High Level Assembler:

Benefiting From Its Powerful New Features

SHARE 96 (Winter 2001), Session 8165

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

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

February, 2001

Table of Contents

Table of Contents

Table of Contents

Topic Overview

- Options and Language enhancements
- Mixed-Case Input and Output
- Old and New USING Statements
- GOFF and Binder Considerations
- Conditional Assembly Functions
- System Variable Symbols
- Assembler I/O Exits
- Macro-Operand Sublists

HLASM Options: Overview

- HLASM accepts option specifications from several sources:
  - `*PROCESS` statements in the program being assembled
  - an external `ASMAOPT` file
  - invocation parameters
  - installation defaults
- Options apply to various assembly activities:
  - Assembly: `BATCH, PROFILE, SIZE`
  - Source file: `DBCS, OPTABLE, COMPAT, SYSPARM`
  - Object file: `GOFF, TEST, TRANSLATE, CODEPAGE`
  - Assembler I/O: `EXIT, ADATA, DECK, OBJECT, TERM`
  - Listing: `ASA, ESD, FOLD, LINECOUNT, RLD, PCONTROL, INFO, LIBMAC, LIST, USING(MAP), THREAD`
    - Messages: `ALIGN, FLAG, LANGUAGE, RENT, RA2, USING(WARN), USING(LIMIT)`
    - Cross-References: symbols, general registers, macro/COPY members, DSECTs

New Ordinary-Assembly Statements

- HLASM provides many new assembler instruction statements:

`*PROCESS`	Source-file assembly options
`ACONTROL`	Dynamic control of certain options
`ADATA`	User data kept with the SYSADATA file
`ALIAS`	Modifies external symbols in object file
`CEJECT`	Conditional control of listing pagination
`CATTR`	Assign class names and attributes
`EXITCTL`	Provide control data to I/O exits
`XATTR`	Assign attributes to external symbols

Enhanced Ordinary-Assembly Statements

- Existing statements are enhanced by HLASM:

`AMODE/RMODE`	Extended to support 64-bit addressing
`COPY`	Supports variable-symbol operand in open code
`DC`	Many new constant types:

`EB,DB,LB`	IEEE Floating Point
`EH,DH,LH`	Hex Floating Point
`AD,FD`	8-byte address, binary
`CU`	Sixteen-bit Unicode
`J,R`	Length, PSECT Address

Blanks allowed in quoted nominal values (except C, G)
No nominal value needed if duplication factor is zero

`PRINT`	Accepts MCALL, MSOURCE, UHEAD operands
`PUSH/POP`	Accepts ACONTROL operand
`RSECT`	Declares a read-only section
`USING/DROP`	Extended for labeled and dependent USINGs

Conditional Assembly Enhancements

- New conditional-assembly statements have been added and enhanced:

`AEJECT/ASPACE`	Control formatting of macro definition listing
`AINSERT`	Place constructed records into “pre-input” buffer
`AREAD`	Supported operands: CLOCKB, CLOCKD, NOPRINT, NOSTMT
`SETAF, SETCF`	Invoke externally-defined conditional assembly function

- Other enhancements include:
  - Many new system (&SYS) variable symbols
  - Simpler variable symbol declaration
  - Enhanced substring notation
  - Predefined absolute symbols in conditional assembly expressions
  - Easier scanning of macro-argument sublists

Macro-Operand Lists and Sublists

- High Level Assembler designed to be upward compatible with previous assemblers
  - Example: 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` allows both cases to be handled the same way

Other Useful Language Enhancements

- Unary minus supported in arithmetic expressions
- DXD operand alignment rationalized
- NOPRINT operand supported on several statements
- Attribute-reference extensions
  - `O'` ("Operation Code")
  - `I', S'` in open code
- Literals as macro operands treated more sensibly
- Literals in machine instructions treated more as "ordinary symbols"
- Attribute references to literals return reliable values

Mixed-Case Input

- All IBM mainframe assemblers accept mixed case in:
  - remarks fields of assembler and machine instruction statements

```
NAME    OPCODE  OPERAND,OPERAND     Remarks may be in mixed case
        PRINT   DATA                PRINT all generated text
```

  - comment statements

```
*       Comment statements may also be in mixed case
```

  - quoted character strings in character constants and self-defining terms

```
MIXCON  DC    C'AbBbCcDdeE'   Character Constant
SELFDEF LA    R1,C'a'         Character self-defining term
```

  - macro instruction statement operand values.

```
MACCALL MACOP  Positional,KEY=KeyValue   Macro call operands
```

Mixed-Case Symbols and Operation Codes

- High Level Assembler permits lowercase characters in
  - symbolic operation codes
  - ordinary symbols
  - variable symbols
    - local and global
    - system (&SYS)
    - macro-instruction positional and keyword parameter names
  - sequence symbols
- Operation codes and symbols treated as identical to their uppercase equivalents.

```
label    a    reg9,storage_operand(indexreg)    )) These are
Label    A    Reg9,Storage_Operand(IndexReg)    )) equivalent
LABEL    A    REG9,STORAGE_OPERAND(INDEXREG)    )) statements
```

- Symbol Table displays each symbol as it was first encountered.

Mixed-Case Macro Arguments

- Mixed-case symbols do **not** change macro argument handling:
  - Characters in macro arguments are always left in their original case
  - Macro calls using mixed-case characters in arguments will work in High Level Assembler just as in previous assemblers.

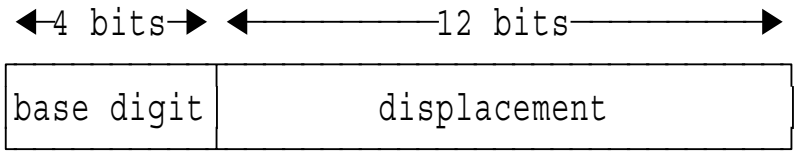
```
LABEL   MACCALL  Positional_Value,KEYWORD=Key_Value     All assemblers

Label   MacCall  Positional_Value,KeyWord=Key_Value     HLASM only
```

- Keyword and Positional **values** are unchanged
  - Passing mixed-case values may require internal macro changes if such values must be recognized.
    - UPPER function can help!
  - Use `COMPAT(MACROCASE)` option if existing macros expect uppercase operands

Ordinary USING Statements

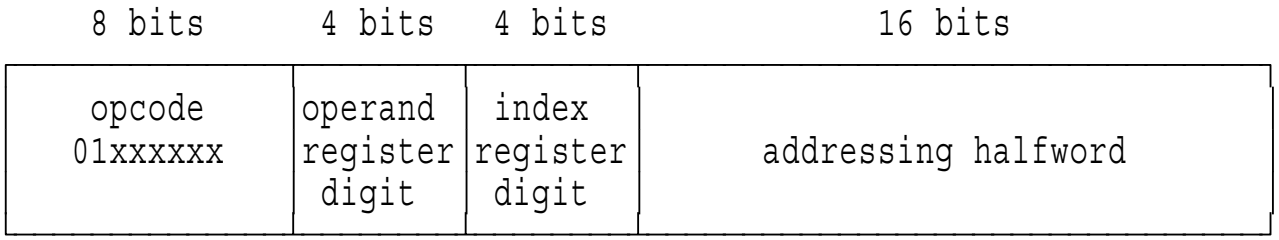
Addressing Halfwords and Effective Addresses

- Many instructions generate addresses from *addressing halfwords*:



Effective Address = displacement + if (b ≠ 0) then C(Rb) else 0

- For RX-type instructions, an *index* may be used:



Effective Address = displacement + if (b ≠ 0) then C(Rb) else 0
+ if (x ≠ 0) then C(Rx) else 0

Manually-Specified Base and Displacement

- Consider assigning bases and displacements symbolically
  - Displacements derived “manually” for each symbol reference

```
Location      Name     Operation  Operand

  0000                 BASR       6,0
  0002        BEGIN    L          2,N-BEGIN(0,6)
  0006                 A          2,ONE-BEGIN(0,6)
  000A                 ST         2,N-BEGIN(0,6)
              -------- 22 bytes of stuff --------
  0024        N        DC         F'8'
  0028        ONE      DC         F'1'
```

  - Each storage address specifies two items: an origin and a register
- Prefer to specify those just once
- Hence, the `USING` statement!

Assembler-Calculated Base and Displacement

- USING combines base-register and base-location information
  - Relation to actual addressing instructions is unknown!

```
         BASR     6,0
         USING    BEGIN,6
BEGIN    L        2,N
         A        2,ONE
         ST       2,N
_____________________________
N        DC       F'8'
ONE      DC       F'1'
```

- Benefits:
  - Simplified references to addressable operands
  - Assembler assigns registers and calculates displacements
  - Improved readability and maintainability

Ordinary USING Statements: Summary

- Your promise to the assembler:
  - Assume this location will be in that register
  - Calculate base-displacement resolutions
  - Run-time addresses will be evaluated correctly
- Limitations
  - Symbolic addressing requires USINGs
    - Whether or not run-time addressing requires distinct registers
  - Multiple resolution problems
  - Base register selection rules are too easy to forget:
    1. Search USING Table for entries with relocatability attribute matching that of the expression to be resolved (no match: ASMA307W)
    2. Select entry (or entries) yielding smallest valid displacement (beyond USING range: ASMA034W indicates how far)
    3. Select highest-numbered register with that smallest displacement
    4. If an absolute expression is unresolved, try R0 with base zero
- It's very easy for you and the assembler to mis-communicate...!

New USING Statements

Goals of Any Addressing Methodology

- Increased opportunities for clear, simple coding
  - Easier to write, understand, and maintain
- Support efficient coding
  - Maximize performance without devious obscurities
  - Minimize need to remember arcane language rules
- Let the Assembler assign registers and displacements
  - Better controls over resolutions
  - More understandable and maintainable code
- Encourage fully-symbolic references to all objects

Problems with Ordinary USING Statements

- Ordinary USINGs have several shortcomings:
  1. Cannot make simultaneous references to multiple instances of a given control section
     - Unless you write “tortured” code
  2. Cannot map more than one DSECT per register
     - Unless you write “tortured” code
  3. Cannot specify fixed relationships among DSECTs at assembly time
     - Unless you write “tortured” code
- New USING statements in High Level Assembler
  - Alleviate all these problems
  - Coding can be simpler, cleaner, more understandable
  - Less need to understand complex assembler rules

New USING Statements in High Level Assembler

1. Labeled USINGs
   - Simultaneous reference to multiple instances of an object
   - One object per register
2. Dependent USINGs
   - Address multiple objects with a single register
   - Greater program efficiency (fewer base registers required)
   - Dynamic structure remapping during execution
3. Labeled Dependent USINGs
   - Combines benefits of Labeled and Dependent USINGs
   - Simultaneous reference to (possibly multiple) occurrences of multiple objects with a single register
   - Easier mapping of complex data structures

Labeled USING Statements and Qualified Symbols

- Some definitions:

  1. A qualified symbol is of the form ***qualifier.ordinary_symbol***

  2. A qualifier is an ordinary symbol also

  3. A qualifier is defined as such by appearing in the name field of a USING statement:

```
qualifier  USING  base,register
```

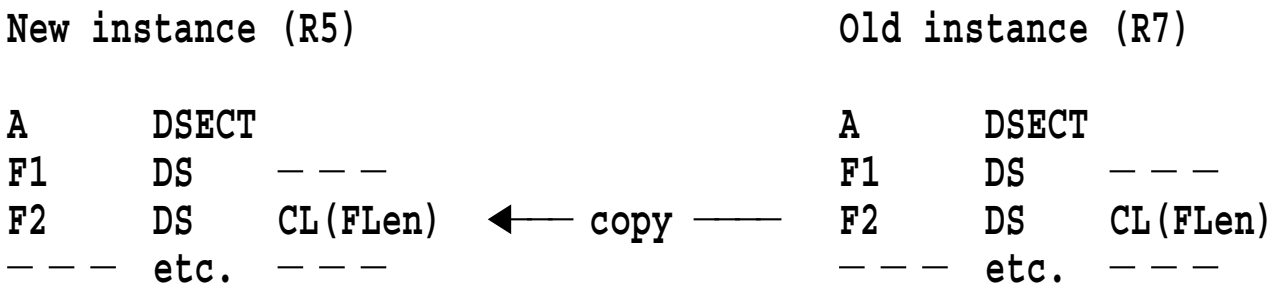
- Examples:

```
A       USING  Z,5          Qualifier A        Use:       A.B
LEFT    USING  BLOCK,9      Qualifier LEFT               LEFT.DATA
RECORD1 USING  MAPPING,3    Qualifier RECORD1        RECORD1.FIELD4
```

- Qualifiers permit “directed resolution” to a specific register

Managing Two Copies of a Data Structure

- We wish to copy a field F2 between two active copies of a DSECT:



- We'd like the assembler to understand statements like

  `MVC` $F2_{NEW}$`,`$F2_{OLD}$ or `MVC NEW_F2,OLD_F2`

- Solutions with ordinary USINGs have some shortcomings...
  - likely to be harder to understand and maintain
  - more opportunities for incorrect or inefficient code
  - harder for assembler to diagnose potential problems
  - require deeper understanding of complex instruction and language rules

Managing Two Copies of a Structure (The Hard Way)

- Some examples of solutions with ordinary USINGs:

  1. Incorrect usage:

```
USING  A,5                    USING  A,7
USING  A,7          or        USING  A,5
MVC    F2,F2                  MVC    F2,F2
```

  2. With manually-calculated displacements (1):

```
USING  A,5            map new instance of A
MVC    F2,F2-A(7)     move from old to new (Correct, but ugly)
```

  3. With manually-calculated displacements (2):

```
USING  A,7            map old instance of A
MVC    F2-A(5),F2     move from old to new (WRONG!)
```

  4. With manually-calculated displacements (3):

```
USING  A,7            map old instance of A
MVC    F2-A(,5),F2    move from old to new (Correct, but uglier)
```

Managing Two Copies of a Structure (The Hard Way)...

5. With (strangely) manually-calculated displacements (4):

```
USING  A,5            map new instance of A
USING  0,7            map old instance of A (somewhat...)
MVC    F2,F2-A        move from old to new

- - -                 more statements (forgetting to drop R0)

LA     1,100          Resolved on R7! (X'41107064')
```

6. With (desperately) manually-calculated displacements (4):

```
USING  A,5             map new instance of A
USING  0+X'F999',7     map old instance of A (differently)
MVC    F2,F2-A+X'F999' move from old to new
```

7. Manual assignments may be **wrong** if the size of DSECT A exceeds 4K bytes

```
       USING  A,5,6           map new instance of A
 *     USING  A,7,8           implicit map of old instance of A
       MVC    F2,F2-A(7)      F2-A might exceed 4095?
```

Managing Two Copies of a Structure (The Hard Way)...

8. With an intermediate temporary (1):

```
        USING  A,7             map old instance of A
        MVC    TEMP(FLen),F2   move from old to temp
        USING  A,5             map new instance of A
        MVC    F2,TEMP         move from temp to new (WRONG!)
```

9. With an intermediate temporary (2):

```
        USING  A,7             map old instance of A
        MVC    TEMP(FLen),F2   move from old to temp
        DROP   7               must DROP register 7 first
        USING  A,5             map new instance of A
        MVC    F2,TEMP         move from temp to new (RIGHT!)
```

10. With a duplicated copy of the DSECT:

```
B       DSECT                  B is a copy of A
G1      DS     ---
G2      DS     CL(FLen)
---     etc.   ---
        USING  B,7             map old instance of A (named B)
        USING  A,5             map new instance of A
        MVC    F2,G2           move from old to new
```

- Each of these examples is not untypical of current coding styles...

Labeled USINGs: The Best Solution

- Labeled USINGs provide a simple solution:

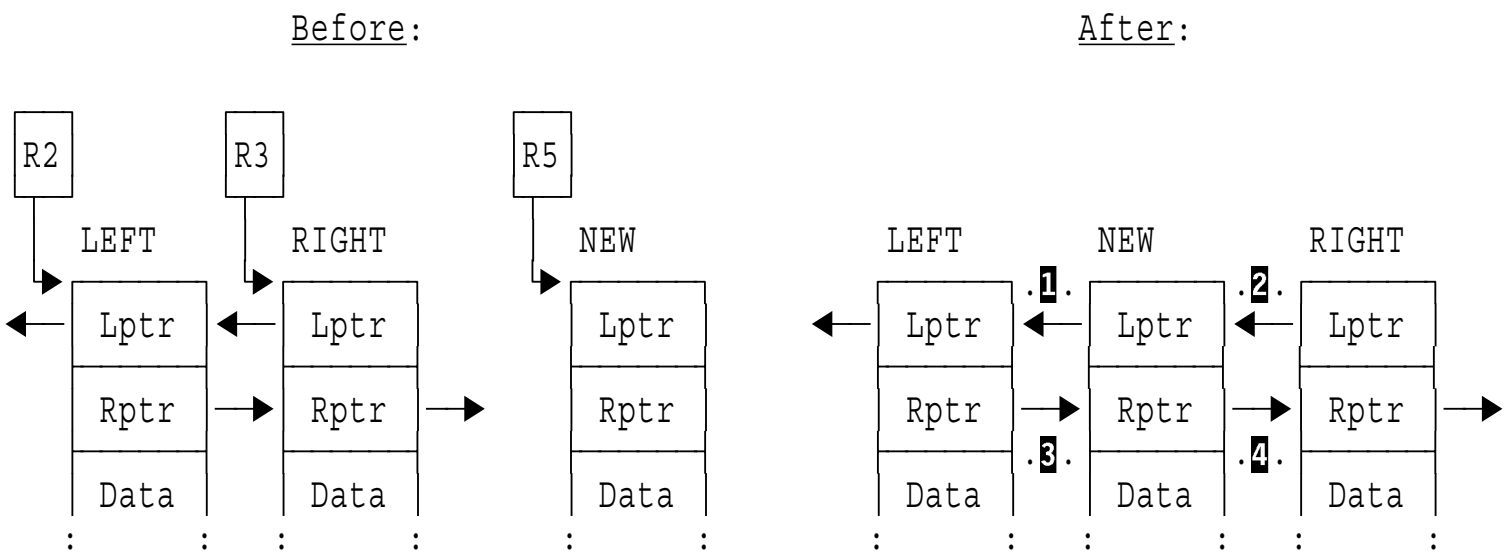
```
OLD .1. USING  A,7                map old instance of A

NEW .2. USING  A,5                map new instance of A

        MVC    NEW.F2,OLD.F2      move field from old to new

               .4.     .3.
```

  - Qualifier `OLD .1.` resolves symbol `.3.` and qualifier `NEW .2.` resolves `.4.`

- Advantages of labeled USINGs
  - data objects need only one definition
  - all references are fully symbolic
  - no manually-specified displacements and registers
  - efficient solution is also the most natural
  - no need to understand obscure details of Assembler Language

Example: Doubly-Linked List Structure

- Insert a NEW element in a doubly-linked list:



- Labeled USINGs provide clean, understandable solution
  - Many complex, obscure solutions possible with ordinary USINGs

Labeled USINGs: Doubly-Linked List

- Code with labeled USINGs is very simple:

```
BLOCK  DSECT
Lptr   DS    A              Pointer to left  element
Rptr   DS    A              Pointer to right element
Data   DS    XL24,D,E etc.  Data fields within BLOCK
       — — —

RNew   Equ   5                          R5 points to New element
Left   USING Block,2                    Labeled USING
Right  USING Block,3                    Labeled USING
New    USING Block,RNew                 Labeled USING
       — — —
       MVC   New.Lptr,Right.Lptr  .1.  Qualified symbols
       ST    RNew,Right.Lptr      .2.  Qualified symbol
       MVC   New.Rptr,Left.Rptr   .3.  Qualified symbols
       ST    RNew,Left.Rptr       .4.  Qualified symbol
```

- Advantages: clarity, simplicity, readability, efficiency, maintainability

Labeled USING Statements: a Summary

- Resolutions done only for symbols with matching qualifier
- Normal resolution rules still apply
  - Matching relocatability attribute
  - Displacement cannot exceed 4095
- May be concurrent with ordinary USING for same register

```
         USING  A,9          Ordinary USING
 Q       USING  A,9          Labeled  USING
         — — —
         LA     0,A+40       Resolved only with Ordinary USING
         LA     1,Q.A+40     Resolved only with Labeled  USING
         DROP   9            Drop ordinary USING; labeled still active
         LA     2,Q.A+40     Resolved only with Labeled  USING
         DROP   Q            Drop labeled USING
```

- Care is recommended!
  - Avoid mixing qualified and unqualified symbol references

Dependent USING Statements

- Lets you address multiple DSECTs with one base register
- Syntax is the same as for ordinary USINGs:

```
USING  symbol,base
```

- Except that the second operand is interpreted differently:

ordinary: second operand is absolute, between 0 and 15

```
USING  symbol,register
```

dependent: second operand is relocatable, addressable

```
USING  symbol,anchor_location
```

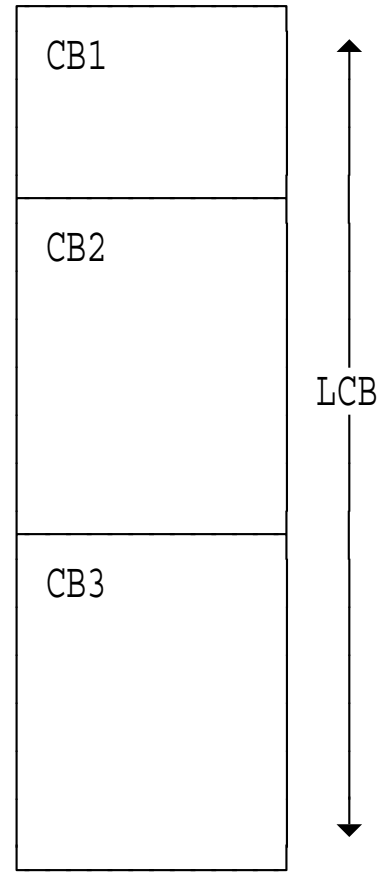
- First operand is “based” or “anchored” at second operand location

Dependent Using Statement Examples

- Example: DSECTs B and C anchored at different offsets within A

```
        R:F  00000          9      USING  A,15     Ordinary: Addr(A) in R15
      F 020  00000 00020   10      USING  B,A+32   Dependent: B at A+X'20'

00058 4100 F028    00008   12      LA     0,B2     B2 at offset X'28' from A

      F 030  00000 00010   14      USING  C,B+16   Dependent: C at B+X'10'

0005C 4100 F080    00050   16      LA     0,C2     C2 at offset X'80' from A

00000                      18 B    DSECT
00000                      19 B1   DS     D        Offset    0  from B
00008                      20 B2   DS     D        Offset    8  from B

00000                      22 C    DSECT
00000                      23 C1   DS     CL80     Offset    0  from C
00050                      24 C2   DS     XL8      Offset X'50' from C

00000                      26 A    DSECT
00000                      27      DS     XL256
```

Dependent USING Example: Contiguous Control Blocks



Contiguous
Control Blocks

```
CB1    DSECT ,         Define control block 1
CB1F1  DS    D
CB1F2  DS    CL40
LCB1   EQU   *-CB1     Length of block 1

CB2    DSECT ,         Define control block 2
CB2F1  DS    24F
LCB2   EQU   *-CB2     Length of block 2

CB3    DSECT ,         Define control block 3
CB3F1  DS    XL8,CL80
LCB3   EQU   *-CB3

LCB    EQU   LCB1+LCB2+LCB3  Total length
```

Contiguous Control Blocks: Ordinary USINGs

- Ordinary USINGs require a register for each DSECT:

```
*       GET (LCB bytes) STORAGE FOR ALL 3 BLOCKS, BASE ADDRESS IN R7
        — — —
        USING CB1,7             Anchor the first storage block
        LA    6,CB1+LCB1        Calculate address of second block
        USING CB2,6             Anchor the second storage block
        LA    4,CB2+LCB2        Calculate address of third block
        USING CB3,4             Anchor the third storage block
```

- Defects:
  - Extra base registers
  - Additional initialization overhead
- Devious coding techniques:

```
        USING CB1,7             Anchor the first storage block
        L     0,CB1+LCB1+(CB2F1—CB2)+8  3rd element of CB2F1 array
        — — —
```

- Defects:
  - Complex coding that is hard to understand and maintain
  - Relationships among CBs is embedded in each referencing instruction

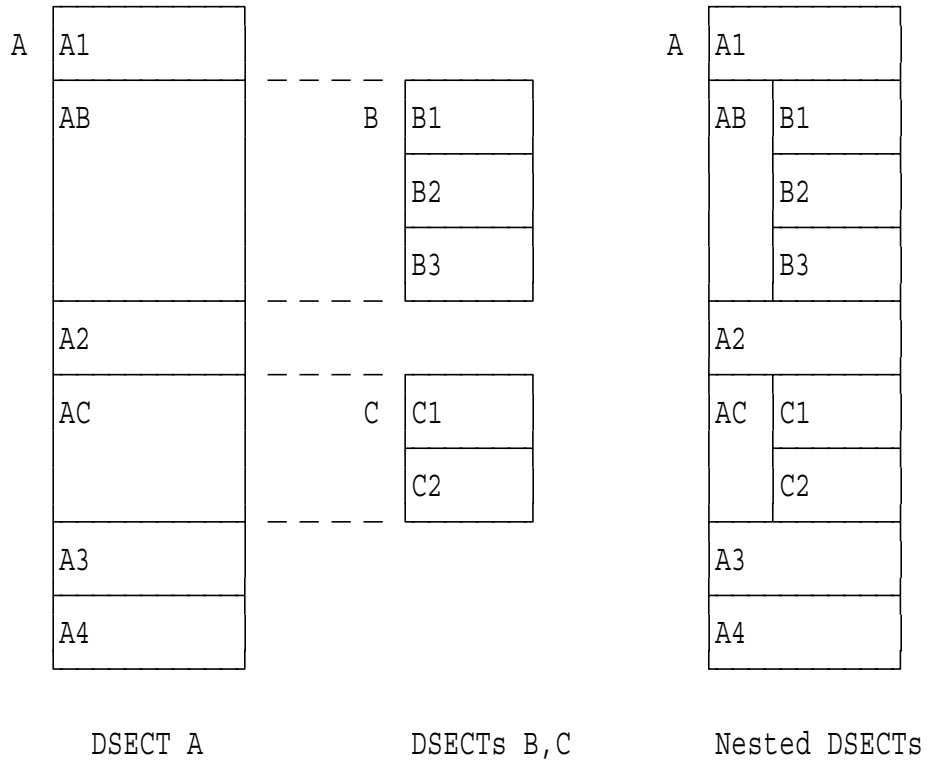
Contiguous Control Blocks: Dependent USINGs

- Dependent USINGs require only a single base register:

```
*       GET (LCB bytes) STORAGE FOR ALL 3 BLOCKS, BASE ADDRESS IN R7
        — — —
        USING CB1,7             Anchor the full storage block
        USING CB2,CB1+LCB1      Adjoin CB2 to CB1 (dependent USING)
        USING CB3,CB2+LCB2      Adjoin CB3 to CB2 (dependent USING)

        STM   14,12,CB2F1+12    Addresses resolved with
        XC    CB3F1,CB3F1       ... just one base register (R7)
        UNPK  CB1F1,CB1F2(4)    ... for all these instructions
```

- Advantages:
  - Minimal number of base registers needed
  - No run-time initialization overhead
  - Independently defined data structures

Dependent USING Example: Nested Structures




```
A     DSECT
A1    DS     24F
AB    DS     CL(LB)
A2    DS     6CL80
AC    DS     CL(LC)
A3    DS     XL16
A4    DS     CL256

B     DSECT
B1    DS     CL44
B2    DS     6D
B3    DS     4A
LB    EQU    *-B

C     DSECT
C1    DS     96D
C2    DS     4XL20
LC    EQU    *-C
```

Nested Structures with Multiple Ordinary USINGs

- Each DSECT requires its own base register:

```
*                       Assume address of A is in R7
         USING A,7      Ordinary USING for A
         LA    5,AB     Address of AB in R5
         USING B,5      Ordinary USING for B
         LA    4,AC     Address of AC in R4
         USING C,4      Ordinary USING for C
```

- Defects:
  - Loss of efficiency: extra registers, execution-time setup
  - Precise relationship of instructions to structure elements is not as clear

Nested Structures with Dependent USINGs

- Dependent USINGs allow these to be addressed with a single register:

```
*                           Assume address of A is in R7
         USING A,7          Ordinary USING for A
         USING B,AB         Dependent USING: anchor B at AB
         USING C,AC         Dependent USING: anchor C at AC
```

- Benefits of dependent USINGs:
  - More efficient solution
    - Minimal number of registers needed for addressing
    - No execution-time register setup
  - Simpler, clearer code
    - Clear separation of data definitions and instructions

Nested Structures with One Ordinary USING

- Can map nested structures with a single ordinary USING
  - Calculate DSECT offsets “manually”

```
*                           Assume address of A is in R7
         USING A,7          Ordinary USING for A
         — — —
         L     0,AB+(B3—B)  Field B3 within DSECT B
         C     0,AC+(C2—C)  Field C2 within DSECT C
```

- Will need to write a lot of this if many references to DSECT fields
- Dependent USING is clearer, easier to write and maintain

```
         USING A,7          Ordinary USING for A
         USING B,AB         Map DSECT B into A at AB
         USING C,AC         Map DSECT C into A at AC
         — — —
         L     0,B3         Field B3 within DSECT B
         C     0,C2         Field C2 within DSECT C
```

- Let the assembler do the hard work!
  - It calculates the same displacements as you did (with difficulty)

Dependent USINGs and Disjoint USING Ranges

- Range-limited USINGs restrict resolution range

```
        USING  (start_range,end_range),anchor
```

- Only “My Code” and “My Literals” addressed by “My USING”
  - Addressing anything else should be an error

```
        USING MyCode,9
MyCode  DS    0H                Start of my code
    ----My code  ---
EndCode DS    0H                End of my code
        DROP  9                 No further use of my base register

    ---other code ---           Code/data that I shouldn't address,
    ---           ---           and that shouldn't use R9 as a base

MyLits  LTORG ,                 Start of my (and others') literals
        ---
EndLits EQU   *                 End of my literals
```

- Specify restricted ranges for “only my stuff”

```
        USING (MyCode,EndCode),9        Address only my code
        USING (MyLits,EndLits),MyCode   Address literals only
```

Labeled Dependent USING Statements

- Labeled dependent USINGs combine the benefits of labeled and dependent USINGs:
  - labeled: multiple copies of an object may be active simultaneously
  - dependent: many objects may be addressed with a single base register
- Syntax combines elements of labeled and dependent USINGs

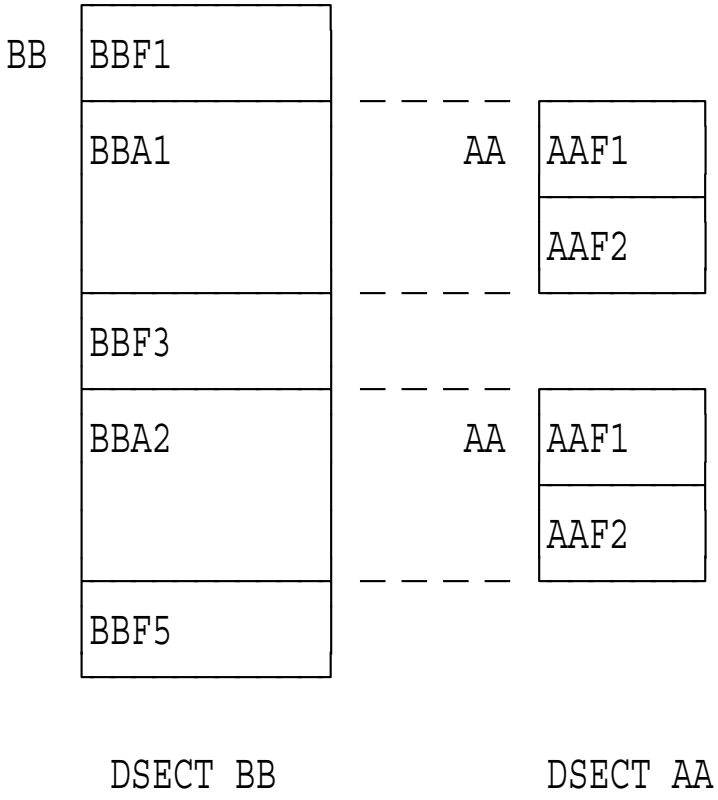
```
label    USING operand1,operand2       Operand2 is relocatable
```

- Example: overlay two instances of DSECT DZ within A

```
Z1       USING DZ,A+12       Overlay DZ at A+12, qualify with "Z1"
Z2       USING DZ,A+82       Overlay DZ at A+82, qualify with "Z2"
```

Two Nested Identical Structures

- Nest two instances of AA within BB




```
AA    DSECT
AAF1  DS    XL5
AAF2  DS    XL8
LAA   EQU   *-AA

BB    DSECT
BBF1  DS    XL17
BBA1  DS    XL(LAA)
BBF3  DS    XL11
BBA2  DS    XL(LAA)
BBF5  DS    XL7
LBB   EQU   *-BB
```

Addressing Two Nested Identical Structures

- With ordinary USINGs

```
.censored.
```

- Labeled USINGs require 3 base registers, “setup” overhead

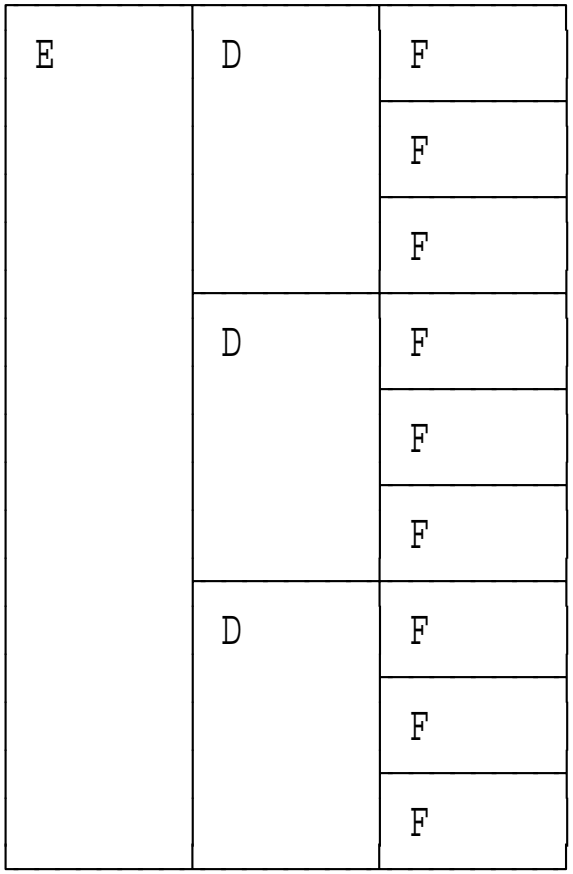
```
      USING BB,10           R10 points to BB
      LA    11,BBA1         R11 points to 1st copy of AA
A1    USING AA,11           Labeled USING for 1st copy of AA
      LA    12,BBA2         R12 points to 2nd copy of AA
A2    USING AA,12           Labeled USING for 2nd copy of AA
```

- Labeled dependent USINGs require only one base register

```
      USING BB,10           R10 points to BB
A1    USING AA,BBA1         Labeled dependent USING for 1st copy of AA
A2    USING AA,BBA2         Labeled dependent USING for 2nd copy of AA
```

- Even if BB exceeds 4K bytes, this is still better

Multiple Nested Structures




```
F       DSECT ,
X1      DS    XL5
X2      DS    XL5
LF      EQU   *-F

D       DSECT ,
F1      DS    XL(LF)
F2      DS    XL(LF)
F3      DS    XL(LF)
LD      EQU   *-D

E       DSECT ,
D1      DS    XL(LD)
D2      DS    XL(LD)
D3      DS    XL(LD)
```

- Problems:
  - Multiple instances of structures D and F
  - Ordinary or labeled USINGs require 13 base registers!

Multiple Nested Structures: Labeled Dependent USINGs

- Mapping nested structures with labeled dependent USINGs

```
         USING E,7                      1 Top level
*                                       |
D1E      USING D,D1       .1.           |  2 Map D1 into E at D1
D1F1     USING F,D1E.F1     .2.         |  |  3 Map F1 into D1 at F1
D1F2     USING F,D1E.F2     .2.         |  |  3 Map F2 into D1 at F2
D1F3     USING F,D1E.F3     .2.         |  |  3 Map F3 into D1 at F3
*                                       |  2 Middle level
D2E      USING D,D2       .1.           |  |  Map D2 into E at D2
D2F1     USING F,D2E.F1     .3.         |  |  3 Map F1 into D2 at F1
D2F2     USING F,D2E.F2     .3.         |  |  3 Map F2 into D2 at F2
D2F3     USING F,D2E.F3     .3.         |  |  3 Map F3 into D2 at F3
*                                       |  2 Middle level
D3E      USING D,D3       .1.           |  |  Map D3 into E at D3
D3F1     USING F,D3E.F1     .4.         |  |  3 Map F1 into D3 at F1
D3F2     USING F,D3E.F2     .4.         |  |  3 Map F2 into D3 at F2
D3F3     USING F,D3E.F3     .4.         |  |  3 Map F3 into D3 at F3
```

- Qualifiers indicate which references apply to which instance

Multiple Nested Structures: Referencing Fields

- All symbol references to individual fields are qualified:

```
*   Move fields named X within DSECTs described by F
    MVC   D1F1.X1,D1F1.X2        Within bottom-level DSECT D1F1
    MVC   D1F3.X2,D1F1.X1        Across bottom-level DSECTs in D1
    MVC   D3F2.X2,D3F3.X2        Across bottom-level DSECTs in D3
    MVC   D2F1.X1,D3F2.X2        Across bottom-level DSECTs in D2 and D3

*   Move DSECTs named F within DSECTs described by D
    MVC   D3E.F1,D3E.F3          Within mid-level DSECT D3E
    MVC   D1E.F3,D2E.F1          Across mid-level DSECTs D1E, D2E

*   Move DSECTs named D within E
    MVC   D1,D2                  Across top-level DSECTs D1, D2
```

- Can address structures as fields, sub-sub-structures, and sub-structures

Two MVS DCBs Within a Program

- Program fragment containing two DCBs and code:
  part of program must copy input-DCB's LRECL to output DCB

```
          LA    3,OUTDCB            Point to Output DCB
          LA    2,INDCB             Point to Input DCB
          USING IHADCB,2
          MVC   DCBLRECL—IHADCB(2,3),DCBLRECL    Copy IN LRECL to OUT
          — — —
INDCB     DCB   DDNAME=..., etc.
OUTDCB    DCB   DDNAME=..., etc.
          — — —
          DCBD  DSORG=PS,DEVD=DA,...etc.  Generate IHADCB DSECT
```

```
          USING *,12
          — — —
IN   .1.  USING IHADCB,INDCB              Labeled dependent USING
OUT  .2.  USING IHADCB,OUTDCB             Labeled dependent USING
          — — —
          MVC   OUT.DCBLRECL,IN.DCBLRECL  Addresses resolved via R12
                .2.          .1.
```

- Only one register needed to address code and two DSECTs!

Personnel-File Employee Record

- Example: a “personnel-file” record describing an employee

```
Employee DSECT ,                  Employee record
EPerson  DS    CL(LPerson)        Person field
EHire    DS    CL(LDate)          Date of hire
EWAddr   DS    CL(LAddr)          Work (external) address
EPhoneW  DS    CL(LPhone)         Work telephone
EPhoneF  DS    CL(LPhone)         Work Fax telephone
EMarital DS    X                  Marital Status
ESpouse  DS    CL(LPerson)        Spouse field
E#Deps   DS    CL2                Number of dependents
EDep1    DS    CL(LPerson)        Dependent 1
EDep2    DS    CL(LPerson)        Dependent 2
EDep3    DS    CL(LPerson)        Dependent 3
LEmploye EQU   *-Employee         Length of Employee record
```

- Many fields are described by other DSECTs:
  - Person, Date, Addr, Phone

Personnel-File Employee Record: “Person” Fields

- An individual is described by the `Person` DSECT:

```
Person   DSECT ,                 Define a "Person" field
PFName   DS    CL20              Last (Family) name
PGName   DS    CL15              First (Given) name
PInits   DS    CL3               Initials
PDoB     DS    CL(LDate)         Date of birth
PAddr    DS    CL(LAddr)         Home address
PPhone   DS    CL(LPhone)        Home telephone number
PSSN     DS    CL9               Social Security Number
PSex     DS    CL1               Gender
LPerson  EQU   *-Person          Length of Person field
```

- Some fields are described by other DSECTs:
  - `Date, Addr, Phone`

Personnel-File Employee Record: “Date,” “Addr” Fields

- Dates and addresses are described by `Date, Addr` DSECTs:

```
Date     DSECT ,                  Define a calendar date field
Year     DS    CL4                YYYY
Month    DS    CL2                MM
Day      DS    CL2                DD
LDate    EQU   *-Date             Length of Date field
         ORG   Date
DateF    DS   0CL(LDate)          Full YYYYMMDD date
         ORG   ,                  End of Date DSECT

Addr     DSECT ,                  Define an address field
AStr#    DS    CL30               Street number
APOBApDp DS    CL15               P.O.Box, Apartment, or Department
ACity    DS    CL24               City name
AState   DS    CL2                State abbreviation
AZip     DS    CL9                U.S. Post Office Zip Code
LAddr    EQU   *-Addr             Length of Address field
         ORG   Addr
AddrF    DS   0CL(LAddr)          Full address
         ORG   ,                  End of Addr DSECT
```

Personnel-File Employee Record: Comparing Birth Dates

- Example 1: Compare employee and spouse birth dates
  - Requires two active instances of `Person` DSECT

```
         USING Employee,10        Assume R10 points to the record

PE  .1.  USING Person,EPerson     Overlay Person DSECT on Empl. field
PS  .2.  USING Person,ESpouse     Overlay Person DSECT on Spouse field

* Example 1: Compare Employee and Spouse Dates of Birth

         CLC   PE.PDoB,PS.PDoB    Compare Employee/Spouse birth dates
               .1.     .2.
```

- Employee's Date of Birth (`PDoB`) qualified by PE (.1.), spouse's by PS (.2.)

Personnel-File Employee Record: Comparing Dates

- Example 2: Compare employee date of hire to dependent 1 birth date
  - Two active instances of `Date` DSECT

```
* Example 2: Compare Date of Hire to Birthdate of Dependent 1

EHD .3.  USING Date,EHire          Overlay Date DSECT on Date of Hire

PD1 .4.  USING Person,EDep1        Overlay Person DSECT on Dependent 1
DD1 .5.  USING Date,PD1.PDoB       Overlay Date DSECT on Dependent 1
                     .4.

         CLC   EHD.DateF,DD1.DateF Compare hire date to Dep 1 DoB
               .3.       .5.

         DROP  EHD,DD1             Remove both date associations
```

- Dependent's `Person` DSECT qualified by PD1 (.4.)
- Hire date qualified by EHD (.3.), dependent birthdate by DD1 (.5.)

Personnel-File Employee Record: Copying Addresses

- Example 3: Copy employee address to dependent 2 address
  - Two active instances of `Addr` DSECT

```
* Example 3: Copy Employee Address to Dependent 2 address

AE  .6.  USING Addr,PE.PAddr       Overlay Addr DSECT on Employee name
                    .1.
PD2 .7.  USING Person,EDep2        Overlay Person DSECT on Dependent 2
AD2 .8.  USING Addr,PD2.PAddr      Overlay Addr DSECT on Dep. 2 Person
                    .7.

         MVC   AD2.AddrF,AE.AddrF  Copy Employee Addr to Dependent 2
               .8.       .6.

         DROP  PD2                 Remove Dependent 2 associations
```

- Dependent's `Person` DSECT qualified by PD1 (.7.)
- Employee address qualified by AE (.6.), dependent's by AD2 (.8.)

Summary of USING Statements

USING Type	La-bel	Register Usage	Oper-and 1 Based on	Operand 2	Operand 2 Location in Storage	Number of Instances of Active Objects
Ordi-nary	no	one register per object	register	absolute [0,15]	anywhere in storage	only one active instance of an object at a time
Label-ed	yes	one register per object	register	absolute [0,15]	anywhere in storage	as many active instances of an object as registers assigned

Summary of USING Statements ...

USING Type	La-bel	Register Usage	Oper-and 1 Based on	Operand 2	Operand 2 Location in Storage	Number of Instances of Active Objects
Depen-dent	no	multiple objects per register	operand 2	relocatable, addressable	within addressability range of ordinary USINGs	multiple active objects of different types
Labeled Depen-dent	yes	multiple objects per register	operand 2	relocatable, addressable	within addressability range of ordinary USINGs	multiple active objects of the same or different types

DROP Statement Extensions

USING Type	DROP Statement
Ordinary	By register number
Labeled	By qualifying label (dropping the register has no effect)
Dependent	By register number (all sub-dependent USINGs dropped automatically)
Labeled Dependent	By qualifying label (dropping the register has no effect)

- Examples:

```
          Ordinary:     DROP  9
           Labeled:     DROP  QUAL
         Dependent:     DROP  12
  Labeled Dependent:    DROP  QUAL
```

Generalized Object File Format (GOFF)

- *Removes limitations associated with old object module format:*
  - External names to 63 characters
  - Section sizes up to 2GB (addresses to 31 bits)
  - Multi-component, multi-modal modules
  - Ability to retain “Assembler Data” with object code
  - And much more...
- Controlled by `GOFF` option
  - Independent of `DECK` or `OBJECT`
    - Assembler produces only one type of object file, old or new
  - Requires “wide” listing format (LIST(133) or LIST(MAX) option)
  - Enables use of CATTR, XATTR statements
    - Assign class names and external symbol attributes
    - One assembly can create many RMODE(24) and RMODE(31) “segments”
    - Entry points can have their own AMODEs
- Utilizes enhanced capabilities of DFSMS Binder, Program Objects
  - Existing programs can use GOFF transparently

Internal Conditional-Assembly Functions

- All IBM System/360/370/390 assemblers provide four functions:
  - Boolean connectives (AND, OR, NOT) and character substrings

```
&Bool1  SetB  (&Bool2 AND (&Bool3 OR NOT &Bool4))   Boolean functions
&Char1  SetC  '&Char2'(&Start,&Length)              Substring function
```

- High Level Assembler provides 16 *internal* functions:
  - Arithmetic functions for arithmetic (fullword integer) values
    - Masking/logical operations: AND, OR, NOT, XOR
    - Shifting operations: SLL, SRL, SLA, SRA
  - Boolean connective: XOR
  - Character functions:
    - Unary operations: UPPER, LOWER, DOUBLE, BYTE, SIGNED
    - Binary operations: INDEX, FIND
  - Extensible to other functions as required

- . . . and two statements for invoking *external* functions:
  - Arithmetic-valued functions: SETAF
  - Character-valued functions: SETCF

Internal Arithmetic-Valued Functions

- Arithmetic functions operate on fullword integer (SETA) values
- Masking/logical operations: AND, OR, NOT, XOR

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

- Shifting operations: SLL, SRL, SLA, SRA

```
&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
```

- Any combination...

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

Boolean Operators

- Logical operators: AND, OR, NOT previously available

```
&A      SetB  (&V gt 0 AND &V le 7)          &V between 1 and 7
&B      SetB  ('&C' lt '0' OR '&C' gt '9')   &C not a digit
&Z      SetB  (&A AND NOT &B)
```

- New operator: XOR

```
&S      SetB  (&B XOR (&G OR &D))
&T      SetB  (&X ge 5 XOR (&Y*2 lt &X OR &D))
```

- Simplifies “either but not both” testing:

```
&NotBoth  SetB  ((&J OR &K) AND NOT (&J AND &K))  Previously
&NotBoth  SetB  (&J XOR &K)                       With XOR
```

- Evaluation priority: NOT, AND, OR, XOR

Internal Character Functions

- Seven internal character-valued functions
- Unary functions: UPPER, LOWER, DOUBLE, BYTE, SIGNED

```
&X_Up    SetC   (Upper '&X')   All letters in &X set to upper case
&Y_Low   SetC   (Lower '&Y')   All letters in &Y set to lower case
&Z_Pair  SetC   (Double '&Z')  Ampersands/apostrophes in &Z doubled
&Blank   SetC   (Byte 64)      Sets &Blank to C' '
&Minus3  SetC   (Signed -3)    Sets &Minus3 to '-3'
```

- Binary arithmetic-valued functions: INDEX, FIND
- INDEX returns offset of first match in 1st operand string of 2nd operand string

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

- FIND returns offset of first match in 1st operand string of any character of the 2nd operand

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

External Conditional-Assembly Functions

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

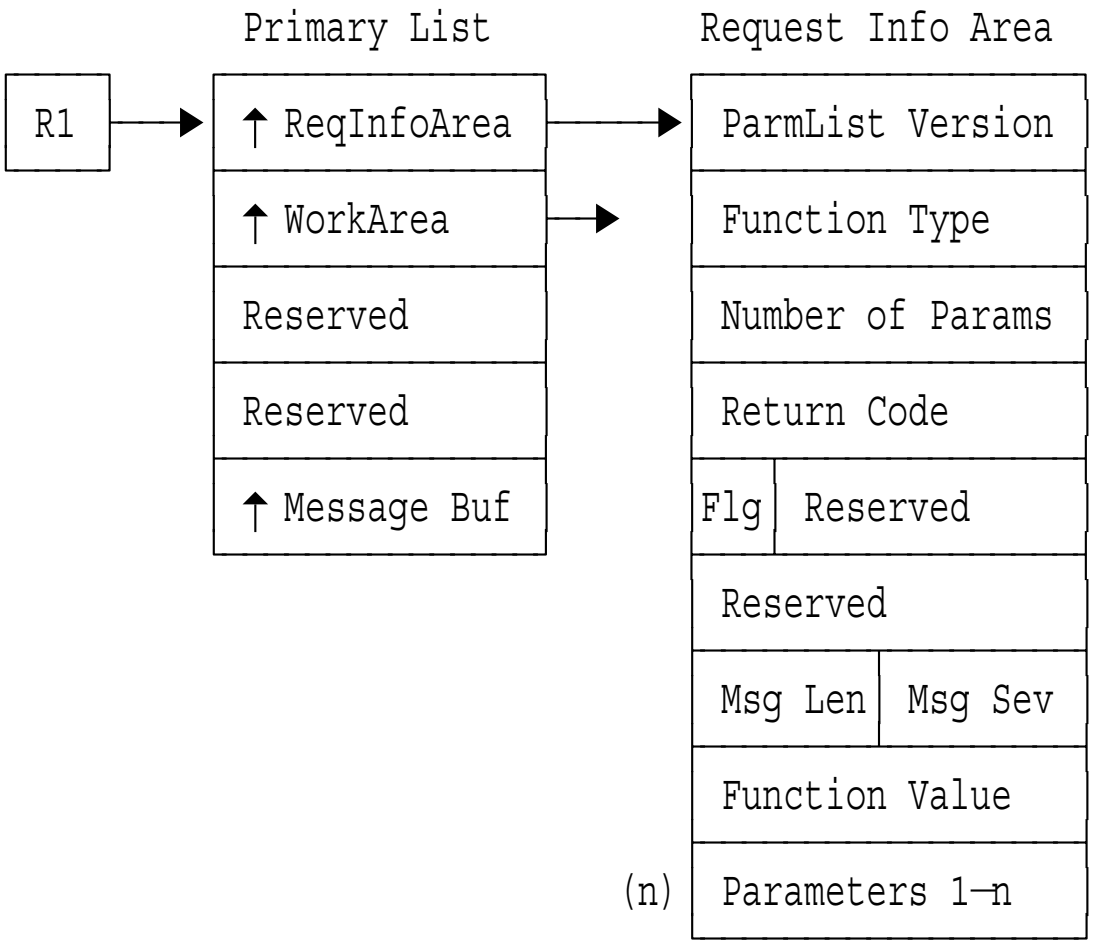
```
&A     SetAF  'AFunc',&V1,&V2,...      Arithmetic arguments
&LogN  SetAF  'Log2',&N                Logb(&N)
```

  2. Character functions: like `&C = CFunc('&S1', '&S2', ...)`

```
&C     SetCF  'CFunc','&S1','&S2',...  String arguments
&RevX  SetCF  'Reverse','&X'           Reverse(&X)
```

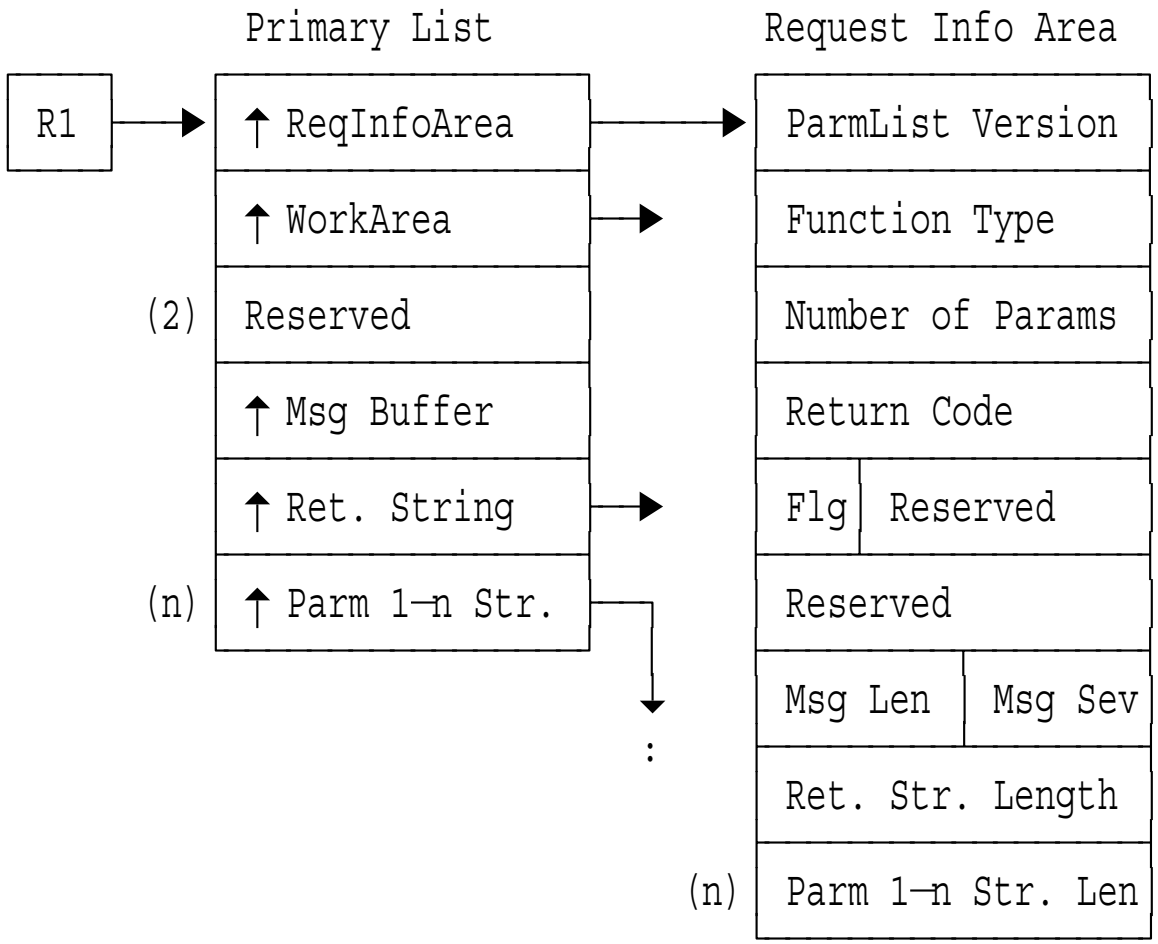
- Functions may have zero to many arguments
- Assembler's call uses standard linkage conventions
  - Assembler provides a save area and a 4-doubleword work area
- Functions may provide messages for the listing (as may I/O exits)
- Return code indicates success or failure
  - Failure return terminates 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 Variable Symbols: History and Overview

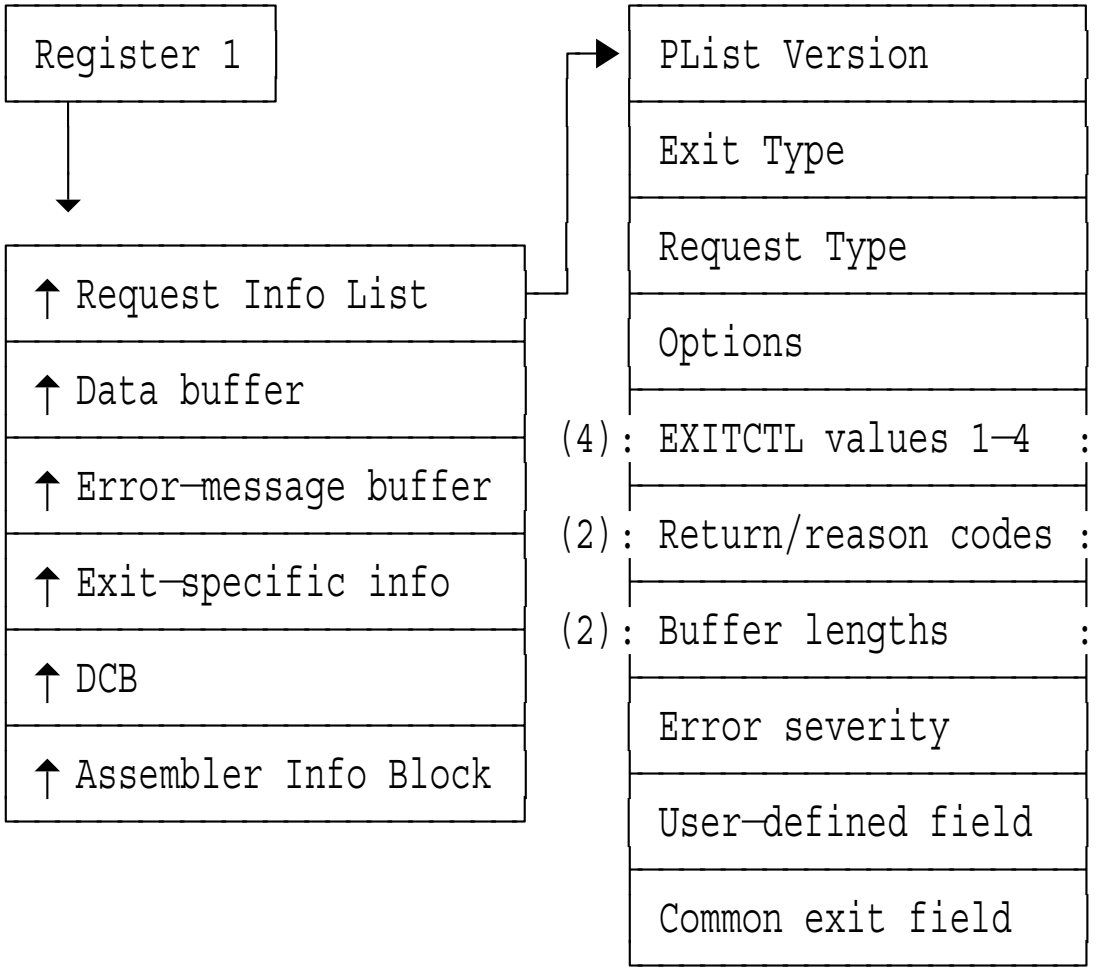
- Symbols whose value is defined by the assembler
  - Three in the OS/360 (1966) assemblers: `&SYSECT, &SYSLIST, &SYSNDX`
  - DOS/TOS Assembler (1968) added `&SYSPARM`
  - Assembler XF (1971) added `&SYSDATE, &SYSTIME`
  - Assembler H (1971) added `&SYSLOC`
  - High Level Assembler provides 39 additional symbols
- Symbol characteristics include
  - Type (arithmetic, boolean, or character)
  - Type attributes (mostly `'U'` or `'O'`)
  - Scope (usable in macros only, or in open code and macros)
  - Variability (when and where values might change)

Input-Output Exits

- HLASM supports powerful exit interfaces for all user files
  - SYSIN, SYSLIB, SYSPRINT, SYSPUNCH, SYSLIN, SYSTERM, SYSADATA
- Exits have as little or as much control as desired
  - Modify, insert, delete records
  - Monitor or assist assembler I/O, or replace it entirely
- Exits may produce diagnostic messages with each interaction
- Three sample exits provided:
  - Print (ASMAXPRT): options page deleted or moved to end of listing; summary page optionally deleted
  - Input (ASMAXINV): accepts V-format SYSIN records
  - ADATA (ASMAXADT): extracts/formats macro/COPY members and their library names
- `EXITCTL` statement provides source-file information to exits

Input-Output Exit Communication

- All assembler/exit communication via I/O Exit Parameter List
- Full control information
  - Control information
  - Data set information
  - Buffers, message area
  - Exit anchor word
- Assembler, exit are “coroutines”



Example Object-File Exit: OBJX

- Add Linkage Editor-Binder control statements after object modules
  - NAME and up to 32 ALIASes, optional SETSSI
    - BATCHed assemblies are properly separated by `NAME` statements
    - Can create of PDS members in two assembly-link steps

- Invoked by specifying EXIT option:

```
          EXIT(OBJEXIT(OBJX[(exit-parm)]))
or              EX(OBX(OBJX[(exit-parm)]))
```

- OBJX exit handles four one-character parameters in `exit-parm`

Q Do not write summary information messages
R Add `(R)` to NAME statements
S Provide SETSSI statements with `YYDDDHHM` date/time
T Provide tracing and debugging information