err_FTyp DC    OH'0'               Wrong function type for this call
         LA    R3,BadFun           Point to error message
         J     Err_Msg             Return message to Assembler

Err_NArg DC    OH'0'               Wrong number of arguments
         LA    R3,BadNum           Point to error message
```

Figure 131 (Part 1 of 2). Conditional-Assembly Function REVERSE: Error Handling

```
Err_Msg  DC    OH'0'               Return error message to HLASM
         XR    R4,R4               Clear R4 for length
         IC    R4,D0(,R3)          Pick up message length
         STH   R4,AEFNMSGL         Save length for HLASM
         MVC   AEFNMSGS,=Y(ErrSev) Set message severity code
         BCTR  R4,Null             Decrement length for executed MVC
         EX    R4,Move_Msg         Move message to buffer

Err_Exit DC    OH'0'
         LM    R2,R5,D28(R13)      Restore R2-R5
         Drop  R2,R15              Addressability now lost
         BR    R14                 Return to Assembler to terminate

Move_Msg DC    OH'0'
         MVC   D0(*-*,R5),D1(R3)   Move message to buffer
```

Figure 131 (Part 2 of 2). Conditional-Assembly Function REVERSE: Error Handling

The instructions in Figure 131 on page 245 set the return code for a severe error in case the parameter interface version is not supported. For other possible error conditions detected during call and argument validation, the appropriate message is moved to the message buffer, and the severity is set to 12 (the value of ErrSev). Control is then returned to the assembler.

```
*********************************************************************
*        Error Messages                                             *
*********************************************************************
InvArg   DC    AL1(L'InvArgM)      Length of message text
InvArgM  DC    C'Argument length invalid'
BadFun   DC    AL1(L'BadFunM)      Length of message text
BadFunM  DC    C'Not invoked by SETCF'
BadNum   DC    AL1(L'BadNumM)      Length of message text
BadNumM  DC    C'Wrong number of arguments (not 1)'
```

Figure 132. Conditional-Assembly Function REVERSE: Error Messages

The error message texts (preceded by a length byte) are shown in Figure 132.

```
Trans    DC    0XL256'0',256AL1(255-(*-Trans)) Table from 255 to 0
EndTrans DC    0X'0'               End of translate string

         LtOrg
```

Figure 133. Conditional-Assembly Function REVERSE: Translate Table

The translate table defined in Figure 133 is a string of 256 byte values in descending order. The “tail” of this string is moved to the result string to be used as a translation source.

```
*********************************************************************
*        Equates for Registers, Lengths, Displacements, etc.        *
*********************************************************************
Null     Equ   0                  For BCTR instructions
R1       Equ   1                  Primary List, A(returned string)
R2       Equ   2                  A(Request Info List)
R3       Equ   3                  Message pointer
R4       Equ   4                  Lengths
R5       Equ   5                  A(TR table), A(message buffer)
R13      Equ   13                 Save area
R14      Equ   14                 Return address
R15      Equ   15                 Code base

D0       Equ   0                  Displacement 0
D1       Equ   1                  Displacement 1
D12      Equ   12                 Displacement 12
D28      Equ   28                 Displacement 28
```

Figure 134. Conditional-Assembly Function REVERSE: Basic Equates

Typical equates for the general purpose registers are defined in Figure 134, along with symbols representing displacements used in various instructions.

```
*********************************************************************
*        Equates for Parameter-List Values and Fields               *
*********************************************************************
OurNArgs Equ   1                  Expected number of arguments
ErrSev   Equ   12                 Error message severity
OurStMax Equ   256                Maximum allowed string length

AEFNCF_PARM1 Equ AEFNCF_PARMA     Name for first string parameter
AEFN_PARM1_L Equ AEFN_PARMN_L     Name for first string length
```

Figure 135. Conditional-Assembly Function REVERSE: Validation Equates

The symbols used in call and argument validation are defined in Figure 135. The last two symbols simplify references to the first (and only) argument.

```
*********************************************************************
*        Dummy Control Sections for SETCF Interface                 *
*********************************************************************
         ASMAEFNP  PRINT=GEN
         End
```

Figure 136. Conditional-Assembly Function REVERSE: Dummy Sections

The DSECT mappings for the Primary Address List and the Request Information Area are created by the expansion of the ASMAEFNP macro, as shown in Figure 136.

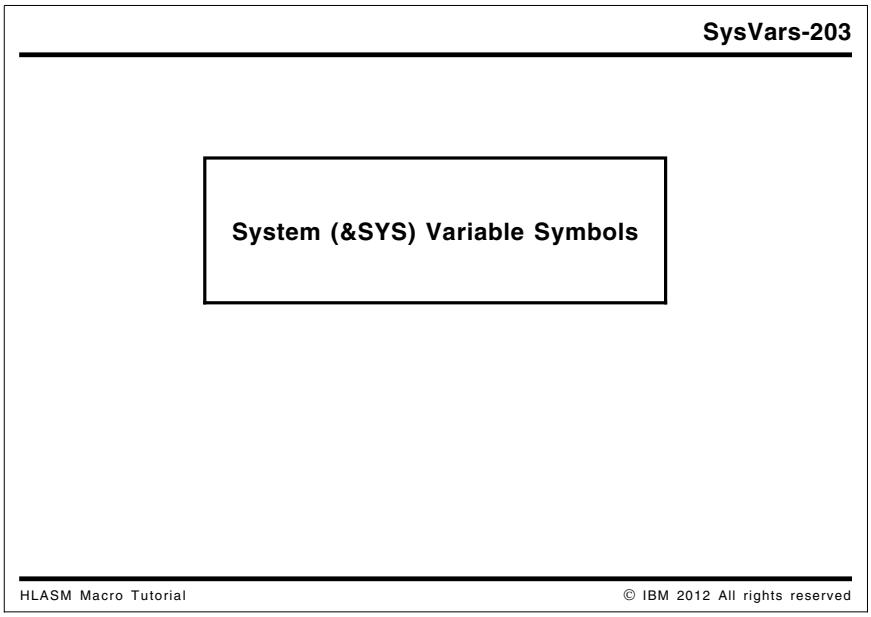
Note: HLASM supports character variables as long as 1024 characters. You may enjoy modifying this function to accept and process arguments from 0 to 1024 characters in length.

Installing the LOG2 and REVERSE Functions

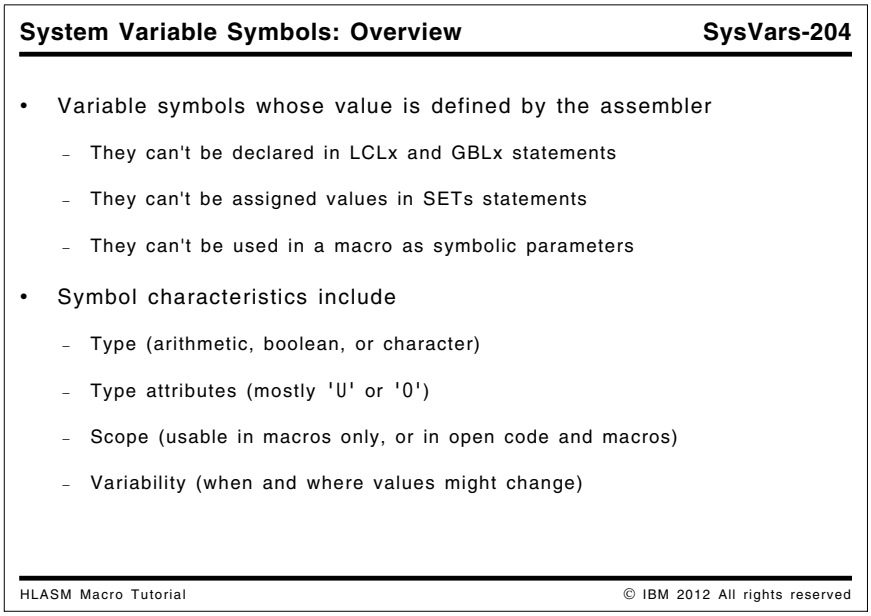
First, assemble the statements for the exit and convert the resulting object file into a loadable module:

- on MVS, the object file is link edited into an appropriate library and given the name LOG2 or REVERSE (as appropriate). It may be marked re-entrant if desired. Be sure that the library containing the function modules is available to the assembler during subsequent assemblies that require the functions.
- on CMS, LOAD the text deck from the assembly with the CLEAR and RLDSAVE options; then GENMOD to obtain a loadable file with name LOG2 or REVERSE (as appropriate) and file type MODULE. Be sure that the minidisk containing the function modules is available to the assembler during subsequent assemblies that require the functions.

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.



System Variable Symbols: Properties

The symbols have a variety of characterizations:

- Type

  Most symbols have character values, and are 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 argument list item.

- Usage scope

  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", and 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. All have local scope.

  - Variables whose values may change within a single macro expansion are designated "Variable"; this applies to &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

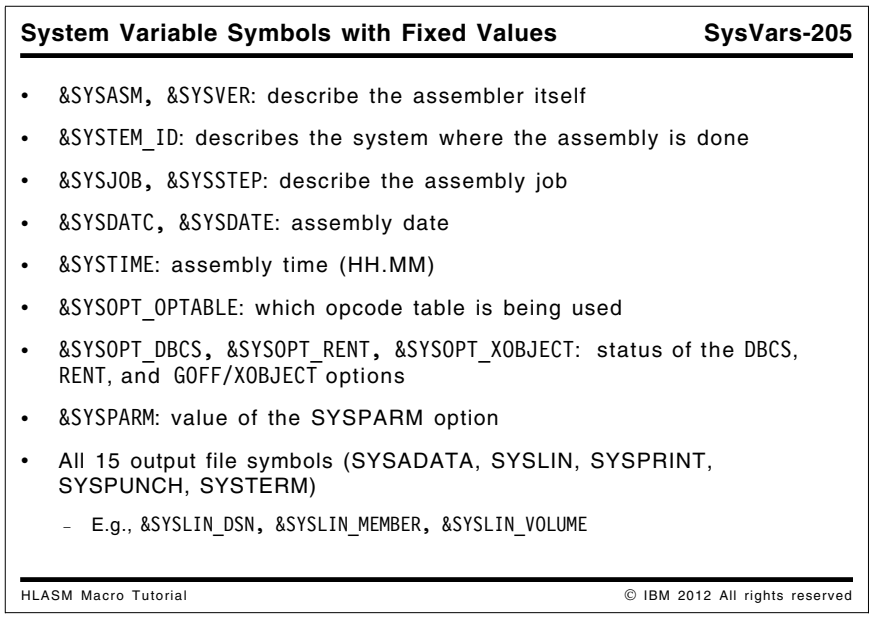
These symbols have many uses: helping to control conditional assemblies, capturing environmental data for inclusion in the generated object code, providing program debugging data, etc. Figure 137 on page 251 summarizes their properties.

Figure 137 (Page 1 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	Type	Type Attr.	Usage Scope	Variability	Content and Use
&SYSADATA_DSN	C	U	Local	Fixed	SYSADATA file data set name
&SYSADATA_MEMBER	C	U	Local	Fixed	SYSADATA file member name
&SYSADATA_VOLUME	C	U	Local	Fixed	SYSADATA file volume identifier
&SYSASM	C	U	Global	Fixed	Assembler name
&SYSCLOCK	C	U	Local	Constant	Date/time macro was generated
&SYSDATC	C,A	N	Global	Fixed	Assembly date, in YYYYMMDD format
&SYSDATE	C	U	Global	Fixed	Assembly date in MM/DD/YY format
&SYSECT	C	U	Local	Constant	Current control section name
&SYSIN_DSN	C	U	Local	Constant	Current primary input data set name
&SYSIN_MEMBER	C	U,O	Local	Constant	Current primary input member name
&SYSIN_VOLUME	C	U,O	Local	Constant	Current primary input data set name volume identifier
&SYSJOB	C	U	Global	Fixed	Assembly job name
&SYSLIB_DSN	C	U	Local	Constant	Current library data set name
&SYSLIB_MEMBER	C	U,O	Local	Constant	Current library member name
&SYSLIB_VOLUME	C	U,O	Local	Constant	Current library data set volume identifier
&SYSLIN_DSN	C	U	Local	Fixed	SYSLIN file data set name
&SYSLIN_MEMBER	C	U	Local	Fixed	SYSLIN file member name
&SYSLIN_VOLUME	C	U	Local	Fixed	SYSLIN file volume identifier
&SYSLIST	C	any	Local	Constant	Macro argument list and sublist elements
&SYSLOC	C	U	Local	Constant	Current location counter name
&SYSM_HSEV	C	N	Global	Variable	Highest MNOTE severity so far in assembly
&SYSM_SEV	C	N	Global	Variable	Highest MNOTE severity for most recently called macro
&SYSMAC	C	U,O	Local	Constant	Name of current macro and its callers
&SYSNDX	C,A	N	Local	Constant	Macro invocation count
&SYSNEST	A	N	Local	Constant	Nesting level of the macro call
&SYSOPT_DBCS	B	N	Global	Fixed	Setting of DBCS invocation parameter
&SYSOPT_OPTABLE	C	U	Global	Fixed	Setting of OPTABLE invocation parameter
&SYSOPT_RENT	B	N	Global	Fixed	Setting of RENT invocation parameter
&SYSOPT_XOBJECT	B	N	Global	Fixed	Setting of XOBJECT/GOFF invocation parameter

Figure 137 (Page 2 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	Type	Type Attr.	Usage Scope	Variability	Content and Use
&SYSPARM	C	U,O	Global	Fixed	Value provided by SYSPARM invocation parameter
&SYSPRINT_DSN	C	U	Local	Fixed	SYSPRINT file data set name
&SYSPRINT_MEMBER	C	U	Local	Fixed	SYSPRINT file member name
&SYSPRINT_VOLUME	C	U	Local	Fixed	SYSPRINT file volume identifier
&SYSPUNCH_DSN	C	U	Local	Fixed	SYSPUNCH file data set name
&SYSPUNCH_MEMBER	C	U	Local	Fixed	SYSPUNCH file member name
&SYSPUNCH_VOLUME	C	U	Local	Fixed	SYSPUNCH file volume identifier
&SYSSEQF	C	U,O	Local	Constant	Sequence field of current open code statement
&SYSSTEP	C	U	Global	Fixed	Assembly step name
&SYSSTMT	C,A	N	Global	Variable	Number of next statement to be processed
&SYSSTYP	C	U,O	Local	Constant	Current control section type
&SYSTEM_ID	C	U	Global	Fixed	System on which assembly is done
&SYSTERM_DSN	C	U	Local	Fixed	SYSTERM file data set name
&SYSTERM_MEMBER	C	U	Local	Fixed	SYSTERM file member name
&SYSTERM_VOLUME	C	U	Local	Fixed	SYSTERM file volume identifier
&SYSTIME	C	U	Global	Fixed	Assembly start time
&SYSVER	C	U	Global	Fixed	Assembler version



Variable Symbols With Fixed Values During an Assembly

These sequence symbol values are established at the beginning of an assembly, and remain unchanged throughout the assembly.

&SYSASM and &SYSVER

&SYSASM provides the name of the assembler; its value is

```
HIGH LEVEL ASSEMBLER
```

&SYSVER describes the version, release, and modification level of the assembler. Its value increases monotonically across versions and releases of HLASM. A typical value of &SYSVER might be

```
1.6.0
```

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

&SYSTEM_ID identifies the operating system under which the assembly is being performed. A typical value of this variable might be

```
z/OS 1.12.0
```

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

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

&SYSJOB and &SYSSTEP

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

When assembling under CMS, the value of &SYSJOB is always (NOJOB); and when assembling under CMS or VSE, the value of &SYSSTEP 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
```

&SYSDATE

&SYSDATE provides the current date, in the form MM/DD/YY; it contains only the last two digits of the year. A typical value of this variable might be

```
06/26/92
```

&SYSTIME

&SYSTIME 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..'
```

Differences among &SYSTIME, &SYSCLOCK, and the CLOCKB and CLOCKD operands of the AREAD statement are discussed on page 261.

&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, such as generating BALR vs. BASR vs. BRAS.

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..'
```

---

* Not really typical, but important: HLASM V1R1 became available on June 26, 1992.

&SYSOPT_DBCS, &SYSOPT_RENT, and &SYSOPT_XOBJECT

&SYSOPT_DBCS, &SYSOPT_RENT, and &SYSOPT_XOBJECT provide the settings of the DBCS, RENT, and GOFF/XOBJECT options, respectively. They can be used to control generation of instructions or data appropriate to the type of desired object code, or to help control the scanning of DBCS 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 non-reentrant situations.

To illustrate, &SYSOPT_RENT 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

&SYSPARM provides the character string provided by the programmer in the SYSPARM option:

```
SYSPARM(string)
```

It 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
```

&SYS Symbols for Output Files

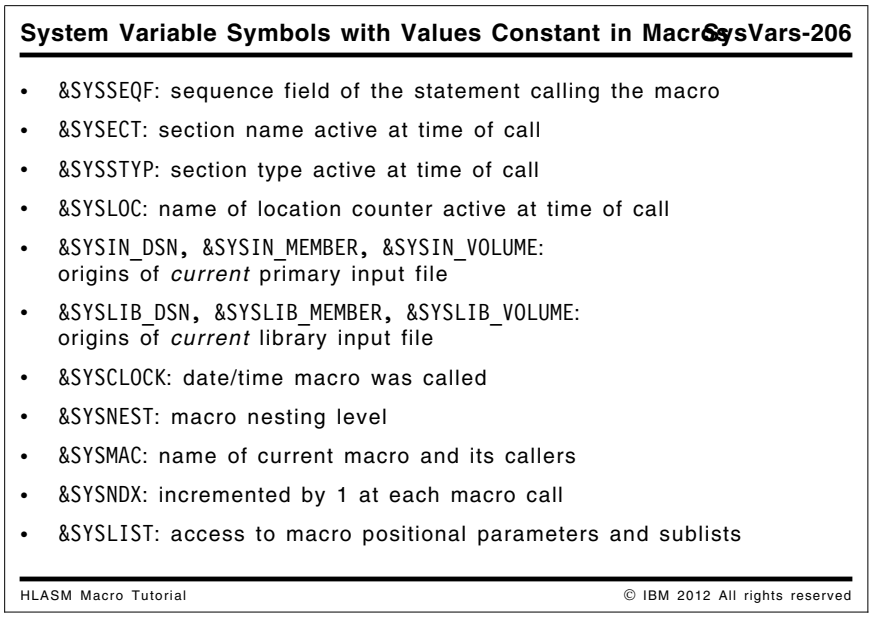
There are fifteen variable symbols describing the output files of High Level Assembler, three for each file:

File	DataSet Name	Member Name	Volume ID
SYSPRINT	&SYSPRINT_DSN	&SYSPRINT_MEMBER	&SYSPRINT_VOLUME
SYSTERM	&SYSTERM_DSN	&SYSTERM_MEMBER	&SYSTERM_VOLUME
SYSPUNCH	&SYSPUNCH_DSN	&SYSPUNCH_MEMBER	&SYSPUNCH_VOLUME
SYSLIN	&SYSLIN_DSN	&SYSLIN_MEMBER	&SYSLIN_VOLUME
SYSADATA	&SYSADATA_DSN	&SYSADATA_MEMBER	&SYSADATA_VOLUME

The &SYSxxxx_DSN symbols provide the file or data set name used for the corresponding output file; the &SYSxxxx_MEMBER symbols provide the member name (if any) used for the output file; and the &SYSxxxx_VOLUME symbols provide the volume identifier used for the output file.

To illustrate, suppose you wish to “capture” information about the destination of the object file written to the SYSLIN data set. You could write a set of statements such as:

```
What_OBJF  DC    C'SYSLIN file name is ''&SYSLIN_DSN.''.'
What_OBJM  DC    C'SYSLIN member is ''&SYSLIN_MEMBER.''.'
What_OBJV  DC    C'SYSLIN volume is ''&SYSLIN_VOLUME.''.'
```



Variable Symbols With Constant Values Within a Macro

These symbols' values are initialized when a macro expansion is initiated, and remain fixed throughout the duration of that expansion.

&SYSSEQF

&SYSSEQF provides the contents of the sequence field of the current input statement. This information can be used for debugging data. For example, suppose you 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

&SYSECT 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 lets your macro 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

&SYSSTYP provides the type of the control section into which statements are being grouped or assembled (CSECT, DSECT, COM, 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. Again, suppose 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.

- &SYSIN_DSN provides the name of the current primary input (SYSIN) data set or file.
- &SYSIN_MEMBER provides the name of the current primary input member, if any.
- &SYSIN_VOLUME 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.

- &SYSLIB_DSN provides the name of the library data set from which each macro and COPY file is retrieved.
- &SYSLIB_MEMBER provides the name of the library member from which this macro and COPY file is retrieved.
- &SYSLIB_VOLUME 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,'LIBINFO: 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 generate information in the object text of the generated program about the macro library.

```
My_Job   LIBINFO
         End
```

&SYSCLOCK

&SYSCLOCK provides the date and time at which the current macro was invoked, as a string of 26 characters:

```
'YYYY-MM-DD HH:MM:SS mmmmmm'
```

where mmmmmm is measured in microseconds. &SYSCLOCK can be used only in macros, not in open code.

Differences among &SYSTIME, &SYSCLOCK, and the CLOCKB and CLOCKD operands of the AREAD statement are discussed on page 261.

&SYSNEST

&SYSNEST provides the nesting level at which the current macro was invoked; the outermost macro called from open code 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
```

&SYSMAC

&SYSMAC provides the name of the macro currently being expanded, and of its entire call chain. If &SYSMAC is used without any subscript, it returns the name of the macro (or open code) in which it was used. If a subscript is provided, &SYSMAC(0) returns the same value as &SYSMAC; &SYSMAC(1) returns the name of the macro that called this one; and so forth for subscripts up to and including &SYSMAC(&SYSNEST), which returns 'OPEN CODE'. For values of the subscript greater than &SYSNEST, a null string is returned.

For example, instructions to display a macro's call chain might look like this:

```
&J      SetA  &SYSNEST
&K      SetA  &SYSNEST-&J
.Loop   ANop  ,
        MNOTE *,'Name at nesting level &J is &SYSMAC(&K)'
&J      SetA  &J.-1
        AIf   (&J ge 0).Loop
```

&SYSNDX

&SYSNDX has a unique value for every macro invocation in the program. It can 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 each time a macro is invoked; its value is constant within that macro, even if it invokes other inner macros.

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).

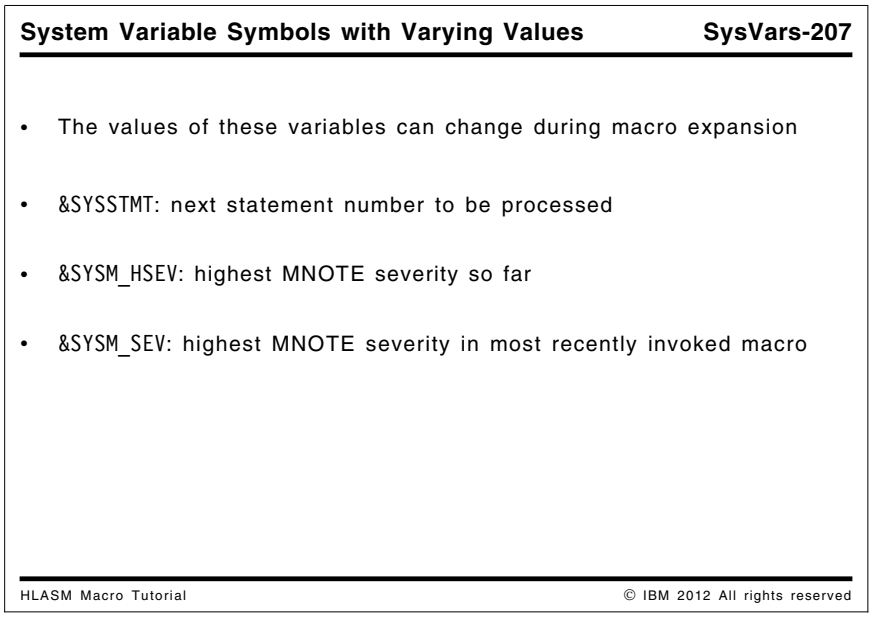
```
            Macro
&L          BDisp  &Target      Branch to non-addressable target
&L          BAS  1,Add&SYSNDX   Skip over constant
Off&SYSNDX  DC   Y(&Target-*)   Target offset
Add&SYSNDX  AH   1,Off&SYSNDX   Add offset
            BR   1              Branch to target
            MEnd
```

Although the *contents* of &SYSNDX is always decimal digits and can be used in SETA expressions, it is actually a character-valued variable.

&SYSLIST

&SYSLIST 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 quite different from the other system variable symbols.

```
&NameFld SETC   '&SYSLIST(0)'    Name field of macro call
&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

Three system variable symbols have values that can vary in all contexts: &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

&SYSSTMT

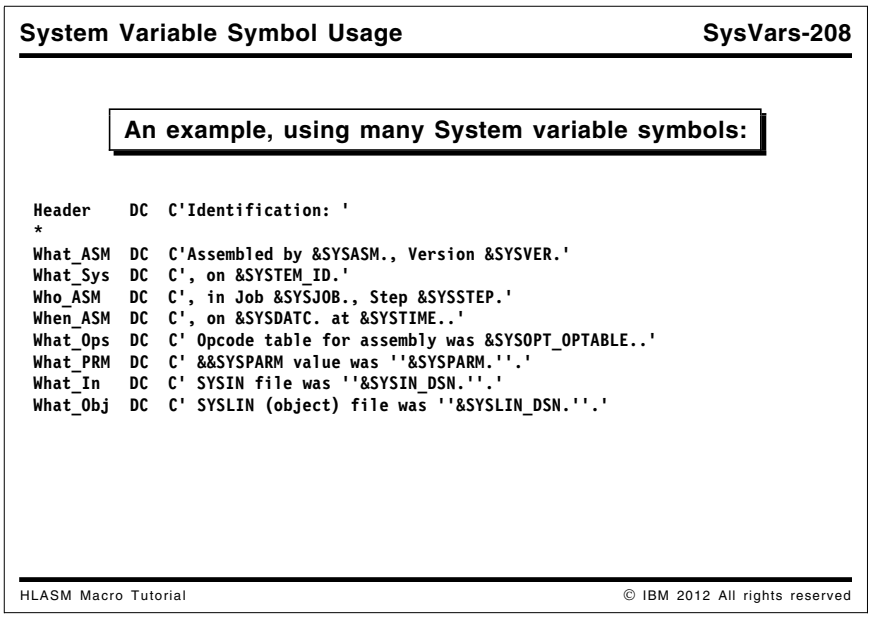
&SYSSTMT provides the number of the next statement to be processed by the assembler. Debugger data that depends on the statement number can be generated with this variable. For example, suppose we have a macro which inserts information about the current statement number into the object code of the program, 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
 +D       BAS   0,*+8             Addr of Statement Number in R0
 +        DC    AL4(00000527)     Statement number information
```

&SYSM_HSEV and &SYSM_SEV

&SYSM_HSEV and &SYSM_SEV provide the severity codes generated by MNOTE statements in macros called during the assembly. This can help a macro to determine that an inner macro call may have detected some special condition requiring action by the caller, without having to set global variables. Their values are returned as three numeric characters, such as 008.

&SYSM_SEV provides the highest MNOTE severity code for the macro most recently called from this macro or from open code. &SYSM_HSEV provides the highest MNOTE severity code for the entire assembly up to the point of reference to &SYSM_HSEV.



Example Using Many System Variable Symbols

You might want to insert information into the object code of a program describing its assembly environment, in a form readable without "translation" from hex. This example shows one way you can do this:

```
Header    DC  C'Identification: '
*
What_ASM  DC  C'Assembled by &SYSASM., Version &SYSVER.'
What_Sys  DC  C', on &SYSTEM_ID.'
Who_ASM   DC  C', in Job &SYSJOB., Step &SYSSTEP.'
When_ASM  DC  C', on &SYSDATC. at &SYSTIME..'
What_Ops  DC  C' Opcode table for assembly was &SYSOPT_OPTABLE..'
What_PRM  DC  C' &&SYSPARM value was ''&SYSPARM.''.'
What_In   DC  C' SYSIN file was ''&SYSIN_DSN.''.'
What_Obj  DC  C' SYSLIN (object) file was ''&SYSLIN_DSN.''.'
```

&SYSTIME, &SYSCLOCK, and the AREAD Statement

&SYSTIME 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.

There are two alternatives to the unvarying quality of &SYSTIME: &SYSCLOCK and the AREAD statement; &SYSCLOCK is described on page 258.

You can use operands of the AREAD statement if a more accurate time stamp is required. The current time can be obtained either in decimal or binary format.

This macro captures the clock reading in both decimal and binary formats:

```
         Macro
&Lab     AREADCLK
         LCLC   &Dec,&Bin
&Dec     Aread  CLOCKD
&Bin     Aread  CLOCKB
&Lab     DC     C'&Dec'      Decimal Clock
         DC     C'&Bin'      Binary  Clock
         MEnd

 A       AREADCLK
+A       DC     C'13020700'  Decimal Clock
+        DC     C'04692700'  Binary  Clock
```

Thus, you can capture time values at three levels of granularity:

- &SYSTIME provides the time at which the assembly began
- &SYSCLOCK provides the time at which the macro expansion began
- AREAD provides the current time whenever it is executed.

Comparing Ordinary and Conditional Assembly

Comparison	Ordinary Assembly	Conditional Assembly
Generality	The "inner" language of instructions and data definitions	The "outer" language that controls, tailors, and generates the inner language
Usage	A language for programming a processor	A language for programming an assembler and its inner language
Inputs	Statements from primary input, library (via COPY or macro call), and generated statements from macros and AINSERT statements	Statements from primary input (and records via AREAD), library (via COPY and macro call), external functions
Outputs	Generated machine language object code, object-file records (via REPRO, PUNCH), listing, ADATA, terminal messages	Ordinary assembly statements and macro instructions, messages (via MNOTE), records (via AINSERT)
Symbols	Ordinary symbols (internal and external)	Variable symbols, sequence symbols
Symbol declaration	Ordinary symbols appear in the name field of ordinary assembly statements (except names in V-type address constants); always explicitly declared	Sequence symbols appear in the name field of any statement; variable symbols are (a) user-declared (implicitly or explicitly), (b) system, or (c) macro parameters (both implicit)
Statement labels	Ordinary symbols take the values of locations in the ordinary assembly statement stream, and other assigned values, or are positional arguments in macro calls	Sequence symbols denote positions in the conditional assembly statement stream
Symbol scope	Internal and external; external symbols persist in the object code beyond assembly time	Variable symbols have local or global scope; sequence symbols have local scope; both discarded at assembly end
Symbol types and values	Ordinary symbols have no types; values are normally assigned from Location Counter values or by EQU statements	Variable symbols have arithmetic, boolean, or character types and values
Symbol attributes	Ordinary symbols may have Program, Length, Scale, Integer, and Assembler attributes	Variable symbols have only the property of maximum subscript (if dimensioned), but their *values* may have attributes
Expression evaluation	Expressions in ordinary statements, and in A-type and Y-type address constants	Expressions in conditional-assembly statements
Expression operators	`+, -, *, /`	`+, -, *, /`; internal arithmetic functions; internal boolean functions; internal character functions; external arithmetic and character functions
Attribute References	`L', I', S'`	`T', L', I', S', D', K', N', O'`

Figure 138. Comparison of Ordinary and Conditional Assembly

APAFOIL Processing Options

APAFOIL **Oct. 6, 2000** **Release 3.2**

Runtime values:

DEVICE 3820A
BIND (Odd, Even) 1.00i, 1.00i
TWOPASS . . . . . NO
INDEX . . . . . YES

Foil Set: 1

Input File (Current) HLA2VARS

Layout of Heading (FOILHD Tag or Default)
FOILHD . . . . . HEAD NULL NUMBER
FRAME . . . . . NONE

Layout of Body (LAYOUT Tag or Default)
FRAME . . . . . RULE
FRAMEWT . . . . . BOLD
RULE . . . . . SOLID
BORDER . . . . . NONE
RUBRICWT . . . . . LIGHT

Layout of Footing (FOILFT Tag or Default)
FOILFT . . . . . CONH NULL CONC
FRAME . . . . . NONE

Statistics:
Title Page 0
Contents 0
Parts 6
Foils 202
Notes 0
Overflow 0
Total Pages 208

APAFOIL Messages:
Information 0
Warning 0
Error 0

System Variables:
SYSVAR B . . . . . MIN
SYSVAR N . . . . . INCLUDE

Table Definitions

Figures

MACG0X6	HLA2MAC	73	22	73, 76
MACG006	HLA2MAC	77	24	76
MCXP04A	HLA2MAC	78	25	115
MACGN01	HLA2MAC	83	26	84
MACGN02	HLA2MAC	83	27	
ATTRTBL	HLA2MAC	87	28	
MACGSC1	HLA2MAC	90	29	90
MACNDFF	HLA2MAC	100	30	
MACFSBK	HLA2MAC	101	31	101
MCTXP02	HLA2MACT	112	32	111, 113
MCTXP03	HLA2MACT	114	33	187
MCTXP05	HLA2MACT	117	34	115, 116
MACMVC2	HLA2MACT	119	35	
MCTXM01	HLA2MACT	121	36	
MCTXMA1	HLA2MACT	122	37	
MCTXP06	HLA2MACT	125	38	
MCTXP07	HLA2MACT	126	39	125
MCTXP08	HLA2MACT	129	40	128
MCTXPA8	HLA2MACT	129	41	
MACCOMT	HLA2MACT	131	42	
MCTXP21	HLA2MACT	134	43	133
MCTXPZ1	HLA2MACT	134	44	
MCTXP22	HLA2MACT	136	45	136
MCTXP23	HLA2MACT	137	46	
MCTXP20	HLA2MACT	138	47	137, 138
MCTXP2X	HLA2MACT	139	48	138
MCTXB01	HLA2MACT	143	49	143
MCTXB02	HLA2MACT	144	50	
MCTXB03	HLA2MACT	146	51	145
MCTXBA4	HLA2MACT	146	52	
MCTXB04	HLA2MACT	147	53	164
MCTXB05	HLA2MACT	147	54	164

MCTXBB4	HLA2MACT	148	55	
MCTXBA5	HLA2MACT	148	56	
MCTXB06	HLA2MACT	149	57	
MCTXBA6	HLA2MACT	150	58	
MCTXB07	HLA2MACT	150	59	
MCTXBA7	HLA2MACT	150	60	
MCTXBF1	HLA2MACT	154	61	
MCTXB11	HLA2MACT	156	62	155
MCTXB12	HLA2MACT	158	63	
MCTXBF3	HLA2MACT	161	64	161
MCTXB13	HLA2MACT	163	65	160, 162
MCTXB14	HLA2MACT	164	66	
MCTBOF1	HLA2MACT	166	67	
MCTXT21	HLA2MACT	167	68	166
MCTXB21	HLA2MACT	169	69	169, 172
MCTXB22	HLA2MACT	171	70	
MCTBOF2	HLA2MACT	172	71	
MCTUD01	HLA2MACT	175	72	177
MCTUD02	HLA2MACT	175	73	
MCTUD0D	HLA2MACT	176	74	176
LOCTRF1	HLA2MACT	179	75	
MCTYCF1	HLA2MACT	181	76	189
MCTYCF2	HLA2MACT	182	77	184, 190
MCTYCF3	HLA2MACT	182	78	
MCTYCF4	HLA2MACT	185	79	
MACTYCX	HLA2MACT	186	80	
MCTYCFF	HLA2MACT	188	81	188
MCTYCF5	HLA2MACT	190	82	
MCTUD11	HLA2MACT	192	83	
MCTUD1A	HLA2MACT	192	84	192
MCTUD12	HLA2MACT	193	85	
MCTUD1B	HLA2MACT	194	86	193
MCTUD13	HLA2MACT	196	87	
MCTUD14	HLA2MACT	196	88	
MCTUD15	HLA2MACT	198	89	
MCTUD16	HLA2MACT			

		199	90	
MCTUD18	HLA2MACT	200	91	
MCTUD17	HLA2MACT	202	92	
FMTPA01	HLA2MACT	205	93	
FMTPA02	HLA2MACT	206	94	
FMTPA03	HLA2MACT	207	95	
FMTPA04	HLA2MACT	209	96	
FMTPA05	HLA2MACT	211	97	
FMTPA06	HLA2MACT	212	98	
FMTPA07	HLA2MACT	213	99	215
FMTPA4A	HLA2MACT	216	100	
FMTPA90	HLA2MACT	219	101	
FMTPA91	HLA2MACT	220	102	
FMTPA92	HLA2MACT	220	103	
MATRIX1	HLA2MACT	224	104	
MATRIX2	HLA2MACT	224	105	
FMTPA5A	HLA2MACT	225	106	
FMTPA5B	HLA2MACT	225	107	
FMTPA5C	HLA2MACT	226	108	
FMTPA5D	HLA2MACT	226	109	
FMTPA5E	HLA2MACT	227	110	
FMTPA5F	HLA2MACT	227	111	
FMTPA5G	HLA2MACT	227	112	
MCTFEX1	HLA2MACT	232	113	
XAFNF00	HLA2XFUN	236	114	236
XAFNF01	HLA2XFUN	237	115	237
XAFNF02	HLA2XFUN	238	116	238
XAFNF03	HLA2XFUN	238	117	238
XAFNF04	HLA2XFUN	239	118	239
XAFNF05	HLA2XFUN	239	119	239
XAFNFA5	HLA2XFUN	240	120	240
XAFNFA6	HLA2XFUN	240	121	240
XAFNF06	HLA2XFUN	241	122	241
XAFNF07	HLA2XFUN	241	123	241
XAFNF08	HLA2XFUN	241	124	241
XAFNF10	HLA2XFUN	242	125	

Headings

MACPAAR	HLA2MAC
MACPARM	HLA2MAC
MACARGS	HLA2MAC
MACPAAS	HLA2MAC
MCXBYS1	HLA2MAC
MACAATS	HLA2MAC
MACARGT	HLA2MAC
LISATTR	HLA2MAC
DATTR	HLA2MAC
MACARGC	HLA2MAC
MACNATR	HLA2MAC
MACSBLS	HLA2MAC
MACARGL	HLA2MAC
ATRSUM1	HLA2MAC
MACGBL	HLA2MAC
MACVSC	HLA2MAC
MACDBGT	HLA2MAC
MACDBGM	HLA2MAC
MACMHLP	HLA2MAC
MACDBAC	HLA2MAC
MACDBLM	HLA2MAC
MACDBPM	HLA2MAC
IBMMACS	HLA2MAC
MACSPEC	HLA2MAC
MACREC	HLA2MAC
MACENCD	HLA2MAC
MACNDFA	HLA2MAC
MACPACK	HLA2MAC
MACPUSE	HLA2MAC
CPATSYL	HLA2MAC
SUBLINM	HLA2MAC
MCT00	HLA2MACT
MCT01	HLA2MACT
MCTGEN1	HLA2MACT

Index Entries

Footnotes

254 *
254

Revisions

id	File	Page	References
S105	HLA2VARS	1	
S108	HLA2VARS	1	

Spots

id	File	Page	References
FDOUBLE	HLA2CONM	38	(no text) 48
CONSTRG	HLA2MAC	104	(no text) 103

Processing Options

Runtime values:

Document fileid	HLA2MTH SCRIPT
Document type	USERDOC
Document style	HLA2STYL
Profile	EDFPRF40
Service Level	0033
SCRIPT/VS Release	4.0.0
Date	12.07.03
Time	18:00:56
Device	3820A
Number of Passes	3
Index	YES
SYSVAR B	MIN
SYSVAR G	INLINE
SYSVAR H	NO
SYSVAR S	OFF
SYSVAR T	C
SYSVAR X	Y

Formatting values used:

Annotation	NO
Cross reference listing	YES
Cross reference head prefix only	NO
Dialog	LABEL
Duplex	SB
DVCF conditions file	(none)
DVCF value 1	(none)
DVCF value 2	(none)
DVCF value 3	(none)
DVCF value 4	(none)
DVCF value 5	(none)
DVCF value 6	(none)
DVCF value 7	(none)
DVCF value 8	(none)
DVCF value 9	(none)
Explode	NO
Figure list on new page	NO
Figure/table number separation	NO
Folio-by-chapter	NO
Head 0 body text	(none)
Head 1 body text	(none)
Head 1 appendix text	Appendix
Hyphenation	YES
Justification	NO
Language	ENGL
Keyboard	395
Layout	OFF

Leader dots	YES
Master index	(none)
Partial TOC (maximum level)	4
Partial TOC (new page after)	INLINE
Print example id's	NO
Print cross reference page numbers	YES
Process value	(none)
Punctuation move characters	(none)
Read cross-reference file	(none)
Running heading/footing rule	NONE
Show index entries	NO
Table of Contents (maximum level)	4
Table list on new page	YES
Title page (draft) alignment	CENTER
Write cross-reference file	(none)

Imbed Trace

Page 0	MYADDR SCRIPT
Page i	HLAPUBS SCRIPT
Page viii	HLA2VARS
Page viii	APAFOIL
Page viii	HLA2DATE
Page viii	HLA2CONM
Page 43	HLA2CO01
Page 45	HLA2CO02
Page 48	HLA2CO03
Page 49	HLA2CO04
Page 55	HLA2MAC
Page 60	HLA2MA01
Page 60	HLA2MA01
Page 61	HLA2MA02
Page 61	HLA2MA02
Page 62	HLA2MA04
Page 62	HLA2MA04
Page 64	HLA2MA05
Page 64	HLA2MA05
Page 65	HLA2MA12
Page 65	HLA2MA12
Page 66	HLA2MA03
Page 67	HLA2MA03
Page 67	HLA2MA06
Page 71	HLA2MA10
Page 77	HLA2MA11
Page 89	HLA2MA14
Page 99	HLA2MA15
Page 100	HLA2MA15
Page 105	HLA2MACT
Page 111	HLA2MT01
Page 123	HLA2MA07
Page 124	HLA2MA07
Page 133	HLA2MT07
Page 143	HLA2MT04
Page 145	HLA2MT05
Page 162	HLA2MT06
Page 165	HLA2MA08
Page 165	HLA2MA08
Page 172	HLA2MA09
Page 233	HLA2XFUN
Page 248	HLA1SVAR
Page 251	HLA1SVAT
Page 253	HLA1SVA2
Page 262	HLA1GLOT