Assembler Language as a Higher Level Language:

Macros and Conditional Assembly Techniques

SHARE 102, Winter Conference 2004, Sessions 8167-8168

John R. Ehrman
ehrman@vnet.ibm.com
ehrman@us.ibm.com

International Business Machines Corporation
Silicon Valley (nee Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, California 95141 USA

Synopsis:

This tutorial introduces the powerful concepts of conditional assembly supported by the IBM High Level Assembler for MVS & VM & 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.

Permission is granted to SHARE Incorporated to publish this material in the proceedings of the SHARE 102, Winter Conference 2004. IBM retains the right to publish this material elsewhere.

Copyright Notices and Trademarks

Publications, Collection Kits, Web Sites

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

- High Level Assembler for MVS & VM & VSE *Language Reference*, SC26-4940
- High Level Assembler for MVS & VM & VSE *Programmer's Guide*, SC26-4941
- High Level Assembler for MVS & VM & VSE *General Information*, GC26-4943
- High Level Assembler for MVS & VM & VSE *Licensed Program Specifications*, GC26-4944
- High Level Assembler for MVS & VM & VSE *Installation and Customization Guide*, SC26-3494

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

- High Level Assembler for MVS & VM & VSE *Release 2 Presentation Guide*, SG24-3910

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

- *VSE Collection*, SK2T-0060
- *MVS Collection*, SK2T-0710
- *Transaction Processing and Data Collection*, SK2T-0730
- *VM Collection*, SK2T-2067
- *OS/390 Collection*, SK2T-6700 (BookManager), SK2T-6718 (PDF)

HLASM publications are available online at the HLASM web site:

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

Acknowledgments

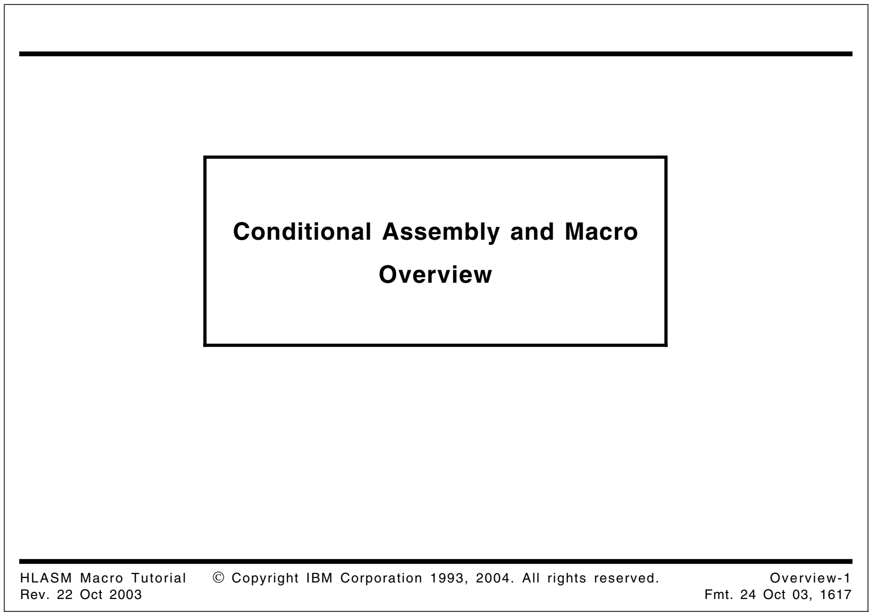
I am grateful to several people have helped make this material more reliable and readable:

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

Contents

Figures

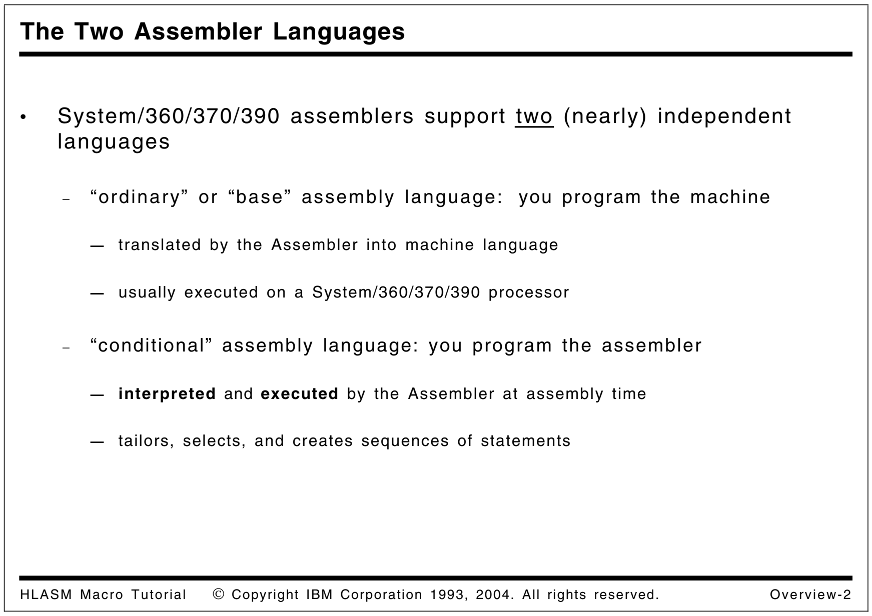
Conditional Assembly and Macro Overview



Conditional Assembly and Macro Overview

HLASM Macro Tutorial Overview-1
Rev. 22 Oct 2003 Fmt. 24 Oct 03, 1617

This presentation discusses a powerful capability of the System/360/370/390 assemblers that allows you to tailor programs to your specific needs: the “Conditional Assembly and Macro Facility”.



The Two Assembler Languages

- System/360/370/390 assemblers support two (nearly) independent languages
  - “ordinary” or “base” assembly language: you program the machine
    - translated by the Assembler into machine language
    - usually executed on a System/360/370/390 processor
  - “conditional” assembly language: you program the assembler
    - **interpreted** and **executed** by the Assembler at assembly time
    - tailors, selects, and creates sequences of statements

HLASM Macro Tutorial Overview-2

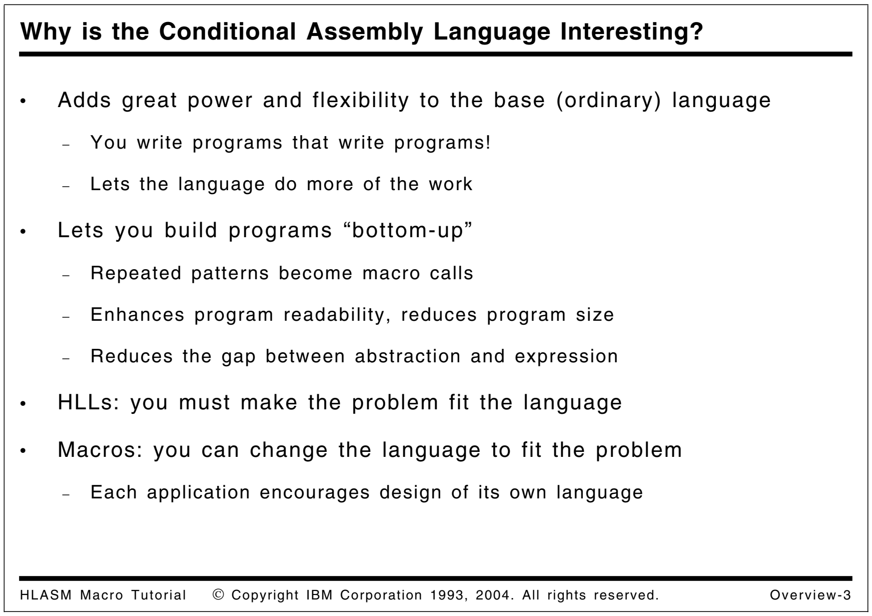
The System/360/370/390 Assembler Language is actually a mixture of two distinct languages:

- Ordinary assembly language — the base language of machine and assembler instruction statements, translated by the Assembler into machine language

- Conditional assembly and macro language — the language of conditional statements, variable symbols, and macros, *interpreted* and *executed* by the Assembler at assembly time to tailor, select, and create sequences of statements.

The conditional assembly and macro language is a special language with its own rules for declaring variables, assigning values, testing conditions, and generating values. The objects being manipulated at the time a program is assembled are primarily statements, character strings, and numeric quantities, so the conditional assembly language is oriented towards those items.

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. (Some people would reverse this characterization: the conditional language is so less well known that it seems to be hidden somewhere “inside” the more familiar ordinary language!)



Why is the Conditional Assembly Language Interesting?

- Adds great power and flexibility to the base (ordinary) language
  - You write programs that write programs!
  - Lets the language do more of the work
- Lets you build programs “bottom-up”
  - Repeated patterns become macro calls
  - Enhances program readability, reduces program size
  - Reduces the gap between abstraction and expression
- HLLs: you must make the problem fit the language
- Macros: you can change the language to fit the problem
  - Each application encourages design of its own language

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Overview-3

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 effectively design the language that best fits the application, rather than adapting the design of the application to fit the rules of a language.

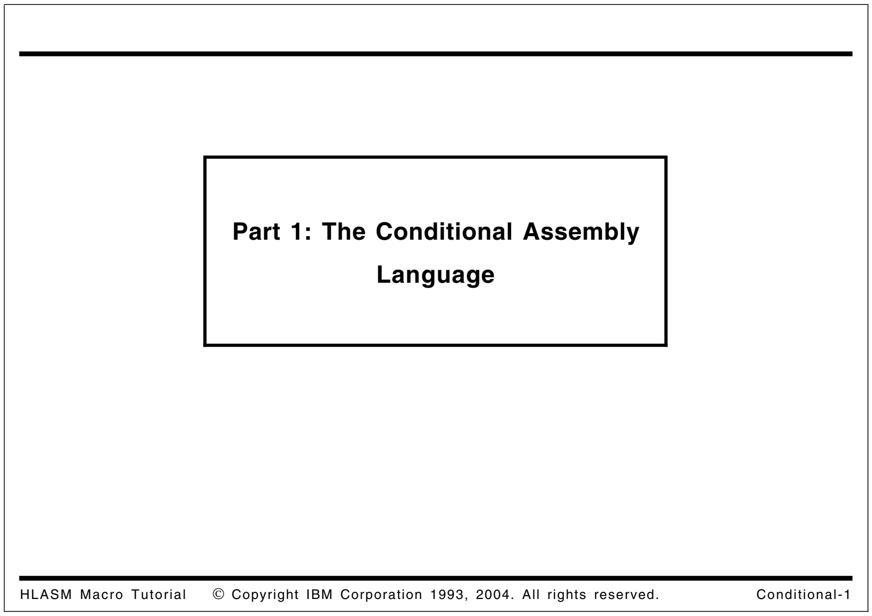
Thus, you can build an application not in the traditional “top-down” sense, but from the “bottom up”. That is, by identifying the common, repeated elements of the application, you can create operations (macros) that reduce your concerns with details, so your program and its language can evolve together. Those common elements can then be used throughout the application (code re-use!).

The approach we will take here is somewhat different from that used in most other tutorials and textbooks where the macro concept is introduced first, and conditional assembly concepts are explained in an ad-hoc, incremental fashion.

Part 1 describes the conditional assembly language and its complete set of facilities and features. Part 2 explores basic aspects of macros and their definition and use in the System/360/370/390 Assembler Language. Part 3 provides “case study” examples of macro programming with IBM High Level Assembler for MVS & VM & 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.

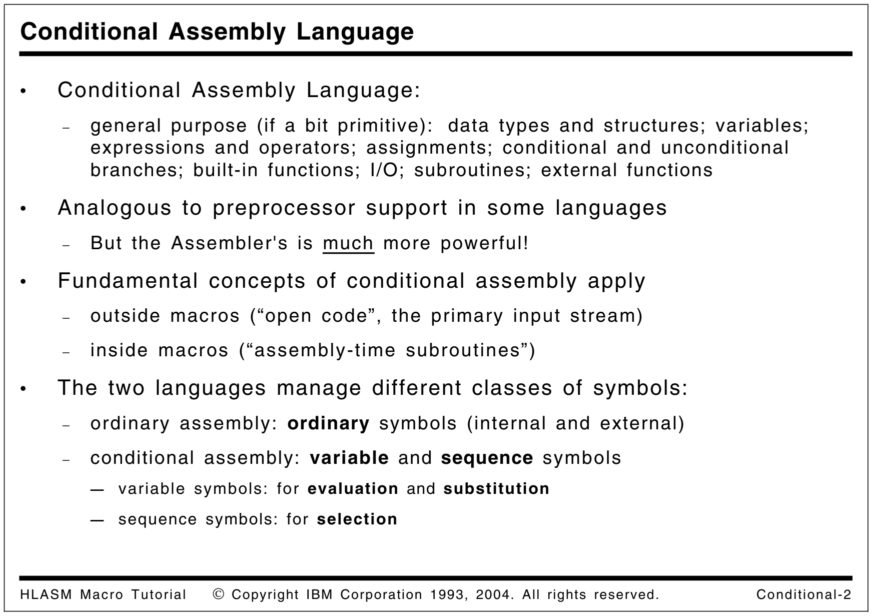
Part 1: The Conditional Assembly Language



Part 1: The Conditional Assembly Language

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-1

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



Conditional Assembly Language

- Conditional Assembly Language:
  - general purpose (if a bit primitive): data types and structures; variables; expressions and operators; assignments; conditional and unconditional branches; built-in functions; I/O; subroutines; external functions
- Analogous to preprocessor support in some languages
  - But the Assembler's is much more powerful!
- Fundamental concepts of conditional assembly apply
  - outside macros (“open code”, the primary input stream)
  - inside macros (“assembly-time subroutines”)
- The two languages manage different classes of symbols:
  - ordinary assembly: **ordinary** symbols (internal and external)
  - conditional assembly: **variable** and **sequence** symbols
    - variable symbols: for **evaluation** and **substitution**
    - sequence symbols: for **selection**

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-2

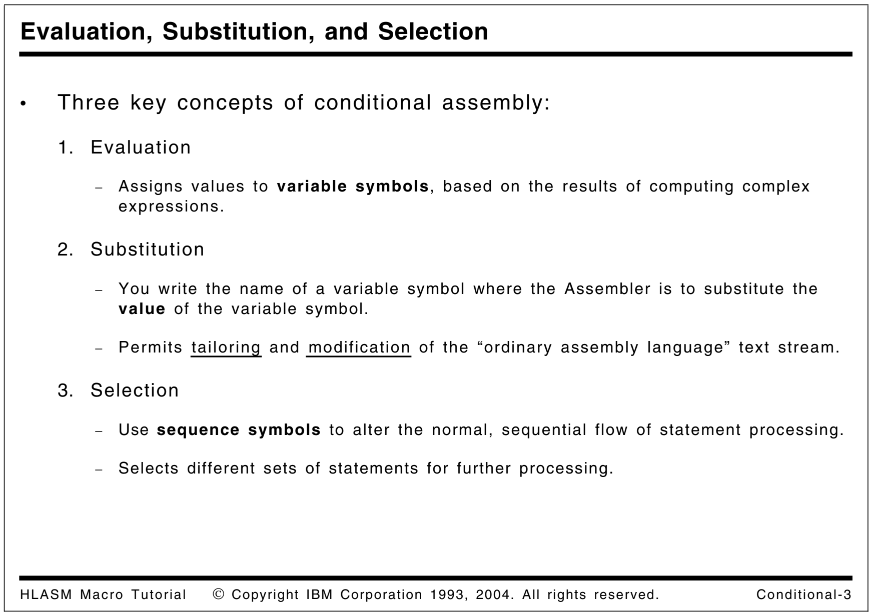
The conditional assembly language is used primarily in macro instructions (or “macros”), which may be thought of as “assembly-time subroutines” invoked during the assembly to perform useful functions. Most of the same techniques can also be used in ordinary assemblies (“open code”, the primary input 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 quite powerful: not only is the conditional assembly language complete (if a bit primitive), but it provides extensive interactions with both the “ordinary” or “base” language and the external assembly environment.

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

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

Just as “normal” or “ordinary” assembly deals with ordinary symbols — assigning values to symbols and using those values to evaluate various kinds of expressions — the conditional assembly language uses variable and sequence symbols. See Figure 118 on page 244 for a comparison of the elements of the ordinary and conditional assembly languages.



Evaluation, Substitution, and Selection

- Three key concepts of conditional assembly:

  1. Evaluation

     – Assigns values to **variable symbols**, based on the results of computing complex expressions.

  2. Substitution

     – You write the name of a variable symbol where the Assembler is to substitute the **value** of the variable symbol.

     – Permits tailoring and modification of the “ordinary assembly language” text stream.

  3. Selection

     – Use **sequence symbols** to alter the normal, sequential flow of statement processing.

     – Selects different sets of statements for further processing.

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-3

Evaluation, Substitution, and Selection

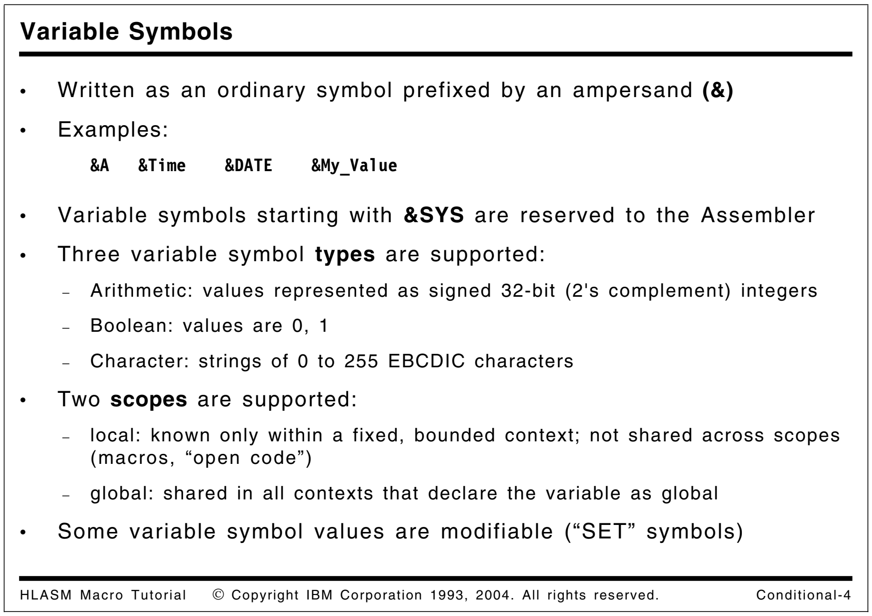
There are three key concepts in the conditional assembly language:

- 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.



Variable Symbols

- Written as an ordinary symbol prefixed by an ampersand **(&)**
- Examples:

```
&A   &Time    &DATE    &My_Value
```

- Variable symbols starting with **&SYS** are reserved to the Assembler
- Three variable symbol **types** are supported:
  - Arithmetic: values represented as signed 32-bit (2's complement) integers
  - Boolean: values are 0, 1
  - Character: strings of 0 to 255 EBCDIC characters
- Two **scopes** are supported:
  - local: known only within a fixed, bounded context; not shared across scopes (macros, “open code”)
  - global: shared in all contexts that declare the variable as global
- Some variable symbol values are modifiable (“SET” symbols)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-4

Variable Symbols

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

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

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

As indicated in these examples, variable symbols may be written in mixed-case characters; all appearances will be treated as being equivalent to their upper-case versions. Variable symbols starting with the characters **&SYS** are called *System variable symbols*, and are reserved to the Assembler. They are described more fully in Appendix B, “System (&SYS) Variable Symbols” on page 224.

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

arithmetic
The allowed values of an arithmetic variable symbol are those of 32-bit (fullword) two's complement integers (i.e., between -2^{31} and $+2^{31}-1$. (Be aware that in certain contexts, their values may be substituted as *unsigned* integers!) (This is discussed further at “Evaluating and Assigning Character Expressions: SETC” on page 22.)

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

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

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 “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* by all scopes that declare the variables 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.

The scope rules for variable symbols are somewhat similar to those of Fortran: some variables are local to each “routine” (main routines are like “open code” and macros are like “subroutines”), and others may be shared in a pool called “blank common”. One key difference is that global variable symbols are shared by *name* in the Assembler Language, whereas they are shared by *offset* in Fortran COMMON.

It is sometimes convenient to distinguish between two types of variable symbols:

1. Symbols whose values you can change:

   these are sometimes called SET symbols, because you use a “SET” statement to assign their values;

2. Symbols whose values you can use, but not change:

   these include system variable symbols and symbolic parameters.

Each of these will be discussed in detail.

Declaring Variable Symbols

- 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
          GBLB  &INIT
          LCLC  &Temp_Chars
```

- 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
```

Declaring Variable Symbols ...

- All explicitly declared variable symbols are SETtable
  - Their values can be changed
- Three forms of **implicit** declaration:
  1. by the Assembler (for **System Variable Symbols**)
     - names always begin with characters **&SYS**
     - most have local scope
  2. by appearing as **symbolic parameters** (dummy arguments) in a macro prototype statement
     - symbolic parameters always have local scope
  3. as local variables, if first appearance is as target of an assignment
     - this is the only implicit form that may be changed (SET)

Declaring Variable Symbols

Variable symbols are declared in several ways:

- 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;
- implicitly by the Assembler (the *System Variable Symbols*, which may not be declared explicitly);
- implicitly, by their appearance as dummy arguments in a macro prototype statement (these are known as *symbolic parameters*; they are of character type, and are local in scope);

- implicitly, as local variables, through appearing for the first time in the name field of a SET statement as the target of an assignment. Their values may be modified in other SET statements.

Explicitly declared variable symbols are declared using two sets of statements that 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 90. For the time being, we will be concerned almost entirely with local variables.

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

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

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

Dimensioned 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 to have 15 elements. We will see in practice that the declared size of an array is ignored, and any valid (positive) subscript value is permitted. Thus, it is sufficient to declare

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

You can determine the maximum subscript used for a subscripted variable symbol with a Number attribute reference (to be discussed later, at “Macro-Instruction Argument Properties: Number Attribute” on page 81). Undimensioned (scalar) variable symbols have number attribute reference value zero (to indicate they are not dimensioned).

Note also that 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 note that 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 (non-subscripted) variable symbol appears.

Created Variable Symbols

Created variable symbols may be created “dynamically”, using characters and the values of other variable symbols. The general form of a created variable symbol is &(**e**), where **e** must (after substitutions) begin with an alphabetic character 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 created from other created variable symbols, to many levels. For example (using some SET statements to be discussed shortly):

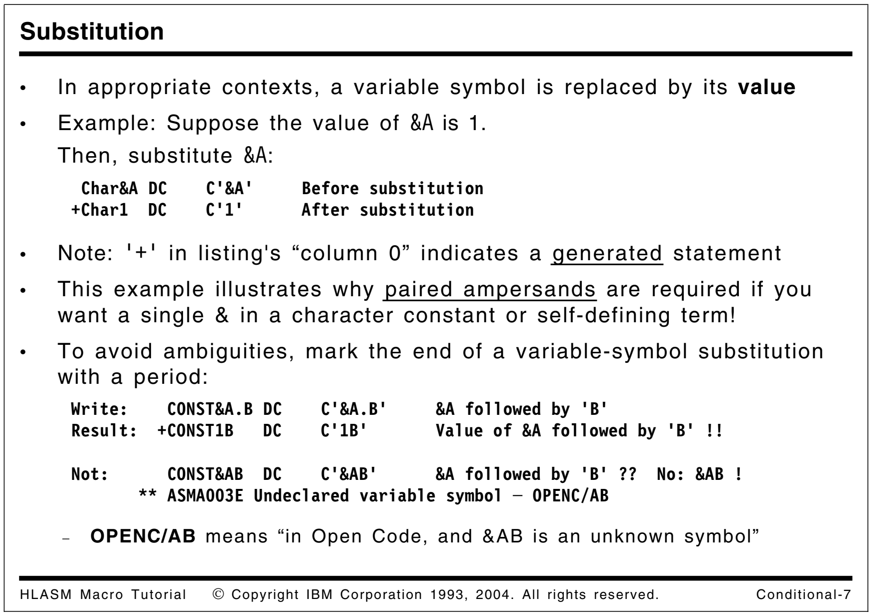
```
&C     SetC  'X'       Variable &X contains the character X
&BX    SetC  'PQ'      Variable &BX contains the character BX
&APQ   SetA  42        Variable &APQ contains the integer 42
```

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” can be quite powerful, and we will use it in several case studies.

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 Appendix B, “System (&SYS) Variable Symbols” on page 224; we will use some of them in later examples.

In the examples that follow, we will typically 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.



Substitution

- In appropriate contexts, a variable symbol is replaced by its **value**
- Example: Suppose the value of &A is 1.

  Then, substitute &A:

  ```
   Char&A DC    C'&A'     Before substitution
  +Char1  DC    C'1'      After substitution
  ```
- Note: '+' in listing's “column 0” indicates a generated statement
- This example illustrates why paired ampersands are required if you want a single & in a character constant or self-defining term!
- To avoid ambiguities, mark the end of a variable-symbol substitution with a period:

  ```
  Write:    CONST&A.B DC    C'&A.B'     &A followed by 'B'
  Result:  +CONST1B   DC    C'1B'       Value of &A followed by 'B' !!

  Not:      CONST&AB  DC    C'&AB'      &A followed by 'B' ??  No: &AB !
         ** ASMA003E Undeclared variable symbol – OPENC/AB
  ```

  - **OPENC/AB** means “in Open Code, and &AB is an unknown symbol”

HLASM Macro Tutorial   © Copyright IBM Corporation 1993, 2004. All rights reserved.   Conditional-7

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.)

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

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

giving

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

Variable symbols are not substituted in remarks fields or in comments statements.

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

Substitution, Evaluation, and Re-Scanning

- Points of substitution identified only by variable symbols
  - HLASM is not a general string- or pattern-matching macro processor
- Statements once scanned for points of substitution are not re-scanned

```
&A     SETC    '3+4'
&B     SETA    5*&A    Is the result 5*(3+4) or (5*3)+4 ??
   ** ASMA102E Arithmetic term is not self—defining term; default = 0
 (Neither! The characters '3+4' are not a self=defining term!)
```

- Substitutions cannot create points of substitution
- Another example (the SETC syntax and the &&s are explained later):

```
&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?
     ** ASMA127S Illegal use of Ampersand
```

The operand is '&B', so the statement gets a diagnostic

HLASM Macro Tutorial Conditional-8

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 substituted.

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

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, 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

&N   SetC  'N'          Set &N to the character N
&C   SetA  5*&N         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 22.)

```
&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?
```

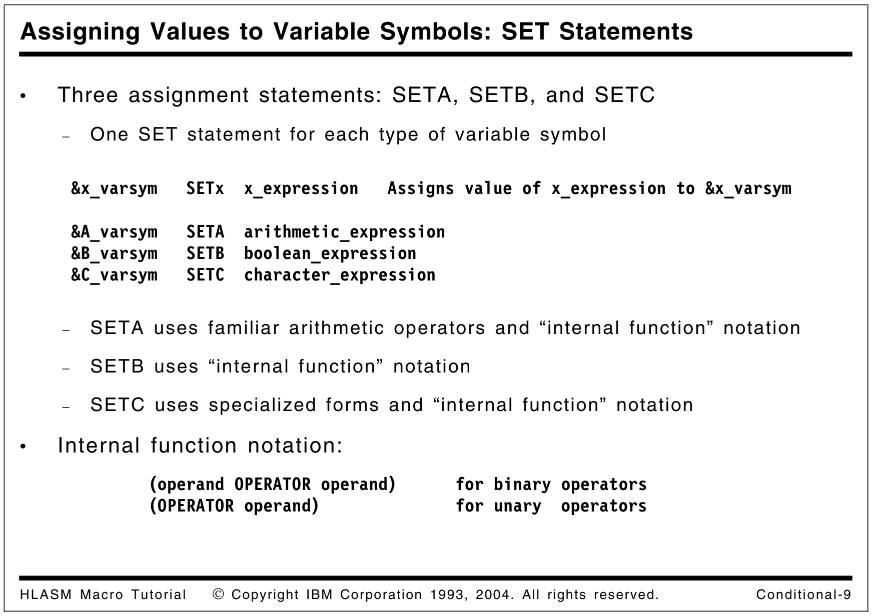
The answer is “no”. In fact, this DC statement results in an error message:

```
    ** 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 185 that there are some ways around this problem.)

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

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



Assigning Values to Variable Symbols: SET Statements

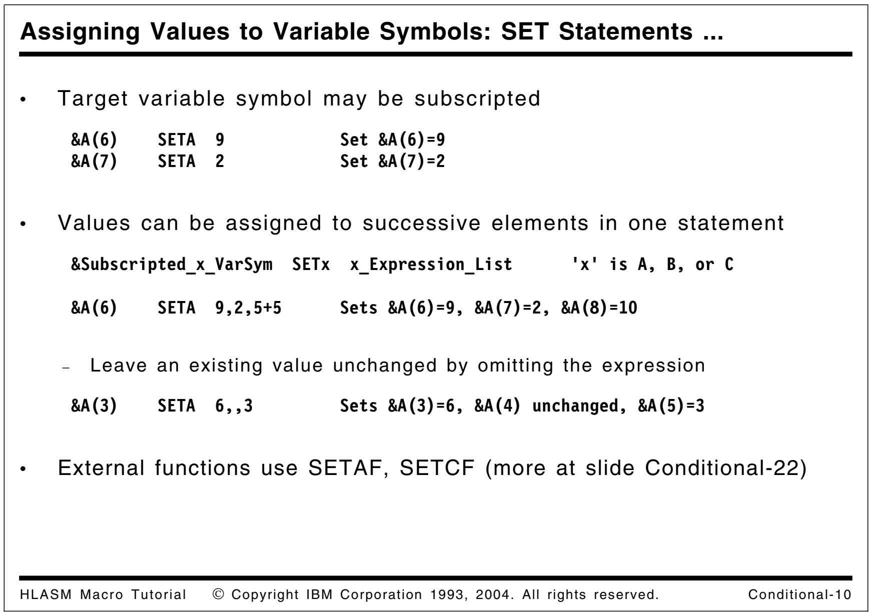
- Three assignment statements: SETA, SETB, and SETC
  - One SET statement for each type of variable symbol

```
&x_varsym   SETx  x_expression   Assigns value of x_expression to &x_varsym

&A_varsym   SETA  arithmetic_expression
&B_varsym   SETB  boolean_expression
&C_varsym   SETC  character_expression
```

  - SETA uses familiar arithmetic operators and “internal function” notation
  - SETB uses “internal function” notation
  - SETC uses specialized forms and “internal function” notation
- Internal function notation:

```
        (operand OPERATOR operand)      for binary operators
        (OPERATOR operand)              for unary  operators
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-9



Assigning Values to Variable Symbols: SET Statements ...

- Target variable symbol may be subscripted

```
&A(6)    SETA  9           Set &A(6)=9
&A(7)    SETA  2           Set &A(7)=2
```

- Values can be assigned to successive elements in one statement

```
&Subscripted_x_VarSym  SETx  x_Expression_List      'x' is A, B, or C

&A(6)    SETA  9,2,5+5      Sets &A(6)=9, &A(7)=2, &A(8)=10
```

  - Leave an existing value unchanged by omitting the expression

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

- External functions use SETAF, SETCF (more at slide Conditional-22)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-10

Assigning Values to Variable Symbols: SET Statements

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

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

- Symbolic parameters have their values assigned by appearing as actual arguments in a macro call statement. They may not appear in the name field of a SETx statement.

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

In addition to the usual arithmetic operators such as +, −, *, and /, conditional assembly expressions may specify unary and binary operators in an "internal function" notation. Rather than the function-call format used in many high-level languages (such as function(arg)), and rather than introduce complex combinations of special characters (such as // or << or &&), the Assembler Language recognizes certain operators in a parenthesized format:

```
(operand OPERATOR operand)      for binary operators
(OPERATOR operand)              for unary  operators
```

We will see examples of this "internal function" notation shortly.[1] External arithmetic functions are invoked by the SETAF command, and are described at "External Conditional-Assembly Functions" on page 32.

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

```
&A(6)    SETA  9,2,5+5      Sets &A(6)=9, &A(7)=2, &A(8)=10
```

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

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

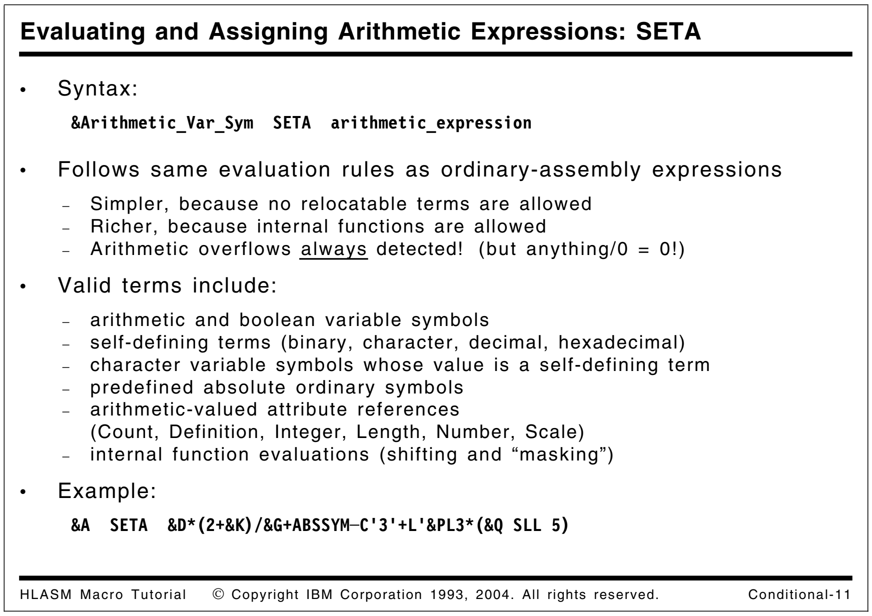
Occasionally, the three declarable types of variable symbol (arithmetic, boolean, and character) are referred to as SETA, SETB, and SETC variables, respectively, and declarable variable symbols are referred to as SET symbols.

Evaluating Conditional-Assembly Expressions

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

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

[1] One of the nice things about internal function notation is that spaces can be used within the parentheses to make statement formatting more readable.



Evaluating and Assigning Arithmetic Expressions: SETA

- Syntax:

```
&Arithmetic_Var_Sym  SETA  arithmetic_expression
```

- Follows same evaluation rules as ordinary-assembly expressions
  - Simpler, because no relocatable terms are allowed
  - Richer, because internal functions are allowed
  - Arithmetic overflows always detected! (but anything/0 = 0!)
- Valid terms include:
  - arithmetic and boolean variable symbols
  - self-defining terms (binary, character, decimal, hexadecimal)
  - character variable symbols whose value is a self-defining term
  - predefined absolute ordinary symbols
  - arithmetic-valued attribute references (Count, Definition, Integer, Length, Number, Scale)
  - internal function evaluations (shifting and “masking”)
- Example:

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-11

Evaluating and Assigning Arithmetic Expressions: SETA

The rules for evaluating conditional-assembly arithmetic expressions are very similar to those for ordinary expressions, with the added great simplification that none of the terms in a conditional-assembly expression may be relocatable. 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 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.

```
&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
```

Overflows are detected and diagnosed:

- addition and subtraction overflow returns 0
- multiplication overflow returns 1
- division overflow (−2147483648/−1) returns 0
- division by zero (including 0/0) returns 0.

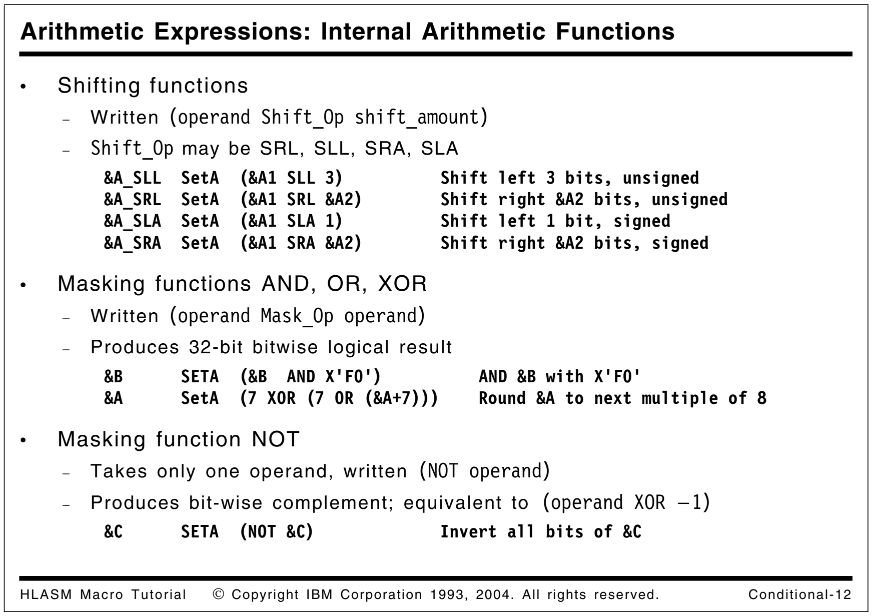
Arithmetic-valued attribute references to ordinary symbols may also be used as terms; these are normally attribute references to character variable symbols whose value is an ordinary symbol. The arithmetic-valued attribute references are:

- 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 76. Attribute references may, of course, be used in “open code”.



Arithmetic Expressions: Internal Arithmetic Functions

- Shifting functions
  - Written (operand Shift_Op shift_amount)
  - Shift_Op may be SRL, SLL, SRA, SLA

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

- Masking functions AND, OR, XOR
  - Written (operand Mask_Op operand)
  - Produces 32-bit bitwise logical result

```
&B      SETA  (&B  AND X'F0')         AND &B with X'F0'
&A      SetA  (7 XOR (7 OR (&A+7)))    Round &A to next multiple of 8
```

- Masking function NOT
  - Takes only one operand, written (NOT operand)
  - Produces bit-wise complement; equivalent to (operand XOR −1)

```
&C      SETA  (NOT &C)            Invert all bits of &C
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-12

Arithmetic Expressions: Internal Arithmetic Functions

Internal function notation may be used in evaluating conditional assembly arithmetic expressions to simplify operations that would otherwise require elaborate or obscure coding. The two classes of internal arithmetic functions are shifting and “masking” operations:

- Shifting operations are written in the form

  ```
            (operand  Shift_Operator  shift_amount)
  ```

  where the Shift_Operator may be one of SLA, SLL, SRA, or SRL. The operand to be shifted may be any arithmetic term, and the shift_amount is an arithmetic term. The actual amount of the shift is determined from the rightmost six bits of the shift_amount, exactly as in the identically named machine instructions.

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

  Arithmetic Overflow is detected for addition, subtraction, multiplication, division, and the SLA operation.

- Masking operations are written in the forms

  ```
            (operand1  Masking_Operator  operand2)
  or
            (NOT  operand)
  ```

  where the Masking_Operator may be one of AND, OR, or XOR. These operators act between the 32-bit operands as bit-wise operations, producing a 32-bit result. The operations are exactly equivalent to the hardware instructions NR, OR, and XR.

NOT is a unary operator, and inverts each bit of its operand to produce the 32-bit one's complement. (NOT operand) is equivalent to (operand XOR −1).

```
  &A_And  SetA  ((&A1 AND &A2) AND X'FF') Low-order 8 bits
  &A_Or   SetA  (&A1 OR (&A2 OR &A3))     Or of 3 variables
  &A_Xor  SetA  (&A1 XOR (&A3 XOR 7))     XOR of 7, 2 variables
  &A_Not  SetA  (NOT &A1)+&A2             Complement and add
```

Suppose you wish to “round up” the value of &A to a multiple of 8 (if it is not already a multiple. Using “old code”, you might have written:

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

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

```
&A       SetA  (7 XOR (7 OR (&A+7)))     Round &A to next multiple of 8
         or
&A       SetA  (&A+7 AND -8)             Round &A to next multiple of 8
```

For example, we might execute the following statement:

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

The value assigned to &A is evaluated as follows:

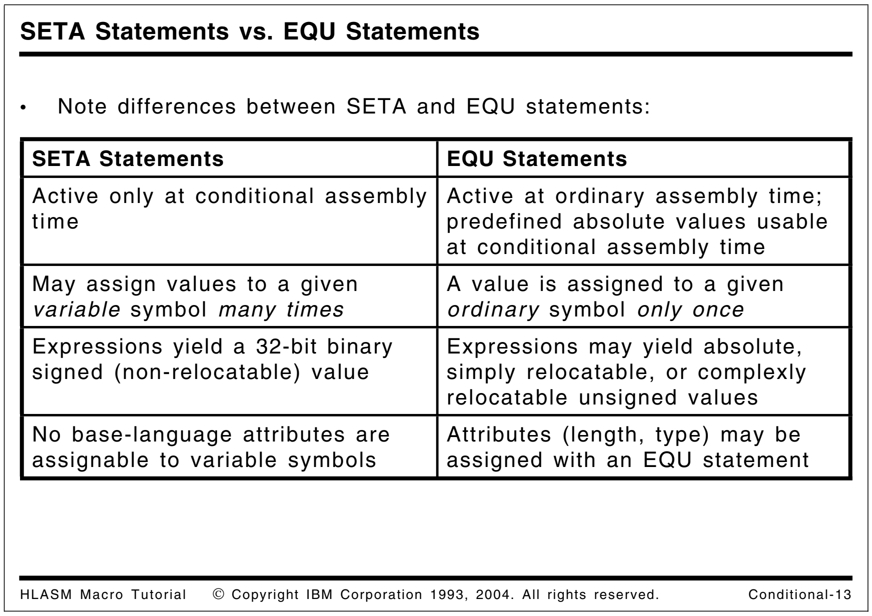
1. multiply the value of &D by the value of (2+&K)
2. divide the result by &G
3. to that result, add the value of the symbol ABSSYM and subtract the character self-defining term C'3'
4. evaluate the product of the length attribute of the symbol PL3 and the value of &Q shifted left logically 5 bit positions,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

These functions can be used in places where the previously available capabilities of the conditional assembly language led to clumsy constructions. Because the conditional assembly language is interpreted by the assembler, there will not always be significant performance gains in using these new arithmetic operators. However, any simpler expression 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)
```



SETA Statements vs. EQU Statements

- Note 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 (non-relocatable) value	Expressions may yield absolute, simply relocatable, or complexly relocatable unsigned values
No base-language attributes are assignable to variable symbols	Attributes (length, type) may be assigned with an EQU statement

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-13

SETA Statements vs. EQU Statements

It may be helpful at this point to identify some of the differences between the results of SETA and EQU statements. The following table compares some key factors:

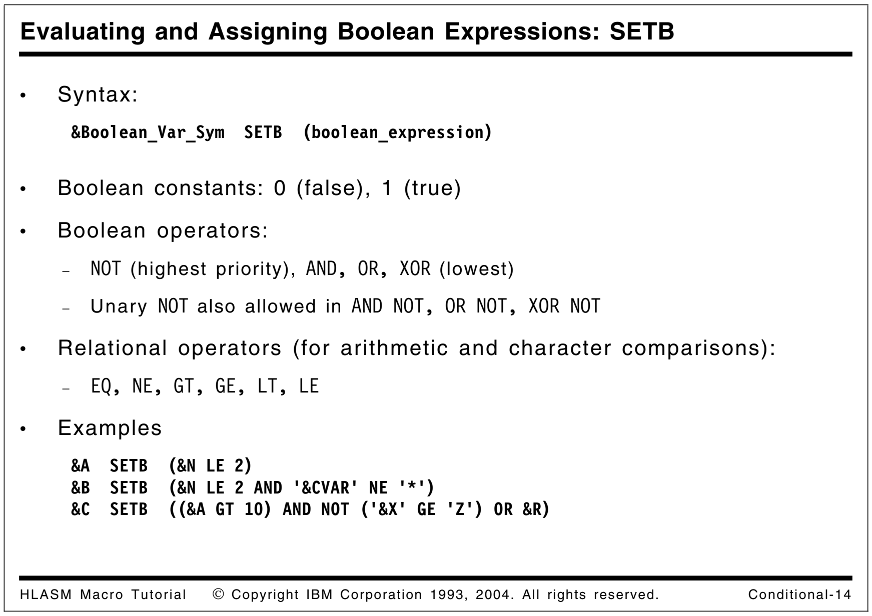
SETA Statements	EQU Statements
Active only at conditional assembly time	Active at ordinary assembly time; predefined absolute values may be usable at conditional assembly time
May assign values to a given *variable* symbol *many times*	A value is assigned to a given *ordinary* symbol *only once*
Expressions yield a 32-bit binary signed (non-relocatable) value	Expressions may yield an absolute, simply relocatable, or complexly relocatable unsigned values
No base-language attributes are assignable to variable symbols	Attributes (length, type) 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 Appendix D, “Ordinary and Conditional Assembly” on page 244.

High Level Assembler permits one 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

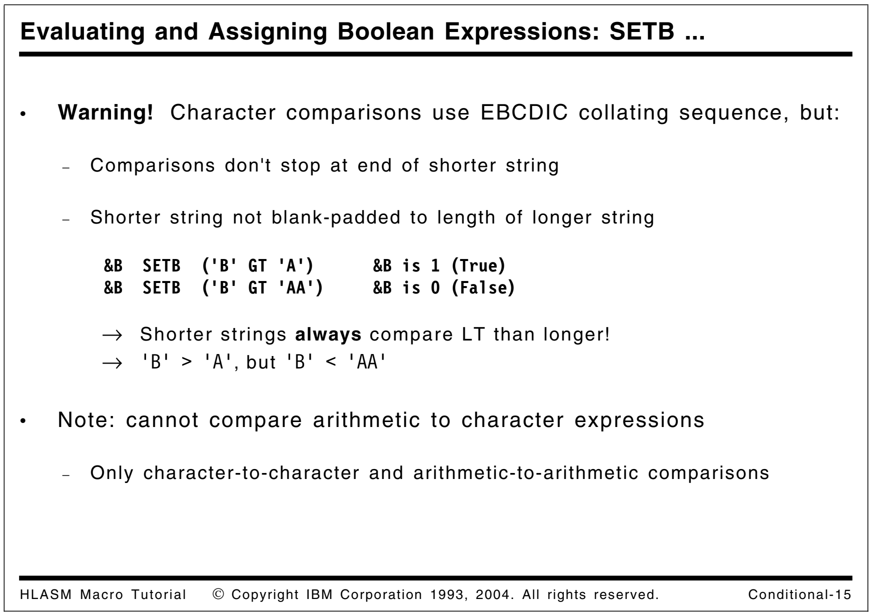
- Syntax:

```
&Boolean_Var_Sym  SETB  (boolean_expression)
```

- Boolean constants: 0 (false), 1 (true)
- Boolean operators:
  - NOT (highest priority), AND, OR, XOR (lowest)
  - Unary NOT also allowed in AND NOT, OR NOT, XOR NOT
- Relational operators (for arithmetic and character comparisons):
  - EQ, NE, GT, GE, LT, LE
- Examples

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-14



Evaluating and Assigning Boolean Expressions: SETB ...

- **Warning!** Character comparisons use EBCDIC collating sequence, but:
  - Comparisons don't stop at end of shorter string
  - Shorter string not blank-padded to length of longer string

```
&B  SETB  ('B' GT 'A')      &B is 1 (True)
&B  SETB  ('B' GT 'AA')     &B is 0 (False)
```

→ Shorter strings **always** compare LT than longer!
→ 'B' > 'A', but 'B' < 'AA'

- Note: cannot compare arithmetic to character expressions
  - Only character-to-character and arithmetic-to-arithmetic comparisons

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-15

Evaluating and Assigning Boolean Expressions: SETB

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

Boolean primaries include boolean variable symbols, the boolean constants 0 and 1, and (most useful) comparisons. 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)
```

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

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 22 below). Comparisons between arithmetic and character terms is not allowed.

The comparison operators are

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

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

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

For character comparisons, a test is first made on the *lengths* of the two comparands: if they are not the same length, the shorter operand is *always* taken to be “less than” the longer. *Note that this may* ***not*** *be what you would get if you did a “hardware” comparison!* (The shorter string is not padded, nor is the comparison done using the shorter string's length.)

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
```

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.

Some examples of SetB statements are:

```
&A     SetB  (&V gt 0 AND &V le 7)          True if &V between 1 and 7
&B     SetB  ('&C' lt '0' OR '&C' gt '9')   True if &C not a digit
&Z     SetB  (&A AND NOT &B)                True if &A true, &B false
&S     SetB  (&B XOR (&G OR &D))
&T     SetB  (&X ge 5 XOR (&Y*2 lt &X OR &D))
```

Evaluating and Assigning Character Expressions: SETC

- Syntax:

```
&Character_Var_Sym  SETC  character_expression
```

- A character constant is a 'quoted string' 0 to 255 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
```

- All terms must be quoted, except type-attribute references (and opcode-attribute references)
  - Type-attribute references are neither quoted nor duplicated nor combined

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

- 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 ****
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-16

Evaluating and Assigning Character Expressions: SETC ...

- Apostrophes and ampersands in strings must be paired
  - 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 substring notation to get a single & (see slide Conditional-19)
- Warning! SETA variables are substituted **without** sign!

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

  - The SIGNED built-in function avoids this problem

```
&C  SETC  (SIGNED &A)   &C has value –5
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-17

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
```

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.

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.)

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

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

&Amp  SETC  '&&'        &Amp has value &&
&Yes  SetB  ('&Amp' eq '&&')  &Yes is 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 a substring (described below) of a pair of ampersands should be used.

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

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

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

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

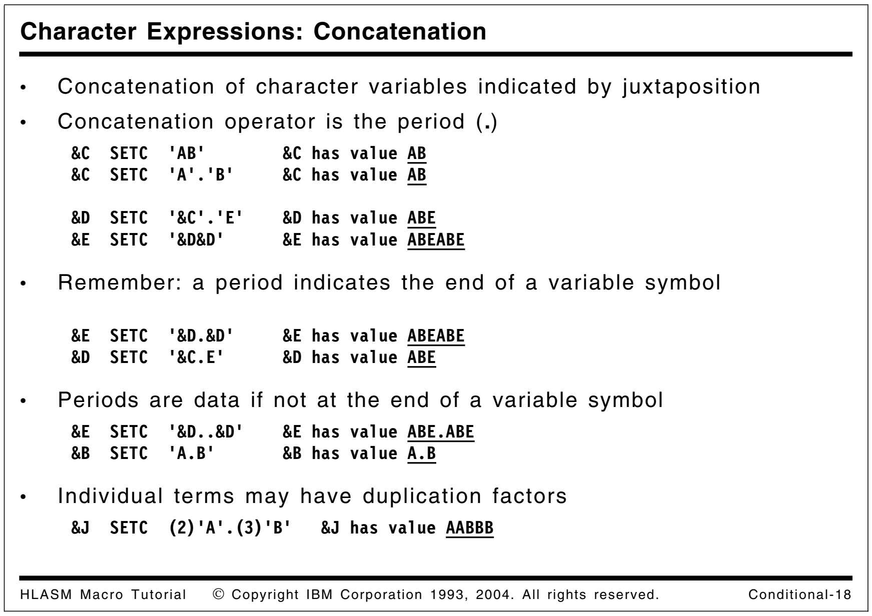
```
&A  SETA  –5
    DC    F'&A'  Generates X'00000005'
&C  SETC  '&A'   &C has value 5 (not -5!)

&B  SETA  X'80000000'     (maximum negative number)
&D  SETC  '&B'   &D has value 2147483648 (!)
&E  SETA  &D     Error! (“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 with the proper sign, then you must either use the SIGNED built-in function (see “Character Expressions: Internal Character Functions” on page 29 for further details), or construct a character variable with the desired sign, as in the following example. (Uses of the AIF and ANOP statements, and the sequence symbol .GenCon will be discussed shortly.)

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

Note: In Release 4 of High Level Assembler, support for predefined absolute symbols in character expressions was removed, because there are some possible ambiguities in how their values should be interpreted.

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



Character Expressions: Concatenation

- Concatenation of character variables indicated by juxtaposition
- Concatenation operator is the period (.)

```
&C  SETC  'AB'       &C has value AB
&C  SETC  'A'.'B'    &C has value AB

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

- Remember: a period indicates the end of a variable symbol

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

- Periods are data if not at the end of a variable symbol

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

- Individual terms may have duplication factors

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-18

String Concatenation

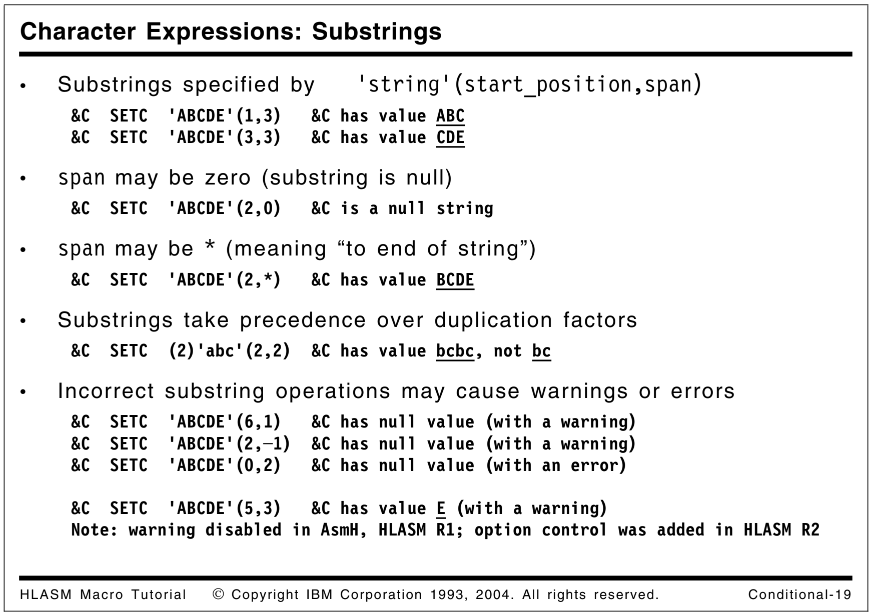
We are somewhat familiar with the notion of string concatenation from some of the earlier examples of substitution, where a substituted value is concatenated with the adjoining characters to create the completed string of characters. As before, the end of a variable symbol may be denoted with a period.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

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

&E  SETC  '&D..&D'   &E 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 desired concatenation results.



Character Expressions: Substrings

- Substrings specified by `'string'(start_position,span)`

```
&C  SETC  'ABCDE'(1,3)   &C has value ABC
&C  SETC  'ABCDE'(3,3)   &C has value CDE
```

- span may be zero (substring is null)

```
&C  SETC  'ABCDE'(2,0)   &C is a null string
```

- span may be * (meaning “to end of string”)

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

- Substrings take precedence over duplication factors

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

- Incorrect substring operations may cause warnings or errors

```
&C  SETC  'ABCDE'(6,1)   &C has null value (with a warning)
&C  SETC  'ABCDE'(2,−1)  &C has null value (with a warning)
&C  SETC  'ABCDE'(0,2)   &C has null value (with an error)

&C  SETC  'ABCDE'(5,3)   &C has value E (with a warning)
Note: warning disabled in AsmH, HLASM R1; option control was added in HLASM R2
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-19

Substrings

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

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

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

To illustrate, consider the following examples:

```
&C  SETC  'ABCDE'(1,3)   &C has value ABC
&C  SETC  'ABCDE'(3,3)   &C has value CDE
&C  SETC  'ABCDE'(5,3)   &C has value E (with a warning, if FLAG(SUBSTR) was specified)
```

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

1. The *length* of the *source_string* must be between 1 and 255.

2. The *span* of the *substring* must be between 0 and 255.

3. If 1≤ *start_position*≤ *length*, and 1≤ *span*≤ *length*, and *start_position* + *span*≤ *length*+1, then a normal substring will be extracted.

4. If *start_position*≤ 0, then the assembler will issue an error message, and the substring will be set to null.

5. If *start_position*>*length*, then the assembler will issue a warning message, and the substring will be set to null.

6. If *span*=0, then the substring will be set to null. No error message will be issued.

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

8. If *start_position* + *span*>*length*+1, then the substring will be that portion of the *source_string* starting at *start_position* to the end. The assembler will issue a warning message. (Note: This warning was documented but disabled in Assembler H and High Level Assembler Release 1; it is now controlled by the FLAG(NOSUBSTR) option.)

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 substringing. One may 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 (which are discussed in “Character Expressions: Internal Character Functions” 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: There is occasional confusion of 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)
```

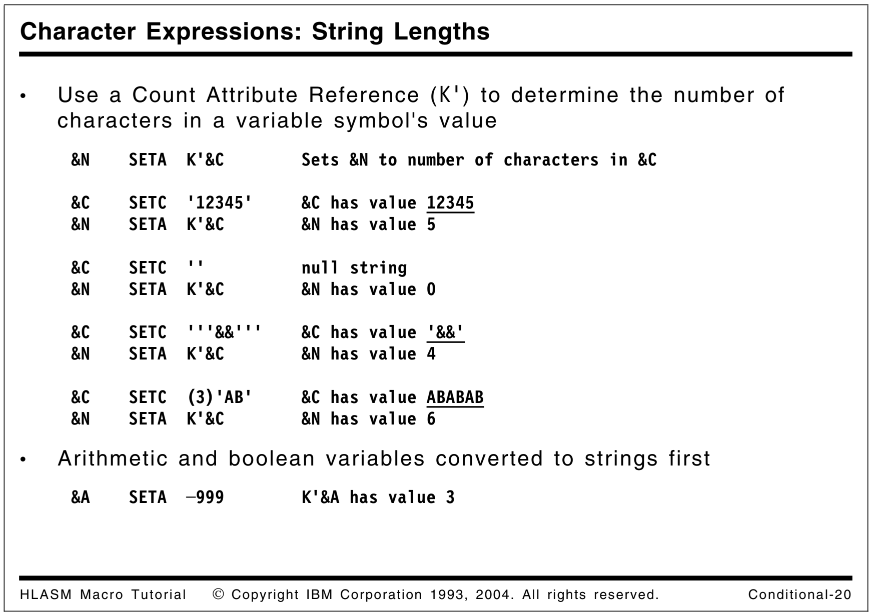
whereas subscripts appear inside the quotes:

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

They may of course appear together:

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

Subscripted variable symbols were discussed at “Dimensioned Variable Symbols” on page 9.



Character Expressions: String Lengths

- 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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-20

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 has value &&
&M    SETA  K'&D       &M has value 4
```

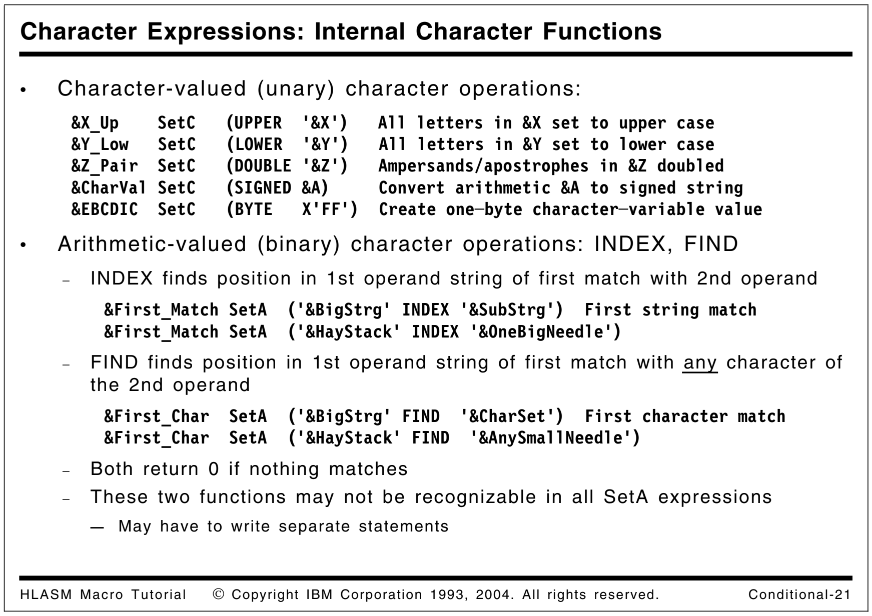
The Count attribute reference is very useful in cases where strings must be scanned from right to left; thus,

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

assigns the rightmost character in the value 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
```



Character Expressions: Internal Character Functions

- Character-valued (unary) character operations:

```
&X_Up    SetC   (UPPER  '&X')   All letters in &X set to upper case
&Y_Low   SetC   (LOWER  '&Y')   All letters in &Y set to lower case
&Z_Pair  SetC   (DOUBLE '&Z')   Ampersands/apostrophes in &Z doubled
&CharVal SetC   (SIGNED &A)     Convert arithmetic &A to signed string
&EBCDIC  SetC   (BYTE   X'FF')  Create one—byte character—variable value
```

- Arithmetic-valued (binary) character operations: INDEX, FIND
  - INDEX finds position in 1st operand string of first match with 2nd operand

```
&First_Match SetA  ('&BigStrg' INDEX '&SubStrg')  First string match
&First_Match SetA  ('&HayStack' INDEX '&OneBigNeedle')
```

  - FIND finds position in 1st operand string of first match with any character of the 2nd operand

```
&First_Char  SetA  ('&BigStrg' FIND  '&CharSet')  First character match
&First_Char  SetA  ('&HayStack' FIND  '&AnySmallNeedle')
```

  - Both return 0 if nothing matches
  - These two functions may not be recognizable in all SetA expressions
    - — May have to write separate statements

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-21

Character Expressions: Internal Character Functions

The assembler supports two types of internal character-string functions: character-valued and arithmetic-valued.

- The five character-valued character functions UPPER, LOWER, DOUBLE, SIGNED, and BYTE are unary operators.
  - The UPPER function operates on a string of characters and produces a string in which all lower-case letters (having EBCDIC representations X'81-89', X'91-99', and X'A2-A9' respectively) are converted to their upper-case equivalents (having EBCDIC representations X'C1-C9', X'D1-D9', and X'E2-E9' respectively).

```
&X_Up    SetC   (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.

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

  - The DOUBLE function scans its operand string for occurrences of ampersands and apostrophes (single quotes), and replaces each occurrence with a pair. This allows the result to be directly substituted into a DC-statement character constant (or a character literal). For example,

```
&Z       SetC   '&&'''         Value is &&'
&Z_Pair  SetC   (DOUBLE '&Z')  Ampersands/apostrophes in &Z doubled (&&&&'')
Z_Const  DC     C'&Z_Pair'     Constant is &&'
```

  - The SIGNED function eliminates the need for special coding to create a properly signed character-string representation of arithmetic values. (Remember that assigning an arithmetic variable in a SetC statement to a character variable produces only the unsigned magnitude of the arithmetic value!)

```
&X       SetA   -10            &X contains -10
&Y       SetC   '&X'           &Y contains '10' !
&Z       SetC   (SIGNED -10)   &Z contains '-10'
```

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

    ```
    &X00     SetC   (BYTE 0)       &X00 contains bit pattern X'00'
    &XFF     SetC   (BYTE X'FF')   &XFF contains bit pattern X'FF'
    ```

    Such assignments were extremely difficult or impossible to achieve in previous assemblers.

  Each of these five character-valued functions may be preceded by a duplication factor, and may be concatenated using the '.' operator. For example:

  ```
  &C1  SetC  (Upper 'a').(Lower 'B')        &C1 = 'Ab'
  &C2  SetC  (2)(Upper 'a').(Lower 'B')     &C2 = 'AAb'
  &C3  SetC  (Upper 'a').(2)(Lower 'B')     &C3 = 'Abb'
  &c4  setc  (2)(upper 'a').(2)(lower 'B')  &C4 = 'AAbb'
  ```

- The two arithmetic-valued character functions INDEX and FIND are used for rapid string scanning. Both are binary operators.

  - The INDEX function finds the position within the first operand string of the first occurrence of a match with the second operand string:

    ```
    &First_Match SetA  ('&BigStrg' INDEX '&SubStrg')  First string match
    ```

    sets &First_Match to the position within &BigStrg of the first substring that matches &SubStrg. If no match is found, the INDEX function returns a zero value.

    ```
    &First_Match SetA  ('&BigStrg' INDEX '&SubStrg')  First string match

    &Found     SetA  ('ABCdefg' Index 'de')   &Found has value 4
    &NotFound  SetA  ('ABCdefg' Index 'DE')   &NotFound has value 0
    ```

    The INDEX function can greatly simplify searches for a match in a list of strings. For example, suppose the character variable symbol &Response might contain one of four values: YES, NO, MAYBE, and NONE, and we wish to set the arithmetic variable symbol &RVal to 1, 2, 3, or 4 respectively (or to zero if no match is found). In the past, you might have written statements like these:

    ```
    &RVal    SetA  0
    .A1      AIf   ('&Response' ne 'YES').A2
    &RVal    SetA  1
             AGo   .B
    .A2      AIf   ('&Response' ne 'NO').A3
    &RVal    SetA  2
             AGo   .B
             - - -     etc.
    .B       ANop
    ```

    Each alternative is tested in turn until a match is found, and the desired value is then set. Alternatively, you might have searched a list of subscripted variable symbols:

    ```
    &OK(1)   SetC  'YES','NO','MAYBE','NONE'  Initialize valid matches
    &RVal    SetA  0                         Initialize match value
    &J       SetA  0                         Initialize count
    .Test    AIf   (&J ge N'&OK).Done        Check for all values tested
    &J       SetA  &J+1                      Increment test value
             AIf   ('&Response' ne '&OK(&J)').Test  Loop if not found
    &RVal    SetA  &J                        Set index of matched value
    .Done    ANop
    ```

    Using the INDEX function, the looping can be eliminated and the search for a match can be done in a single statement:

```
&OK      SetC  'YES  NO   MAYBENONE'      5 positions per term
&RVal    SetA  ('&OK' Index '&Response')  Search for match
&RVal    SetA  1+&RVal/5                  Set corrected result
```

- The FIND function finds the position within the first operand string of the first occurrence of a match with *any* character of the second operand string:

  ```
  &First_Char  SetA  ('&BigStrg' FIND  '&CharSet')  First char match
  ```

  sets &First_Char to the position within &BigStrg of the first character that matches any single character of the &CharSet. If no matching character is found, the FIND function returns a zero value. For example, suppose you want to search an “expression string” for the presence of the arithmetic operators +, –, *, and /. Without the FIND function, you might have written a code fragment like this:

  ```
  .Scan    ANop
  &C       SetC  '&String'(&J,1)         Pick off &J'th character
           AIf   ('&C' eq '+').Plus      Branch if plus
           AIf   ('&C' eq '-').Minus     Branch if minus
           AIf   ('&C' eq '*').Mult      Branch if asterisk
           AIf   ('&C' eq '/').Div       Branch if slash
  &J       SetA  &J+1                    Increment &J
           AIf   (&J le K'&String).Scan  Try again
  .NoChar  ANop                          No match found
           - - -
  ```

  Note that *every* character must be tested inside the loop! With the FIND function, the scanning can be done more simply, and the “selection branch” to handle the desired characters is done only when such a character has been found:

  ```
  &OpPosn  SetA  ('&String' Find '+-*/') Search for operator character
           AIf   (&OpPosn eq 0).NoChar   Skip if no match found
           AGo   (&OpPosn).Plus,.Minus,.Mult,.Div   Branch accordingly
           - - -  etc.
  ```

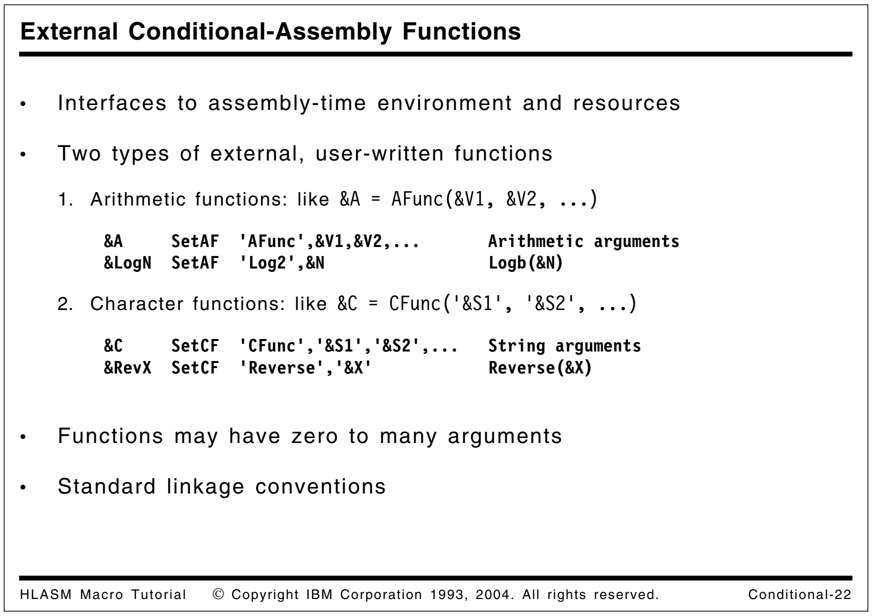
Note that it might not be possible in all cases to use character functions in arithmetic expressions. For example, you might want to write

```
         AIf   (0 ne ((UPPER '&SysParm') Index 'YES')).YESOK
```

but you might in fact have to write three statements:

```
&TempC   SetC  (UPPER '&SysParm')
&TempA   SetA  ('&TempC' Index 'YES')
         AIf   (0 ne &TempA).YESOK
```

You should verify that your copy of HLASM is capable of evaluating complex expressions before writing lots of conditional assembly code involving mixtures of character functions and arithmetic expressions.



External Conditional-Assembly Functions

- Interfaces to assembly-time environment and resources
- Two types of external, user-written functions

1. Arithmetic functions: like &A = AFunc(&V1, &V2, ...)

```
&A     SetAF  'AFunc',&V1,&V2,...       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 may have zero to many arguments
- Standard linkage conventions

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-22

External Conditional-Assembly Functions

IBM High Level Assembler for MVS & VM & VSE supports a powerful and flexible capability for invoking externally-defined functions during the assembly. These “conditional-assembly functions” can perform almost 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 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 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 require no parameters, others may require several. The type of each parameter is the same as that of the target variable: arithmetic parameters for SETAF, and character parameters for SETCF.

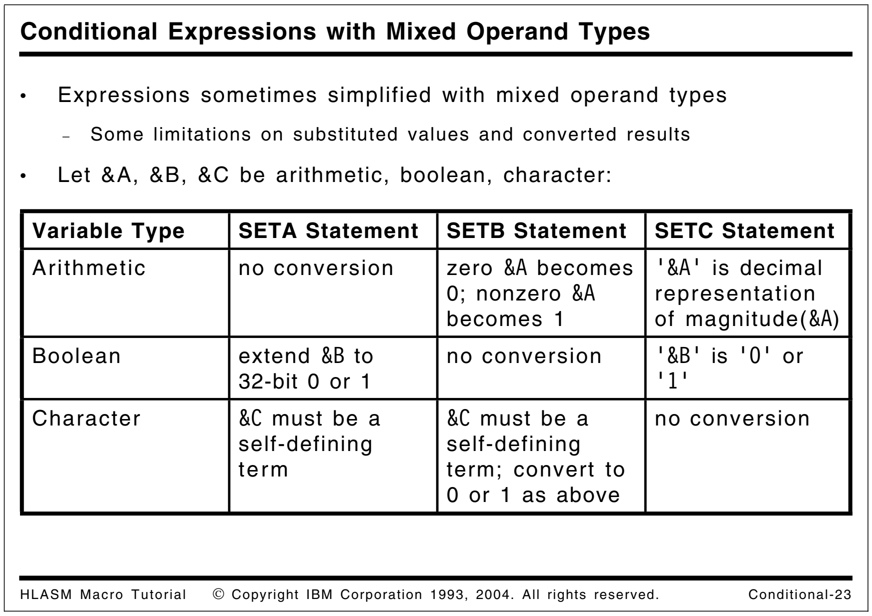
A compact notational representation of this description is

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

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

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

Interface descriptions and sample code for these two functions is described in Appendix A, “External Conditional Assembly Functions” on page 209. Details of external function interfaces are described in the *IBM High Level Assembler for MVS & VM & VSE* product publications.



Conditional Expressions with Mixed Operand Types

- 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-23

Conditional Expressions with Mixed Operand Types

Conditional assembly expressions can be sometimes be simplified if mixed operand types are used, to avoid a need for additional statements for converting to the desired type. The following table indicates the allowed combinations of SETx statement types and operands; the variables &A, &B, and &C respectively represent arithmetic, boolean, and character variable symbols.

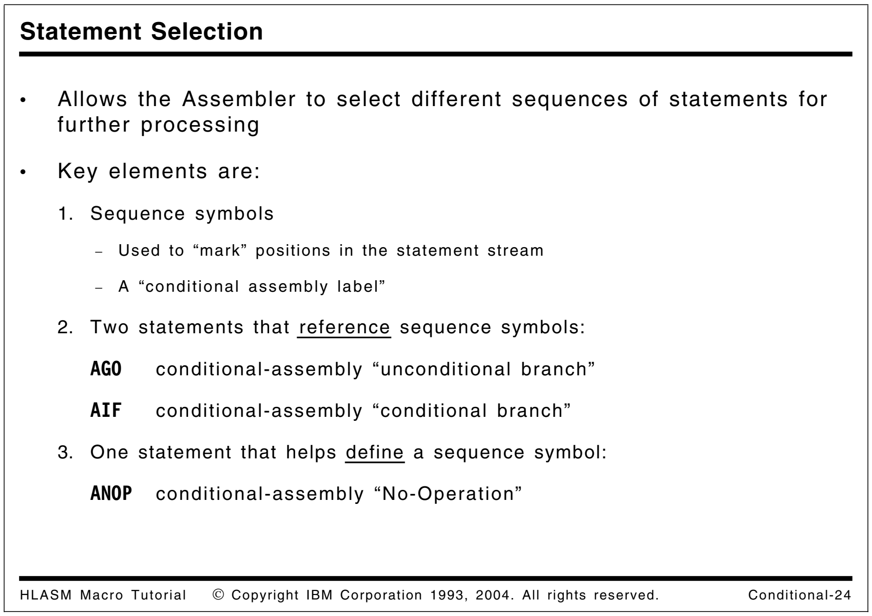
Variable Type	SETA Statement	SETB Statement	SETC Statement
Arithmetic	no conversion	zero &A becomes 0; nonzero &A becomes 1	'&A' is decimal representation of magnitude of &A
Boolean	extend &B to 32-bit 0 or 1	no conversion	'&B' is '0' or '1'
Character	&C must represent a self-defining term	&C must represent 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 does not mean that the arithmetic value is treated as an unsigned 32-bit quantity, but that the *magnitude* of the arithmetic term is converted to decimal! (Further details are given below.)
- Character values substituted into arithmetic expressions must be self-defining decimal, hexadecimal, boolean, or character self-defining terms.

- Character values substituted into boolean expressions must be self-defining decimal, hexadecimal, boolean, or character self-defining terms, which are then converted to 0 or 1 following the first case above.



Statement Selection

- Allows the Assembler to 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 helps define a sequence symbol:

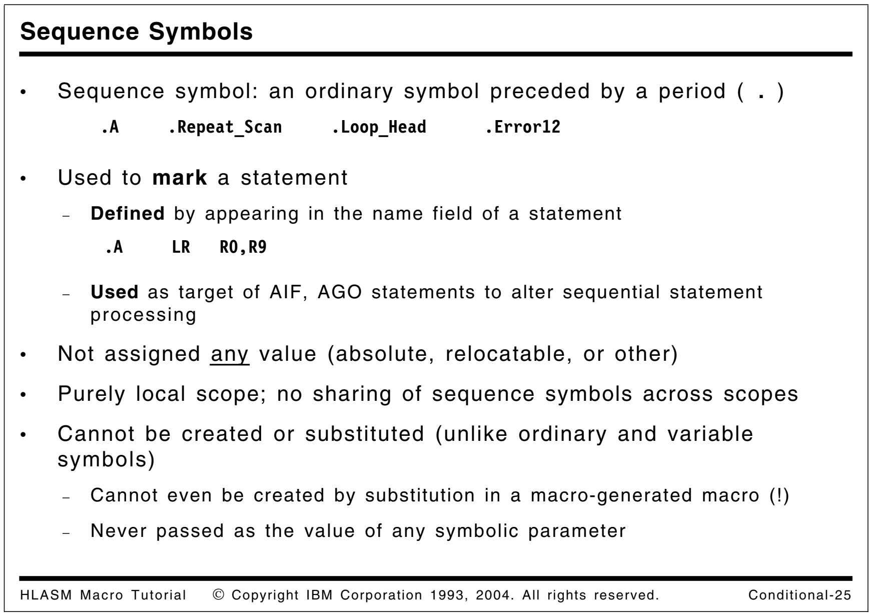
     ANOP conditional-assembly “No-Operation”

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-24

Statement Selection

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

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



Sequence Symbols

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

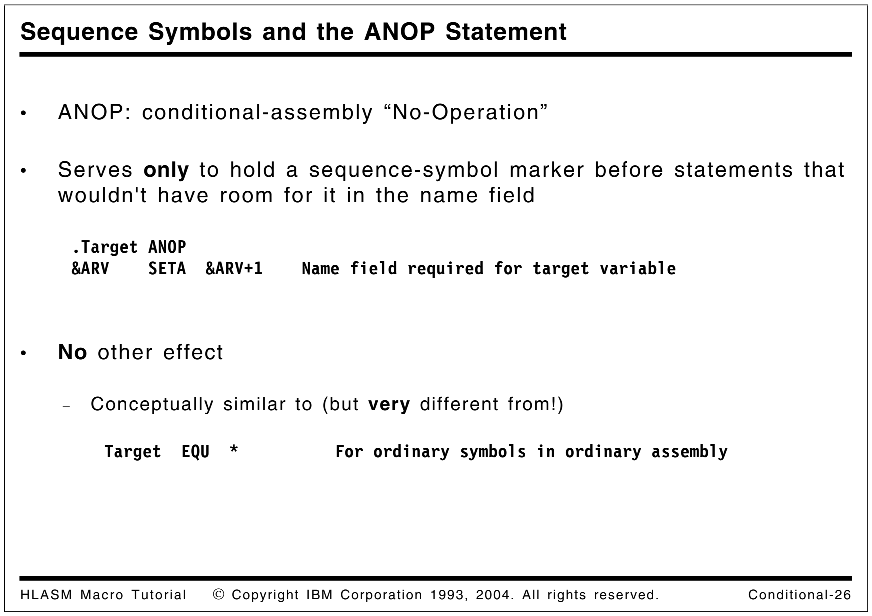
```
.A    .Repeat_Scan    .Loop_Head     .Error12
```

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

```
.A     LR   R0,R9
```

  - **Used** as target of AIF, AGO statements to alter sequential statement processing
- Not assigned any value (absolute, relocatable, or other)
- Purely local scope; no sharing of sequence symbols across scopes
- Cannot be created or substituted (unlike ordinary and variable symbols)
  - Cannot even be created by substitution in a macro-generated macro (!)
  - Never passed as the value of any symbolic parameter

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-25



Sequence Symbols and the ANOP Statement

- ANOP: conditional-assembly “No-Operation”
- Serves **only** to hold a sequence-symbol marker before statements that wouldn't have room for it in the name field

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

- **No** other effect
  - Conceptually similar to (but **very** different from!)

```
Target  EQU  *         For ordinary symbols in ordinary assembly
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-26

Sequence Symbols and the ANOP Statement

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

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

Sequence symbols have some unusual properties compared to ordinary symbols.

- Sequence symbols are defined by appearing in name field of any statement. They may appear on ordinary-assembly statements and on conditional-assembly statements, with no difference in meaning or behavior.
- Sequence symbols are not assigned an absolute or relocatable value, and they do not appear in the assembler's Symbol Table. They cannot be used in expressions of any kind.

- Sequence symbols have purely local scope. That is, there is no sharing of sequence symbols between macros, or between macros and ordinary “open code” assembly.
- Sequence symbols cannot be created or substituted (unlike ordinary and variable symbols).
- Sequence symbols are never passed as values of any symbolic parameter. Thus, although they can appear in the name field of a macro instruction statement (or macro “call”), they are never made available to the macro definition as the value of a name-field variable symbol.
- Sequence symbols are used as the target of AIF and AGO statements to alter sequential statement processing, and for no other purpose.
- Sequence symbols may be defined before or after references to them. This means that both forward and backward “branches” are possible (including the possibility of endless loops).[2]

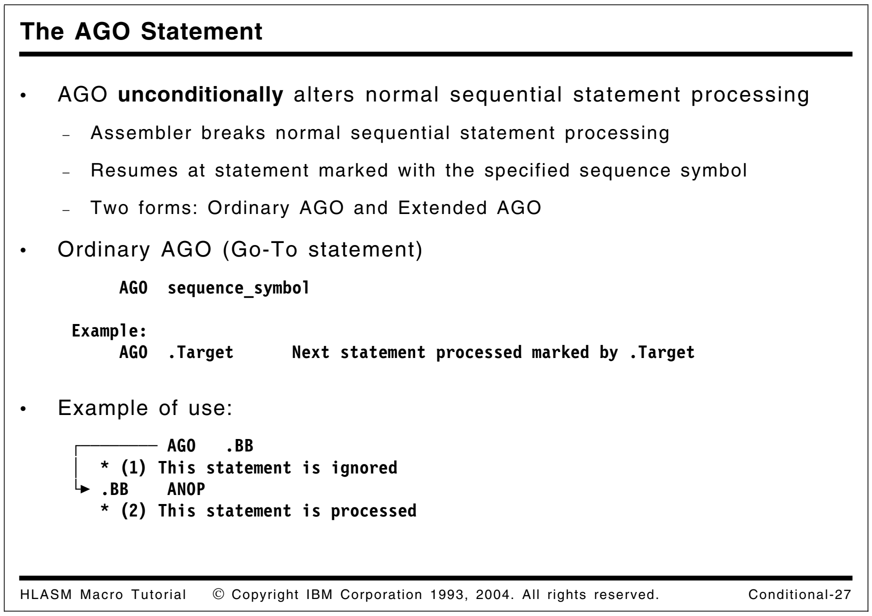
ANOP Statement

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

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

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

[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 be able to create a listing with an error message.



The AGO Statement

- AGO **unconditionally** alters normal sequential statement processing
  - Assembler breaks normal sequential statement processing
  - Resumes at statement marked with the specified sequence symbol
  - Two forms: Ordinary AGO and Extended AGO
- Ordinary AGO (Go-To statement)

```
      AGO  sequence_symbol

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

- Example of use:

```
┌──────── AGO   .BB
│  * (1) This statement is ignored
└► .BB    ANOP
   * (2) This statement is processed
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-27

The AGO Statement

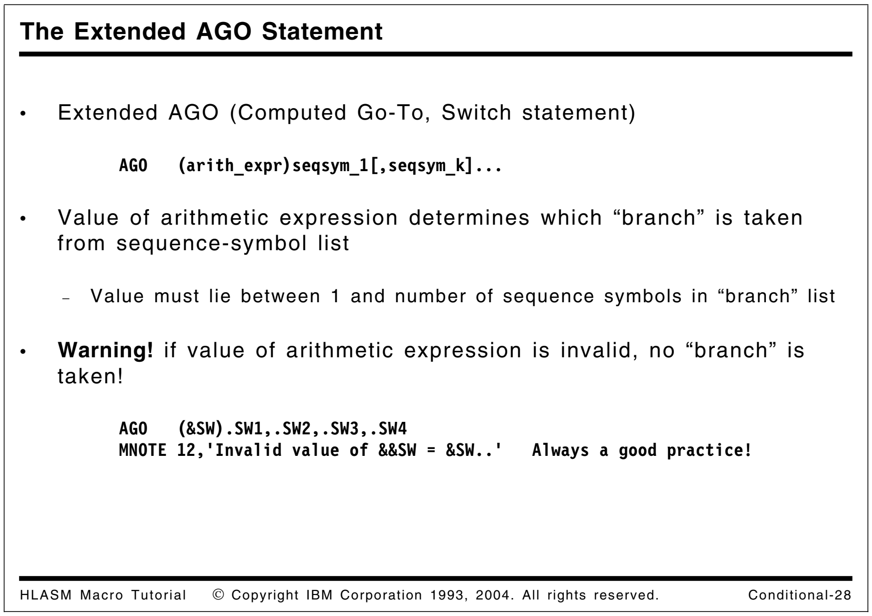
The function of the AGO statement is to unconditionally alter the sequence of statement processing, which resumes at the statement “marked” with the specified sequence symbol. It is written in the form

```
      AGO  sequence_symbol

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

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

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

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



The Extended AGO Statement

- Extended AGO (Computed Go-To, Switch statement)

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

- Value of arithmetic expression determines which “branch” is taken from sequence-symbol list
  - Value must lie between 1 and number of sequence symbols in “branch” list
- **Warning!** if value of arithmetic expression is invalid, no “branch” is taken!

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-28

The Extended AGO Statement

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

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

Figure 4. General Form of the Extended AGO Statement

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

A recommended technique is the following:

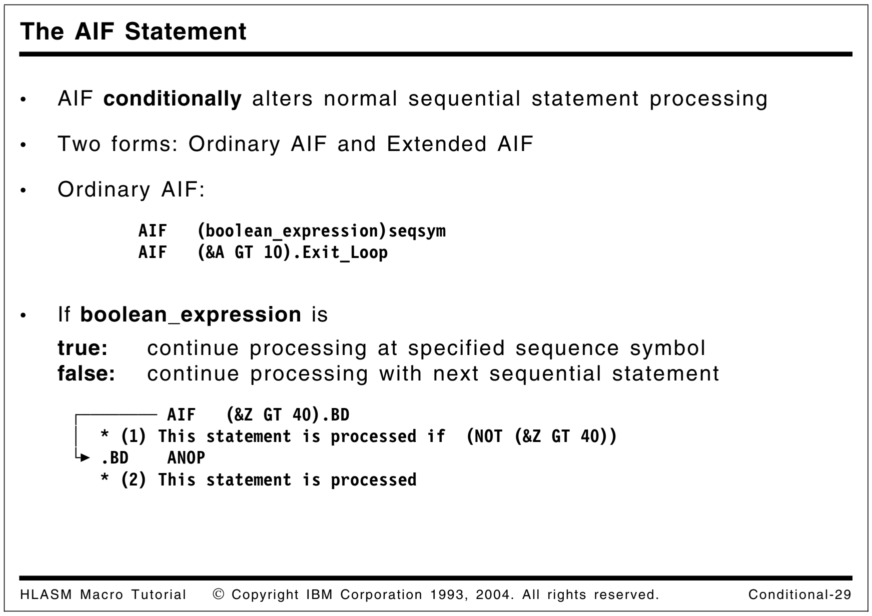
```
AGO   (&SW).SW1,.SW2,.SW3,.SW4
MNOTE 12,'Invalid value of &&SW = &SW..'   Always a good practice!
```

where a message indication is placed after the AGO to trap cases where the arithmetic variable's value is invalid.

The operation of the extended AGO statement illustrated in Figure 4 is precisely equivalent to the following set of AIF statements (which will be described shortly):

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

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



The AIF Statement

- AIF **conditionally** alters normal sequential statement processing
- Two forms: Ordinary AIF and Extended AIF
- Ordinary AIF:

```
AIF   (boolean_expression)seqsym
AIF   (&A GT 10).Exit_Loop
```

- If **boolean_expression** is

  true: continue processing at specified sequence symbol
  false: continue processing with next sequential statement

```
┌─────────── AIF   (&Z GT 40).BD
│  * (1) This statement is processed if  (NOT (&Z GT 40))
└► .BD    ANOP
   * (2) This statement is processed
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-29

The AIF Statement

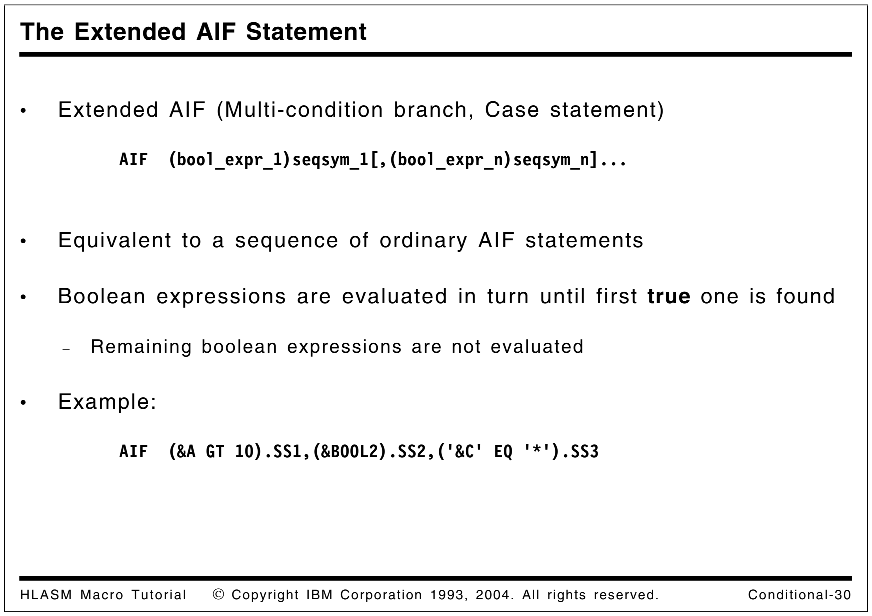
The AIF statement provides a method for conditionally selecting a sequence of statements, by testing a condition before deciding to “branch” or not to the statement designated by a specified sequence symbol. The ordinary AIF statement is written in this form:

```
      AIF  (boolean_expression)seqsym

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

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

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

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



The Extended AIF Statement

- Extended AIF (Multi-condition branch, Case statement)

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

- Equivalent to a sequence of ordinary AIF statements
- Boolean expressions are evaluated in turn until first **true** one is found
  - Remaining boolean expressions are not evaluated
- Example:

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

HLASM Macro Tutorial Conditional-30

The Extended AIF Statement

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

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

Figure 5. General Form of the Extended AIF Statement

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

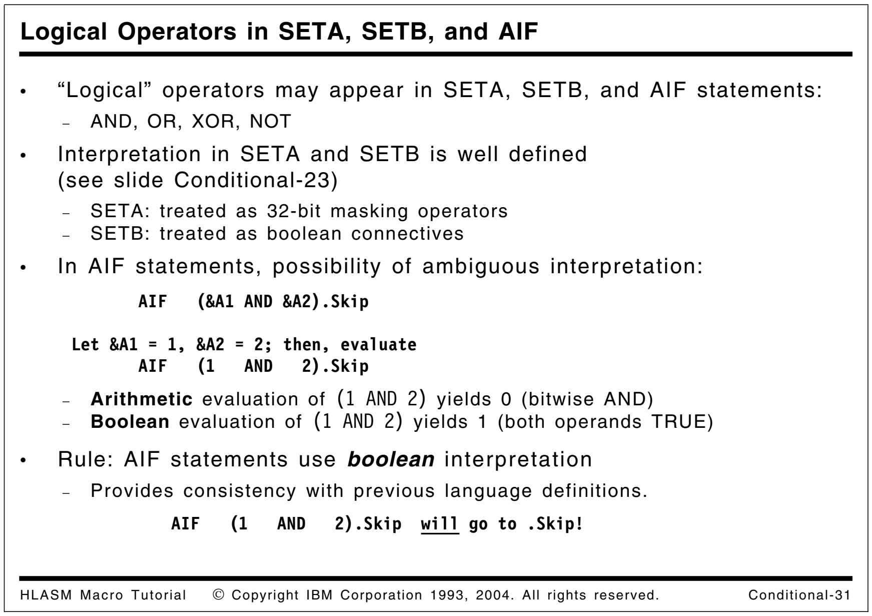
An example of an extended AIF statement is:

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

The extended AIF statement illustrated in Figure 5 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.



Logical Operators in SETA, SETB, and AIF

- “Logical” operators may appear in SETA, SETB, and AIF statements:
  - AND, OR, XOR, NOT
- Interpretation in SETA and SETB is well defined (see slide Conditional-23)
  - SETA: treated as 32-bit masking operators
  - SETB: treated as boolean connectives
- In AIF statements, possibility of ambiguous interpretation:

```
         AIF   (&A1 AND &A2).Skip

 Let &A1 = 1, &A2 = 2; then, evaluate
         AIF   (1   AND   2).Skip
```

  - **Arithmetic** evaluation of (1 AND 2) yields 0 (bitwise AND)
  - **Boolean** evaluation of (1 AND 2) yields 1 (both operands TRUE)
- Rule: AIF statements use ***boolean*** interpretation
  - Provides consistency with previous language definitions.

```
              AIF   (1   AND   2).Skip  will go to .Skip!
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-31

Logical Operators in SETA, SETB, and AIF Statements

Certain logical operators may appear in SETA, SETB, and AIF statements and pose a possibly ambiguous interpretation: AND, OR, XOR, and NOT. 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 the discussion at “Conditional Expressions with Mixed Operand Types” on page 33 for details.)

However, in AIF statements there is a possible ambiguity, as the following example illustrates:

```
         AIF   (&A1 AND &A2).Skip
```

Suppose variable &A1 has value 1, and &A2 has value 2. Consider this AIF statement:

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

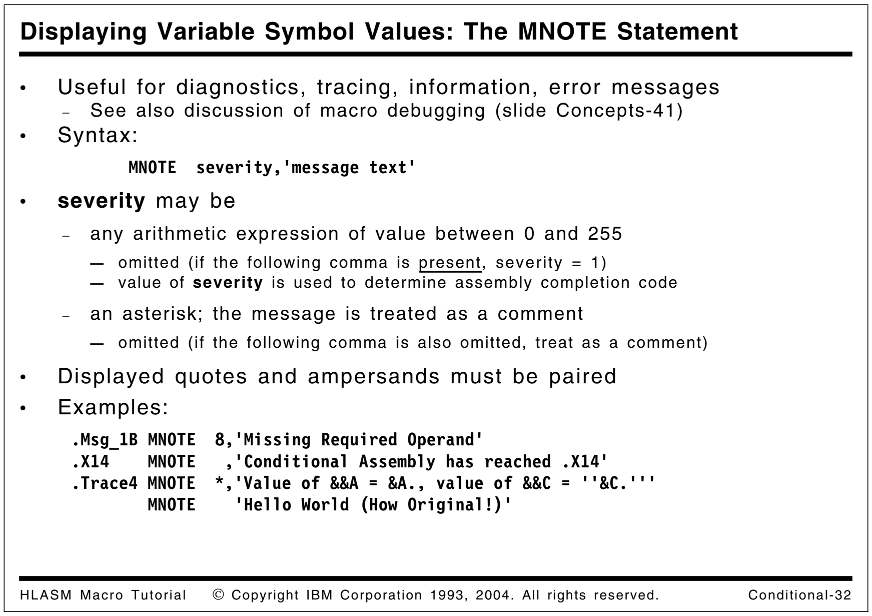
If the expression is evaluated using “SETA rules”, its value is zero: the arithmetic representations of 1 and 2 have no one-bits in common, so their logical AND is zero.

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 33, the result must be 1 (because both 1 and 2 are nonzero, 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.



Displaying Variable Symbol Values: The MNOTE Statement

- Useful for diagnostics, tracing, information, error messages
  - See also discussion of macro debugging (slide Concepts-41)
- Syntax:

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

- **severity** may be
  - any arithmetic expression of value between 0 and 255
    - omitted (if the following comma is present, severity = 1)
    - value of **severity** is used to determine assembly completion code
  - an asterisk; the message is treated as a comment
    - omitted (if the following comma is also omitted, treat as a comment)
- Displayed quotes and ampersands must be paired
- Examples:

```
.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!)'
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-32

Displaying Variable Symbol Values: The MNOTE Statement

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

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

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

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

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

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

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

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

The High Level Assembler provides two system variable symbols (&SYSM_SEV and &&SYSM_HSEV) that allow you to determine the values of MNOTE statement severities. These two variables will be discussed in “&SYSM_HSEV and &SYSM_SEV” on page 236.

Examples of Conditional Assembly

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

Example: Generate a Byte String with Values 1-N

- Sample 0: write everything by hand

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

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

- Sample 1: generate separate statements
  - Pseudocode: DO for J = 1 to N (GEN(DC AL1(J)))

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-33

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

- Sample 2: generate a string with the 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'

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-34

Example 1: Generate a Sequence of Byte Values

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

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

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

Conditional assembly techniques can be used to solve this problem so that changing the EQU statement defining N will not require any rewriting. Pseudo-code for such a code sequence might look like this:

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

We can write conditional-assembly statements to generate the sequence of DC statements as follows:

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

Figure 6. Generating a Sequence of Bytes, Individually Defined

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

Alternatively, we could generate only a single DC statement by using a technique that constructs the nominal value string for the DC statement. A pseudo-code sketch of the 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 written as follows:

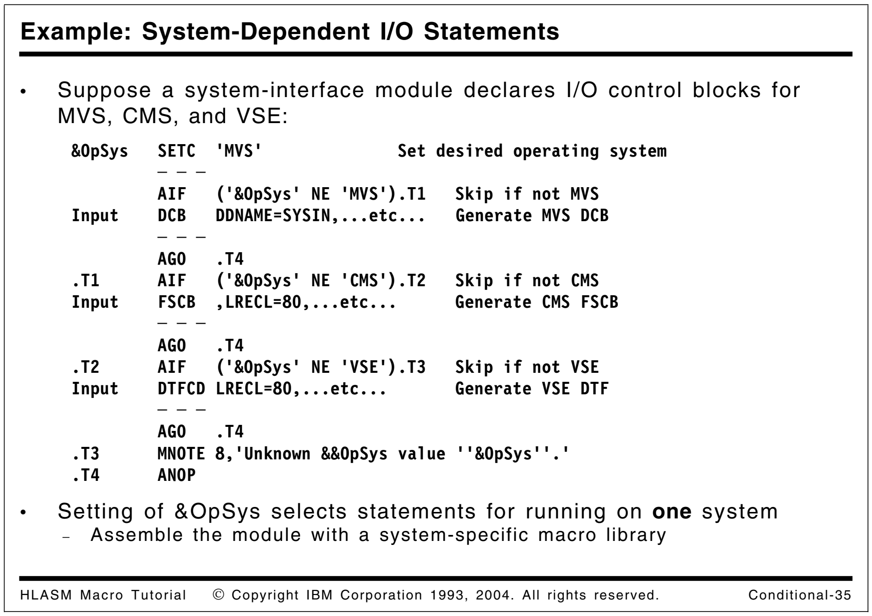
```
N      EQU   5              Predefined absolute symbol
       LCLA  &K             Local arithmetic variable symbol
       LCLC  &S             Local character variable symbol
&K     SETA  1              Initialize counter
       AIF   (&K GT N).Done2 Test for completion (N could be LE 0!)
&S     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 7. Generating a Sequence of Bytes, as a Single Operand String

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

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

Both these examples share a shortcoming: if more than one such sequence of byte values is needed in a program, with different numbers of elements in each sequence, these “blocks” of conditional assembly statements must be repeated. We will see in “Case Study 2: Generating a Sequence of Byte Values” on page 108 that a simple macro definition can make this task easier to solve.



Example: System-Dependent I/O Statements

- 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
  - Assemble the module with a system-specific macro library

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-35

Example 2: Generating System-Dependent I/O Statements

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

This is very simply solved using conditional-assembly statements to select the sequences appropriate to the system for which the module is intended. Suppose you have defined a character-valued variable symbol &OpSys whose values may be MVS, CMS, or VSE. Then the needed code sequences might be defined as in Figure 8:

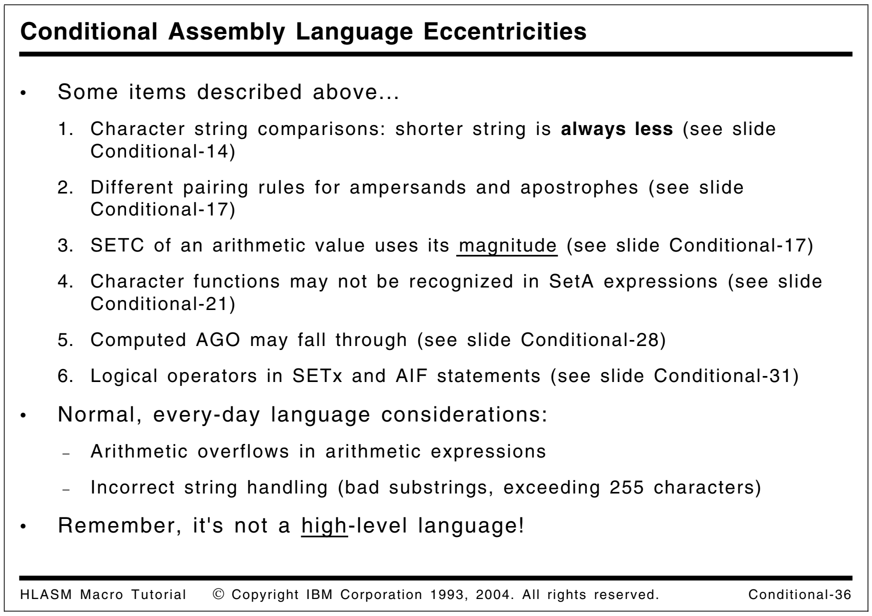
```
&OpSys   SETC  'MVS'              Set desired operating system
         - - -
         AIF   ('&OpSys' NE 'MVS').T1   Skip if not MVS
Input    DCB   DDNAME=SYSIN,...etc...   Generate MVS DCB
         - - -
         AGO   .T4
.T1      AIF   ('&OpSys' NE 'CMS').T2   Skip if not CMS
Input    FSCB  ,LRECL=80,...etc...      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 8. 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 particular to one of the environments, conditional assembly statements allow the assembler to select the correct statements. By setting a single variable symbol &OpSys to an appropriate value, you can tailor the application to a chosen environment without having to make into multiple copies of its processing logic, one for each environment.

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

The technique illustrated here allows you to make your programs more portable across operating environments, 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

- Some items described above...
  1. Character string comparisons: shorter string is **always less** (see slide Conditional-14)
  2. Different pairing rules for ampersands and apostrophes (see slide Conditional-17)
  3. SETC of an arithmetic value uses its magnitude (see slide Conditional-17)
  4. Character functions may not be recognized in SetA expressions (see slide Conditional-21)
  5. Computed AGO may fall through (see slide Conditional-28)
  6. Logical operators in SETx and AIF statements (see slide Conditional-31)
- Normal, every-day language considerations:
  - Arithmetic overflows in arithmetic expressions
  - Incorrect string handling (bad substrings, exceeding 255 characters)
- Remember, it's not a high-level language!

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Conditional-36

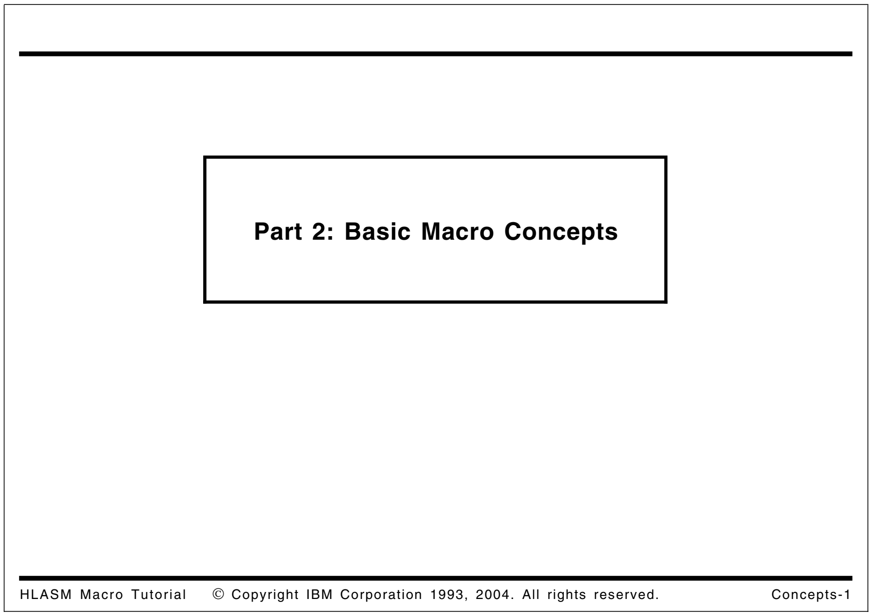
Conditional Assembly Language Eccentricities

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 20.)
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 22.)
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 22.)
4. Internal function evaluations involving string functions cannot always be “nested” in arithmetic expressions. (See “Character Expressions: Internal Character Functions” 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 38.)
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 41.)

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

Part 2: Basic Macro Concepts



Part 2: Basic Macro Concepts

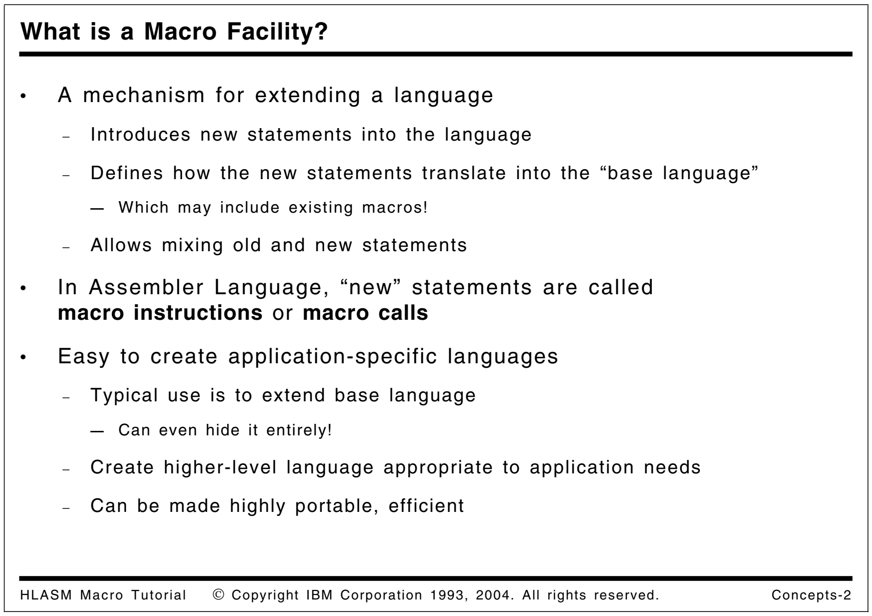
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-1

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

We will begin with a conceptual overview of the basic concepts of macros, in a way that is not specific to the Assembler Language.[3] This will be followed by an investigation of the System/360/370/390 Assembler Language's implementation of macros, including the following topics:

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

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



What is a Macro Facility?

- A mechanism for extending a language
  - Introduces new statements into the language
  - Defines how the new statements translate into the “base language”
    - Which may include existing macros!
  - Allows mixing old and new statements
- In Assembler Language, “new” statements are called **macro instructions** or **macro calls**
- Easy to create application-specific languages
  - Typical use is to extend base language
    - Can even hide it entirely!
  - Create higher-level language appropriate to application needs
  - Can be made highly portable, efficient

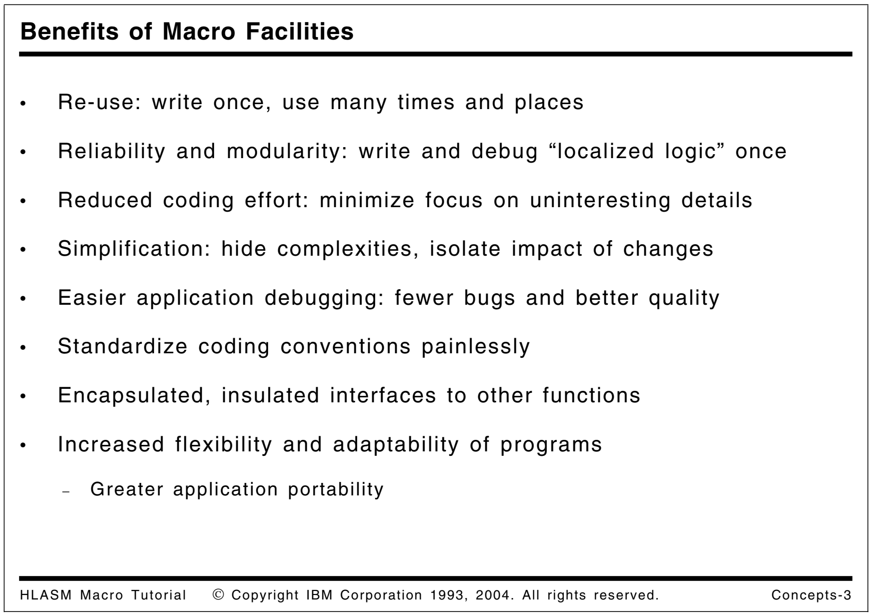
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-2

What is a Macro Facility?

Most simply, a macro facility is a mechanism for extending a language. It can be used not only to introduce new statements into the language, but also to define how the new statements should be translated into the “base language” on which they are built. One major advantage of macros is that they allow you to mix “old” (existing) and “new” statements, so that your language can grow incrementally to accommodate new functions, added requirements, and other benefits as and when you are able to take advantage of them. The “old” statements may include existing macros, providing 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 are compared in more detail at “Macros and Subroutines” on page 99.



Benefits of Macro Facilities

- Re-use: write once, use many times and places
- Reliability and modularity: write and debug “localized logic” once
- Reduced coding effort: minimize focus on uninteresting details
- Simplification: hide complexities, isolate impact of changes
- Easier application debugging: fewer bugs and better quality
- Standardize coding conventions painlessly
- Encapsulated, insulated interfaces to other functions
- Increased flexibility and adaptability of programs
  - Greater application portability

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-3

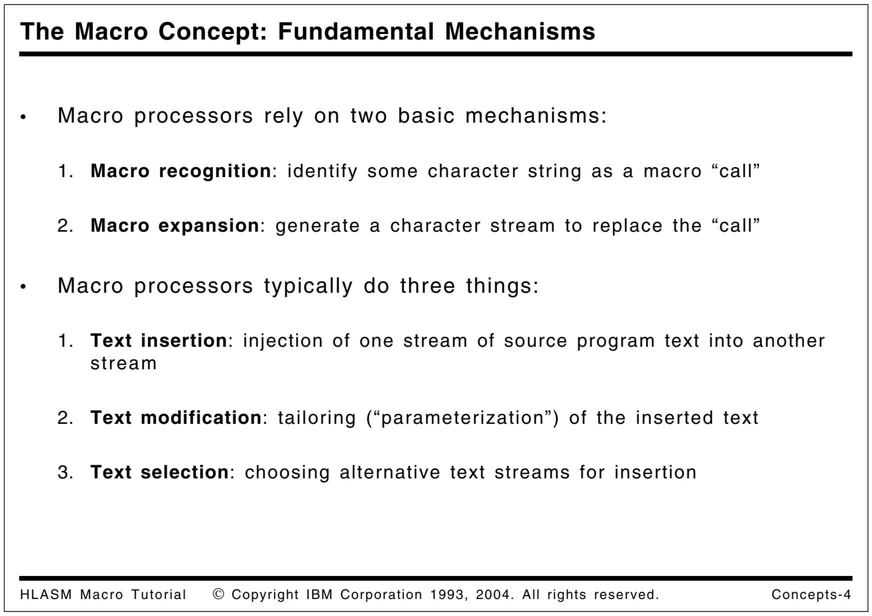
Benefits of Macro Facilities

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

- Code re-use: once a macro is written, it becomes available to as many programmers and applications as are appropriate. A single definition can find multiple uses (even within a single application).
- Reliability and modularity: code and debug the logic in one place.
- Reduced coding effort: the coding in a macro needs to be written only once, and then can be used in many places.
- Reduced focus on uninteresting details: macros allow you to create “higher-level” elements of your programming language, relieving you of the need to be concerned with details that are typically 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.[4]
- 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.

[4] 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.

- Provide encapsulated interfaces to other functions, insulated from interface changes: using macros, you can support interfaces among different elements of your applications, and between applications and operating environments, in a controlled and defined way. This means that changes to those interfaces can be made in the macros, without affecting the coding of the applications themselves.
- Localized logic: specific and detailed (and often complex) code sequences can be implemented once in a macro, and used wherever needed, without the need for every user of the macro to understand the “inner workings” of the macro's logic.
- 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 rely on two basic mechanisms:
  1. **Macro recognition**: identify some character string as a macro “call”
  2. **Macro expansion**: generate a character stream to replace the “call”
- Macro processors typically do three things:
  1. **Text insertion**: injection of one stream of source program text into another stream
  2. **Text modification**: tailoring (“parameterization”) of the inserted text
  3. **Text selection**: choosing alternative text streams for insertion

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-4

The Macro Concept: Fundamental Mechanisms

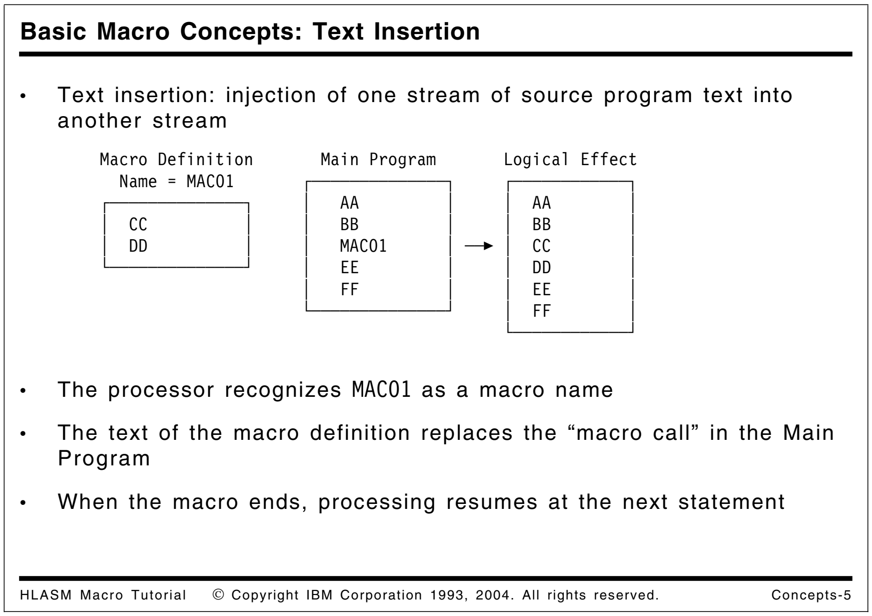
Macro processors typically rely on two basic processes:

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

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

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

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



Basic Macro Concepts: Text Insertion

- Text insertion: injection of one stream of source program text into another stream

```
Macro Definition      Main Program       Logical Effect
  Name = MAC01
                       AA                  AA
  CC                   BB                  BB
  DD                   MAC01        -->    CC
                       EE                  DD
                       FF                  EE
                                           FF
```

- The processor recognizes MAC01 as a macro name
- The text of the macro definition replaces the “macro call” in the Main Program
- When the macro ends, processing resumes at the next statement

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-5

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 9, a set of statements has been defined to be a macro with the name MAC01. When the processor of the Main Program recognizes the string MAC01 as matching that of the macro, that string is *replaced* by the text within the macro definition. Finally, when the macro ends, statement processing resumes at the next statement following the macro call.

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

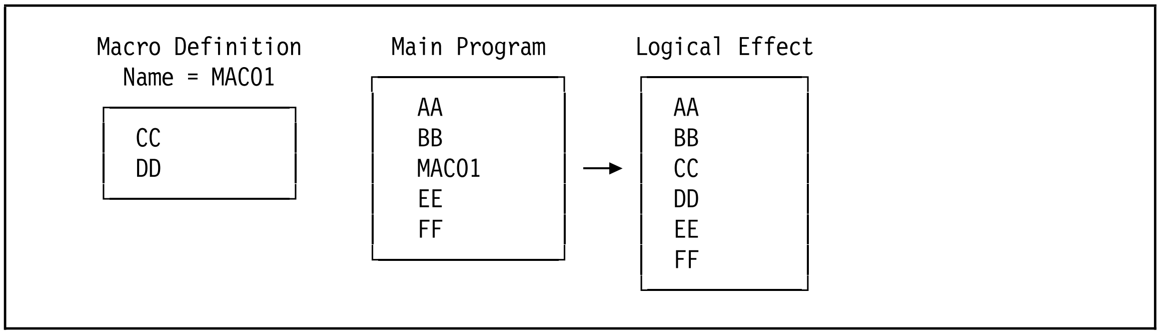
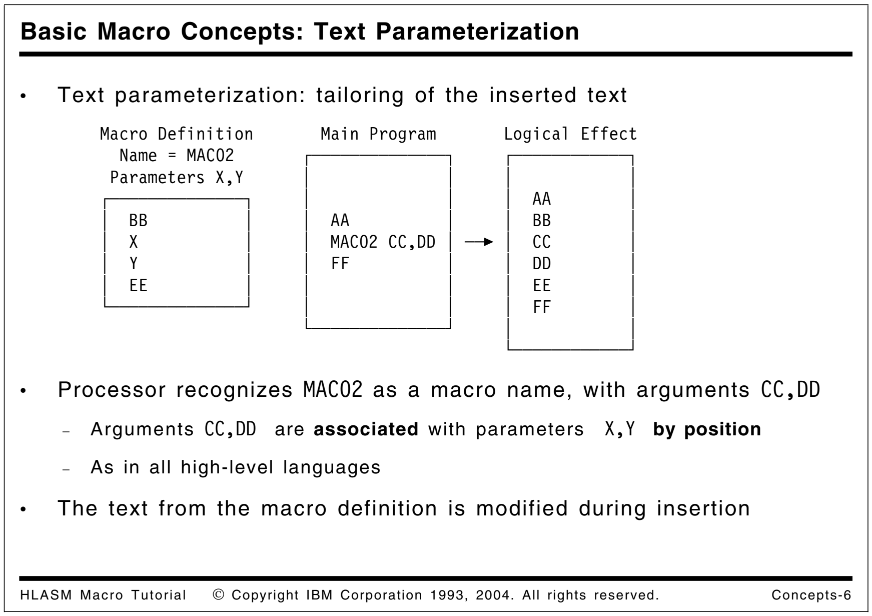


Figure 9. Basic Macro Mechanisms: Text Insertion



Basic Macro Concepts: Text Parameterization

- Text parameterization: tailoring of the inserted text

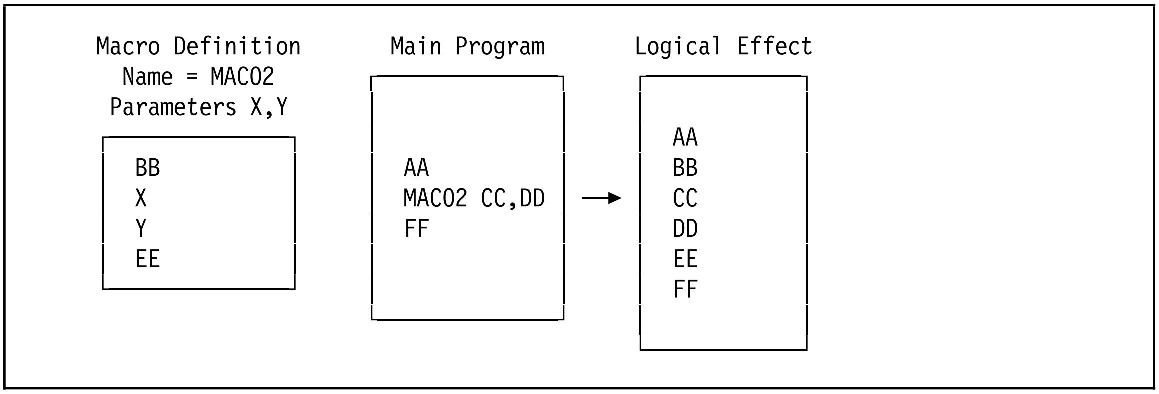
```
  Macro Definition      Main Program      Logical Effect
    Name = MAC02
   Parameters X,Y
                                              AA
   BB                   AA                    BB
   X                    MAC02 CC,DD   →       CC
   Y                    FF                    DD
   EE                                         EE
                                              FF
```

- Processor recognizes MAC02 as a macro name, with arguments CC,DD
  - Arguments CC,DD are **associated** with parameters X,Y **by position**
  - As in all high-level languages
- The text from the macro definition is modified during insertion

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-6

Text Parameterization and Argument Association

Simple text insertion has rather limited uses, because we usually want to tailor and adapt the inserted text to accommodate the various conditions and situations of each macro invocation. The simplest form of such adaptation is “text parameterization”. In Figure 10, the macro with name MAC02 is defined with two *parameters* X and Y: that is, they are merely place-holders in the definition that indicate where other text strings are expected to be inserted when the macro is expanded.



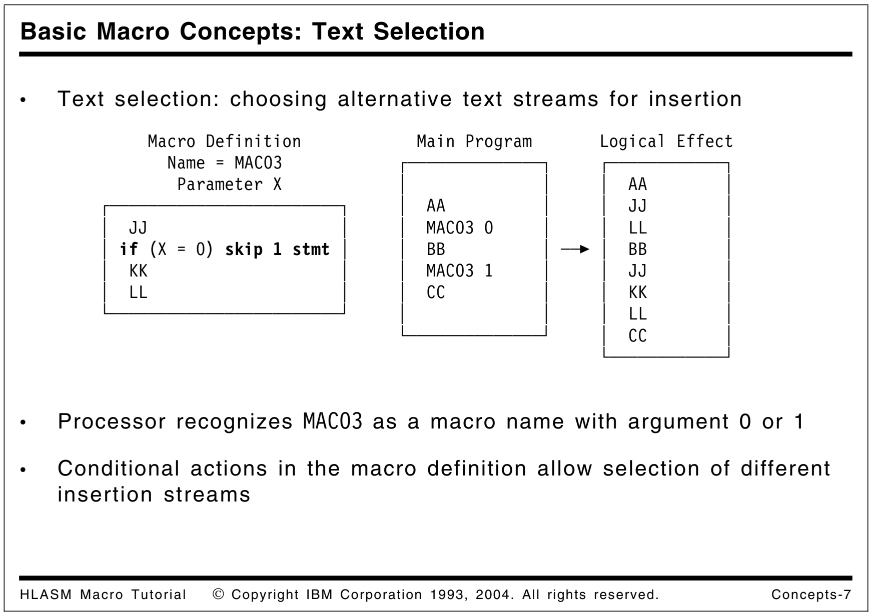
```
  Macro Definition      Main Program      Logical Effect
    Name = MAC02
   Parameters X,Y
                                              AA
   BB                   AA                    BB
   X                    MAC02 CC,DD   →       CC
   Y                    FF                    DD
   EE                                         EE
                                              FF
```

Figure 10. Basic Macro Mechanisms: Text Parameterization

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

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

The argument CC is *associated* with parameter X, and DD is associated with Y. This simple example of parameter-argument association is typical of many macro processors: association proceeds in left-to-right order, matching each positional parameter in turn with its corresponding positional argument. This is the familiar form of association used in almost all high-level programming languages. Other forms of association are possible.



Basic Macro Concepts: Text Selection

- Text selection: choosing alternative text streams for insertion

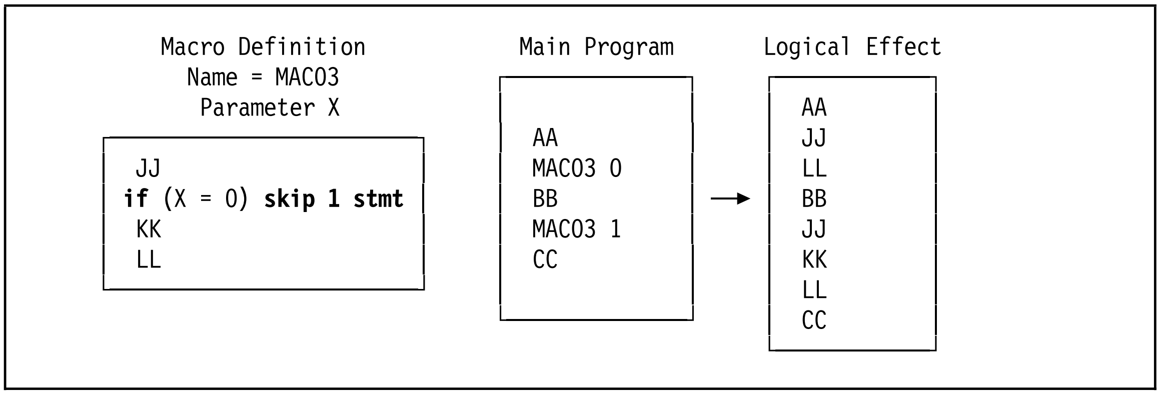
```
     Macro Definition          Main Program     Logical Effect
      Name = MAC03
       Parameter X                                AA
                                  AA              JJ
   JJ                             MAC03 0         LL
  if (X = 0) skip 1 stmt          BB        ──▶   BB
   KK                             MAC03 1         JJ
   LL                             CC              KK
                                                  LL
                                                  CC
```

- Processor recognizes MAC03 as a macro name with argument 0 or 1
- Conditional actions in the macro definition allow selection of different insertion streams

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-7

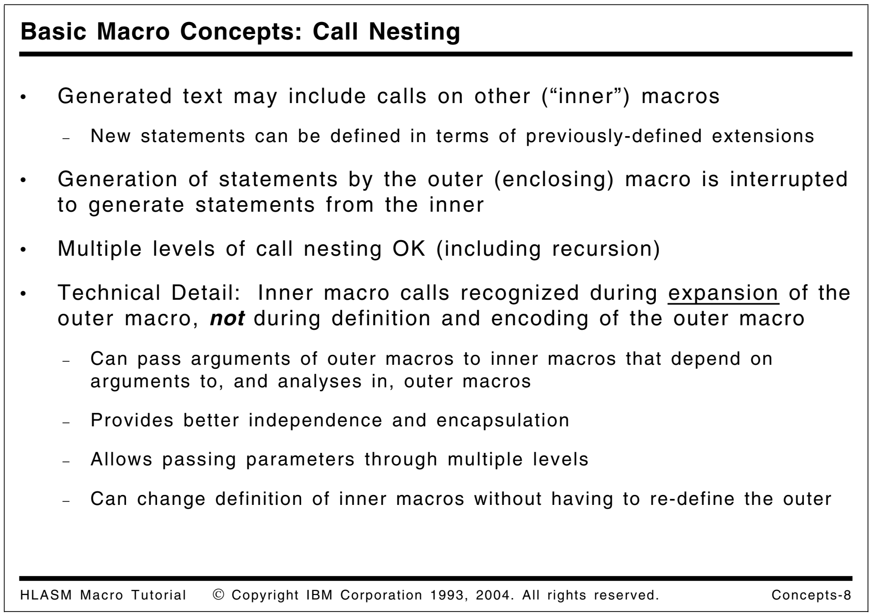
Text Selection

Text selection is fundamental to most macro processors, because it allows choices among alternative sequences of generated text. In Figure 11, 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 KK. If the argument is 0, KK is not generated; otherwise it is.



```
     Macro Definition          Main Program     Logical Effect
      Name = MAC03
       Parameter X                                AA
                                  AA              JJ
   JJ                             MAC03 0         LL
  if (X = 0) skip 1 stmt          BB        ──▶   BB
   KK                             MAC03 1         JJ
   LL                             CC              KK
                                                  LL
                                                  CC
```

Figure 11. Basic Macro Mechanisms: Text Selection



Basic Macro Concepts: Call Nesting

- Generated text may include calls on other (“inner”) macros
  - New statements can be defined in terms of previously-defined extensions
- Generation of statements by the outer (enclosing) macro is interrupted to generate statements from the inner
- Multiple levels of call nesting OK (including recursion)
- Technical Detail: Inner macro calls recognized during expansion of the outer macro, ***not*** during definition and encoding of the outer macro
  - Can pass arguments of outer macros to inner macros that depend on arguments to, and analyses in, outer macros
  - Provides better independence and encapsulation
  - Allows passing parameters through multiple levels
  - Can change definition of inner macros without having to re-define the outer

HLASM Macro Tutorial Concepts-8

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 called “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.

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

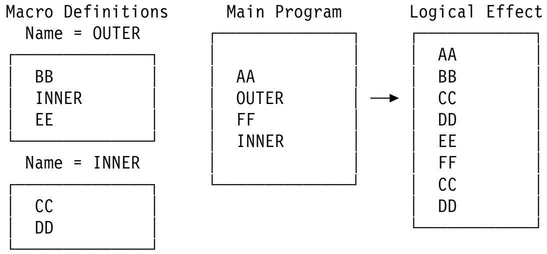
- Generation of statements by the outer (enclosing) macro is suspended temporarily to generate statements from the inner.
- Multiple levels of call nesting are quite acceptable (including recursion: a macro may call itself directly or indirectly), and are often a source of added power and flexibility.

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

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

Macro Call Nesting: Example

- Two macro definitions: OUTER contains a call on INNER



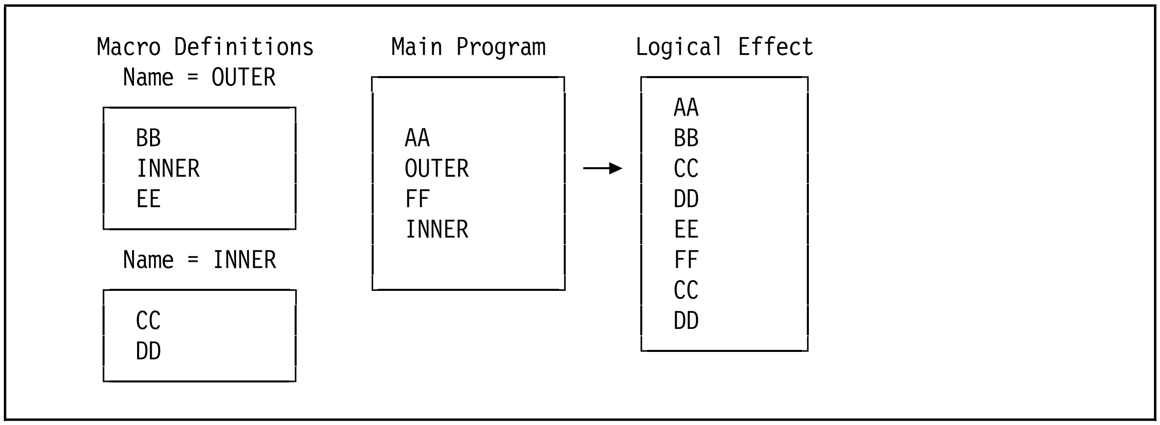
```
Macro Definitions     Main Program     Logical Effect
  Name = OUTER
                                          AA
   BB                  AA                 BB
   INNER               OUTER      →       CC
   EE                  FF                 DD
                       INNER              EE
  Name = INNER                            FF
                                          CC
   CC                                     DD
   DD
```

- Expansion of OUTER is suspended until expansion of INNER completes

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-9

In the example in Figure 12, two macros named OUTER and INNER are known to the processor of the Main Program. When the name OUTER is recognized as a macro name, processing of the Main Program is suspended and expansion of the OUTER macro begins. When INNER is recognized as as macro name, processing of the OUTER macro is also suspended and expansion of the INNER macro begins. When the INNER macro expansion completes, the OUTER macro resumes expansion 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 the call is recognized.

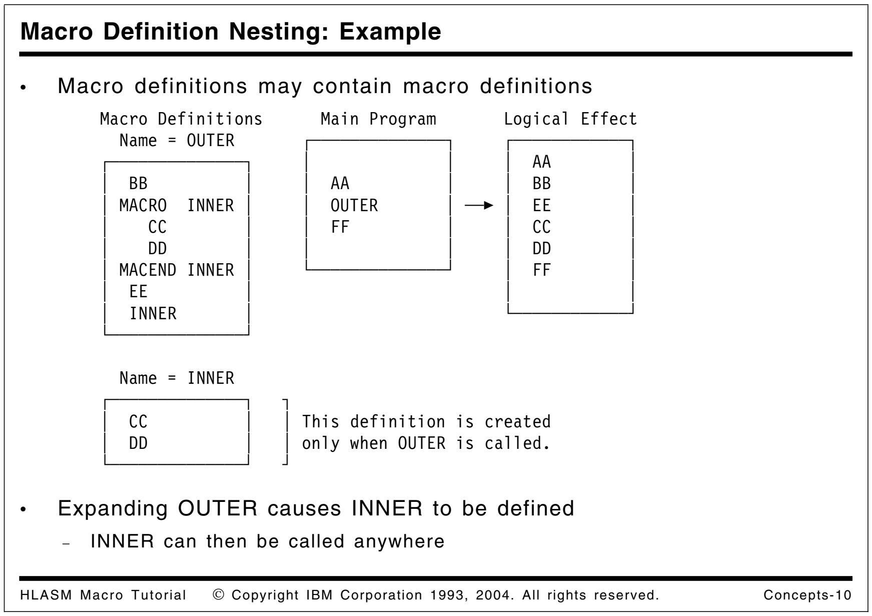


```
Macro Definitions     Main Program     Logical Effect
  Name = OUTER
                                          AA
   BB                  AA                 BB
   INNER               OUTER      →       CC
   EE                  FF                 DD
                       INNER              EE
  Name = INNER                            FF
                                          CC
   CC                                     DD
   DD
```

Figure 12. Basic Macro Mechanisms: Call Nesting

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

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



Macro Definition Nesting: Example

- Macro definitions may contain macro definitions

```
Macro Definitions      Main Program       Logical Effect
  Name = OUTER
                                            AA
  BB                    AA                  BB
 MACRO  INNER           OUTER      →        EE
    CC                  FF                  CC
    DD                                      DD
 MACEND INNER                               FF
  EE
  INNER

  Name = INNER

  CC                This definition is created
  DD                only when OUTER is called.
```

- Expanding OUTER causes INNER to be defined
  - INNER can then be called anywhere

HLASM Macro Tutorial Concepts-10

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 13, where we suppose that the definition of the macro named INNER is enclosed within the MACRO and MACEND statements.

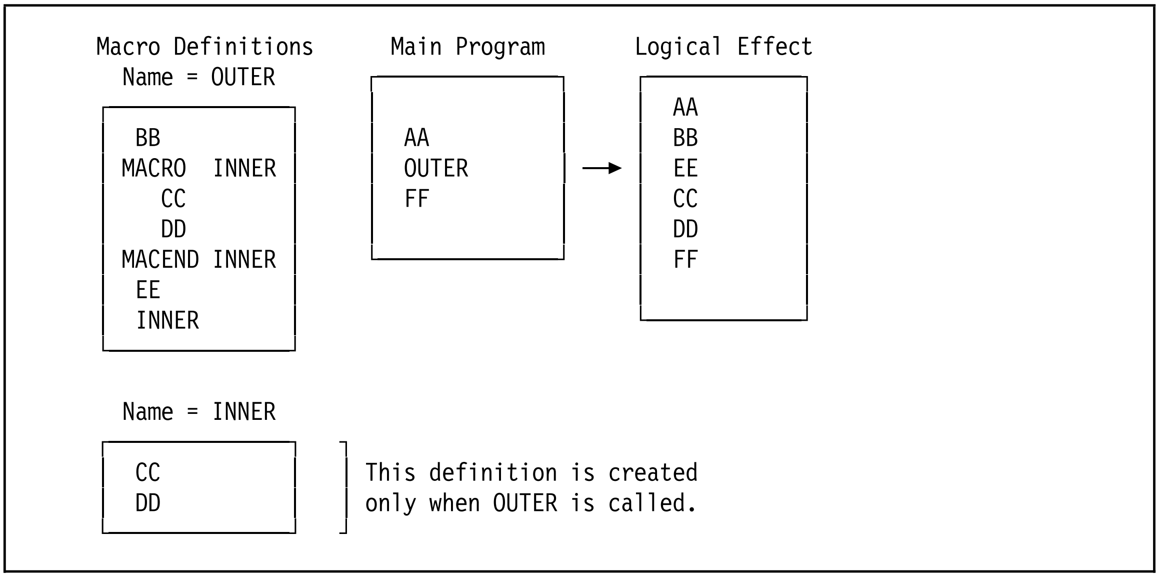


Figure 13. Basic Macro Mechanisms: Nested Macro Definitions

In this example, only the OUTER macro is known to the processor of the Main Program. When the name OUTER is recognized as a macro call, processing of the Main Program is suspended and expansion of the OUTER macro begins. When the 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.

Note that the INNER macro 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 safely be called from anywhere in the main program or from other macros.

The Assembler Language Macro Definition

- A macro definition has four parts:

```
        ┌─────────────────────────┐
  (1)   │   MACRO                 │  Macro Header (begins a definition).
        ├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
  (2)   │   Prototype Statement   │  Model of the macro instruction
        │                         │  that can call on this definition;
        │                         │  a model or “template” of the new
        │                         │  statement introduced into the
        │                         │  language by this definition.
        │                         │  A single statement.
        ├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
  (3)   │   Model Statements      │  Declarations, conditional assembly
        │                         │  statements, and text for selection,
        │                         │  modification, and insertion.
        │                         │  Zero to many statements.
        │                         │
        ├ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┤
  (4)   │   MEND                  │  Macro Trailer (ends a definition).
        └─────────────────────────┘
```

The Assembler Language Macro Definition ...

1. Declares a macro name representing a stream of program text

2. MACRO and MEND statements delimit start and end of the definition

3. Prototype statement declares parameter variable symbols

4. 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

- Recognition rules affected by where the definition is found

The Assembler Language Macro Definition

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

1. a macro header statement (MACRO: the start of the definition)

2. a prototype statement, which provides the macro name and a model 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 14:

```
(1)  |   MACRO                |  Macro Header (begins a definition).
     |- - - - - - - - - - - - |
(2)  |   Prototype Statement  |  Model of the macro instruction
     |                        |  that can call on this definition;
     |                        |  a model or “template” of the new
     |                        |  statement introduced into the
     |                        |  language by this definition.
     |                        |  A single statement.
     |- - - - - - - - - - - - |
(3)  |   Model Statements     |  Declarations, conditional assembly
     |                        |  statements, and text for selection,
     |                        |  modification, and insertion.
     |                        |  Zero to many statements.
     |                        |
     |- - - - - - - - - - - - |
(4)  |   MEND                 |  Macro Trailer (ends a definition).
```

Figure 14. 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 will be described in “Macro-Instruction Recognition: Details” on page 62.

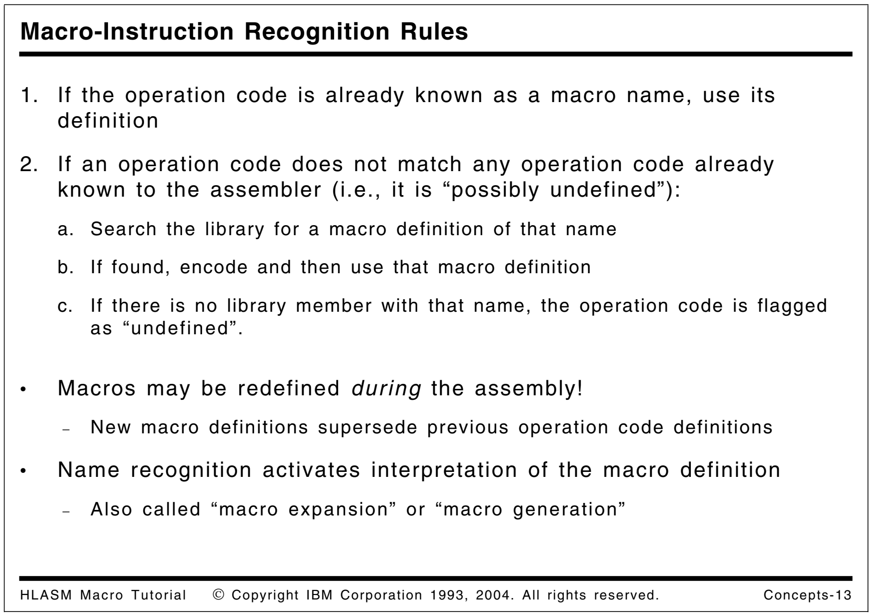
Macro-Instruction Definition Example

We can rewrite the example in Figure 9 on page 52 to look like a “real” macro, as follows:

```
                        Main Program      Logical Effect
  Macro Definition
                           START              START
     MACRO                 AA                 AA
     MAC01                 BB                 BB
     CC                    MAC01          +   CC
     DD                    EE             +   DD
     MEND                  FF                 EE
                           END                FF
                                              END
```

Figure 15. Assembler Language Macro Mechanisms: Text Insertion by a “Real” Macro

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



Macro-Instruction Recognition Rules

1. If the operation code is already known as a macro name, use its definition
2. If an operation code does not match any operation code already known to the assembler (i.e., it is “possibly undefined”):
   a. Search the library for a macro definition of that name
   b. If found, encode and then use that macro definition
   c. If there is no library member with that name, the operation code is flagged as “undefined”.

- Macros may be redefined *during* the assembly!
  - New macro definitions supersede previous operation code definitions
- Name recognition activates interpretation of the macro definition
  - Also called “macro expansion” or “macro generation”

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-13

Macro-Instruction Recognition Rules

The assembler recognizes a macro instruction as follows:

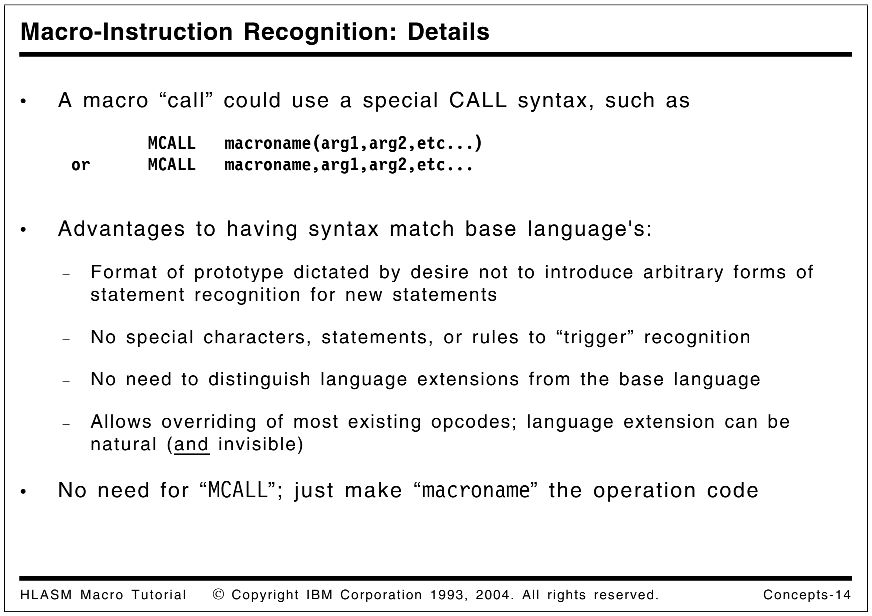
1. If the macro name has already been defined in the program (as a “source” or “in-line” definition, either explicitly or because a COPY statement brought it in-line from a library, or because a previous macro instruction statement brought the definition from the library), use it in preference to any other definition of that operation.
   - You may use a macro definition to override the assembler's default definitions of all machine instruction statements, and of most “native” Assembler Instruction statements (generally, the conditional-assembly statements cannot be overridden).
2. If an operation code does not match any operation code “known” to the assembler (i.e., it is “possibly undefined”), the assembler will then:
   a. Search the library for a macro definition of that name.
   b. If the assembler finds a library member with that name, the macro name defined on the prototype statement must match the member name. The assembler will then encode and use this definition.
   c. If there is no library member with that name, then the operation code is flagged as “undefined”.

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

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

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

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



Macro-Instruction Recognition: Details

- A macro “call” could use a special CALL syntax, such as

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

- Advantages to having syntax match base language's:
  - Format of prototype dictated by desire not to introduce arbitrary forms of statement recognition for new statements
  - No special characters, statements, or rules to “trigger” recognition
  - No need to distinguish language extensions from the base language
  - Allows overriding of most existing opcodes; language extension can be natural (and invisible)
- No need for “MCALL”; just make “macroname” the operation code

HLASM Macro Tutorial Concepts-14

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 System/360/370/390 Assembler Language. Some macro languages and preprocessors 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...
```

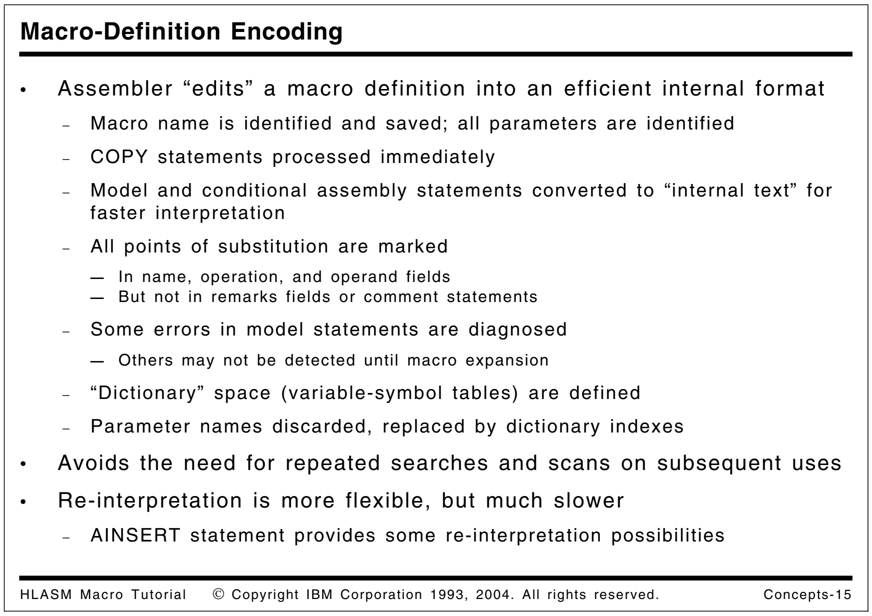
However, there are advantages to having the syntax of macro calls match the base language's, and to allow overriding of existing opcodes; hence, we simply elide the MCALL and make the “macroname” become the operation code and the arguments become the operands of the macro instruction statement.

While many possible forms of macro definition and recognition are possible, the general format used in the System/360/370/390 Assembler Language is dictated by a desire not to introduce arbitrary forms of statement syntax and recognition rules for new statements.

(Note that the syntax of the SETAF and SETCF instructions uses explicit invocation:

```
         SETxF   macroname,arg1,arg2,etc...
```

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



Macro-Definition Encoding

- Assembler “edits” a macro definition into an efficient internal format
  - Macro name is identified and saved; all parameters are identified
  - COPY statements processed immediately
  - Model and conditional assembly statements converted to “internal text” for faster interpretation
  - All points of substitution are marked
    - In name, operation, and operand fields
    - But not in remarks fields or comment statements
  - Some errors in model statements are diagnosed
    - Others may not be detected until macro expansion
  - “Dictionary” space (variable-symbol tables) are defined
  - Parameter names discarded, replaced by dictionary indexes
- Avoids the need for repeated searches and scans on subsequent uses
- Re-interpretation is more flexible, but much slower
  - AINSERT statement provides some re-interpretation possibilities

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-15

Macro-Definition Encoding

Because the System/360/370/390 Assemblers have been designed to support extensive use of macros, their implementation reflects a need to provide efficient processing. Thus, the assembler initially converts macro definitions into an 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 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 a new value represented by `'&A'`.

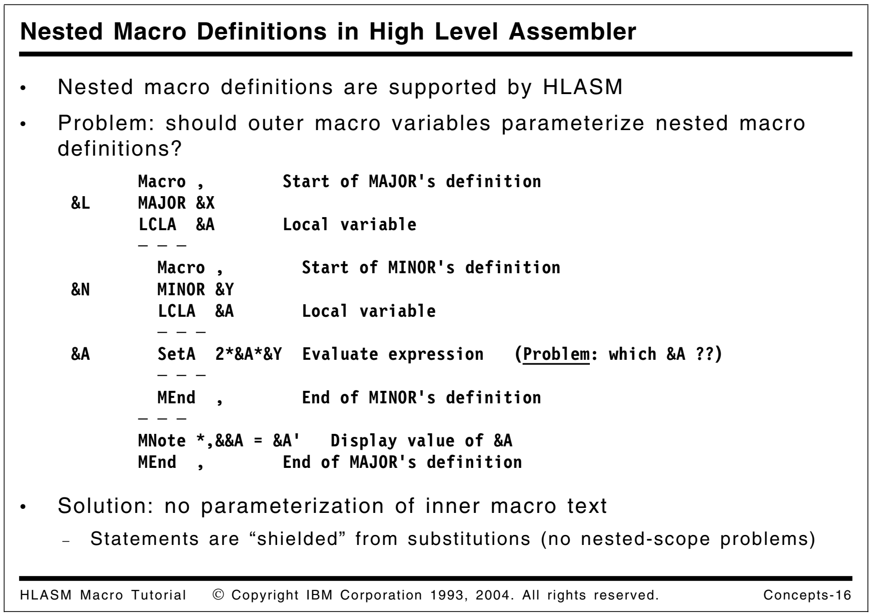
  Note: Because generated machine instruction statements are scanned differently from generated macro instructions, you can create substitutions in remarks fields by creating an “operand” that contains the true operands, one or more blanks, and the characters of the remarks field. This technique is laborious, and is not recommended.

- Some errors in model statements are diagnosed, but others may not be detected until macro expansion is attempted.

- “Dictionary” space (variable-symbol tables) are defined for local variable symbols, and space is added to the global variable symbol dictionary for newly-encountered global names.

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

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 in High Level Assembler

- Nested macro definitions are supported by HLASM
- Problem: should outer macro variables parameterize nested macro definitions?

```
       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: which &A ??)
         - - -
         MEnd  ,       End of MINOR's definition
       - - -
       MNote *,&&A = &A'   Display value of &A
       MEnd  ,       End of MAJOR's definition
```

- Solution: no parameterization of inner macro text
  - Statements are “shielded” from substitutions (no nested-scope problems)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-16

Nested Macro Definition in High Level Assembler

Nested macro definition is supported in the System/360/370/390 Assembler Language, but are more complicated than illustrated in the simple example in Figure 13 on page 57. 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: which &A ??)
         - - -
         MEnd  ,       End of MINOR's definition
       - - -
       MNote *,&&A = &A'   Display value of &A
       MEnd  ,       End of MAJOR's definition
```

Figure 16. Macro Definition Nesting in High Level Assembler

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

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 or “tailor” it in any way.

Some of the limitations imposed by this choice can be overcome by using the AINSERT statement, described in “The AINSERT Statement” on page 185.

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.

Macro Expansion and MEXIT

- Macro **expansion** or **generation** is initiated by **recognition** of a macro instruction
- Assembler suspends current activity, begins to “execute” or “interpret” the encoded definition
  - Parameter values assigned from associated arguments
  - Conditional assembly statements interpreted, variable symbols assigned values
  - Model statements substituted, and output to base language processor
- Generated statements *immediately* scanned for inner macro calls
  - Recognition of inner call suspends current expansion, starts new one
- Expansion terminates when MEND is reached, or MEXIT is interpreted
  - Some error conditions may also cause termination
  - MEXIT is equivalent to “AGO to MEND” (but quicker)

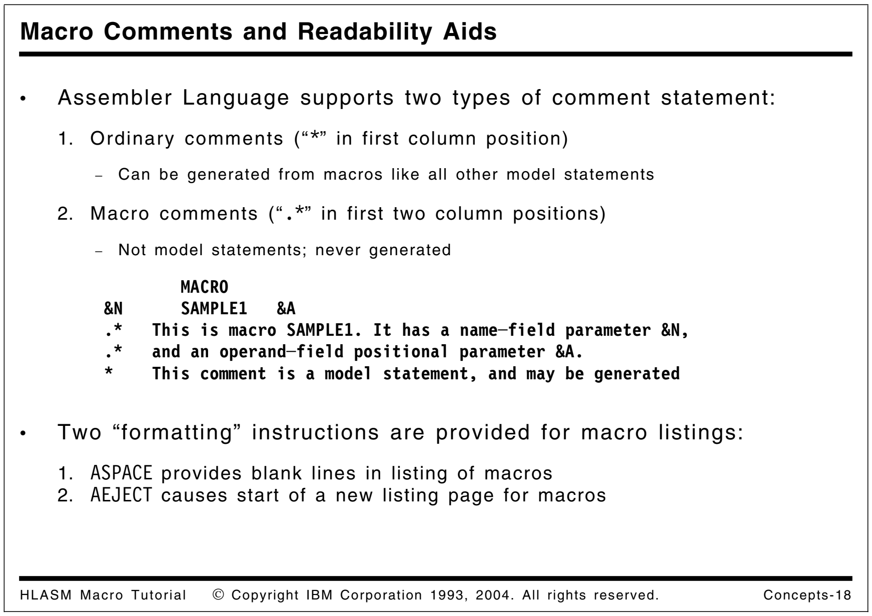
Macro Expansion, Generated Statements, and the MEXIT Statement

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

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

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

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



Macro Comments and Readability Aids

- Assembler Language supports two types of comment statement:
  1. Ordinary comments ("*" in first column position)
     - Can be generated from macros like all other model statements
  2. Macro comments (".*" in first two column positions)
     - Not model statements; never generated

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

- Two "formatting" instructions are provided for macro listings:
  1. ASPACE provides blank lines in listing of macros
  2. AEJECT causes start of a new listing page for macros

HLASM Macro Tutorial Concepts-18

Macro Comments and Readability Aids

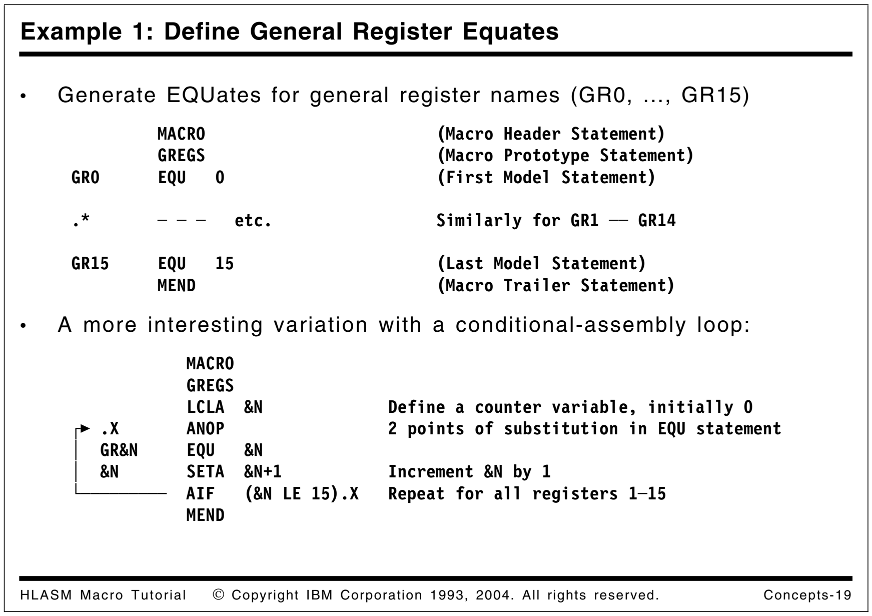
The macro facility provides a way to embed "macro comments" into the body of a macro definition. Because both ordinary comment statements (with an asterisk in the left margin) and blank lines (for spacing) are *model statements*, they may be part of the generated text from a macro expansion. Macro comments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

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

Figure 17. Example of Ordinary and Macro Comment Statements

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

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



Example 1: Define General Register Equates

- 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                  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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-19

Example 1: Define Equated Symbols for Registers

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

```
         MACRO                        (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
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)
```

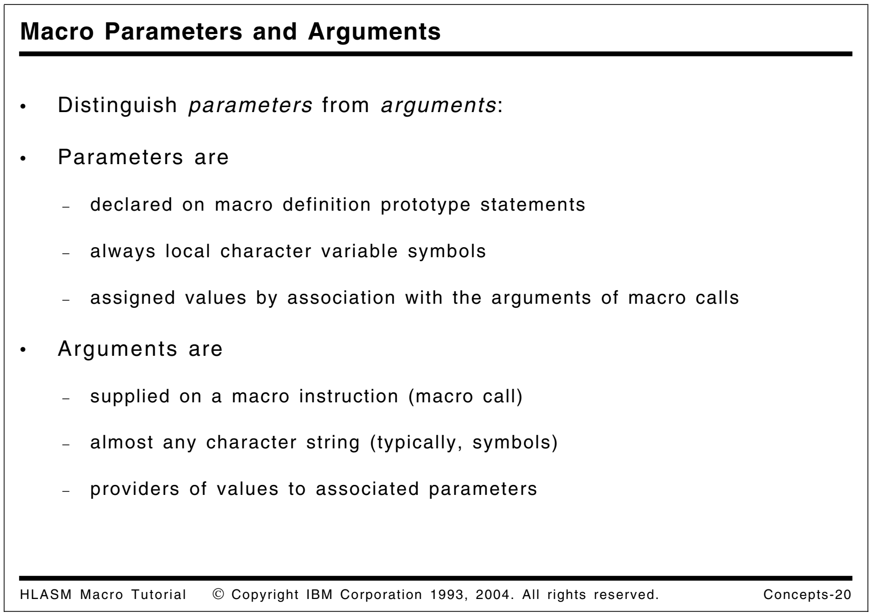
Figure 18. Simple Macro to Generate Register Equates

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

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

```
         MACRO
         GREGS
         LCLA  &N            Define a counter variable
.X       ANOP                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 19. Macro to Generate Register Equates Differently



Macro Parameters and Arguments

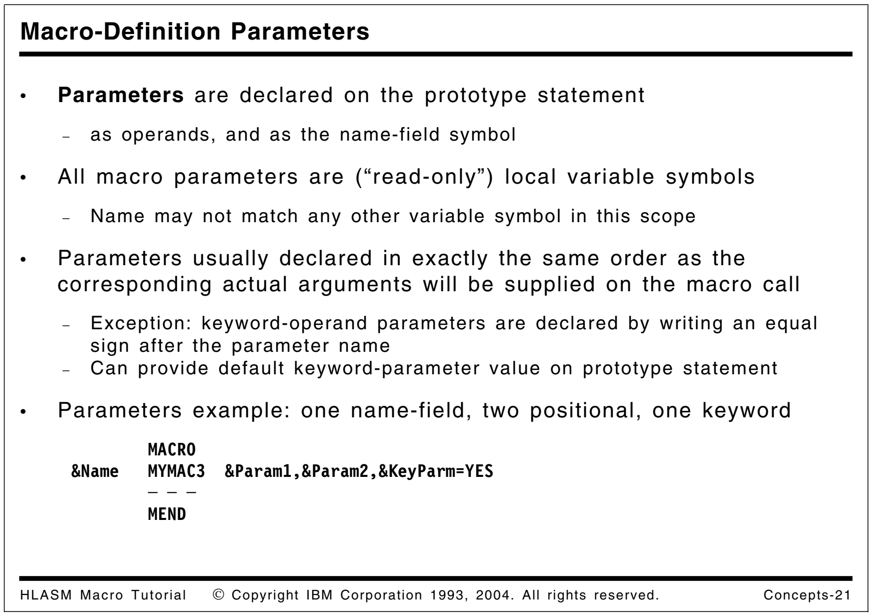
- Distinguish *parameters* from *arguments*:
- Parameters are
  - declared on macro definition prototype statements
  - always local character variable symbols
  - assigned values by association with the arguments of macro calls
- Arguments are
  - supplied on a macro instruction (macro call)
  - almost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

HLASM Macro Tutorial Concepts-20

Macro Parameters and Arguments

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

- Parameters are
  - declared on the prototype statements of macro definitions
  - always local 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** are declared on the prototype statement
  - as operands, and as the name-field symbol
- All macro parameters are ("read-only") local variable symbols
  - Name may not match any other variable symbol in this scope
- Parameters usually declared in exactly the same order as the corresponding actual arguments will be supplied on the macro call
  - Exception: keyword-operand parameters are declared by writing an equal sign after the parameter name
  - Can provide default keyword-parameter value on prototype statement
- Parameters example: one name-field, two positional, one keyword

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-21

Macro-Definition Parameters

The parameters in a macro definition are declared by virtue of their appearing as operands (and the name-field symbol) on the prototype statement. These declared parameters *are* variable symbols! (However, they cannot be assigned a value in the body of the macro; the value is assigned by *association* when the macro is called, as described in "Macro Parameter-Argument Association" on page 71) Usually, they are declared in exactly the same order as the corresponding actual arguments will be supplied on the macro call.

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

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

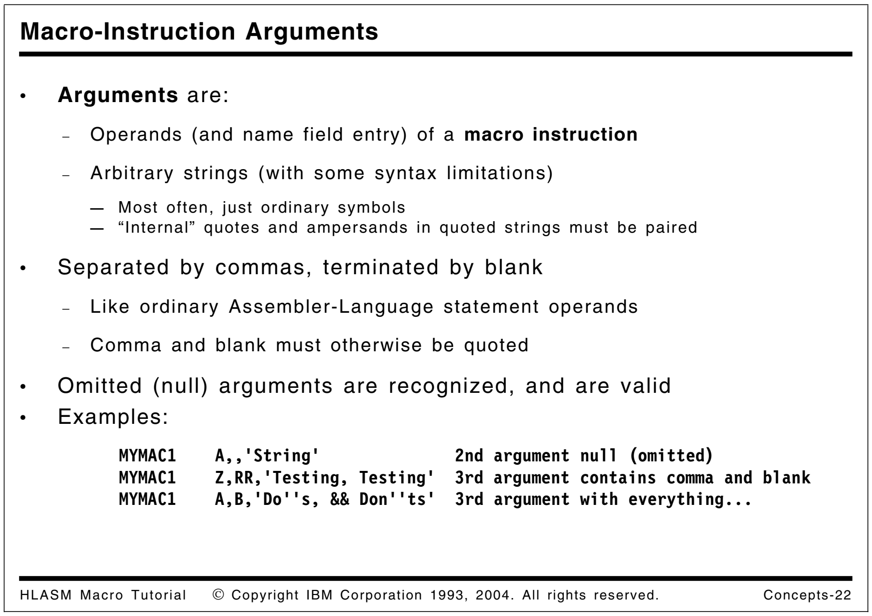
For example, suppose we write a macro prototype statement as shown in Figure 20:

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

Figure 20. Sample Macro Prototype Statement

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

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



Macro-Instruction Arguments

- **Arguments** are:
  - Operands (and name field entry) of a **macro instruction**
  - Arbitrary strings (with some syntax limitations)
    - Most often, just ordinary symbols
    - “Internal” quotes and ampersands in quoted strings must be paired
- Separated by commas, terminated by blank
  - Like ordinary Assembler-Language statement operands
  - Comma and blank must otherwise be quoted
- Omitted (null) arguments are recognized, and are valid
- Examples:

```
MYMAC1   A,,'String'             2nd argument null (omitted)
MYMAC1   Z,RR,'Testing, Testing'  3rd argument contains comma and blank
MYMAC1   A,B,'Do''s, && Don''ts'  3rd argument with everything...
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-22

Macro-Instruction Arguments

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

The operands are separated by commas, and terminated by a blank (conforming to the normal Assembler Language syntax rules). If 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 their positional sequence.

Omitted (null) arguments are perfectly acceptable.

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

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

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

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

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

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

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

- Three ways to associate (caller's) arguments with (definition's) parameters:
  1. by position, referenced by declared name (most common way)
  2. by position, by argument number (using &SYSLIST variable symbol)
  3. by keyword: always referenced by name, arbitrary order
     - Argument *values* supplied by writing **keyname=value**
- Example 1: (Assume prototype statement as on slide/foil Concepts-21)

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

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

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-23

Macro Parameter-Argument Association ...

- Example 2:

```
Lab2   MYMAC3   A                   Call: 1 positional argument

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

- Example 3:

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

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

  - Note: it's good practice to put positionals first, keywords last

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-24

Macro Parameter-Argument Association

There are three ways to associate arguments with parameters:

1. by position, referenced by the declared positional parameter name (this is the most usual way for macros to refer to their arguments)

2. by position and argument number (using the &SYSLIST system variable symbol, which will be discussed in “Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol” on page 84)

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.

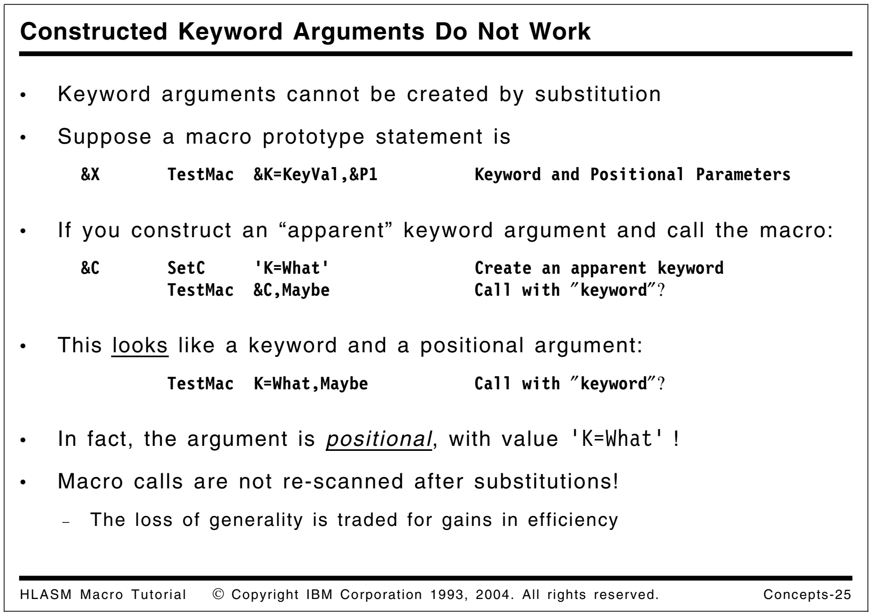
To illustrate, consider the examples in Figure 21. Assuming the same macro definition prototype statement shown in Figure 20 on page 69, the resulting values associated with the parameters are as shown:

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

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

Lab2   MYMAC3   A                    1 positional argument

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

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

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

Figure 21. Macro Parameter-Argument Association Examples

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

Mixing positional and keyword parameters and arguments is not a good practice, because it may be difficult to count the positional items correctly.



Constructed Keyword Arguments Do Not Work

- 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″?
```

- 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-25

[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!

Constructed Keyword Arguments

It is sometimes tempting to construct argument lists for macro calls, particularly 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:

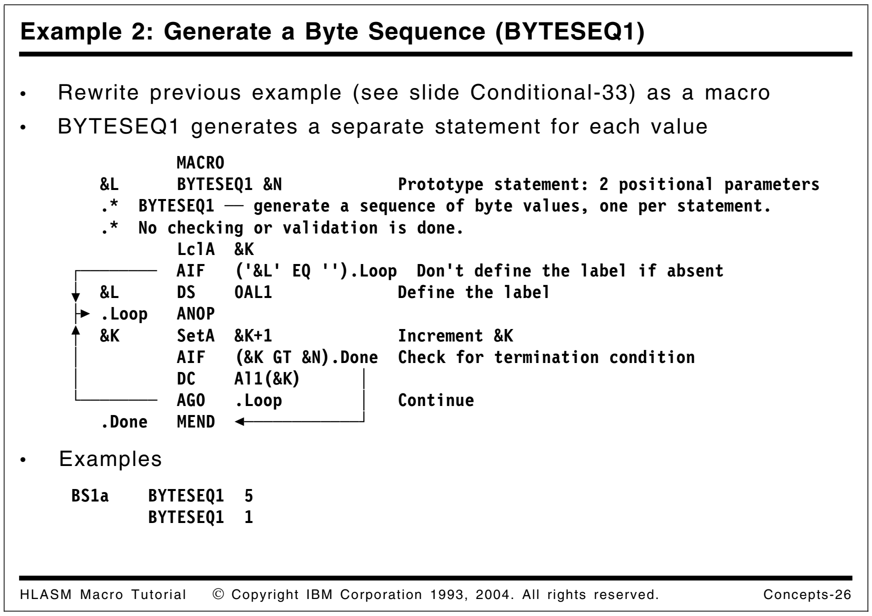
```
&C       SetC     'K=What'              Create an apparent keyword
         TestMac  &C,Maybe              Call with ″keyword″?
```

While this appears to be a properly formed keyword argument K=What, it is in fact 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 22 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 22. Example of a Substituted (Apparent) Keyword Argument

The original design of the System/360/370/390 assemblers focused on efficient macro expansion, so macro calls containing substitutions were not re-scanned.



Example 2: Generate a Byte Sequence (BYTESEQ1)

- Rewrite previous example (see slide Conditional-33) as a macro
- BYTESEQ1 generates a separate statement for each value

```
         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    OAL1             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

```
BS1a    BYTESEQ1  5
        BYTESEQ1  1
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-26

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 6 on page 44. The pseudo-code for the BYTESEQ1 macro is quite simple:

```
IF (name-field label is present) GEN(label DS OAL1)
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    OAL1             Define the label
.Loop   ANOP
&K      SetA  &K+1             Increment &K
        AIF   (&K GT &N).Done  Check for termination condition
        DC    Al1(&K)
        AGO   .Loop            Continue
.Done   MEND
* Two test cases
BS1a    BYTESEQ1  5
        BYTESEQ1  1
```

Figure 23. Macro to Define a Sequence of Byte Values

Macro Parameter Usage in Model Statements

- Parameter values supplied by arguments in the macro instruction ("call") are substituted as character strings
- Parameters may be substituted in name, operation, and operand fields of model statements
  - Substitutions ignored in remarks fields and comment statements
    - Can sometimes play tricks with operand fields containing blanks
    - AINSERT lets you generate fully substituted statements
- Some limitations on which opcodes may be substituted in conditional assembly statements
  - Can't substitute ACTR, AGO, AIF, ANOP, AREAD, COPY, GBLx, ICTL, LCLx, MACRO, MEND, MEXIT, REPRO, SETx, SETxF
  - The assembler must understand basic macro structures at the time it encodes the macro!
- Implementation trade-off: generation speed vs. generality

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-27

Macro Parameter Usage in Model Statements

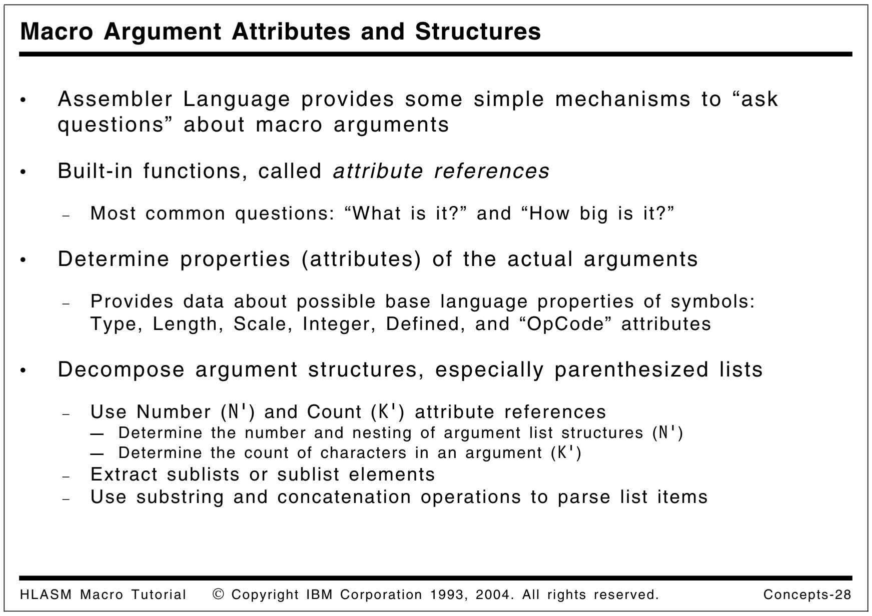
Values are assigned to macro parameters from the corresponding arguments on the macro-instruction statement, either by position in left-to-right order (for positional arguments), or by name (for keyword arguments). These are then substituted as character strings into model statements (wherever points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

- name field
- operation field
- operand field

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

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

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 macroprocessors or preprocessors, the choice seemed reasonable.



Macro Argument Attributes and Structures

- Assembler Language provides some simple mechanisms to “ask questions” about macro arguments
- Built-in functions, called *attribute references*
  - Most common questions: “What is it?” and “How big is it?”
- Determine properties (attributes) of the actual arguments
  - Provides data about possible base language properties of symbols: Type, Length, Scale, Integer, Defined, and “OpCode” attributes
- Decompose argument structures, especially parenthesized lists
  - Use Number (N') and Count (K') attribute references
    - Determine the number and nesting of argument list structures (N')
    - Determine the count of characters in an argument (K')
  - Extract sublists or sublist elements
  - Use substring and concatenation operations to parse list items

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-28

Macro Argument Attributes and Structures

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

Three major classes of attribute inquiry facilities are provided:

1. The “mechanical” or “physical” characteristics of macro arguments can be determined by using

   - two attribute references:

     - Count (K') supplies the actual count of characters in the argument, and

     - Number (N') tells you how many elements appear in an argument list structure (it can also provide the largest subscript assigned to a dimensioned variable symbol, as described in “Declaring Variable Symbols” on page 8)

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

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

     - determine the number and nesting of all such list structures

     - extract any sublists or sublist elements

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

     List structures and techniques for scanning them are described at “Macro-Instruction Argument Lists and Sublists” on page 82.

2. The type attribute reference (T') allows you to ask "What base-language meaning is attached to it?" about a macro argument. The value of the type attribute reference[6] 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 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'). All four have numeric values.

4. 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.

There is an important difference between the number (N') and count (K') attributes and all the others: N' and K' treat their operands as strings of characters, independent of any meaning that might be associated with the strings. Thus, if the value of a parameter &X is the five characters (A,9), then K'&X is 5 and N'&X is 2.

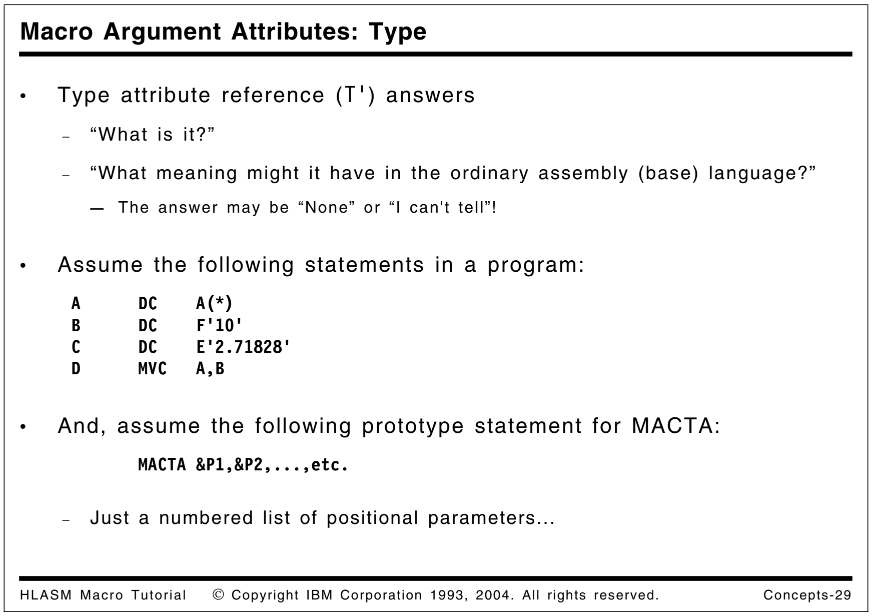
The other attribute references probe more deeply into the possible meanings of a parameter. Thus, T'&X(1) would test what the character A might designate: if A is a label on a constant, T'&X(1) would return information about the type of the constant. If the type attribute is indeed that of a constant, 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 in contexts where such terms are valid.

One way to think of this difference is that N' and K' only look at the character value of a parameter, while the other attribute references look one level deeper into the possible meanings of the character value of a parameter.

Attribute references of attribute references (such as K'L'&X) are not allowed.

Summary information about attributes is described at "Summary of Attribute References" on page 88.

[6] A single character; only the opcode (O') and type (T') attribute references have character values. All the others have numeric attributes.



Macro Argument Attributes: Type

- Type attribute reference (T') answers
  - “What is it?”
  - “What meaning might it have in the ordinary assembly (base) language?”
    - — The answer may be “None” or “I can't tell”!
- Assume the following statements in a program:

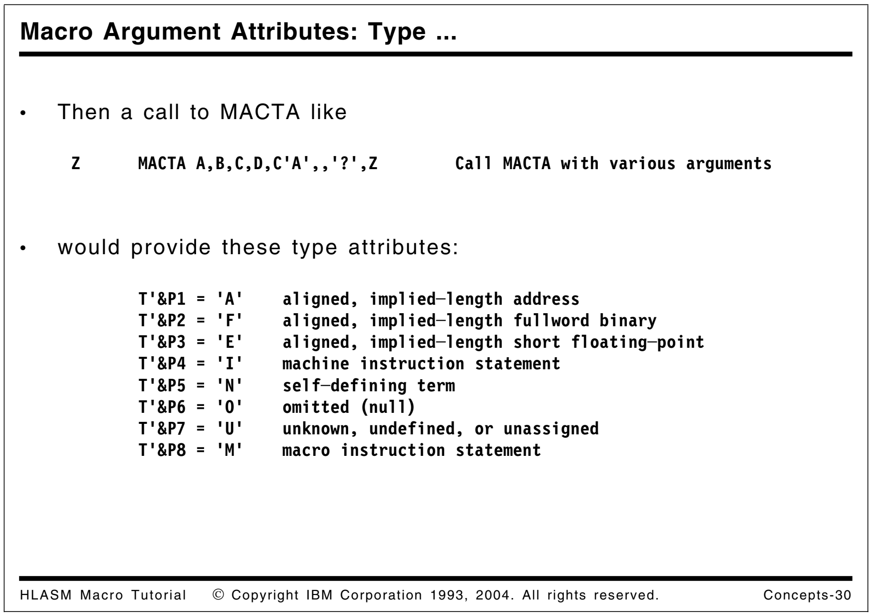
```
A      DC    A(*)
B      DC    F'10'
C      DC    E'2.71828'
D      MVC   A,B
```

- And, assume the following prototype statement for MACTA:

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

  - Just a numbered list of positional parameters...

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-29



Macro Argument Attributes: Type ...

- Then a call to MACTA like

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

- would provide these type attributes:

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-30

Macro-Instruction Argument Properties: Type Attribute

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

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

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

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

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

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

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

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

Length, Integer, and Scale Attributes

The Length (L') attribute of a symbol refers to the explicit length or the number of bytes (for implicit lengths) 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.

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

For symbols naming constants defined by DC and DS statements, the Integer (I'), and Scale (S') attributes are 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 (e.g., 31 or 15) minus the Scale attribute.

```
FullW   DC    FS6'1.3'          S'FullW = 6, I'FullW = 25
HalfW   DC    HS7'98.765432'    S'HalfW = 7, I'HalfW = 8
```

- For hexadecimal floating point constants of types E, D, and L, the Scale attribute is the number of denormalizing 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.

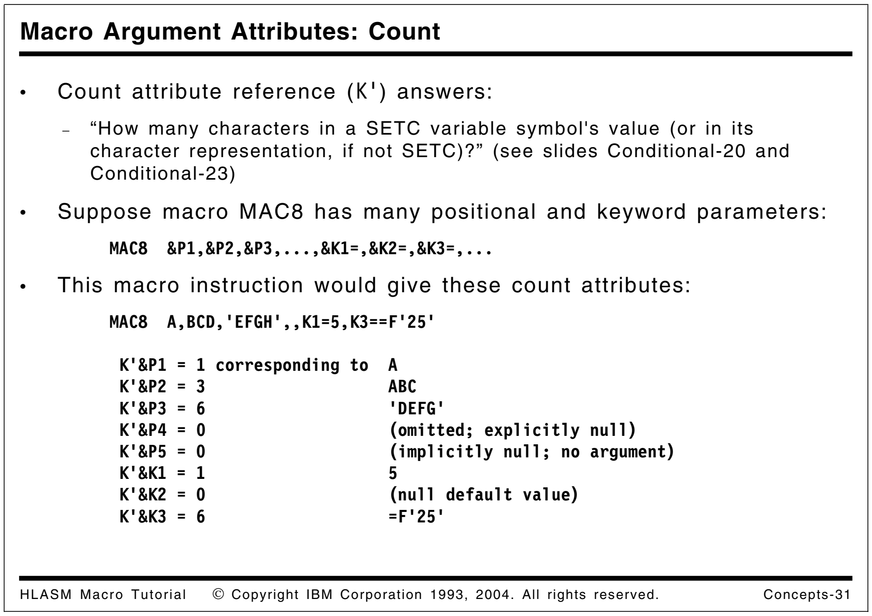
The Length, Integer, and Scale attributes are used much less often in conditional than in ordinary assembly statements.

Defined Attribute

Sometimes a macro will need to define a symbol that may have been previously defined by another macro. The D' (“Defined”) attribute is 1 if the symbol is known to the assembler, and 0 otherwise.

```
          AIF   (D'&Symbol eq 1).Known
 &Symbol  DS    A
 .Known   ANOP  ,
```

A symbol may not have been encountered in the text of the program where its Defined attribute is tested, but the assembler will enter Lookahead Mode to search for it and enter it into the symbol table if found. Thus, the value of the D' attribute is not strictly dependent on the order of statements in the program.



Macro Argument Attributes: Count

- Count attribute reference (K') answers:
  - “How many characters in a SETC variable symbol's value (or in its character representation, if not SETC)?” (see slides Conditional-20 and Conditional-23)
- 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 Concepts-31

Macro-Instruction Argument Properties: Count Attribute

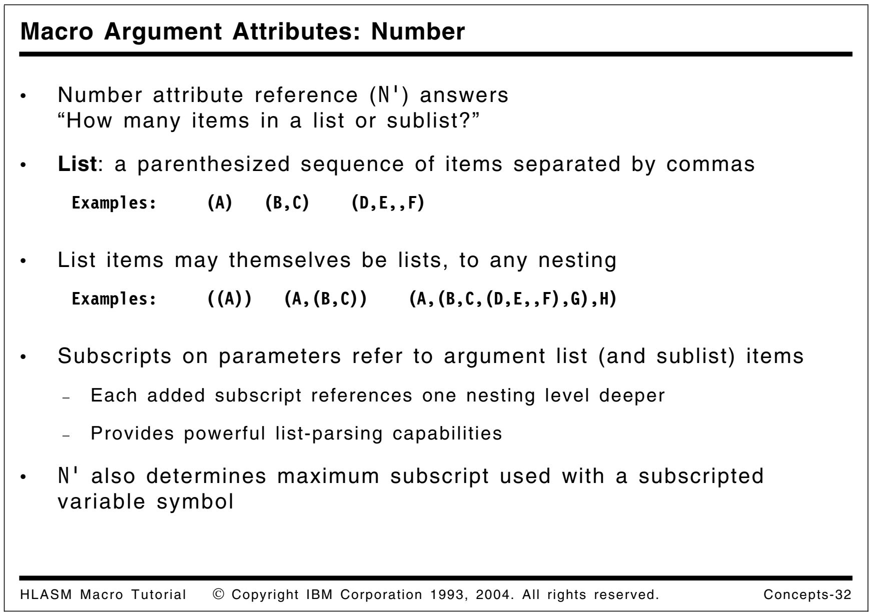
A macro argument has one irreducible, inherent property: the count of the number of characters it contains. These can be determined for any argument using the count attribute reference, K'. For example, if MAC8 has positional parameters &P1, &P2, ..., etc., and keyword parameters &K1, &K2, ..., etc.,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

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

we would find that

```
K'&P1 = 1  corresponding to   A
K'&P2 = 3                     ABC
K'&P3 = 6                     'DEFG'
K'&P4 = 0                     (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*.



Macro Argument Attributes: Number

- 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)**
- List items may themselves be lists, to any nesting

  Examples: **((A))** **(A,(B,C))** **(A,(B,C,(D,E,,F),G),H)**
- Subscripts on parameters refer to argument list (and sublist) items
  - Each added subscript references one nesting level deeper
  - Provides powerful list-parsing capabilities
- N' also determines maximum subscript used with a subscripted variable symbol

HLASM Macro Tutorial Concepts-32

Macro-Instruction Argument Properties: Number Attribute

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

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

Figure 24. Macro Argument List Structures

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

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

Figure 25. 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 255 characters.

The number attribute reference (N') is used to determine the number of elements in a list or sublist, or the number of elements in a subscripted variable symbol. For example, if the three lists in Figure 24 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 17
```

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

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

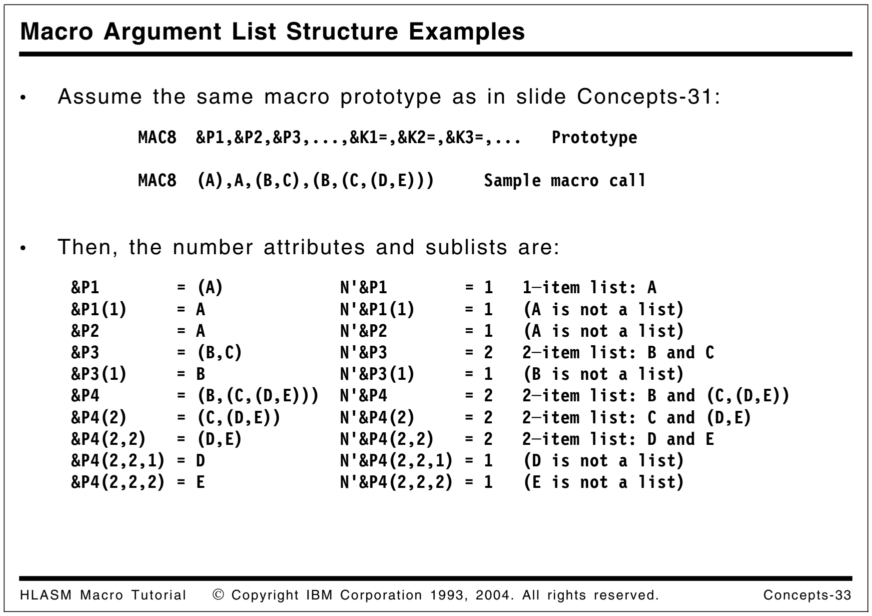
has two arguments, the first “obviously” a list with one item. However, the number attributes and sublists are:

```
  &P1     = (A)      N'&P1     = 1   1-item list: A
  &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. The following “rules of thumb” may help in understanding 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 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.

List and sublist structures provide a convenient way to pass multiple values as a single argument.



Macro Argument List Structure Examples

- Assume the same macro prototype as in slide Concepts-31:

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

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

- Then, the number attributes and sublists are:

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

HLASM Macro Tutorial Concepts-33

Macro-Instruction Argument Lists and Sublists

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

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

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

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

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

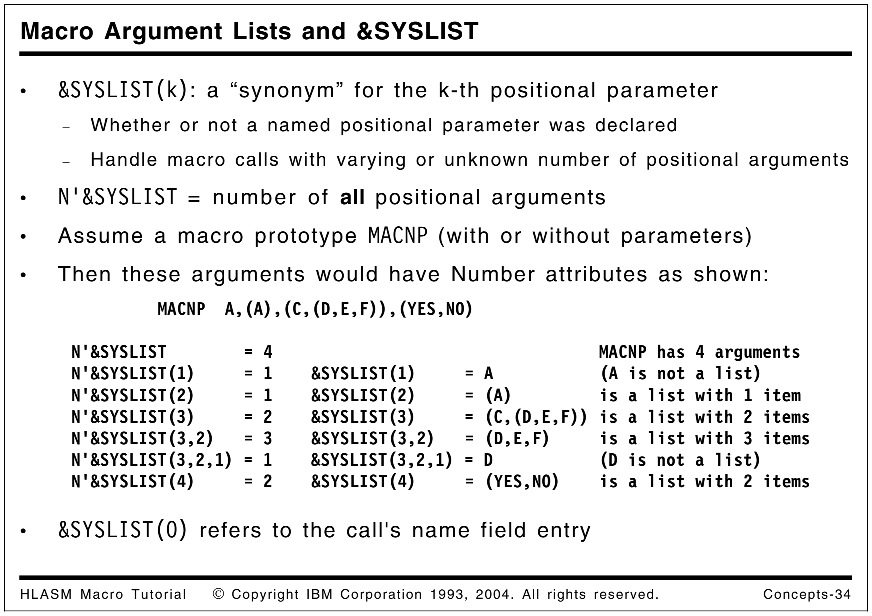
```
         MAC8   (A),A,(B,C),(B,(C,(D,E)))     Sample macro call

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

There is an oddity in the assembler's interpretation of the number attribute for items which are not themselves lists. As can be seen from the first two samples above, both '(A)' and 'A' return a number attribute of 1. The assembler will treat parameter references &P and &P(1) as the same string if the argument corresponding to &P is not a properly formed list. This means that if it is important to know whether or not a list item is in fact a parenthesized list, you will need to test the first *and* last characters to verify that the list is properly enclosed in parentheses. (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.)

In practice, it often is not a problem if a single item is or is not enclosed in parentheses (depending on where the argument is substituted). For example,

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

will be processed the same way by the assembler.



Macro Argument Lists and &SYSLIST

- &SYSLIST(k): a “synonym” for the k-th positional parameter
  - Whether or not a named positional parameter was declared
  - Handle macro calls with varying or unknown number of positional arguments
- N'&SYSLIST = number of **all** positional arguments
- Assume a macro prototype MACNP (with or without parameters)
- Then these arguments would have Number attributes as shown:

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

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

- &SYSLIST(0) refers to the call's name field entry

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-34

Macro-Instruction Argument Lists and the &SYSLIST Variable Symbol

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

The system variable symbol &SYSLIST can be used to refer to the positional elements of the argument list: &SYSLIST(**k**) refers to the **k**-th positional argument, whether or not a corresponding positional parameter was declared on the macro's prototype statement. &SYSLIST(0) refers to the entry in the name field of the macro call (which of course need not be present). The total number of *positional* arguments in the macro instruction's operand list can be determined using a Number attribute reference: N'&SYSLIST is the number of positional arguments.

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

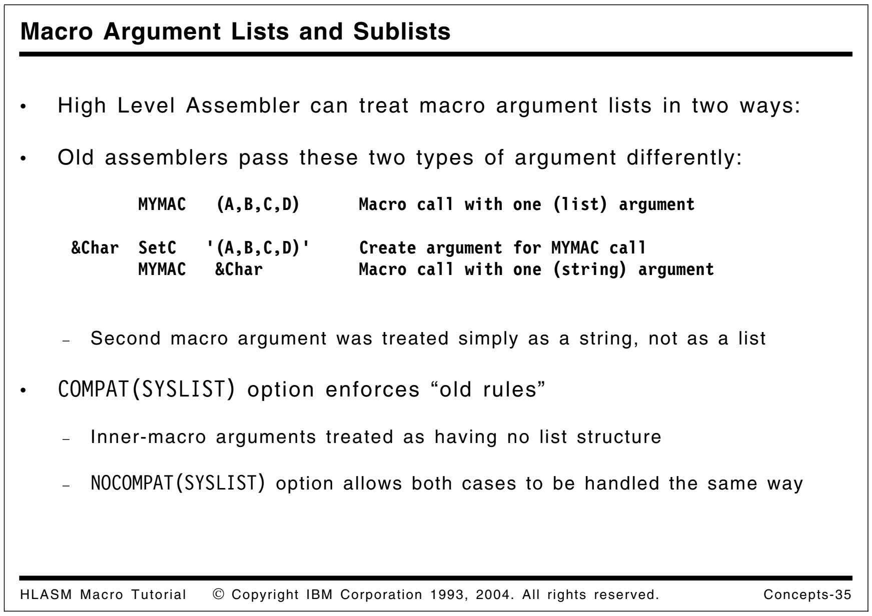
To illustrate the use of &SYSLIST references, suppose we have defined a macro named MACNP; whether or not any positional parameters are declared doesn't matter for this example. If we write the following macro call:

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

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

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

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



Macro Argument Lists and Sublists

- High Level Assembler can treat macro argument lists in two ways:
- Old assemblers pass these two types of argument differently:

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

 &Char  SetC   '(A,B,C,D)'     Create argument for MYMAC call
        MYMAC   &Char          Macro call with one (string) argument
```

  - Second macro argument was treated simply as a string, not as a list
- COMPAT(SYSLIST) option enforces “old rules”
  - Inner-macro arguments treated as having no list structure
  - NOCOMPAT(SYSLIST) option allows both cases to be handled the same way

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-35

Macro Argument Lists and Sublists

There can be differences in the handling of *lists* of arguments passed to macros, depending on setting of the COMPAT(SYSLIST) option. While this is rarely a concern, there are situations where your macros can be written much 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 *positional* list of four arguments (A, B, C, and D) appears in the call

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

and these may be treated as a list through references in the macro to 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 (list) argument
```

where the argument (A,B,C,D) is a list of four items. We will discuss only the second of these forms, where an argument is itself 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
```

In this case, 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 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. However, they can be treated quite differently, depending on High Level Assembler's option settings. For example: suppose you want to write a macro with positional operands that will pass some number of those operands to an inner macro. This can be done by constructing a list for the inner macro. Let an outer macro TOPMAC be called with a variable number of of arguments:

```
         TOPMAC  X,Y,Z,...
```

and you wish to use some or all of the items in this varying-length list to create *another* varying-length list to be passed to an inner macro BOTMAC.

To construct the varying-length argument list for the inner macro, build a string with the arguments:

```
&ARG     SetC  '&ARG'.'&SysList(&N)'   Get &N-th argument
```

for as many arguments as necessary (so long as &ARG will not exceed 253 characters).

Then, add parentheses:

```
&ARGLIST SetC  '(&ARG.)'
```

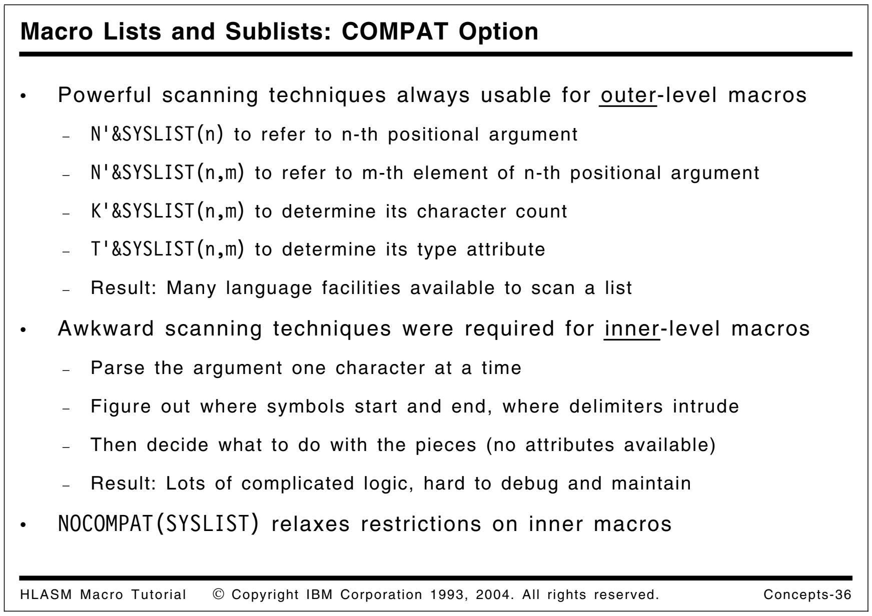
and call the inner macro:

```
         BOTMAC  &ARGLIST
```

The inner macro will then be able to scan the list using notations like &SYSLIST(&N,1). Note that calling the inner macro with

```
         BOTMAC  (&ARG)
```

only passes the complete (unstructured, parenthesized) string, which will not be recognizable as a list by the inner macro.



Macro Lists and Sublists: COMPAT Option

- Powerful scanning techniques always usable for outer-level macros
  - N'&SYSLIST(n) to refer to n-th positional argument
  - N'&SYSLIST(n,m) to refer to m-th element of n-th positional argument
  - K'&SYSLIST(n,m) to determine its character count
  - T'&SYSLIST(n,m) to determine its type attribute
  - Result: Many language facilities available to scan a list
- Awkward scanning techniques were required for inner-level macros
  - Parse the argument one character at a time
  - Figure out where symbols start and end, where delimiters intrude
  - Then decide what to do with the pieces (no attributes available)
  - Result: Lots of complicated logic, hard to debug and maintain
- NOCOMPAT(SYSLIST) relaxes restrictions on inner macros

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-36

Control of Macro Argument Sublists

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 generally recognized no list structure, even if it is present (as in example 3 above). 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 and macro language such as number attribute references (N') and subscripted &SYSLIST notation cannot be used be used to “parse” the operand to extract individual list elements.

For example, if (S,T,U) is the argument to INNER, you might have wanted to write statements like

```
         Macro
&Label   INNER  &Arg
&NItems  SetA   N'&Arg        Determine number of list elements in &ARG
         - - -
.*       Do something to each of the list elements in turn
&Temp    SetC   '&Arg(&ArgNum)'  Extract a list item into &Temp
```

If you specify the COMPAT(SYSLIST) option, the argument string providing the value of &Arg must be scanned 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; instead, substituted arguments to inner macros will be 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. This means that the full power of the &SYSLIST notation, sublist notations, and number attributes are available.

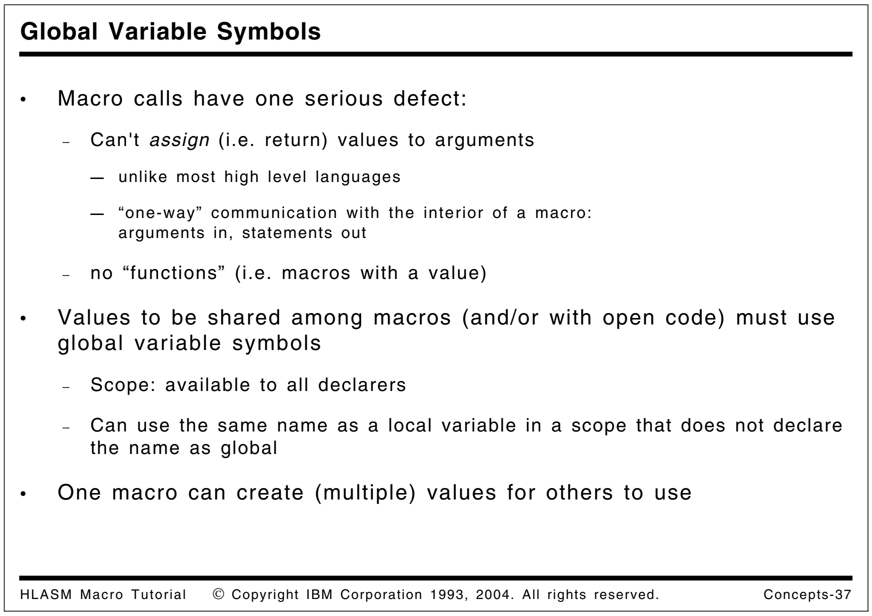
Summary of Attribute References

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 limitations on their use depend on the type of the value of the reference.

Figure 26. Attribute Usage in the Base and Conditional Languages

			Conditional Language	
Attr.	Val.	Base 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	Not allowed	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:
(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

- Macro calls have one serious defect:
  - Can't *assign* (i.e. return) values to arguments
    - unlike most high level languages
    - “one-way” communication with the interior of a macro: arguments in, statements out
  - no “functions” (i.e. macros with a value)
- Values to be shared among macros (and/or with open code) must use global variable symbols
  - Scope: available to all declarers
  - Can use the same name as a local variable in a scope that does not declare the name as global
- One macro can create (multiple) values for others to use

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-37

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.

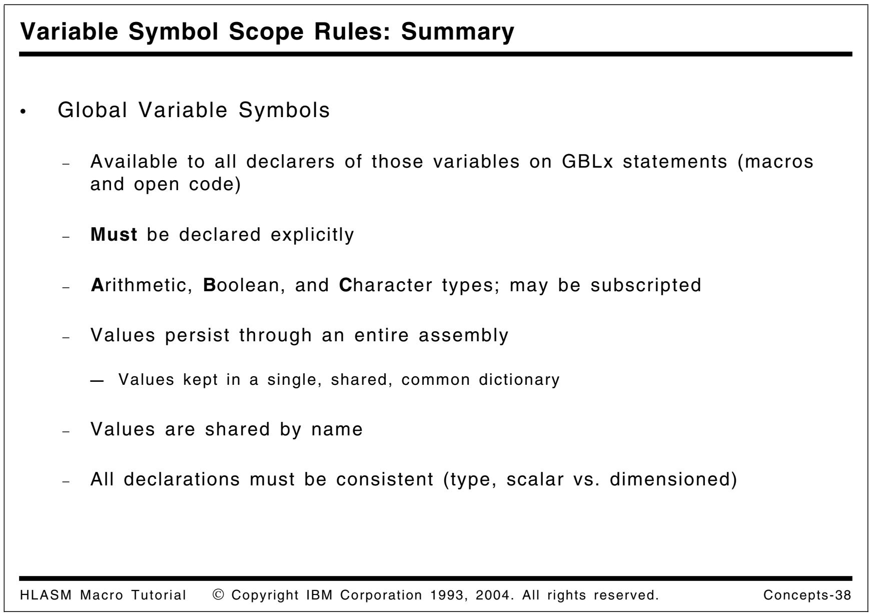
In the System/360/370/390 Assembler Language, macro calls have one serious omission: they can't *assign* (i.e. return) values to arguments, unlike most high level languages. That is, all macro arguments are “input only”. Thus, communication with the interior of a macro by way of the argument list appears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

Furthermore, there is no provision for defining macros which act as “functions” (that is, 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).

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

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

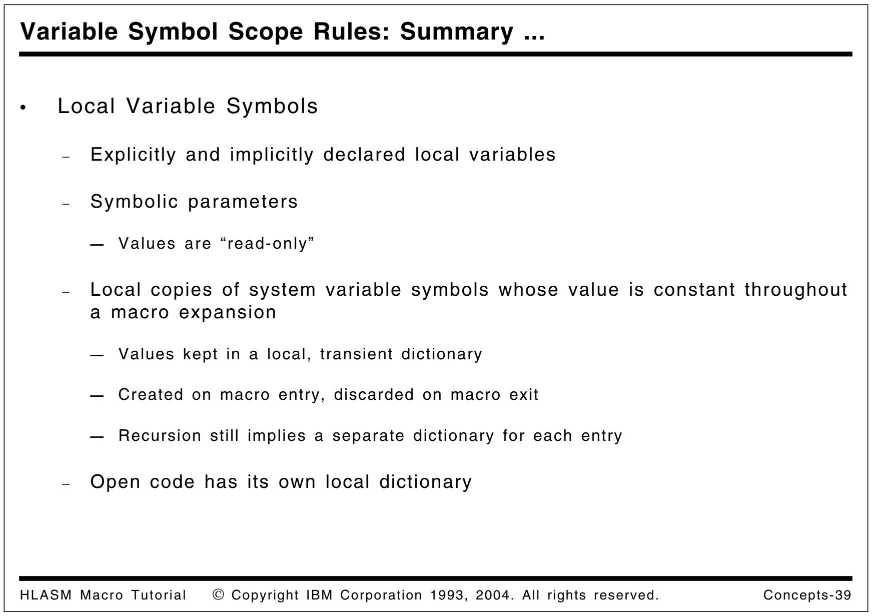
The “dictionary” or “pool” of global symbols has similar behavior to certain kinds of external variables in high level languages, such as Fortran COMMON: all declarers of variables in COMMON may refer to them. However, the assembler imposes no conformance rules of ordering or size on declared global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



Variable Symbol Scope Rules: Summary

- Global Variable Symbols
  - Available to all declarers of those variables on GBLx statements (macros and open code)
  - **Must** be declared explicitly
  - **A**rithmetic, **B**oolean, and **C**haracter types; may be subscripted
  - Values persist through an entire assembly
    - Values kept in a single, shared, common dictionary
  - Values are shared by name
  - All declarations must be consistent (type, scalar vs. dimensioned)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-38



Variable Symbol Scope Rules: Summary ...

- Local Variable Symbols
  - Explicitly and implicitly declared local variables
  - Symbolic parameters
    - Values are “read-only”
  - Local copies of system variable symbols whose value is constant throughout a macro expansion
    - Values kept in a local, transient dictionary
    - Created on macro entry, discarded on macro exit
    - Recursion still implies a separate dictionary for each entry
  - Open code has its own local dictionary

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-39

Variable Symbol Scope Rules: Summary

At this point, we will 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 local copies of system variable symbols whose value is constant throughout a macro expansion. They are not shared with other macros, or with open code (and vice versa). Open code has its own local dictionary, which is active throughout

an assembly. Local variable symbols may be referenced or set only in their local context.

- Variable symbol values for macros are kept in a local, transient 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.
- System variable symbols and parameters are treated as “read-only”, meaning that their values are constant throughout a macro invocation, and cannot be changed.

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

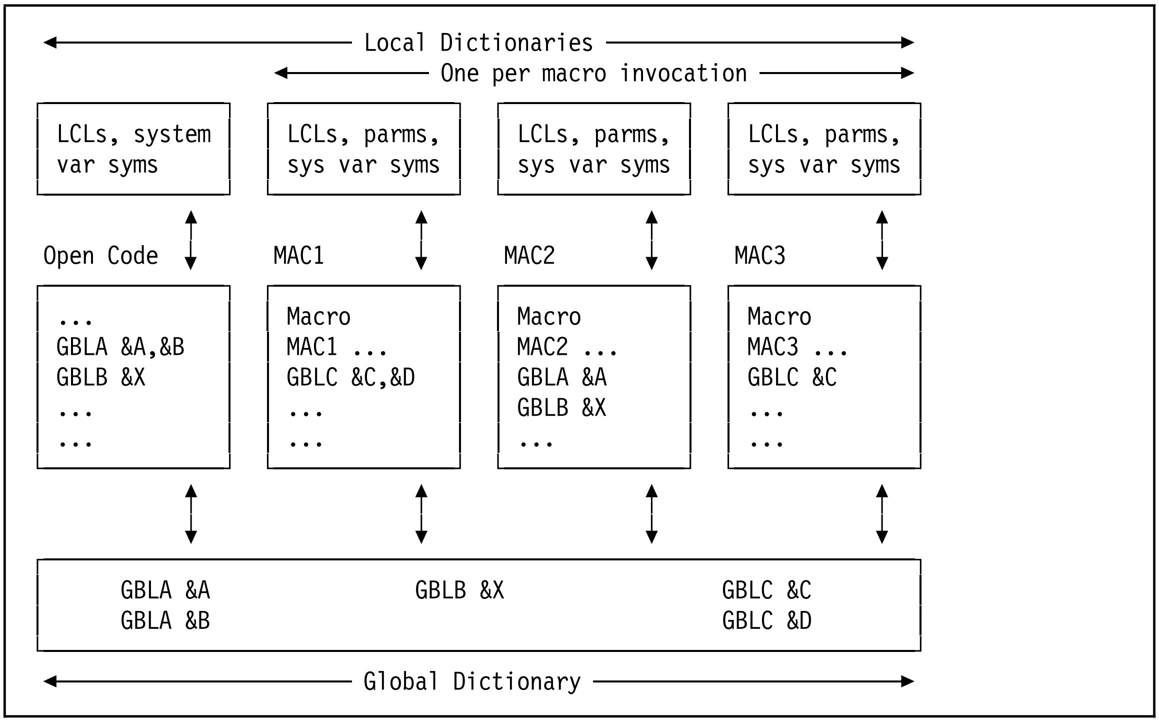
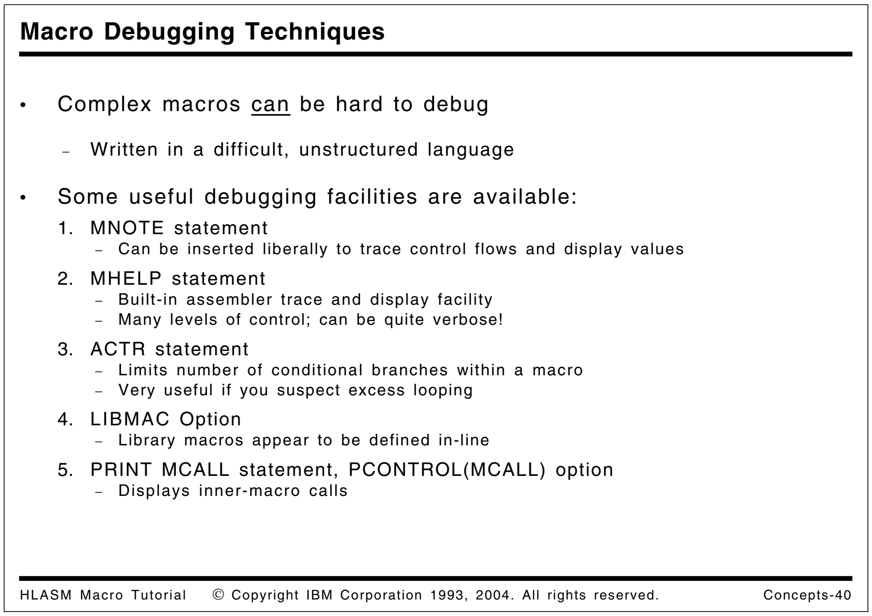

Local Dictionaries
One per macro invocation
LCLs, system var syms
LCLs, parms, sys var syms
LCLs, parms, sys var syms
LCLs, parms, sys var syms
Open Code
MAC1
MAC2
MAC3
... GBLA &A,&B GBLB &X ... ...
Macro MAC1 ... GBLC &C,&D ... ...
Macro MAC2 ... GBLA &A GBLB &X ...
Macro MAC3 ... GBLC &C ... ...
GBLA &A GBLA &B
GBLB &X
GBLC &C GBLC &D
Global Dictionary

Figure 27. 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; for example, only open code and macro MAC2 may refer to the GLBL symbol &X.



Macro Debugging Techniques

- Complex macros can be hard to debug
  - Written in a difficult, unstructured 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; 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

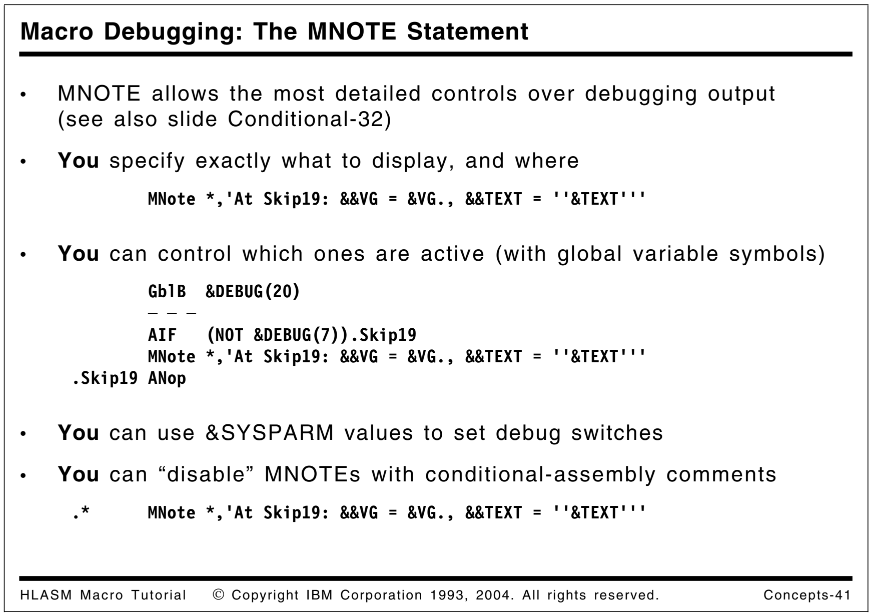
HLASM Macro Tutorial Concepts-40

Macro Debugging Techniques

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

We will briefly describe 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

- MNOTE allows the most detailed controls over debugging output (see also slide Conditional-32)
- **You** specify exactly what to display, and where

```
         MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
```

- **You** can control which ones are active (with global variable symbols)

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

- **You** can use &SYSPARM values to set debug switches
- **You** can “disable” MNOTEs with conditional-assembly comments

```
.*       MNote *,'At Skip19: &&VG = &VG., &&TEXT = ''&TEXT'''
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-41

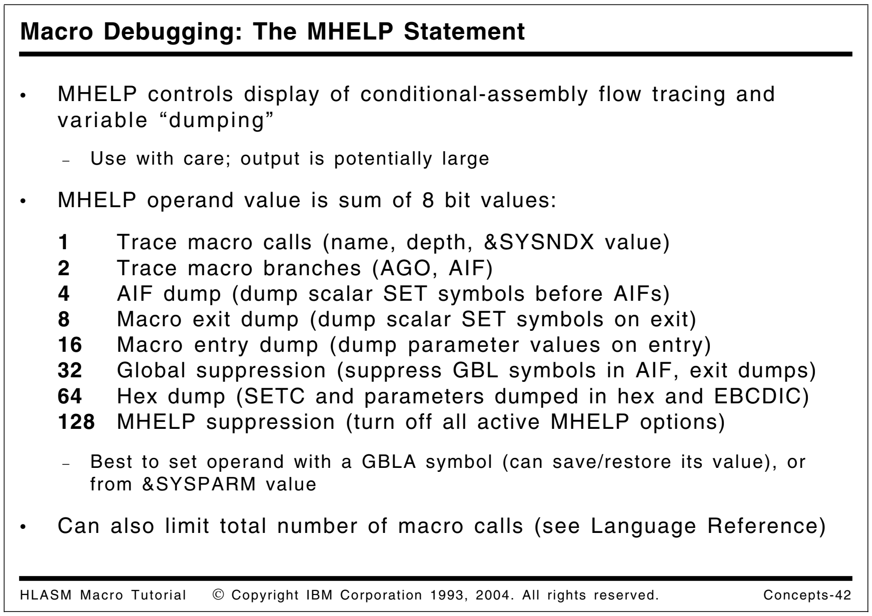
Macro Debugging: The MNOTE Statement

We have already touched on the use of MNOTE statements in “Displaying Variable Symbol Values: The MNOTE Statement” on page 42. Their main benefits for debugging macros are:

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

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

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



Macro Debugging: The MHELP Statement

- MHELP controls display of conditional-assembly flow tracing and variable “dumping”
  - Use with care; output is potentially large
- MHELP operand value is sum of 8 bit values:

  1 Trace macro calls (name, depth, &SYSNDX value)
  2 Trace macro branches (AGO, AIF)
  4 AIF dump (dump scalar SET symbols before AIFs)
  8 Macro exit dump (dump scalar SET symbols on exit)
  16 Macro entry dump (dump parameter values on entry)
  32 Global suppression (suppress GBL symbols in AIF, exit dumps)
  64 Hex dump (SETC and parameters dumped in hex and EBCDIC)
  128 MHELP suppression (turn off all active MHELP options)

  - Best to set operand with a GBLA symbol (can save/restore its value), or from &SYSPARM value
- Can also limit total number of macro calls (see Language Reference)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-42

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 by using the LIBMAC option: if you specify LIBMAC, tracing is active for library macros.

4 AIF dump

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

8 Macro exit dump

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

16 Macro entry dump

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

32 Global suppression

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

64 Hex dump

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

128 MHELP suppression

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

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

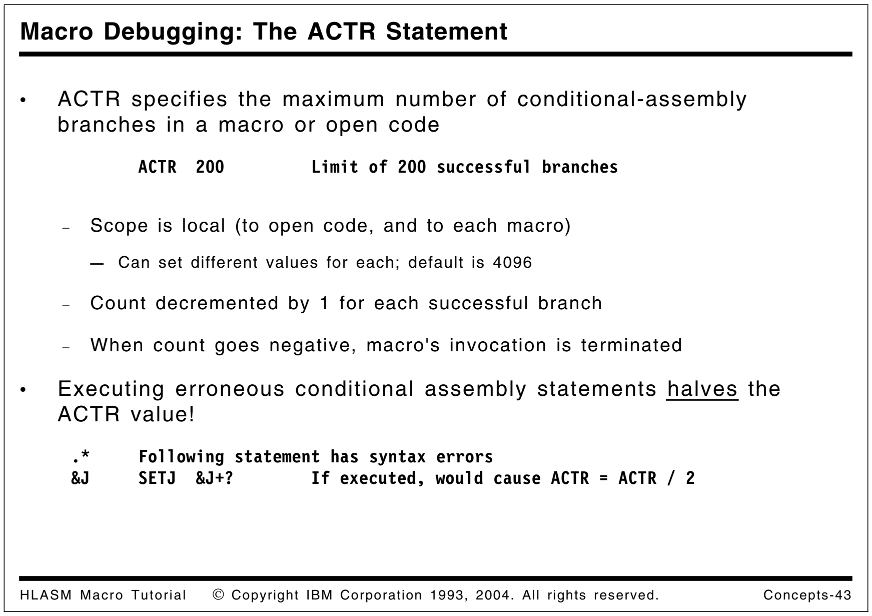
```
         MHELP  1+2             Trace macro calls and AIFs
```

will request both macro call tracing and AIF branch tracing.

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

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

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



Macro Debugging: The ACTR Statement

- ACTR specifies the maximum number of conditional-assembly branches in a macro or open code

```
         ACTR  200        Limit of 200 successful branches
```

  - Scope is local (to open code, and to each macro)
    - Can set different values for each; default is 4096
  - Count decremented by 1 for each successful branch
  - When count goes negative, macro's invocation is terminated
- Executing erroneous conditional assembly statements halves the ACTR value!

```
.*      Following statement has syntax errors
&J      SETJ  &J+?       If executed, would cause ACTR = ACTR / 2
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-43

Macro Debugging: The ACTR Statement

The ACTR statement can be 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
```

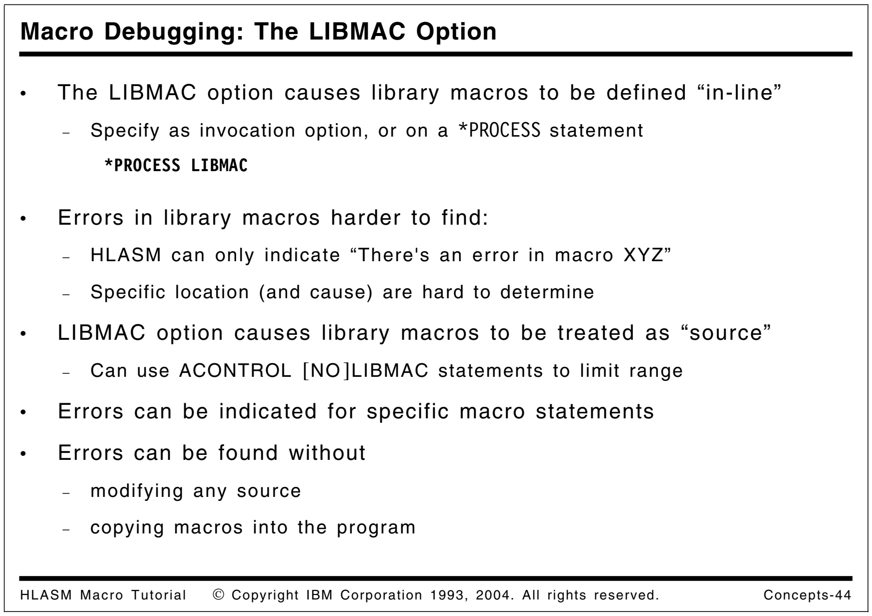
where the value of the “arithmetic_expression” will be used to set an upper limit on the number of branches executed by the assembler. In the absence of an ACTR statement, the default ACTR value is 4096, which is adequate for most macros.

ACTR is most useful in two situations:

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

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. If exceeded in a macro, the expansion is terminated; if exceeded in open code, the rest of the source program is “flushed” as comments, and is not processed. Its value can be changed within its “owning” scope by executing other ACTR statements.



Macro Debugging: The LIBMAC Option

- The LIBMAC option causes library macros to be defined “in-line”
  - Specify as invocation option, or on a *PROCESS statement

    ```
    *PROCESS LIBMAC
    ```
- Errors in library macros harder to find:
  - HLASM can only indicate “There's an error in macro XYZ”
  - Specific location (and cause) are hard to determine
- LIBMAC option causes library macros to be treated as “source”
  - Can use ACONTROL [NO]LIBMAC statements to limit range
- Errors can be indicated for specific macro statements
- Errors can be found without
  - modifying any source
  - copying macros into the program

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Concepts-44

Macro Debugging: The LIBMAC Option

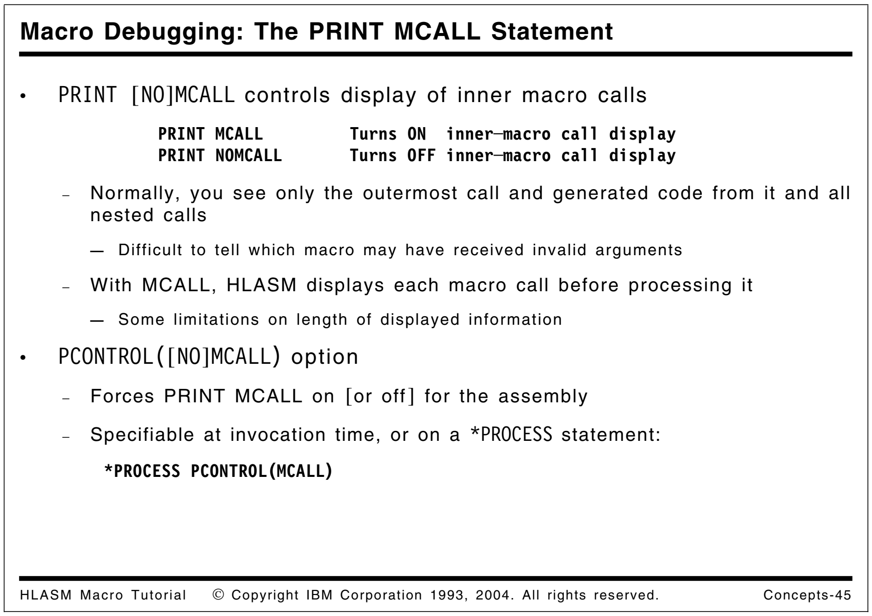
The LIBMAC assembler option can be very helpful in locating errors in macros whose definitions have been placed in a macro library. Because library macros are edited as they are read, they do not have statement numbers associated with each statement of the definition, as with “source-stream” macros.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

To help overcome this limitation, the LIBMAC option will cause the assembler to treat library macro definitions as though they were found in the primary source stream. When a macro call causes a macro definition to be brought from the library, the assembler treats all of its statements in the same way as source macros are treated; when an error condition is detected, the assembler is then able (in most cases) to supply the number of the relevant statement. This makes locating and correcting errors much easier.

If the program contains calls on many macros, but only one or two need this form of analysis, you can “bracket” the call with ACONTROL statements to limit the range of statements over 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 were 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 the macro was written and “certified”.



Macro Debugging: The PRINT MCALL Statement

- PRINT [NO]MCALL controls display of inner macro calls

```
        PRINT MCALL         Turns ON  inner—macro call display
        PRINT NOMCALL       Turns OFF inner—macro call display
```

  - Normally, you see only the outermost call and generated code from it and all nested calls
    - Difficult to tell which macro may have received invalid arguments
  - With MCALL, HLASM displays each macro call before processing it
    - Some limitations on length of displayed information
- PCONTROL([NO]MCALL) option
  - Forces PRINT MCALL on [or off] for the assembly
  - Specifiable at invocation time, or on a *PROCESS statement:

```
*PROCESS PCONTROL(MCALL)
```

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 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 High Level Assembler to display inner macro calls before they are processed; this can help in ensuring that the arguments passed to each macro in a nest have the expected values. For example, suppose you have defined two macros, OUTER and INNER:

```
         Macro
&L       OUTER  &P,&Q,&R
&T       SetC   '&P._C'
&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 displayed result in the listing will show only the call to OUTER and (if PRINT GEN is in effect) the generated DC statement. However, if PRINT MCALL is in effect, the displayed result 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 subjected to various modifications during their passage 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 may specify 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.[7] 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 macro definitions.

Please bear in mind 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.

Macros and Subroutines

You can think of a macro as an “assembly-time subroutine”. The analogy is quite close: they are both

- “named” collections of statements invoked by that name
- to which various arguments are passed
- 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), whereas a macro is executed by “software” *during* the translation (assembly) process, prior to the generation of machine code.

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

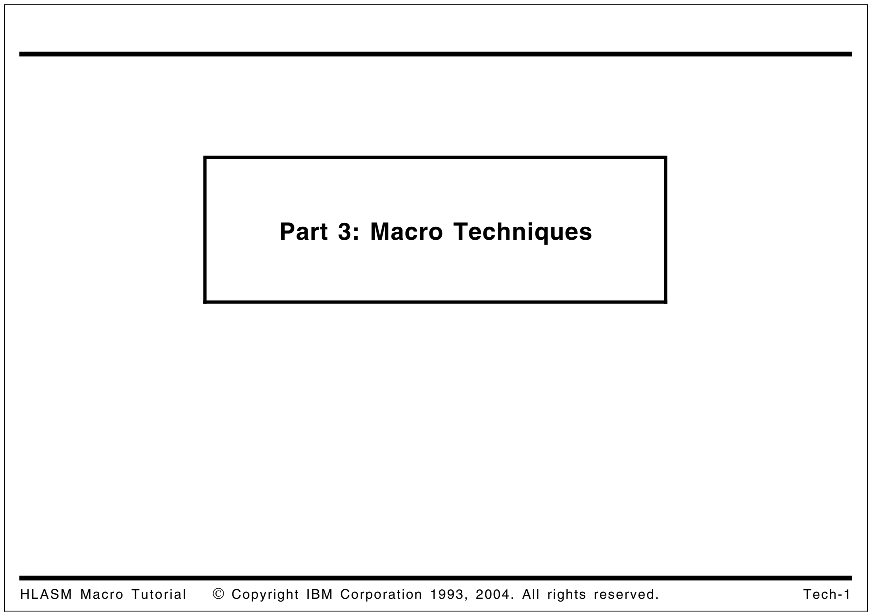
- access to attribute information about arguments
- great flexibility in specifying and processing arguments
- simple methods for managing complex argument list structures.

However, macros also have several limitations:

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

[7] 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.

Part 3: Macro Techniques



Part 3: Macro Techniques

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-1

Macro instructions (or *macros* for short) provide the Assembler Language programmer with a wonderfully flexible set of possibilities. Macros share many of the properties of ordinary subroutines (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 (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.

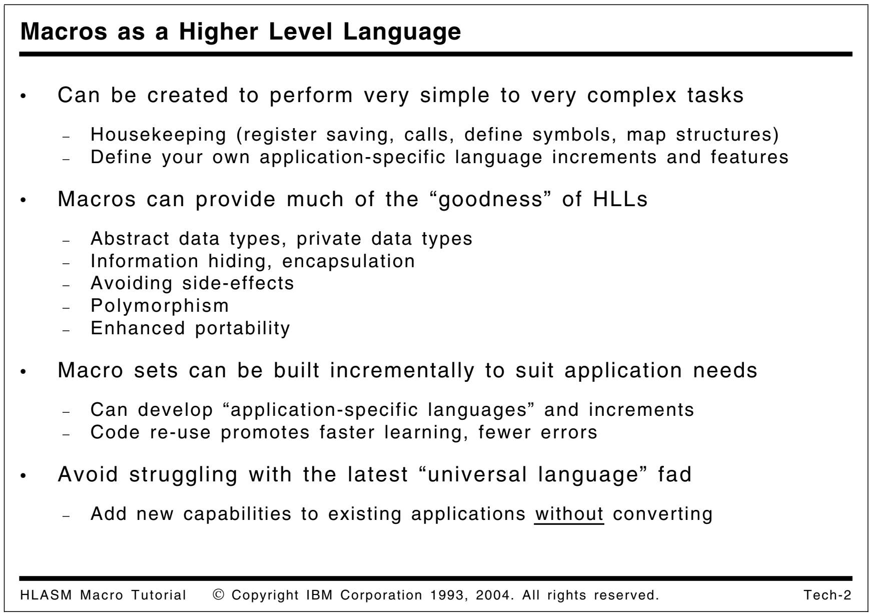
to the very complex:

- macros have been created to define “artificial languages” in which entire applications are written. Examples include the SNOBOL4[8] language; specialized compiler-writing operations[9]; and banking, marketing, and teleprocessing applications.

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

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

[9] 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.



Macros as a Higher Level Language

- Can be created to perform very simple to very complex tasks
  - Housekeeping (register saving, calls, define symbols, map structures)
  - Define your own application-specific language increments and features
- Macros can provide much of the “goodness” of HLLs
  - Abstract data types, private data types
  - Information hiding, encapsulation
  - Avoiding side-effects
  - Polymorphism
  - Enhanced portability
- Macro sets can be built incrementally to suit application needs
  - Can develop “application-specific languages” and increments
  - Code re-use promotes faster learning, fewer errors
- Avoid struggling with the latest “universal language” fad
  - Add new capabilities to existing applications <u>without</u> converting

HLASM Macro Tutorial Tech-2

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

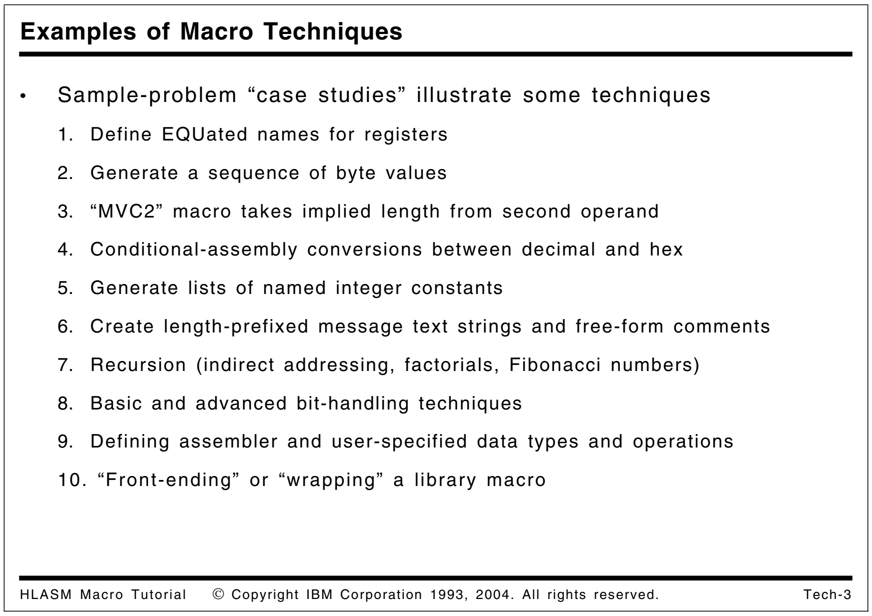
- *Abstract Data Types* — are user-specified types for data objects, and sets of procedures used to access and manipulate them. This “encapsulation” of data items and logic is one of the key benefits claimed for object-oriented programming; it is a natural consequence of macro programming.
- *Information Hiding* — is an established technique for hiding the details of an implementation from the user. The concept of separating application logic from data representations is 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 the implementing procedures are hidden.
- *Avoiding Side-Effects* — is achieved by having functions only return a value without altering either input values or setting of shared variables not declared in the invocation of an implementing procedure.
- *Polymorphism* — allows functions to accept arguments of different types, and enhances the possibility of reusing components in many contexts.

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

- Macros may be written to perform as much or as little as is needed for a particular task.
- Macros can be built incrementally, so that simple functions can be used by more complex functions, as they are written.
- New “language” implemented by macros can be adapted to the needs of the application, giving an application-specific language that may well be better suited to its needs than a general-purpose “higher level” language designed to (nearly) fit (nearly) everything. When completed, a macro can be used by everyone, giving immediate benefits of code re-use.
- Macro-based implementations can often be much more efficient than compiled code. A compiler must be prepared to accept quite 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 your view 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.

- 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 any useful and interesting language fragment easily without having to worry about extensive side-effects. Macros also allow you to investigate trade-offs involved in compile-time vs. run-time issues such as a choice between generating in-line code or calls to a run-time library.



Examples of Macro Techniques

- Sample-problem “case studies” illustrate some techniques
  1. Define EQUated names for registers
  2. Generate a sequence of byte values
  3. “MVC2” macro takes implied length from second operand
  4. Conditional-assembly conversions between decimal and hex
  5. Generate lists of named integer constants
  6. Create length-prefixed message text strings and free-form comments
  7. Recursion (indirect addressing, factorials, Fibonacci numbers)
  8. Basic and advanced bit-handling techniques
  9. Defining assembler and user-specified data types and operations
  10. “Front-ending” or “wrapping” a library macro

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-3

Macro Techniques Case Studies

We will now examine some sample macro “case studies” that illustrate various aspects of the macro language.

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

1. The example macros at “Case Study 1: Defining Equated Symbols for Registers” on page 104 generate a set of EQU statements to define symbols to be used for register references. They illustrate the use of a global variable symbol to set a “one-time” switch, text parameterization, use of the &SYSLIST system variable symbol, and created variable symbols. (This case study is a generalization of the macro discussed at “Example 1: Define Equated Symbols for Registers” on page 67.)

2. Two example macros at “Case Study 2: Generating a Sequence of Byte Values” on page 108 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 74.)

3. The standard MVC instruction takes its implied length from the length attribute of the first (target) operand. A simple “MVC2” macro at “Case Study 3: “MVC2” Macro Uses Source Operand Length” on page 111 takes its implied length from the length attribute of its *second* (source) operand.

4. The “utility” macros at “Case Study 4: Conversion Between Hex and Decimal” on page 113 might be used by other macros to perform conversions between decimal and hexadecimal representations. They illustrate construction of self-defining terms, global variables for communicating among macros, and substring operations.

5. The example macro at “Case Study 5: Generate a List of Named Integer Constants” on page 118 generates a list of constants from a varying-length list of arguments, using &SYSLIST to refer to each argument in turn, and constructs a name for each constant using its value.

6. “Case Study 6: Using the AREAD Statement” on page 122 illustrates two uses of the AREAD statement:

   a. First, the three example macros at “Case Study 6a: Creating Length-Prefixed Message Texts” on page 123 show how to generate a length-prefixed “message” string. The first and second examples illustrate some familiar techniques, while the third uses the AREAD statement and a full scan of a “human-format” message to generate an insertion-text character string for the final DC statement containing the message.

   b. Second, “Case Study 6b: Block Comments” on page 129 show how to use the AREAD statement to help you write free-form or “block” comments in your program.

7. Three example macros at “Case Study 7: Macro Recursion” on page 131 illustrate recursive macro calls. The first implements “indirect addressing”, and the remaining two illustrate the use of global variable symbols and recursive macro calls to generate factorials and Fibonacci numbers.

8. Two styles of macros illustrate techniques that can be used to define a private “bit” data type, with bit addressing by name and type checking within the bit handling macros themselves. After describing some basic bit-handling techniques, simple and optimized macros are created:

   a. “Case Study 8a: Bit-Handling Macros -- Simple Forms” on page 141 describes basic forms of declaring and using a bit data type.

   b. “Case Study 8b: Bit-Handling Macros -- Advanced Forms” on page 151 shows how to improve the basic forms for safety and efficiency, generating optimized code.

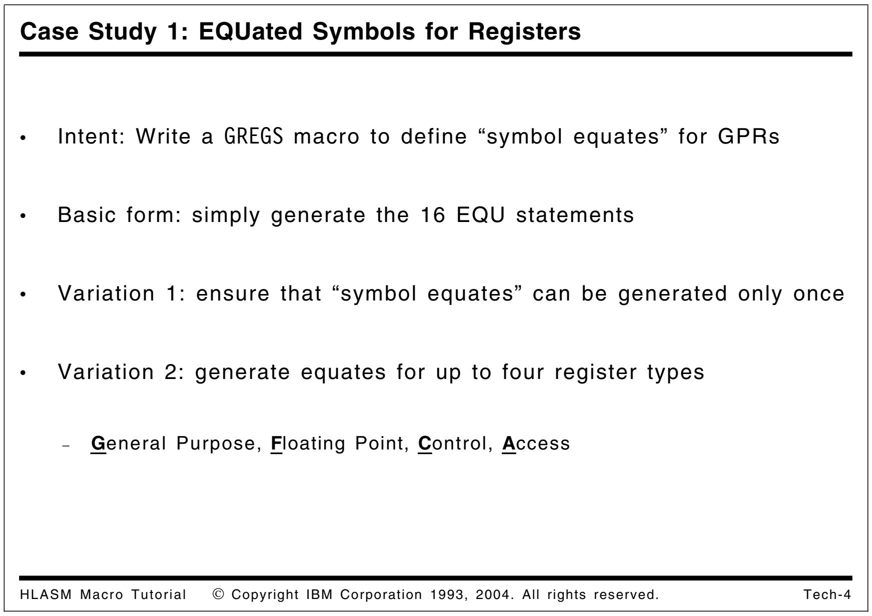
9. A set of macros illustrated at “Case Study 9: Defining and Using Data Types” on page 173 illustrate some techniques that can be used to implement type-sensitive operations (“polymorphism”), and user-defined data types and user-defined operations on them, with type checking and information hiding.

   a. “Case Study 9a: Type Sensitivity -- Simple Polymorphism” on page 175 shows how the assembler's type attributes can be used to tailor generated code sequences to the types of operands.

   b. “Case Study 9b: Type Checking” on page 181 shows how user-assigned type attributes can be used to perform type checking “conformance” between instructions, operands, and registers.

   c. “Case Study 9c: Encapsulated Abstract Data Types” on page 193 shows how user-defined data types and operations can “encapsulate” the details of data definitions and low-level operations on the data objects.

10. Sometimes it is useful to be able to capture and analyze the arguments passed to another macro, while still using the original macro definition for its intended purposes. This is called “front-ending” or “wrapping” a macro, and “Case Study 10: “Front-Ending” a Macro” on page 205 will illustrate a simple way to do this.



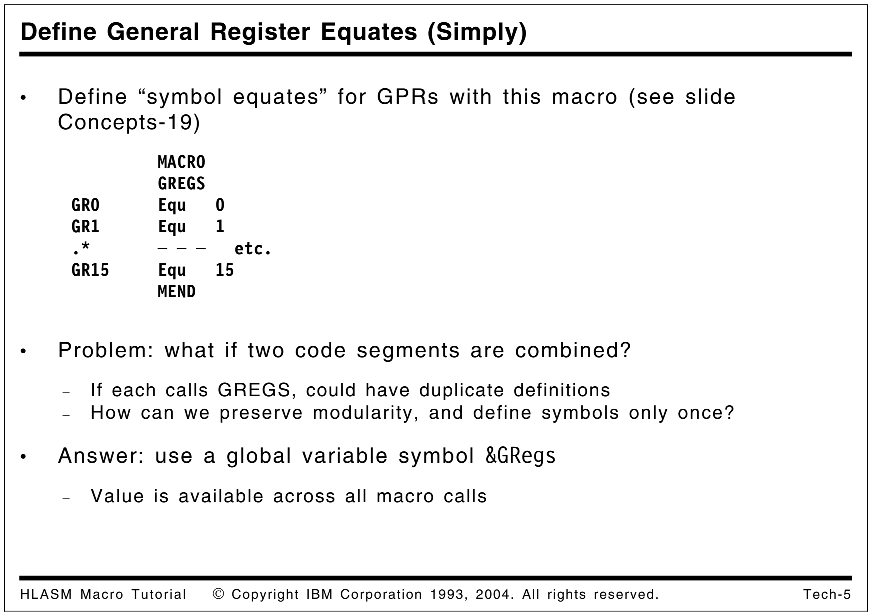
Case Study 1: EQUated Symbols for Registers

- Intent: Write a GREGS macro to define “symbol equates” for GPRs
- Basic form: simply generate the 16 EQU statements
- Variation 1: ensure that “symbol equates” can be generated only once
- Variation 2: generate equates for up to four register types
  - **G**eneral Purpose, **F**loating Point, **C**ontrol, **A**ccess

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-4

Case Study 1: Defining Equated Symbols for Registers

The technique illustrated in “Example 1: Define Equated Symbols for Registers” on page 67 is quite acceptable unless we need at some point to combine multiple code segments, each of which may possible contain a call to GREGS (which was needed for its own modular development). How can we avoid generating multiple copies of the EQU statements, with their accompanying diagnostics for multiply-defined symbols?



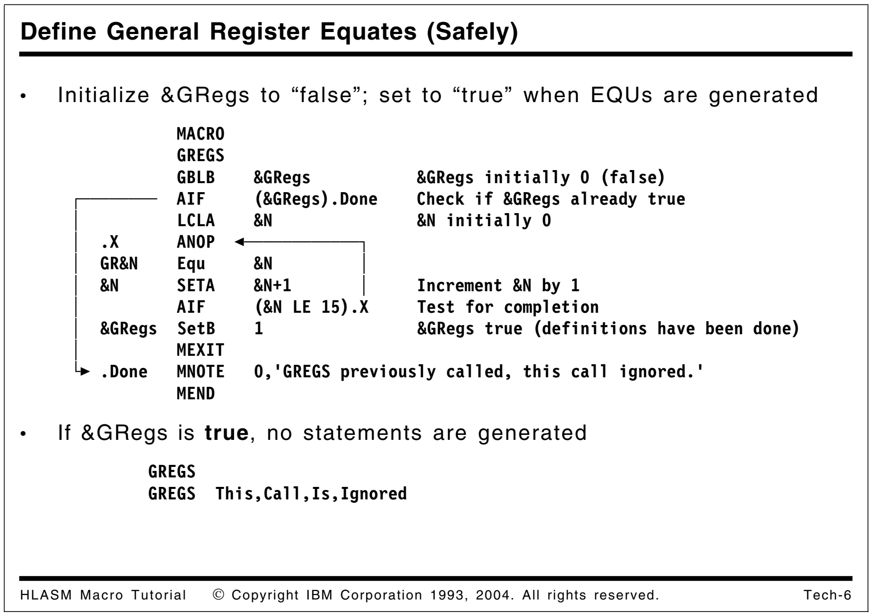
Define General Register Equates (Simply)

- Define “symbol equates” for GPRs with this macro (see slide Concepts-19)

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

- Problem: what if two code segments 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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-5



Define General Register Equates (Safely)

- 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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-6

The solution is simple: use a global variable symbol whose value will indicate that the GREGS macro has been called already. This is illustrated in Figure 28.

```
         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 28. Macro to Define General Purpose Registers Once Only

Defining Register Equates Safely: Pseudo-Code

- 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 (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 are not defined again
```

Define All Register Equates (Safely)

```
        MACRO
        REGS
        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
&N      SETA    0
.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
```

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

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:

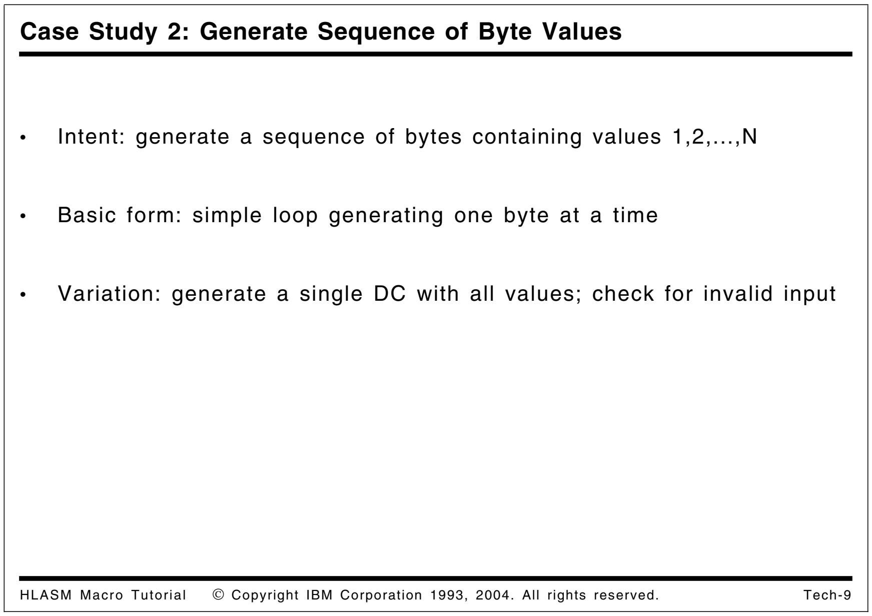
```
IF (number of arguments is zero) EXIT
FOR each argument:
   Verify valid register type (A, C, F, or G):
      IF invalid, ERROR EXIT with message
   IF (that type already done) Give message and ITERATE
   Generate equates
   Set appropriate 'Type_Done' flag and ITERATE
```

The following example uses the technique illustrated in Figure 28 on page 105 above, but generalizes it by using a “created set symbol” to select the name of the proper global variable symbol.

```
        MACRO
        REGS
        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
&N      SetA    0
.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 29. 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!).

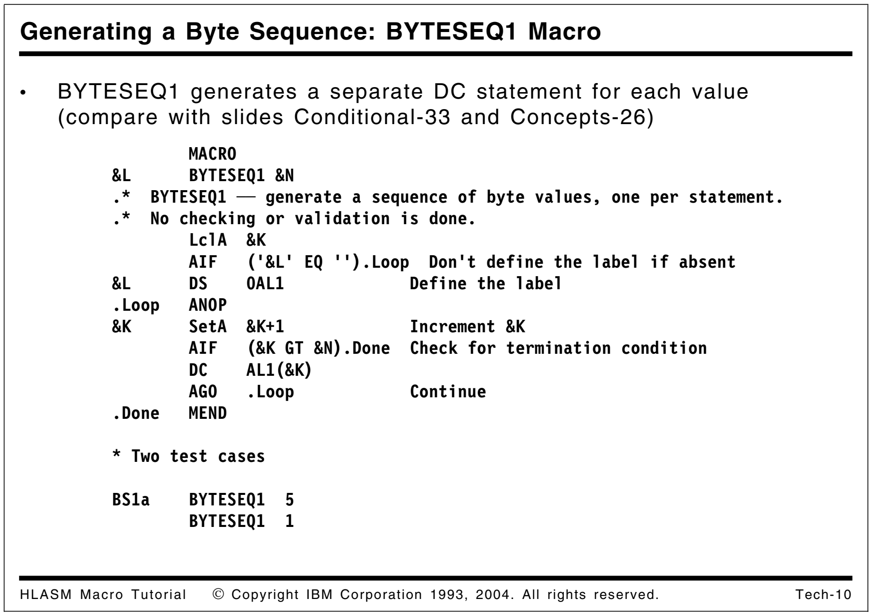


Case Study 2: Generate Sequence of Byte Values

- Intent: generate a sequence of bytes containing values 1,2,...,N
- Basic form: simple loop generating one byte at a time
- Variation: generate a single DC with all values; check for invalid input

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-9

Case Study 2: Generating a Sequence of Byte Values



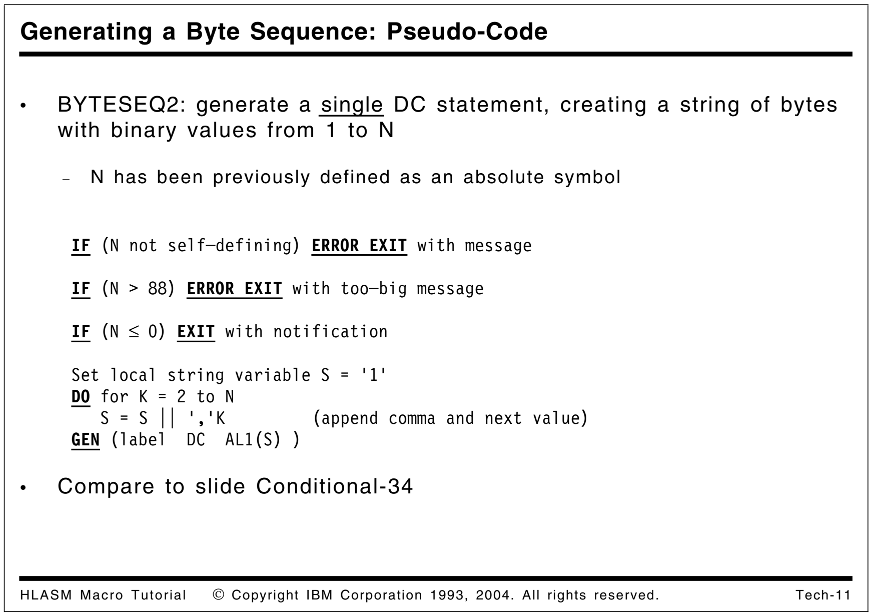
Generating a Byte Sequence: BYTESEQ1 Macro

- BYTESEQ1 generates a separate DC statement for each value (compare with slides Conditional-33 and Concepts-26)

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

* Two test cases

BS1a    BYTESEQ1  5
        BYTESEQ1  1
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-10

The sample BYTESEQ2 macro illustrated in Figure 30 on page 110 uses the same techniques as the conditional-assembly examples given in Figure 6 on page 44 and Figure 7 on page 45. and the corresponding BYTESEQ1 macro illustrated in Figure 23 on page 74.



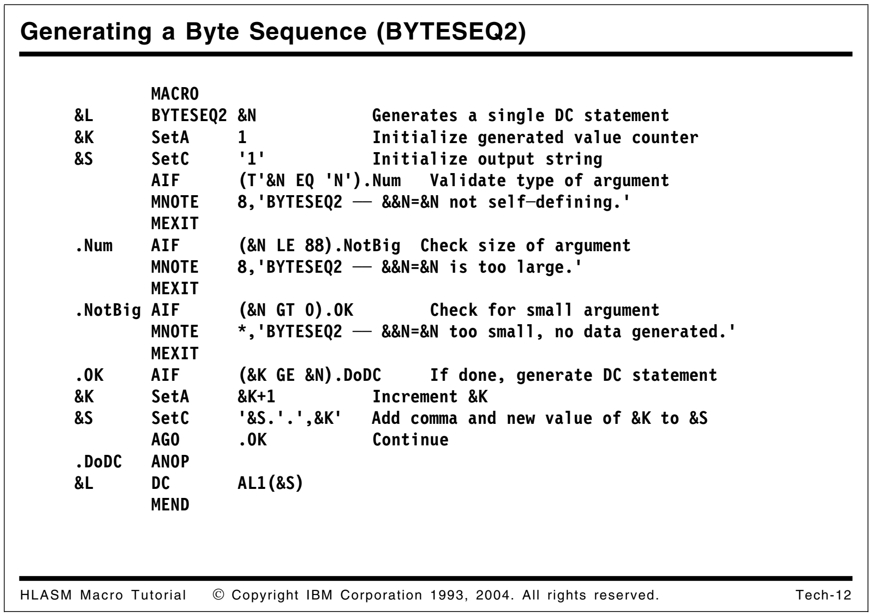
Generating a Byte Sequence: Pseudo-Code

- BYTESEQ2: 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 > 88) 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 Conditional-34

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-11



Generating a Byte Sequence (BYTESEQ2)

```
        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 88).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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-12

A pseudo-code outline of the macro implementation is as follows:

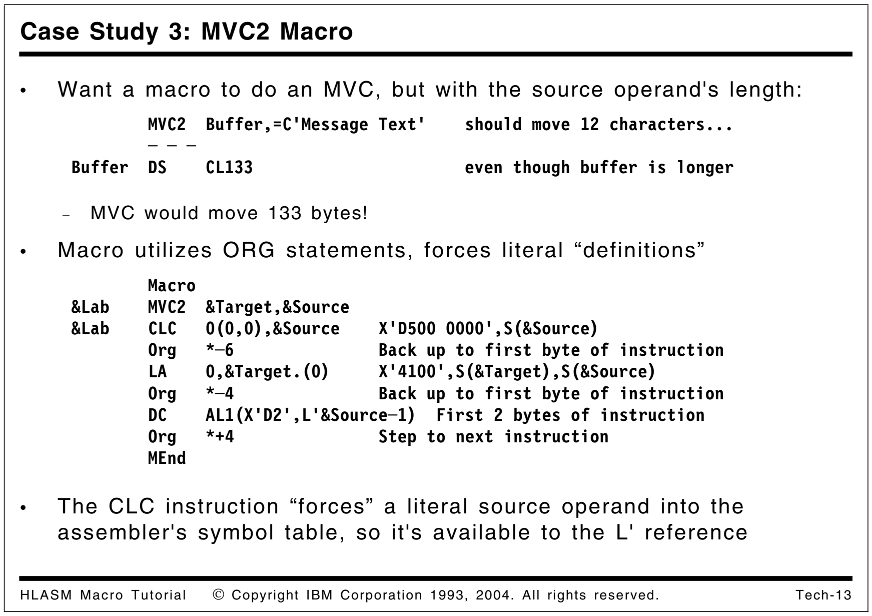
```
IF (N not self-defining) ERROR EXIT with message

IF (N > 88) 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) )
```

The BYTESEQ2 macro shown in Figure 30 on page 110 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, without reading the text following the next figure?

```
        MACRO
&L      BYTESEQ2 &N
.*  BYTESEQ2 -- generate a sequence of byte values, one per statement.
.*  The argument is checked and validated, and the entire constant is
.*  generated in a single DC statement.
        LclA     &K
        LclC     &S
&K      SetA     1             Initialize generated value counter
&S      SetC     '1'           Initialize output string
        AIF      (T'&N EQ 'N').Num   Validate type of argument
        MNOTE    8,'BYTESEQ2 -- &&N=&N not self-defining.'
        MEXIT
.Num    AIF      (&N LE 88).NotBig  Check size of argument
        MNOTE    8,'BYTESEQ2 -- &&N=&N is too large.'
        MEXIT
.NotBig AIF      (&N GT 0).OK        Check for small argument
        MNOTE    *,'BYTESEQ2 -- &&N=&N too small, no data generated.'
        MEXIT
.OK     AIF      (&K GE &N).DoDC     If done, generate DC statement
&K      SetA     &K+1          Increment &K
&S      SetC     '&S.'.',&K'   Add comma and new value of &K to &S
        AGO      .OK           Continue
.DoDC   ANOP
&L      DC       AL1(&S)
        MEND
* Test cases
BS2e    BYTESEQ2  0
BS2b    BYTESEQ2  1
BS2a    BYTESEQ2  5
BS2d    BYTESEQ2  X'58'
BS2c    BYTESEQ2  256
```

Figure 30. Macro to Define a Sequence of Byte Values As a Single String

Because no character variable symbol may contain more than 255 characters, the argument to BYTESEQ2 may not exceed 88; otherwise &S exceeds 255 characters. We leave as an exercise to the interested reader what steps could be taken to adapt this macro to accept arguments up to and including 255, and still generate a single DC statement.



Case Study 3: MVC2 Macro

- Want a macro to do an MVC, but with the source operand's length:

```
        MVC2  Buffer,=C'Message Text'    should move 12 characters...
        – – –
 Buffer DS    CL133                      even though buffer is longer
```

  - MVC would move 133 bytes!
- Macro utilizes ORG statements, forces literal “definitions”

```
        Macro
 &Lab   MVC2  &Target,&Source
 &Lab   CLC   0(0,0),&Source    X'D500 0000',S(&Source)
        Org   *-6               Back up to first byte of instruction
        LA    0,&Target.(0)     X'4100',S(&Target),S(&Source)
        Org   *-4               Back up to first byte of instruction
        DC    AL1(X'D2',L'&Source-1)  First 2 bytes of instruction
        Org   *+4               Step to next instruction
        MEnd
```

- The CLC instruction “forces” a literal source operand into the assembler's symbol table, so it's available to the L' reference

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-13

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 something clumsy and error-prone 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
```

There are several reasons why this might not work as simply as it is written; the most difficult situation (and also probably the most useful!) occurs when a literal is used as the source operand. When the assembler processes the length expression L'&Source, it must find the symbol (or literal) corresponding to &Source in the symbol table; otherwise the expression cannot be evaluated and is treated as invalid.

The following macro avoids this problem by first generating a CLC instruction (for which literals are valid in both the first and second operands), which causes any literal operands in the macro call to be entered into the symbol table. Then, the CLC instruction is overlaid with the fields appropriate to the desired MVC instruction.

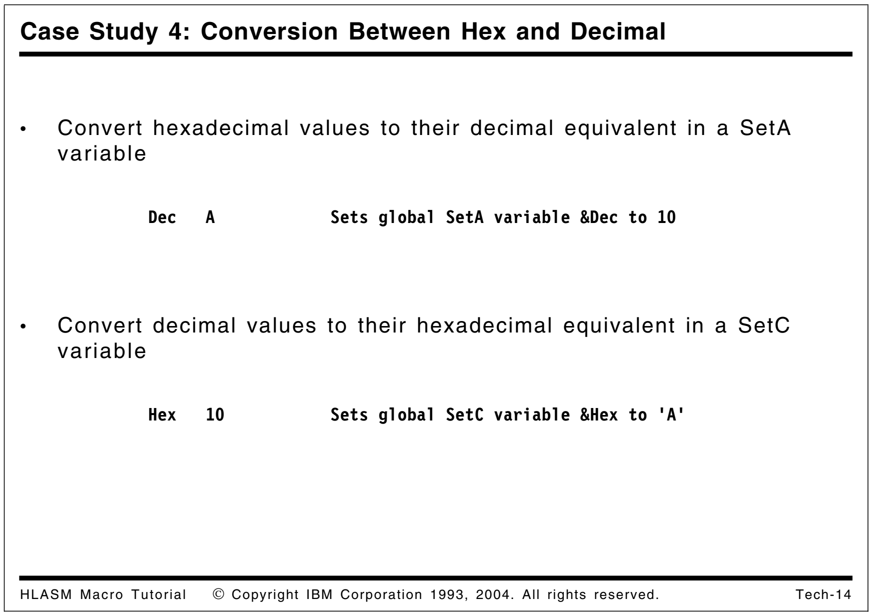
```
        Macro
&Lab    MVC2  &Target,&Source
        AIf   (N'&SysList eq 2).OK
        MNote 8,'Wrong number of operands in MVC2 macro call.'
        MExit
.OK     ANop
.*      Generate the CLC instruction with correct source operand
&Lab    CLC   0(0,0),&Source    X'D500 0000',S(&Source)
        Org   *-6               Back up to first byte of instruction
.*      Generate the addressing halfword for the target operand
        LA    0,&Target.(0)     X'4100',S(&Target),S(&Source)
        Org   *-4               Back up to first byte of instruction
.*      Generate the MVC opcode and the length byte
        DC    AL1(X'D2',L'&Source-1)  First 2 bytes of instruction
        Org   *+4               Step to end of MVC instruction
        MEnd
```

Figure 31. MVC2 Macro Definition

The CLC instruction causes any literals used as source operands to be placed into the symbol table prior to the length attribute reference in the DC statement.

An example of the code generated by the MVC2 macro is shown in the following:

```
                      ...       MVC2 Buff,=C'-Error: '
000600 D500 0000 F8AD ... +     CLC  0(0,0),=C'-Error: '  X'D500 0000',S(&Source)
000606                ... +     Org  *-6                Back up to first byte of instruction
000600 4100 F765      ... +     LA   0,Buff(0)          X'4100',S(&Target),S(&Source)
000604                ... +     Org  *-4                Back up to first byte of instruction
000600 D207           ... +     DC   AL1(X'D2',L'=C'-Error: '-1) First 2 bytes of instruction
000602                ... +     Org  *+4                Step to end of MVC instruction

000765                ...  Buff DS   CL133

0008AD 60C5999996997A40            =C'-Error: '
```



Case Study 4: Conversion Between Hex and Decimal

- Convert hexadecimal values to their decimal equivalent in a SetA variable

```
        Dec   A            Sets global SetA variable &Dec to 10
```

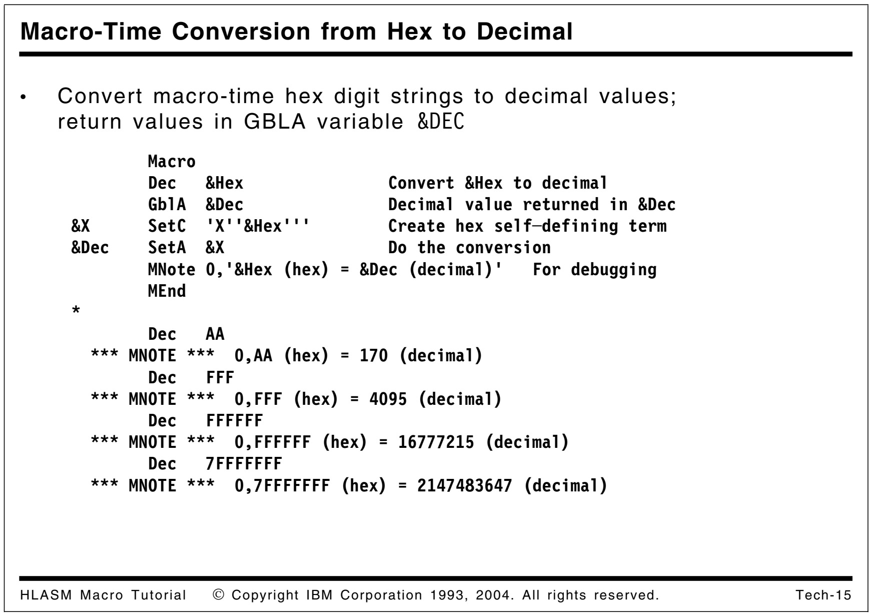
- Convert decimal values to their hexadecimal equivalent in a SetC variable

```
        Hex   10           Sets global SetC variable &Hex to 'A'
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-14

Case Study 4: Conversion Between Hex and Decimal

If you are writing macros, you may need to convert between two different representations of a data item. Some of these conversions are already available in the conditional assembly language; for example, arithmetic variables are automatically converted to character form by substituting them in SETC expressions.



Macro-Time Conversion from Hex to Decimal

- Convert macro-time hex digit strings to decimal values; return values in GBLA variable &DEC

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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-15

Macro-Time Conversion from Hex to Decimal

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

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

Figure 32. Macro-Time Conversion from Hex to Decimal

Note that the added level of substitution creating the variable &X is required. If you tried direct substitution into the SETA statement, the assembler issues a diagnostic:

```
&Dec    SetA  X'&Hex'
 ** ASMA037E Illegal self-defining value - X'&Hex'
```

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

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

Figure 33. Macro-Time Conversion from Hex to Decimal: Examples

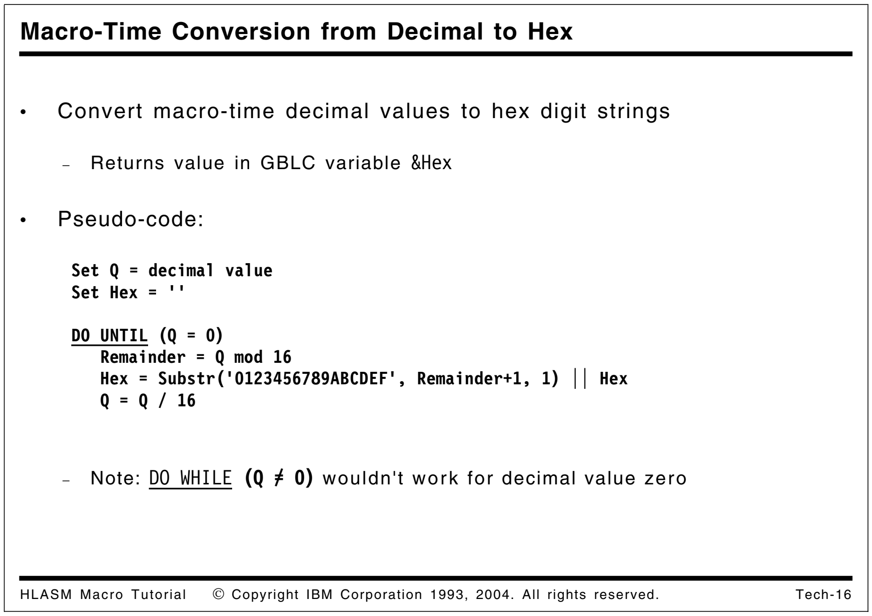
Note that this macro may appear to have a problem: any hex value exceeding X'7FFFFFFF' will not be displayed as a negative number. However, its internal representation in the variable &Dec will be correct.

Another shortcoming of this macro is that it makes no checks for the validity of the argument &Hex. This can be done using internal functions, as follows:

```
        Macro
        Dec   &Hex              Convert &Hex to decimal
        GblA  &Dec              Decimal value returned in &Dec
&X      SetC  (Upper '&Hex')    Convert to argument upper case
.Check  ANOP
&J      SetA  &J+1              Increment &J
&T      SetA  ('&X'(&J,1) Find '0123456789ABCDEF')  Validate character
        AIf   (&T eq 0).Bad         Error if invalid character
        AIf   (&J lt K'&X).Check    Look at next character
&X      SetC  'X''&Hex'''       Create hex self-defining term
&Dec    SetA  &X                Do the conversion
        MNote 0,'&Hex (hex) = &Dec (decimal)'   For debugging
        MExit
.Bad    MNote 5,'Invalid hex argument &&Hex = &Hex'
        MEnd
```

Figure 34. Macro-Time Conversion from Hex to Decimal, with Checking

The added statements first convert the alphabetic characters in the argument to upper case (to simplify the Find function). Then, each character of the argument is validated; if an invalid character is found, the macro branches to .Bad and issues an error message and terminates the macro, with severity code value 5.



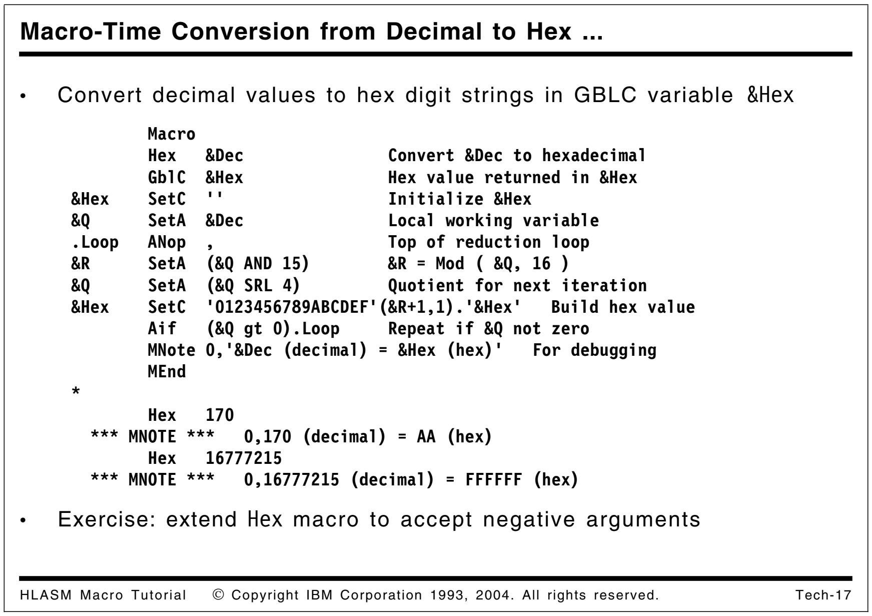
Macro-Time Conversion from Decimal to Hex

- Convert macro-time decimal values to hex digit strings
  - Returns value in GBLC variable &Hex
- Pseudo-code:

```
Set Q = decimal value
Set Hex = ''

DO UNTIL (Q = 0)
   Remainder = Q mod 16
   Hex = Substr('0123456789ABCDEF', Remainder+1, 1) || Hex
   Q = Q / 16
```

  - Note: DO WHILE **(Q ≠ 0)** wouldn't work for decimal value zero

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-16



Macro-Time Conversion from Decimal to Hex ...

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

```
        Macro
        Hex   &Dec              Convert &Dec to hexadecimal
        GblC  &Hex              Hex value returned in &Hex
&Hex    SetC  ''                Initialize &Hex
&Q      SetA  &Dec              Local working variable
.Loop   ANop  ,                 Top of reduction loop
&R      SetA  (&Q AND 15)       &R = Mod ( &Q, 16 )
&Q      SetA  (&Q SRL 4)        Quotient for next iteration
&Hex    SetC  '0123456789ABCDEF'(&R+1,1).'&Hex'   Build hex value
        Aif   (&Q gt 0).Loop    Repeat if &Q not zero
        MNote 0,'&Dec (decimal) = &Hex (hex)'   For debugging
        MEnd
*
        Hex   170
  *** MNOTE ***   0,170 (decimal) = AA (hex)
        Hex   16777215
  *** MNOTE ***   0,16777215 (decimal) = FFFFFF (hex)
```

- Exercise: extend Hex macro to accept negative arguments

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-17

Macro-Time Conversion from Decimal to Hex

Conversion from decimal to hexadecimal requires reducing the decimal value one hex digit at a time, using successive divisions by sixteen. A pseudo-code description of the conversion process is as follows:

```
Set Q = decimal value   (assumed non-negative!)
Set Hex = ''

DO UNTIL (Q = 0)
   Remainder = Q mod 16      (other bases possible, too)
   Hex = Substr('0123456789ABCDEF', Remainder+1, 1) || Hex
   Q = Q / 16
```

The Hex macro is shown in Figure 35. It accepts a single non-negative decimal argument, and returns its value in the GBLC variable &Hex.

```
        Macro
        Hex   &Dec              Convert &Dec to hexadecimal
        GblC  &Hex              Hex value returned in &Hex
&Hex    SetC  ''                Initialize &Hex
&Q      SetA  &Dec              Local working variable
.Loop   ANop  ,                 Top of reduction loop
&R      SetA  (&Q AND 15)       &R = Mod ( &Q, 16 )
&Q      SetA  (&Q SRL 4)        Quotient for next iteration
&Hex    SetC  '0123456789ABCDEF'(&R+1,1).'&Hex'   Build hex value
        Aif   (&Q gt 0).Loop    Repeat if &Q not zero
        MNote 0,'&Dec (decimal) = &Hex (hex)'   For debugging
        MEnd
```

Figure 35. Macro-Time Conversion from Decimal to Hex

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

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

Figure 36. Macro-Time Conversion from Decimal to Hex: Examples

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

It is an interesting further exercise to extend the function of the Hex macro to handle positive (unsigned) or negative (signed) arguments. This can be done with a few extra statements in the &Hex macro:

```
         - - -
&Hex     SetC  ''
         AIF   ('&Dec'(1,1) NE '-').NotNeg  Test for negative
&T       SetC  '&Dec'(2,*)          Save magnitude of &Dec
&Q       SetA  -&T                  Set &Q to signed value
         AGO   .Loop
.NotNeg  ANOP
&Q       SetA  &Dec                 Non-negative argument
.Loop    ANOP
         - - -
```

Completion and testing of the revised macro is left as part of the exercise! (Note that we can't directly substitute a negative value of &Dec into the SETA statement for &Q, because it does not have the form of a self-defining term, and must therefore be handled specially.)

In practice, it would probably be simpler (and maybe more efficient) to write external functions to do these conversions, without any need for global variables to communicate between the conversion “routine” and its caller.

Case Study 5: Generate Named Integer Constants

- Intent: generate a list of “intuitively” named halfword or fullword integer constants
- For example:
  - Fullword value “1” is a constant named F1
  - Halfword value “−1” is a constant named HM1

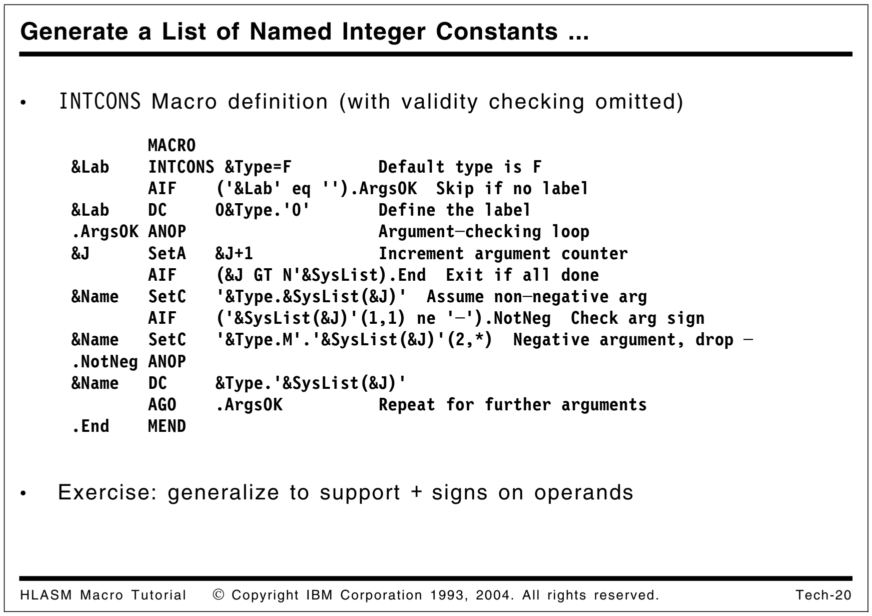
Case Study 5: Generate a List of Named Integer Constants

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

Generate a List of Named Integer Constants

- Syntax: INTCONS n_1[,n_2]...[,Type=F]
  - Default constant type is F
- Examples:

```
 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'
```



Generate a List of Named Integer Constants ...

- INTCONS Macro definition (with validity checking omitted)

```
         MACRO
&Lab     INTCONS &Type=F         Default type is F
         AIF    ('&Lab' eq '').ArgsOK  Skip if no label
&Lab     DC     0&Type.'0'       Define the label
.ArgsOK  ANOP                    Argument-checking loop
&J       SetA   &J+1             Increment argument counter
         AIF    (&J GT N'&SysList).End  Exit if all done
&Name    SetC   '&Type.&SysList(&J)'  Assume non-negative arg
         AIF    ('&SysList(&J)'(1,1) ne '-').NotNeg  Check arg sign
&Name    SetC   '&Type.M'.'&SysList(&J)'(2,*)  Negative argument, drop -
.NotNeg  ANOP
&Name    DC     &Type.'&SysList(&J)'
         AGO    .ArgsOK          Repeat for further arguments
.End     MEND
```

- Exercise: generalize to support + signs on operands

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-20

The syntax of the macro might look like this:

INTCONS n_1[,n_2]...[,Type=F]

If we wrote

```
INTCONS 1,-1
```

the macro would generate these statements:

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

Similarly, if we wrote

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

then the macro would generate:

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

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

The second part (beginning at the sequence symbol .ArgsOK) uses the &SYSLIST system variable symbol to step through each of the positional arguments in turn, by applying a subscript (&J) to indicate which positional argument is desired. The argument is checked for being non-null, and then to see if its first character is a minus sign. If the minus sign is present, it is removed for constructing the constant's name; finally, the constant is generated with the required name.

```
        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     0&Type.'0'       Define the label
.*    Verify that arguments are present; no harm if none.
.NoLab  AIF    (N'&SysList gt 0).ArgsOK  Check presence of args
        MNOTE  *,'INTCONS -- No arguments provided.'
        MEXIT
.*    Argument-checking loop
.ArgsOK ANOP
&J      SetA   &J+1             Increment argument counter
        AIF    (&J GT N'&SysList).End  Exit if all done
        AIF    (K'&SysList(&J) gt 0).DoArg
        MNOTE  4,'INTCONS -- Argument No. &J. is empty.'
        AGO    .ArgsOK          Go for next argument
.DoArg  ANOP
&Name   SetC   '&Type.&SysList(&J)'  Assume 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 37. Macro Parameter-Argument Association Example: Create a List of Constants

Some test cases for the INTCONS macro are shown in the following figure. The first two and the last two test various unusual conditions, and the third and fourth display the statements generated by the macro. (The '+' characters in the left margin are inserted by the assembler to indicate generated statements).

```
 * Test cases -- first has no label, no args; second has no args.
         INTCONS
 C1a     INTCONS

 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'

 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 38. Macro Example: List-of-Constants Test Cases

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

Case Study 6: Using the AREAD Statement

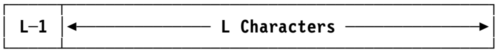
1. Case Study 6a: Generate strings of message text
   - Prefix string with “effective length” byte (length-1)
   - Basic form: count characters
   - Variation 1: create an extra symbol, use its length attribute
   - Variation 2: use the AREAD statement and conditional-assembly functions to support “readable” input
2. Case Study 6b: Block comments
   - Write free-form text comments (without * in column 1)

Case Study 6: Using the AREAD Statement

This case study shows two examples of the power of the AREAD statement. The first shows how to simplify creating message texts prefixed by a length byte, and the second illustrates a technique for entering “block comments” into your source program.

Case Study 6a: Create Length-Prefixed Message Texts

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

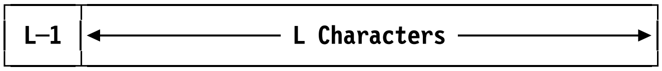


- How they might be used:

```
  HW      PFMSG 'Hello World'       Define a sample message text
 +HW      DC    AL1(10),C'Hello World'   Length-prefixed message text
          - - -
          LA    2,HW                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
```

Case Study 6a: Creating Length-Prefixed Message Texts

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



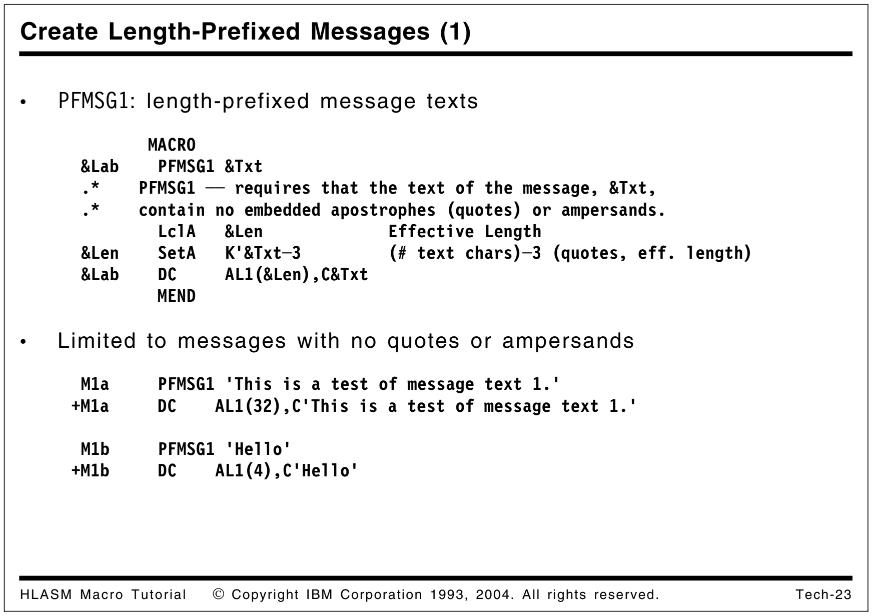
Such a length-prefixed message text could be used in code sequences like the following. The message is declared using a PFMSG macro, which generates the length byte followed by the message text:

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

Then, this small “data structure” could be used in instructions like these to move the message text to a buffer:

```
           LA    2,HW                Prepare to move message to buffer
           - - -
           - - -                     Call message-buffering routine?
           - - -
           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)

- PFMSG1: length-prefixed message texts

```
          MACRO
 &Lab      PFMSG1 &Txt
 .*      PFMSG1 —— requires that the text of the message, &Txt,
 .*      contain no embedded apostrophes (quotes) or ampersands.
          LclA   &Len              Effective Length
 &Len      SetA   K'&Txt-3          (# text chars)-3 (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'
```

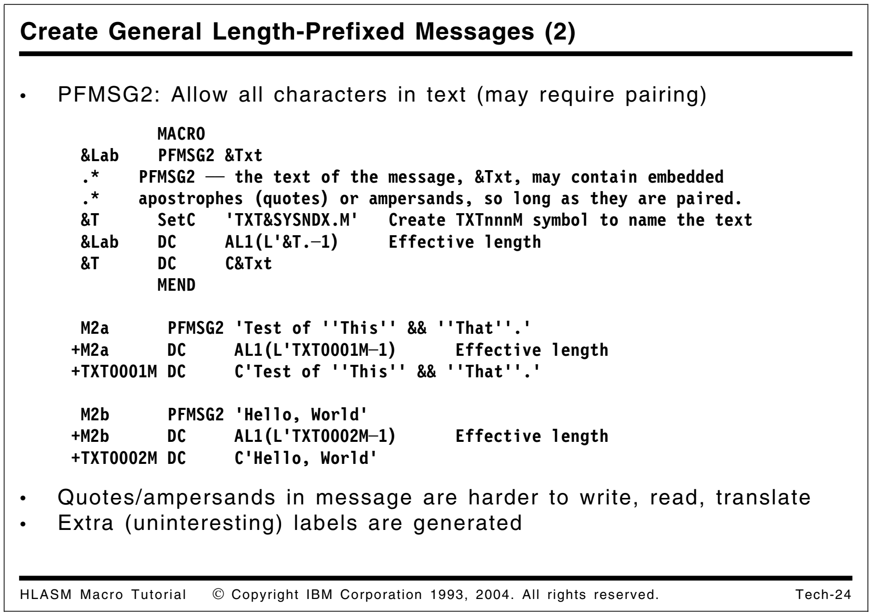
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-23

Simplest Prefixed Message Text

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

```
         MACRO
&Lab     PFMSG1 &Txt
.*     PFMSG1 -- requires that the text of the message, &Txt,
.*     contain no embedded apostrophes (quotes) or ampersands.
         LclA   &Len            Effective Length
&Len     SetA   K'&Txt-3        (# text chars)-3 (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 39. Macro to Define a Length-Prefixed Message



Create General Length-Prefixed Messages (2)

- PFMSG2: Allow all characters in text (may require pairing)

```
         MACRO
 &Lab    PFMSG2 &Txt
 .*    PFMSG2 — the text of the message, &Txt, may contain embedded
 .*    apostrophes (quotes) or ampersands, so long as they are 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-24

More General Prefixed Message Text

The requirement that no ampersands or quotes may be used in the message text defined by PFMSG1 may not be acceptable in some situations. Thus, in Figure 40 on page 125 we will define a second macro PFMSG2 that allows such characters in the message, but requires that they be properly paired in the argument string. It also generates an ordinary symbol so that a length attribute reference may be used.

```
        MACRO
&Lab    PFMSG2 &Txt
.*    PFMSG2 -- the text of the message, &Txt, may contain embedded
.*    apostrophes (quotes) or ampersands, so long as they are
.*    properly paired. The macro expansion generates a symbol
.*    using the &SYSNDX system variable symbol, and uses a Length
.*    attribute reference for the effective length.
&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 40. 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.

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 an elegant and powerful feature of the macro language, the AREAD statement.

Readable Length-Prefixed Messages (3): Pseudo-Code

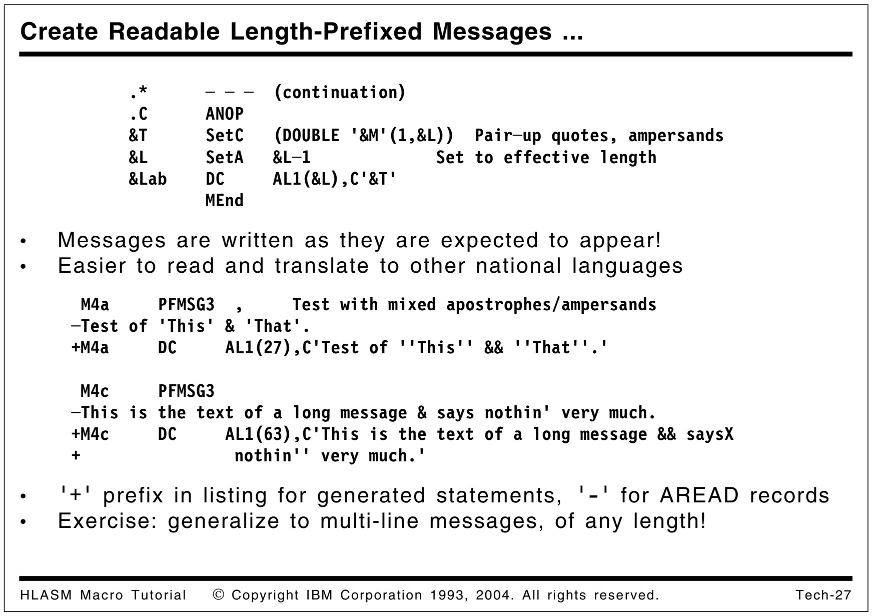
- User writes “plain text” messages (single line, ≤ 72 characters)
- PFMSG3: AREAD statement within the macro “reads” the next source record (following the macro call) into a character variable symbol
- Pseudo-code:

```
IF (any positional arguments) ERROR EXIT with message

AREAD a message from the following source record
Trim off sequence field (73-80) and trailing blanks

Create paired quotes and ampersands (for nominal value in DC)

GEN (label  DC  AL1(Text_Length-1),C'MessageText')
```

Create Readable Length-Prefixed Messages

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

```
        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,&N            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  8,'PFMSG3 -- no operands should be provided.'
        MEXIT                   Terminate macro processing
.OK     ANOP
&N      SetA   1                Initialize char-scan pointer to 1
.*  Read the record following the 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
.*      - - -  (continued)
```



Create Readable Length-Prefixed Messages ...

```
        .*      - - -  (continuation)
        .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
```

- 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 AREAD records
- Exercise: generalize to multi-line messages, of any length!

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-27

Prefixed Message Text with the AREAD Statement

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

```
  &CVar  AREAD
```

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

We will exploit this capability in the PFMSG3 macro, which reads the text of a message written in its desired final form from the line following the macro call. The operation of the PFMSG3 macro is summarized in the following 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 41 on page 128 scans the text of the string, creating pairs of quotes and ampersands wherever needed; thus, the writer of the message need not be aware of the peculiar rules of the Assembler Language.

A note on style: to allow users of the PFMSG3 macro to add comments to 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 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,&N            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  8,'PFMSG3 -- no operands should be provided.'
        MEXIT                   Terminate macro processing
.OK     ANOP
&N      SetA   1                Initialize char-scan pointer to 1
.*  Read the record following the 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 41. Macro to Define a Length-Prefixed Message With “True Text”

Some test cases for the PFMSG3 macro are shown in the following figure.

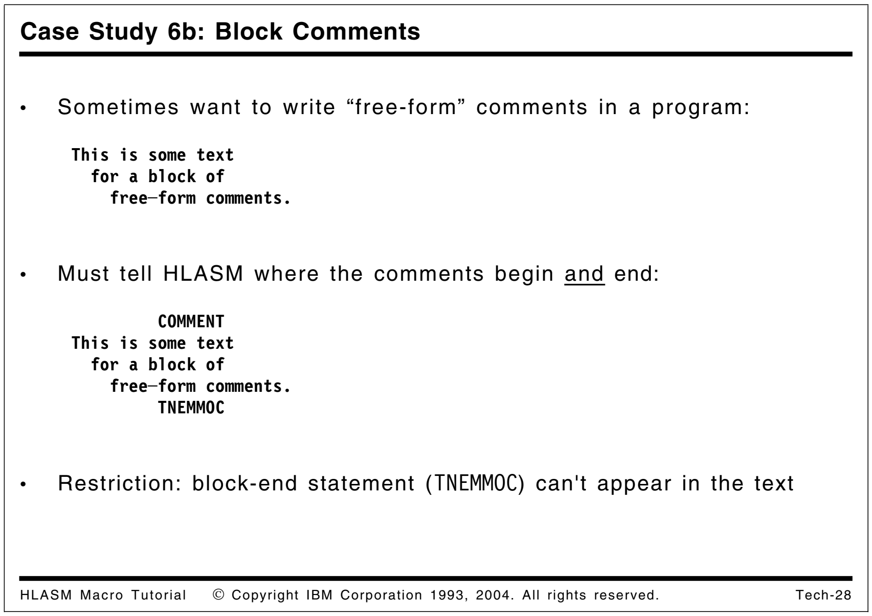
```
 M4a     PFMSG3  ,     Test with mixed apostrophes/ampersands
-Test of 'This' & 'That'.
+M4a     DC     AL1(27),C'Test of ''This'' && ''That''.'

 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 42. 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.

An instructive exercise can be found in generalizing the above macro to accept multi-line messages, first with total length less than 255 characters, and then with no limitations on total length.

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.



Case Study 6b: Block Comments

- Sometimes want to write “free-form” comments in a program:

```
This is some text
  for a block of
    free-form comments.
```

- Must tell HLASM where the comments begin and end:

```
         COMMENT
This is some text
  for a block of
    free-form comments.
         TNEMMOC
```

- Restriction: block-end statement (TNEMMOC) can't appear in the text

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-28

Case Study 6b: 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.
```

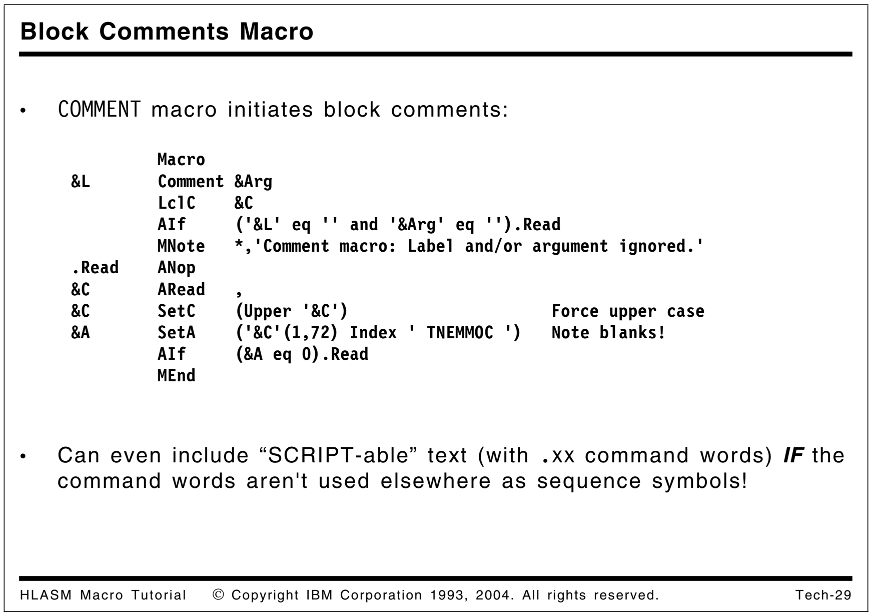
Naturally, we will need some way to tell the assembler that the comment “lines” are not to be scanned as normal input statements. Thus, we need something that indicates the start (and end) of a “block comment”.

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 might even be able to embed program documentation in your source code!)



Block Comments Macro

- 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    ('&C'(1,72) Index ' TNEMMOC ')   Note blanks!
          AIf     (&A eq 0).Read
          MEnd
```

- Can even include “SCRIPT-able” text (with .xx command words) ***IF*** the command words aren't used elsewhere as sequence symbols!

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-29

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    ('&C'(1,72) Index ' TNEMMOC ')   Note blanks!
         AIf     (&A eq 0).Read
         MEnd
```

Figure 43. 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 converts each line of the block comment text to upper case to simplify 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:

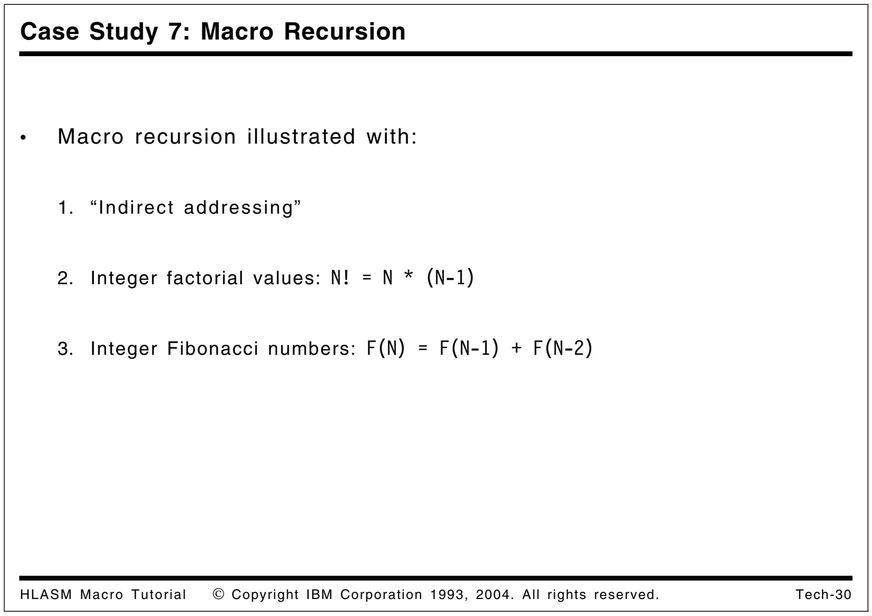
```
         Comment
 Note that the block-comment terminator can't appear in the
 comments! That's because the embedded terminator string on
 this line causes an error when TNEMMOC is encountered:
         Tnemmoc
```

The presence of the string ' TNEMMOC ' in an input line 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 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'.

Note also that you can keep your code and its documentation in one file, by embedding the SCRIPT-able documentation as block comments. This will require some care, however, to avoid possible confusion between SCRIPT command words (like .SP) and sequence symbols of the same name.



Case Study 7: Macro Recursion

- Macro recursion illustrated with:

  1. “Indirect addressing”
  2. Integer factorial values: N! = N * (N-1)
  3. Integer Fibonacci numbers: F(N) = F(N-1) + F(N-2)

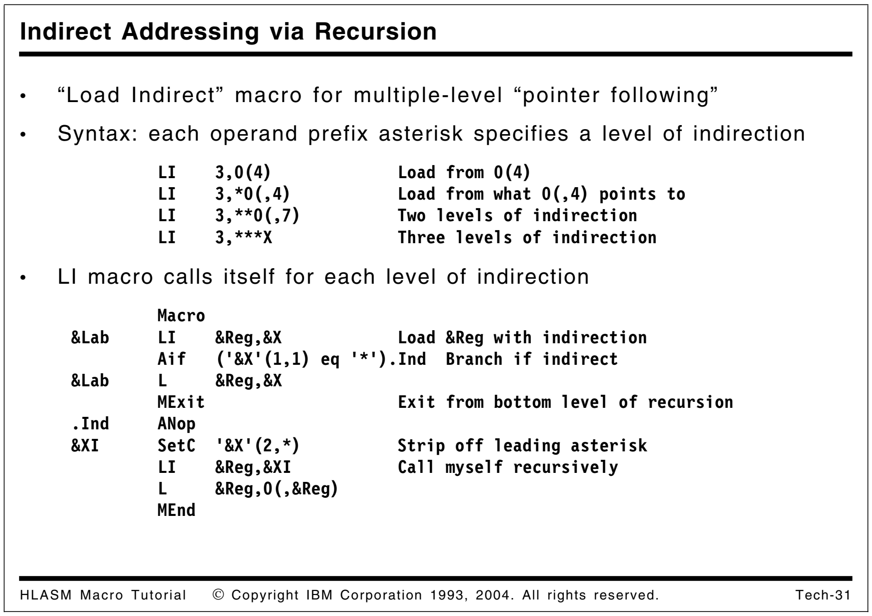
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-30

Case Study 7: Macro Recursion

Macros that call themselves either directly or indirectly are *recursive*. Three examples are given:

- a “Load Indirect” macro LI
- a “Factorial” macro FACTORAL
- a FIBONACI macro to calculate Fibonacci numbers.

We will first illustrate a recursive call with a simple “Load Indirect” macro, which introduces a simple form of indirect addressing.



Indirect Addressing via Recursion

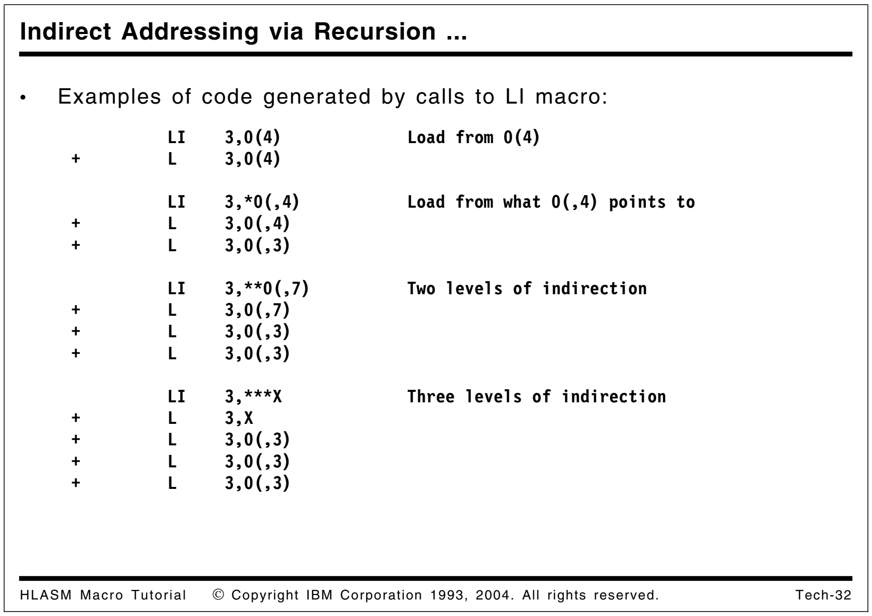
- “Load Indirect” macro for multiple-level “pointer following”
- Syntax: each operand prefix asterisk specifies a level of indirection

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

- LI macro calls itself for each level of indirection

```
         Macro
&Lab     LI    &Reg,&X           Load &Reg with indirection
         Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
&Lab     L     &Reg,&X
         MExit                   Exit from bottom level of recursion
.Ind     ANop
&XI      SetC  '&X'(2,*)         Strip off leading asterisk
         LI    &Reg,&XI          Call myself recursively
         L     &Reg,0(,&Reg)
         MEnd
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-31



Indirect Addressing via Recursion ...

- Examples of code generated by calls to LI macro:

```
         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)
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-32

Recursion Example 1: Indirect Addressing

In Figure 44 on page 133, 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.[10] Thus, if an instruction was written as

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

[10] 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.

then the item to be loaded into R8 is not at XXX, but is at the position “pointed to” by XXX. Thus, the asterisk can be thought of as a “de-referencing” operator.

Note that R0 cannot be used for &Reg if any levels of indirection are indicated.

This definition is recursive, in the sense that the “operand” preceded by an asterisk may itself be preceded by an asterisk, which thus provides multiple levels of indirection. A macro to implement this form of indirect addressing is shown in Figure 44.

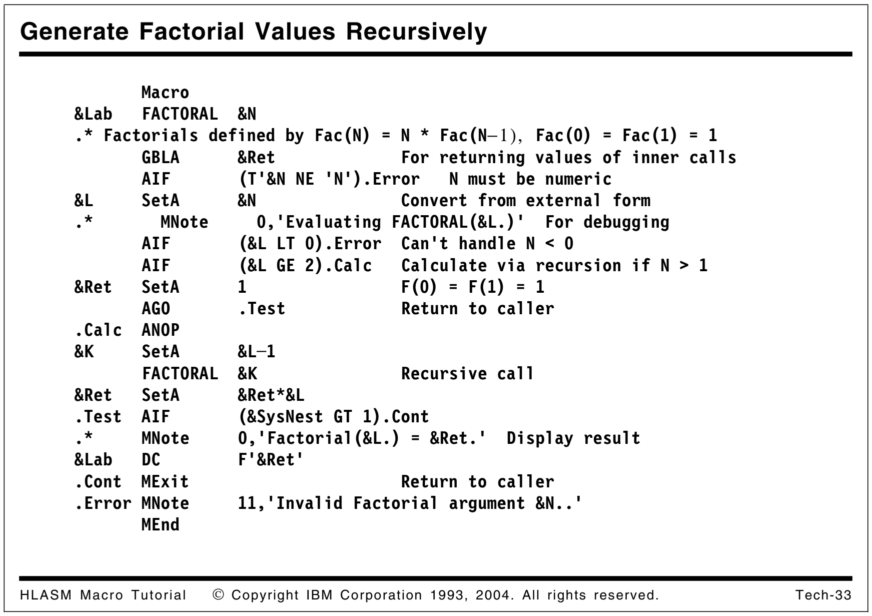
```
         Macro
&Lab     LI    &Reg,&X            Load &Reg with indirection
         Aif   ('&X'(1,1) eq '*').Ind  Branch if indirect
&Lab     L     &Reg,&X
         MExit                    Exit from bottom level of recursion
.Ind     ANop
&XI      SetC  '&X'(2,K'&X-1)     Strip off leading asterisk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

Figure 44. Recursive Macro to Implement Indirect Addressing

Some examples of calls to the LI macro are shown in Figure 45, where the “+” characters at the left margin are the assembler's indication of a macro-generated statement.

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

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

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

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

Figure 45. Recursive Macro to Implement Indirect Addressing: Examples



Generate Factorial Values Recursively

```
       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
       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               F(0) = F(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).Cont
.*     MNote     0,'Factorial(&L.) = &Ret.'  Display result
&Lab   DC        F'&Ret'
.Cont  MExit                     Return to caller
.Error MNote     11,'Invalid Factorial argument &N..'
       MEnd
```

HLASM Macro Tutorial Tech-33

Recursion Example 2: Factorial Function Values

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

In the macro in Figure 46 on page 135, 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. Macros called from open code are always 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                F(0) = F(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).Cont
.*     MNote     0,'Factorial(&L.) = &Ret.'  Display result
&Lab   DC        F'&Ret'
.Cont  MExit                      Return to caller
.Error MNote     11,'Invalid Factorial argument &N..'
       MEnd
```

Figure 46. 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 47. Macro to Calculate Factorials Recursively: Examples

As noted previously, the generated statements are tagged with a '+' character in the left margin.

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

Generate Fibonacci Numbers: Pseudo-Code

- Defined by F(0) = F(1) = 1, F(n) = F(n-1) + F(n-2)
- Use a global arithmetic variable &Ret for returned values
  - Macros have no other way to return “function” values
- Pseudo-code:

```
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
```

Generate Fibonacci Numbers Recursively

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

Recursion Example 3: Fibonacci Numbers

The Fibonacci numbers are defined by the recursion relations

```
F(N) = F(N–1) + F(N–2)
with F(0) = 1 and F(1) = 1
```

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

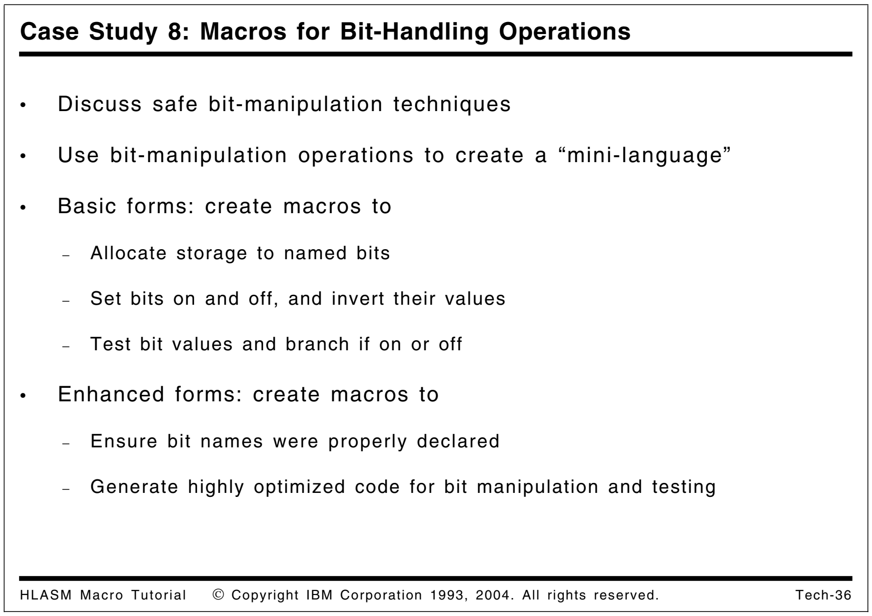
A pseudo-code description of the macro's operation is as follows:

```
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 48. 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 is used to hold 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.'
       AIF       (&N LT 0).Error  Negative values not allowed
       AIF       (&N GE 2).Calc   If &N > 1, use recursion
&Ret   SETA      1                Return F(0) or F(1)
       AGO       .Test            Return to caller

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

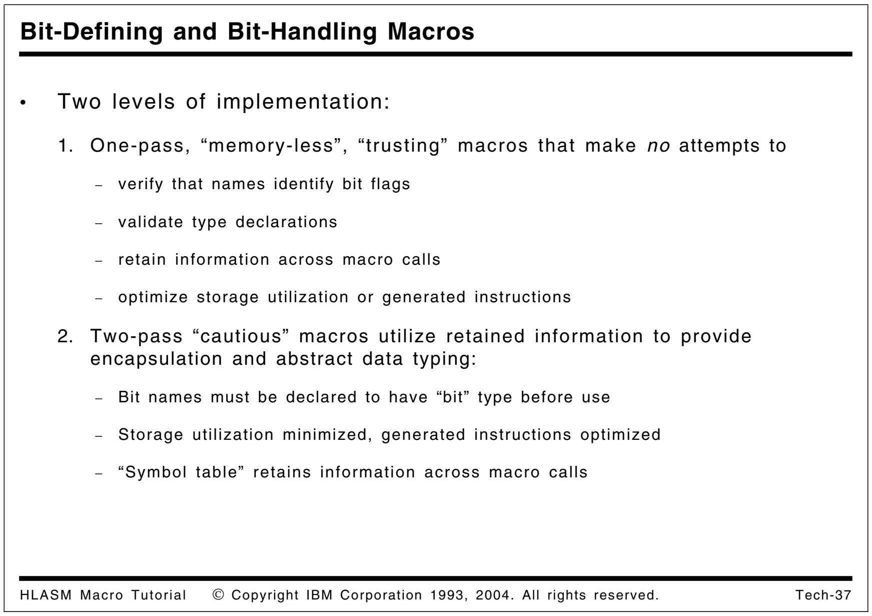
Figure 48. Macro to Calculate Fibonacci Numbers Recursively



Case Study 8: Macros for Bit-Handling Operations

- Discuss safe bit-manipulation techniques
- Use bit-manipulation operations to create a "mini-language"
- Basic forms: create macros to
  - Allocate storage to named bits
  - Set bits on and off, and invert their values
  - Test bit values and branch if on or off
- Enhanced forms: create macros to
  - Ensure bit names were properly declared
  - Generate highly optimized code for bit manipulation and testing

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-36



Bit-Defining and Bit-Handling Macros

- Two levels of implementation:
  1. One-pass, "memory-less", "trusting" macros that make *no* attempts to
     - verify that names identify bit flags
     - validate type declarations
     - retain information across macro calls
     - optimize storage utilization or generated instructions
  2. Two-pass "cautious" macros utilize retained information to provide encapsulation and abstract data typing:
     - Bit names must be declared to have "bit" type before use
     - Storage utilization minimized, generated instructions optimized
     - "Symbol table" retains information across macro calls

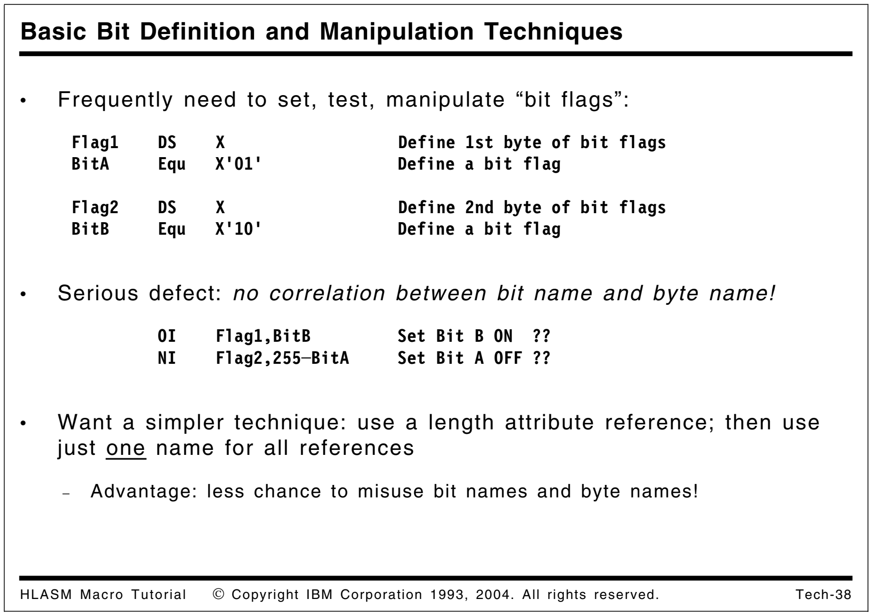
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-37

Case Study 8: Defining Macros for Bit-Handling Operations

We will 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 8a) will illustrate 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 8b) will do the same functions, but will in addition validate the declared names when they are used elsewhere, and generate 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 storage requirements and/or execution time must be minimized, the second set of macros can be used.



Basic Bit Definition and Manipulation Techniques

- Frequently need to set, test, manipulate “bit flags”:

```
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
```

- Serious defect: *no correlation between bit name and byte name!*

```
         OI    Flag1,BitB       Set Bit B ON  ??
         NI    Flag2,255—BitA   Set Bit A OFF ??
```

- Want a simpler technique: use a length attribute reference; then use just one name for all references
  - Advantage: less chance to misuse bit names and byte names!

HLASM Macro Tutorial   © Copyright IBM Corporation 1993, 2004. All rights reserved.   Tech-38

Basic Bit Handling Techniques

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

It is a very common technique in Assembler Language programming to define bits using statements like the following:

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

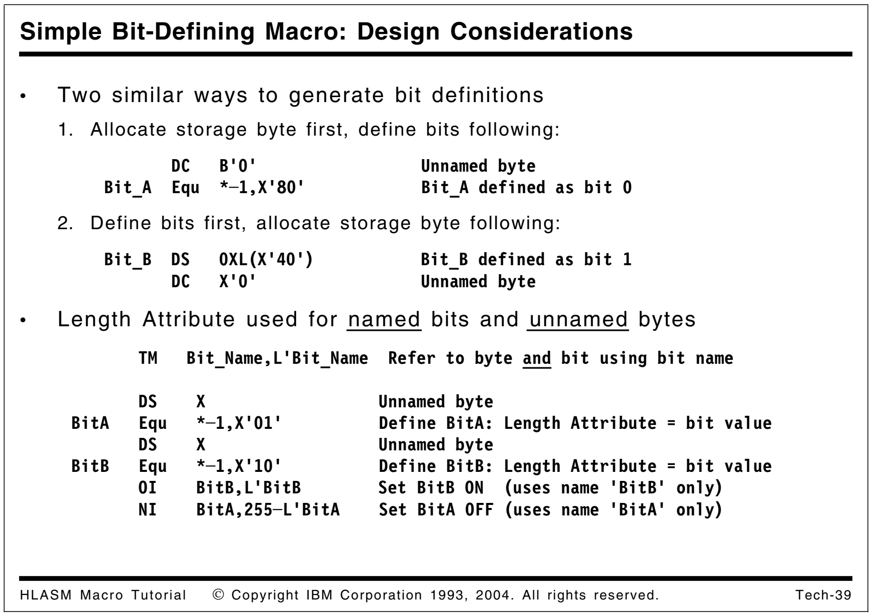
and then doing bit operations like

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

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

```
         OI    Flag1,BitB       Set Bit B ON  ??
         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-Defining Macro: Design Considerations

- Two similar ways to generate bit definitions
  1. Allocate storage byte first, define bits following:

```
        DC   B'0'               Unnamed byte
  Bit_A Equ  *–1,X'80'          Bit_A defined as bit 0
```

  2. Define bits first, allocate storage byte following:

```
  Bit_B DS   0XL(X'40')         Bit_B defined as bit 1
        DC   X'0'               Unnamed byte
```

- Length Attribute used for <u>named</u> bits and <u>unnamed</u> bytes

```
        TM   Bit_Name,L'Bit_Name  Refer to byte and bit using bit name

        DS    X                 Unnamed byte
 BitA   Equu  *–1,X'01'         Define BitA: Length Attribute = bit value
        DS    X                 Unnamed byte
 BitB   Equ   *–1,X'10'         Define BitB: Length Attribute = bit value
        OI    BitB,L'BitB       Set BitB ON  (uses name 'BitB' only)
        NI    BitA,255–L'BitA   Set BitA OFF (uses name 'BitA' only)
```

HLASM Macro Tutorial Tech-39

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:

```
          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
```

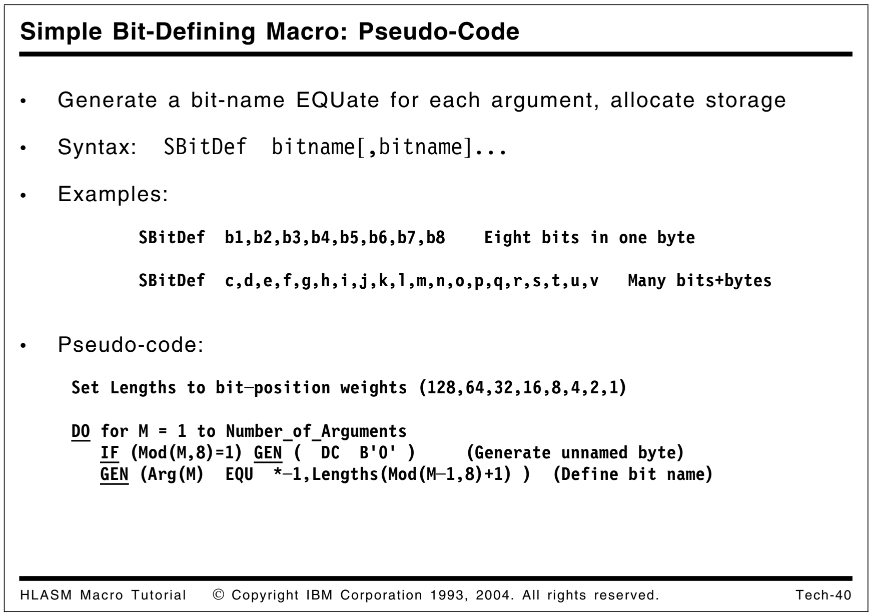
In each case, the bit name is the same as the name of the byte that contains it. Then, all bit references are made only with the bit “names”:

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

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

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

but there is clearly something peculiar about the statement; and, 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-Defining Macro: Pseudo-Code

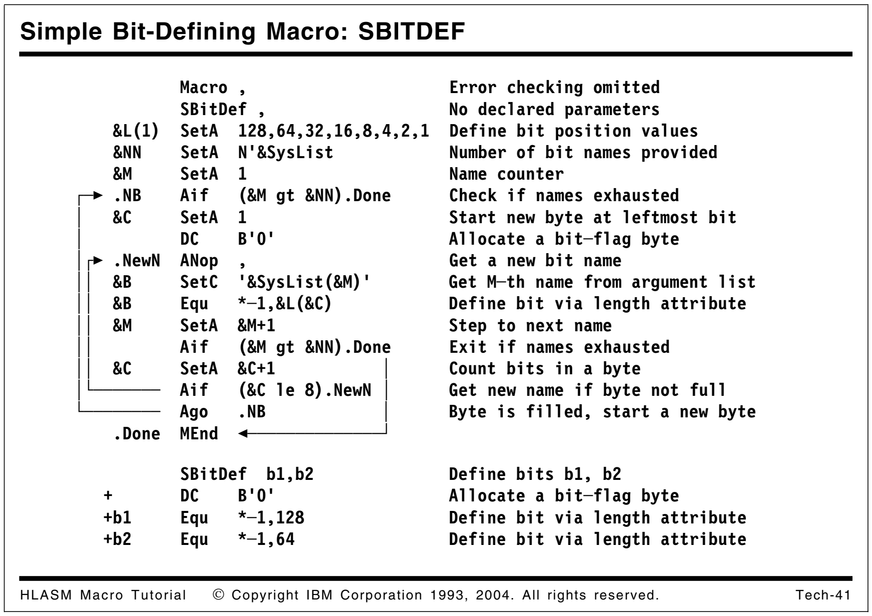
- Generate a bit-name EQUate for each argument, allocate storage
- Syntax: SBitDef bitname[,bitname]...
- Examples:

```
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 Tech-40



Simple Bit-Defining Macro: SBITDEF

```
         Macro ,                      Error checking omitted
         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
 .NB     Aif   (&M gt &NN).Done       Check if names exhausted
 &C      SetA  1                      Start new byte at leftmost bit
         DC    B'0'                   Allocate a bit-flag byte
 .NewN   ANop  ,                      Get a new bit name
 &B      SetC  '&SysList(&M)'         Get M-th name from argument list
 &B      Equ   *-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 byte not full
         Ago   .NB                    Byte is filled, start a new byte
 .Done   MEnd

         SBitDef  b1,b2               Define bits b1, b2
+        DC    B'0'                   Allocate a bit-flag byte
+b1      Equ   *-1,128                Define bit via length attribute
+b2      Equ   *-1,64                 Define bit via length attribute
```

HLASM Macro Tutorial Tech-41

Case Study 8a: Bit-Handling Macros -- Simple Forms

The simplest way to “encourage” correct matching of bit names and byte names is to make all bit references with macros. We will illustrate a simple set of macros to do this.

First, suppose we want to “define” bit names, and allocate storage for them. We will write a macro that accepts a list of bit names, and defines bit values in successive bytes, eight bits to a byte. A pseudo-code description of the macro's operation is as follows:

```
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 on page 142 takes the names in the argument list and allocates a single bit to each, eight bits to a byte. Each call to the SBitDef macro starts a new byte. We use the &SYSLIST system variable symbol to access the arguments, and a Number attribute reference, N'&SYSLIST, to determine 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 Definitions

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

```
        SBitDef  b1,b2,b3,b4,b5,b6,b7,b8    Eight bits in one byte
+       DC    B'0'                Allocate a bit-flag byte
+b1     Equ   *-1,128             Define bit via length attribute
+b2     Equ   *-1,64              Define bit via length attribute
+b3     Equ   *-1,32              Define bit via length attribute
+b4     Equ   *-1,16              Define bit via length attribute
+b5     Equ   *-1,8               Define bit via length attribute
+b6     Equ   *-1,4               Define bit via length attribute
+b7     Equ   *-1,2               Define bit via length attribute
+b8     Equ   *-1,1               Define bit via length attribute

        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: Examples of Bit Definitions

This simple macro has several limitations:

- Bits cannot be “grouped” so that related bits are certain to reside in the same byte, except by writing a statement with a new 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, it will cause HLASM to issue a diagnostic ASMA043E message for a previously defined symbol.

We will explore some techniques that can be used to overcome these limitations.

Simple Bit-Manipulation Macros: Pseudo-Code

- Operations on “named” bits
- Setting bits on: one OI instruction per named bit

```
IF (Label ≠ null) GEN (Label  DC  0H'0')

DO for M = 1 to Number_of_Arguments
   GEN ( OI  Arg(M),L'Arg(M) )         to set bits on
```

- Length Attribute reference specifies the bit
  - As illustrated in the simple bit-defining macro
- Similar macros for setting bits off, or inverting bits

```
IF (Label ≠ null) GEN (Label  DC  0H'0')
   GEN ( NI  Arg(M),255−L'Arg(M) )     to set bits off

   GEN ( XI  Arg(M),L'Arg(M) )         to invert bits
```

- Warning: these simple macros are very trusting!

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-42

Simple Bit-Manipulation Macros

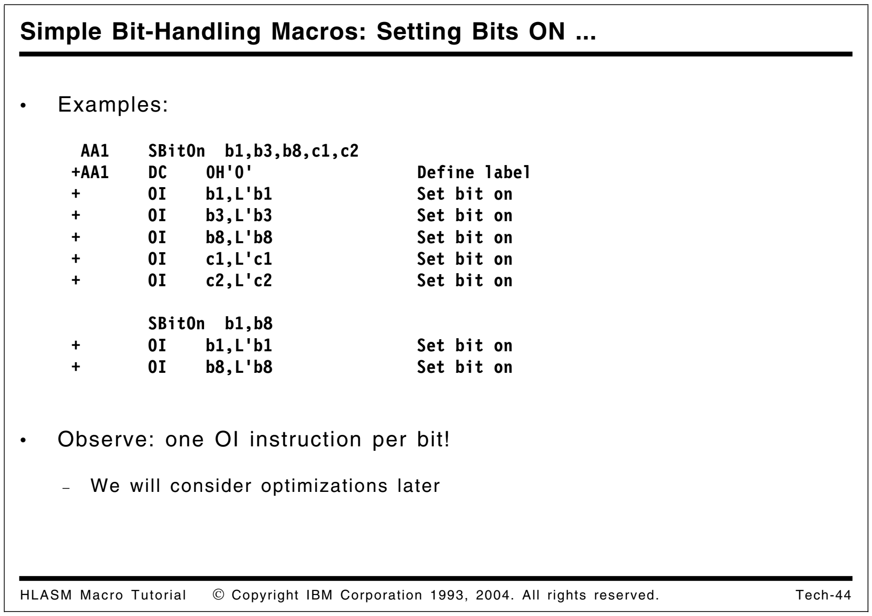
Now, we will illustrate some simple macros that “utilize” the bit definitions just described. (The macros are useful, but do very little checking; improvements will be discussed later, at “Case Study 8b: Bit-Handling Macros -- Advanced Forms” on page 151.)

Simple Bit-Handling Macros: Setting Bits ON

- Macro SBitOn to set one or more bits ON
- Syntax: SBitOn bitname[,bitname]...

```
         Macro ,                     Error Checking omitted
   &Lab  SBitOn
   &NN   SetA  N'&SysList            Number of Names
   &M    SetA  1
         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 argument)
  │ .Go  OI    &B,L'&B               Set bit on
  │ &M   SetA  &M+1                  Step to next bit name
  └───── Aif   (&M le &NN).Next      Go get another name
         MEnd
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-43



Simple Bit-Handling Macros: Setting Bits ON ...

- 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 later

HLASM Macro Tutorial Tech-44

Simple Bit-Manipulation Macros: Setting Bits ON

Having created the SBitDef macro to define bit names, we can now write some macros to manipulate them by setting them on and off, and by inverting ("flipping") their state. First, we will write a macro SBitOn that will set a bit to an "on" state (i.e., to 1).

A pseudo-code description of the SBitOn macro is as follows:

```
IF (Label ≠ null) GEN (Label  DC  0H'0')

DO for M = 1 to Number_of_Arguments
   GEN (  OI EQU  Arg(M),L'Arg(M) )
```

The SBitOn macro is shown 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

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

```
 AA1    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
```

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 8b: Bit-Handling Macros -- Advanced Forms" on page 151 how we might remedy this defect.

Simple Bit-Handling Macros: Set OFF and Invert Bits

- 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-45

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

- Examples:

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

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

- Observe: one NI or XI instruction per bit

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-46

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

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

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

Figure 54. Simple Bit-Handling Macros: Examples of Bit Resetting

Similarly, the SBitInv macro inverts the designated bits, using XI instructions:

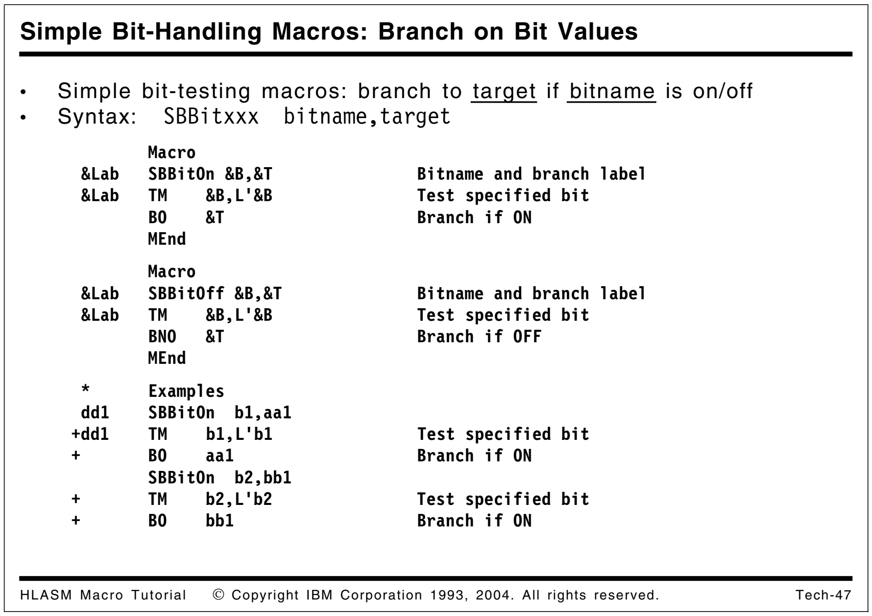
```
       Macro
&Lab   SBitInv
.*     - - -  etc., as for SBitOn
.Go    XI    &B,L'&B              Invert bit
.*     - - -  etc., as for SBitOn
       MEnd
```

Figure 55. Simple Bit-Handling Macros: Bit Inversion

Some calls to SBitInv illustrate its operation:

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

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

Figure 56. Simple Bit-Handling Macros: Examples of Bit Inversion



Simple Bit-Handling Macros: Branch on Bit Values

- Simple bit-testing macros: branch to target if bitname is on/off
- Syntax: `SBBitxxx  bitname,target`

```
        Macro
 &Lab   SBBitOn &B,&T               Bitname and branch label
 &Lab   TM    &B,L'&B               Test specified bit
        BO    &T                    Branch if ON
        MEnd

        Macro
 &Lab   SBBitOff &B,&T              Bitname and branch label
 &Lab   TM    &B,L'&B               Test specified bit
        BNO   &T                    Branch if OFF
        MEnd

 *      Examples
 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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-47

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:

```
SBBitxxx  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
```

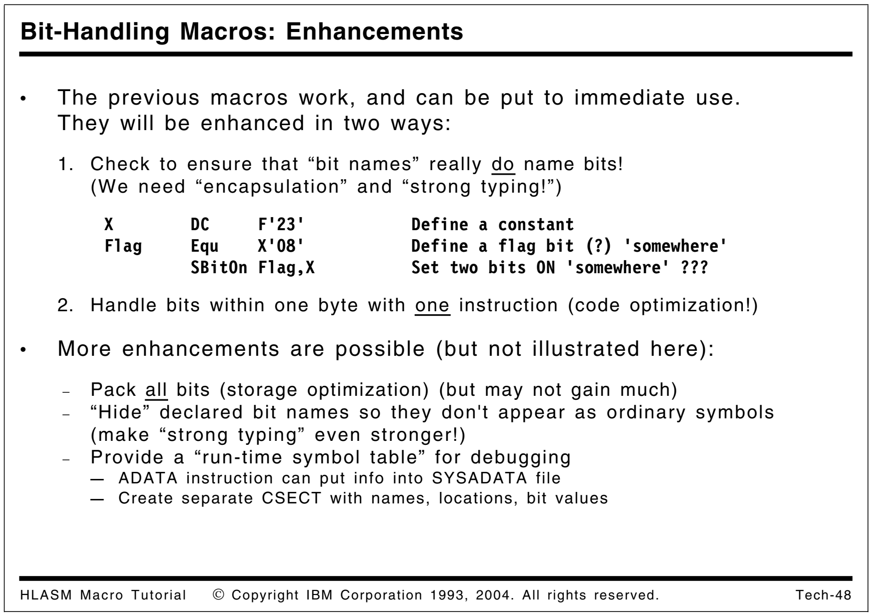
Figure 59. Simple Bit-Handling Macros: Branch if Bit is Off

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. It is evident that a fairly helpful set of capabilities can be written with a very small effort, and be put to immediate use.



Bit-Handling Macros: Enhancements

- The previous macros work, and can be put to immediate use. They will be enhanced in two ways:
  1. Check to ensure that “bit names” really do name bits! (We need “encapsulation” and “strong typing!”)

```
X        DC     F'23'           Define a constant
Flag     Equ    X'08'           Define a flag bit (?) 'somewhere'
         SBitOn Flag,X          Set two bits ON 'somewhere' ???
```

  2. Handle bits within one byte with one instruction (code optimization!)
- More enhancements are possible (but not illustrated here):
  - Pack all bits (storage optimization) (but may not gain much)
  - “Hide” declared bit names so they don't appear as ordinary symbols (make “strong typing” even stronger!)
  - Provide a “run-time symbol table” for debugging
    - ADATA instruction can put info into SYSADATA file
    - Create separate CSECT with names, locations, bit values

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-48

Case Study 8b: Bit-Handling Macros -- Advanced Forms

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

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

```
         DS    X
BitJ     Equ   *-1,X'40'
BitK     Equ   *-1,X'20'
```

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

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

   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 degree of “optimization” in our bit-handling macros.

2. It's worth observing that these simple macros are very trusting (and therefore require that you be very careful). There is no checking of the “bit names” presented as arguments in the bit-manipulation macros to verify that they were indeed *declared* as bits in a “bit definition” macro. For example, one might have written (through some oversight, probably not as drastic as this!)

```
Flag     Equ      X'08'             Define a flag bit
         - - -
         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
```

would generate the instruction

```
         OI    X,L'X
```

which is unlikely to give the result you intended!

Thus, we need some degree of “strong typing” and “type checking” in our bit-handling macros.

Bit-Handling “Micro-Compiler”

- Goal: Create a “Micro-compiler” for bit operations
  - Micro: Limit scope of actions to specific data types and operations
  - Compiler: Perform typical syntax/semantic scans, generate code
    - Each macro can check syntax of definitions and uses
    - Build and use “Symbol Tables” of created global variable symbols
- “Bit Language” the same as for the simple bit-handling macros:
  - Data type: named bits
  - Operations: define; set on/off/invert; test-and-branch
- Can incrementally add to and improve each language element
  - As these enhancements will illustrate

HLASM Macro Tutorial Tech-49

Bit-Handling “Micro Language” and “Micro-Compiler”

Solving these problems provides us with an opportunity to create a “micro-compiler” for handling bit declarations and operations. Because we have limited our concerns to bit operations, the macros can be fairly simple, while illustrating some of the types of functions needed in a typical compiler for a high-level language.

We will start with a BitDef macro that declares bit flags, and keeps track of which ones have been declared. We will add an extra feature 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 of the decisions influencing the design of these macros is that we wish to optimize execution performance more than we wish to minimize storage utilization; because bits are small, wasting a few shouldn't be a major concern. Each instruction saved represents many bits! (Storage optimization is left as an exercise for the reader.)

Bit-Handling Macros: Data Structures

- Bit declaration requires three simple “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 --
     Each declared bit name creates a global arithmetic variable:
     - Its name &(BitDef_MyBit_ByteNo) is constructed from
       - a prefix **BitDef_** (whatever you like, to avoid global-name collisions)
       - the declared bit name MyBit (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 bytes themselves will be 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 will 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 quite arbitrary, and need only contain the bit name somewhere; we will construct the name from a prefix BitDef_, the bit name, and a suffix _ByteNo. If such names collide with global names used by other macros, it is easy to change the prefix or suffix.

General Bit-Defining Macro: Design

- 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 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
```

General Bit-Defining Macro: Pseudo-Code

```
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
```

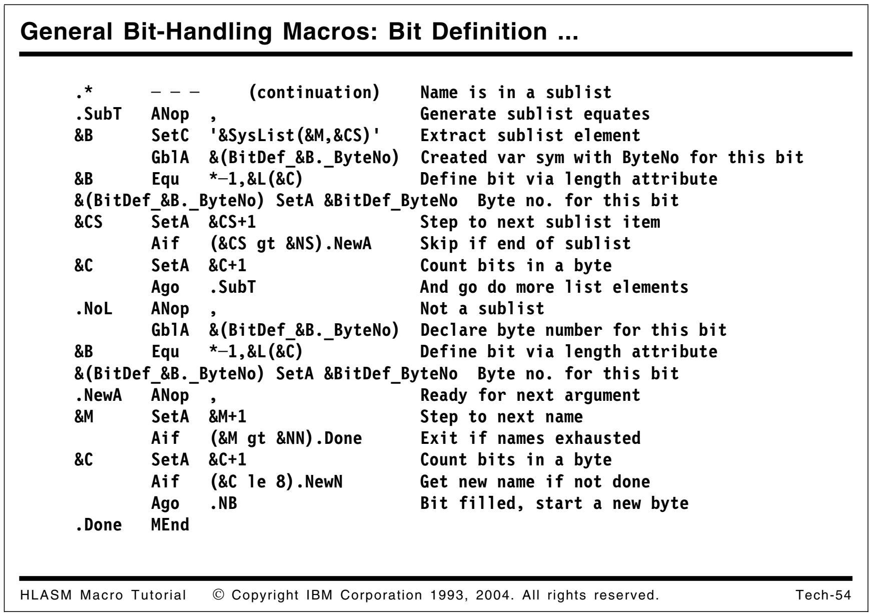
- Created symbol contains bit name; its value is the byte number

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.

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 a nonzero value determined by the Byte_Number in which the bit was allocated. (This “Byte_Number” is simply a count of the number of bytes allocated to hold bits declared by the BitDef macro.)

General Bit-Handling Macros: Bit Definition

```
        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 Definition ...

```
.*      - - -     (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      Equu  *-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
```

HLASM Macro Tutorial Tech-54

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 used to keep track of a “byte number” in which the various bits are allocated; each time a new byte is allocated, this variable is incremented by 1. The first SETA statement initializes the local arithmetic array variables &L(1) through &L(8) to values corresponding to the binary weights of the bits in a byte, in left-to-right order.

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

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

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

A pseudo-code description of the macro is as follows:

```
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
```

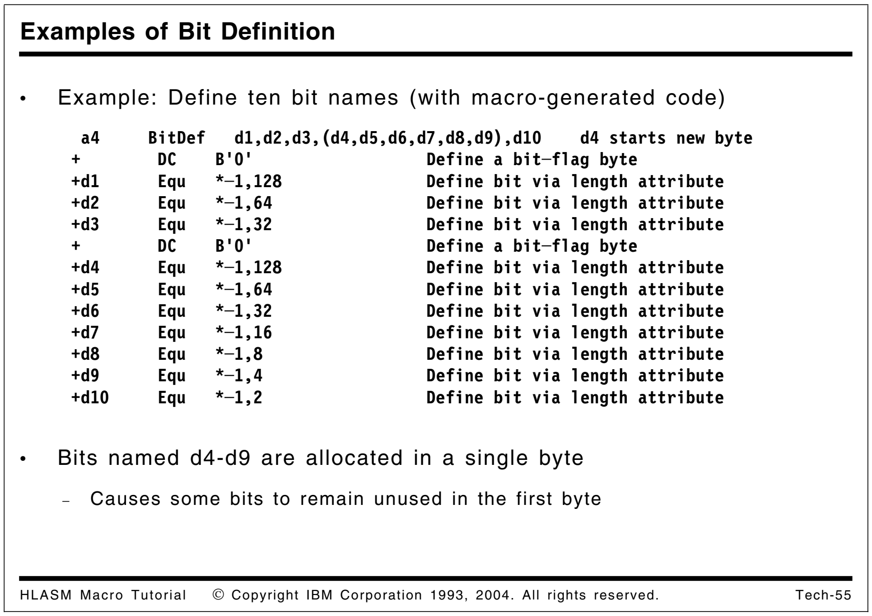
Figure 61. Bit-Handling Macros: Define Bit Names: Pseudo-Code

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, shown in the following figure, does include them.

```
        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
.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 Definition

- Example: Define ten bit names (with macro-generated code)

```
 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      Equu  *–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
```

- Bits named d4-d9 are allocated in a single byte
  - Causes some bits to remain unused in the first byte

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-55

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

```
 a4     BitDef   d1,d2,d3,(d4,d5,d6,d7,d8,d9),d10    d4 starts new byte
+        DC    B'0'                 Define a bit-flag byte
+d1      Equ   *-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

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

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

 a7     BitDef   (h2,h3,h4,h5,h6,h7,h8,h9,h10),h11   error, 9 in a byte
 a9     BitDef   (k1,k2,k3,k4),(k5,k6,k7,k8),k9,k10  two sublists
+        DC    B'0'                 Define a bit-flag byte
+k1      Equ   *-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

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

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

Figure 63 (Part 2 of 2). Bit-Handling Macros: Examples of Defining Bit Names

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

General Bit-Setting Macro: Data Structures

Two "phases" used to generate bit-operation instructions:

1. Check that bit names are declared (the "strong typing"), and collect information about bits to be set:
   a. Number of distinct Byte_Numbers (what bytes "own" the bit names?)
   b. For each byte, the number of instances of bit names in that byte
   c. An associatively addressed "name table" (variable symbol)
      - 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 itself
2. Use the information to generate optimal instructions
   - Names and number of name instances needed to build each operand

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-56

General Bit-Setting Macro: Design

- Optimize generated code using variables declared by BitDef macro
- 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

HLASM Macro Tutorial Tech-57

General Bit-Setting Macro: Pseudo-Code

- Detailed pseudo-code:

```
Save macro-call label
Set NBN (Number of known Byte Numbers) = 0
DO for M = 1 to Number_of_Arguments
   Set B = Arg(M)
   Declare created global variable &(BitDef_&B._Byte_Number)
   IF (Its value is zero) ERROR EXIT 'Undeclared Bitname &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 bitname list for this Byte Number

(End Arg scan: have all byte numbers and their associated bit names)
DO for M = 1 to number of distinct Byte Numbers
   Set Operand = 'First_Bitname,L''First_Bitname'  (local character string)
   DO for K = 2 to Number of bitnames in this Byte
      Operand = Operand || '+L''Bitname(K)'
   GEN (label  OI  Operand  ); set label = ''
```

- Easy generalization to Bit_Off (NI) and Bit_Invert (XI)

HLASM Macro Tutorial Tech-58

Improved Bit-Manipulation Macros

We will now explore some improved techniques for managing bit variables, including verifying that they were declared properly, and minimizing the number of instructions needed to manipulate and test them.

These macros use created variable symbols as an associatively-addressed symbol table, reducing the effort needed for table searches.

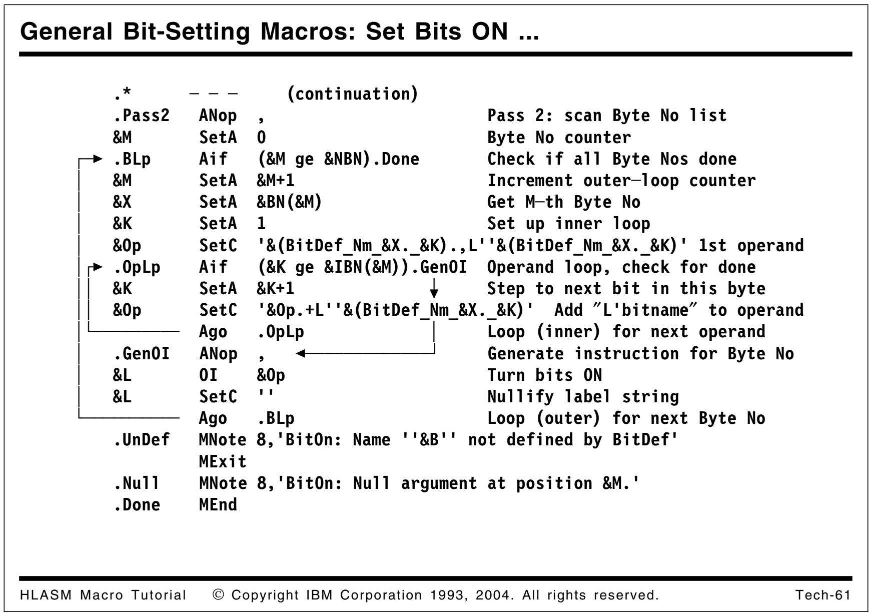
General Bit-Setting Macros: Set Bits ON

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

```
         Macro
&Lab     BitOn
&L       SetC  '&Lab'                  Save label
&NBN     SetA  0                       No. of distinct Byte Nos.
&M       SetA  0                       Name counter
&NN      SetA  N'&SysList              Number of names provided
.NmLp    Aif   (&M ge &NN).Pass2       Check if all names scanned
&M       SetA  &M+1                    Step to next name
&B       SetC  '&SysList(&M)'          Pick off a name
         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 ...

```
.*      - - -     (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 ...

```
.*      - - -     (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
```

HLASM Macro Tutorial   © Copyright IBM Corporation 1993, 2004. All rights reserved.   Tech-61

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 (to 1), 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 the global arithmetic variable &(BitDef_&B._&ByteNo) (where the value of &B is the bit name) is constructed and declared, and its value is checked. If the value is zero, we know that the name was not declared in a call to a BitDef macro (which would have assigned a nonzero byte number value to the variable).
- If the bit name was defined, the value of the constructed name is the byte number of the byte to which the bit was assigned. The array &BN() is searched to see if other bits with the same byte number have been supplied as arguments to this BitOn macro; if not, a new entry is made in the &BN() array.
- A second array &IBN() (paralleling the &BN() array) is used to count the number of **I**nstances of the **B**yte **N**umber that have occurred thus far.
- Finally, the bit name is saved in a created local character variable symbol &(BitDef_Nm_&bn._&in), where &bn is the byte number for this bit name, and &in is the "instance number" of this bit within this byte. (By checking the current bit name from the argument list against these names, the macro can also determine that a bit name has been "duplicated" in the argument list.)

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

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

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

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
   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)
DO for M = 1 to number of distinct Byte Numbers
   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

The definition of the BitOn macro is shown in Figure 65.

```
         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
```

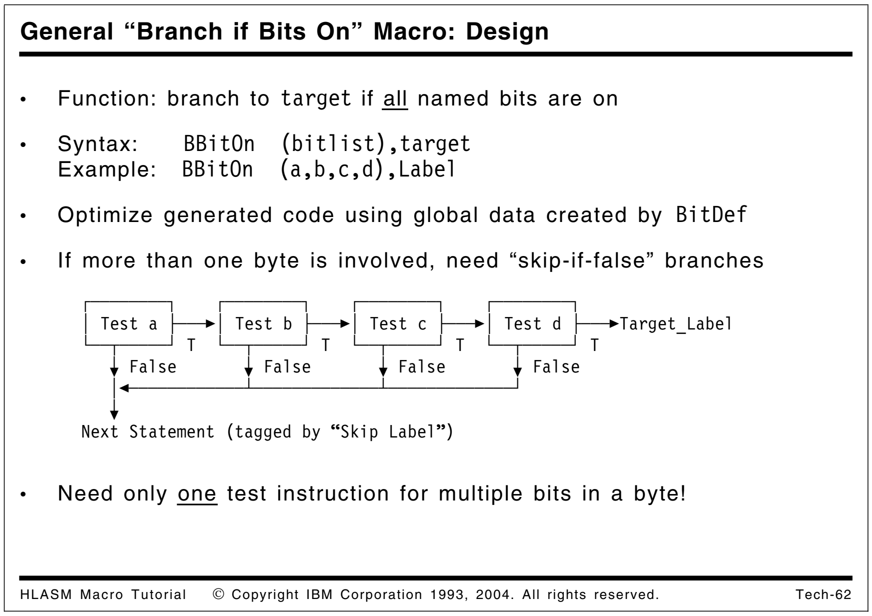
Figure 65 (Part 2 of 2). Bit-Handling Macros: Set Bits ON

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

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

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

 Fbd     BitOn jj              Undeclared bit name 'jj'

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

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

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

Figure 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 55 on page 148), and is left as the traditional “exercise for the reader”.



General “Branch if Bits On” Macro: Design

- 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 global 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 Tech-62

Using Declared Bit Names in a BBitOn Macro

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

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

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

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

```
         DC    B'0'
BitA     Equ   *-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 we see that only a single test instruction is needed. Now, suppose the two bits have been allocated to *distinct* bytes:

```
         DC    B'0'
BitA     Equ   *-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    0H'0'               Label holder for 'skip target'
```

This situation is illustrated in the following “flowchart”:

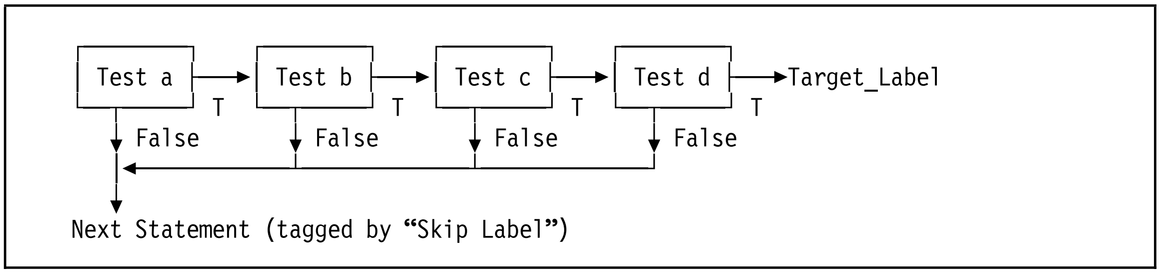
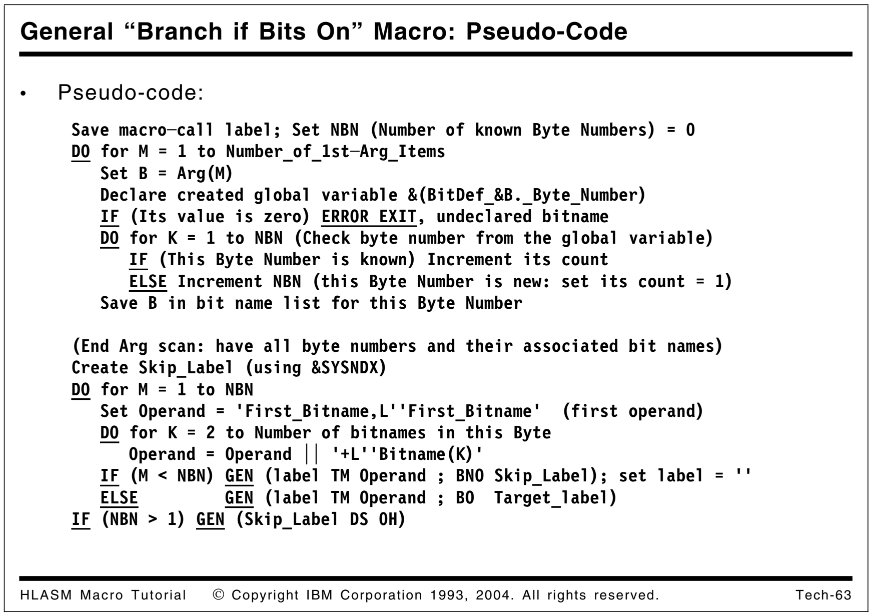


Figure 67. Bit-Handling Macros: Branch if Bits are ON (Flow Diagram)

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 will be extracted, and the same list of variables will be constructed. The second “pass” will need some modifications:

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



General “Branch if Bits On” Macro: Pseudo-Code

- Pseudo-code:

```
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 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
   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 Tech-63

A pseudo-code description of the BBitOn macro is shown in Figure 68.

```
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

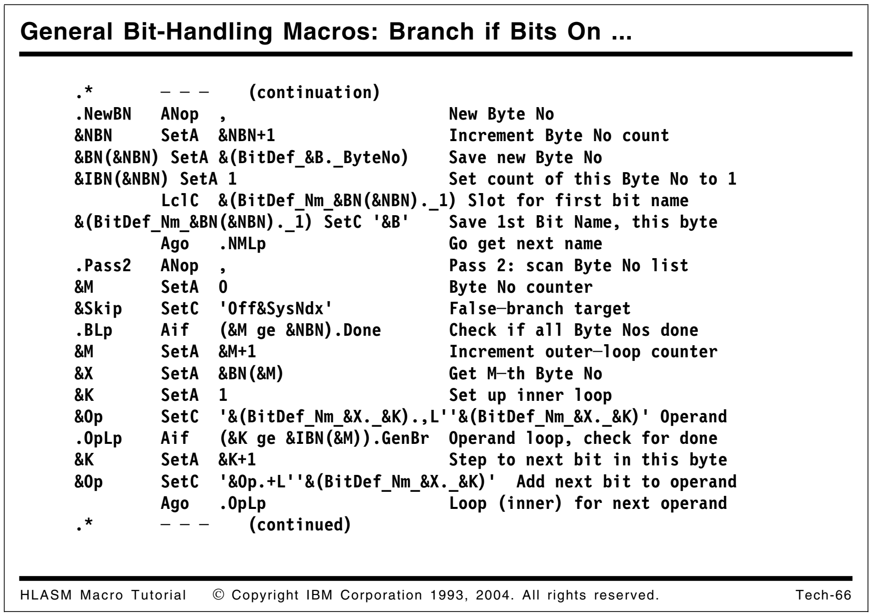
General Bit-Handling Macros: Branch if Bits On

- 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)
```

General Bit-Handling Macros: Branch if Bits On ...

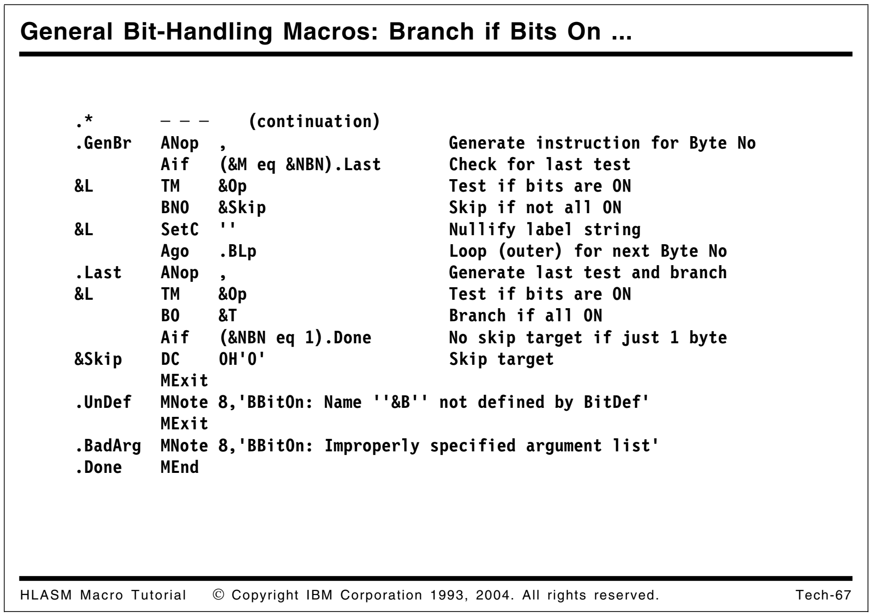
```
.*       – – –    (continuation)
&M       SetA  &M+1                   Step to next name
&B       SetC  '&NL(&M)'              Pick off a name
         GblA  &(BitDef_&B._ByteNo)    Declare GBLA with Byte No.
         Aif   (&(BitDef_&B._ByteNo) eq 0).UnDef  Exit if undefined
&K       SetA  0                      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 ...

```
.*       - - -    (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)
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-66



General Bit-Handling Macros: Branch if Bits On ...

```
.*       - - -    (continuation)
.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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-67

The actual BBitOn macro definition is shown in Figure 69 on page 170.

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

Figure 69 (Part 1 of 2). Bit-Handling Macros: Macro to Branch if Bits are ON

```
         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 are shown in the following figure; the generated instructions are indicated by "+" characters in the left margin:

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

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

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

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

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

Figure 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 also left as an exercise. This 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 170 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 “flowchart”:

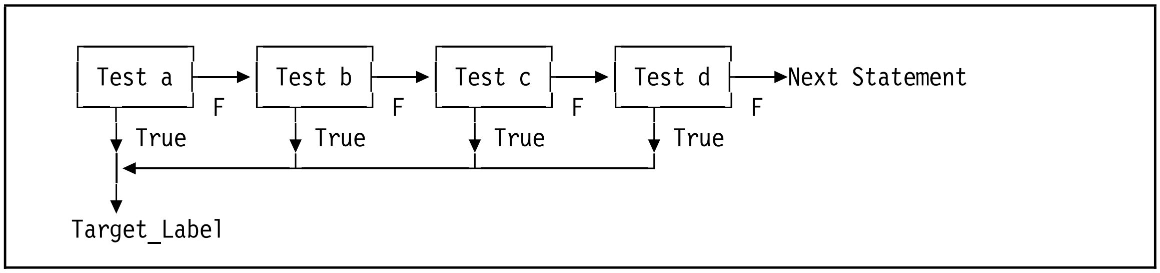
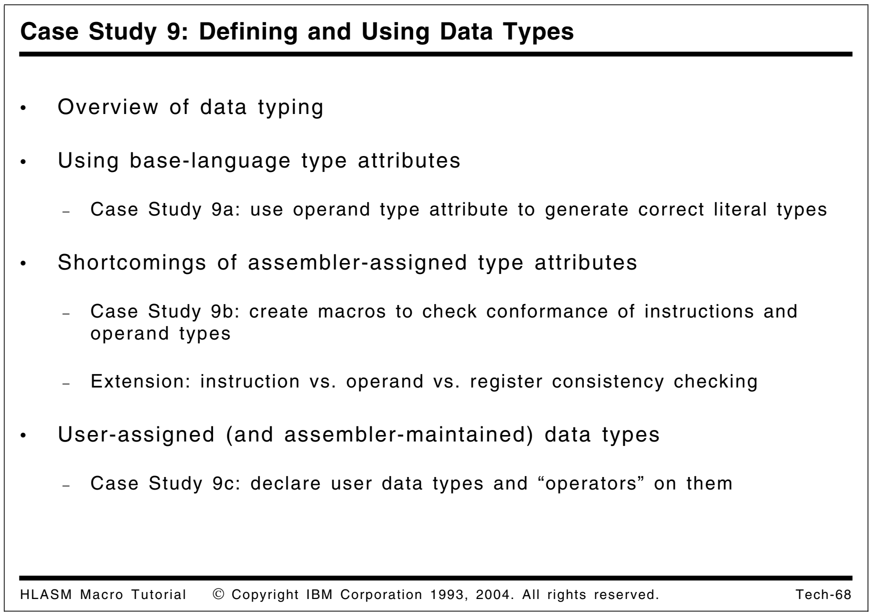


Figure 71. Bit-Handling Macros: Branch if Any Bits are ON (Flow Diagram)

In this “Any” case, no Skip_Label is needed!

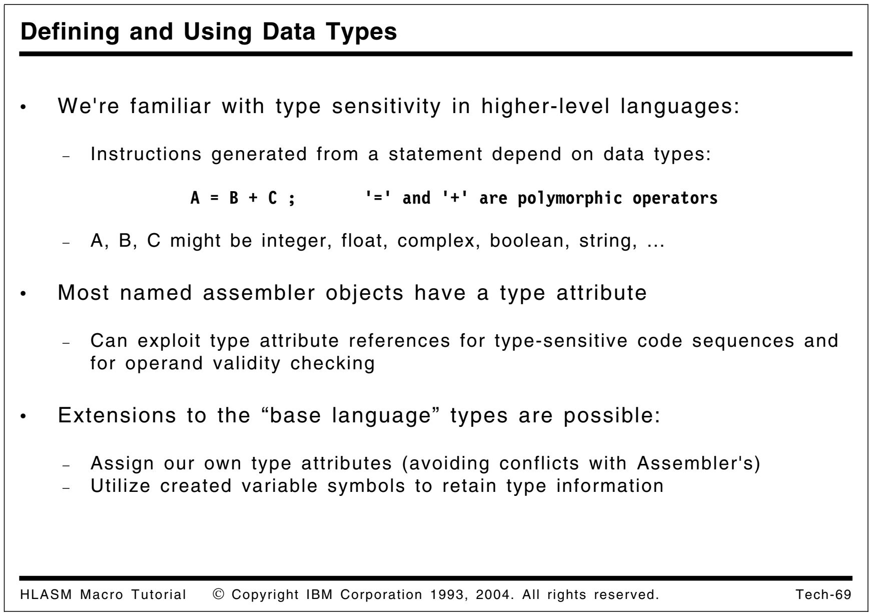


Case Study 9: Defining and Using Data Types

- Overview of data typing
- Using base-language type attributes
  - Case Study 9a: use operand type attribute to generate correct literal types
- Shortcomings of assembler-assigned type attributes
  - Case Study 9b: create macros to check conformance of instructions and operand types
  - Extension: instruction vs. operand vs. register consistency checking
- User-assigned (and assembler-maintained) data types
  - Case Study 9c: declare user data types and "operators" on them

HLASM Macro Tutorial Tech-68

Case Study 9: Defining and Using Data Types



Defining and Using Data Types

- We're familiar with type sensitivity in higher-level languages:
  - Instructions generated from a statement depend on data types:

```
           A = B + C ;       '=' and '+' are polymorphic operators
```

  - A, B, C might be integer, float, complex, boolean, string, ...
- Most named assembler objects have a type attribute
  - Can exploit type attribute references for type-sensitive code sequences and for operand validity checking
- Extensions to the "base language" types are possible:
  - Assign our own type attributes (avoiding conflicts with Assembler's)
  - Utilize created variable symbols to retain type information

HLASM Macro Tutorial Tech-69

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

```
A = B + C
```

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

These case studies will show how macros can be used to provide increasingly powerful levels of control over generated code.

- Case study 9a uses the assembler's “native” type attributes to determine what kind of literal should be used in an instruction.
- Case study 9b creates macros that check for consistency between instructions and their operands, utilizing the AINSERT statement to simplify macro creation.
- Case study 9c 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

- Intent: INCR macro increments 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 something
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,Reg=15      Decrease rate by 3
       Incr  MyPay,150.50        Add 150.50 to my salary
JJ     Incr  Dist,-3.16227766    Decrease distance by sqrt(10)
KK     Incr  Wt,-2E4,Reg=6       Decrement weight by 10 tons
       Incr  WXY,2               Test with unsupported type
```

- INCR uses assembler type attribute of &var to create compatible literals
  - type of amt guaranteed to match type of var

Base-Language Type Sensitivity: Simple Polymorphism ...

- Supported types: 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,'INCR: Cannot use type ''&T'' of ''&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
```

Case Study 9a: Type Sensitivity -- Simple Polymorphism

The assembler's assignment of type attributes to most forms of declared data lets us write macros that utilize the type information to make decisions about instructions to be generated.

For example, suppose we want to write a macro INCR to add a constant value to a variable, with default increment 1 if no value is specified in the macro call. Because we know the assembler-assigned type of 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,'INCR: Cannot use type ''&T'' of ''&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

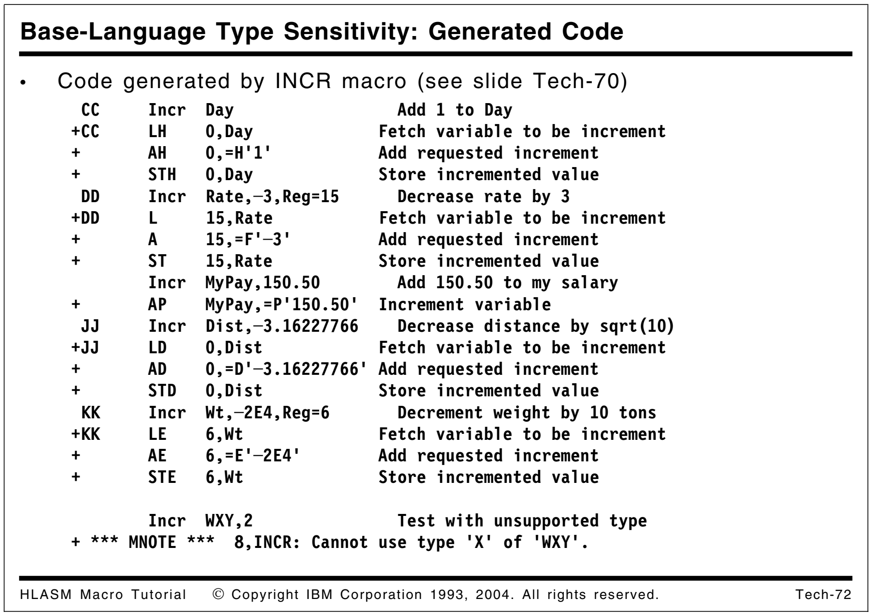
The macro first determines the type attribute of the variable &V, and sets the increment value &I. The type attribute is checked for one of the five allowed types: D, E, F, H, and P. Finally, an instruction sequence appropriate to the variable's type is generated to perform the requested incrementation.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

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 something
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,Reg=15     Decrease rate by 3
       Incr MyPay,150.50       Add 150.50 to my salary
JJ     Incr Dist,-3.16227766   Decrease distance by sqrt(10)
KK     Incr Wt,-2E4,Reg=6      Decrement weight by 10 tons
       Incr WXY,2              Test with unsupported type
```

Figure 73. Examples: Macro Type Sensitivity to Base Language Types



Base-Language Type Sensitivity: Generated Code

- Code generated by INCR macro (see slide Tech-70)

```
 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' of 'WXY'.
```

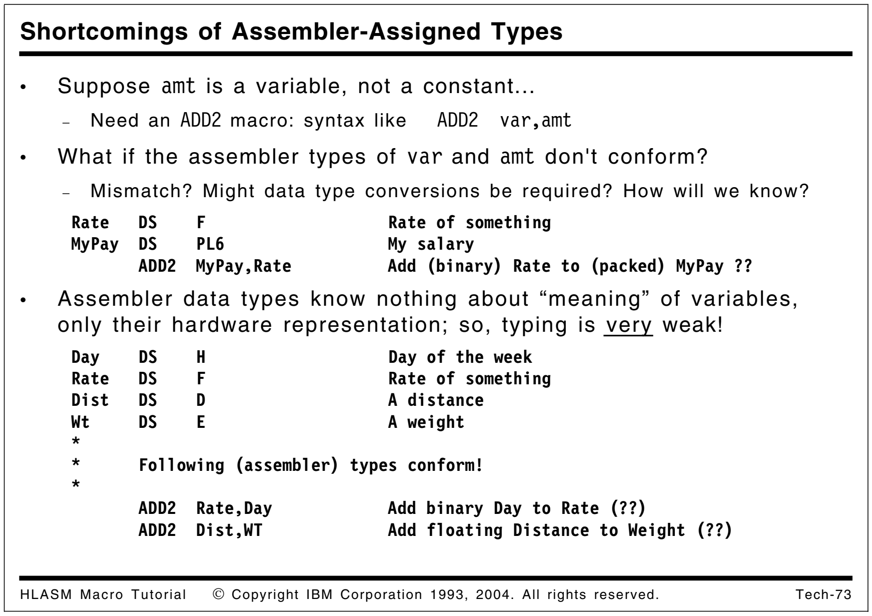
HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-72

Examples of the code generated by the INCR macro are shown in Figure 74 on page 177.

```
 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' of '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.



Shortcomings of Assembler-Assigned Types

- Suppose amt is a variable, not a constant...
  - Need an ADD2 macro: syntax like ADD2 var,amt
- What if the assembler types of var and amt don't conform?
  - Mismatch? Might data type conversions be required? How will we know?

```
Rate   DS    F                   Rate of something
MyPay  DS    PL6                 My salary
       ADD2  MyPay,Rate          Add (binary) Rate to (packed) MyPay ??
```

- Assembler data types know nothing about “meaning” of variables, only their hardware representation; so, typing is very weak!

```
Day    DS    H                   Day of the week
Rate   DS    F                   Rate of something
Dist   DS    D                   A distance
Wt     DS    E                   A weight
*
*      Following (assembler) types conform!
*
       ADD2  Rate,Day            Add binary Day to Rate (??)
       ADD2  Dist,WT             Add floating Distance to Weight (??)
```

HLASM Macro Tutorial Tech-73

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. Suppose, for example, that we wish to add two *variables* using a macro named ADD2 that works like the INCR macro just described. Two problems arise:

1. The types of the variables to be added may not “conform” by having the same assembler-assigned type attribute. For example, let some variables be defined as in Figure 72 on page 175:

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

   Then, if we can write a macro call like

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

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

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

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

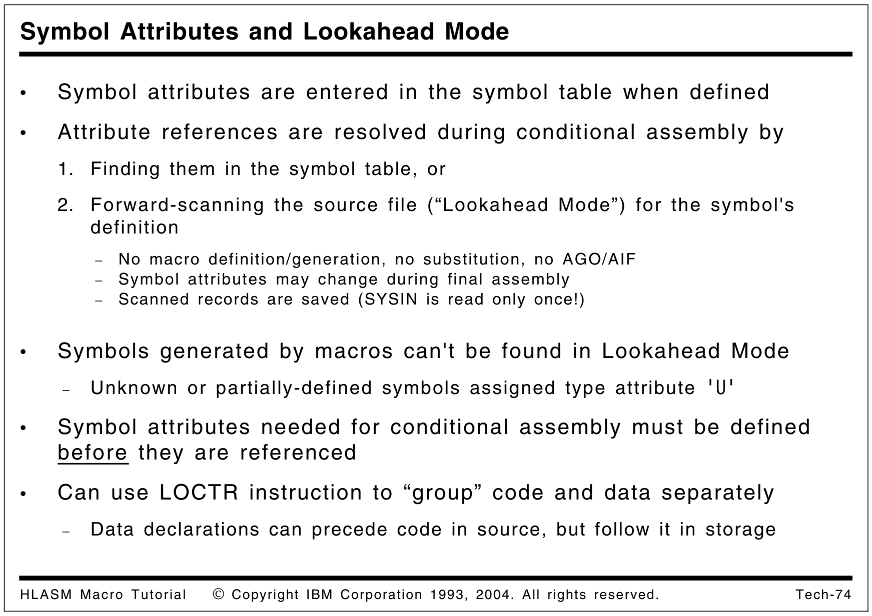
   Then, it is clear that we can write simple macros to implement these additions:

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

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

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

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.



Symbol Attributes and Lookahead Mode

- Symbol attributes are entered in the symbol table when defined
- Attribute references are resolved during conditional assembly by
  1. Finding them in the symbol table, or
  2. Forward-scanning the source file ("Lookahead Mode") for the symbol's definition
     - No macro definition/generation, no substitution, no AGO/AIF
     - Symbol attributes may change during final assembly
     - Scanned records are saved (SYSIN is read only once!)
- Symbols generated by macros can't be found in Lookahead Mode
  - Unknown or partially-defined symbols assigned type attribute `'U'`
- Symbol attributes needed for conditional assembly must be defined before they are referenced
- Can use LOCTR instruction to "group" code and data separately
  - Data declarations can precede code in source, but follow it in storage

Symbol Attributes and Lookahead Mode

There is one potential problem in utilizing attribute information in conditional assembly: the attributes might not be available at the time they are needed. For example, a statement with a symbol definition might occur later in the source file than a macro that references that 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" (later conditional assembly statements might choose among several possible definitions). When the assembly completes, the attributes of a symbol might be different from the attributes assigned during Lookahead mode.

The straightforward solution is to execute 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 provides a simple technique for "grouping" related segments of the object code: the LOCTR statement.

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:

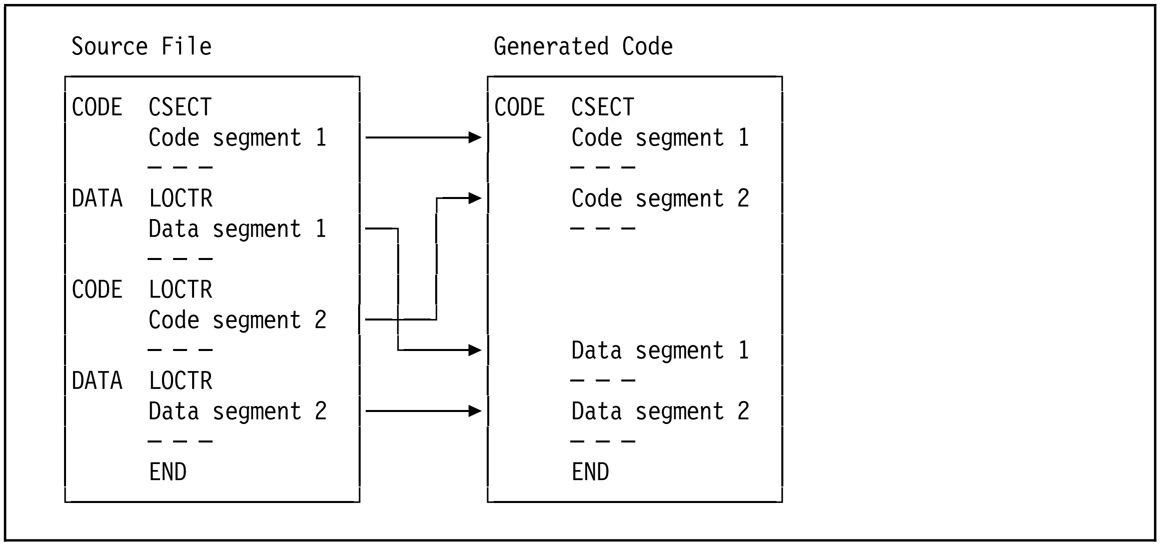
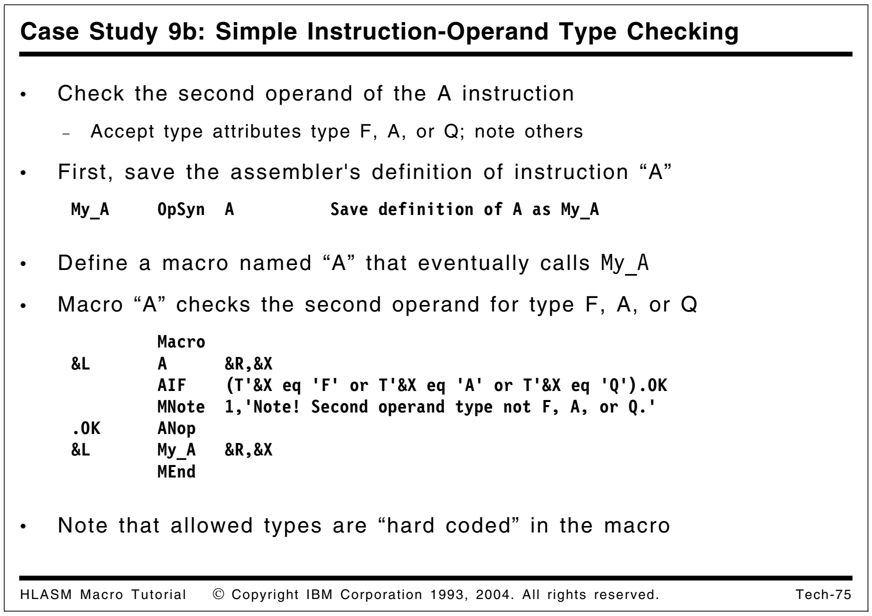


Figure 75. Using the LOCTR Statement to “Group” Code and Data



Case Study 9b: Simple Instruction-Operand Type Checking

- Check the second operand of the A instruction
  - Accept type attributes type F, A, or Q; note others
- First, save the assembler's definition of instruction “A”

```
 My_A     OpSyn  A         Save definition of A as My_A
```

- 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
```

- Note that allowed types are “hard coded” in the macro

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-75

Case Study 9b: 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.

As an example, suppose we wish to check the second operand of the Add (A) instruction to verify that its type is only F, A, or Q. After preserving the original definition of the A opcode as My_A with an OPSYN, we could write a macro like the following:

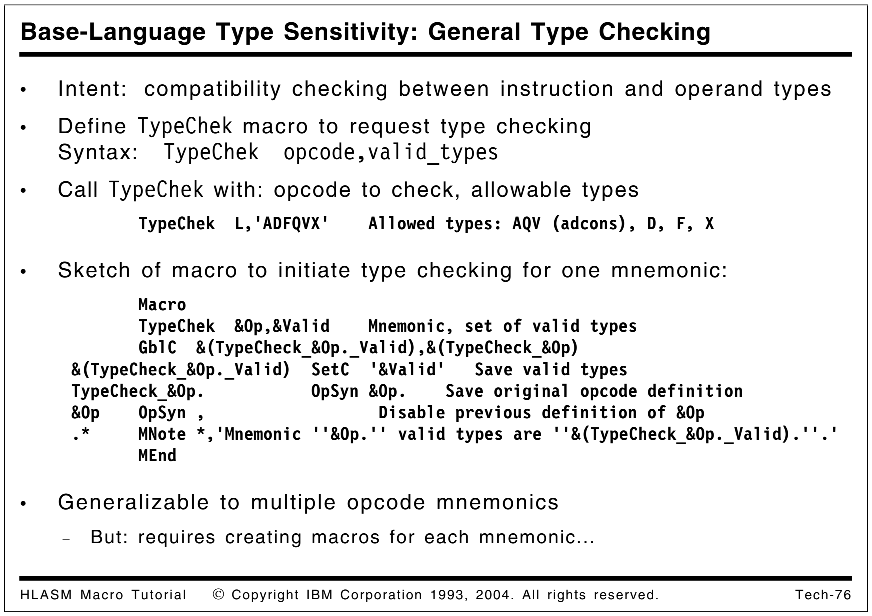
```
My_A     OpSyn  A          Save assembler's definition of A

         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
```

This simple macro requires a non-macro statement (OPSYN) to preserve the assembler's definition of the A instruction. 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

- Intent: compatibility checking between instruction and operand types
- Define TypeChek macro to request type checking
  Syntax: TypeChek opcode,valid_types
- Call TypeChek with: opcode to check, allowable types

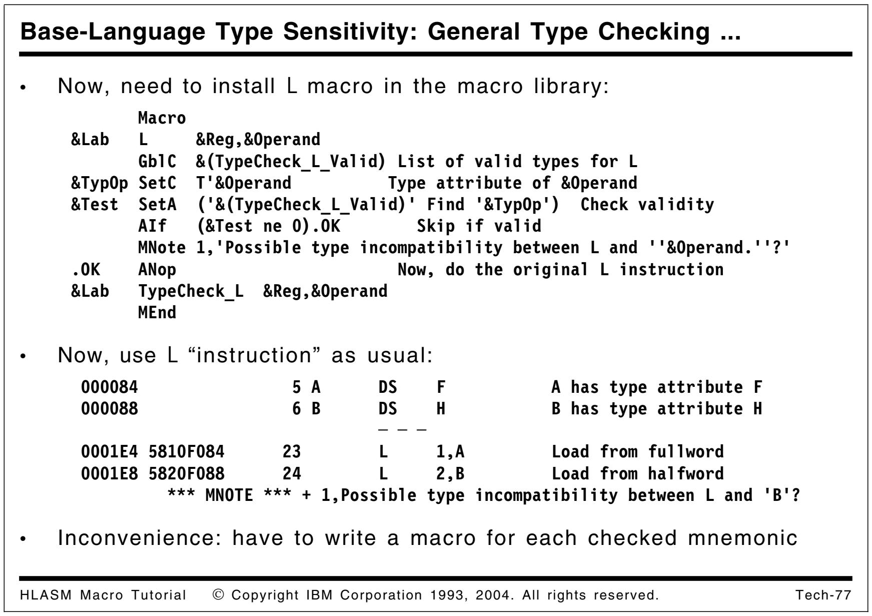
```
        TypeChek  L,'ADFQVX'    Allowed types: AQV (adcons), D, F, X
```

- Sketch of 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
 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
```

- Generalizable to multiple opcode mnemonics
  - But: requires creating macros for each mnemonic...

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-76



Base-Language Type Sensitivity: General Type Checking ...

- Now, need to install L macro in the macro library:

```
        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  ('&(TypeCheck_L_Valid)' Find '&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
```

- Now, use L “instruction” as usual:

```
   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'?
```

- Inconvenience: have to write a macro for each checked mnemonic

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-77

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
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  ('&(TypeCheck_L_Valid)' Find '&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

- Previous technique requires writing 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, such as

    ```
    TypeChek  (L,ST,A,AL,S,SL,N,X,O),'ADFQVX'
    ```

  - The TypeChek macro generates the replacement macros as needed

HLASM Macro Tutorial Tech-78

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 attractive possibility would be to have the TypeChek macro generate the “mnemonic” macros for the mnemonics to be checked, as they will all have 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 Definition in High Level Assembler” on page 64.) The ability to parameterize generated macros would make it much easier to create the “mnemonic” macros directly.

This shortcoming has been eliminated by the AINSERT statement introduced with HLASM Release 3.

The AINSERT Statement

- AINSERT allows generation of fully parameterized records

```
          AINSERT 'string',[FRONT|BACK]
```

- Placed at front or back of assembler's internal buffer queue
  - HLASM pads or truncates string 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 1.4.0
```

  - The '>' character in “column 0” indicates AINSERTed statement
- We will use AINSERT to generate macro definitions

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-79

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 you like, at the same time removing many of the restrictions associated with encoding.

The syntax of AINSERT is

```
          AINSERT 'string',[FRONT|BACK]
```

The first operand may contain points of substitution.

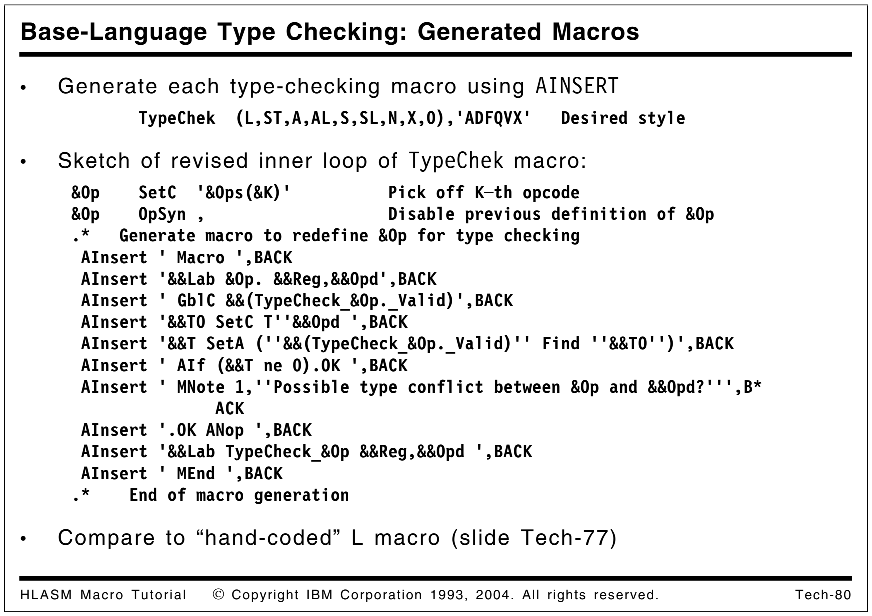
The assembler maintains an internal buffer queue into which AINSERT strings are placed, padded or truncated to an 80-byte record. Each record is placed either at the front or back end of the buffer, depending on the second AINSERT operand. When the assembler is ready to read records from the primary input (SYSIN) file, it first checks the AINSERT buffer: if non-empty, records are taken from the buffer until it is empty, and input then resumes from the primary input stream.

This technique removes many limitations on substitutable fields:

```
          AInsert  '* comment about &SysAsm. &SysVer.',BACK
 >* comment about HIGH LEVEL ASSEMBLER 1.4.0
          AInsert  '* Assembled &SYSDatC.',BACK
 +        AInsert  '* Assembled 20000708',BACK
 >* Assembled 20000708
```

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 see how we can use AINSERT to generate the desired instruction-replacement macros as needed.



Base-Language Type Checking: Generated Macros

- 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 opcode
&Op    OpSyn ,                   Disable previous definition of &Op
.*   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 (''&&(TypeCheck_&Op._Valid)'' Find ''&&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
 AInsert ' MEnd ',BACK
.*    End of macro generation
```

- Compare to “hand-coded” L macro (slide Tech-77)

HLASM Macro Tutorial Tech-80

In Figure 77 on page 183 we saw how the “instruction” macro was created for a single mnemonic (L). We can use the AINSERT statement so the TypeChek macro creates such macros for *each* mnemonic.

These examples have used RX-type instructions to show how to set up a type-checking macro. Assuming that we will want to generalize to other instruction types, we will first write a TypChkRX macro (based on the TypeChek macro illustrated above). The same techniques are used, and now we will 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 opcode
       GblC  &(TypeCheck_&Op._Valid),&(TypeCheck_&Op.)
&(TypeCheck_&Op._Valid) SetC  '&Valid'        Save valid types
&(TypeCheck_&Op.)       SetC  'TypeCheck_&Op.' Create new opcode
&(TypeCheck_&Op.)       OpSyn &Op             Save original opcode
&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 (''&&(TypeCheck_&Op._Valid)'' Find ''&&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
 AInsert ' MEnd ',BACK
.*    End of macro generation
&K     SetA  &K+1                Increment &K
       AIf   (&K le &N).Prcss    If not finished get next opcode
       MEnd
```

Figure 79. 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
```

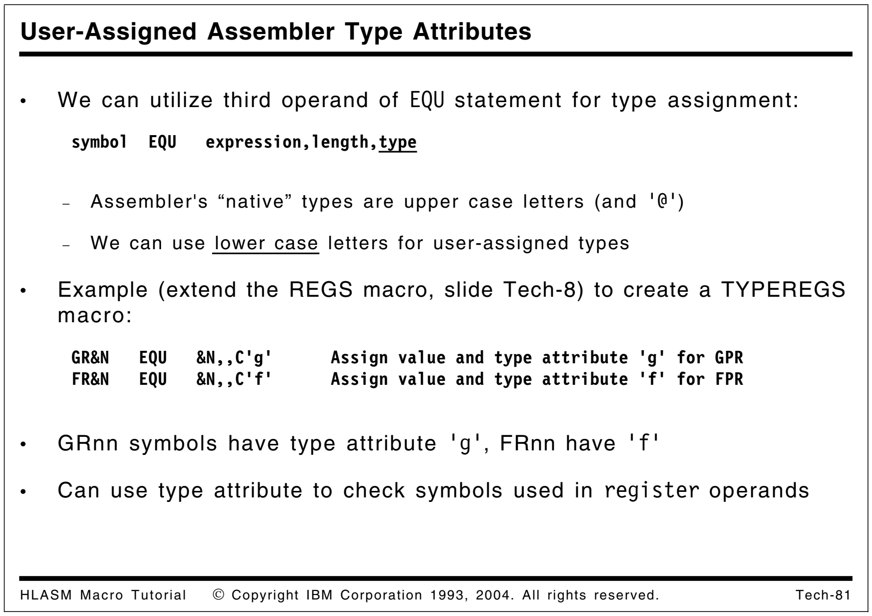
will generate macros for mnemonics L, A, and ST, each of which will validate that their operand types are one of the six allowed types.

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) types in the &Valid operand in the future. If the user has specified the COMPAT(MACROCASE) option, unquoted lower-case letters would be converted 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:

> Macro
>&Lab L &Reg,&Opd
> GblC &(TypeCheck_L_Valid)
>&TO SetC T'&Opd
>&T SetA ('&(TypeCheck_L_Valid)' Find '&TO')
> AIf (&T ne 0).OK
> MNote 1,'Possible type conflict between L and &Opd?'
>.OK ANop
>&Lab TypeCheck_L &Reg,&Opd
> MEnd
> Macro
>&Lab ST &Reg,&Opd

 ... 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

- We can utilize third operand of EQU statement for type assignment:

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

  - Assembler's “native” types are upper case letters (and '@')
  - We can use lower case letters for user-assigned types

- Example (extend the REGS macro, slide Tech-8) 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

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-81

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 or the '@' character, so the other values can be used for user-assigned attributes.

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 29 on page 107) 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.
```

As an example, suppose we want to extend the REGS macro described in “Case Study 1: Defining Equated Symbols for Registers” on page 104 to create a TYPEREGS macro that assigns a special type attribute to the symbols naming each register. Figure 81 on page 190 shows how to do this.

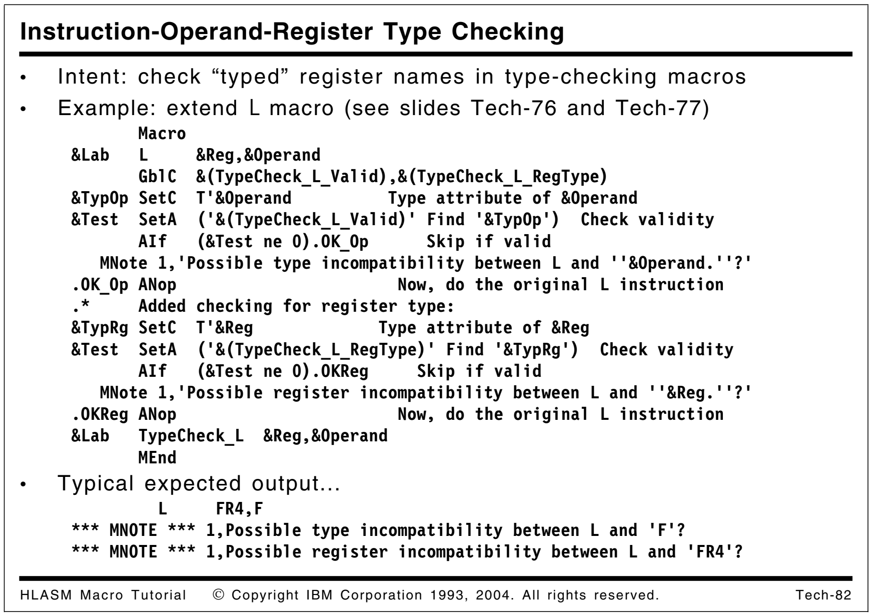
```
        MACRO
        TypeRegs
        AIF     (N'&SysList eq 0).Exit
&J      SetA    1                  Initialize argument counter
.GetArg ANOP
&T      SetC    (Upper '&SysList(&J)')     Pick up an argument
&N      SetA    ('ACFG' Index '&T')        Check type
        AIF     (&N eq 0).Bad     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, 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 the Type-Regs 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

- Intent: check “typed” register names in type-checking macros
- Example: extend L macro (see slides Tech-76 and Tech-77)

```
       Macro
&Lab   L     &Reg,&Operand
       GblC  &(TypeCheck_L_Valid),&(TypeCheck_L_RegType)
&TypOp SetC  T'&Operand          Type attribute of &Operand
&Test  SetA  ('&(TypeCheck_L_Valid)' Find '&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  ('&(TypeCheck_L_RegType)' Find '&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 expected output...

```
         L     FR4,F
*** MNOTE *** 1,Possible type incompatibility between L and 'F'?
*** MNOTE *** 1,Possible register incompatibility between L and 'FR4'?
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-82

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 183) could 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 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 183) 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  ('&(TypeCheck_L_Valid)' Find '&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  ('&(TypeCheck_L_RegType)' Find '&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
```

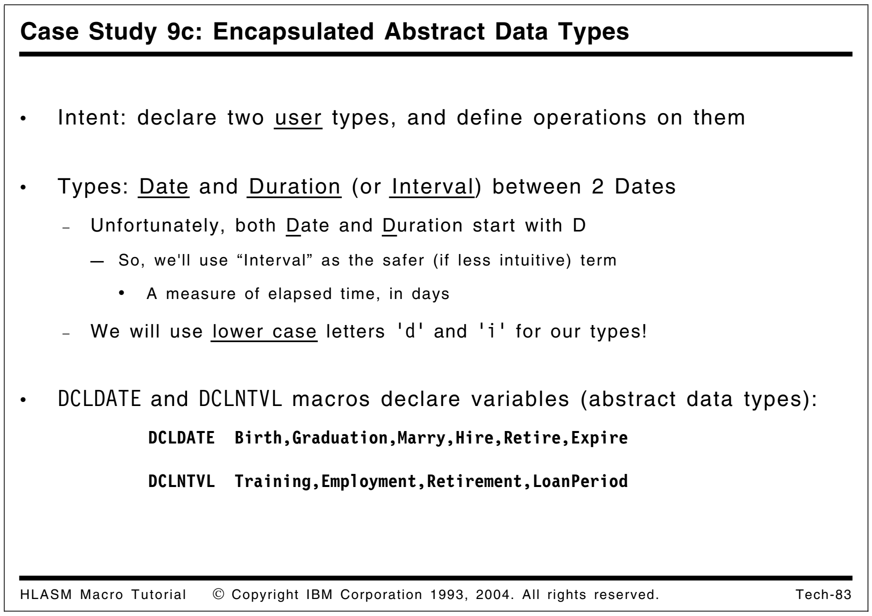
Figure 82. Instruction-Operand-Register Type Checking: “Instruction” Macro

This modification would then check 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 *** Pssible 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 and inclinations.



Case Study 9c: Encapsulated Abstract Data Types

- Intent: declare two user types, and define operations on them
- Types: Date and Duration (or Interval) between 2 Dates
  - Unfortunately, both Date and Duration start with D
    - So, we'll use “Interval” as the safer (if less intuitive) term
      - A measure of elapsed time, in days
  - We will use lower case letters 'd' and 'i' for our types!
- DCLDATE and DCLNTVL macros declare variables (abstract data types):

```
         DCLDATE  Birth,Graduation,Marry,Hire,Retire,Expire

         DCLNTVL  Training,Employment,Retirement,LoanPeriod
```

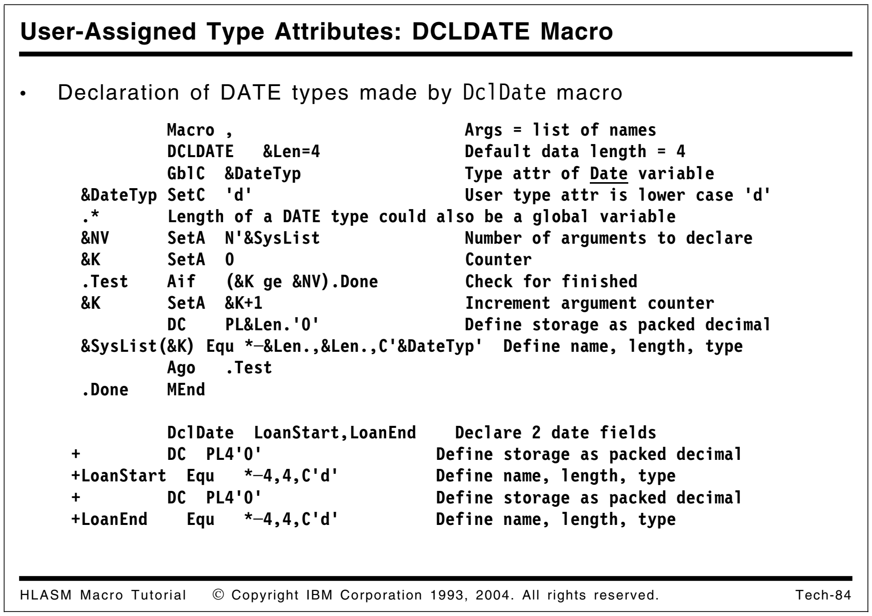
Case Study 9c: Encapsulated Abstract Data Types

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.) With these two data types, we can perform 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
- dates may be compared with dates, and intervals with intervals

Any other operation involving dates and intervals is invalid.

First, we will examine two macros that “declare” variables of type “date” and “interval”, (DCLDATE and DCLNTVL, respectively). Each macro will accept a list of names to be declared with that type, assign “private” type attributes 'd' and 'i', and allocate storage for the variables.



User-Assigned Type Attributes: DCLDATE Macro

- Declaration of DATE types made by DclDate macro

```
         Macro ,                         Args = list of names
         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

         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
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-84

First, we will 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'

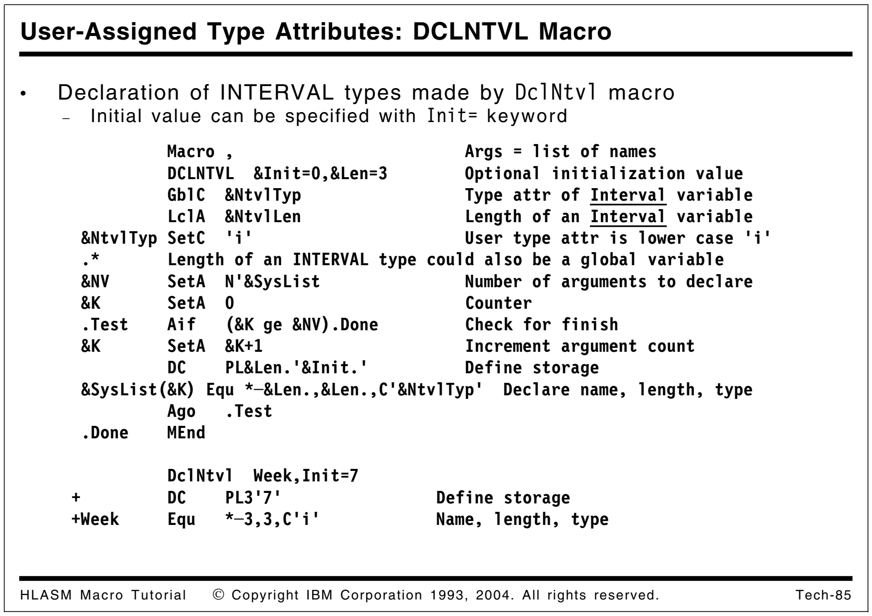
```
         Macro
         DCLDATE   &Len=4               Default data length = 4
         GblC  &DateTyp                 Type attr of Date variable
&DateTyp SetC  'd'                      User type attr is lower case 'd'
.*       Length of a DATE type could also be a global variable
&NV      SetA  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 illustrated 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



User-Assigned Type Attributes: DCLNTVL Macro

- Declaration of INTERVAL types made by DclNtvl macro
  - Initial value can be specified with Init= keyword

```
          Macro ,                        Args = list of names
          DCLNTVL  &Init=0,&Len=3        Optional initialization value
          GblC  &NtvlTyp                 Type attr of Interval variable
          LclA  &NtvlLen                 Length of an 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 arguments 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.'          Define storage
 &SysList(&K) Equ *-&Len.,&Len.,C'&NtvlTyp'  Declare name, length, type
          Ago   .Test
 .Done    MEnd

          DclNtvl  Week,Init=7
+         DC    PL3'7'                Define storage
+Week     Equ   *-3,3,C'i'            Name, length, type
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-85

The DclNtvl macro also accepts a list of names, and allocates a packed decimal field of 3 bytes for each, which we will 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 the variables declared on any one macro call.

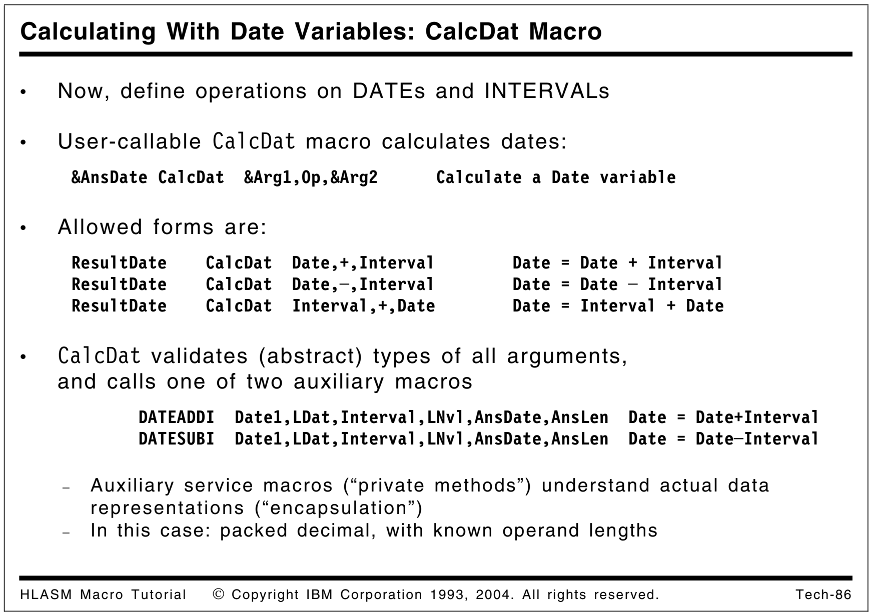
```
          Macro
          DCLNTVL  &Init=0,&Len=3       Optional initialization value
          GblC  &NtvlTyp            Type attr of Interval variable
          LclA  &NtvlLen            Storage length of interval variable
&NtvlTyp SetC  'i'                  User type attr is lower case 'i'
.*       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 196 below:

```
 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

- Now, define operations on DATEs and INTERVALs
- User-callable CalcDat macro calculates dates:

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

- Allowed forms are:

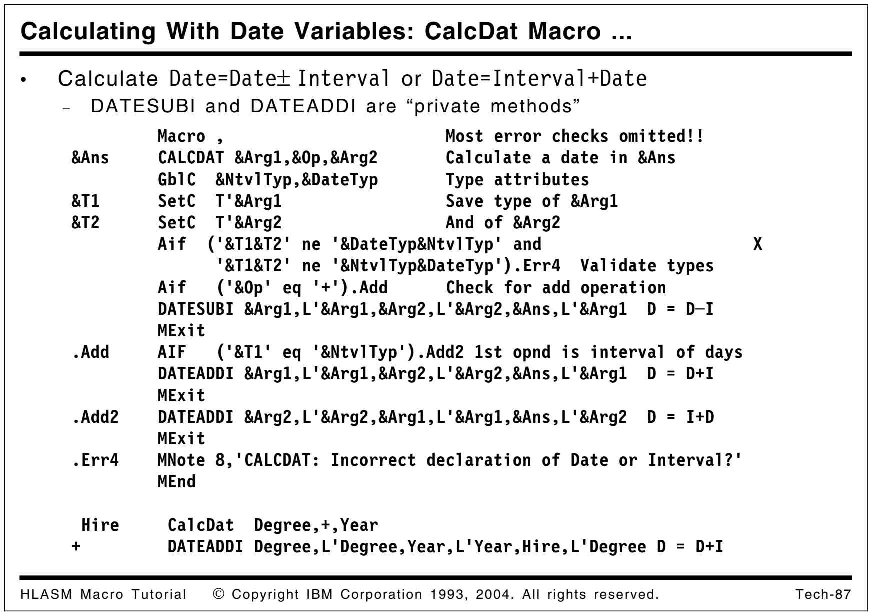
```
ResultDate    CalcDat  Date,+,Interval        Date = Date + Interval
ResultDate    CalcDat  Date,–,Interval        Date = Date – Interval
ResultDate    CalcDat  Interval,+,Date        Date = Interval + Date
```

- CalcDat validates (abstract) types of all arguments, and calls one of two 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”)
  - In this case: packed decimal, with known operand lengths

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-86



Calculating With Date Variables: CalcDat Macro ...

- Calculate Date=Date± Interval or Date=Interval+Date
  - DATESUBI and DATEADDI are “private methods”

```
         Macro ,                      Most error checks omitted!!
&Ans     CALCDAT &Arg1,&Op,&Arg2      Calculate a date in &Ans
         GblC  &NtvlTyp,&DateTyp      Type attributes
&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
         DATESUBI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Arg1  D = D–I
         MExit
.Add     AIF   ('&T1' eq '&NtvlTyp').Add2 1st opnd is interval of days
         DATEADDI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Arg1  D = D+I
         MExit
.Add2    DATEADDI &Arg2,L'&Arg2,&Arg1,L'&Arg1,&Ans,L'&Arg2  D = I+D
         MExit
.Err4    MNote 8,'CALCDAT: Incorrect declaration of Date or Interval?'
         MEnd

 Hire     CalcDat  Degree,+,Year
+         DATEADDI Degree,L'Degree,Year,L'Year,Hire,L'Degree D = D+I
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-87

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'&arg1  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'&Arg1  D = D+I
         MExit
.Add2    DATEADDI &Arg2,L'&Arg2,&Arg1,L'&Arg1,&Ans,L'&Arg2  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
         MNote 8,'&M.Target ''&Ans'' not declared by DCLDATE'
         MExit
.Err3A   MNote 8,'&M.Target Date variable omitted from name field'
         MExit
.Err4    MNote 8,'&M.Incorrect declaration of Date/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 be able to 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 the rather complex in-line code to handle all possible cases.

Calculating Interval Variables: CalcNvl Macro

- 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”):

```
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
DATESUBD  Date1,LDat1,Date2,LDat2,AnsI,AnsLen   Nvl = Date-Date
```

Calculating Interval Variables: CalcNvl Macro ...

```
          Macro
&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:

```
Interval   CalcNvl  Date,-,Date          Difference of two date variables
Interval   CalcNvl  Interval,+,Interval  Sum of two interval variables
Interval   CalcNvl  Interval,-,Interval  Difference of two interval variables
Interval   CalcNvl  Interval,*,Number    Product of an interval and a number
Interval   CalcNvl  Interval,/,Number    Quotient of an interval and a number
```

The CalcNvl macro validates its arguments before generating calls to the “operational” macros that do the actual arithmetic.

```
         Macro
&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
         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
         MNote 8,'&M.Target variable omitted'
         MExit
.Err2A   MNote 8,'&M.Target ''&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

Note that this macro 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.

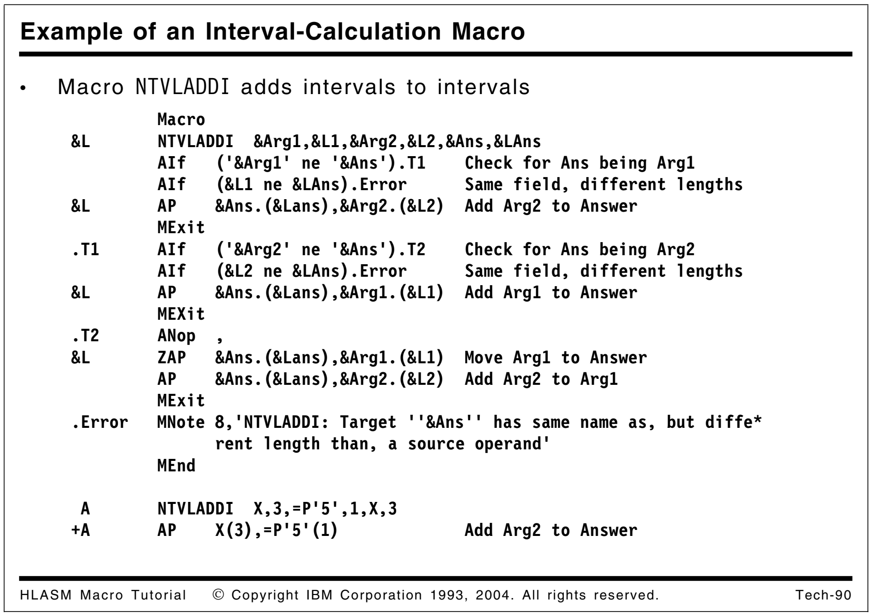
The calls to the “private” NTVLxxxI macros are generated with a form of “associative indirect addressing” by using the single-character operator (such as + or −) as an index into a four-entry “table” of strings specifying which macro name will be generated.

```
 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

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

It is clear that these macros can be extended to avoid using the Assembler's (rather limited) type-attribute mechanism, by maintaining global data structures containing information such as a programmer-declared type, length, and so forth.



Example of an Interval-Calculation Macro

- 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 ''&Ans'' has same name as, but diffe*
               rent 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-90

The “service” macros for handling intervals will probably be much simpler than those for dates (except for DATESUBD, which subtracts two dates to yield an interval, and therefore must account for the choice of date representation, leap years, and the like). As an example of an interval-handling macro, consider the possible 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 ''&Ans'' has same name as, but diffe*
               rent length than, a source operand'
         MEnd
```

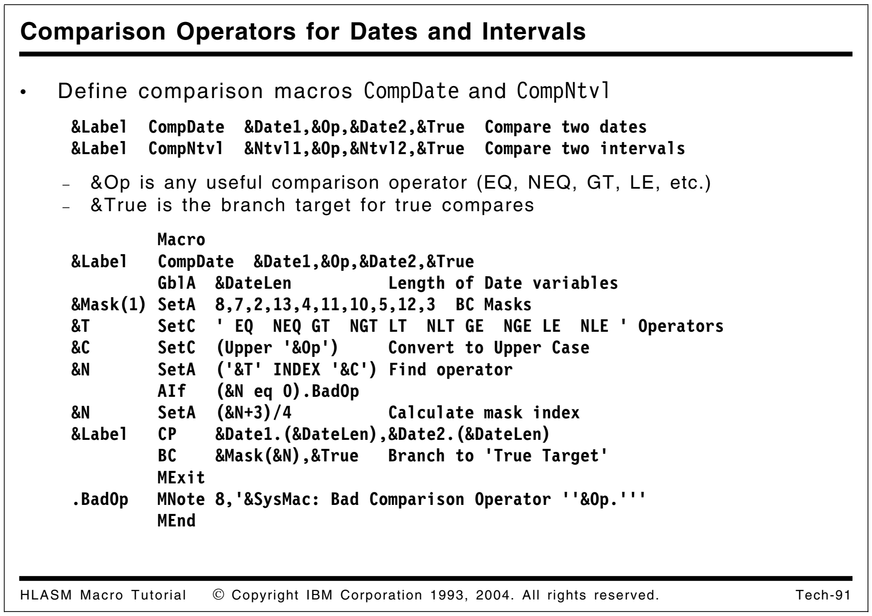
Figure 91. Macro to Add an Interval to an Interval

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 '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

- 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  ('&T' INDEX '&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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-91

Comparison Operators for Dates and Intervals

One further set of functions is needed to complete the 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, NEW, 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. As an example, the CompDate macro could be written as follows:

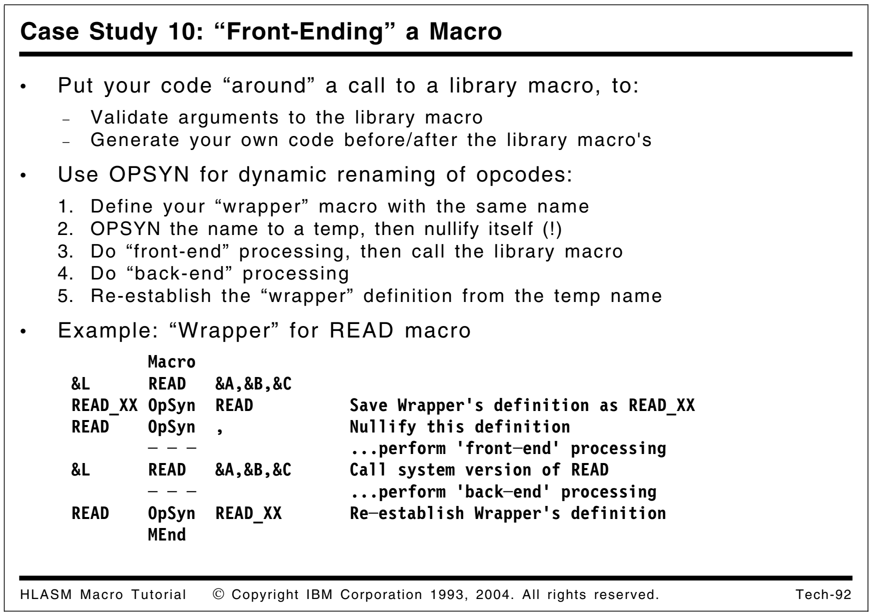
```
         Macro
&Label   CompDate  &Date1,&Op,&Date2,&True
         GblA  &DateLen          Length of Date variables
&Mask(1) SetA  8,7,2,13,4,11,10,5,12,3  BC Masks
&T       SetC  ' EQ  NEQ GT  NGT LT  NLT GE  NGE LE  NLE ' Operators
&C       SetC  (Upper '&Op')     Convert to Upper Case
&N       SetA  ('&T' INDEX '&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.

The code generated by the macro is 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 10: “Front-Ending” a Macro

- Put your code “around” a call to a library macro, to:
  - Validate arguments to the library macro
  - Generate your own code before/after the library macro's
- Use OPSYN for dynamic renaming of opcodes:
  1. Define your “wrapper” macro with the same name
  2. OPSYN the name to a temp, then nullify itself (!)
  3. Do “front-end” processing, then call the library macro
  4. Do “back-end” processing
  5. Re-establish the “wrapper” definition from the temp name
- Example: “Wrapper” for READ macro

```
        Macro
&L      READ   &A,&B,&C
READ_XX OpSyn  READ          Save Wrapper's definition as READ_XX
READ    OpSyn  ,             Nullify this definition
        - - -                ...perform 'front-end' processing
&L      READ   &A,&B,&C      Call system version of READ
        - - -                ...perform 'back-end' processing
READ    OpSyn  READ_XX       Re-establish Wrapper's definition
        MEnd
```

HLASM Macro Tutorial Tech-92

Case Study 10: “Front-Ending” a 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. 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.

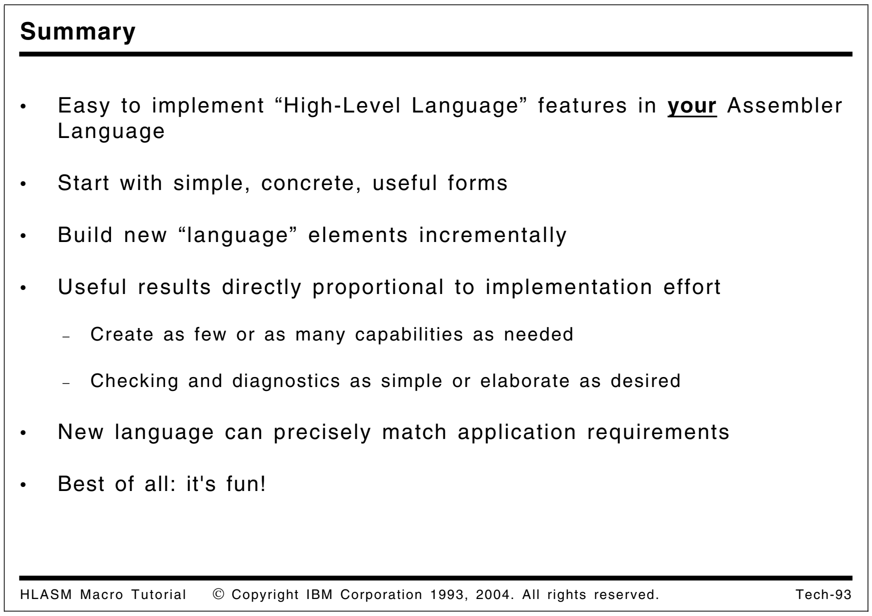
To illustrate, suppose we want to “front-end” the READ macro, as shown in Figure 93 on page 206 below:

```
        Macro
&L      READ   &A,&B,&C
READ_XX OpSyn  READ          Save Wrapper's definition as READ_XX
READ    OpSyn  ,             Nullify this definition
        - - -                ...perform 'front-end' processing
&L      READ   &A,&B,&C      Call system version of READ
        - - -                ...perform 'back-end' processing
READ    OpSyn  READ_XX       Re-establish Wrapper's definition
        MEnd
```

Figure 93. 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 the 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

- 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 © Copyright IBM Corporation 1993, 2004. All rights reserved. Tech-93

Summary

We hope that 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:

- One can start with very simple, concrete examples before attempting complex or abstract designs.
- From a simple base, one can elaborate and extend the macros in many directions, to enhance whatever features are interesting.
- One 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, inter-conversion between them, operations on each (possibly involving mixing of types), and input-output operations (possibly involving conversions to and from “external” representations).[11]

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*.[12]

> 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:

[11] 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!

[12] Reprinted by permission.

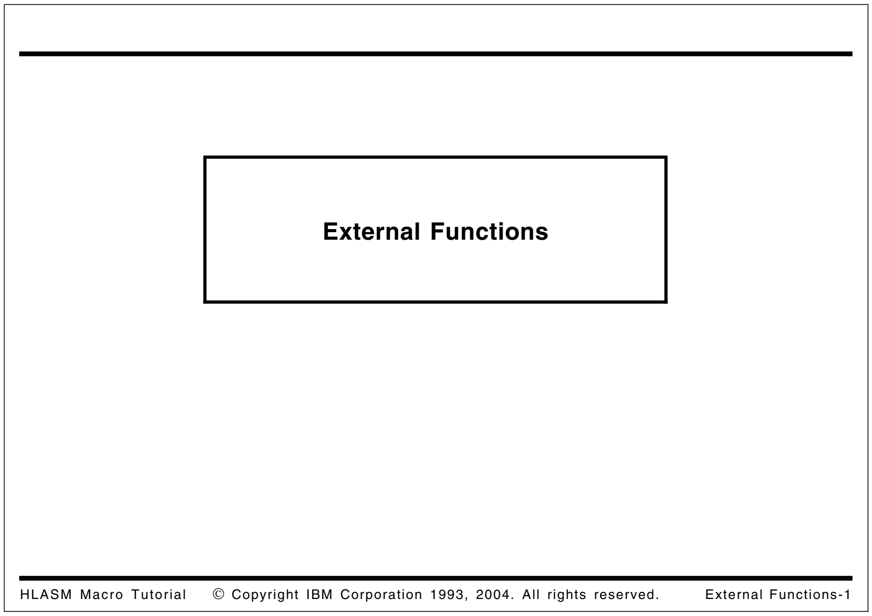
For example, in Year 1 that useless letter “c” would be dropped to be replased by either “k” or “s,” and likewise “x” would no longer be part of the alphabet. The only kase in which “c” would be retained would be the “ch” formation, which will be dealt with later. Year 2 might well reform “w” spelling, 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!

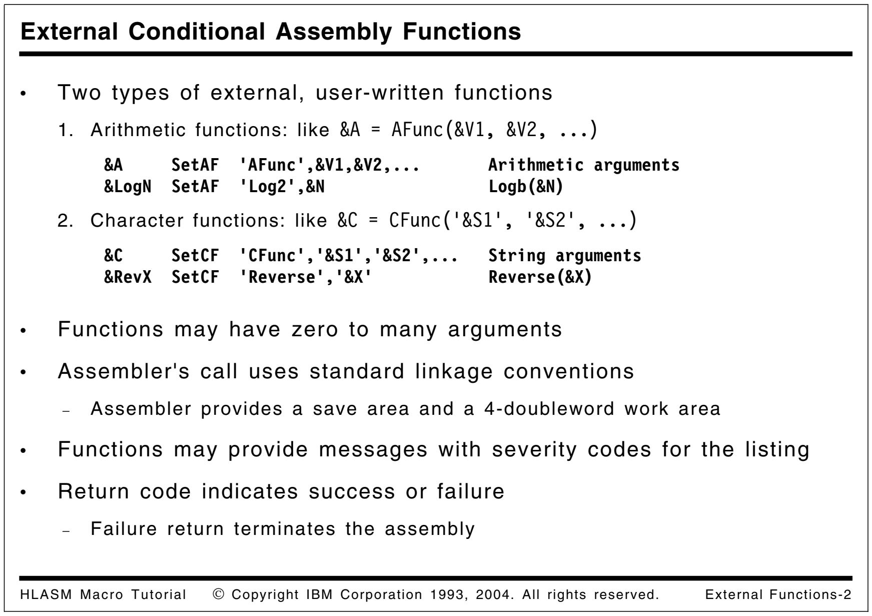
-- Iorz feixfuli, M. J. Yilz.

Appendix A. External Conditional Assembly Functions



External Functions

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. External Functions-1



External Conditional Assembly Functions

- Two types of external, user-written functions
  1. Arithmetic functions: like &A = AFunc(&V1, &V2, ...)

```
&A     SetAF  'AFunc',&V1,&V2,...       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 may have zero to many arguments
- Assembler's call uses standard linkage conventions
  - Assembler provides a save area and a 4-doubleword work area
- Functions may provide messages with severity codes for the listing
- Return code indicates success or failure
  - Failure return terminates the assembly

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. External Functions-2

External Conditional Assembly Functions

IBM High Level Assembler for MVS & VM & VSE supports a powerful capability for invoking externally-defined functions during the assembly. These functions are known as “conditional-assembly functions”, and 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 parameters, others may require several. The type of each of these parameters is the same as that of the target variable: arithmetic parameters for SETAF, and character parameters 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)
```

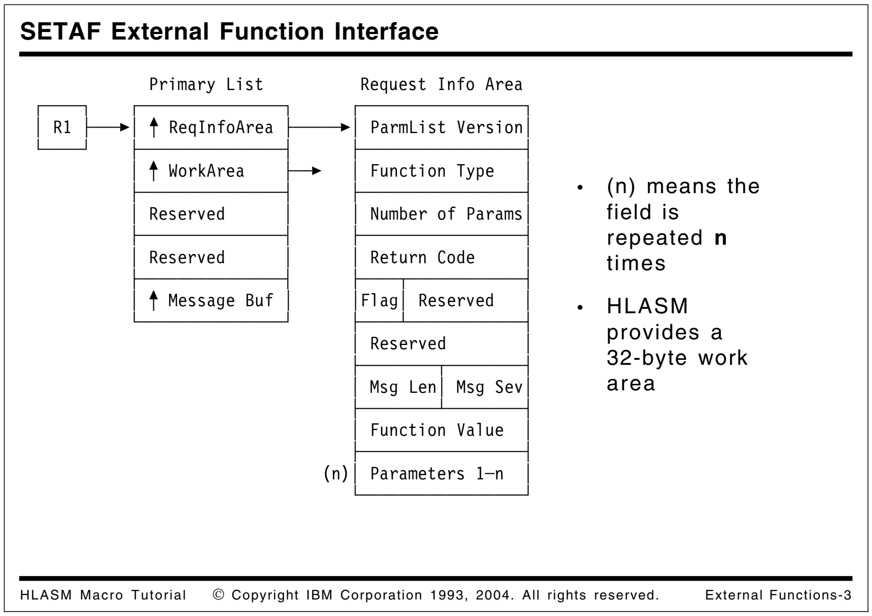
When a function is first invoked, the assembler dynamically loads the module containing the function into working storage, and prepares the necessary control structures for invoking the function. The call to the function uses standard operating system calling conventions; the assembler creates the calling sequence using the parameters and the function name supplied in the SETxF statement.

Following normal parameter-passing conventions, the assembler sets R1 to point to a list of addresses. The first address in this primary list is that of a “Request Information Area”, a list of fullword integer values which describe 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 a means whereby an external function can 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 will check 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 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, external functions have considerably greater power and flexibility than the conditional language can provide.



SETAF External Function Interface

Primary List

Request Info Area

R1

↑ ReqInfoArea

↑ WorkArea

Reserved

Reserved

↑ Message Buf

ParmList Version

Function Type

Number of Params

Return Code

Flag | Reserved

Reserved

Msg Len | Msg Sev

Function Value

(n) Parameters 1–n

- (n) means the field is repeated **n** times
- HLASM provides a 32-byte work area

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. External Functions-3

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 layout of the Primary Address List and the Request Information Area is shown in Figure 94. (Symbolic mappings of the Primary List and the Request Information Area are provided by the ASMAEFNP macro.)

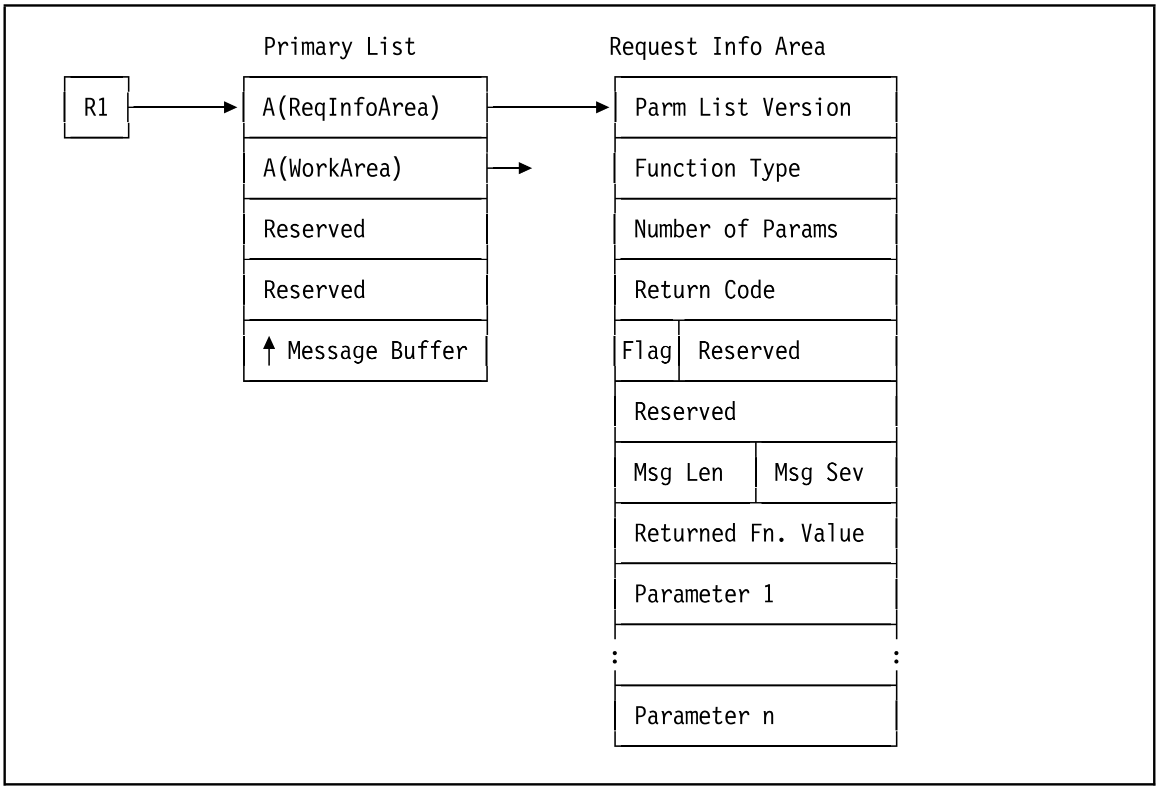


Figure 94. Interface for Arithmetic (SETAF) External Functions

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( log2(x) )
```

This result can be used easily to calculate the actual value of the corresponding power of two. For example, if &Exponent is an arithmetic variable symbol returned by a call to LOG2, the value of the actual power of two can be found using statements such as

```
&Exponent    SETAF  'Log2',&Arith_Var
&Power_of_2  SETA   (1 SLA &Exponent)
```

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 the 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  SETFA  'LOG2',&Int_Arg                                    *
*                                                                     *
*       If &Int_Expr > 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 95. Conditional-Assembly Function LOG2: Initial Commentary

The block of comments in Figure 95 describes the operation of the function, the returned function values, and return codes.

```
**********************************************************************
*       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 96. Conditional-Assembly Function LOG2: Entry

The entry point instructions illustrated in Figure 96 saves appropriate general registers, and establishes 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
         BNE   Err_LVer               Branch if wrong PList version
         CLC   AEFNTYPE,=A(AEFNSETAF) Check for SETAF function call
         BNE   Err_FTyp               Branch if wrong function type
         CLC   AEFNUMBR,=A(OurNArgs)  Check for single argument
         BNE   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 non-negative argument
         BZ    Zero_Arg            Branch if zero argument
         BM    Neg_Arg             Branch if negative argument
```

Figure 97. Conditional-Assembly Function LOG2: Validation

The instructions illustrated in Figure 97 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. (Naturally, these checks could be eliminated if efficiency is a major concern.)

```
*********************************************************************
*        Calculate Floor(Log2(argument)) in R0                      *
*********************************************************************
         LA    R4,31               Set answer to 1 past max possible
TestLoop DC    0H'0'               Check magnitude of the argument
         BCTR  R4,Null             Count answer down by 1
         BXH   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 98. Conditional-Assembly Function LOG2: Computation

The “computation” of the logarithm itself is quite simple, as shown in Figure 98. The BXH 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 BXH 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    0H'0'                Return for negative argument
         LA    R4,BadArgZ           Point to error message
         B     Err_Exit             And return with a message

Neg_Arg  DC    0H'0'                Return for negative argument
         LA    R4,BadArgN           Point to error message
         B     Err_Exit             And return with a message

**********************************************************************
*        Handle invalid calling sequences                            *
**********************************************************************
Err_LVer DC    0H'0'                Wrong interface version
         MVC   AEFNRETC,=A(AEFNBAD) Can't count on doing a message
         B     Return               Return to Assembler immediately

Err_FTyp DC    0H'0'                Wrong function type
         LA    R4,BadFun            Point to error message
         B     Err_Exit             Return to Assembler

Err_NArg DC    0H'0'                Wrong number of arguments
         LA    R4,BadNum            Point to error message
```

Figure 99. Conditional-Assembly Function LOG2: Error Handling

The error-handling code in Figure 99 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.

```
Err_Exit DC    0H'0'
         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,D0(,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    0H'0'                Return to HLASM
         LM    R2,R4,28(R13)        Restore R2-R4
         Drop  R2,R15               Release addressability
         BR    R14                  Return to assembler

Move_Msg MVC   D0(*-*,R1),D1(R4)    Executed
```

Figure 100. Conditional-Assembly Function LOG2: Error Message Handling

The error-handling code in Figure 100 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 101. Conditional-Assembly Function LOG2: Error Message Handling

Each message text shown in Figure 101 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 102. Conditional-Assembly Function LOG2: Symbol Equates

The equates shown in Figure 102 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) 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 103. Conditional-Assembly Function LOG2: Validation Equates

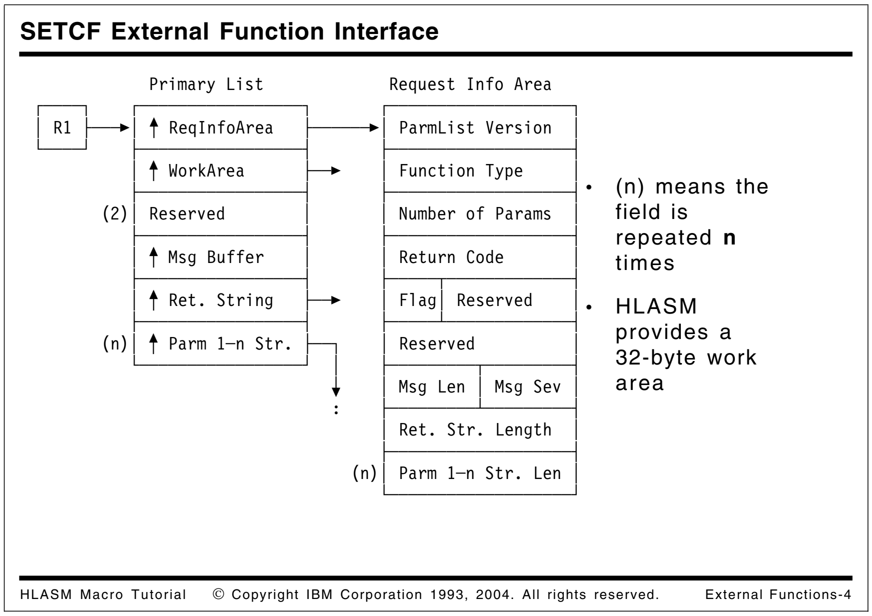
The symbols defined in Figure 103 define the expected value of the number of arguments in the Request Information Area provided by the assembler, and the severity code used for messages. The symbol AEFN_PARM1 is equated to the first item in the argument list; it is used only for its symbolic value.

```
*********************************************************************
*        Dummy Control Sections for SETAF Interface                 *
*********************************************************************
         ASMAEFNP  PRINT=GEN
         End
```

Figure 104. Conditional-Assembly Function LOG2: Dummy Sections

Finally, the Request Information Area is mapped by calling the ASMAEFNP macro supplied with HLASM, as shown in Figure 104.

Installation of the LOG2 function will be described in “Installing the LOG2 and REVERSE Functions” on page 223.



SETCF External Function Interface

- (n) means the field is repeated **n** times
- HLASM provides a 32-byte work area

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. External Functions-4

SETCF External Function Interface

The assembler interface for character functions is illustrated in Figure 105, where the layout of the Primary Address List and the Request Information Area are shown.

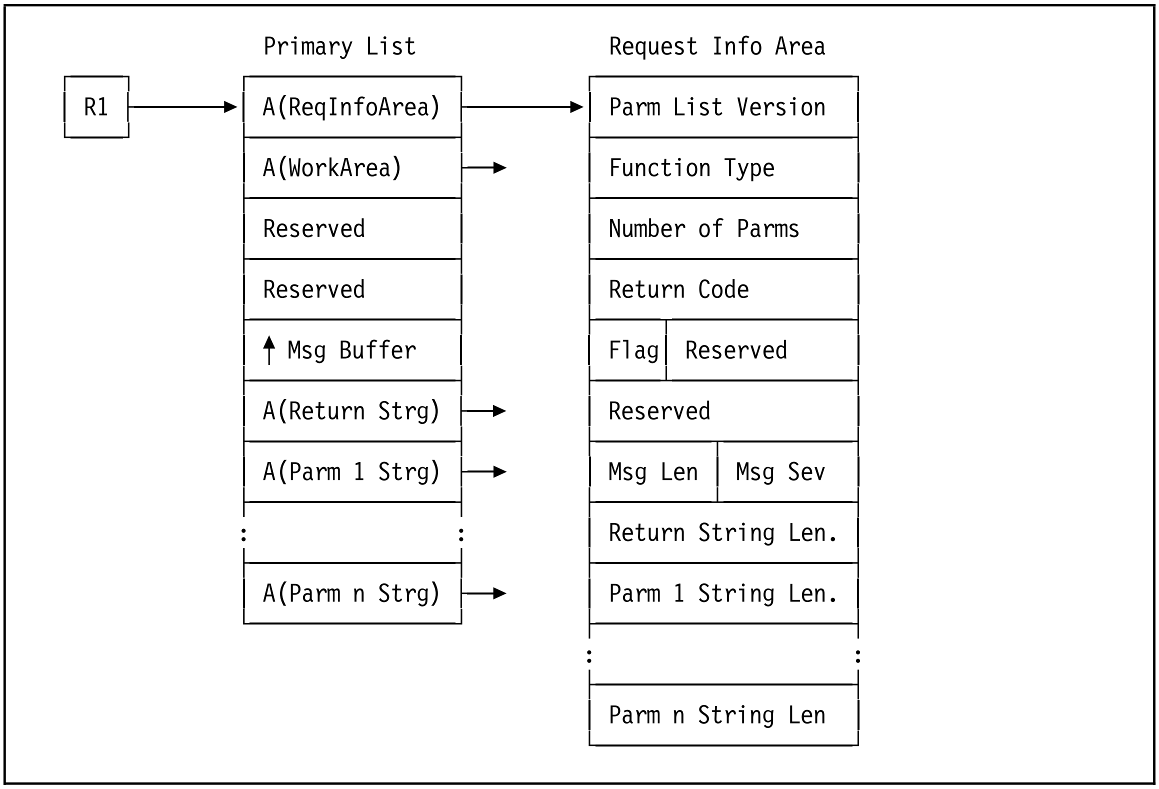


Figure 105. Interface for Character (SETCF) External Functions

String-Valued Function Example: REVERSE

The REVERSE function accepts a single string argument, and returns a string of the same length, but with the characters in reverse order. All valid string lengths are accepted.

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 invalid (not 0-255)           *
*                                                                    *
**********************************************************************
```

Figure 106. Conditional-Assembly Function REVERSE: Prologue Text

The prologue text for the REVERSE function shown in Figure 106 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 ,
REVERSE  RMode Any
REVERSE  AMode Any
         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 107. Conditional-Assembly Function REVERSE: Entry Point

The entry point instructions in Figure 107 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 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
         BNE   Err_LVer               Branch if bad PList version
         CLC   AEFNTYPE,=A(AEFNSETCF) Check for SETCF function call
         BNE   Err_FTyp               Branch if bad function type
         CLC   AEFNUMBR,=A(OurNArgs)  Check for single argument
         BNE   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 108. Conditional-Assembly Function REVERSE: Call Validation

The instructions shown in Figure 108 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
         BM    Err_Arg             Branch if invalid argument
         BZ    Return              Branch if input string is null
         C     R4,=A(OurStMax)     Check for excess length
         BH    Err_Arg             Branch if invalid argument
```

Figure 109. Conditional-Assembly Function REVERSE: Argument Validation

While it should not normally be necessary, the length of the argument string is validated. The instructions shown in Figure 109 should not in fact be required if the assembler is functioning correctly, but the added “insurance” helps avoid further damage that might occur if there is some mis-communication between the function and the assembler.

If efficiency is a major concern, all of 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    0H'0'               Result string was null, just return
         LM    R2,R5,D28(R13)      Restore R2-R5
         BR    R14                 Return to Assembler

Move_TR  DC    0H'0'               Executed, length in R4
         MVC   D0(*-*,R1),D0(R5)   Move trimmed arg to result string

Tran_Ans DC    0H'0'               Executed, length in R4
         TR    D0(*-*,R1),D0(R3)   Translate with reversal into answer
```

Figure 110. Conditional-Assembly Function REVERSE: String Reversal

The instructions in Figure 110 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. 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 there is no guarantee it is available on the system doing the assembly.

```
***********************************************************************
*       Error Returns and Message Handling                            *
***********************************************************************
Err_LVer DC    0H'0'                Unsupported parameter list version
         MVC   AEFNRETC,=A(AEFNBAD) Termination return code
         B     Err_Exit             Return to Assembler

Err_Arg  DC    0H'0'               Return for invalid argument
         LA    R3,InvArg           Point to error message
         B     Err_Msg             Return message to Assembler

Err_FTyp DC    0H'0'               Wrong function type for this call
         LA    R3,BadFun           Point to error message
         B     Err_Msg             Return message to Assembler

Err_NArg DC    0H'0'               Wrong number of arguments
         LA    R3,BadNum           Point to error message

Err_Msg  DC    0H'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    0H'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    0H'0'
         MVC   D0(*-*,R5),D1(R3)   Move message to buffer
```

Figure 111. Conditional-Assembly Function REVERSE: Error Handling

The instructions shown in Figure 111 set the return code for a severe error in case the parameter interface version is not supported. For the 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 112. Conditional-Assembly Function REVERSE: Error Messages

The error message texts (preceded by a length byte) are shown in Figure 112.

```
         Print Data
Trans    DC    0XL256'0',256AL1(255-(*-Trans)) Table from 255 to 0
EndTrans DC    0X'0'               End of translate string

         LtOrg
```

Figure 113. Conditional-Assembly Function REVERSE: Translate Table

The translate table defined in Figure 113 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 114. Conditional-Assembly Function REVERSE: Basic Equates

Standard equates for the general purpose registers are defined in Figure 114, along with three equated 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   255                 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 115. Conditional-Assembly Function REVERSE: Validation Equates

The symbols used in call and argument validation are defined in Figure 115.

```
*********************************************************************
*        Dummy Control Sections for SETCF Interface                 *
*********************************************************************
         ASMAEFNP  PRINT=GEN
         End
```

Figure 116. Conditional-Assembly Function REVERSE: Dummy Sections

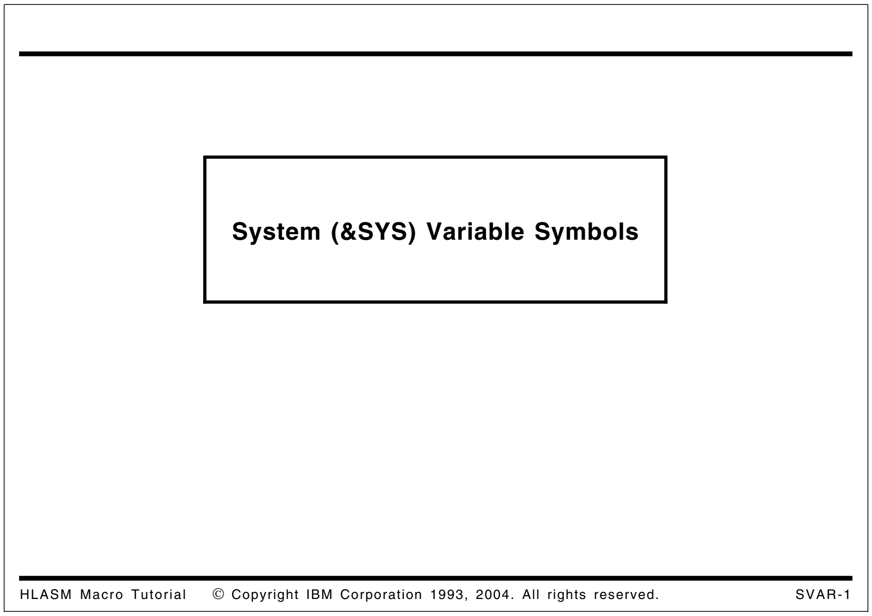
The DSECT mappings for the Primary Address List and the Request Information Area are created by the call to the ASMAEFNP macro, as shown in Figure 116.

Installing the LOG2 and REVERSE Functions

Installing the functions for use during assembly time is simple. First, the statements for the exit are assembled, and the resulting object file is converted 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 filetype MODULE. Be sure that the minidisk containing the function modules is available to the assembler during subsequent assemblies that require the functions.

Appendix B. System (&SYS) Variable Symbols

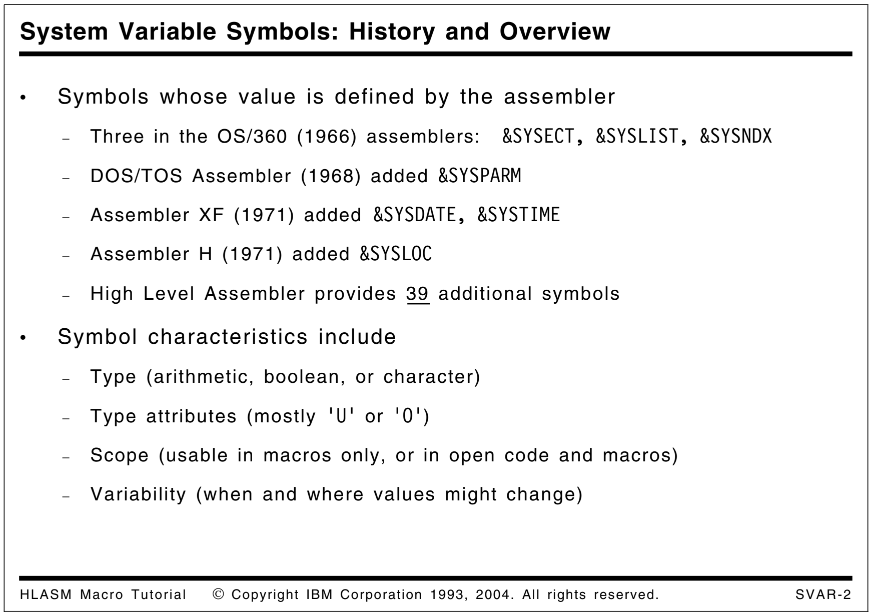


System (&SYS) Variable Symbols

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. SVAR-1

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

High Level Assembler provides many new system variable symbols: thirty-nine will be new to users of the H-Level Assembler, and three additional symbols will be new to users of the DOS/VSE Assembler. Four symbols are available in all three assemblers: &SYSECT, &SYSLIST, &SYSNDX, and &SYSPARM. Figure 117 on page 226 summarizes their properties.



System Variable Symbols: History and Overview

- Symbols whose value is defined by the assembler
  - Three in the OS/360 (1966) assemblers: &SYSECT, &SYSLIST, &SYSNDX
  - DOS/TOS Assembler (1968) added &SYSPARM
  - Assembler XF (1971) added &SYSDATE, &SYSTIME
  - Assembler H (1971) added &SYSLOC
  - High Level Assembler provides 39 additional symbols
- Symbol characteristics include
  - Type (arithmetic, boolean, or character)
  - Type attributes (mostly 'U' or 'O')
  - Scope (usable in macros only, or in open code and macros)
  - Variability (when and where values might change)

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. SVAR-2

System Variable Symbols: Properties

The symbols have a variety of characterizations:

- Availability

  Symbols that were available in Assembler H are designated “AsmH”; High Level Assembler provides a rich set of 39 additional system variable symbols, designated “HLAn” (where “**n**” indicates the release of High Level Assembler in which the symbol first appeared).

- Type

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

- Type attributes

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

- Scope of usage

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

- Variability

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

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

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

  – Variables whose values may change within a single macro expansion are designated “Variable”. Currently, this designation applies only to &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

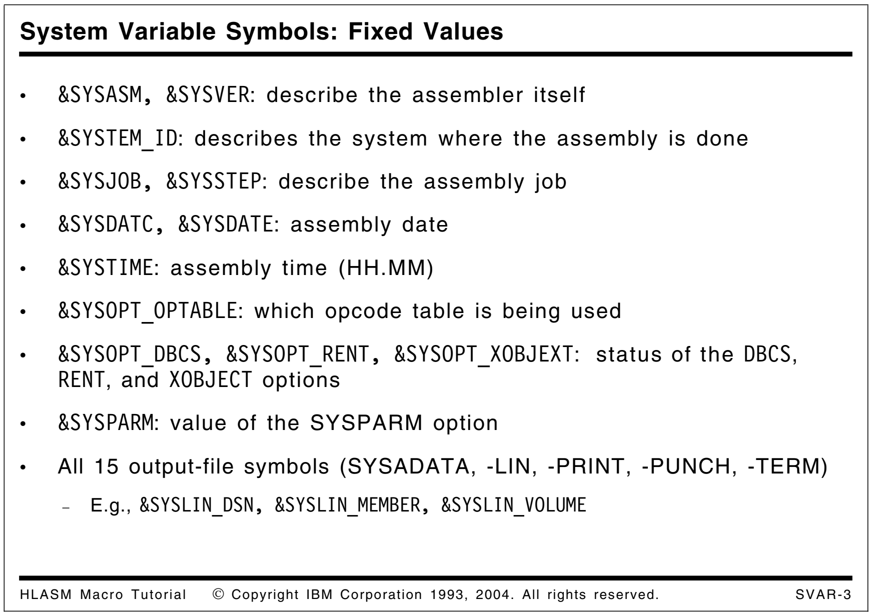
These symbols have many uses: helping to control conditional assemblies, capturing environmental data for inclusion in the generated object code, providing program debugging data, and more.

Figure 117 (Page 1 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	Avail-ability	Type	Type Attr.	Usage Scope	Vari-ability	Content and Use
&SYSADATA_DSN	HLA2	C	U	Local	Fixed	SYSADATA file data set name
&SYSADATA_MEMBER	HLA2	C	U	Local	Fixed	SYSADATA file member name
&SYSADATA_VOLUME	HLA2	C	U	Local	Fixed	SYSADATA file volume identifier
&SYSASM	HLA1	C	U	Global	Fixed	Assembler name
&SYSCLOCK	HLA3	C	U	Local	Constant	Date/time macro was generated
&SYSDATC	HLA1	C,A	N	Global	Fixed	Assembly date, in YYYYMMDD format
&SYSDATE	AsmH	C	U	Global	Fixed	Assembly date in MM/DD/YY format
&SYSECT	AsmH	C	U	Local	Constant	Current control section name
&SYSIN_DSN	HLA1	C	U	Local	Constant	Current primary input data set name
&SYSIN_MEMBER	HLA1	C	U,O	Local	Constant	Current primary input member name
&SYSIN_VOLUME	HLA1	C	U,O	Local	Constant	Current primary input data set name volume identifier
&SYSJOB	HLA1	C	U	Global	Fixed	Assembly job name
&SYSLIB_DSN	HLA1	C	U	Local	Constant	Current library data set name
&SYSLIB_MEMBER	HLA1	C	U,O	Local	Constant	Current library member name
&SYSLIB_VOLUME	HLA1	C	U,O	Local	Constant	Current library data set volume identifier
&SYSLIN_DSN	HLA2	C	U	Local	Fixed	SYSLIN file data set name
&SYSLIN_MEMBER	HLA2	C	U	Local	Fixed	SYSLIN file member name
&SYSLIN_VOLUME	HLA2	C	U	Local	Fixed	SYSLIN file volume identifier
&SYSLIST	AsmH	C	any	Local	Constant	Macro argument list and sublist elements
&SYSLOC	AsmH	C	U	Local	Constant	Current location counter name
&SYSM_HSEV	HLA3	C	N	Global	Variable	Highest MNOTE severity so far in assembly
&SYSM_SEV	HLA3	C	N	Global	Variable	Highest MNOTE severity for most recently called macro
&SYSMAC	HLA3	C	U,O	Local	Constant	Name of current macro and its callers
&SYSNDX	AsmH	C,A	N	Local	Constant	Macro invocation count
&SYSNEST	HLA1	A	N	Local	Constant	Nesting level of the macro call
&SYSOPT_DBCS	HLA1	B	N	Global	Fixed	Setting of DBCS invocation parameter
&SYSOPT_OPTABLE	HLA1	C	U	Global	Fixed	Setting of OPTABLE invocation parameter
&SYSOPT_RENT	HLA1	B	N	Global	Fixed	Setting of RENT invocation parameter

Figure 117 (Page 2 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	**Avail-ability**	**Type**	**Type Attr.**	**Usage Scope**	**Vari-ability**	**Content and Use**
&SYSOPT_XOBJECT	HLA3	B	N	Global	Fixed	Setting of XOBJECT/GOFF invocation parameter
&SYSPARM	AsmH	C	U,O	Global	Fixed	Value provided by SYSPARM invocation parameter
&SYSPRINT_DSN	HLA2	C	U	Local	Fixed	SYSPRINT file data set name
&SYSPRINT_MEMBER	HLA2	C	U	Local	Fixed	SYSPRINT file member name
&SYSPRINT_VOLUME	HLA2	C	U	Local	Fixed	SYSPRINT file volume identifier
&SYSPUNCH_DSN	HLA2	C	U	Local	Fixed	SYSPUNCH file data set name
&SYSPUNCH_MEMBER	HLA2	C	U	Local	Fixed	SYSPUNCH file member name
&SYSPUNCH_VOLUME	HLA2	C	U	Local	Fixed	SYSPUNCH file volume identifier
&SYSSEQF	HLA1	C	U,O	Local	Constant	Sequence field of current open code statement
&SYSSTEP	HLA1	C	U	Global	Fixed	Assembly step name
&SYSSTMT	HLA1	C,A	N	Global	Variable	Number of next statement to be processed
&SYSSTYP	HLA1	C	U,O	Local	Constant	Current control section type
&SYSTEM_ID	HLA1	C	U	Global	Fixed	System on which assembly is done
&SYSTERM_DSN	HLA2	C	U	Local	Fixed	SYSTERM file data set name
&SYSTERM_MEMBER	HLA2	C	U	Local	Fixed	SYSTERM file member name
&SYSTERM_VOLUME	HLA2	C	U	Local	Fixed	SYSTERM file volume identifier
&SYSTIME	AsmH	C	U	Global	Fixed	Assembly start time
&SYSVER	HLA1	C	U	Global	Fixed	Assembler version



System Variable Symbols: Fixed Values

- &SYSASM, &SYSVER: describe the assembler itself
- &SYSTEM_ID: describes the system where the assembly is done
- &SYSJOB, &SYSSTEP: describe the assembly job
- &SYSDATC, &SYSDATE: assembly date
- &SYSTIME: assembly time (HH.MM)
- &SYSOPT_OPTABLE: which opcode table is being used
- &SYSOPT_DBCS, &SYSOPT_RENT, &SYSOPT_XOBJEXT: status of the DBCS, RENT, and XOBJECT options
- &SYSPARM: value of the SYSPARM option
- All 15 output-file symbols (SYSADATA, -LIN, -PRINT, -PUNCH, -TERM)
  - E.g., &SYSLIN_DSN, &SYSLIN_MEMBER, &SYSLIN_VOLUME

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. SVAR-3

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

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

```
HIGH LEVEL ASSEMBLER
```

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

```
1.4.0
```

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..'
```

The value of &SYSVER increases monotonically across versions and releases of HLASM.

&SYSTEM_ID

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

```
MVS/ESA SP 5.2.0
```

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

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

&SYSJOB and &SYSSTEP

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

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

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

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

&SYSDATC

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

```
19920626
```

Observe that the &SYSDATE variable provides only two digits of the year.

&SYSDATE

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

```
06/26/92
```

&SYSTIME

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

This variable, along with &SYSDATE or &SYSDATC, could be used to provide identification within an assembled program of the date and time of assembly:

```
When_ASM  DC  C'Assembled on &SYSDATC., at &SYSTIME..'
```

Differences among &SYSTIME, &SYSCLOCK, and the CLOCKB and CLOCKD operands of the AREAD statement are discussed at “&SYSTIME, &SYSCLOCK, and the AREAD Statement” on page 237.

&SYSOPT_OPTABLE

This variable provides the name of the current operation code table being used for this assembly, as established by the OPTABLE option. A typical value of this variable might be

```
ESA
```

This variable is useful for creating programs that must execute on machines with limitations on the set of available instructions. For macro-generated code, this variable can be used to determine what instructions should be generated for various operations, e.g. BALR vs. BASR.

This variable could be used to provide identification within an assembled program of the operation code table used to assemble it:

```
What_Ops  DC  C'Opcode table for assembly was &SYSOPT_OPTABLE..'
```

&SYSOPT_DBCS, &SYSOPT_RENT, and &SYSOPT_XOBJECT

The &SYSOPT_DBCS, &SYSOPT_RENT, and &SYSOPT_XOBJECT binary variables provide the settings of the DBCS, RENT, and XOBJECT options, respectively. Their values can be used to control the 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.

For example, the &SYSOPT_RENT variable could be used to provide conditional assembly support for different code sequences:

```
          AIF    (&SYSOPT_RENT).Do_Rent
          MYMAC  Parm1,Parm2,GENCODE=NORENT  Generate non-RENT code
          AGO    .Continue
.Do_Rent  MYMAC  Parm1,Parm2,GENCODE=RENT    Generate RENT code
.Continue ANOP
```

&SYSPARM

The &SYSPARM variable symbol provides the character string provided by the programmer in the invoking parameter string, in the SYSPARM option:

```
SYSPARM(string)
```

This variable could be used to provide identification within an assembled program of the &SYSPARM value used to assemble it, as well as to control conditional assembly activities:

```
What_PRM  DC    C'&&SYSPARM value was ''&SYSPARM.''.'

.X14      AIF   ('&SYSPARM' NE 'TRACE').Skip_Trace
          MNOTE 'Assembly reached Sequence Symbol .X14'
.Skip_Trace  ANOP
```

&SYS Symbols for Output Files

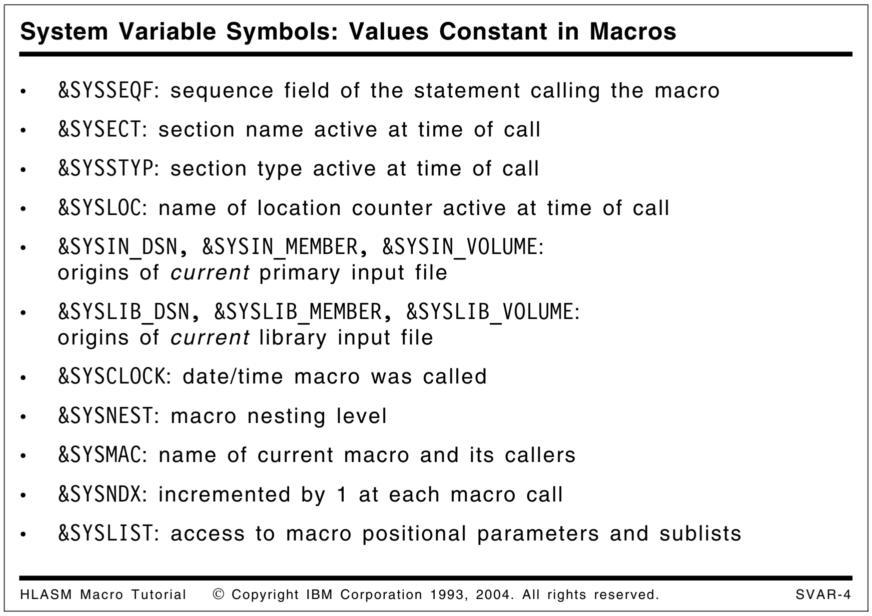
There are fifteen variable symbols describing the output files of High Level Assembler, three for each file:

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 variable symbols provide the file or data set name used for the corresponding output file; the &SYSxxxx_MEMBER variable symbols provide the member name (if any) used for the output file; and the &SYSxxxx_VOLUME variable 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 to do this, 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.''.'
```



System Variable Symbols: Values Constant in Macros

- &SYSSEQF: sequence field of the statement calling the macro
- &SYSECT: section name active at time of call
- &SYSSTYP: section type active at time of call
- &SYSLOC: name of location counter active at time of call
- &SYSIN_DSN, &SYSIN_MEMBER, &SYSIN_VOLUME: origins of *current* primary input file
- &SYSLIB_DSN, &SYSLIB_MEMBER, &SYSLIB_VOLUME: origins of *current* library input file
- &SYSCLOCK: date/time macro was called
- &SYSNEST: macro nesting level
- &SYSMAC: name of current macro and its callers
- &SYSNDX: incremented by 1 at each macro call
- &SYSLIST: access to macro positional parameters and sublists

HLASM Macro Tutorial SVAR-4

Variable Symbols With Constant Values Within a Macro

These sequence symbol values are initialized at the point where a macro expansion is initiated, and remain fixed throughout the duration of that expansion.

&SYSSEQF

The &SYSSEQF symbol provides the contents of the sequence field of the current input statement. This information can be used for debugging data. For example, suppose we have a macro which inserts information about the current sequence field into the object code of the program, and sets R0 to its address (so that a debugger can tell you which statement was identified in some debugging activity). A macro like the following might be used:

```
         Macro
&L       DebugPtA
&L       BAS   0,*+12             Addr of Sequence Field in R0
         DC    CL8'&SYSSEQF'      Sequence Field info
         MEnd
         - - -
B        DebugPtA
```

&SYSECT

The &SYSECT symbol provides the name of the control section (CSECT, DSECT, COM, or RSECT) into which statements are being grouped or assembled at the time the referencing macro was invoked. If a macro must generate code or data in a different control section, this variable permits the macro to restore the name of the previous environment before exiting. (Note also its relation to &SYSSTYP.) An example illustrating &SYSECT and &SYSSTYP is shown below.

&SYSSTYP

The &SYSSTYP symbol provides the type of the control section into which statements are being grouped or assembled (CSECT, DSECT, or RSECT) at the time the referencing macro was invoked. If a macro must generate code or data in a different control section, this variable permits the macro to restore the proper type of control section for the previous environment, before exiting.

For example, suppose we need to generate multiple copies of a small DSECT. The macro shown in the following example generates the DSECT so that each generated name is prefixed with the characters supplied in the macro argument. The environment in which the macro was invoked is then restored on exit from the macro.

```
         Macro
         DSectGen  &P
&P.Sect  DSect ,                 Generate tailored DSECT name
&P.F1    DS     D                DSECT Field No. 1
&P.F2    DS     18F              DSECT Field No. 2, a save area
&SYSECT  &SYSSTYP                Restore original section
         MEnd
```

&SYSLOC

&SYSLOC contains the name of the current location counter, as defined either by a control section definition or a LOCTR statement.

As in the example of &SYSSTYP, the &SYSLOC variable can be used to capture and restore the current location counter name. We again suppose in this example that we are interrupting the statement flow to generate a small DSECT:

```
         Macro
         DSectGen  &P
&P.Sect  DSect                   Generate the DSECT name
&P.F1    DS     D                DSECT Field No. 1
&P.F2    DS     18F              DSECT Save Area
&SYSLOC  LOCTR                   Restore previous location counter
         MEnd
```

&SYSIN_DSN, &SYSIN_MEMBER, and &SYSIN_VOLUME

These three symbols identify the origins of the current primary input file. Their values change across input-file concatenations. This information can be used to determine reassembly requirements.

The &SYSIN_DSN symbol provides the name of the current primary input (SYSIN) data set or file.

The &SYSIN_MEMBER symbol provides the name of the current primary input member, if any.

The &SYSIN_VOLUME symbol provides the name of the current primary input volume. For example, the following SYSINFO macro will capture the name of the current input file, its member name, and the volume identifier. (If the input does not come from a library member, the member name will be replaced by the characters “(None)”.)

```
         Macro
&L       SYSINFO
&L       DC    C'Input: &SYSIN_DSN'
&Mem     SetC  '&SYSIN_MEMBER'
         AIF   ('&Mem' ne '').Do_Mem
&Mem     SetC  '(None)'
.Do_Mem  DC    C'Member: &Mem'
         DC    C'Volume: &SYSIN_VOLUME'
         MEnd
My_Job   SYSINFO
```

&SYSLIB_DSN, &SYSLIB_MEMBER, and &SYSLIB_VOLUME

These three symbols identify the origins of the current library member. Their values change from member to member. This information can be used to determine reassembly requirements.

The &SYSLIB_DSN symbol provides the name of the library data set from which each macro and COPY file is retrieved.

The &SYSLIB_MEMBER symbol provides the name of the library member from which this macro and COPY file is retrieved.

The &SYSLIB_VOLUME symbol provides the volume identifier (VOLID) of the library data set from which this macro and COPY file is retrieved.

For example, suppose the LIBINFO macro below is stored in a macro library accessible to the assembler at assembly time. (The macro includes a test for a blank member name, which should never occur.)

```
         Macro
&L       LIBINFO
&L       DC    C'Library Input: &SYSLIB_DSN'
&Mem     SetC  '&SYSLIB_MEMBER'
         AIF   ('&Mem' ne '').Do_Mem
         MNote 4,'The library member name should not be null.'
.Do_Mem  DC    C'Member: &Mem'
         DC    C'Volume: &SYSLIB_VOLUME'
         MEnd
```

Then the following small test assembly would capture information into the object text of the generated program about the macro library.

```
My_Job   LIBINFO
         End
```

&SYSCLOCK

The &SYSCLOCK character variable 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. Note that &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 at “&SYSTIME, &SYSCLOCK, and the AREAD Statement” on page 237.

&SYSNEST

The &SYSNEST arithmetic variable 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

The &SYSMAC character variable 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    MNOTE *,'Name at nesting level &J is &SYSMAC(&K)'
&J       SetA  &J.-1
         AIf   (&J ge 0).Loop
```

&SYSNDX

The &SYSNDX variable provides a unique value for every macro invocation in the program. It may be used as a suffix for symbols generated in the macro, so that they will not “collide” with similar symbols generated in other invocations. It is incremented by 1 for every macro call in the program.

For values of &SYSNDX less than or equal to 9999, the value will always be four characters long (padded on the left with leading zeros, if necessary).

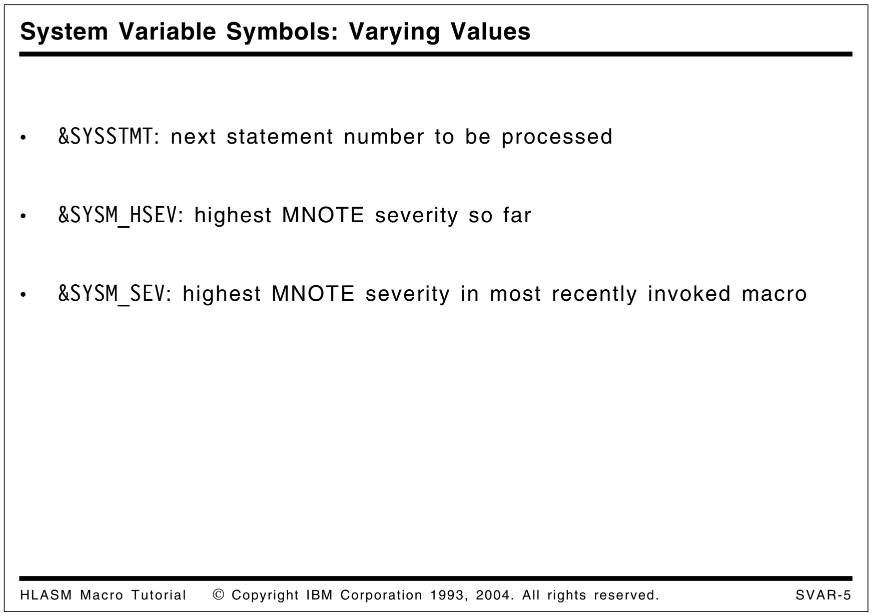
```
            Macro
&L          BDisp  &Target     Branch to non-addressable target
&L          BAS  1,Add&SYSNDX  Skip over constant
Off&SYSNDX  DC   Y(&Target-*)  Target offset
Add&SYSNDX  AH   1,Off&SYSNDX  Add offset
            BR   1             Branch to target
            MEnd
```

Note that although the *contents* of &SYSNDX is always decimal digits, it is actually a character-valued variable.

&SYSLIST

The &SYSLIST variable can be used to access positional parameters on a macro call (whether named or not). &SYSLIST supports a very rich set of sublist and attribute capabilities, and is therefore quite different from the other system variable symbols.

```
&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
```



System Variable Symbols: Varying Values

- &SYSSTMT: next statement number to be processed
- &SYSM_HSEV: highest MNOTE severity so far
- &SYSM_SEV: highest MNOTE severity in most recently invoked macro

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. SVAR-5

Variable Symbols Whose Values May Vary Anywhere

There are three system variable symbols whose value can vary in all contexts: &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

&SYSSTMT

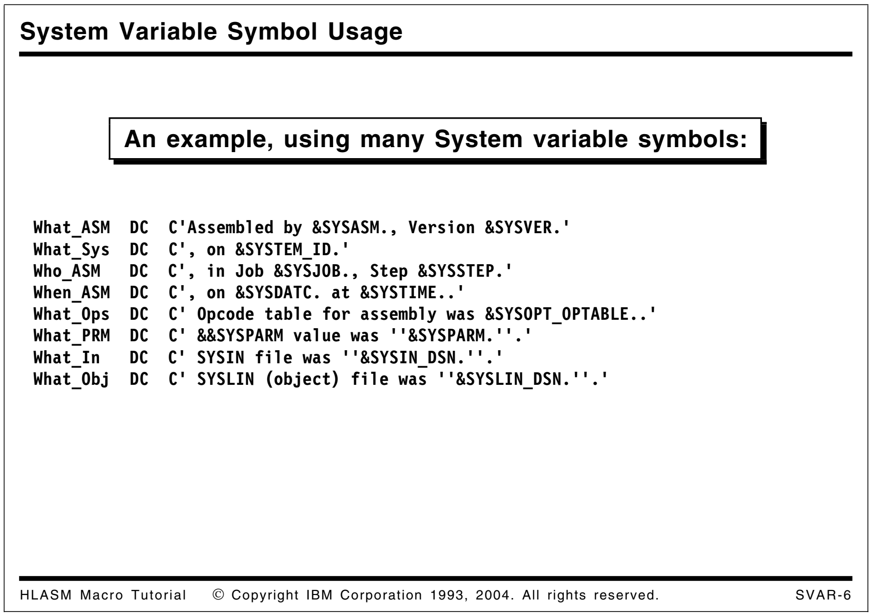
The &SYSSTMT symbol provides the number of the next statement to be processed by the assembler. Debugger data that depends on the statement number can be generated with this variable. For example, suppose we have a macro which inserts information about the current statement number into the object code of the program, and sets R0 to its address (so that a debugger can tell you which statement was identified in some debugging activity). A macro like the following might be used:

```
         Macro
 &L      DebugPtN
 &L      BAS   0,*+8             Addr of Statement Number in R0
         DC    AL4(&SYSSTMT)     Statement number information
         MEnd

  D      DebugPtN
 +D      BAS   0,*+8             Addr of Statement Number in R0
 +       DC    AL4(00000527)     Statement number information
```

&SYSM_HSEV and &SYSM_SEV

The &SYSM_HSEV and &SYSM_SEV symbols provide access to 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.



System Variable Symbol Usage

An example, using many System variable symbols:

```
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.''.'
```

HLASM Macro Tutorial © Copyright IBM Corporation 1993, 2004. All rights reserved. SVAR-6

Example Using Many System Variable Symbols

You might want to insert information into the object code of a program giving information about its assembly environment, in a form readable without “translation” from hex. The following example shows one way you might do this:

```
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.''.'
```

Relationships of Old and New System Variable Symbols

Some of the new system variable symbols introduced with High Level Assembler complement and supplement the data provided by system variable available in previous assemblers.

&SYSDATE and &SYSDATC

The variable symbol &SYSDATE is available in High Level Assembler and Assembler H, but was not supported by the earliest IBM assemblers. It provides a date in “American” (MM/DD/YY) format, without any century indication. As such, users in other countries sometimes had to extract and re-compose its fields to obtain a date conforming to local custom, convention, or standards. Further, the date could not be placed directly into fields as a sort key, because the year digits were in the lowest-order positions. Finally, no century was indicated.

High Level Assembler's introduction of the &SYSDATC variable solves all these problems very simply.

&SYSECT and &SYSSTYP

All previous assemblers have supported the &SYSECT variable to hold the name of the enclosing control section at the time a macro was invoked. This allows a macro which needs to change control sections (e.g., to declare a DSECT or to create code or data for a different CSECT) to resume the original control section on exit from the macro. There was, however, a sticky problem: there was no way for the macro to determine what *type* of control section to resume!

High Level Assembler provides the &SYSSTYP variable to rectify this omission: it provides the type of the control section named by &SYSECT. This permits a macro to restore the correct previous “control section environment” on exit.

&SYSNDX and &SYSNEST

All previous assemblers have supported the &SYSNDX variable symbol, which is incremented by one for every macro invocation in the program. This permits macros to generate unique ordinary symbols if they are needed as “local labels”. Occasionally, in recursively nested macro calls, the value of the &SYSNDX variable was used to determine either the depth of nesting, or to determine when control had returned to a particular level.

Alternatively, the programmer could define a global variable symbol of his own, and in each macro insert statements to increment that variable on entry and decrement it on exit. This technique is both clumsy (because it requires extra coding in every macro) and insecure (because not every macro called in a program is likely to be under the programmer's control, particularly IBM-supplied macros).

High Level Assembler provides the &SYSNEST variable to keep track of the level of macro-call nesting in the program. The value of &SYSNEST is incremented globally on each macro entry, and decremented on each exit.

&SYSTIME, &SYSCLOCK, and the AREAD Statement

The &SYSTIME variable symbol is provided by High Level Assembler and Assembler H, but not by earlier assemblers. It provides the local time of the start of the assembly in HH/MM format. This “time stamp” may not have sufficient accuracy or resolution for some applications.

There are two alternatives to the unvarying quality of &SYSTIME: the &SYSCLOCK variable, and the AREAD statement; &SYSCLOCK is described at “&SYSCLOCK” on page 233.

High Level Assembler provides an extension to the AREAD statement that may be useful if a more accurate time stamp is required. The current time can be obtained either in decimal or binary format.

The macro in the following example captures the clock reading in both decimal and binary formats:

```
         Macro
&Lab     AREADCLK
         LCLC   &D,&B
&D       Aread  CLOCKD
&B       Aread  CLOCKB
&Lab     DC     C'&D'        Decimal Clock
         DC     C'&B'        Binary  Clock
         MEnd

A        AREADCLK
+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.

Appendix C. Glossary of Abbreviations and Terms

absolute symbol. A symbol whose value does not change if *Location Counter* values change in the program; a non-relocatable symbol.

ADATA. See *SYSADATA file*.

address. (1) (*n*) A number used by the processor at *execution time* to locate and reference operands or instructions in central processor storage. In the context of this document, an address is what reference manuals (such as the *Principles of Operation*) would call a virtual address.
(2) (*v*) To reference; to provide an *address* (sense no. 1) that may be used to reference an item in storage.
(3) Sometimes used to mean an *assembly time location*.

address constant. A field in a program containing values calculated at *assembly time*, *bind time*, or *execution time*, typically containing an *address*, an offset, or a length. The operands of an address constant often are expressions involving *internal symbols*, *external symbols*, or both.

address resolution. The process whereby the assembler converts *implied addresses* into *addressing halfwords*, using information in its *USING Table*.

addressable. (1) At *execution time* an operand is addressable if it lies either in the 4096 bytes starting at address zero, or in any 4096-byte region of storage whose lowest address is contained in one of *general purpose registers* 1 through 15.
(2) At *assembly time* an *implied address* is addressable if it can be validly *resolved* by the Assembler into a *base-displacement addressing halfword*, using information contained in the *USING Table* at the time of the resolution.

addressing halfword. A two-byte field in the second and/or third halfwords of a *machine language* instruction, composed of a 4-bit *base digit* and a 12-bit *displacement*. An address expressed in *base-displacement* format.

anchor. (1) The *base location* or *base register* specified in the second operand of a USING statement.
(2) The starting point of a chained list.

Assembler. A program which converts source statements written in *Assembler Language* into *machine language*, providing additional useful information such as diagnostic messages, symbol usage cross-references, and the like.

Assembler Language. The symbolic language accepted by High Level Assembler, in which program statements are written. (Often, these statements describe individual instructions; this is why Assembler Language is frequently characterized as a “low level” language.) The *Assembler* translates these statements to an equivalent representation of the program in *machine language*. Assembler Language is intelligible to human beings trained in the art, but excessive art may render it unintelligible. Compare *machine language*.

In this document, we sometimes distinguish two components: (1) *conditional assembly language* and (2) *ordinary assembly language*. See also Figure 118 on page 244.

assembly language. See *Assembler Language*.

assembly time. The period in the lifetime of a program when its representation as a sequence of symbolic statements is being converted to the desired equivalent *machine language* form.

attribute. A property of a *symbol* known to the *assembler*, typically the characteristics of the item named by the symbol, such as its type, length, etc. A program may request the assembler to provide values of symbol attributes using *attribute references*.

A *variable symbol* may have one attribute specific to the symbol itself (the number attribute), and many attributes specific to the *value* of the *variable symbol*.

attribute reference. A notation used to request the value of a *symbol attribute* from the assembler's *symbol table*, or of a *variable symbol* or its value.

BAL (acronym). Basic Assembler Language. Intended to mean *Assembler Language*. The use of this term is deprecated, due to possible confusions with the BAL (Branch and Link) instruction and the BASIC programming language. The *Assembler Language* implemented by High Level Assembler is neither basic nor BASIC.

base. See *base register*, *base address*.

base address. The *address* in one of *general purpose registers* 1 to 15 to which a *displacement* is added to obtain an *effective address*.

base digit. See *base register specification digit*.

base-displacement addressing. A technique for addressing central storage using a compact *base-displacement* format for representing the derivation of storage addresses.

base location. (1) In *base-displacement address resolution*, the first operand of a USING statement, from which *displacements* are to be calculated. For ordinary USING statements, the base location is assumed to be at a relative offset (*displacement*) of zero from the address contained in the *base register*; for *dependent USING* statements, the base location may be at a positive nonzero offset from the location specified in the *base register* eventually used to resolve an *implied address*.
(2) Informally, this term is sometimes used to mean (a) the origin of a control section, (b) a *base address* in a register at *execution time*, and (c) whatever the speaker likes.

base register. The *General Purpose Register* specified in the second operand of a *labeled USING* or *ordinary USING*.

base register specification digit. The 4-bit field in bit positions 0-3 of an *addressing halfword*.

bind time. The time following *assembly time* during which one or more *object modules* are combined to form an executable module, ready for loading into central storage at *execution time*. Also known as "link time".

COM. A statement declaring the start or resumption of a *common section.*

common section. A special *dummy control section* whose name is an *external symbol*. Common sections receive special treatment during program linking: space is allocated for the greatest length received for all common sections with a given name.

complex relocatability. An *attribute* of a *symbol* indicating that its value is neither constant nor variable in exactly the same way as changes to the origin of its containing section. See *relocatability attribute*.

conditional assembly. A form of assembly whose input is a mixture of *conditional assembly language* and *ordinary assembly language* statements, and whose outputs are statements of the *ordinary assembly language*. Statements of the *ordinary assembly language* are treated only as "text", and are not obeyed during conditional assembly.

conditional assembly language. The "outer" language that controls the sequencing, selection, and tailoring of *ordinary assembly language* statements, through the use of *variable symbols*, *sequence symbols*, *conditional assembly* expressions, and substitutions. See also Figure 118 on page 244.

conditional assembly function. See *external function* and *internal function*.

control section. The smallest independently *relocatable* unit of instructions and/or data. All elements of a given control section maintain the same fixed relative positions to one another at *assembly time*. These fixed relative positions at *assembly time* are usually (but not necessarily) maintained by the program after control sections are placed into storage at *execution time*.

CSECT. See *control section*

dependent USING. A form of USING statement in which the first operand is based or *anchored* at a relocatable address. May also take the form of a labeled dependent USING statement. See also *anchor*, *labeled USING*, and *ordinary USING*.

displacement. The 12-bit field in bit positions 4-15 of an *addressing halfword*. Frequently used to describe the offset (difference) between a given storage address and a *base address* that might be used to *address* (sense no. 2) it.

DSECT. See *dummy control section* and *control section*.

dummy control section. A *control section* with the additional special property that no object code is generated for any of its statements. Most DSECT definitions are used as mappings or templates for data structures. The three types of dummy control sections are (1) ordinary dummy control sections, (2) *common sections*, and (3) *dummy external control sections*.

EAR. See *Effective Address Register*.

effective address. The storage address or similar value calculated at *execution time* from a *base address* and a *displacement*. See also *indexed effective address*.

Effective Address Register. An internal register used by the processor for calculating an *effective address*.

ESD. See *External Symbol Dictionary*.

execution time. The period in the lifetime of a program when its representation in *machine language* is interpreted by the processor as a sequence of instructions.
(2) The time at which programmers whose programs consistently fail to execute correctly are themselves executed.

explicit address. An instruction address in which the *displacement*, and either the *base* or *index* or both, are fully specified in the instruction, and for which no *resolution* into *base-displacement* format is required.

extended object module. A new *generalized object file format* supporting long external names, section sizes up to 1GB, multi-segment modules, and other enhancements. Produced by High Level Assembler when the XOBJECT or *GOFF option* is specified. See also *object module*.

external dummy section. A dummy control section (DSECT) whose name is made part of the *External Symbol Dictionary*. The Binder, Linkage Editor or Loader will resolve the lengths and alignment requirements of external dummy sections in such a way that storage may be allocated to the entire collection of external dummy sections (see the definition of the CXD Assembler Instruction Statement in the Assembler Language Reference), and the offset of each dummy section may be defined to the program using Q-type address constants (again, refer to the Assembler Language Reference).

external function. A function defined by the user and invoked by the assembler by the SETAF and/or SETCF statements during *conditional assembly*. External functions may access the assembler's operating system environment and return either arithmetic or character values, and optional messages to be placed into the listing.

external symbol. A symbol whose name and value are a part of the object module text provided by the Assembler. Such names include (1) *control section* names, (2) referenced names declared in V-type address constants or EXTRN statements, (3) names of *common sections*, (4) names of *Pseudo Registers* or *external dummy sections*, (5) referenced names declared on ENTRY statements, and (6) symbols and character strings renamed through the use of the ALIAS statement. Compare to *internal symbol*.

External Symbol Dictionary. The set of *external symbols* defined or referenced in an assembly, and provided in the *object module* for later use during program linking or binding.

function. See *external function* and *internal function*.

generalized object file format (GOFF). A new form of *object module* produced by High Level Assembler, providing numerous enhancements and extensions not supported by the traditional *object module* format.

GOFF. See *generalized object file format*.

GOFF option. An *option* that causes High Level Assembler to generate an *object module* using the *generalized object file Format*.

General Purpose Registers. A set of 16 32-bit registers used in the System/360/370/390 family of processors for addressing, arithmetic, logic, shifting, and other general purposes. Compare to special purpose registers such as *Access Registers*, *Control Registers*, and *Floating Point Registers*.

GPR. See *General Purpose Register*

HLASM. High Level Assembler/MVS & VM & VSE (Release 1); High Level Assembler for MVS & VM & VSE (Release 2 and later).

High Level Assembler. IBM's most modern and powerful symbolic assembler for the System/370 and System/390 series of computers, running on the MVS, VM, and VSE operating systems. Not necessarily an oxymoron, as High Level Assembler can do much more than ordinary (low-level) assemblers.

implied address. An instruction address requiring *resolution* by the Assembler into *base-displacement* format; an address for which base and displacement are not explicitly specified. Also *implicit address*.

index. (1) The contents of that *index register* specified by the *index register specification digit* in an RX-type instruction.
(2) Less frequently, the *index register specification digit* itself.

index digit. See *index register specification digit*.

index register specification digit. In an RX-type instruction, the 4-bit field contained in bit positions 12 through 15 of the instruction; the digit which, if not zero, specifies an *index register* to be used in calculating the *indexed effective address*

indexed effective address. The storage address or similar value calculated during program execution from a *base address*, a *displacement*, and an *index*. The term *effective address* is commonly used whether or not indexing is present.

index register. One of *general purpose registers* 1 through 15 specified by the *index register specification digit* in an RX-type instruction.

internal function. A function defined and executed by the assembler during *conditional assembly*, which acts on arithmetic, boolean, and character expressions to produce arithmetic, boolean, or character values. Compare *external function*.

internal symbol. A symbol naming an element of an *Assembler Language* program, which is assigned a single value by the *assembler*. Internal symbols are normally discarded at the end of the assembly, but may be retained in the *SYSADATA file*. Compare to *external symbol*.

internal symbol dictionary. See *symbol table*.

label. (1) The name field entry of an assembler or machine instruction statement. Normally, the presence of a label in the name field of an instruction statement will *define* the value of that label.
(2) In common parlance, the name of an instruction or data definition. This is more properly called a *name field symbol*.
(3) In High Level Assembler, the name field symbol of a USING statement, designating that statement as a *labeled USING*.

labeled USING. A form of USING statement with a *qualifier* symbol in the name field. Symbolic expressions resolved with respect to a labeled USING must use a *qualified symbol* with the *qualifier* of that labeled USING.

LC. See *location counter*.

Location Counter. A counter used by the Assembler to determine relative positions of all elements of a program as it is assembled.

location. A position within the object code of an assembled program, as determined by assigning values of the *Location Counter* during assembly. An *assembly time* value, sometimes confused with an *execution time address*.

machine language. The binary instructions and data interpreted and manipulated by the processor when the program is executed (at *execution time*). It is not meant to be intelligible to ordinary or normal human beings. Compare *Assembler Language*.

object module. A file produced by the Assembler, containing the *external symbols*, *machine language* instructions and data, and other data produced by assembling the source program. See also *extended object module*.

open code. Statements that are not within a macro definition or expansion. The statements in an assembly source file are typically in open code. See also *ordinary assembly language*.

options. Directives to the *Assembler* specifying various “global” controls over its behavior. For example, the PRINT option specifies that the assembler should produce a listing file. Options are specified by the user as a string of characters, as part of the command or statement that invokes the assembler, or on `*PROCESS` statements.

ordinary assembly language. The portion of the *Assembler Language* that includes machine instructions, data definitions, and assembler controls, but not including statements involved in *conditional assembly*. See *conditional assembly language*. See also Figure 118 on page 244.

ordinary symbol. See *internal symbol*.

ordinary USING. The oldest form of USING statement, in which (a) no entry is present in the name field, (b) the first operand specifies a *base address*, and (c) the second and successive operands are absolute expressions designating *General Purpose Registers* to be used as *base registers*.

PR. See *Pseudo Register* and *external dummy section*.

Pseudo Register. The name used by other processors such as the Linkage Editor and Loader for what the assembler calls an *external dummy section*. See *external dummy section*.

qualified symbol. An ordinary symbol preceded by a *qualifier*, and separated from the *qualifier* by a period.

qualifier. An ordinary symbol, defined as a qualifier by its appearance in the name field of a *labeled USING statement*. It is used only in *qualified symbols* to direct *base-displacement addressing* resolutions to a specified register or *anchor location*.

RA. See *relocatability attribute*.

reenterable. See *reentrant*.

reentrant. (1) Capable of simultaneous execution by two or more asynchronously executing processes or processors, with only a single instance of the code image. Typically, reentrant programs are expected not to modify themselves, but this is neither a necessary nor sufficient condition for reentrancy.
(2) When requested by the RENT option, or in an *RSECT*, simple tests are made by High Level Assembler for conditions of obvious self-modification of the program being assembled.

relocatability attribute. Each independently relocatable element of an *Assembler Language* program (such as a *control section* or *external symbol*) is assigned a distinct relocatability attribute. Each symbol in the *symbol table* is assigned the relocatability attribute of the element to which it belongs. An *absolute symbol* is assigned a zero relocatability attribute. See also *simple relocatability* and *complex relocatability*.

relocatable. (1) Capable of being placed into storage at an arbitrary (possibly properly aligned) address; not requiring placement at a fixed or pre-specified address in order to execute correctly.
(2) Having a non-zero *relocatability attribute*, which can mean either *simple relocatability* or *complex relocatability*.

relocation. The assignment of new or different locations or addresses to a set of symbols or addresses, by adding or subtracting constants depending on a module's assigned storage addresses.

relocation ID. Same as *relocatability attribute*. A numeric value assigned by the assembler to each independently relocatable element of a program such as *control sections* and *external symbols*.

resolution. See *address resolution*.

resolved. See *address resolution*.

RSECT. A *reentrant control section*, distinguished from an ordinary *control section* (CSECT) only by (a) the presence of a flag in the *External Symbol Dictionary* and (b) that High Level Assembler will perform *reentrant* checking of instructions within the RSECT.

run time. See *execution time*.

sequence symbol. A *conditional assembly symbol* used to mark positions in a statement stream, typically inside a macro definition.

simple relocatability. An *attribute* of a *symbol* indicating that changes to the value of the origin location of a *control section* will cause the value of the symbol to change by the same amount. See also *absolute symbol* and *complex relocatability*.

symbol table. A table created and maintained by the Assembler, to assign values and attributes to all symbols in the program, including ordinary and variable symbols. Except for symbols named in V-type address constants, the symbol table contains only a single occurrence of an ordinary symbol.

SYSADATA file. A file created by the High Level Assembler when the ADATA *option* is specified, containing machine-readable information about all aspects of the assembled program and the assembly process.

system variable symbol. A *variable symbol* defined by the *assembler*;\, containing information about the assembly process. Its value cannot be changed by the programmer.

USING Table. A table maintained at *assembly time* by the Assembler, used for *resolution* of *implied addresses* into *base-displacement* form. Each entry contains the number of a *base register* and a *base location*.

variable symbol. A symbol prefixed with a single ampersand (&). Used during *conditional assembly* to assist with substitution, expression evaluation, and statement selection and sequencing. Unlike *ordinary symbols*, the values of certain variable symbols may change freely during an assembly.

Appendix D. Ordinary and Conditional Assembly

Comparison	Ordinary Assembly	Conditional Assembly
Generality	the “inner” language of instructions and data definitions	the “outer” language that controls and tailors the inner language
Usage	a language for programming a machine	a language for programming an assembler and its 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, records (via REPRO, PUNCH)	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 (implicit or explicit declaration), (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 have many 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 Operators	L', I', S'	T', L', I', S', D', K', N', O'

Figure 118. Comparison of Ordinary and Conditional Assembly

Appendix E. Index