

**Assembler Language  
as a Higher Level Language:  
Macros and  
Conditional Assembly Techniques  
SHARE 102, Sessions 8167-8168**

John R. Ehrman  
Ehrman@VNet.IBM.Com

IBM Silicon Valley (Santa Teresa) Laboratory  
555 Bailey Avenue  
San Jose, California 95141

February, 2004

---

# **Conditional Assembly and Macro Overview**

# 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

## **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

---

**Part 1: The Conditional  
Assembly Language**

# 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**

# 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.

## Variable Symbols

---

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

<b>&amp;A</b>	<b>&amp;Time</b>	<b>&amp;DATE</b>	<b>&amp;My_Value</b>
---------------	------------------	------------------	----------------------
- 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)



## Declaring Variable Symbols

---

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

	Arithmetic Type	Boolean Type	Character Type
Local Scope	<b>LCLA</b>	<b>LCLB</b>	<b>LCLC</b>
Global Scope	<b>GBLA</b>	<b>GBLB</b>	<b>GBLC</b>
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)

# 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"

## 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

## Assigning Values to Variable Symbols: SET Statements

---

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

|                            |                   |                           |                                                                          |
|----------------------------|-------------------|---------------------------|--------------------------------------------------------------------------|
| <code>&amp;x_varsym</code> | <code>SETx</code> | <code>x_expression</code> | Assigns value of <code>x_expression</code> to <code>&amp;x_varsym</code> |
| <code>&amp;A_varsym</code> | <code>SETA</code> | arithmetic_expression     |                                                                          |
| <code>&amp;B_varsym</code> | <code>SETB</code> | boolean_expression        |                                                                          |
| <code>&amp;C_varsym</code> | <code>SETC</code> | 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:

|                                         |                      |
|-----------------------------------------|----------------------|
| <code>(operand OPERATOR operand)</code> | for binary operators |
| <code>(OPERATOR operand)</code>         | for unary operators  |

## Assigning Values to Variable Symbols: SET Statements ...

---

- Target variable symbol may be subscripted

|                  |             |          |                        |
|------------------|-------------|----------|------------------------|
| <b>&amp;A(6)</b> | <b>SETA</b> | <b>9</b> | <b>Set &amp;A(6)=9</b> |
| <b>&amp;A(7)</b> | <b>SETA</b> | <b>2</b> | <b>Set &amp;A(7)=2</b> |

- 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)

## 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)`

## Arithmetic Expressions: Internal Arithmetic Functions

---

- Shifting functions
  - Written (operand `Shift_0p` shift\_amount)
  - `Shift_0p` 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_0p` 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



## **SETA Statements vs. EQU Statements**

---

- Note differences between SETA and EQU statements:

| <b>SETA Statements</b>                                           | <b>EQU Statements</b>                                                                            |
|------------------------------------------------------------------|--------------------------------------------------------------------------------------------------|
| 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 <i>variable symbol many times</i>   | A value is assigned to a given <i>ordinary symbol only once</i>                                  |
| 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                                  |

## 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)
```

## 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

## 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 STSTI  
&J SETA 2  
&Y SETC (2*&J)'*'    &Y has value ****
```

## 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&&B&&B
```
  - Use substrings 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
```

## 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
```

## 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

## 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 <u>12345</u>
&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 <u>'&amp;&amp;'</u>
&N	SETA	K'&C	&N has value 4
&C	SETC	(3)'AB'	&C has value <u>ABABAB</u>
&N	SETA	K'&C	&N has value 6

- Arithmetic and boolean variables converted to strings first

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



## Character Expressions: Internal Character Functions

---

- Character-valued (unary) character operations:

<code>&amp;X_Up</code>	SetC	(UPPER ' &X')	All letters in &X set to upper case
<code>&amp;Y_Low</code>	SetC	(LOWER ' &Y')	All letters in &Y set to lower case
<code>&amp;Z_Pair</code>	SetC	(DOUBLE ' &Z')	Amperands/apostrophes in &Z doubled
<code>&amp;CharVal</code>	SetC	(SIGNED &A)	Convert arithmetic &A to signed string
<code>&amp;EBDIC</code>	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

## 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

## 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:

<b>Variable Type</b>	<b>SETA Statement</b>	<b>SETB Statement</b>	<b>SETC Statement</b>
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

## 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”

## 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

## 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
```

# The AGO Statement

---

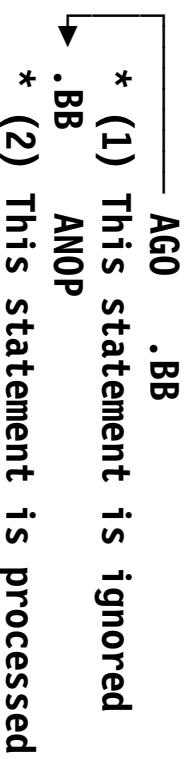
- **AGO unconditionally** alters normal sequential statement processing
  - Assembler breaks normal sequential statement processing
  - Resumes at statement marked with the specified sequence symbol
  - Two forms: Ordinary AGO and Extended AGO
- Ordinary AGO (Go-To statement)

`AGO sequence_symbol`

**Example:**

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

- Example of use:



## 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!
```



## 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  
* (2) This statement is processed
```

## 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, (&B00L2).SS2, ('&C' EQ '*').SS3
```

## 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!
```

## 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!)'
```

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

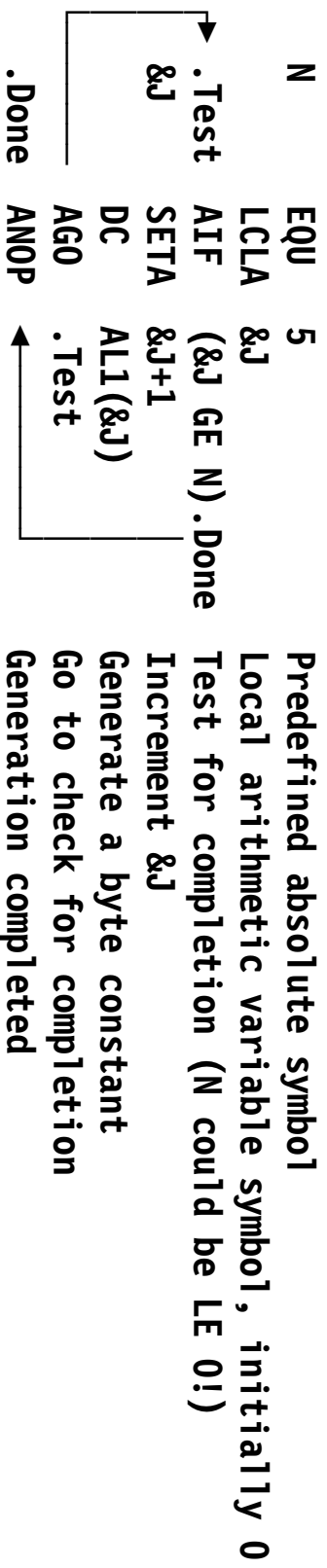
---

- 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: D0 for J = 1 to N (GEN( DC AL1(J)))



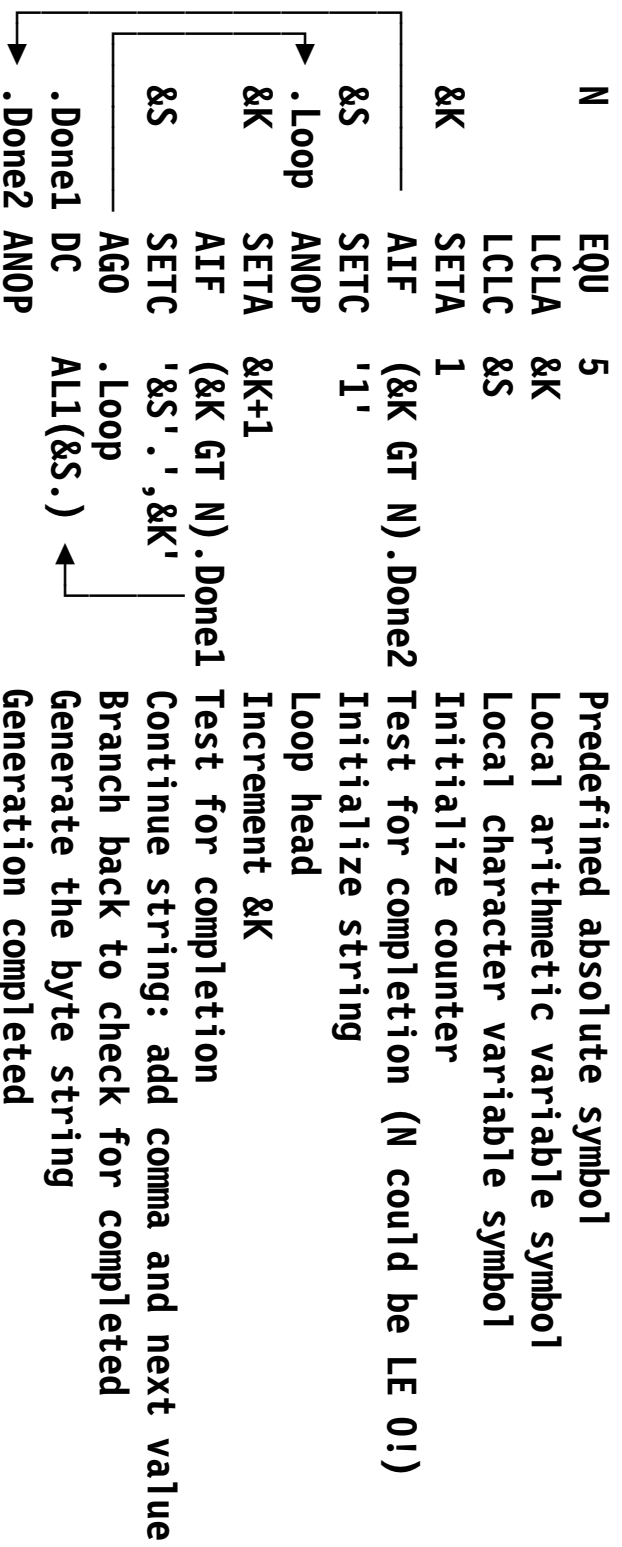
## Example: Generate a Byte String with Values 1-N ...

---

- 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))



- Try it with 'N EQU 30', 'N EQU 90', 'N EQU 300'

## 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
-----
Input   AIF  ('&OpSys' NE 'MVS').T1  Skip if not MVS
        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

## 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!



---

## **Part 2: Basic Macro Concepts**

## 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

## **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

## The Macro Concept: Fundamental Mechanisms

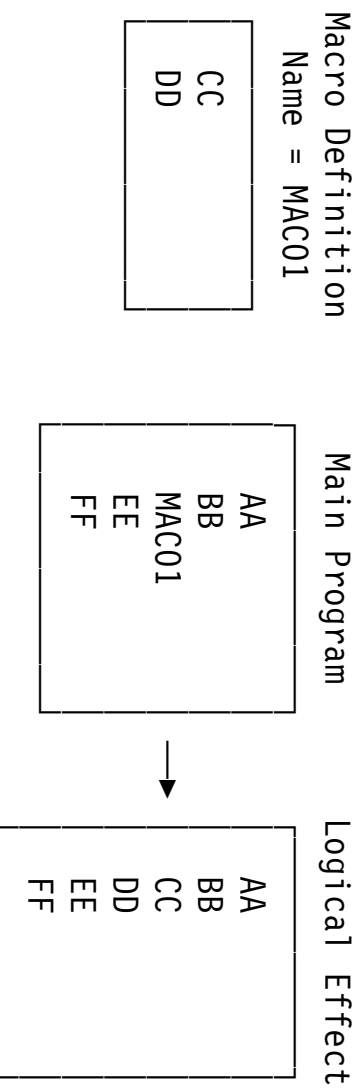
---

- 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

## Basic Macro Concepts: Text Insertion

---

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

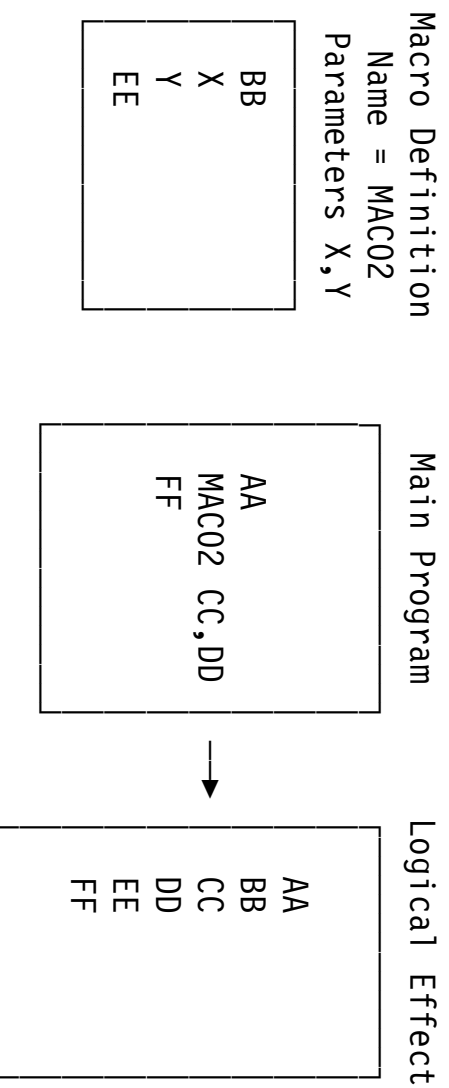


- The processor recognizes MAC01 as a macro name
- The text of the macro definition replaces the “macro call” in the Main Program
- When the macro ends, processing resumes at the next statement

## Basic Macro Concepts: Text Parameterization

---

- Text parameterization: tailoring of the inserted text

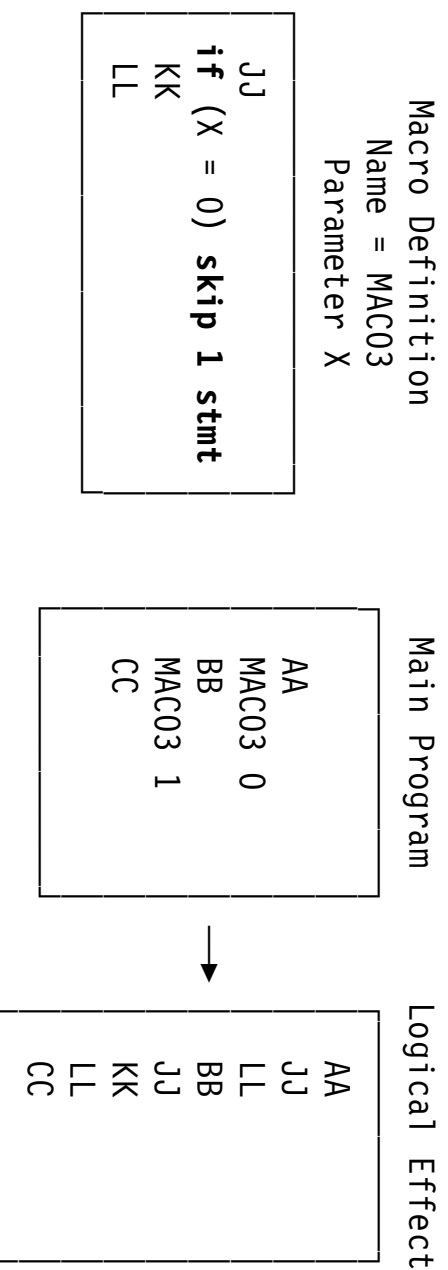


- Processor recognizes MAC02 as a macro name, with arguments CC, DD
  - Arguments CC, DD are **associated** with parameters X, Y **by position**
  - As in all high-level languages
- The text from the macro definition is modified during insertion

## Basic Macro Concepts: Text Selection

---

- Text selection: choosing alternative text streams for insertion



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

## Basic Macro Concepts: Call Nesting

---

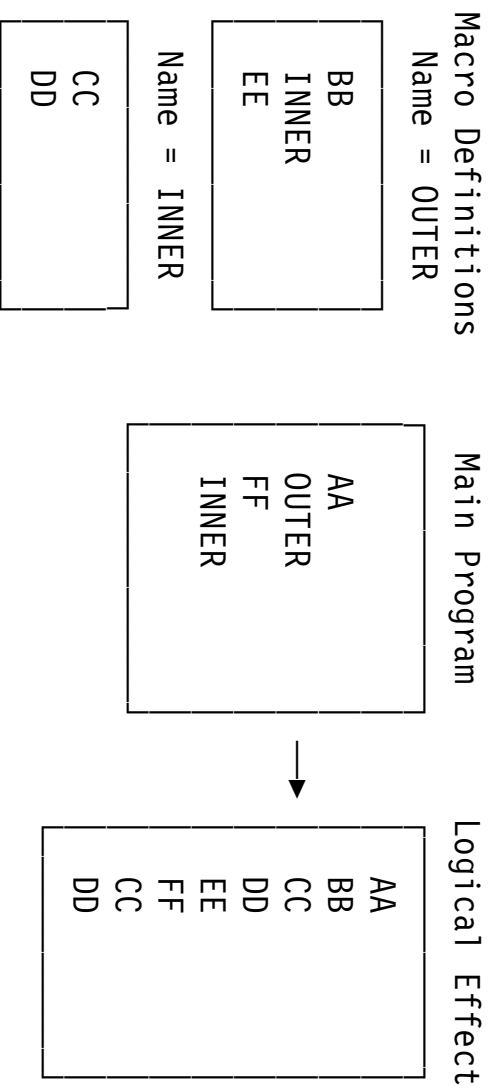
- Generated text may include calls on other (“inner”) macros
  - New statements can be defined in terms of previously-defined extensions
- Generation of statements by the outer (enclosing) macro is interrupted to generate statements from the inner
- Multiple levels of call nesting OK (including recursion)
- 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



## Macro Call Nesting: Example

---

- Two macro definitions: OUTER contains a call on INNER

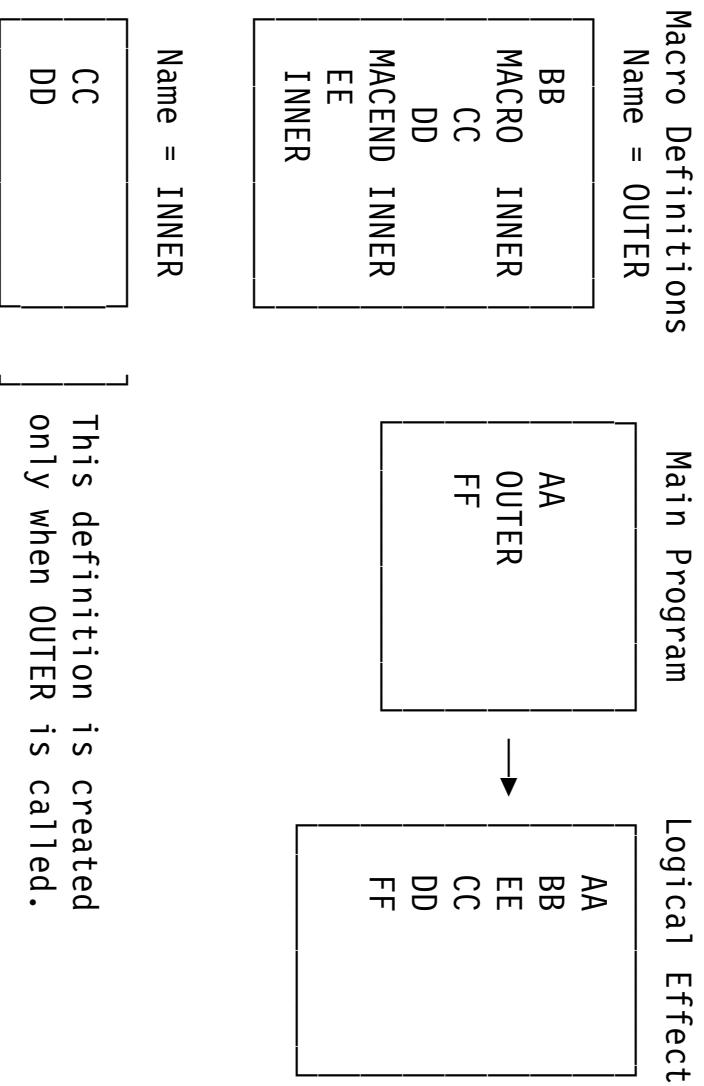


- Expansion of OUTER is suspended until expansion of INNER completes

## Macro Definition Nesting: Example

---

- Macro definitions may contain macro definitions



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

# The Assembler Language Macro Definition

---

- A macro definition has four parts:

(1)	MACRO
(2)	Prototype Statement
(3)	Model Statements
(4)	MEND

Macro Header (begins a definition).

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.

Declarations, conditional assembly statements, and text for selection, modification, and insertion. Zero to many statements.

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

## 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”

## 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

## 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

## Nested Macro Definitions in High Level Assembler

---

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

```
&L      Macro ,           Start of MAJOR's definition
        MAJOR &X
        LCLA &A          Local variable
        ---
        Macro ,           Start of MINOR's definition
        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)



## 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 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

## Example 1: Define General Register Equates

---

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

```
MACRO0                                (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:

```
MACRO0
GREGS
LCLA  &N
ANOP  EQU  &N
SETA  &N+1
AIF   (&N LE 15) .X
MEND
```

Define a counter variable, initially 0  
2 points of substitution in EQU statement

Increment &N by 1  
Repeat for all registers 1–15

## 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

## 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
```

## 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...

## 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,&KeyParam=YES Prototype
Lab1 MYMAC3 X,Y,KeyParam=NO Ca11: 2 positional, 1 keyword argument

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

## Macro Parameter-Argument Association ...

---

- Example 2:

```
Lab2 MYMAC3 A          Ca11: 1 positional argument

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

- Example 3:

```
MYMAC3 H,KeyParm=MAYBE,J Ca11: 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



## 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  Ca11 with "keyword"?
```

- This looks like a keyword and a positional argument:

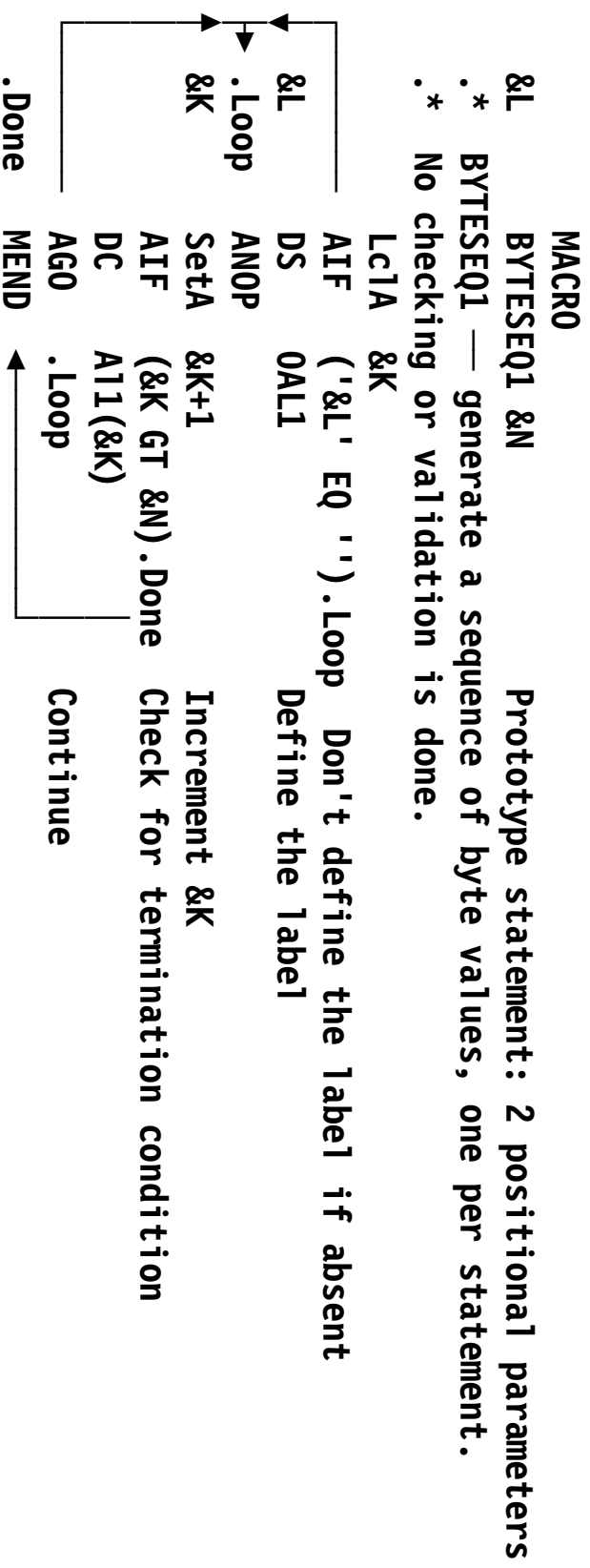
```
TestMac  K=What,Maybe  Ca11 with "keyword"?
```

- 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

## 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



- Examples

```

BS1a   BYTESEQ1 5
       BYTESEQ1 1

```

## 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, REPR0, SETX, SETXF
  - The assembler must understand basic macro structures at the time it encodes the macro!
- Implementation trade-off: generation speed vs. generality

## 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 ( T' ), Length ( L' ), Scale ( S' ), Integer ( I' ), Defined ( D' ), and “OpCode” ( O' ) 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 sublist elements
  - Use substring and concatenation operations to parse list items

## 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...

## 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

## 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'
```

## 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



## 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 sublistis 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)
```

## 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          (A is not a 1st)
N'&SYSLIST(2)      = 1          is a 1st with 1 item
N'&SYSLIST(3)      = 2          is a 1st with 2 items
N'&SYSLIST(3,2)    = 3          is a 1st with 3 items
N'&SYSLIST(3,2,1)  = 1          (D is not a 1st)
N'&SYSLIST(4)      = 2          is a 1st with 2 items
```

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

## 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

## 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

## 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

## 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
  - **Arithmetic, Boolean, and Character** 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)

## **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

# 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



## 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)

```
Gb1B &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''
```

## 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)

## 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
```

## 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

## 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)**

---

## **Part 3: Macro Techniques**

## **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 without converting

## **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



## **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
  - General Purpose, Floating Point, Control, Access

## Define General Register Equates (Simply)

---

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

```
MACRO0
GREGS
GR0    Equ    0
GR1    Equ    1
. *
GR15   Equ    15   etc.
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

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

- If &GRegs is true, no statements are generated

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

## Defining Register Equates Safely: Pseudo-Code

---

- Allow declaration of multiple register types on one call:  
Example: REGS type<sub>1</sub>[,type<sub>2</sub>]... 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.Reggs\_Done)
  - If &(&T.Reggs\_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
&J      SetA      (N'&SsYsList eq 0).Exit
         ANOP      Initialize argument counter
         SetC      (Upper '&SsYsList(&J)')      Pick up an argument
         SetA      ('ACFG' Index '&T')          Check type
&N      AIF       (&N eq 0).Bad                Error if not a supported type
         GBLB     (&T.Reggs_Done)              Declare global variable symbol
         AIF      (&(&T.Reggs_Done)).Done Test if true already
&N      SetA      0
         ANOP     ,                               Generate Equ statements
         &T.R&N   Equ      &N
&N      SetA      &N+1
         AIF      (&N 1e 15).Gen
         ANOP     (&(&T.Reggs_Done) SetB (1)      Indicate definitions have been done
         .Next   &J      Count to next argument
&J      SetA      &J+1
         AIF      (&J 1e N'&SsYsList).GetArg     Get next argument
         MEXIT   MEXIT
         MNOTE   MNOTE      8, '&SsYsMac.: Unknown type ''&T.''.'
         MEXIT   MEXIT
         AGO     MNOTE      0, '&SsYsMac.: Previously called for type &T..'
         MEND   .Exit      .Next
```

## **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

## 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.
      Lc1A &K
      AIF ('&L' EQ '').Loop Don't define the label if absent
      DS  0A11 Define the label
      .Loop ANOP
      &K    SetA &K+1 Increment &K
          AIF (&K GT &N).Done Check for termination condition
          DC  AL1(&K)
          AGO .Loop Continue
      .Done MEND
```

\* Two test cases

```
BS1a  BYTESEQ1 5
      BYTESEQ1 1
```

## Generating a Byte Sequence: 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



## 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 ALL(&S)
MEND
```

## 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      X'D500 0000',S(&Source)  
&lab  CLC   0(0,0),&Source      Back up to first byte of instruction  
Org    *-6                               X'4100',S(&Target),S(&Source)  
LA     0,&Target.(0)                   Back up to first byte of instruction  
Org    *-4                               ALL(X'D2',L'&Source-1) First 2 bytes of instruction  
DC     ALL(X'D2',L'&Source-1)          Step to next instruction  
Org    *+4  
MEnd
```

- The CLC instruction “forces” a literal source operand into the assembler's symbol table, so it's available to the L' reference

## 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'

## 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
Gb1A &Dec          Decimal value returned in &Dec
&X   SetC  'X'&Hex'' Create hex self-defining term
&Dec SetA  &X      Do the conversion
      MNote 0, '&Hex (hex) = &Dec (decimal)' For debugging
MEnd
*
Dec  AA
*** MNOTE *** 0,AA (hex) = 170 (decimal)
Dec  FFF
*** MNOTE *** 0,FFF (hex) = 4095 (decimal)
Dec  FFFFFF
*** MNOTE *** 0,FFFFFF (hex) = 16777215 (decimal)
Dec  7FFFFFFF
*** MNOTE *** 0,7FFFFFFF (hex) = 2147483647 (decimal)
```

## Macro-Time Conversion from Decimal to Hex

---

- 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

## 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
Gb1C    &Hex      Hex value returned in &Hex
SetC    ' '       Initialize &Hex
SetA    &Dec      Local working variable
.Loop   ,         Top of reduction loop
&R      (&Q AND 15) &R = Mod ( &Q, 16 )
&Q      (&Q SRL 4)   Quotient for next iteration
&Hex    '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) = FFFFFFFF (hex)
```

- Exercise: extend Hex macro to accept negative arguments

## **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

## Generate a List of Named Integer Constants

---

- Syntax:    INTCONS n<sub>1</sub>[,n<sub>2</sub>]...[,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)

```
MACRO0
&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
&Name   SetC   (&J GT N'&SsYsList).End Exit if all done
&Name   SetC   '&Type.&SsYsList(&J)'. Assume non-negative arg
&Name   AIF    ('&SsYsList(&J) '(1,1) ne '-').NotNeg Check arg sign
&Name   SetC   '&Type.M'.'&SsYsList(&J) '(2,*)'. Negative argument, drop -
.NotNeg ANOP
&Name   DC     &Type.'&SsYsList(&J)'.
&Name   AGO    .ArgsoK          Repeat for further arguments
.End    MEND
```

- Exercise: generalize to support + signs on operands

## **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

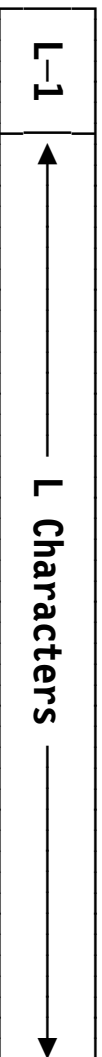
### **2. Case Study 6b: Block comments**

- Write free-form text comments (without \* in column 1)

## Case Study 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
```

## Create Length-Prefixed Messages (1)

---

- PFMSG1: length-prefixed message texts

```
MACRO
&Lab  PFMSG1 &Ttxt
. *   PFMSG1 — requires that the text of the message, &Ttxt,
. *   contain no embedded apostrophes (quotes) or ampersands.
      Lc1A  &len      Effective Length
      SetA  K'&Ttxt-3 (# text chars)-3 (quotes, eff. length)
&len  DC      ALL(&len), C&Ttxt
&Lab  MEND
```

- Limited to messages with no quotes or ampersands

```
M1a   PFMSG1 'This is a test of message text 1.'
+M1a  DC      ALL(32), C'This is a test of message text 1.'

M1b   PFMSG1 'Hello'
+M1b  DC      ALL(4), C'Hello'
```

## Create General Length-Prefixed Messages (2)

---

- PFMSG2: Allow all characters in text (may require pairing)

```
MACR0
&Lab  PFMSG2 &Ttxt
.*    PFMSG2 — the text of the message, &Ttxt, may contain embedded
.*    apostrophes (quotes) or ampersands, so long as they are paired.
&T    SetC  'TXT&SYSNDX.M'  Create TXInnm symbol to name the text
&Lab  DC   AL1(L'&T.-1)    Effective length
&T    DC   C&Ttxt
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

## **Readable Length-Prefixed Messages (3): Pseudo-Code**

---

- User writes “plain text” messages (single line,  $\leq$  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 ALL(Text\_Length-1),C'MessageText')

## Create Readable Length-Prefixed Messages

---

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

```
MACRO0
&lab    PFMSG3  &Nu11      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.

      Lc1A    &L,&N          Local arithmetic variables
      Lc1C    &T,&C,&M       Local character variables
      AIF     ('&Nu11' eq '') .OK   Null argument OK
      AIF     (N'&SYSLIST EQ 0).OK  No arguments allowed
      MNote   8,'PFMSG3 — no operands should be provided.'
      MEXIT
      .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     ALL(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!



## 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.  
TNEMOC**

- Restriction: block-end statement (TNEMOC) can't appear in the text

## Block Comments Macro

---

- COMMENT macro initiates block comments:

```
Macro
&L      Comment &Arg
        Lc1C      &C
        AIf      ('&L' eq '' and '&Arg' eq '') .Read
        MNote    *, 'Comment macro: Label and/or argument ignored. '
        .Read
        &C
        &C      ARead      ,
        &A      SetC      (Upper '&C')
                   Force upper case
                   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!

## Case Study 7: Macro Recursion

---

- Macro recursion illustrated with:
  1. “Indirect addressing”
  2. Integer factorial values:  $N! = N * (N-1)$
  3. Integer Fibonacci numbers:  $F(N) = F(N-1) + F(N-2)$

## 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  &Reg,&X      Load &Reg with indirection
Aif  ('&X'(1,1) eq '*').Ind Branch if indirect
L    &Reg,&X
MExit
ANop
.Ind  ' &X'(2,*)   Strip off leading asterisk
&XI  &Reg,&XI     Call myself recursively
L    &Reg,0(, &Reg)
MEnd
```

## Indirect Addressing via Recursion ...

---

- Examples of code generated by calls to LI macro:

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

Load from 0(4)
```

Load from what 0(,4) points to

```
+      LI      3,*0(,4)
+      L       3,0(,4)
+      L       3,0(,3)
```

Two levels of indirection

```
+      LI      3,**0(,7)
+      L       3,0(,7)
+      L       3,0(,3)
+      L       3,0(,3)
```

Three levels of indirection

```
+      LI      3,***X
+      L       3,X
+      L       3,0(,3)
+      L       3,0(,3)
+      L       3,0(,3)
```

# 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    ANOP  .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
```

## 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 &SSysNest.'
      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
&Temp SetA     &K        Evaluate F(K) = F(N-1) (Recursive call)
      FIBONACI &L       Hold computed value
&Ret  SetA     &L        Evaluate F(L) = F(N-2) (Recursive call)
      FIBONACI &L       Evaluate F(N) = F(K) + F(L)
&Ret  SetA     &Ret+&Temp
      AIF      (&SSysNest 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
```



## **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

## **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

## 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!

## 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 named bits and unnamed bytes

| TM       | Bit_Name,L'Bit_Name | Refer to byte and bit using bit name      |
|----------|---------------------|-------------------------------------------|
| DS       | X                   | Unnamed byte                              |
| BitA Equ | *-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)      |

## 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`

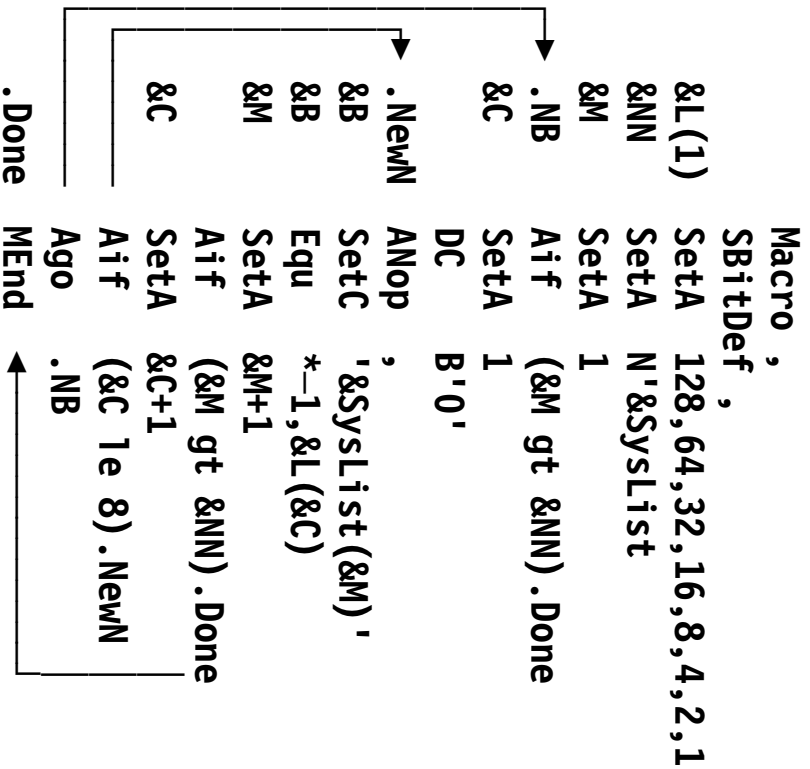
`IF (Mod(M,8)=1) GEN ( '_DC B'0' )      (Generate unnamed byte)`

`GEN (Arg(M) EQU *-1,Lengths(Mod(M-1,8)+1) )      (Define bit name)`

## Simple Bit-Defining Macro: SBITDEF

---

```
Macro ,
SBITDef ,
    &L(1)   SetA 128,64,32,16,8,4,2,1
    &NN     SetA N'&SsYsList
    &M      SetA 1
    .NB     Aif (&M gt &NN).Done
    &C      SetA 1
    DC      B'0'
    .NewN   ANop ,
            ' &SsYsList (&M) '
            &B   SetC *-1,&L(&C)
            &M   SetA &M+1
            &C   Aif (&M gt &NN).Done
            &C   SetA &C+1
            .Done Aif (&C 1e 8).NewN
    .Done   MEnd
```



```
SBITDef b1,b2
+       DC      B'0'
+b1     Equ     *-1,128
+b2     Equ     *-1,64
```

Error checking omitted  
No declared parameters  
Define bit position values  
Number of bit names provided  
Name counter  
Check if names exhausted  
Start new byte at leftmost bit  
Allocate a bit-flag byte  
Get a new bit name  
Get M-th name from argument list  
Define bit via length attribute  
Step to next name  
Exit if names exhausted  
Count bits in a byte  
Get new name if byte not full  
Byte is filled, start a new byte

Define bits b1, b2  
Allocate a bit-flag byte  
Define bit via length attribute  
Define bit via length attribute

## Simple Bit-Manipulation Macros: Pseudo-Code

---

- Operations on “named” bits
- Setting bits on: one OI instruction per named bit

IF (Label  $\neq$  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  $\neq$  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!

## Simple Bit-Handling Macros: Setting Bits ON

---

- Macro SBitOn to set one or more bits ON
- Syntax: SBitOn bitname[,bitname]...

|         |                                          |
|---------|------------------------------------------|
| Macro , | Error Checking omitted                   |
| SBitOn  | Number of Names                          |
| &lab    | Skip if no name field                    |
| &NN     | Define label                             |
| &M      | Get a bit name                           |
|         | Extract name (&M-th positional argument) |
| &lab    | Set bit on                               |
| .Next   | Step to next bit name                    |
| &B      | Go get another name                      |
| .Go     |                                          |
| &M      |                                          |

```
Macro ,
SBitOn
&lab
&NN
&M
    .Next
    &B
    .Go
    &M
        ANop
        SetC
        OI
        SetA
        Aif
        MEnd
        , '&SysList(&M)'
        &B,L'&B
        &M+1
        (&M 1e &NN).Next
```



## Simple Bit-Handling Macros: Setting Bits ON ...

---

- Examples:

```
AA1  SBit0n  b1,b3,b8,c1,c2
+AA1 DC  0H'0'

+  0I  b1,L'b1      Define Label
+  0I  b3,L'b3      Set bit on
+  0I  b8,L'b8      Set bit on
+  0I  c1,L'c1      Set bit on
+  0I  c2,L'c2      Set bit on

SBit0n  b1,b8
+  0I  b1,L'b1      Set bit on
+  0I  b8,L'b8      Set bit on
```

- Observe: one OI instruction per bit!
  - We will consider optimizations later

## 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
```

## Simple Bit-Handling Macros: Set OFF and Invert Bits ...

---

- Examples:

|      |         |                |              |
|------|---------|----------------|--------------|
| bb1  | SBitOff | b1,b3,b8,c1,c2 | Define label |
| +bb1 | DC      | 0H'0'          | Set bit off  |
| +    | 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 | Define label |
| +cc1 | DC      | 0H'0'          | Invert bit   |
| +    | 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

## Simple Bit-Handling Macros: Branch on Bit Values

---

- Simple bit-testing macros: branch to target if bitname is on/off
- Syntax: `SBBitxxx bitname, target`

|                        |                                    |                                 |
|------------------------|------------------------------------|---------------------------------|
| <code>&amp;lab</code>  | <code>SBBit0n &amp;B,&amp;T</code> | <b>Bitname and branch label</b> |
| <code>&amp;l1ab</code> | <code>TM &amp;B,L'&amp;B</code>    | <b>Test specified bit</b>       |
|                        | <code>B0 &amp;T</code>             | <b>Branch if 0N</b>             |
|                        | <code>MEnd</code>                  |                                 |

|                        |                                     |                                 |
|------------------------|-------------------------------------|---------------------------------|
| <code>&amp;lab</code>  | <code>SBBit0ff &amp;B,&amp;T</code> | <b>Bitname and branch label</b> |
| <code>&amp;l1ab</code> | <code>TM &amp;B,L'&amp;B</code>     | <b>Test specified bit</b>       |
|                        | <code>BNO &amp;T</code>             | <b>Branch if 0FF</b>            |
|                        | <code>MEnd</code>                   |                                 |

|                   |                             |                           |
|-------------------|-----------------------------|---------------------------|
| <b>*</b>          | <b>Examples</b>             |                           |
| <code>ddl</code>  | <code>SBBit0n b1,aa1</code> | <b>Test specified bit</b> |
| <code>+ddl</code> | <code>TM b1,L'b1</code>     | <b>Branch if 0N</b>       |
| <code>+</code>    | <code>B0 aa1</code>         |                           |
|                   | <code>SBBit0n b2,bb1</code> | <b>Test specified bit</b> |
| <code>+</code>    | <code>TM b2,L'b2</code>     | <b>Branch if 0N</b>       |
| <code>+</code>    | <code>B0 bb1</code>         |                           |

## 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!”)

|             |               |               |                                          |
|-------------|---------------|---------------|------------------------------------------|
| <b>X</b>    | <b>DC</b>     | <b>F'23'</b>  | <b>Define a constant</b>                 |
| <b>Flag</b> | <b>Equ</b>    | <b>X'08'</b>  | <b>Define a flag bit (?) 'somewhere'</b> |
|             | <b>SBitOn</b> | <b>Flag,X</b> | <b>Set two bits ON 'somewhere' ???</b>   |

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

## **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

## 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!

## 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

## General Bit-Handling Macros: Bit Definition

---

```
Macro ,
BitDef
    Gb1A &BitDef_ByteNo      Used to count defined bytes
    SetA 128,64,32,16,8,4,2,1 Define bit position values
    SetA N'&Sylst            Number of bit names provided
    &M 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 '&Sylst(&M)'    Get M-th name from argument list
    Aif ('&B'(1,1) ne '(').NoL Branch if not a sublist
    &NS SetA N'&Sylst(&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 ...

---

|                                           |                   |                                              |                                          |
|-------------------------------------------|-------------------|----------------------------------------------|------------------------------------------|
| <code>.*</code>                           | <code>--</code>   | (continuation)                               | Name is in a sublist                     |
| <code>.SubT</code>                        | <code>ANOp</code> | <code>,</code>                               | Generate sublist equates                 |
| <code>&amp;B</code>                       | <code>SetC</code> | <code>'&amp;SsYsList(&amp;M,&amp;CS)'</code> | Extract sublist element                  |
| <code>&amp;B</code>                       | <code>Gb1A</code> | <code>&amp;(BitDef &amp;B._ByteNo)</code>    | Created var sym with ByteNo for this bit |
| <code>&amp;B</code>                       | <code>Equ</code>  | <code>*-1,&amp;L(&amp;C)</code>              | Define bit via length attribute          |
| <code>&amp;(BitDef &amp;B._ByteNo)</code> | <code>SetA</code> | <code>&amp;BitDef_ByteNo</code>              | Byte no. for this bit                    |
| <code>&amp;CS</code>                      | <code>SetA</code> | <code>&amp;CS+1</code>                       | Step to next sublist item                |
|                                           | <code>Aif</code>  | <code>(&amp;CS gt &amp;NS).NewA</code>       | Skip if end of sublist                   |
| <code>&amp;C</code>                       | <code>SetA</code> | <code>&amp;C+1</code>                        | Count bits in a byte                     |
|                                           | <code>Ago</code>  | <code>.SubT</code>                           | And go do more list elements             |
| <code>.NoL</code>                         | <code>ANOp</code> | <code>,</code>                               | Not a sublist                            |
|                                           | <code>Gb1A</code> | <code>&amp;(BitDef &amp;B._ByteNo)</code>    | Declare byte number for this bit         |
| <code>&amp;B</code>                       | <code>Equ</code>  | <code>*-1,&amp;L(&amp;C)</code>              | Define bit via length attribute          |
| <code>&amp;(BitDef &amp;B._ByteNo)</code> | <code>SetA</code> | <code>&amp;BitDef_ByteNo</code>              | Byte no. for this bit                    |
| <code>.NewA</code>                        | <code>ANOp</code> | <code>,</code>                               | Ready for next argument                  |
| <code>&amp;M</code>                       | <code>SetA</code> | <code>&amp;M+1</code>                        | Step to next name                        |
|                                           | <code>Aif</code>  | <code>(&amp;M gt &amp;NN).Done</code>        | Exit if names exhausted                  |
| <code>&amp;C</code>                       | <code>SetA</code> | <code>&amp;C+1</code>                        | Count bits in a byte                     |
|                                           | <code>Aif</code>  | <code>(&amp;C 1e 8).NewN</code>              | Get new name if not done                 |
|                                           | <code>Ago</code>  | <code>.NB</code>                             | Bit filled, start a new byte             |
| <code>.Done</code>                        | <code>MEnd</code> |                                              |                                          |

## 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'
+d1     Equ     *-1,128      Define a bit-flag byte
+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

## 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

## General Bit-Setting Macro: Design

---

- Optimize generated code using variables declared by BitDef macro
- Syntax:     Bit0n   bitname[,bitname]...
- Example:     Bit0n   a,b,c,d
- High-level pseudo-code:
  - D0 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
    - D0 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

## 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)

## General Bit-Setting Macros: Set Bits ON

---

- Macro Bit0n optimizes generated instructions (most error checks omitted)

```

Macro
  &Lab      Bit0n      ' &Lab '          Save Label
  &L        SetC      ' &Lab '          No. of distinct Byte Nos.
  &NBN     SetA      0                  Name counter
  &M       SetA      0                  Number of names provided
  &NN      SetA      N'&SsYsList       Check if all names scanned
  .NmLp    Aif       (&M ge &NN).Pass2 Step to next name
  &M       SetA      &M+1              Pick off a name
  &B       SetC      ' &SsYsList(&M)'  Check for null item
  .NmLp    Aif       ('&B' eq '').Nu11 Declare GBLA for Byte No.
  &B       Gb1A     &(BitDef_&B._ByteNo) eq 0).Undef Exit if undefined
  &K       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

```



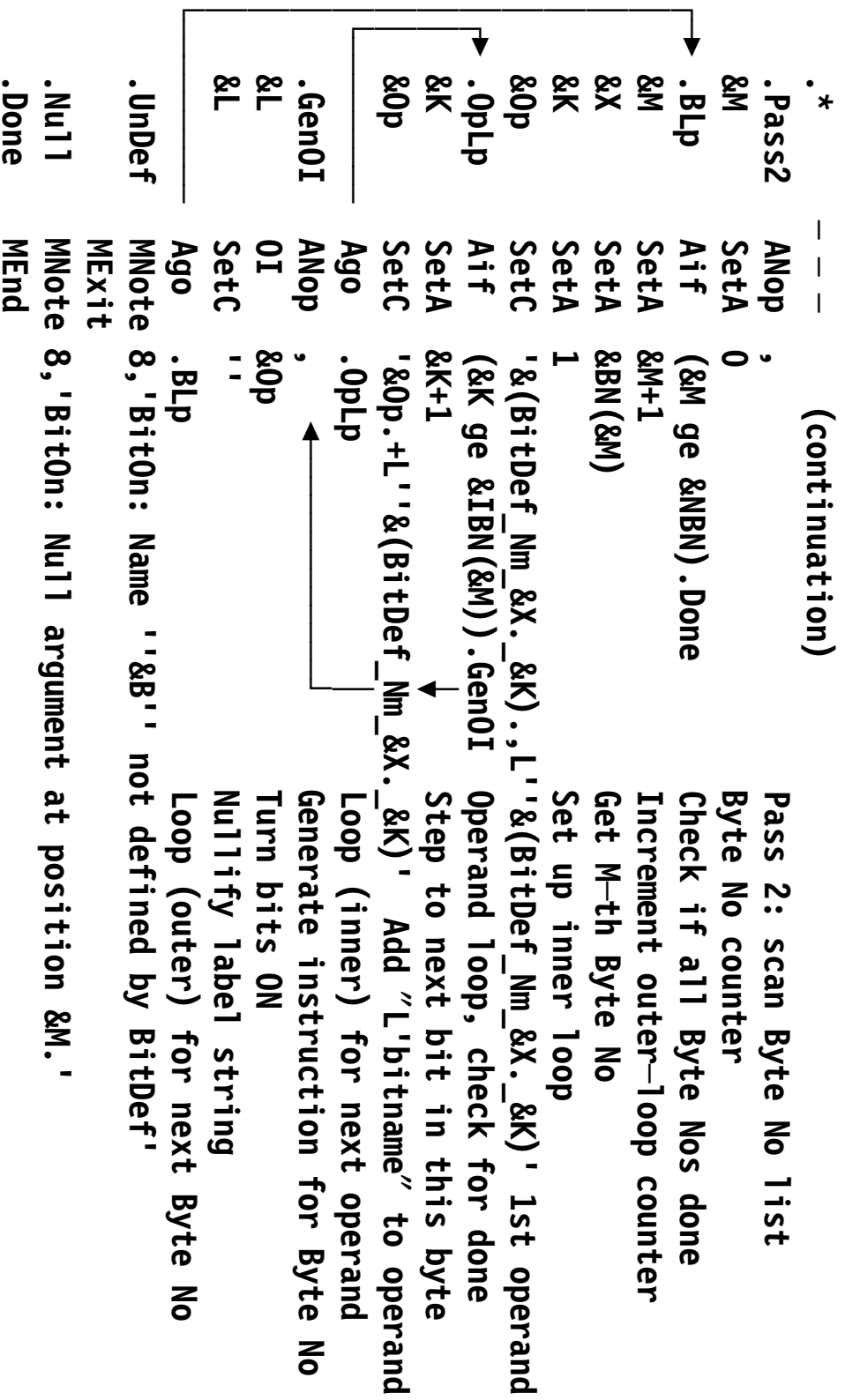
## General Bit-Setting Macros: Set Bits ON ...

---

```
.*      --- (continuation)
&J     SetA 1          Check if name already specified
.CkDup Aif (&J gt &IBN(&K)).NmOK  Branch if name is unique
&J     Aif ('&B' eq '&(BitDef_Nm_&BN(&K)._&J)') DupNm Duplicated
      SetA &J+1       Search next name in this byte
      Ago .CkDup      Check further for duplicates
.DupNm MNote 8, 'Bit0n: 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
      Lc1C &(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
      Lc1C &(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 ...

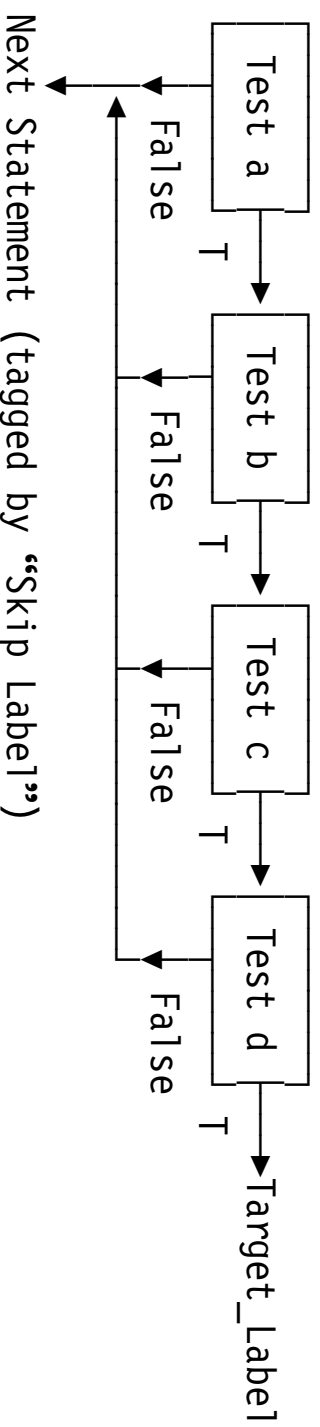
---



## General “Branch if Bits On” Macro: Design

---

- Function: branch to target if all named bits are on
- Syntax:    BBitOn (bit1ist), 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!

## 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 ; B0 Target_Label)
    IF (NBN > 1) GEN (Skip_Label DS OH)
```

## General Bit-Handling Macros: Branch if Bits On

---

- Bbit0n 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 Bit0n)
  2. Generate optimized tests and branches;  
if multiple bytes, generate “skip” tests/branches and label

```
Macro
&lab      BBit0n &NL,&T      Bit Name List, Branch Target
&L        Aif      (N'&Syslist ne 2 or '&NL' eq '' or '&T' eq '').BadArg
&NBN      Setc    '&lab'    Save Label
&M        Seta    0         No. of distinct Byte Nos.
&NN       Seta    0         Name counter
.NmLp     Seta    N'&NL     Number of names provided
.*        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
      GbIA  &(BitDef_&B._ByteNo)  Declare GBIA 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
&J     Aif  (&BN(&K) ne &(BitDef_&B._ByteNo)).BNLp  Check match
      .CKDup SetA  1        Check if name already specified
      Aif  (&J gt &IBN(&K)).NmOK  Branch if name is unique
&J     Aif  ('&B' eq '&(BitDef_Nm_&BN(&K))._&J')'.DupNm  Duplicated
      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
      LcIC  &(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
                Lc1C  &(BitDef_Nm_&BN(&NBN))._1) Slot for first bit name
&(BitDef_Nm_&BN(&NBN))._1) SetC '8B'  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      '8Op.+L'(&(BitDef_Nm_&X_&K)' Add next bit to operand
. *      Ago      .OpLp     Loop (inner) for next operand
                -- --      (continued)
```

## 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
        TM       &Op      Test if bits are 0N
        BNO      &Skip     Skip if not all 0N
        &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 0N
        B0      &T      &T      Branch if all 0N
        &Skip   Aif     (&NBN eq 1).Done  No skip target if just 1 byte
        DC      0H'0'   Skip target
        MExit
        .Undef  MNote 8, 'BBit0n: Name ''&B'' not defined by BitDef'
        .BadArg MExit
        .Done  MNote 8, 'BBit0n: Improperly specified argument 1ist'
        MEnd
```



## **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

## Defining and Using Data Types

---

- We're familiar with type sensitivity in higher-level languages:
  - Instructions generated from a statement depend on data types:  
$$A = B + C ; \quad '=' \text{ and } '+' \text{ are polymorphic operators}$$
  - 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

## 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 ,
&Lab INCR &V,&A,&Reg=0      Increment &V by amount &A (default 1)
&T   SetC T'&V              Default work register = 0
&Op  SetC ' &T'             Type attribute of 1st arg
&I   SetC '1'               Save type of &V for mnemonic suffix
    Aif ('&A' eq '').IncoK   Default increment
    SetC '&A'                Increment now set OK
.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
```

## 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

+ **** MNOTE ****      8,INCR: Cannot use type 'X' of 'WXY' .
      Incr  WXY,2
      Test with unsupported type
```

## 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?
- Assembler data types know nothing about “meaning” of variables, only their hardware representation; so, typing is very weak!

|       |      |             |  |                                        |
|-------|------|-------------|--|----------------------------------------|
| Rate  | DS   | F           |  | Rate of something                      |
| MyPay | DS   | PL6         |  | My salary                              |
|       | ADD2 | MyPay, Rate |  | Add (binary) Rate to (packed) MyPay ?? |

|      |    |   |  |                   |
|------|----|---|--|-------------------|
| 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 (??) |

## 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

## 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"
- Macro "A" checks the second operand for type F, A, or Q

```
My_A    OpSyn  A      Save definition of A as My_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.'
      ANop
&L     My_A   &R,&X
      MEnd
```

- Note that allowed types are "hard coded" in the macro



## **Base-Language Type Sensitivity: General Type Checking**

---

- Intent: compatibility checking between instruction and operand types
- Define TypeCheck macro to request type checking  
Syntax: TypeCheck opcode, valid\_types
- Call TypeCheck with: opcode to check, allowable types

```
TypeCheck L, 'ADDFQVX'    Allowed types: A QV (adcons), D, F, X
```

- Sketch of macro to initiate type checking for one mnemonic:

```
Macro
TypeCheck &Op, &Valid    Mnemonic, set of valid types
Gb1C &(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...

## Base-Language Type Sensitivity: General Type Checking ...

---

- Now, need to install L macro in the macro library:

```
Macro
&Lab  L  &Reg,&0operand
      Gb1C  &(TypeCheck_L_Valid) List of valid types for L
&TypOp SetC  T'&0operand      Type attribute of &0operand
&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 ''&0operand.''?'
      .OK   ANop
&Lab  TypeCheck_L  &Reg,&0operand
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

## **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 TypeCheck, don't use the instruction-replacement macros!
- Better:
  - Specify a list of instructions to be checked, such as  
**TypeCheck (L,ST,A,AL,S,SL,N,X,0), 'ADFFQVX'**
  - The TypeCheck macro generates the replacement macros as needed

## 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

## Base-Language Type Checking: Generated Macros

---

- Generate each type-checking macro using AINSERT

```
                TypeChek  (L,ST,A,AL,S,SL,N,X,0), 'ADFGVX'   Desired style
```
- Sketch of revised inner loop of TypeChek macro:

```
&0p  Setc  '&0ps(&K)'           Pick off K-th opcode
&0p  OpSyn ,                    Disable previous definition of &0p
.*   Generate macro to redefine &0p for type checking
AInsert ' Macro ',BACK
AInsert '&&lab &0p. &&Reg,&&0pd',BACK
AInsert ' GblC &&(TypeChek_&0p._Valid)',BACK
AInsert '&&T0 Setc T''&&0pd',BACK
AInsert '&&T SetA (''&&(TypeChek_&0p._Valid)'' Find ''&&T0''),BACK
AInsert ' AIf (&&T ne 0).OK ',BACK
AInsert ' MNote 1, ''Possible type conflict between &0p and &&0pd?''',B*
      ACK
AInsert '.OK ANop ',BACK
AInsert '&&lab TypeChek_&0p &&Reg,&&0pd ',BACK
AInsert ' MEnd ',BACK
.*   End of macro generation
```
- Compare to “hand-coded” L macro (slide Tech-77)

## 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

## 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,&Opoperand
      GblC  &(TypeCheck_L_Valid),&(TypeCheck_L_RegType)
&TypOp SetC  T'&Opoperand      Type attribute of &Opoperand
&Test  SetA  ('&(TypeCheck_L_Valid)' Find '&TypOp')  Check validity
      AIf    (&Test ne 0).OK_0p  Skip if valid
      MNote 1,'Possible type incompatibility between L and ''&Opoperand.''?'
      .OK_0p ANop
      .*      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
      &lab  TypeCheck_L  &Reg,&Opoperand
      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'?
```

## 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**



## 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
Gb1C  &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'&SystList                   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
&SystList(&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
```

## User-Assigned Type Attributes: DCLNTVL Macro

---

- Declaration of INTERVAL types made by Dc1Ntv1 macro
  - Initial value can be specified with Init= keyword

```
Macro ,                               Args = list of names
DCLNTVL &Init=0,&len=3                Optional initialization value
Gb1C &Ntv1Typ                          Type attr of Interval variable
Lc1A &Ntv1Len                           Length of an Interval variable
&Ntv1Typ SetC 'i'                       User type attr is lower case 'i'
.*                                       Length of an INTERVAL type could also be a global variable
&NV SetA N'&SystList                   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
&SystList(&K) Equ *-&len.,&len.,C'&Ntv1Typ' Declare name, length, type
Ago .Test
.Done MEnd
```

```
Dc1Ntv1 Week,Init=7
+ DC PL3'7'                               Define storage
+Week Equ *-3,3,C'i'                     Name, length, 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:

|                         |                      |                         |                       |                                     |
|-------------------------|----------------------|-------------------------|-----------------------|-------------------------------------|
| <code>ResultDate</code> | <code>CalcDat</code> | <code>Date,+</code>     | <code>Interval</code> | <code>Date = Date + Interval</code> |
| <code>ResultDate</code> | <code>CalcDat</code> | <code>Date,-</code>     | <code>Interval</code> | <code>Date = Date - Interval</code> |
| <code>ResultDate</code> | <code>CalcDat</code> | <code>Interval,+</code> | <code>Date</code>     | <code>Date = Interval + Date</code> |

- CalcDat validates (abstract) types of all arguments, and calls one of two auxiliary macros

|                       |                                                      |                                   |
|-----------------------|------------------------------------------------------|-----------------------------------|
| <code>DATEADDI</code> | <code>Date1,LDat,Interval,LNv1,AnsDate,AnsLen</code> | <code>Date = Date+Interval</code> |
| <code>DATESUBI</code> | <code>Date1,LDat,Interval,LNv1,AnsDate,AnsLen</code> | <code>Date = Date-Interval</code> |

- Auxiliary service macros (“private methods”) understand actual data representations (“encapsulation”)
- In this case: packed decimal, with known operand lengths

## 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
      Gb1C &Ntv1Typ,&DateTyp        Type attributes
      &T1  Setc T'&Arg1              Save type of &Arg1
      &T2  Setc T'&Arg2              And of &Arg2
      Aif ('&T1&T2' ne '&DateTyp&Ntv1Typ' and
          '&T1&T2' ne '&Ntv1Typ&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 '&Ntv1Typ').Add2 1st opnd is interval of days
          DATEADDI &Arg1,L'&Arg1,&Arg2,L'&Arg2,&Ans,L'&Arg1  D = D+I
          MExit
          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
```

## Calculating Interval Variables: CalcNv1 Macro

---

- Define user-called CalcNv1 macro to calculate intervals
- Allowed forms are:

|                |         |                       |                                  |
|----------------|---------|-----------------------|----------------------------------|
| ResultInterval | CalcNv1 | Date, -, Date         | Difference of two date variables |
| ResultInterval | CalcNv1 | Interval, +, Interval | Sum of two interval variables    |
| ResultInterval | CalcNv1 | Interval, -, Interval | Difference of two intervals      |
| ResultInterval | CalcNv1 | Interval, *, Number   | Product of interval, number      |
| ResultInterval | CalcNv1 | Interval, /, Number   | Quotient of interval, number     |

- CalcNv1 validates declared types of arguments, and calls one of five auxiliary macros (more “private methods”):

|          |                                            |                 |
|----------|--------------------------------------------|-----------------|
| NTVLADDI | Nv11, Len1, Nv12, Len2, AnSI, Anslen       | Nv1 = Nv1 + Nv1 |
| NTVLSUBI | Nv11, Len1, Nv12, Len2, AnSI, Anslen       | Nv1 = Nv1 - Nv1 |
| NTVLMULI | Nv11, Len1, Nv12, Len2, AnSI, Anslen       | Nv1 = Nv1 * Num |
| NTVLDIVI | Nv11, Len1, Nv12, Len2, AnSI, Anslen       | Nv1 = Nv1 / Num |
| DATESUBD | Date1, LDate1, Date2, LDate2, AnSI, Anslen | Nv1 = Date-Date |

# Calculating Interval Variables: CalcNvl Macro ...

---

```

&Ans      Macro
CALCNVL  &Arg1,&Op,&Arg2
Gb1C  &NtV1Typ,&DateTyp      Type attributes
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
&X(C'/') Setc  'C'&Op''      Convert &Op char to self-def term
&Z      Setc  T'&Arg1         Type of Arg1
&T1     Setc  T'&Arg2         Type of Arg2
&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      I op I
Days      CALCNVL  Hire,-,Degree      Date - Date
+         DATESUBD  Hire,L'Hire,Degree,L'Degree,Days,L'Days  date-date

```

## 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
    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
    AP &Ans.(&Lans),&Arg1.(&L1)  Add Arg1 to Answer
    MExit
.T2 ANop ,                        Move Arg1 to Answer
    ZAP &Ans.(&Lans),&Arg1.(&L1)
    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
```

## Comparison Operators for Dates and Intervals

---

- Define comparison macros CompDate and CompNtv1

```
&label  CompDate  &Date1,&Op,&Date2,&True  Compare two dates
&label  CompNtv1  &Ntv11,&Op,&Ntv12,&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
          Gb1A    &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
```



## 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
READ   OpSyn ,
      ---
&L   READ   &A,&B,&C
      ---
READ OpSyn READ_XX
MEnd

Save Wrapper's definition as READ_XX
Nullify this definition
...perform 'front-end' processing
Call system version of READ
...perform 'back-end' processing
Re-establish Wrapper's definition
```

## 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!

---

## **External Functions**

## External Conditional Assembly Functions

---

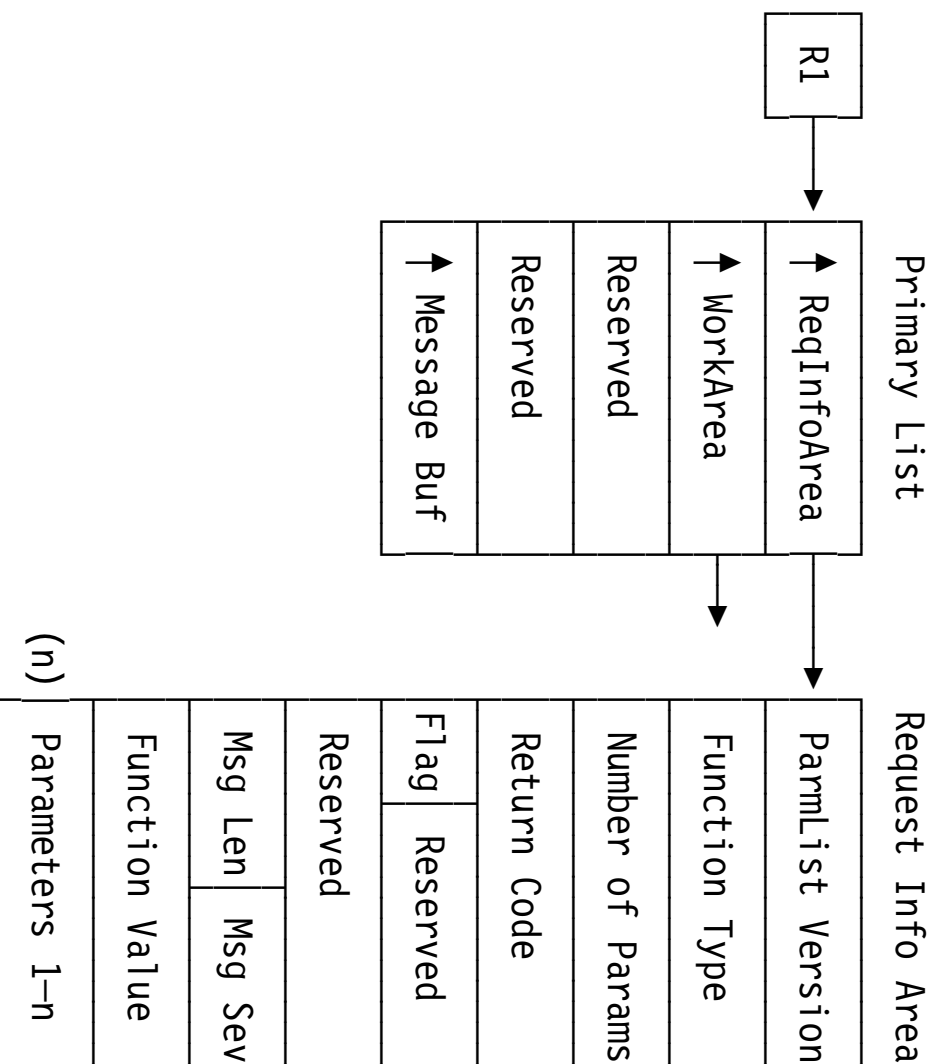
- Two types of external, user-written functions
  1. Arithmetic functions: like &A = AFunc (&V1, &V2, ...)

|       |       |                        |                      |
|-------|-------|------------------------|----------------------|
| &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

# SETAF External Function Interface

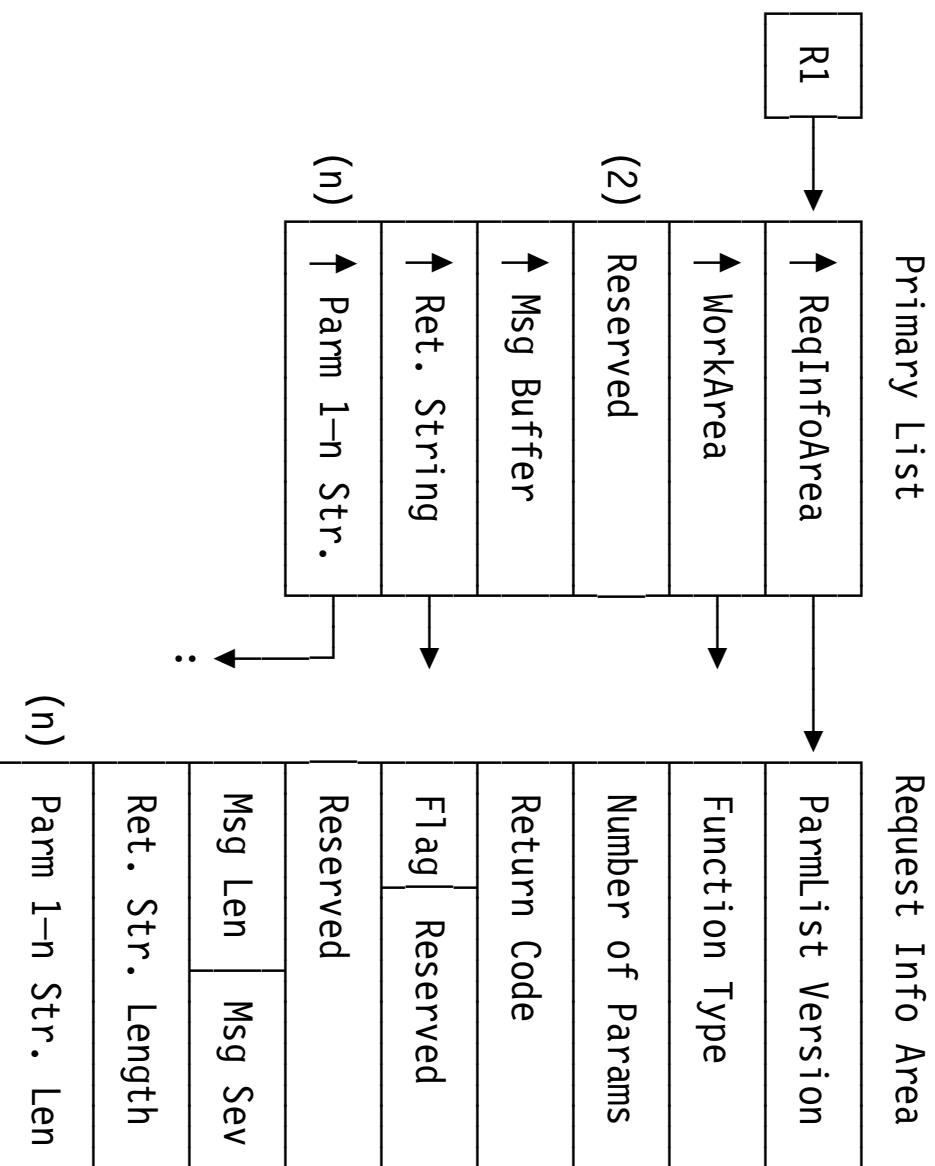
---



- (n) means the field is repeated **n** times
- HLASM provides a 32-byte work area

# SETCF External Function Interface

---



- $(n)$  means the field is repeated  $n$  times
- HLASM provides a 32-byte work area

---

**System (&SYS) Variable  
Symbols**

## **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, &SYSTEME
  - 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)