Finding and Fixing Assembler Language Problems:

How High Level Assembler Can Help

SHARE 103 (August 2004), Session 8173

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

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

Invitation

Suggestions for improvements and additions are welcomed.

Synopsis:

Assembler Language problems are sometimes difficult to locate and correct. The IBM High Level Assembler for MVS & VM & VSE provides many helpful features that can help; these notes provide an overview of those features.

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

Permission is granted to SHARE Incorporated to publish this material in the proceedings of SHARE 103 (August 2004). IBM retains the right to publish this material elsewhere.

Notice

Copyright Notices and Trademarks

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

IBM	ESA	System/370	System/370/390
System/390	MVS/ESA	OS/390	VM/ESA
VSE/ESA	VSE	z/OS	z/VM
z/VSE	z/Architecture	zSeries	DFSMS
OS/2	OS/2 Warp		

The following are trademarks or registered trademarks of other corporations:

Windows 95 Windows 98 Windows 2000 Windows NT Windows XP

Publications, Collection Kits, Web Sites

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

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

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

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

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

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

HLASM publications are available online at the HLASM web site:

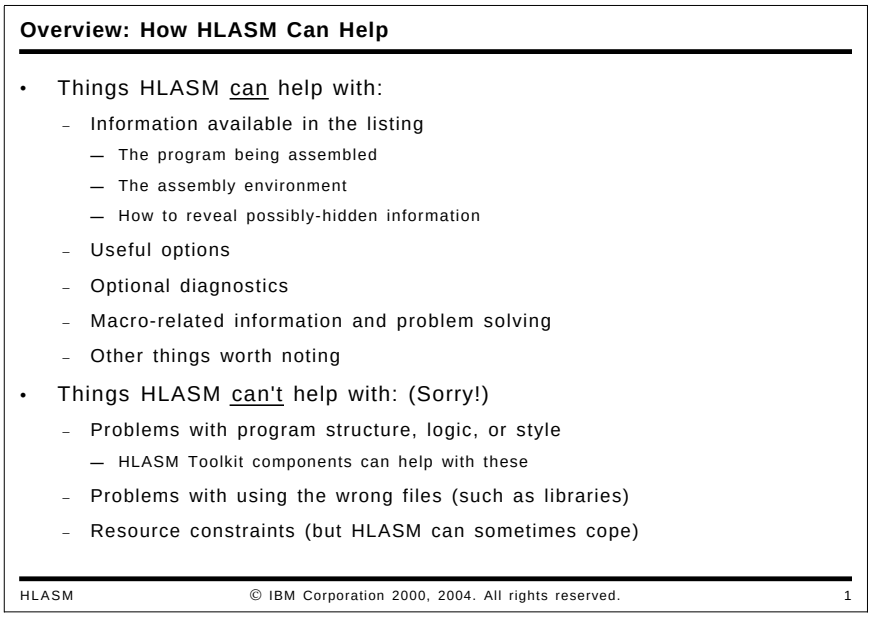
`http://www.ibm.com/software/ad/hlasm/`

(Revised 01 Jul 2004, 1940, Formatted 01 Jul 04, 1106.)

Contents

Figures

Overview



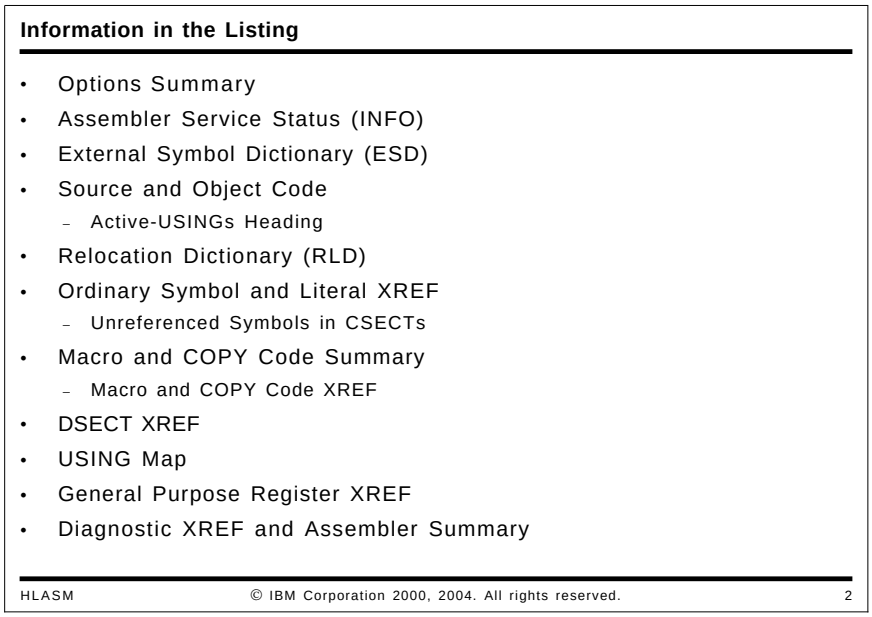
The High Level Assembler for MVS & VM & VSE provides extensive information about the programs it assembles and the assembly environment, and supports flexible controls over both the displayed information and diagnostics to be applied to the program.

This document summarizes many ways to benefit from the capabilities of the High Level Assembler, particularly for locating problems with Assembler Language programs.

High Level Assembler is designed to assemble programs efficiently, but it does not try to create an “overview” or comprehensive analysis of the program as a whole. Thus, matters such as coding style, program structure and organization, and logic are largely invisible to the assembler. Certain statements with wide-ranging effects, such as USING instructions, are analyzed with care, but this analysis is based only on the information known at the time the statement is processed.

The High Level Assembler for MVS & VM & VSE Toolkit Feature provides several components that can help with understanding and managing programs “in the large” such as the Interactive Debug Facility, the Program Understanding Tool, and the Source Cross-Reference Utility.

Information in the Listing

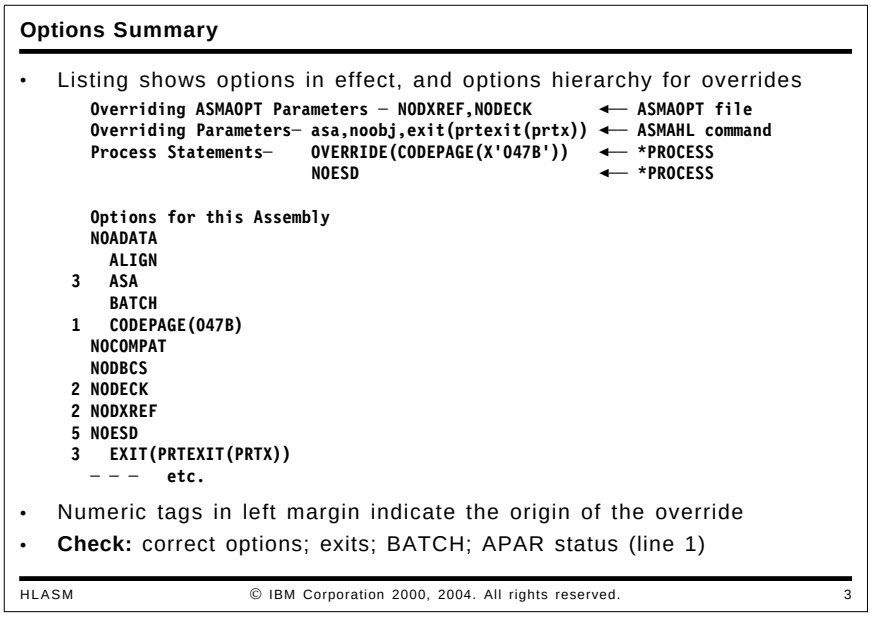


The High Level Assembler listing contains useful information about all aspects of an assembly. This information is produced in many different areas, some parts of which can be included or excluded under the control of options and source-program statements. The listing includes some or all of the following sections, in this order:

- Options Summary
- Assembler Service Status (INFO)
- External Symbol Dictionary (ESD)
- Source and Object Code
  - Active-USINGs Heading
- Relocation Dictionary (RLD)
- Ordinary Symbol and Literal XREF
  - Unreferenced Symbols in CSECTs
- Macro and COPY Code Summary
  - Macro and COPY Code XREF
- DSECT XREF
- USING Map
- General Purpose Register XREF
- Diagnostic XREF and Assembler Summary

Each of these is discussed in turn, describing useful information that you can find in each section of the listing.

Options Summary



The first page of the listing file contains:

- Fixed installation default options that were specified with DELETE operand of the ASMAOPTS installation macro
- User-supplied options from the ASMAOPT file
- User-supplied options from invocation parameters
- *PROCESS-statement options
- a list of all options in effect
- a list of overriding ddnames

An example is shown in slide 3.

The options used for an assembly are determined according to the following hierarchy (highest to lowest; the numbering corresponds to the numbers used in the options summary to indicate overrides):

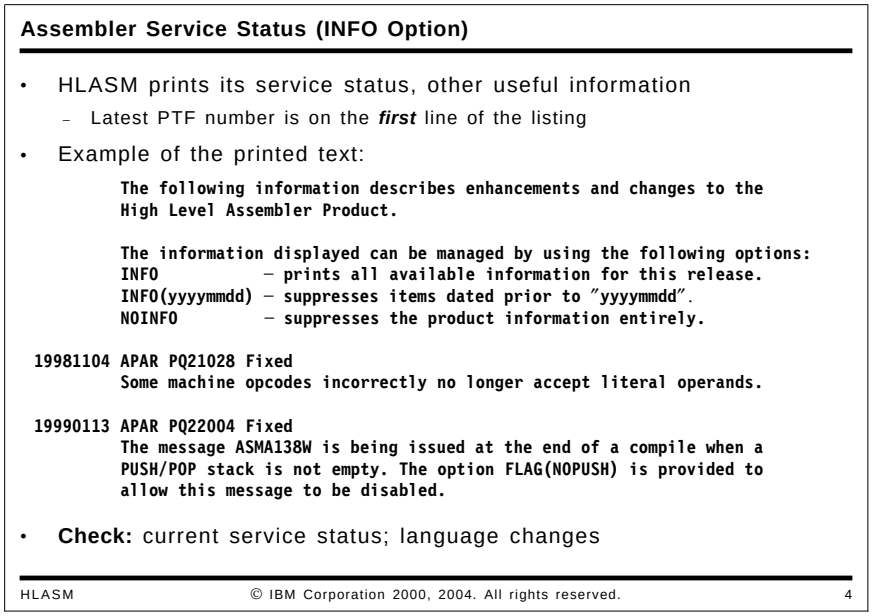
0. Fixed installation defaults
1. *PROCESS OVERRIDE options
2. Options from the ASMAOPT file (or VSE Librarian member ASMAOPT.USEROPT)
3. Options in JCL PARM (MVS, VSE) or ASMAHL command (CMS)
4. Options on JCL OPTION statement (VSE)
5. Options on *PROCESS statements
6. Non-fixed installation defaults

This summary will tell you which options were provided for the assembly. Additional information is provided about errors in the requested options.

Things Worth Checking

- First, check that the specified options are the ones you really want. Information about overrides can help you determine how the final values of the options were derived.
- Check whether I/O exits are specified: such exits have control over all assembler files (except its utility "work" file), and therefore can control what is read into the assembler and what is produced as output. For example, certain parts of the listing may have been moved, modified, or suppressed by a listing exit (including the fact that the exit is present)! (See also "I/O Exits" on page 66.)
- Check whether BATCH mode has been specified. Programs that assemble or link one way with NOBATCH may behave differently with BATCH. (See also "BATCH Option" on page 27.)
- The first line of the first page shows the latest service applied to the assembler. If you're interested in knowing what was fixed, specify the INFO option.

Assembler Service Status (INFO Option)

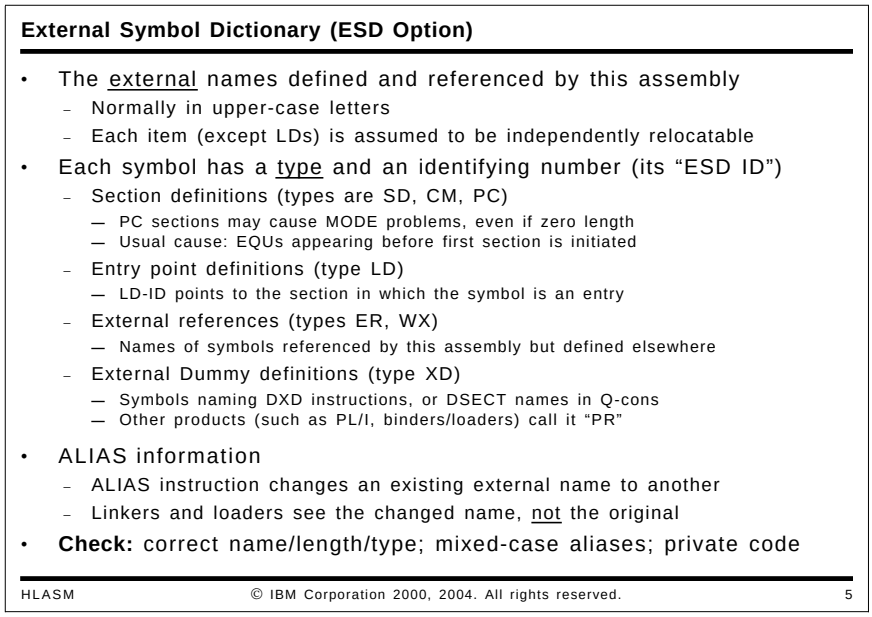


If you specify the INFO option, High Level Assembler will print a summary of the status of all service that has been applied to *your* copy of the assembler. Slide 4 shows an example.

You can request a subset of this service status data by specifying a date: INFO(yyyymmdd) requests the assembler not to display service-status information that is dated prior to the specified date.

Things Worth Checking: If the behavior of your assembler has changed, it is possible that service may have been applied that you weren't aware of. Conversely, you may have been told that some problem has been fixed, but still seems to be present on your assembler. Checking the output of the INFO option can help you determine its exact service status.

External Symbol Dictionary (ESD Option)



The ESD option causes HLASM to display information about all external symbols in the object file. There are four general types of external symbol produced by the assembler if the NOGOFF option is specified, or five if the GOFF option is specified.

The following figure illustrates an ESD listing for the traditional object module format.

```
Symbol   Type   ID     Address  Length   LD ID   Flags Alias-of
SECT_A    SD 00000001 00000000 0000002C             01
MYCOM     CM 00000002 00000000 00000060             00
MY_XD     XD 00000003 00000007 00000018
SECT_B    SD 00000004 00000030 00000038             02
B_DATA    LD          00000030          00000004
```

Figure 1. Example of External Symbol Dictionary Listing (NOGOFF Option)

In the ESD listing, several fields are displayed:

- Symbol: the external symbol. Note that all external symbols are converted to upper case letters, even if they appear in mixed case in the source file. The ALIAS instruction (see “ALIAS Information” on page 8) must be used to obtain lower case letters in external symbols.
- Symbol type: a two-letter code indicating the symbol's type.
  - Section definition

    SD An ordinary control section, specified by a CSECT, RSECT, or START instruction

    CM A common control section, specified by a COM instruction

    PC A control section with a blank name, typically caused by instructions like EQU preceding one of the other section-initiating statements (see “Private Code Sections” on page 9)
  - Label definition (**LD**)

An entry point in a previously defined section. For each LD item, there is an associated “LD ID”, which is the ESDID of the section in which this name is an entry. The “address” of the LD item should be compared to the address of its owning section; the difference between the two is the offset within the section of the entry point.

- External reference

  ER A strong external reference that is expected to be resolved to a symbol definition during linking and binding.

  WX A weak external reference that may or may not be resolved to a symbol definition during linking and binding.

- External Dummy definition (**XD**)

  A symbol appearing in the name field of a DXD instruction, or the name of a DSECT that has appeared as the operand of a Q-type address constant.

- Element definition (**ED**) (GOFF option only)

  This is the name of a binder class; the combination of a section name and a class name defines an element. The LD-ID of the class is that of the section to which it belongs. Entry points within an element are assigned an LD-ID of the owning element. (See Figure 2 on page 7 for an example.)

- ID: the ESD ID assigned to the symbol

  Each symbol is assigned a unique identification number called its “ESD ID” (except for LD items, as explained below). Because each distinct ID is assumed to represent an independently relocatable item, this is also called its “relocation ID”.

- Address: the assembler-assigned address of the symbol.

  If a section starts at a nonzero address (either due to a START instruction operand, or due to the presence of multiple control sections), this starting address is adjusted to zero during program linking and binding, and all related addresses are adjusted by the same amount. (The NOTHREAD option causes non-initial sections to have zero origin.)

  For XD items, this field contains a number one less than the desired boundary alignment of the item at bind time:

  0 byte alignment
  1 halfword alignment
  3 fullword alignment
  7 doubleword alignment
  15 quadword alignment

- Length: the assembler-assigned length of the symbol

- LD-ID: the ESDID of the section or element in which the LD item is an entry point.

- Flags: these indicate the AMODE and RMODE information associated with the symbol. Five bits in the Flags byte are used to indicate the AMODE and RMODE of an external symbol: these can be designated `B'..RA.rab'`, where

  `R` means RMODE(64)
  `A` means AMODE(64)
  `r` means RMODE(31)
  `a` means AMODE(31)
  `b` means AMODE(24)

  If all bits are zero, the modes were unspecified, and default to 24/24. Additional combinations are supported:

  - if both “a” and “b” are one, it means AMODE(ANY); that is, 24 or 31

- if both “R” and “A” are one (meaning both RMODE and AMODE are 64), the RMODE(31) bit “r” is also set, so that existing loaders can load programs below the 2GB “bar”

In Figure 1 on page 5, symbol SECTA has flags X'01' meaning RMODE(24),AMODE(24); and symbol SECTB has flags X'02' meaning RMODE(24),AMODE(31).

Note that in the old object module format, entry (LD) names have no addressing mode assigned.

- Alias: the character string that appears in the object file in place of the symbol's name, if an alias was specified. Details are provided at “ALIAS Information” on page 8.

Not all of these fields appear for every symbol type.

The following figure illustrates an ESD listing for the new (GOFF) object module format.

```
Symbol   Type   ID     Address  Length   LD ID   Flags Alias-of
SECT_A    SD 00000001
B_PRV     ED 00000002                    00000001
B_TEXT    ED 00000003 00000000 0000002C 00000001   01
SECT_A    LD 00000004 00000000           00000003   01
MYCOM     SD 00000005
B_PRV     ED 00000006                    00000005
B_TEXT    ED 00000007 00000000 00000060 00000005   00
MYCOM     CM 00000008 00000000           00000007
MY_XD     XD 00000009 00000007 00000018
SECT_B    SD 0000000A
B_PRV     ED 0000000B                    0000000A
B_TEXT    ED 0000000C 00000030 00000038 0000000A   02
SECT_B    LD 0000000D 00000030           0000000C   02
B_DATA    LD 0000000E 00000030           0000000C   02
```

Figure 2. Example of External Symbol Dictionary Listing (GOFF Option)

In Figure 2, there are several differences from the simpler form shown in Figure 1 on page 5: for each SD item, the assembler assumes that no other classes will be declared, so it emits

- ED items for classes
  - B_PRV for external dummy symbols
  - B_TEXT for text not directed to a declared class
- LD items for the name of the SD item (note that RMODE/AMODE information is provided for LD items)

Addresses and lengths are assigned to ED items, not to SD items.

One other difference is that the common declaration for symbol MYCOM appears as an SD item and as a CM item: the latter is simply an indication to the Binder that MYCOM should be linked according to the old rules for common sections.

Things Worth Checking

- Verify that all external symbols have the correct names, types, addresses, and lengths.
- For LD items, check that the LD offset is within its owning section or element. That is, the LD address does not exceed the section (or element) address plus its length.
- If an assembly has multiple sections, it is possible to refer to from one section to symbols in another without declaring names in other sections using EXTRN instructions. If the sections are always assembled together, this is not a problem; but if subsequent program linking replaces one of the sections, references to it from other sections will very probably no longer be correct.

ALIAS Information

The assembler's ALIAS instruction changes a syntactically valid external symbol to another character string in the object module. This assembly-time operation causes a “normal” external symbol defined by the program — that is, a symbol using characters valid in the Assembler Language — to be changed to a different name (its “alias”) in the External Symbol Dictionary. This permits Assembler object modules to be linked with those from other languages whose external symbols contain characters that would otherwise be syntactically invalid in the Assembler Language.

The ALIAS instruction permits 64-character external names when the GOFF option is specified, and 8-character names if the old object format is used.

This fragment of a source program:

```
         EXTRN STAR_X,Lower
STAR_X   ALIAS C'*X'
Lower    ALIAS C'Lower'
```

produces this portion of the ESD listing:

```
                                    External Symbol Dictionary

Symbol   Type   Id     Address  Length     LD ID  Flags Alias-of
X         SD 00000001 00000000 00000000              00
*X        ER 00000002                                    STAR_X
Lower     ER 00000003                                    LOWER
```

Figure 3. Example of ALIAS and the External Symbol Dictionary

In Figure 3, the EXTRN instruction in the source program declares two external symbols, STAR_X and Lower. In the absence of the ALIAS instructions, these would appear in the ESD as STAR_X and LOWER (both in upper case). The ALIAS instructions cause HLASM to change those names to *X and Lower in the object file's External Symbol Dictionary records. Internal to the program, they would still be referenced by their original names. For example,

```
          DC   V(Star_X,LOWER)
```

would generate adcons that resolve only to the “ALIASed” names (*X and Lower).

Things Worth Checking: If mixed case symbols are generated by ALIAS instructions, check that your binder/linker can recognize and process them correctly.

Private Code Sections

"Private Code" is the name the assembler gives to blank section names, because without ENTRY points in blank sections, there is no way for other code to refer to it.

The presence of PC sections in an assembly is usually unintended, and can have some unexpected side-effects. Suppose you wrote code as in the following figure: note that the ESD contains a zero-length PC section, which has a *different* AMODE/RMODE setting than the Code section.

This fragment of a source program:

```
R1    Equ   1
CODE  Start 0
CODE  RMode 31
      - - -
```

produces this portion of the ESD listing:

```
Symbol   Type   Id     Address  Length     LD ID  Flags Alias-of
          PC 00000001 00000000 00000000             00
CODE      SD 00000002 00000000 00000004             06
```

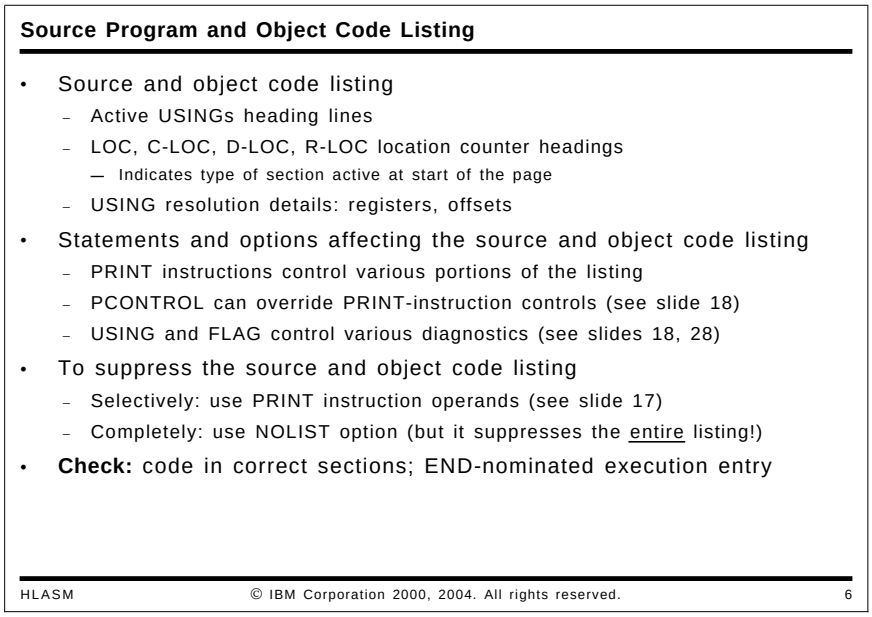
Figure 4. Program that Generates Unintended Private Code

When linked, the default AMODE/RMODE values of the PC section (24/24) may cause the desired values specified for the CODE section to be downgraded.

Things Worth Checking

- Typically, Private Code (PC) sections in an assembly have zero length, and should have no impact on a program's size at link time. However, the presence of PC sections in an assembly can lead to other problems:
  - Because they are assigned default AMODE(24) and RMODE(24) values, an entire bound module may be assigned those attributes even though other attributes were intended. Examples are shown in Figure 4 and in Figure 30 on page 29.
  - In the absence of LTORG instructions, the assembler puts the literal pool at the end of the first control section. If this is the Private Code section, it is unlikely that the literals will be addressable from other sections, causing diagnostics to be issued for each literal reference. The literals may also reside in an unexpected section, even though they appear to be addressable.

Source Program and Object Code Listing



The source and object code listing contains several useful bits of information in addition to statements, object code, and messages.

- The page heading can contain a summary of USINGs currently active as of the start of that page. This helps you understand how symbolic addresses on each page will be resolved, without having to read and annotate the entire program up to that point. This USING heading can be enabled or disabled with PRINT instruction UHEAD and NOUHEAD operands, or by specifying the PCONTROL option (see “PRINT Instructions and the PCONTROL Option” on page 30).

  The information for each USING specifies the base location, the range of the USING (in parentheses), and the base register. The range is not shown if it is the default (X'1000' times the number of base registers). If the USING is dependent, it will show the base register and the offset at which the USING is anchored; if it is labeled, the label will appear preceding the name of the DSECT.

```
Active Usings: Record,R4  ZipCode(X'F9A'),R4+X'66'  Sect1(X'CA6'),R15
HF.PhoneNo(X'F90'),R4+X'70'  WF.PhoneNo(X'F86'),R4+X'7A'  OLD.Record,R5
NEW.Record,R6
```

Figure 5. Example of Active Usings Heading

  Thus, in Figure 5, the first and third USINGs (based at Record and Sect1) are ordinary; the second is dependent; the fourth and fifth are labeled dependent; and the last two are labeled. Only the third USING specified a range limit; the other USINGs whose ranges are less than the X'1000' default are dependent USINGs anchored at a nonzero offset from the basing register. (See Figure 13 on page 21 for an example of a Using Map corresponding to this Using Heading.)

- The Location Counter column heading indicates whether a CSECT, DSECT, RSECT, or COM section is currently in effect. The heading is LOC, D-LOC, R-LOC, or C-LOC, respectively.
- For USING-resolution displays, both the second operand value and the registers specified as bases are shown for ordinary USINGs. The base-displacement resolution and first and

second operand addresses used are provided for dependent USINGs. Dependent USINGs display the actual offset of the anchor location.

- In the space between the statement number and the statement itself, a non-blank character indicates that the statement was derived from some other source than the primary input stream, or was processed in a special way:

  + The statement was generated by a macro, or is the result of conditional assembly substitution.

  – The statement was taken from the input stream and assigned to a conditional assembly variable by an AREAD instruction.

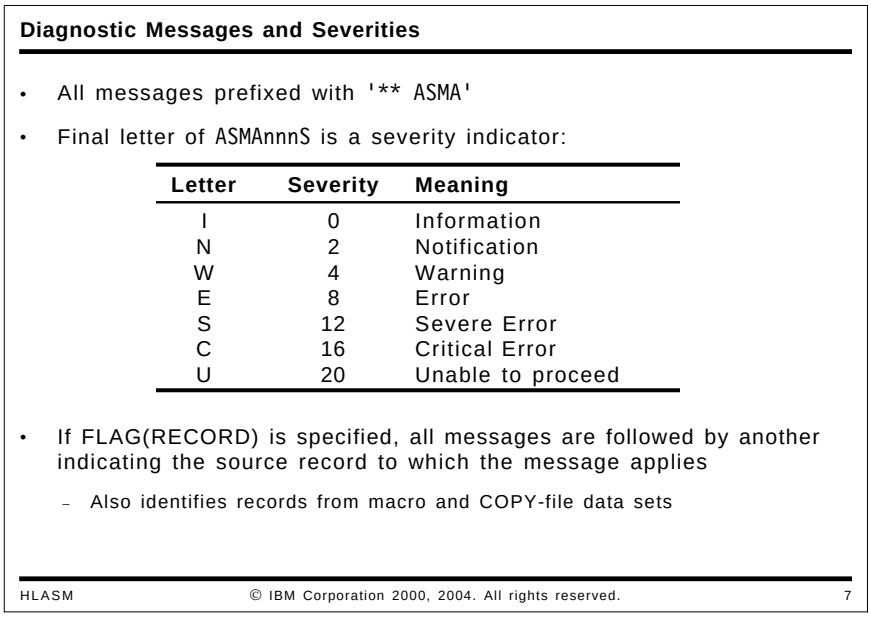
  > The statement was introduced into the input stream by an AINSERT instruction.

  = The statement was introduced into the input stream by a COPY instruction.

Things Worth Checking

- The “LOC” heading of each page's location counter designates the type of section currently being assembled; verify that it's what you intend.
- Verify that the nominated entry point on the END instruction refers to the desired execution start address, and that it lies within the bounds of the section to which it belongs.

Diagnostic Messages and Severities

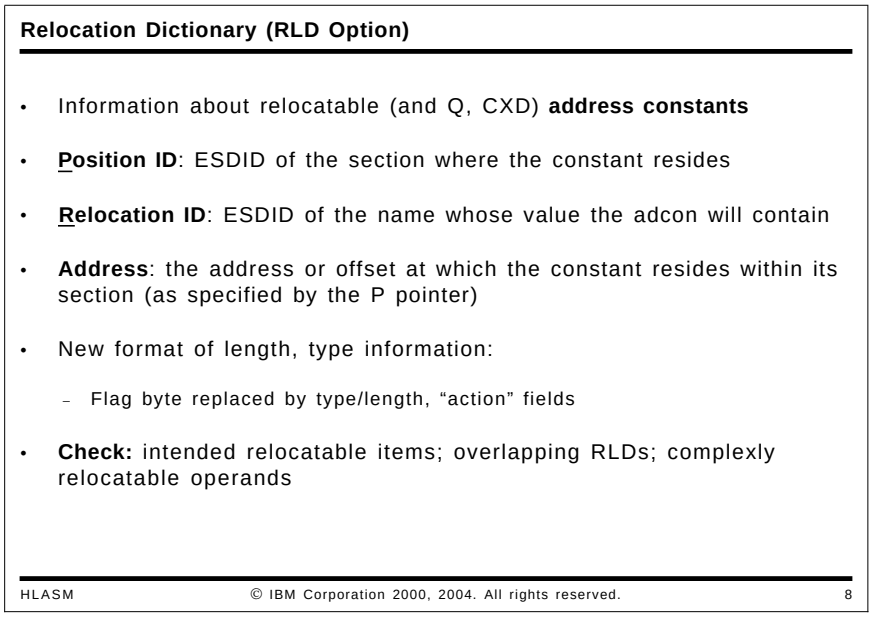


All messages issued by High Level Assembler have the form ** ASMAnnnS, where nnn is a three-digit message number, and S is a severity indicator, as shown in slide 7.

Some messages may not appear in the listing if portions have been suppressed by PRINT OFF instructions (see “TERM Option” on page 26) or if the diagnostics are subject to option control and have been disabled (see “FLAG(severity) Option” on page 35).

If the FLAG(RECORD) option is active, each diagnostic message is followed by an ASMA435I message indicating which source or library file and relative record number is associated with the message.

Relocation Dictionary (RLD Option)



The sample program in Figure 6 illustrates four types of address constants requiring relocation following assembly. The four adcon types are V, A, CXD, and Q. (Types A and V are addresses, type Q is an offset, and a CXD constant is a length.)

```
ASect  Start 0              ◄── Control section
       Extrn EX             ◄── External symbol
       Entry NTRY           ◄── Entry declaration
       DC    V(EX)          ◄── V-type adcon
NTRY   DC    A(EX-ASect)    ◄── A-type adcon with two operands
       CXD                  ◄── Cumulative external dummy length
EXDUM  DXD   F              ◄── External dummy section
       DC    QL2(EXDUM)     ◄── Q-type adcon
       END
```

Figure 6. Sample Program with Address Constant Types

The ESD listing from the assembly is shown in Figure 7 (the NOGOFF option was specified when the source file was assembled):

```
Symbol   Type   Id     Address  Length     LD ID
ASECT     SD 00000001 00000000 00000010
EX        ER 00000002
NTRY      LD          00000004           00000001
EXDUM     XD 00000003 00000003 00000004
```

Figure 7. Sample Program with Address Constants: ESD Listing

The ESD listing shows the ESD IDs assigned to the symbols, as described in “External Symbol Dictionary (ESD Option)” on page 5.

When the RLD option is specified, HLASM provides a summary of all fields requiring relocation, as shown in Figure 8 on page 13.

```
 Pos.Id   Rel.Id   Address Type  Action
00000001 00000000 00000008  J 4    ST   ◄── Type J; Rel.Id = 0 for CXD
00000001 00000001 00000004  A 4     –   ◄── Type A; note address 004, subtraction
00000001 00000002 00000000  V 4    ST   ◄── Type V; stored
00000001 00000002 00000004  A 4     +   ◄── note address 004; added
00000001 00000003 0000000C  Q 2    ST   ◄── Type Q; length 2 bytes
```

Figure 8. Sample Program with Address Constants: RLD Listing

The four fields in the RLD listing are:

- Position ID: the ESD ID of the section in which the adcon resides.
- Relocation ID: the ESD ID of the item according to which the adcon field will be relocated (or “adjusted”); the term “relocation” is used even if the contents of the adcon is not a relocated address. Note that CXD items have Relocation ID zero.
- Address: the location within the section indicated by the Position ID where the adcon is located.
- Type: the two entries indicate the type of constant and its length.
- Action: at link/bind time, the result of relocating the adcon is either stored into the adcon's field (indicated by ST), or added to (+) or subtracted from (−) the text in the adcon's field.

In the example in Figure 6 on page 12, the constant named NTRY indicates that the difference of two external symbols should be placed in the adcon field; the corresponding RLD items appear in the RLD Listing in Figure 8 at address 00000004, with two different Relocation IDs.

Things Worth Checking

- Check that there are as many relocatable items as you expect.
- Check that overlapping RLDs (with the same Position ID and address) are really intended. Sometimes, two adcons are accidentally placed in the same location, causing unexpected double relocations at bind time, execution time, or both.

Symbol Cross-Reference (XREF Option)

Ordinary Symbol and Literal Cross-Reference (XREF Option)

- XREF has three sub-options:
  - XREF(FULL) for all symbols, referenced or not
  - XREF(SHORT) for referenced symbols only
  - XREF(UNREFS) lists unreferenced non-DSECT symbols
    - Ignored if XREF(FULL) is specified
- Displays information about each symbol:
  - Symbol, length attribute, value
  - Relocation ID, relocatability tags (especially “C”), symbol type, where defined
  - References, including tags indicating use:
    Branch, Drop, Modification, Using, eXecute
- See the example on slide 10
  - Symbol `Batch_Init` is a branch target (`B` tag)
  - Symbol `Err_Buff` modified (`M` tag); a USING base (`U` tag)
  - Symbol `Move_Msg` is executed (`X` tag)
  - Symbol `R1` appears in USING (`U` tag) and DROP (`D` tag) instructions
- **Check:** usage tags; relocation ID and type; attributes; duplicate literals

HLASM 9

When the XREF option is specified, HLASM produces several styles of symbol cross-reference information. Because large programs may contain many unreferenced symbols, a more readable listing is produced if you specify the XREF(SHORT) option.

Slide 10 shows an excerpt from a symbol cross-reference, illustrating the key features of that part of the listing.

Ordinary Symbol and Literal Cross-Reference Example

```
Symbol      Len  Value    Id    R Type Defn  References
  ...
BadEQU        1 000004 00000001 C   U   666                 ←— note C type
Batch_Init    2 00035C 00000001     H   493  399B  490B      ←— note B tag
Batch_Len     8 000018 FFFFFFFF A   U   772  497
Batch_Msg_1  39 0006A1 00000001     C   681  469   469   716
  ...
Err_Buff      1 000000 FFFFFFFD     J   787  127U  517M  789 ←— note U tag
Err_Msg     255 000000 FFFFFFFD     C   788  277M  397M  469M  476M  481M  486M  ...
ESD_Amt       2 00000A FFFFFFFC     Y   797  347
ESD_Data     48 000010 FFFFFFFC     C   800  348
ESD_ESD       3 000001 FFFFFFFC     C   795  342   383
ESD_Item      1 000000 FFFFFFFB     J   804  349U  831         ←— note U tag
  ...
Move_Msg      6 0004A6 00000001     I   645  641X              ←— note X tag
  ...
  ...                                                  ┌──── note U,D tags
  ...                                                  ↓   ↓
R1            1 000001 00000001 A   U    53  123U  128D  132M  133M  133   135   ...
                                             215D  248M  262   348M  349U  373M  ...
                                             460   560M  574M  578M  582M  586M  ...
  ... etc.
```

HLASM 10

The ordinary symbol and literal cross-reference displays the following items:

- The symbol, in the form it was first encountered and entered into the symbol table
- The length attribute of the symbol (in decimal)

- The value of the symbol (in hexadecimal)
- The relocation ID (relocatability attribute) of the symbol, which relates the symbol directly to its “owning” control section. (See slide 10 on page 14.)

  Relocation ID values are also found in the ESD listing, or in the DXREF listing (described on page 19) for symbols defined in DSECTs.
- A column headed by “R” provides information about the relocatability properties of each symbol. Absolute symbols are flagged with an A, complexly relocatable symbols are flagged with a C, and simply relocatable symbols (the most common case) are not flagged.
- The type attribute of the symbol. Note that this is the type attribute of the symbol's final definition, and may *not* be the attribute value used during conditional assembly! (This situation is discussed further at “Lookahead Mode” on page 69.)
- The statement number at which the symbol was defined.
- In the cross-reference of symbol uses, High Level Assembler provides indicator tags in several contexts:
  - In USING (U) and DROP (D) instructions
  - As targets of EXecute (X) instructions
  - Modification (M) tags for operand symbols naming fields whose contents may be modified by the action of the instruction
  - Branch target (B) tags for symbols used as operands of branch instructions.

  The modification (M) tags permit rapid determination of which symbolic usages are for read references, and which are for write references. This feature eliminates much of the tedium in hunting for the few instructions that might have changed the value of a variable or the contents of a named register. Examples are shown in slide 10 on page 14.

Things Worth Checking: The XREF contains a wealth of useful information, and is worth checking carefully when problems arise.

- Check that the tags indicate expected uses of the symbol. For example, a symbol naming a constant should not have an ″M″ tag, indicating an instruction that appears to have modified it.
- Check whether the relocation type (the R column) of any symbol is ″C″. While the assembler allows you to define complexly relocatable symbols, such uses are very rare, and are most often an error that can otherwise be difficult to find.
- Verify that the length and type attributes are as expected. Sometimes a symbol defining a field is defined with an EQU instruction and assigned a (default) length attribute 1, where a different length was intended.
- Sometimes a type attribute value will be assigned by a macro, using a value that does not print as a normal character (or print at all!). Verify that such symbols have the properties you expect, or that the symbols are of no direct interest to your program.
- Check for multiple appearances of literals; these may occur if there is more than one LTORG instruction in the program. Sometimes the number of literals can be reduced by careful placement of literal pools.

XREF(SHORT,UNREFS) Options and Unreferenced Symbols

Symbols defined in ordinary (non-dummy) control sections but not referenced elsewhere in the program may be selectively displayed by specifying the XREF(SHORT,UNREFS) option, without the necessity of displaying *all* unreferenced symbols. This display is normally much shorter than a FULL cross-reference, and can help in eliminating unneeded constants, storage areas, and "dead code" statements. An example is shown in the following figure, where the unreferenced symbol was defined at statement 418.

```
                    Unreferenced Symbols Defined in CSECTs
  Defn  Symbol

   418  Normal_Dump
```

Figure 9. Example of XREF(UNREFS) Listing

If XREF(FULL) is specified, High Level Assembler ignores the UNREFS suboption, and displays all symbols whether referenced or not.

Things Worth Checking: Programs can often be "cleaned up" by checking areas in which unreferenced symbols appear:

- Unreferenced symbols among statements can sometimes indicate segments of "dead" that is never referenced, and that can be removed, either by deleting them, or by adding conditional-assembly AGO statements to skip over the unused statements (this keeps the statements in the source file in case they ever need to be reactivated).
- Unreferenced constants and work areas can often be removed.

Unreferenced DSECTs

Sometimes DSECTs are declared (or copied into, or macro-generated) but none of its symbols are used. To reduce the "clutter" in the program (and to minimize chances for accidental misuse of symbols), you may want to remove unreferenced DSECTs. To check whether a DSECT can be removed:

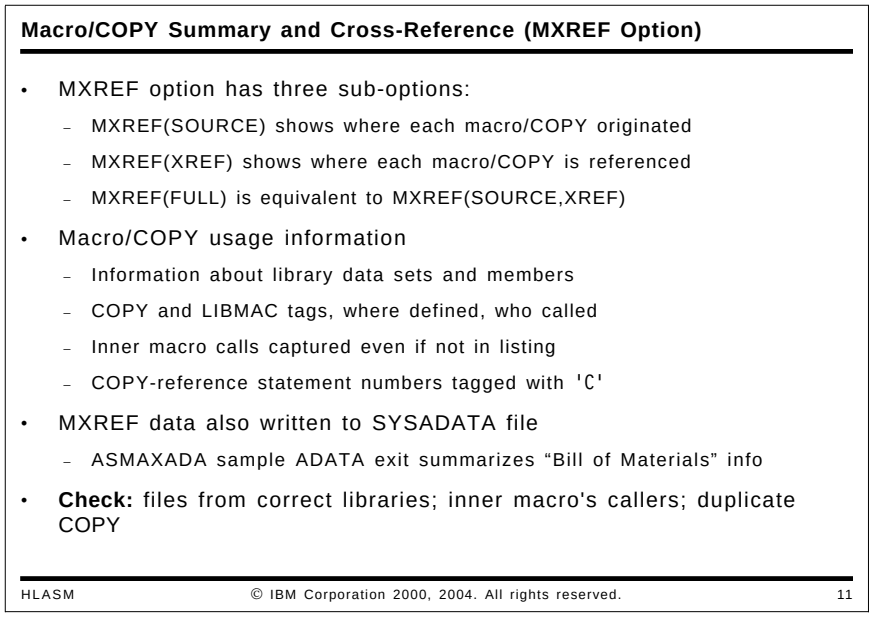
- Assemble the program with the XREF(FULL) and DXREF options.
- Locate the relocation ID of the DSECT in the DSECT XREF (see Figure 12 on page 19 for an example).
- Scan the symbol XREF for occurrences of symbols with that ID. If there are none, the DSECT is unreferenced. Otherwise,
- Check that all references to those symbols are made from statements within the DSECT. For example, in

```
DS1     DSECT ,
DSA     DS    A
DSB     DS    XL4
DSC     EQU   *-DSA
        ORG   DSA
DSD     DS    CL8
```

  the symbol DSA is referenced twice by symbols belonging to the DSECT.

- If the only references are made by statements in the DSECT, it can be considered unreferenced.

Macro-COPY Summary and Cross-Reference (MXREF Option)



The MXREF option specifies whether or not macro and COPY-member information should be included in the output listing. It has three suboptions: SOURCE, XREF, and FULL.

- If you specify the MXREF(SOURCE) option, the macro and COPY source summary provides the data set or file name and volume identification for every file from which a member was taken, as well as an indication of whether it was a primary source (SYSIN) file or a library (SYSLIB) file, and a concatenation number to distinguish among concatenated input or library files. A list of members used is provided for each library file.

  An example of the Source Summary is shown in the following figure.

```
                        Macro and Copy Code Source Summary

 Con Source                        Volume   Members
  L1 OSMACRO  MACLIB   S2          MNT190   FREEMAIN GETMAIN  RETURN   SAVE     TIME
  L4 ASMAMAC  MACLIB   S2          MNT190   ASMAXITP
```

Figure 10. Example of MXREF(SOURCE) output

  In this assembly, two macro libraries (OSMACRO and ASMAMAC) were referenced, even though four were specified in the SYSLIB concatenation. The members retrieved from each library are shown. The library files are identified by the information in the first column as “L1” and “L4”: these refer to library files (L) in concatenation sequence positions 1 and 4 respectively.

  If a macro definition was part of the SYSIN stream, the words “PRIMARY INPUT” would have appeared instead of a file name. (See the discussion of the allocated-files information in “Assembly Summary” on page 23.)

- If you specify the MXREF(XREF) option, the cross-reference provides, for each macro or COPY segment, its member name (if from a library), the concatenation number, which macro called it, the statement number at which it was defined, and the statement numbers at which references to it were made. This XREF information is provided even if inner calls do not appear on the listing (e.g. if PRINT NOMCALL has been specified).

An example of the XREF is shown in the following figure.

```
                    Macro and Copy Code Cross Reference

Macro     Con  Called By        Defn  References
ASMAXITP   L4  PRIMARY INPUT      -  833
FREEMAIN   L1  PRIMARY INPUT      -  564
GETMAIN    L1  PRIMARY INPUT      -  165
RETURN     L1  PRIMARY INPUT      -  294, 527, 567
SAVE       L1  PRIMARY INPUT      -  110
TIME       L1  PRIMARY INPUT      -  256
```

Figure 11. Example of MXREF(XREF) output

The same library concatenation-number indicators are used here as in the previous listing: “L4” for the fourth library in the concatenation, and “L1” for the first. This example shows that all the macros except RETURN are referenced only once.

The ″-″ in the “Defn” column means that the macro was not defined in the source stream. (Again, see the discussion of the allocated-files information in “Assembly Summary” on page 23.)

- If you specify the MXREF(FULL) option, both the source summary and the cross-reference are produced.

The MXREF output is a valuable resource for tracking macro and COPY-file usage. Note also that the MXREF data is written to the SYSADATA file when you specify the ADATA option; it is then easy to extract useful data from that file, either during the assembly (using an I/O exit) or later. A sample ADATA I/O exit (ASMAXADA) is provided with the High Level Assembler; it provides information about all members read from each library file.

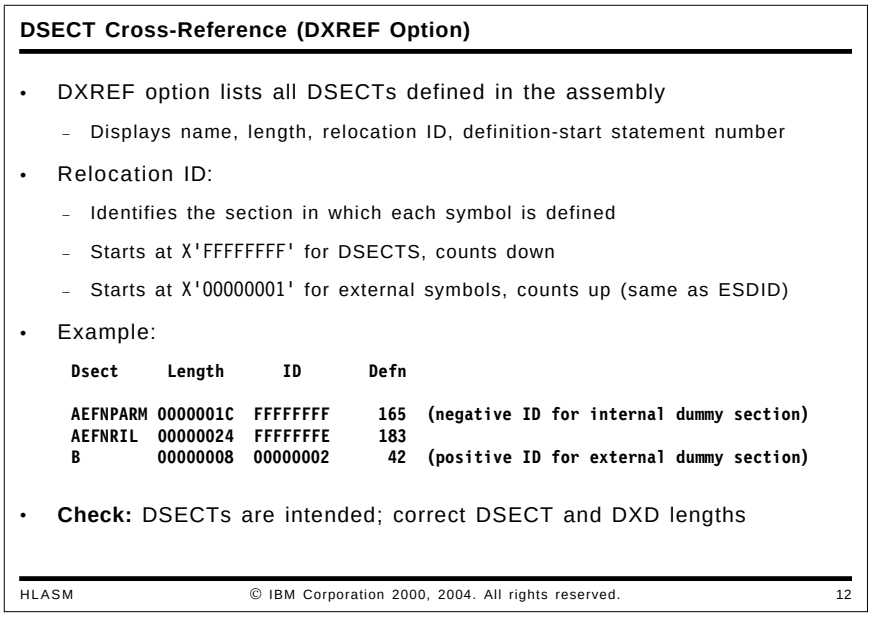
This information helps with version control, impact analysis, reassembly analysis, multiple library controls, and other code management tasks.

When you specify LIBMAC (via option or ACONTROL instruction) and MCALL (via PCONTROL(MCALL) option or PRINT MCALL instruction), additional valuable information may be available to you in the MXREF listing. This data is also helpful in possible error-tracing situations (see “Macros and Conditional Assembly” on page 57).

Things Worth Checking

- Verify that each macro and COPY file comes from the intended library. Multiple instances of a macro in different concatenated data sets can produce unexpected results.
- If a COPY file is referenced more than once, verify that the enclosing scope of the COPYed text will not cause duplicate definitions or a COPY loop (if conditional-assembly backward branches are used in the COPY text). An example showing how this problem can arise is discussed on page 70.
- Check that inner macros are called by the intended callers.

DSECT Cross-Reference (DXREF Option)



The DXREF option controls whether or not a DSECT cross-reference should be printed in the output listing. It provides for each internal *and* external dummy section:

- its name
- the section's length
- the relocation ID assigned to the section (this helps in identifying all symbols “belonging” to the section)
- the statement number where the definition begins.

Relocation IDs are assigned to external symbols starting at one and counting up (this is the ESD ID of the external symbol), and starting at X'FFFFFFFF' for internal sections and counting down. Both internal and external dummy sections will appear in the DSECT cross-reference listing, as shown in the following example.

```
Dsect     Length      ID       Defn

AEFNPARM 0000001C  FFFFFFFF     165  (negative ID for internal dummy section)
AEFNRIL  00000024  FFFFFFFE     183
B        00000008  00000002      42  (positive ID for external dummy section)
```

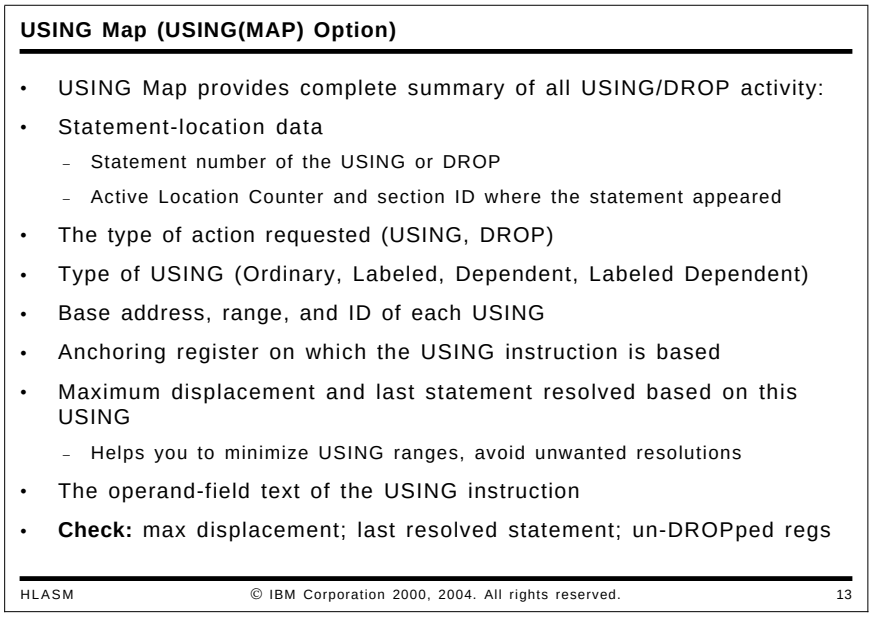
Figure 12. DSECT Cross-Reference with Internal and External Dummy Sections

The first two DSECTs are internal (their IDs begin with X'FFFF..') and the last (B) is an external dummy section. The presence of a positive ID value means that the external symbol is either a DXD name, or a DSECT name that has appeared as the operand of a Q-type address constant.

Things Worth Checking

- Verify that DSects (with negative relocation IDs) are indeed intended to be dummy sections, and that their length is as expected. (Sometimes one may forget to resume a CSect, making the preceding DSect contain extra or unwanted statements.)
- Check that DXD (external dummy section) names have the correct length. Their alignment is shown in the ESD listing, as noted on page 6.

USING Map (USING(MAP) Option)



The USING(MAP) option requests that HLASM create a USING Map in the listing. It summarizes all activity relating to the USING and DROP (and PUSH and POP USING) instructions in the program that control resolutions of symbolic addresses into base-displacement form. The information provided includes:

- The instruction number of the USING or DROP instruction.
- Under the “Location” heading, the “Count” and “Id” columns give the location counter value and the relocation ID active at the time the USING was issued.
- The “Action” column indicates whether the statement is USING or DROP.
- Under the “Using” heading, the “Type”, “Value”, “Range”, and “Id” columns describe the type of each USING, the base address of the USING, the range of resolution (which will be less than a multiple of X'1000' if a range limit is specified, or if a dependent USING is anchored at a nonzero offset), and the relocation ID of the base address. The “Reg” column shows the register(s) assigned as base(s), or involved in DROP instructions.
- The “Max Disp” column shows the maximum displacement calculated for the specified base register. Large values indicate that the object(s) addressed by that register may soon run out of addressability.
- The “Last Stmt” column indicates the last statement in which the register was used to resolve an address (that is, the effective end of the USING's domain). This can help you to place your DROP instructions to minimize the domain of a USING, in order to avoid accidental resolutions with incorrect base registers.
- The “Label and Using Text” displays the operands of the USING instruction, including the qualifier (which is moved from the name field).

```
                                    Using Map

 Stmt  ---Location--- Action --------------Using----------- Reg Max  Last Label and Using Text
       Count     Id          Type          Value  Range  Id      Disp Stmt
    2  000000  000001 USING  ORDINARY     000000 000CA6 0001  15 C94   805 (*,End),R15
    6  00001E  000001 USING  ORDINARY     000000 001000 FFFF   4 0A0       Record,R4
   47  00021C  000001 USING  LAB+DEPND   +000070 000F90 FFFD   4 007       HF.PhoneNo,HomeFone
   48  00021C  000001 USING  LAB+DEPND   +00007A 000F86 FFFD   4 007       WF.PhoneNo,WorkFone
   49  00021C  000001 USING  DEPENDENT   +000066 000F9A FFFE   4 006       ZipCode,Zip
   93  00000C  000001 DROP                                     4           HF
   93  00000C  000001 DROP                                     4           WF
  710  0005BC  000001 USING  LABELED      000000 001000 FFFF   5 07E       OLD.Record,R5
  711  0005BC  000001 USING  LABELED      000000 001000 FFFF   6 07E       NEW.Record,R6
  812  00000C  000001 DROP                                     4           R4
  812  00000C  000001 DROP                                    15           R15
```

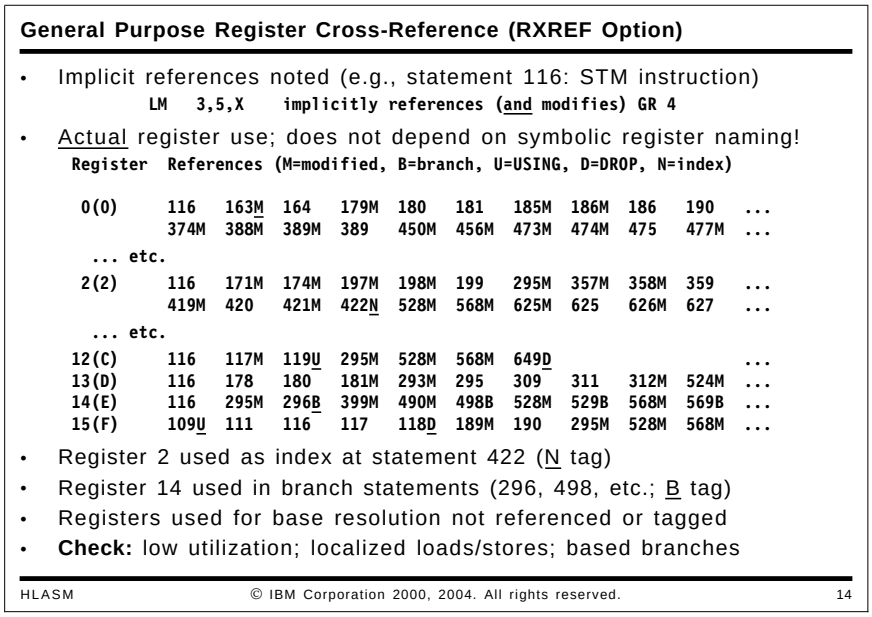
Figure 13. Example of a Using Map

The example has been edited slightly to fit a narrower field than is actually displayed on the listing (many of the fields are normally wider). An example of each of the four USING types is shown here, and a corresponding Using Heading is shown in Figure 5 on page 10.

Things Worth Checking

- If the maximum displacement value is zero, it can indicate either that there is only a reference to the base address of a target field (such as the beginning of a DSect), or that there are no addressing references to that register (in which case the USING can be eliminated and the register assigned to other uses).
- Check for maximum-displacement values approaching X'FFF'. This can indicate that the base register may be about to exceed its addressability range.
- Check the statement number of the last statement resolved for each USING, and put appropriate DROP instructions as closely as possible after that statement. This can help avoid later undesired resolutions against that base register.
- Check for un-DROPped registers. Their presence can also cause undesired resolutions.

General Purpose Register Cross-Reference (RXREF Option)



HLASM produces a cross-reference of all general purpose register usage when the RXREF option is specified. Slide 14 shows an excerpt from a typical general-register cross-reference; parts of the complete listing have been deleted for simplicity. All sixteen registers are shown (in a complete listing), with implicit and explicit references indicated; cases where a register is assigned as a base register by normal base-displacement resolution are not shown. The register cross-reference is *independent* of the symbol cross-reference: “absolute” register references (such as are generated by many system macros) are not shown in the symbol cross-reference, but *are* shown in the register cross-reference.

Implicit uses (such as registers involved in multiple-register instructions like LM, STM, D, M, SLDL, etc.) are also shown, even though the register number is not specified in the instruction itself.

The register cross-reference includes tags on statement numbers where the statement causes the register to be

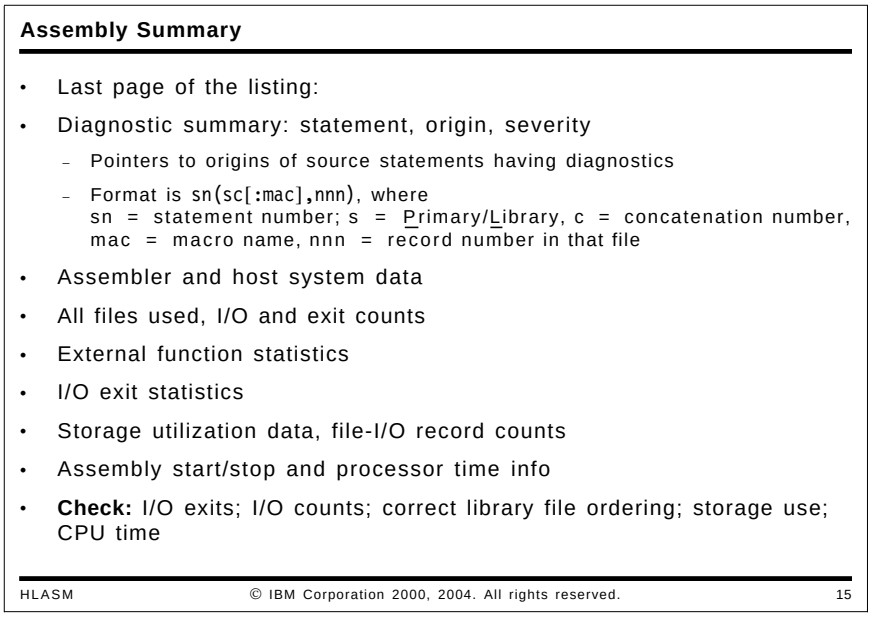
- modified ('M' tag)
- specified as a base register ('U' tag)
- used as an index register ('N' tag)
- used in a DROP instruction ('D' tag)
- used as a branch-target address ('B' tag)

This information can be very helpful, especially in finding instructions that modify a register in unexpected ways.

Things Worth Checking

- Registers with low levels of (or no) utilization can be identified to achieve better register-usage balance.
- Closely located store/load activity for a register can indicate a need for more registers usable as base registers. Modifying the code to utilize relative branch instructions can release some registers from providing code addressability for other uses.

Assembly Summary



The diagnostic and assembly summary page includes information about the entire assembly, including detailed information about the number of I/O actions, memory usage, number of diagnostic messages, host system, Assembler version, and other related data.

- All flagged statements are listed by statement number; if the FLAG(RECORD) option is active, each statement number is accompanied with information specifying the exact source record and the file from which it was read. For example, a small assembly with several intentional errors produced this portion of the diagnostic summary:

```
Statements Flagged

   1(P1,1), 4(L1:XXX,2), 6(P1,3), 7(L1:YYY,5), 8(L1:YYY,4), 9(P1,5)

    6 Statements Flagged in this Assembly     12 was Highest Severity Code
```

Figure 14. Example of Compact Diagnostic Summary

  In Figure 14, the list of flagged statements gives the statement number, and in parentheses the file identifier (described in Figure 15 on page 24) and the member (if any) from that file, and the record number within that file that caused the error. The final line gives the number of statements causing diagnostics, and the highest severity associated with the messages.

  Using this information, you can quickly locate and correct the files or library members.

- The name of the assembler, its release level, and current PTF level. The system environment is also shown.
- All input and output data set names, member names, and volume IDs are displayed, listed by ddname, including concatenations, as shown in the following figure.

```
Datasets Allocated for this Assembly

 Con DDname   Dataset Name                    Volume  Member
  P1 SYSIN    ASMAOXO2 ASSEMBLE D1           EHR192
  L1 SYSLIB   OSMACRO  MACLIB   S2           MNT190
  L2          ASMAFMAC MACLIB   A1           EHR191
  L3          ASMSMAC  MACLIB   L2           EHR195
  L4          ASMAMAC  MACLIB   S2           MNT190
     SYSLIN   ASMAOXO2 TEXT     D1           EHR192
     SYSPRINT ASMAOXO2 LISTING  D1           EHR192
```

Figure 15. Example of Allocated Data Sets/Files Summary Information

This example shows that a single file was allocated for all but SYSLIB, for which four concatenated libraries were allocated. The order of concatenation is indicated in the "Con" column, where "L" means "Library" and "P" means "Primary Input", and the following numeric gives the concatenation order.

- External function statistics include the name of each function, the number of calls of each type, and the number of messages issued and their maximum severity.

  The following example shows the results of a test program that passed valid and invalid data to two functions.

```
External Function Statistics
 ----Calls----  Message   Highest  Function
 SETAF   SETCF    Count  Severity  Name
   195       1       55      12    LOG2
     1      13        2      12    REVERSE
```

Figure 16. Example of Assembly Summary Function Statistics

LOG2 is an arithmetic function, REVERSE is a character function; each was called once in the wrong mode to verify that it produced a severity-12 error message.

- I/O exit statistics describe the exit type, exit name, number of calls to the exit, number of messages produced, and number of records added or deleted. The following figure shows the information produced from a sample exit.

```
Input/Output Exit Statistics
Exit Type  Name      Calls  ---Records---  Diagnostic
                            Added Deleted  Messages
OBJECT     OBJX         10      3       0        1
```

Figure 17. Example of Assembly I/O Exit Activity

- The central storage used for the assembly.

```
   5551K allocated to Buffer Pool
```

Figure 18. Example of Assembly Storage Utilization

- I/O statistics describe the number of reads and writes for each file used by the assembler.

```
  792 Primary Input Records Read       1283 Library Records Read
    0 ASMAOPT Records Read             1359 Primary Print Records Written
   14 Punch Records Written               0 ADATA Records Written
```

Figure 19. Example of Assembly I/O Activity

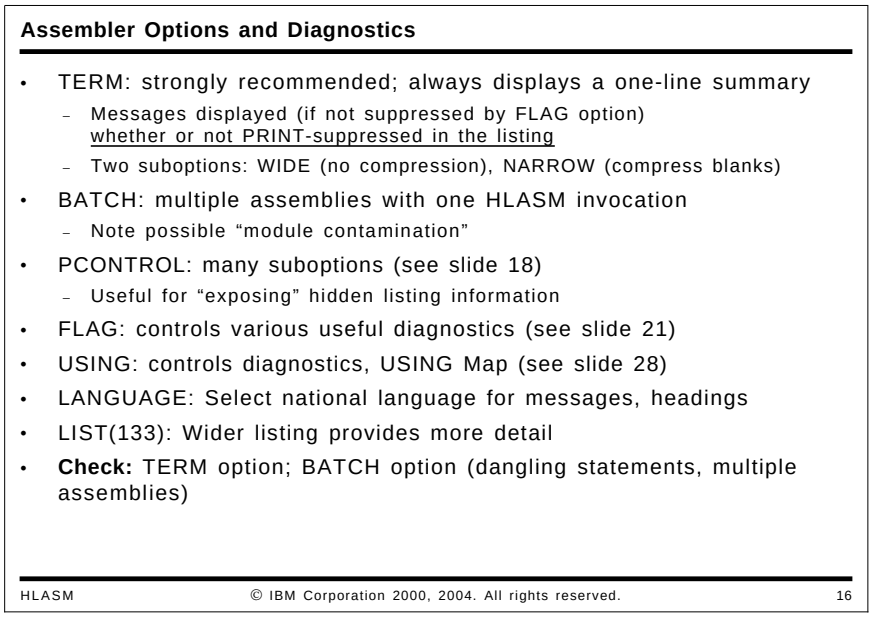
- The start and stop times of the assembly, an estimate of processor time required for the assembly, and the final return code of the assembly.

```
Assembly Start Time: 11.00.11 Stop Time: 11.00.12 Processor Time: 00.00.00.1101
Return Code 000
```

Figure 20. Example of Assembly Time Estimates

Things Worth Checking

- Verify that any I/O exits are ones you expect, and that the processing performed on input and output records looks reasonable. A listing exit can even suppress or modify information in the listing (including data about I/O exits!).
- Check that the files and data sets used for the assembly are as intended, and that library concatenations are in the correct order.
- I/O counts can help you determine that a reasonable number of records has been read or written.
- Information about storage use can be very helpful in setting storage sizes accurately, to avoid over-allocation and added costs and assembly-time overheads.
- Check that the estimate of processing time appears to be consistent with other assemblies of the size of the current program. Larger values could indicate that some macros could be improved, or that the assembler itself is processing certain items inefficiently.



You can specify several High Level Assembler options to help find problems that might not otherwise be evident. These options include TERM, BATCH, PCONTROL, FLAG, and USING.

TERM Option

The terminal output display (on SYSTERM) provides a useful summary of an assembly's behavior, showing all diagnostic messages (if any). The message display is *not* affected by PRINT or other statements that may prevent them from being seen on the listing, but may be suppressed by the FLAG(severity) option if the severity of the message is lower than the FLAG value.

TERM supports two suboptions (WIDE and NARROW). The blank-compressed NARROW layout enhances readability and allows the output to fit easily on an 80-character-wide display without wraparound. The WIDE format does not compress blanks, and may be more appropriate for screens that can display the full listing line length.

- A single-line summary is given for a successful assembly. This summary line reduces the amount of clutter on your terminal when assembling a “batch” of modules.
- The Deck-ID (if any; from a TITLE instruction) is included in the summary message. This information helps you monitor the progress of a batch of assemblies and associate diagnostic messages with the proper portion of the input file.

Suppose we assemble the small program shown in the following figure; the SPLAT mnemonic is intentionally unknown, and will generate an error message.

```
        PRINT OFF
        SPLAT
        PRINT ON
        SPLAT
        END
```

Figure 21. Source Program With Intentional Errors

When this program is assembled, the PRINT OFF instruction will cause the first SPLAT instruction *and its associated diagnostic* to be suppressed, as shown in the following extract from the listing file:

```
                                        2      PRINT OFF
                                        4      PRINT ON
                                        5      SPLAT
** ASMA057E Undefined operation code - SPLAT
                                        6      END
```

Figure 22. Source Program With Errors (Some Not Visible)

In Figure 22, the ASMA057E error message associated with statement 3 has been suppressed by the preceding PRINT OFF instruction.

If the TERM option is specified, then all error messages are visible, as shown in the following extract of the terminal display for this assembly:

```
                                        3      SPLAT
ASMA057E Undefined operation code - SPLAT
                                        5      SPLAT
ASMA057E Undefined operation code - SPLAT
Assembler Done      2 Statements Flagged /   8 was Highest Severity Code
```

Figure 23. Source Program With Errors (All Visible)

Things Worth Checking: Specifying the TERM option is recommended, especially during program development.

BATCH Option

The BATCH option allows you to complete multiple assemblies with a single invocation of the assembler. However, programs assembled with the BATCH option sometimes produce unexpected assembly-time errors, multiple assemblies, or even bind-time errors.

When the NOBATCH option is specified, the assembler stops reading the input file when it has processed the first END instruction, whether or not that END record is the last record in the input file. Thus, any trailing records are ignored.

When the BATCH option is specified, the assembler completes each assembly when its END instruction is encountered, and then continues to read the input file for further source modules. If the records following an END instruction do not form a valid source program, unexpected diagnostics may be produced; and if these following records *do* form a valid source program, their presence in the object file may cause undesirable behavior at link/bind time.

Extra Statements

Sometimes the END instruction of an assembly is followed by extra records, whose presence is apparent only when the BATCH option is specified.

Suppose we create an input file with a complete assembly plus some additional “dangling” statements, as illustrated in the following figure:

```
A    CSect
Con  DC    F'1'
     END
*    Define a constant
```

Figure 24. Source File With Dangling Statement

This program was assembled with various combinations of the BATCH and TERM options:

- NOBATCH and NOTERM: no errors

```
000000                00000 00004     1 A    CSect
000000 00000001                       2 Con  DC    F'1'
                                      3      End
```

Figure 25. Source File With Dangling Statement: NOBATCH, NOTERM (Listing)

The assembly appears to be error-free, as expected.

- NOBATCH and TERM: no errors, and a terminal message

```
Assembler Done No Statements Flagged
```

Figure 26. Source File With Dangling Statement: NOBATCH, TERM (Terminal Display)

Again, the assembly appears to be problem-free.

- BATCH and NOTERM: a second assembly with diagnostics:

```
000000                00000 00004     1 A    CSect
000000 00000001                       2 Con  DC    F'1'
                                      3      End
--- many lines later ---
                                      1 * Define a constant
** ASMA140W END record missing
```

Figure 27. Source File With Dangling Statement: BATCH, NOTERM (Listing)

The assembly now completes with a nonzero return code (the severity depends on what kinds of statements follow the first END statement). Thus, a program that “always” assembled successfully in the past may appear to have been contaminated with new, unsuspected errors. This can be particularly annoying if the first assembly is very large, because the small second assembly may not be easy to find at the end of the main assembly.

- BATCH and TERM: a second assembly with diagnostics:

```
000000                00000 00004     1 A    CSect
000000 00000001                       2 Con  DC    F'1'
                                      3      End
 --- many lines later ---
                                      1 * Define a constant
** ASMA140W END record missing
```

Figure 28. Source File With Dangling Statement: BATCH, TERM (Listing)

The terminal display shows clearly that more than one assembly was being processed:

```
Assembler Done No Statements Flagged
ASMA140W END record missing
Assembler Done      1 Statement  Flagged /   4 was Highest Severity Code
```

Figure 29. Source File With Dangling Statement: BATCH, TERM (Terminal Display)

Batch Assemblies and Private Code

The presence of Private Code (PC) sections can “contaminate” a bound module (see also the discussion of Private Code on page 9). To show how this can happen, suppose you assemble a program with an added fragment, specifying the BATCH option:

This example of a source file:

```
A    CSect
A    AMode 31
A    RMode 31
              ...and lots of other stuff
     End
R1   Equ   1
     End
```

when assembled produces two object files with ESDs:

```
Symbol   Type   Id     Address  Length     LD ID  Flags Alias-of
A         SD 00000001 00000000 00000AF8             06

Symbol   Type   Id     Address  Length     LD ID  Flags Alias-of
          PC 00000001 00000000 00000000             00
```

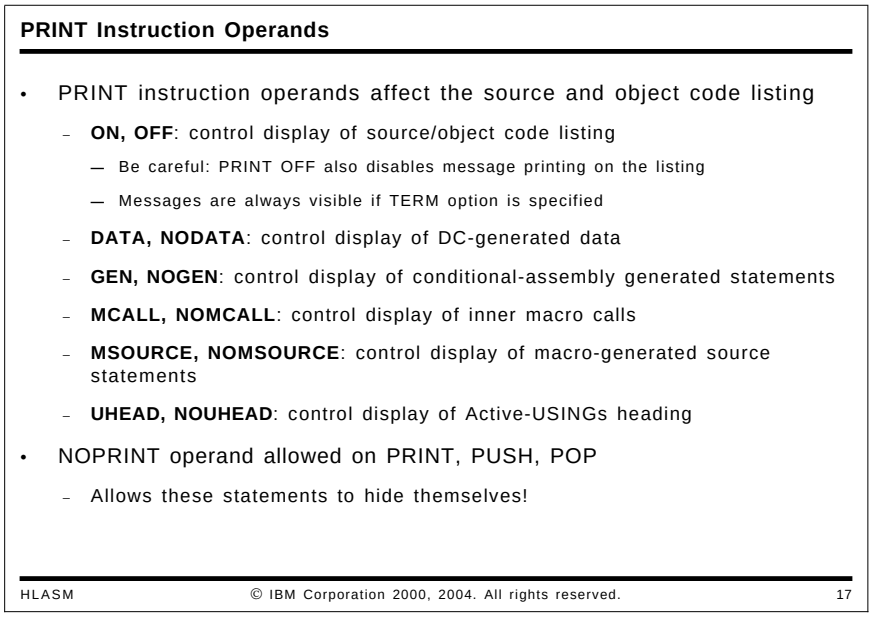
Figure 30. ESD from Source File With Extra Assembly

When this program is linked, the added Private Code (PC) section will very probably force the linked module's AMODE/RMODE to the lowest values (24/24), rather than the intended 31/31.

Things Worth Checking

- If unexpected errors occur in an otherwise “clean” assembly, check for the presence of the BATCH option and more than one assembly listing.
- If bind-time addressing or residence modes are not as expected, check for extra code following the first END instruction that was recognized due to the BATCH option. (Note that Private Code can also cause this condition, as discussed on page 9.)
- Message ASMA140W may indicate the presence of “dangling” statements, possibly following the only intended END instruction.

PRINT Instructions and the PCONTROL Option



The PRINT instruction supports six pairs of complementary operands, each pair enabling or disabling the printing of selected portions of the source and object code listing. Thus, parts of the listing can be made visible or invisible, as desired. However, useful information may have been hidden from view because parts of the listing may have been suppressed through use of the NOxxx or OFF operands.

The operands of the PRINT instruction are:

ON, OFF
These operands enable and disable the display of all subsequent lines in the source and object code listing, including diagnostic messages.

Specifying the TERM option will let you see all diagnostic messages not suppressed by the FLAG(n) option, even though PRINT settings may prevent their appearance in the listing. The examples at “TERM Option” on page 26 show how this works.

DATA, NODATA
If NODATA is specified, the listing will show only the first line of generated data from DC instructions (usually, at most 8 bytes). If DATA is specified, all generated data will be displayed.

GEN, NOGEN
If the GEN operand is specified, all statements generated by conditional assembly or macros, except for inner macro calls, are displayed with a + character to the left of the statement. The NOGEN operand suppresses the display of these generated statements.

When PRINT NOGEN is in effect, High Level Assembler displays the location counter value in effect for the first macro-generated code on the same line as the source statement. This makes debugging simpler for programs containing macro calls.

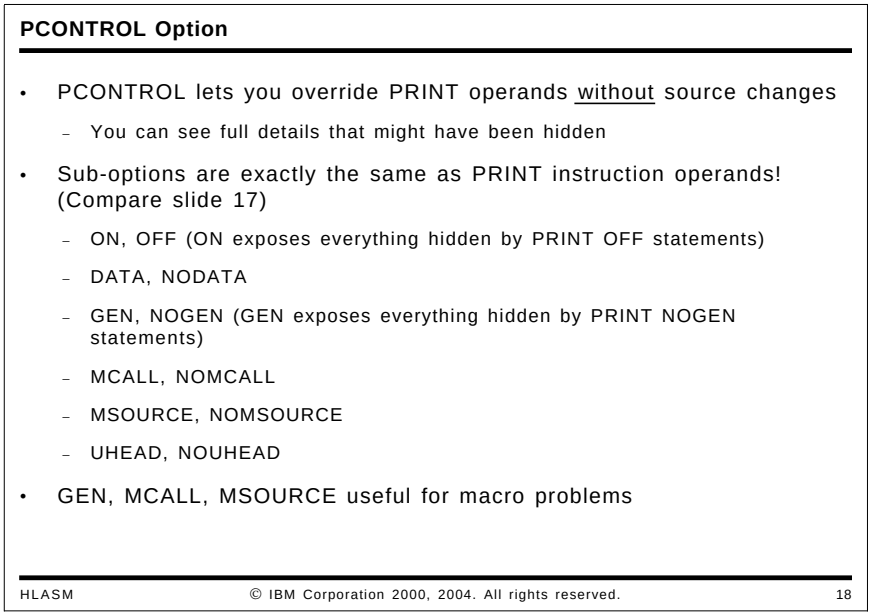
MCALL, NOMCALL
The MCALL operand controls the printing of inner macro calls. When an outer-level (or “top-level”) macro is called, the assembler displays that call on the listing (if other controls do not prevent its appearance). However, inner macro calls are not normally shown. Specifying the MCALL operand of a PRINT instruction will cause subsequent inner calls to be displayed. This helps you to debug complex nested macro interactions. Additional information is shown in the Macro Summary and XREF, described on page 17.

Macro calls displayed by the MCALL facility are not affected by the COMPAT(MACROCASE) option, discussed on page 61.

MSOURCE, NOMSOURCE
The NOMSOURCE operand suppresses the display of subsequent macro-generated source statements while still showing the generated object code. See "PCONTROL(MSOURCE) Option" on page 33 for further details.

UHEAD, NOUHEAD
The UHEAD operand controls the printing of the USING heading at the top of each page. Specifying NOUHEAD suppresses the USING heading on subsequent pages of the listing.

PCONTROL Option



The PCONTROL option can be used to "globally" override all occurrences of the internal settings of selected listing control (PRINT) instructions appearing anywhere in the source program: OFF and ON, [NO]DATA, [NO]UHEAD, [NO]GEN, [NO]MCALL, and [NO]MSOURCE. With this option you can force the display of code that would otherwise be invisible or obscured in normal listings, without having to modify *any* element of the source code itself.

PCONTROL(ON) Option

Long sequences of statements that appear in every assembly may become so familiar that they need not be present in the listing, so they are sometimes suppressed by PRINT OFF instructions. Later changes to the program may cause errors elsewhere in the program; if these are in PRINT-OFF regions, the reasons for the error are hard to find. Rather than modify PRINT instructions in *every* block of statements in such regions, simply reassemble with the PCONTROL(ON) option, and all statements in such PRINT-OFF regions will be displayed. (The TERM option is also very helpful in these situations.)

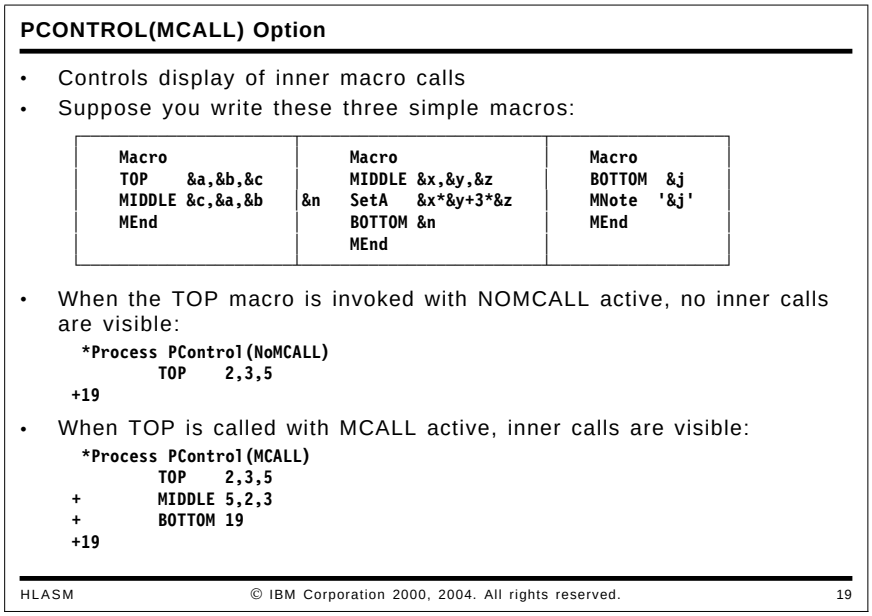
PCONTROL(DATA) Option

Most DC instructions generate small enough strings of data bytes that all “interesting” bytes are displayed. If, however, you want to verify that correct data has been generated *without* having to insert PRINT DATA instructions in the source program, just reassemble with the PCONTROL(DATA) option.

PCONTROL(GEN) Option

The PCONTROL(GEN) option can help find problems with macros and data declarations by causing generated statements from macros or open-code conditional assembly to appear in the listing. This provides additional detail that may help in locating problems with such statements and their generators.

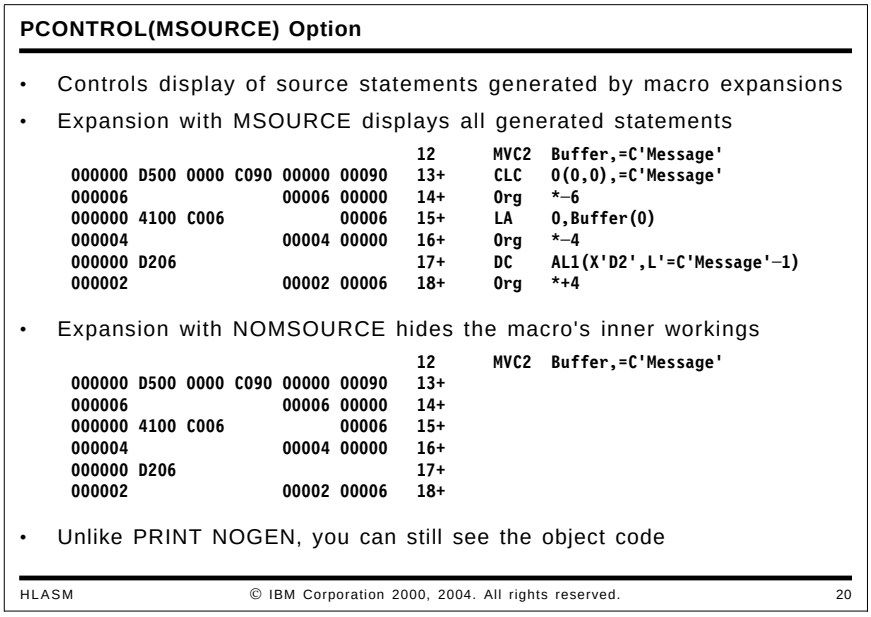
PCONTROL(MCALL) Option



The PCONTROL(MCALL) option specifies that *all* macro calls should be displayed, inner calls as well as top-level calls from open code. This can help in determining the flow of control *and* the values of arguments through levels of macro calls. The example in slide 19 shows how MCALL exposes the arguments of inner macro calls.

Macro calls are displayed in their original “as-created” case, independent of the COMPAT(MACROCASE) option.

PCONTROL(MSOURCE) Option



There are times when complicated schemes are used in macros to generate object code, and while seeing the generated machine language may be helpful, the generated statements themselves may be confusing. By specifying the PCONTROL(NOMSOURCE) option, the source lines may be suppressed without eliminating the generated machine language. An example is shown in slide 20.

Note that this option requires that PRINT GEN instructions or the PCONTROL(GEN) option be active; otherwise, the generated statements will not be printed in the listing.

PCONTROL(UHEAD) Option

Some types of USING-related problems can be analyzed more easily if all active USING instructions are known when a set of statements are being reviewed. Specifying the PCONTROL(UHEAD) option will cause High Level Assembler to display the standard “Active Usings” heading described on page 10.

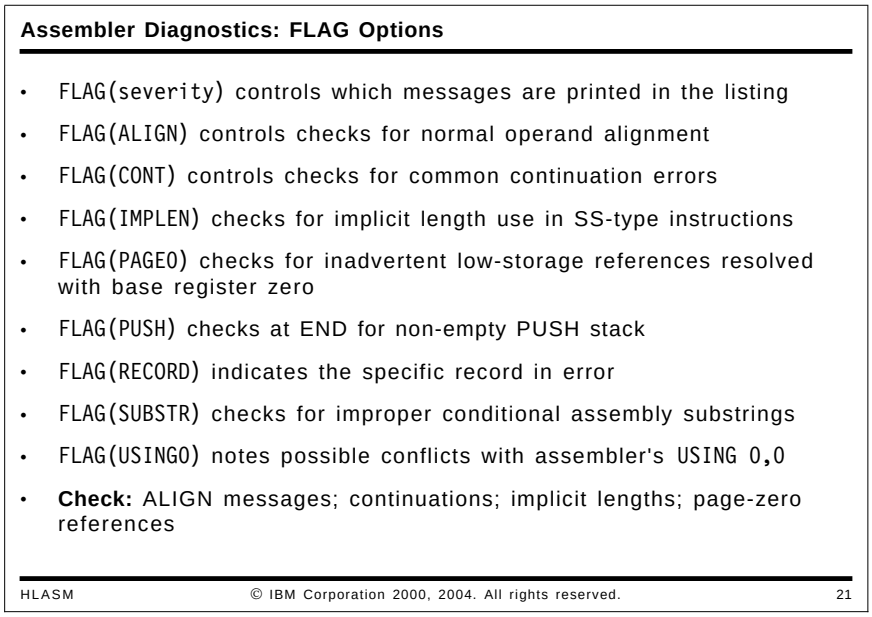
NOPRINT Operands on Certain Statements

The NOPRINT operand of the PRINT, PUSH, and POP instructions can help to eliminate distracting detail in the listing due to uninteresting generated statements and makes it easier to use High Level Assembler as a “cross-assembler” for other hardware architectures. However, the NOPRINT operand can hide the presence of the statement itself, which may make it harder to locate statements controlling the listing.

Note: Even if a program contains statements with NOPRINT operands, they are still placed in the SYSADATA file, which is described on page 67.

Things Worth Checking: Check that all needed source statements and/or generated data in the listing are visible. If parts are not visible, assemble the program with the appropriate PCONTROL options.

FLAG Option



You can request High Level Assembler diagnostics to help find problems that might not otherwise be evident. Many of these are controlled by the FLAG option (page 34) and USING option (page 42).

FLAG options include the following:

- FLAG(severity) controls which messages are printed in the listing, as described on page 35.
- FLAG(ALIGN) controls checking for potential problems with operand alignment, as described on page 35.
- FLAG(CONT) controls checks for possible errors in coding continuation statements. This option is discussed on page 36.
- FLAG(IMPLEN) controls checks for possibly unintentional omission of the length specification in SS-type instructions. Examples are shown on page 37.
- FLAG(PAGE0) controls checks for possible inadvertent references to addresses in the first 4K bytes of storage, as illustrated on page 38.
- FLAG(PUSH) checks at the end of the assembly for a non-empty PUSH stack, as described on page 39.
- FLAG(RECORD) causes HLASM to add an second message following a diagnostic to indicate the record number and source file from which the flagged statement was read. This option is discussed on page 39.
- FLAG(SUBSTR) controls checks for possible errors in coding conditional assembly substring notation. This is discussed on page 59.
- FLAG(USING0) notes possible conflicts with the assembler's implicit USING 0,0 instruction. The implications of such conflicts are discussed on page 40.

All the FLAG sub-options except RECORD are controllable dynamically with the ACONTROL instruction, discussed on page 65.

FLAG(severity) Option

The **severity** value in the FLAG option is a decimal value between 0 and 255. Terminal messages (controlled by the TERM option) and listing messages are treated as follows:

- Assembler messages with severity indicators less than **severity** will not appear on the terminal or in the listing.
- MNOTE messages with severity indicators less than **severity** will not appear on the terminal, but will be be printed in the listing without the prefixed text:

  ```
  ** ASMA254I *** MNOTE ***
  ```

Messages displayed on the terminal under control of the TERM option will be displayed if their severity is greater than or equal to **severity**.

Things Worth Checking: The FLAG(severity) option should always specify 0; otherwise, useful messages may be obscured.

FLAG(ALIGN) Option

FLAG(NOALIGN) causes High Level Assembler to suppress all warning messages when an alignment inconsistency is detected between a storage operand and a referencing instruction. When FLAG(ALIGN) is specified, the messages issued depend on the ALIGN option; the relationship between these two options is shown in the following table.

ALIGN,FLAG(ALIGN)	NOALIGN,FLAG(ALIGN)
• ASMA033I • ASMA212W • ASMA213W	• ASMA212W • ASMA213W

Table 1. Alignment Warning Messages and Their Dependence on ALIGN

The three messages are:

ASMA033I Issued where operand alignment is optional; for example, a LH instruction.

ASMA212W Issued when a branch-target address is odd.

ASMA213W Issued where operand alignment is required; for example, a CS instruction.

For example, suppose your program makes reference to an operand that is not aligned on a normal boundary:

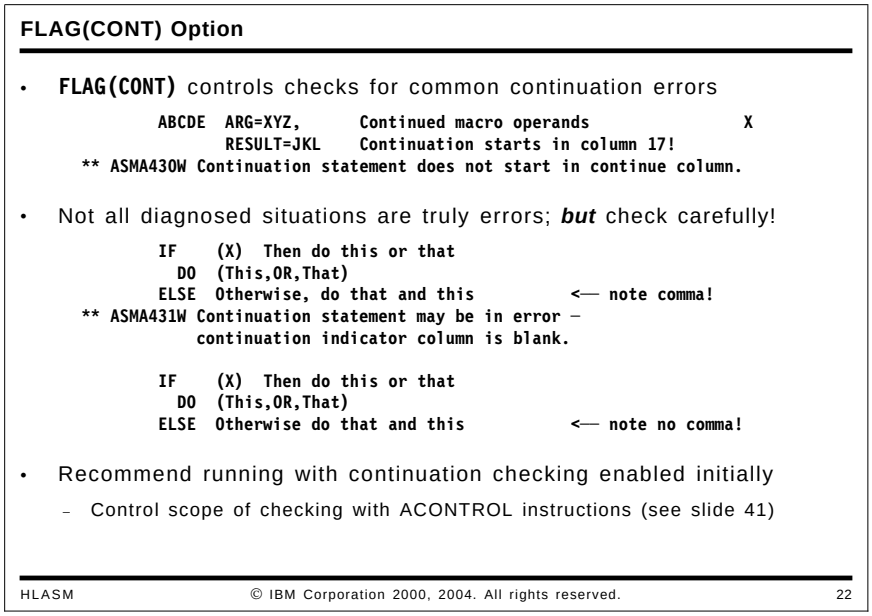
```
*PROCESS FLAG(ALIGN)
      L     0,X           X is not on a fullword boundary
** ASMA033I Storage alignment for X unfavorable

*PROCESS FLAG(NOALIGN)
      L     0,X           X still not on a fullword boundary; no message
```

Figure 31. Example of FLAG(ALIGN) Option

The informational message is suppressed by the FLAG(NOALIGN) option.

Things Worth Checking: Alignment errors may or may not be serious in terms of performance degradation, but they could imply misaligned references to fields in a shared data structure mapped with a DSECT.

FLAG(CONT) Option and Continuation Statement Checking



A common source of errors is the improper coding of continuation statements (the assembler language is unfortunately rather inflexible in its statement format!). Such errors can be extremely difficult to find and correct, so HLASM provides the FLAG(CONT) option to enable checking for continuation errors. The checks are neither exhaustive nor definitive, but they will detect a great many typical errors.

In this example of a call of the macro-instruction ABCDE,

```
        ABCDE  ARG=XYZ,      Continued macro operands               X
               RESULT=JKL    Continuation starts in column 17!
** ASMA430W Continuation statement does not start in continue column.
```

the continuation line begins in column 17, rather than in column 16 as intended. This would normally result in the second operand (RESULT=JKL) being ignored, which is probably not what was desired.

Occasionally this continuation checking will flag statements that are in fact correct or harmless. For example, suppose you are using some structured-programming macros in which the ELSE macro has no operands; a typical statement might look like this:

```
        ELSE         Take the other action
```

The remark **"**Take**"** appears to the continuation checking as a normal positional operand, so no diagnostic appears. However, if you had written

```
        ELSE         No, take the other action
```

Then the presence of the comma after the word “No” would make it appear to be the first operand in a list. The absence of a continuation indicator is (mis-)understood to indicate a missing operand, and the assembler will flag the statement with an ASMA431W warning message.

Continuation checking can be disabled by specifying the FLAG(NOCONT) option.

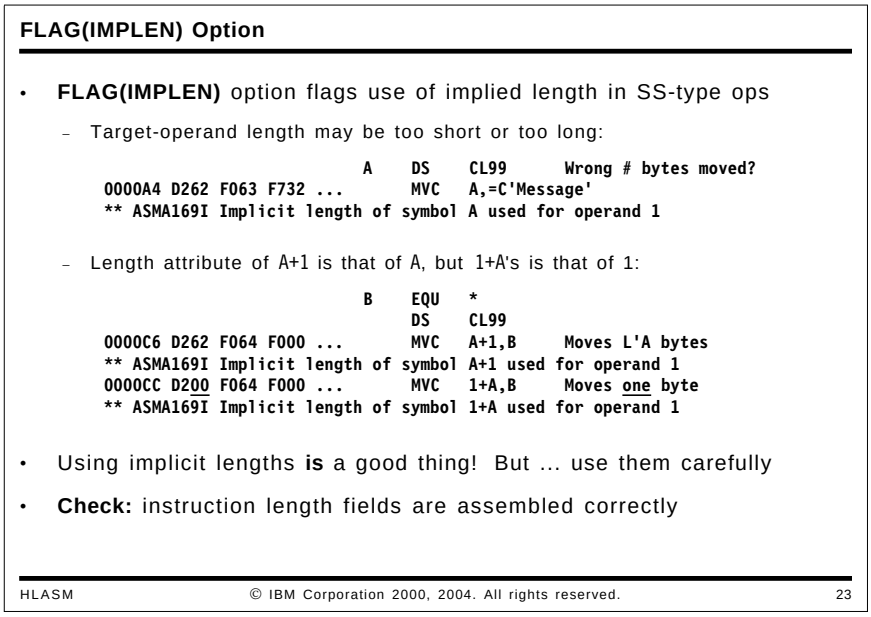
FLAG(CONT) helps you to find one of the most insidious types of Assembler Language errors: when statements are continued, misplacement of a single character can cause portions of statements to be ignored. Specifying this option causes High Level Assembler to

flag unusual or suspicious but difficult-to-find errors in specifying continued and continuation statements:

- An operand on a continued record ends with a comma, and a continuation statement is present, but the continuing statement does not begin in the “continue” column (usually, 16).
- A list of operands ends with a comma, but the continuation column (usually, 72) is blank.
- The continuing statement starts in the continue column, but there is no comma present following the operands on the previous continued record.
- The continued record is full, but the continuation record does not start in the continue column.

Things Worth Checking: Continuation checking can be *very* valuable; its use is strongly recommended as a regular diagnostic. You can use the ACONTROL instruction (described on page 65) with FLAG(NOCONT) and FLAG(CONT) operands to localize checking around statements known to be correct.

FLAG(IMPLEN) Option and Length Specifications



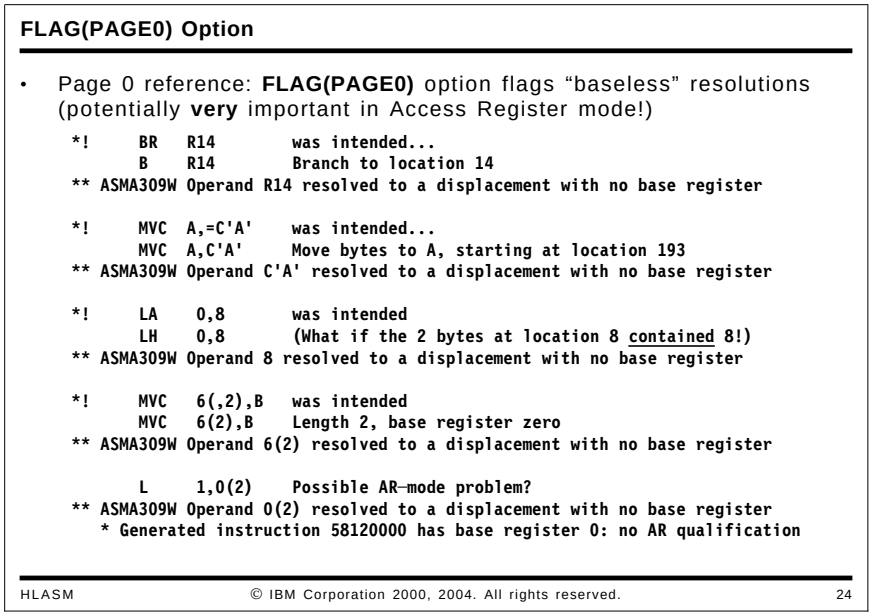
Occasionally an SS-type instruction requiring a length field will resolve the length implicitly, by using the length attribute of the the appropriate operand. This may or may not be intended:

```
                          A    DS    CL99
0000A4 D262 F063 F732 ...       MVC   A,=C'Message'   Wrong number of bytes moved?
** ASMA169I Implicit length of symbol A used for operand 1
0000C6 D262 F064 F000 ...       MVC   A+1,B           Moves L'A bytes
** ASMA169I Implicit length of symbol A+1 used for operand 1
0000CC D200 F064 F000 ...       MVC   1+A,B           Moves one byte
** ASMA169I Implicit length of symbol 1+A used for operand 1
```

If you specify FLAG(IMPLEN) as an option or as an operand of an ACONTROL instruction, HLASM will issue an informational (severity zero) message when implicit lengths are used.

Things Worth Checking: While such usage is common (and almost always correct), having the assembler check these uses is a good way to ensure that an error has not been overlooked.

FLAG(PAGE0) Option and Unintended Low-Storage References



Infrequently a programmer will write an instruction operand that assembles without diagnostics, but the operand address is resolved with respect to register zero. For example:

```
B    R14             (Intended  BR    R14)
CLC  X(1),C'0'       (Intended  CLC   X(1),=C'0')
LH   R2,X'0018'      (Intended  LH    R2,=X'0018')
```

and HLASM resolves the absolute expression in the operand with register zero as the base register.

In each case, a reference is made to the first 4K bytes of storage, sometimes called "Page 0". If you specify FLAG(PAGE0) as an option or as an operand on an ACONTROL instruction, HLASM will flag such uses with a warning message (except when the operand is used in an LA instruction).

Note that *based* references to page zero are valid, and are not flagged:

```
USING PSA,R0          Page-zero mapping DSect
L     R1,PSACVT       etc.
```

Similarly, *explicit* references to page zero are not flagged:

```
L     1,16(0,0)  Explicit reference is OK

L     1,16       Typical implicit reference
** ASMA309W Operand 16 resolved to a displacement with no base register
```

This warning can be extremely important for programs executing in Access-Register (AR) mode, because the index field of an instruction is not qualified by an Access Register. For example:

```
L     1,0(2)
```

is assembled as though it had been written

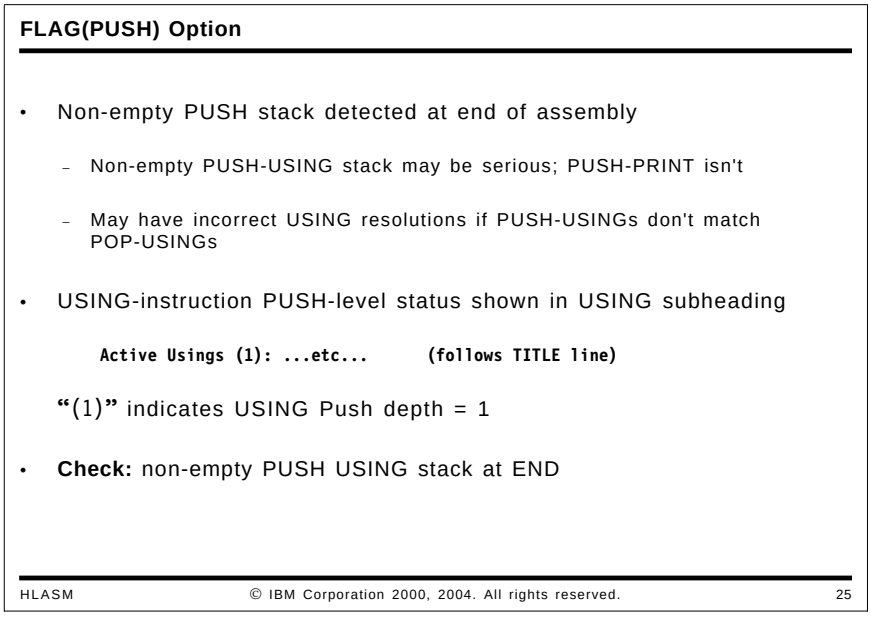
```
L     1,0(2,0)       Object code 5012 0000
```

This will use general register 2 for a “base” address when not in AR mode; but in AR mode, Access Register 2 will not be referenced! To obtain correct addressing in all modes, the statement should be written

```
L     1,0(0,2)       Object code 5810 2000
```

Things Worth Checking: This diagnostic is strongly recommended: it can catch errors that may otherwise be unnoticed for a long time. It is especially important for programs that may execute in AR mode.

FLAG(PUSH) Option and Non-Empty PUSH Stack



The PUSH instruction lets you save the status of PRINT, USING, and ACONTROL statements. At the end of the assembly, HLASM checks to see if the PUSH stack is empty; if not, it issues diagnostic ASMA138W. The depth of the USING stack is indicated in the “Active Usings” heading on each page. The diagnostic can be suppressed with the FLAG(NOPUSH) option.

Things Worth Checking: While a non-empty PUSH stack is rarely serious, it could indicate that a USING environment was suspended and not restored, which could mean that incorrect base-displacement resolutions were derived for statements that follow the point where a POP USING instruction should have appeared. Specifying this option can help detect such oversights.

FLAG(RECORD) Option

The FLAG(RECORD) option causes High Level Assembler to provide supplementary information (with each diagnostic message, and in the diagnostic summary) about the data set name and relative record number within that data set for the statement involved. This option can help with locating the specific original source statement requiring correction.

The following figure illustrates the additional ASMA435I message produced when FLAG(RECORD) is specified:

```
** ASMA062E Illegal operand format - T'V
** ASMA435I Record 26 in TATST    ASSEMBLE A1 on volume: EHR191

** ASMA137S Invalid character expression - 4)
** ASMA435I Record 35 in TATST    ASSEMBLE A1 on volume: EHR191
```

The ASMA435I messages show the name of the file and the number of the record in that file where the diagnostic was issued. This makes it easy to edit and correct the source file at the same time the listing is being scanned for diagnostic messages.

FLAG(USING0) Option: USINGs With Absolute Base Address

FLAG(USING0) Option

- Helps catch accidental use of absolute base address
- Examples of USINGs with absolute base addresses that overlap the assembler's implicit USING 0,0

```
                          USING 12,12
  ** ASMA306W USING range overlaps implicit USING 0,0

 4110 000A                LA    1,10
 4110 C008                LA    1,20
```

  - Note the different resolutions: one based on register 0, one on 12

```
                          USING -1000,12
    ** ASMA306W USING range overlaps implicit USING 0,0
 4110 C3DE                LA    1,-10
 4120 C3F2                LA    2,10

                          USING +1000,11
    ** ASMA306W USING range overlaps implicit USING 0,0
 4130 B3E9                LA    3,2001
 4145 B0C8                LA    4,1200(5)
```

- Message ASMA306W is controlled with the FLAG(USING0) option

HLASM 26

If the user specifies a USING instruction with an absolute base address whose range overlaps that of the assembler's “implicit USING 0,0”, HLASM will issue message ASMA306W.

As a simple example, suppose the following statements appear in a program:

```
                          USING 12,12
  ** ASMA306W USING range overlaps implicit USING 0,0

 4110 000A                LA    1,10
 4110 C008                LA    1,20
```

Note that the two values 10 and 20 are resolved quite differently: because a valid displacement cannot be calculated for the implied absolute address 10 from existing USINGs, the assembler uses its implicit USING 0,0 to resolve the value with base register zero. The absolute address 20 can be resolved with a *smaller* displacement (8) using base register 12, as indicated.

Further examples of absolute base address resolution follow.

If a USING instruction specifies an absolute base address whose range overlaps the range of the assembler's “implicit” USING 0,0 then unexpected resolutions might occur:

```
                          USING -1000,12
    ** ASMA306W USING range overlaps implicit USING 0,0
4110 C3DE                 LA    1,-10
4120 C3F2                 LA    2,10

                          USING +1000,11
    ** ASMA306W USING range overlaps implicit USING 0,0
4130 B3E9                 LA    3,2001
4145 B0C8                 LA    4,1200(5)
```

Additional USING diagnostics are described beginning on page 41.

Things Worth Checking: While absolute base values are rarely used, they can cause serious problems in code written to assume they will never be present. This diagnostic can help avoid errors that may otherwise be difficult to find.

USING Diagnostic Messages

USING Diagnostic Messages

- Message not controlled by an option:

  ASMA308W Repeated register in USING

- Messages controlled by the USING(WARN(nn)) option (see slide 28)

  ASMA300W, ASMA301W
  Nullification of one USING by another

  ASMA302W Base register 0 specified with nonzero base address

  ASMA303W Multiple valid resolutions

  ASMA304W Resolved displacement exceeds specified limit

- Message controlled by the FLAG(USING0) option (see slide 26):

  ASMA306W USING range overlaps implicit USING 0,0

- **Check:** examine all USING-related messages carefully

HLASM 27

Because USING errors can be difficult to find, and may have serious impacts on program execution, HLASM provides extensive checking for potential misuses.

Remember that the assembler uses the following resolution rules for base-displacement addressing:

1. The assembler searches the USING Table for entries with a relocatability attribute matching that of the implied address (which will almost always be simply relocatable, but may be absolute). (If the implied address is complexly relocatable, no match will be found.)
2. For all such matching entries, the assembler checks to see if a valid displacement can be derived. If so, it will select as a base register that register which yields the smallest valid displacement. If the smallest valid displacement exceeds the USING range (usually 4095 bytes), the assembler will indicate the amount by which the implied address was not “reachable”.
3. In the event that more than one register yields the same smallest displacement, the assembler will select as a base register the highest-numbered register.

4. If no resolution has been completed, and the implied address is absolute, attempt a resolution with register zero and base zero.

Five USING diagnostics are controlled by the USING(WARN) option, and one of them also depends on the USING(LIMIT) option (described on page 43). One USING diagnostic is controlled by the FLAG(USING0) option, as discussed previously on page 40.

One other USING diagnostic is not controlled by an option. Previous assemblers did not diagnose repeated uses of the same base register in one USING instruction; such (admittedly unusual) specifications could lead to unexpected resolutions, when all but the last instance of a register was ignored! Previously, the instruction

```
USING base,12,11,12,11
```

was treated as being equivalent to the four statements

```
USING base,12
USING base+4096,11
USING base+8192,12
USING base+12288,11
```

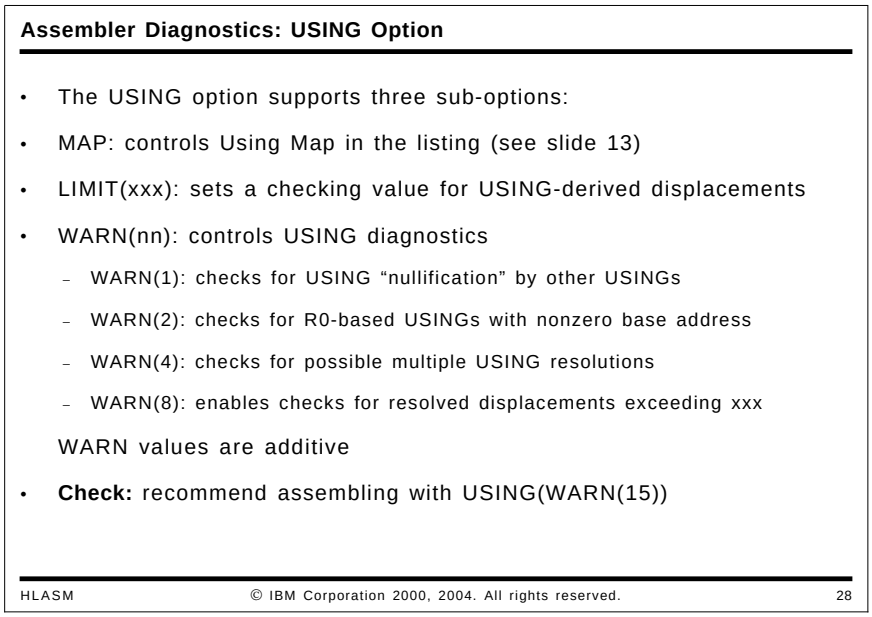
and therefore the first two “USINGs” were effectively ignored.

High Level Assembler diagnoses this situation:

```
                USING base,12,11,12,11
** ASMA308E Repeated register 12
** ASMA308E Repeated register 11
```

The statement is ignored; you could find that some or all of your program has no addressability.

USING Option



USING options to be described include the following:

- USING(MAP) controls the presence of the USING Map, as described on page 20.

- USING(LIMIT(xxx)) checks for displacements that exceed the specified limit (“xxx”) in addressability resolutions.
- USING(WARN(nn)) controls checking for errors associated with the USING instruction.

USING(LIMIT(xxx))

This option sets a value that the assembler will use to check against all calculated displacements. If the USING(WARN(8)) option is active and any displacement exceeds this value, a warning message will be issued. This is very useful in detecting portions of the program that may be in danger of running out of addressability when new statements are added.

USING(WARN(nn))

The WARN(nn) suboption of the USING option enables checking for several (often obscure) but common USING-instruction errors and oversights. Because USING instructions are the most important (and probably, the most confusing) of the Assembler Language's addressing facilities, these features help with specifying them and diagnosing possible misuses.

These sub-options are:

WARN(1) Warnings for two common (but extremely difficult to find) errors can be enabled. These involve cases where an ordinary USING instruction will cause either a previous or a subsequent USING instruction to be ignored by the assembler. The terminology used in the High Level Assembler's message is that one USING is “made inactive” (or “nullified”) by another.

WARN(2) The use of register zero as a base register with a nonzero absolute base is detected. This helps to isolate errors caused by forgetting that General Register 0 cannot be used as a base register.

WARN(4) Multiple valid resolutions can be flagged. In some programs, this condition represents an oversight on the part of the programmer, and typically is an error. In other cases, the structure of the program causes USING ranges to overlap. (An overlap of exactly one byte will not be flagged.)

This diagnostic may indicate a potential problem when in fact there is none. This condition can be controlled by specifying a USING range limit, as described in “USING Range Limits” on page 47. However, it should not be suppressed automatically: a detailed example of the importance of this diagnostic will be shown in “Fixing USING Problems with Multiple Resolutions (ASMA303W)” on page 45.

WARN(8) Warnings can be issued if the range of addressability of a base register exceeds a specified threshold. This is very helpful when a growing program is nearing the limits of addressability provided by its base registers, and you wish to be warned when the remaining addressability falls below a specified threshold. (The threshold value is specified in the LIMIT sub-option of the USING option.)

You may obtain combinations of these diagnostics by adding the WARN values: for example, WARN(3) specifies that the first two conditions should be flagged.

USING instructions with absolute base expressions and nonzero base registers whose range overlaps that of the assembler's “implicit” USING 0,0 are controlled by the FLAG(USING0) option, as described on page 40.

Things Worth Checking: It is recommended that all programs be assembled with WARN(15) as the default.

USING Diagnostics: Examples

Examples of USING Diagnostics

- Assembler options included USING(WARN(15),LIMIT(X'F98'))

```
                              1 START  CSECT
                     00000    2        USING *,10
                     00000    3        USING *,11         A later USING, but...
** ASMA301W Prior active USING on statement number 2
                     overridden by this USING

                     00000    4        USING *,9          Another later USING
** ASMA300W USING overridden by a prior active USING on statement number 3

                     00000    6        USING B,0
** ASMA302W USING specifies register 0 with a nonzero
                   absolute or relocatable base address

                     00FFA    8        USING *+4090,7
** ASMA303W Multiple address resolutions may result from this USING

000000 4120 BFA0     00FA0    10       LA    2,START+4000
  ** ASMA304W Displacement exceeds LIMIT value specified
                     00004    12 B     EQU   4
```

HLASM 29

The following small fragment of a program illustrates how High Level Assembler detects possible USING error conditions. The LIMIT sub-option was specified as LIMIT(X'F98').

```
                              1 START  CSECT
                     00000    2        USING *,10
                     00000    3        USING *,11
** ASMA301W Prior active USING on statement number 2 overridden by this USING

                     00000    4        USING *,9
** ASMA300W USING overridden by a prior active USING on statement number 3

                     00000    6        USING B,0
** ASMA302W USING specifies register 0 with a nonzero
                   absolute or relocatable base address

                     00FFA    8        USING *+4090,7
** ASMA303W Multiple address resolutions may result from this USING

000000 4120 BFA0  1  00FA0    10       LA    2,START+4000
  ASMA304W ** WARNING ** Displacement exceeds LIMIT value specified
                     00004    12 B     EQU   4
```

Figure 32. USING Diagnostics Example

This example shows each of the four diagnostic messages issued under the control of the WARN sub-option of the USING option.

1. The ASMA301W message is issued because the second USING instruction (statement 3) will cause the preceding USING to be made inactive. The second USING has the same base address but a higher-numbered register, so it will always be selected in preference to the first.

2. The ASMA300W message is issued because the third USING instruction (statement 4) will be ignored. *Even though it appears in the program later than the preceding USING instruction* (statement 3), it will be ineffective because it has both the same base address and a *lower-numbered* register than the preceding USING.

   This condition often occurs because it is easy to forget the assembler's complex address-resolution rules, and to assume that a “later” USING automatically supersedes an “earlier” USING.

3. The ASMA302W message occurs because a nonzero base address has been specified in a USING instruction designating base register zero.

4. The ASMA303W message indicates that the range of this USING instruction overlaps the range of some previous USING instruction.

   While this is not necessarily an error, it often represents an oversight on the part of the programmer, because it is unusual to provide more than one base register for a given part of a program. Thus, this message can help to locate inadvertent or improper USING specifications more easily. (See “Fixing USING Problems with Multiple Resolutions (ASMA303W)” for examples.)

5. The ASMA304W message indicates that the displacement calculated for the LA instruction at statement 10 (X'FA0', at key **1**) exceeded the LIMIT value specified.

   It is also worth noting that the base register specification digit X'B' indicates that register 11 was used for address resolution: only the second USING instruction (statement number 3) is truly “in effect!”

Fixing USING Problems with Multiple Resolutions (ASMA303W)

The ASMA303W warning message indicates that multiple base-displacement resolutions may be derived from two or more USING instructions in the program. Sometimes the warning is produced for a situation that is obviously safe; but there are times when a subtle problem may be highlighted by the warning. This discussion should help convince you not to disable this warning message “automatically”.

We will show three examples of coding that could produce this warning. The first is quite harmless, the second is intentional and/or unavoidable, while the third illustrates a potentially very dangerous situation.

Multiple USING Resolutions(1): Entry-Point USINGs

Overlapping USING-Range Warning: Simple Case

- Typical warning for overlapping USINGs in prolog/entry code:

```
        1 Enter    Start 0
        2          Using *,15
        3          STM   14,12,12(13)      Save registers
        4          LR    11,15             Set local base register in R12
        5          LR    12,11             Second base
        6          AH    12,HW4096         Add 4096 for second base value
        7          B     DoSaves           Skip over constant
        8 HW4096   DC    H'4096'           Constant
        9          Using Enter,11,12       Provide local addressability
           ** ASMA303W Multiple address resolutions may result from
                       this USING and the USING on statement number 2
       10          Drop  15                Drop R15
```

- First impulse: suppress the warning
  - May not be the best idea...
- Easy to fix: move the 'Drop 15' at statement 10 to precede the 'Using Enter,11,12' at statement 9

HLASM 30

Consider a typical entry-point coding sequence like the one shown in Figure 33. If the USING warnings are enabled, the assembler will flag the second USING instruction with a warning:

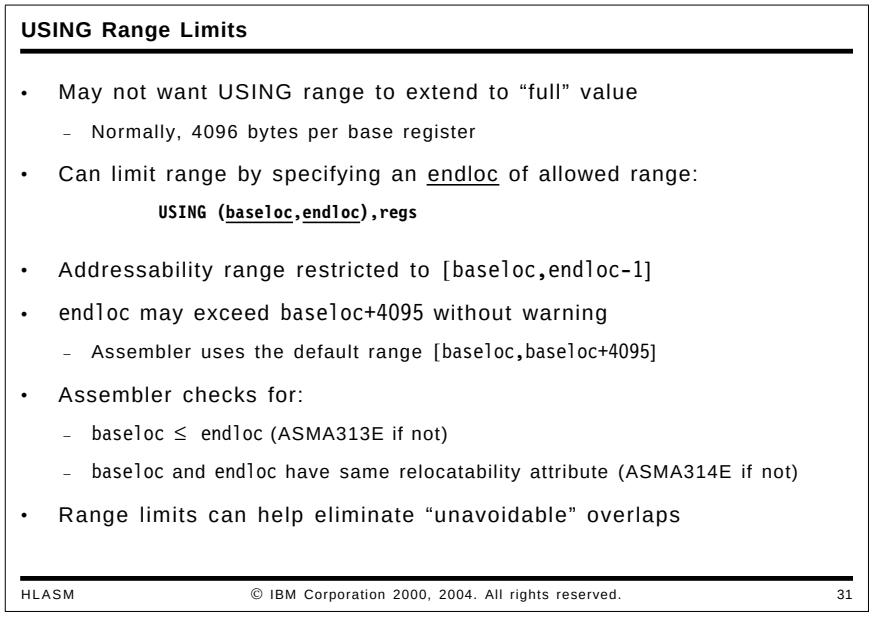
```
        1 Enter    Start 0
        2          Using *,15
        3          STM   14,12,12(13)      Save registers
        4          LR    11,15             Set local base register in R12
        5          LR    12,11             Second base
        6          AH    12,HW4096         Add 4096 for second base value
        7          B     DoSaves           Skip over constant
        8 HW4096   DC    H'4096'           Constant
        9          Using Enter,11,12       Provide local addressability
           ** ASMA303W Multiple address resolutions may result from
                       this USING and the USING on statement number 2
       10          Drop  15                Drop R15
```

Figure 33. Simple Multiple-Resolution USING Warning

In this case, the warning is not terribly helpful, because the DROP instruction on the following line removes the possibility of subsequent resolutions that might “ignore” R12.

This “problem” is easy to fix: simply move the DROP 15 instruction before the preceding USING instruction.

USING Range Limits



The second instance of USING range overlaps occurs when two segments of code or data are assigned their own base registers. Before showing how this might occur, we will discuss a feature of USING instructions that helps with this and other USING-range situations.

You might want to ensure that a given USING instruction is not utilized by the assembler to resolve implied addresses beyond a known limit. In previous assemblers, there was no way to control this. High Level Assembler provides a method allowing you to explicitly specify the precise range of validity for a USING instruction, by providing a *pair* of values as the first operand of the USING instruction:

```
USING (baseloc,endloc),regs
```

The endloc address is the first location *not* addressed by this USING; thus the addressable locations in the USING's range lie between baseloc and endloc-1. If the endloc value exceeds baseloc+4095, the range is the normal 4096 bytes starting at baseloc. In effect, the number of bytes addressed is

```
bytes addressed = min(4096,endloc-baseloc)
```

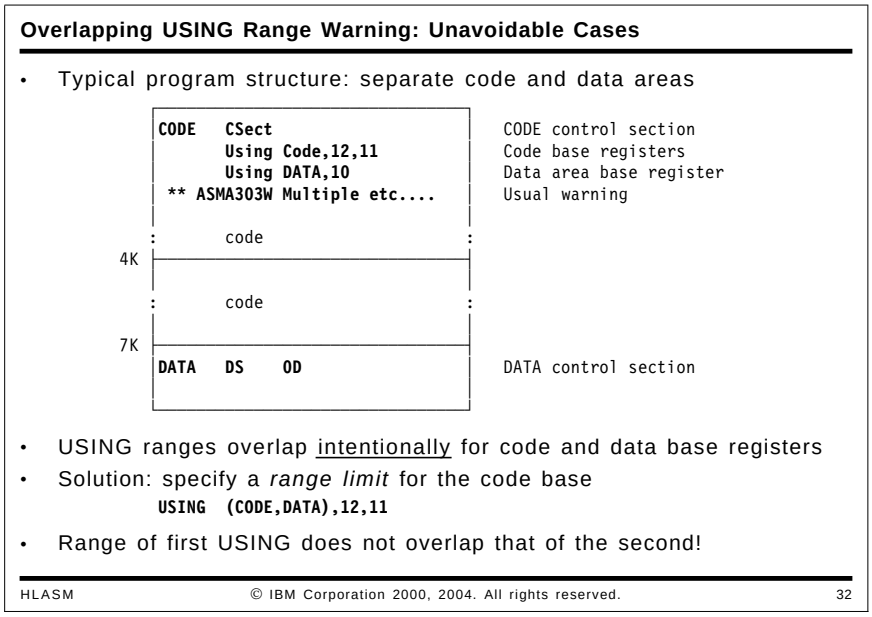
The assembler also checks that the intended range is non-empty; if endloc ≤ baseloc, the assembler issues an error message:

```
         USING (*,*),10
** ASMA313E The end value specified in the USING is less than or equal
            to the base value
```

Similarly, if the baseloc and endloc values have different relocatability attributes, the endloc address cannot be used to limit the range starting at baseloc, and the assembler indicates an error:

```
         USING (*,1000),10
** ASMA314E The base and end values have differing relocation attributes
```

Multiple USING Resolutions(2): Unavoidable Range Overlaps



It may happen that the structure of a program is such that the USING instruction ranges must overlap. For example, a program in which the code area is followed by a data area requiring different base registers cannot control the fact that the USING ranges overlap. This situation will normally encounter the ASMA303W diagnostic indicating the (known and expected) fact that the ranges overlap. Sometimes, users feel compelled to disable the warning (at the possible expense of not detecting errors like the one illustrated in “Multiple USING Resolutions(3): A Complex Example” on page 49 below).

Suppose your program is structured as in the following example:

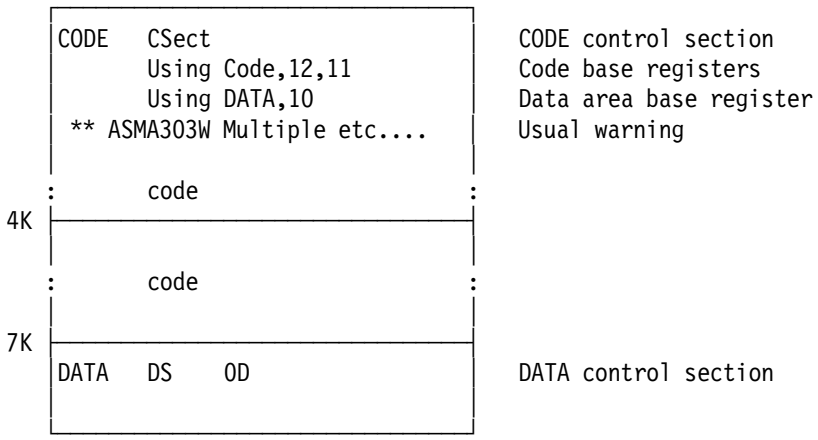


Figure 34. Program Structure with Unavoidable USING Range Overlaps

HLASM provides the range limit sub-operand of the first USING instruction operand to handle this situation: simply specify

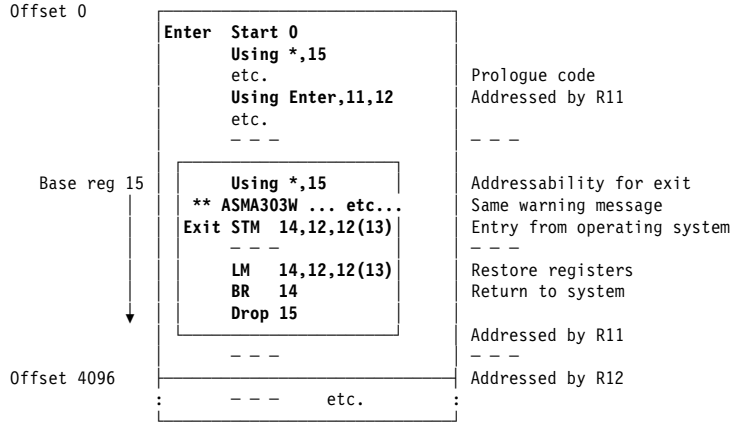
```
USING  (CODE,DATA),12,11
```

and the range of the USING for code addressability will be exactly the code area, and the ASMA303W diagnostic message will not appear.

Multiple USING Resolutions(3): A Complex Example

Overlapping USING Warning: Complex Example

- Program has grown larger, and now has an “asynchronous exit”



- Originally assembled without HLASM: didn't flag range overlap

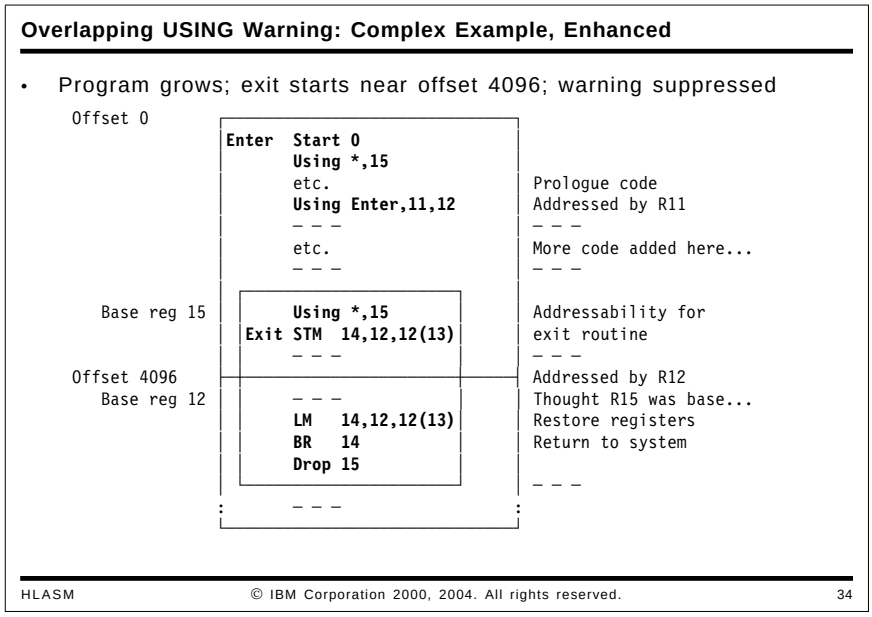
Now, suppose we add further code to the program in Figure 34 on page 48. It now includes a small routine that acts as an “exit” from some system service (such as a timer or program check interruption), and will be entered from the operating system at the label `Exit` with R15 set to the address of the entry point. The exit routine performs something and then returns to the system at the address in R14. The DS instruction reserving 3000 bytes is meant to indicate the presence of additional code.

```
Enter    Start 0
         Using *,15
         STM   14,12,12(13)      Save registers
         LR    11,15             Set local base register in R12
         LR    12,11             Second base
         AH    12,HW4096         Add 4096 for second base value
         B     DoSaves           Skip over constant
HW4096   DC    H'4096'           Constant
         Drop  15                Drop R15
         Using Enter,11,12       Provide local addressability
*
DoSaves  DC    0H'0'
         LA    10,SaveArea       Point to local save area
         ST    13,SaveArea+4     Store back chain in our area
         ST    10,8(,13)         Forward chain in caller's area
         LR    13,10             Point R13 to our area

*------------------------------ Begin processing
         DS    3000x             Lots of code doing good things
*
         Using Exit,15
  ** ASMA303W Multiple address resolutions may result from
              this USING and the USING on statement number 10
Exit     STM   14,12,12(13)      Save exit registers
         B     *+4               Code to ...
         B     *+4               do some useful stuff

         - - - - -               ...lots more code

         LM    14,12,12(13)      Restore registers
         BR    14                Return to caller
*
         DS    2000x             More code doing good things
SaveArea DC    18F'0'            Local save area
```

Figure 35. USING-Warning Program, Elaborated

This code segment will work as the writer expected, and (after testing is complete) he will probably believe that it has been fully debugged. The ASMA303W warning for the `USING Exit,15` instruction will appear to be spurious, and the programmer may be tempted to suppress it.

Multiple USING Resolutions(3): Complex Example, Enhanced



Now, suppose that over a period of time further code has been added preceding the exit routine, so that instead of 3000 bytes of code preceding the exit, there are now 4040 bytes, as illustrated below.

```
Enter    Start 0
         - - -                  (As above)
*------------------------------ Begin processing
         DS    4040x            Lots more code than before!
*
         Using Exit,15
Exit     STM   14,12,12(13)     Save exit registers
         B     *+4              Code to ...
         B     *+4              do some useful stuff

         - - - - -              ...lots more code

         LM    14,12,12(13)     Restore registers
         BR    14               Return to caller
*
SaveArea DC    18F'0'           Local save area
         End
```

Figure 36. USING-Warning Program Elaborated and Extended

The source code in the exit routine looks (and is) no different from the code in Figure 35 on page 50, but the *assembled statements* have a subtle and critically important difference, as shown in this fragment of the assembly listing:

```
  Loc  Object Code  Addr1 Addr2 Stmt   Source Statement
000000                            1 Enter    Start 0
              R:F  00000          2          Using *,15
000000 90EC D00C          0000C   3          STM   14,12,12(13)      Save registers
000004 18BF                       4          LR    11,15             Set local base register in R12
000006 18CB                       5          LR    12,11             Second base
000008 4AC0 F010          00010   6          AH    12,HW4096         Add 4096 for second base value
00000C 47F0 F012          00012   7          B     DoSaves           Skip over constant
000010 1000                       8 HW4096   DC    H'4096'           Constant
                                  9          Drop  15                Drop R15
             R:BC  00000         10          Using Enter,11,12       Provide local addressability
                                 11 *
000012                           12 DoSaves  DC    0H'0'
000012 41A0 C018          01018  13          LA    10,SaveArea       Point to local save area
000016 50D0 C01C          0101C  14          ST    13,SaveArea+4     Store back chain in our area
00001A 50A0 D008          00008  15          ST    10,8(,13)         Forward chain in caller's area
00001E 18DA                      16          LR    13,10             Point R13 to our area
                                 17 *
000020                           18          DS    4040x             Lots of code
                                 19 *
              R:F  00FE8         20          Using Exit,15
   ** ASMA303W Multiple address resolutions may result from this USING
               and the USING on statement number 10
000FE8 90EC D00C          0000C  21 Exit     STM   14,12,12(13)      Save exit registers
000FEC 47F0 F008          00FF0  22          B     *+4               Do some useful stuff
                                                 - - - - -
000FFC 47F0 C000  <--     01000  26          B     *+4               Do some useful stuff
001000 47F0 C004  <--     01004  27          B     *+4               Do some useful stuff
                                                 _ _ _ _ _
001010 98EC D00C          0000C  31          LM    14,12,12(13)      Restore registers
001014 07FE                      32          BR    14                Return to caller
```

Figure 37. USING-Warning Program Elaborated and Extended: Problems

The ASMA303W warning message following statement 20 is the key to understanding what has happened here. Had this message been suppressed, most programmers would not observe in the object code for statements 22 through 30 that register 15 is not being used as a base address throughout the exit. (Starting at statement 26, the assembler has used register 12 rather than register 15 as a base register for resolving the implied addresses.)

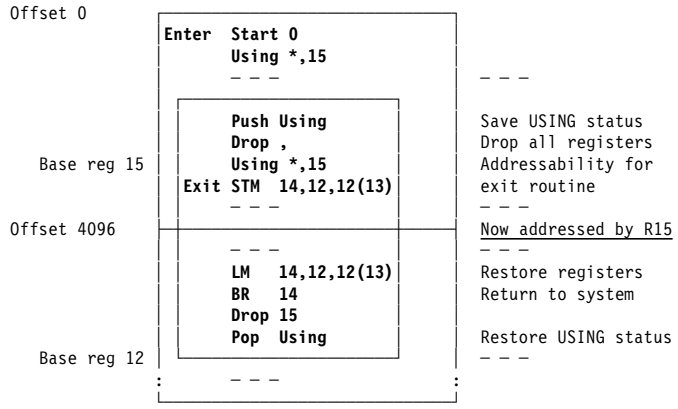
If the general register contents at entry to the exit routine have not been set by the operating system to contain the base registers used for the main program, the branch instructions resolved with respect to registers 11 and 12 will undoubtedly branch “wildly”. Finding this problem will probably be obscured by the quite sensible procedure of hunting through the newly added code in the main program (after all, the program worked correctly before the new code was added!) rather than by checking the base registers assigned in the exit.

The added code in the main program has “pushed” the start of the exit routine near to the boundary of addressability for registers 11 and 12. The 4K-byte boundary marking the start of addressability for R12 begins a few bytes after the exit routine begins; at that point, the register yielding the smallest displacements for the exit's instructions is no longer R15, but R12, so the assembler must use it to resolve implied addresses whose value exceeds X'1000' (the limit of addressability using R11).

Unfortunately, the rules used by the assembler to resolve implied addresses into base-displacement form (summarized on page 41) are difficult to remember, and their complexity (and sometimes, subtlety) can lead to programming errors that can be quite difficult to correct. The High Level Assembler's warning messages for USINGs will help make it less imperative that you remember such rules.

Fixing Unavoidably Overlapping USINGs

- Ensure that only R15 is a base in the exit routine:



HLASM 35

Fixing the problem is easy: insert PUSH USING and DROP instructions before the exit, and a POP USING following the exit, to ensure that only register 15 is used as a base register for the exit's statements. The effect of doing this can be seen in the following extract of an assembly of the program with the PUSH and POP instructions:

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement

000020                            18          DS    4040x            Lots of code

                                  20          Push  Using            Save USING-table status
                                  21          Drop  ,                Drop all base registers

                  R:F  00FE8              23          Using Exit,15
000FE8 90EC D00C             0000C    24 Exit     STM   14,12,12(13)     Save exit registers
000FEC 47F0 F008             00FF0    25          B     *+4              Do some useful stuff
                                                  - - - - -
000FFC 47F0 F018             01000    29          B     *+4              Do some useful stuff
001000 47F0 F01C             01004    30          B     *+4              Do some useful stuff
                                                  - - - - -
001010 98EC D00C             0000C    34          LM    14,12,12(13)     Restore registers
001014 07FE                           35          BR    14               Return to caller

                                  37          Pop   Using            Restore USING status
001016 5800 C01C             0101C    38          L     0,SaveArea       Check address resolution
00101A 0000
00101C 0000000000000000                 40 SaveArea DC    18F'0'           Local save area
                                  41          End
```

Figure 38. USING-Warning Program Elaborated and Extended: Problem Fixed

The USING at instruction number 23 is now used for address resolution throughout the “exit”, and all implied addresses are correctly resolved with a base register digit X'F'.

This example also provides one other indicator that the PUSH/POP technique has worked: at statement 38, the instruction referring to the symbol SaveArea has been resolved using register 12 as a base address.

One important observation: you can't fix this instance of overlapping USING ranges with range limits, because these ranges are *intentionally* overlapping rather than merely adjacent!

As these examples have illustrated, High Level Assembler for MVS & VM & VSE attempts to indicate potential trouble areas involving USING instructions. There will be cases where the messages describe situations that may not actually be errors; but a careful analysis of such diagnostics is recommended.

ASMA031E: Invalid Immediate or Mask Field

Immediate operands are used in many contexts. The recent addition of Halfword-Immediate instructions to the System/390 and z/Architecture series provides three types of operand:

- arithmetic, with values in the range −32768 to +32767.
- logical, with values in the range 0 to X'FFFF'.
- mask, with values in the range 0 to X'FFFF'.

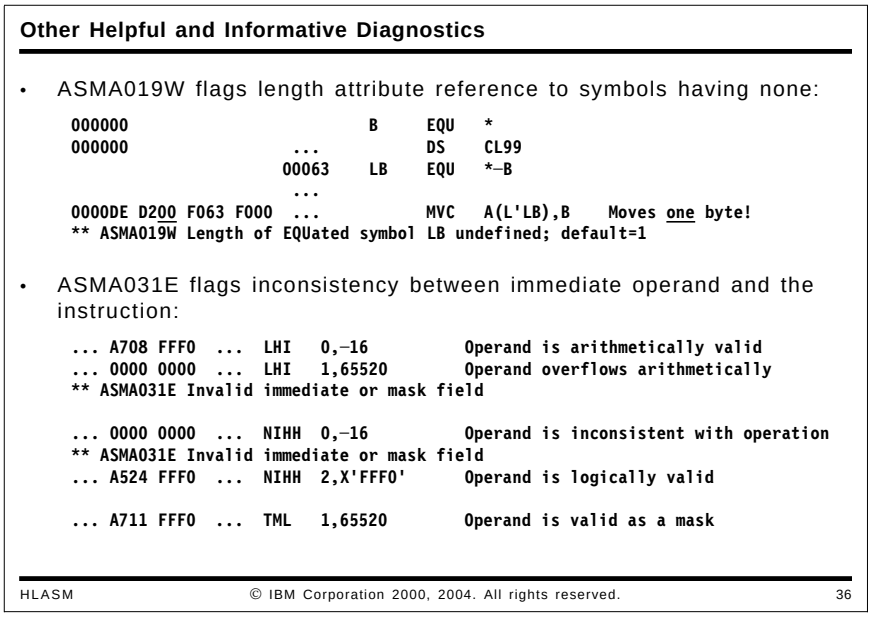
Because such operands are evaluated to 32 bits, you should take care to specify them in a form consistent with their use in the instruction. For example:

```
... A708 FFF0  ...  LHI   0,-16          Operand is arithmetically valid
... 0000 0000  ...  LHI   1,65520        Operand overflows arithmetically
** ASMA031E Invalid immediate or mask field
... A708 0FFF  ...  LHI   2,X'0FFF'      Operand is arithmetically valid
... 0000 0000  ...  LHI   2,X'FFF0'      Operand overflows arithmetically
** ASMA031E Invalid immediate or mask field
... A708 FFF0  ...  LHI   2,X'FFFFFFF0'  Operand is valid arithmetically

... 0000 0000  ...  NIHH  0,-16          Operand is inconsistent with operation
** ASMA031E Invalid immediate or mask field
... A514 FFF0  ...  NIHH  1,65520        Operand is logically valid
... A524 FFF0  ...  NIHH  2,X'FFF0'      Operand is logically valid

... 0000 0000  ...  TML   0,-16          Operand is inconsistent with operation
** ASMA031E Invalid immediate or mask field
... A711 FFF0  ...  TML   1,65520        Operand is valid as a mask
... A721 FFF0  ...  TML   2,X'FFF0'      Operand is valid as a mask
```

Figure 39. Examples of ASMA031E Diagnostic

HLASM attempts to verify that the form of the immediate operand is consistent with the type of instruction. Thus, for example, X'FFF0' is considered invalid for the LHI instruction, because the result will be negative, not positive. If you want to specify a negative arithmetic operand using hexadecimal notation, be sure to specify the correct number of high-order 1-bits.

Other Helpful and Informative Diagnostics



Some other diagnostics may help you find obscure behaviors.

ASMA019W: Length of EQUated Symbol Undefined

If a length value is defined using an EQU expression, it may happen that someone forgets to use that value and uses its length attribute instead:

```
000000                          B     EQU   *
000000                   ...          DS    CL99
                        00063    LB    EQU   *-B
                         ...
0000DE D200 F063 F000    ...          MVC   A(L'LB),B    Moves one byte!
** ASMA019W Length of EQUated symbol LB undefined; default=1
```

Figure 40. Example of ASMA019W Diagnostic for Length Attribute of a Length

Using the length attribute of a length is probably not what was intended; just remove the L' to fix the statement. Another “cure” is to define a length attribute on the EQU statement that defines LB:

```
000000                          B     EQU   *
000000                   ...          DS    CL99
                        00063    LB    EQU   *-B,99       Define length
                         ...
0000DE D262 F063 F000    ...          MVC   A(L'LB),B    Moves 99 bytes
```

This is valid, but the added obscurity seems self-defeating.

LANGUAGE Option

High Level Assembler supports English, German, Japanese, and Spanish language messages and listing headings. These are specified as:

LANGUAGE(DE)	German
LANGUAGE(EN)	Mixed-case English
LANGUAGE(ES)	Spanish
LANGUAGE(JP)	Japanese
LANGUAGE(UE)	Upper-case English

For example, here is the same message in English, German, and Spanish:

```
** ASMA019W Length of EQUated symbol LB undefined; default=1
** ASMA019W Länge des EQU Symbols LB nicht definiert; Standardwert=1
** ASMA019W Longitud del símbolo EQUated LB indefinida; por omisión=1
```

LIST(133) Option

The traditional listing width is 120 characters (plus one for carriage control). By expanding the listing to 132 characters, HLASM is able to provide additional detail:

- location counter values are displayed as 8 hexadecimal digits, corresponding to support for larger program sizes
- an extra digit is available for statement numbers
- more information is provided for macro names and nesting levels on generated statements

Figures 41 and 42 illustrate the differences: Figure 41 is the familiar 120-column “narrow” listing format, as specified by the LIST(121) option.

```
000110                          130+IHB0012A DS    0H                                                  01-WTO
000110 0A23                     131+         SVC   35                  ISSUE SVC 35                    01-WTO
000112 181D                     133          LR    1,13              Point to local save area
000114 58D0 D004          00004 134          L     13,4(,13)         Get system's save area pointer
                                136          FREEMAIN R,A=(1),LV=72  Free the local save area
000118 4100 0048          00048 137+         LA    0,72(0,0)                 LOAD LENGTH           01-FREEM
00011C 0A0A                     138+         SVC   10                        ISSUE FREEMAIN SVC    01-FREEM
                                140          RETURN (14,12),RC=0     Return with code zero
00011E 98EC D00C          0000C 141+         LM    14,12,12(13)              RESTORE THE REGISTERS 01-RETUR
000122 41F0 0000          00000 142+         LA    15,0(0,0)                 LOAD RETURN CODE      01-RETUR
```

Figure 41. Example of Macro Expansion with LIST(121)

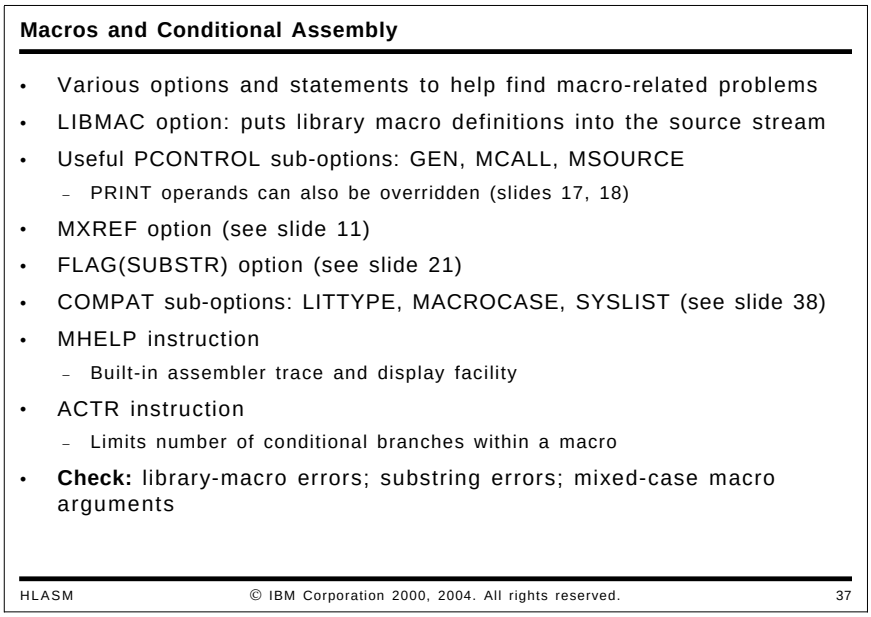
In the wider 132-column listing format shown in Figure 42, location counter and address values display eight hexadecimal digits, and all eight characters of the statement-generating macro name appear at the right end of the line.

```
00000098                                  70+IHB0006A DS    0H                                                  01-WTO
00000098 0A23                             71+         SVC   35                  ISSUE SVC 35                    01-WTO
0000009A 181D                             73          LR    1,13              Point to local save area
0000009C 58D0 D004          00000004      74          L     13,4(,13)         Get system's save area pointer
                                          76          FREEMAIN R,A=(1),LV=72  Free the local save area
000000A0 4100 0048          00000048      77+         LA    0,72(0,0)                 LOAD LENGTH           01-FREEMAIN
000000A4 0A0A                             78+         SVC   10                        ISSUE FREEMAIN SVC    01-FREEMAIN
                                          80          RETURN (14,12),RC=0     Return with code zero
000000A6 98EC D00C          0000000C      81+         LM    14,12,12(13)              RESTORE THE REGISTERS 01-RETURN
000000AA 41F0 0000          00000000      82+         LA    15,0(0,0)                 LOAD RETURN CODE      01-RETURN
000000AE 07FE                             83+         BR    14                        RETURN                01-RETURN
```

Figure 42. Example of Macro Expansion with LIST(133)

The GOFF option requires the LIST(133) option, because 8-digit values are used for the location counter and other address-related fields.

Macros and Conditional Assembly



There are many options and statements that can help you locate problems with macros.

- The LIBMAC option causes HLASM to bring the source statements of the macro definition into the input stream. Normally, macros are read from the library and encoded internally, so that errors found during encoding and expansion cannot be identified with a specific line within the macro definition. By placing the macro definition in the source stream, each line has a statement number that can be used to identify specific statements in case of error.
- The PCONTROL option lets you force portions of the listing to be displayed. These options are especially helpful with macro problems:
  - PCONTROL(GEN) is described on page 32.
  - PCONTROL(MCALL) is described on page 32.
  - PCONTROL(MSOURCE) is described on page 33.
- The MXREF option causes HLASM to show all macros and COPY segments used by the program, where they were used, and where they were called. Details are described on page 17.
- The FLAG(SUBSTR) option causes HLASM to check for potential problems with conditional assembly substring operations, as described on page 59.
- Three COMPAT sub-options can help with problems of compatibility with older assemblers, as well as providing greater freedom in the way you write your programs. Details are on page 60.
- For really difficult macro problems, the MHELP instruction provides extensive tracing and display capabilities. These are described on page 63.
- In situations where you suspect a macro may be looping, the ACTR instruction provides a way to limit the number of conditional assembly “branches”. See page 63 for details.

LIBMAC Option

The LIBMAC option causes High Level Assembler to treat library macros as though they were defined inline at the point of their first reference in the source program. With this option you can detect and track the causes of errors in library macro definitions without having to manually extract them from the library for insertion at a proper place into the source program.

In this example, a macro BadMac with a potential error was installed in a macro library, and called with an argument that causes an error. Without LIBMAC, HLASM reports the error as being “somewhere” in the macro.

```
                             4        BadMac   65535
** ASMA103E Multiplication overflow; default product=1 - BADMA
                             5+*,&A*&A = 1                                    01-BADMA
```

Figure 43. Error from Library Macro

Lacking better clues as to the source of the error, you might be forced to retrieve the macro definition and study or test it to find the problem. However, if you specify the LIBMAC option, or precede the call with an ACONTROL LIBMAC instruction (see page 65), then the macro definition will be placed inline at the point of the call. This allows HLASM to identify the specific line in the macro definition that causes the error.

In this example, the same macro is called; HLASM indicates the number (00008) of the problem statement.

```
                             6        MACRO
                             7 &L     BADMAC  &A
                             8 &X     SETA    &A*&A
                             9        MNOTE   *,'&&A.*&&A. = &X.'
                            10        MEND
                            11        BadMac   65535
** ASMA103E Multiplication overflow; default product=1 - 00008
                            12+*,&A*&A = 1                                    01-00009
```

Figure 44. Error from Library Macro Pinpointed by LIBMAC

Note also that the MNOTE output identifies the statement number (00009) in the macro that generated the output; without LIBMAC, it simply displays the name of the macro, as shown in Figure 43.

PCONTROL Options Relating to Macros

Three PCONTROL sub-options are useful in finding and debugging macro-related problems: GEN, MCALL, and MSOURCE. Details are provided at “PRINT Instructions and the PCONTROL Option” on page 30.

Macro-COPY Cross-Reference (MXREF Option)

The MXREF option produces a great deal of helpful information (see page 17). When used with the LIBMAC and PCONTROL(MCALL) options, the most detailed levels of data are produced.

FLAG(SUBSTR) Option and Conditional-Assembly Substrings

The Language Reference manuals for the Assembler Language have always stated that substring operations of the form

```
&SubStr  SetC  '&CharVar'(&Start,&Len)   &Len characters starting at &Start
```

are valid only if the extracted substring lies entirely within the bounds of the subject string (&CharVar in this example), and that the assembler would diagnose any misuse. However, Assembler H and HLASM Release 1 did not diagnose this situation.

When IBM applied a correction that caused the error to be flagged, it happened that many programs had relied on the error. A typical technique for extracting the remainder of a character string was to write something like

```
&SubStr  SetC  '&CharVar'(&Start,255)   Take rest of characters at &Start
```

In order to allow such programs to continue to assemble without diagnostic, you may specify the FLAG(NOSUBSTR) option.

A better approach is to use the explicit “remainder of string” notation:

```
&SubStr  SetC  '&CharVar'(&Start,*)     Take rest of characters at &Start
```

as this will then allow the assembler to diagnose “true” errors in the specification of substrings.

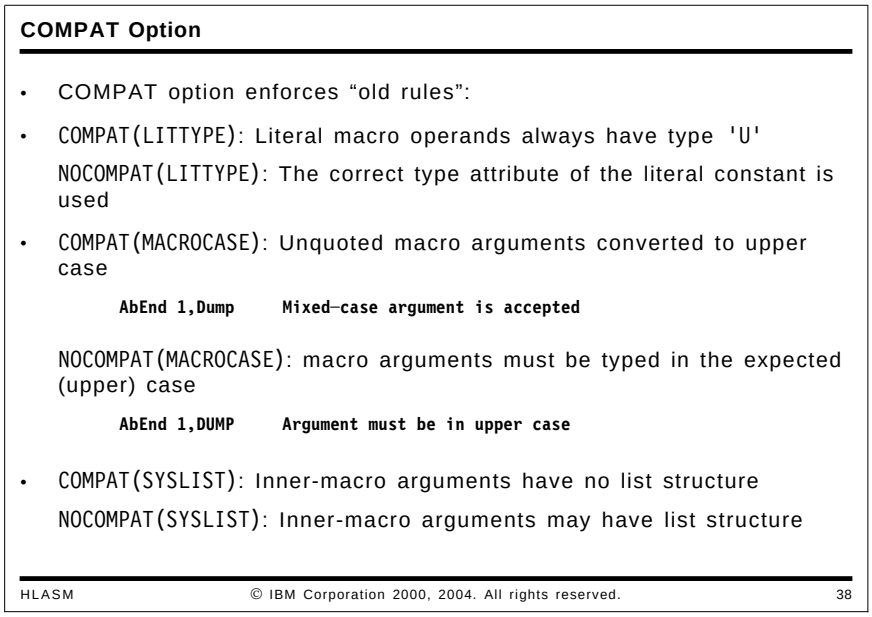
FLAG(SUBSTR) specifies that High Level Assembler should diagnose improper character substrings in the conditional-assembly language. For example:

```
&C     SETC   'ABCDE'(4,5)
```

specifies a substring (five characters, starting with 'DE') that extends beyond the end of the original string; this is an error. Normally, High Level Assembler issues the ASMA094W warning message; if FLAG(NOSUBSTR) is specified, High Level Assembler suppresses the warning.

Things Worth Checking: FLAG(SUBSTR) can help in macros and conditional assembly statements to detect coding errors that might produce unexpected or unpredictable results.

COMPAT Option



The COMPAT option allows you to control the degree of compatibility High Level Assembler should enforce for programs accepted by previous assemblers. This is useful in situations when programs are developed using the benefits and features of High Level Assembler, but the programs must be distributed to sites which might assemble them on a different assembler, or where the costs of modifying the differences in existing programs is not justified. (Not all such differences can be controlled, of course!)

There are four areas of intentional incompatibility between HLASM and Assembler H in the ways they treat the input text of the program:

- sensitivity to the case of the input text
- the handling of substituted sublists in macro arguments and character-variable strings
- the case of unquoted macro operands
- the type attribute of literals as macro arguments.

These differences can be controlled by using the COMPAT option, or with ACONTROL COMPAT(...) statements, as described on page 65.

COMPAT(LITTYPE) Option: Attribute References to Literals

Note: Attribute references to literals used as macro operands may result in different values from previous assemblers. For example, Assembler H returned value 'U' for type attribute references to literal operands, whereas HLASM may return 'U' if the literal was not previously defined, or its "normal" type if it was previously defined. If the behavior of Assembler H is required, specify the COMPAT(LITTYPE) option and HLASM will then return 'U' as the type attribute of all literals in macro operands.

Previous assemblers treated literals in macro arguments as always having type 'U', whereas High Level Assembler tries to provide the actual type if it is known. You can specify the COMPAT(LITTYPE) option to request that HLASM always return 'U' as the type attribute of a macro operand whose argument is a literal.

Things Worth Checking

- Look for macros that depend on the type attribute of literals: their type can change during an assembly depending on whether or not they have been "defined" in the symbol table.
- Assuming that type attribute 'U' implies a literal operand is dangerous because many other operands can have that type.
- Examining the first character of an operand for an equal sign is safer, but not always error-proof.

COMPAT(MACROCASE) Option: Mixed-Case Macro Operands

The COMPAT(MACROCASE) option allows you to write macro calls using mixed-case operands for macros that expect upper-case operands. HLASM allows symbols and other statement elements to be specified in mixed case. In some situations, you may want to write macro argument strings in mixed case, for macros that otherwise would be able to handle only uppercase argument strings. COMPAT(MACROCASE) specifies that High Level Assembler should *internally* translate lowercase characters in unquoted macro arguments to uppercase before the macro is expanded.

For example, in older assemblers where all instruction mnemonics and operands were required to be in upper case letters, you would write something like this:

```
AbEnd  13,DUMP
```

to invoke the system Abnormal End service. With the availability of mixed-case support in High Level Assembler, you may have wanted to write

```
AbEnd  13,Dump
```

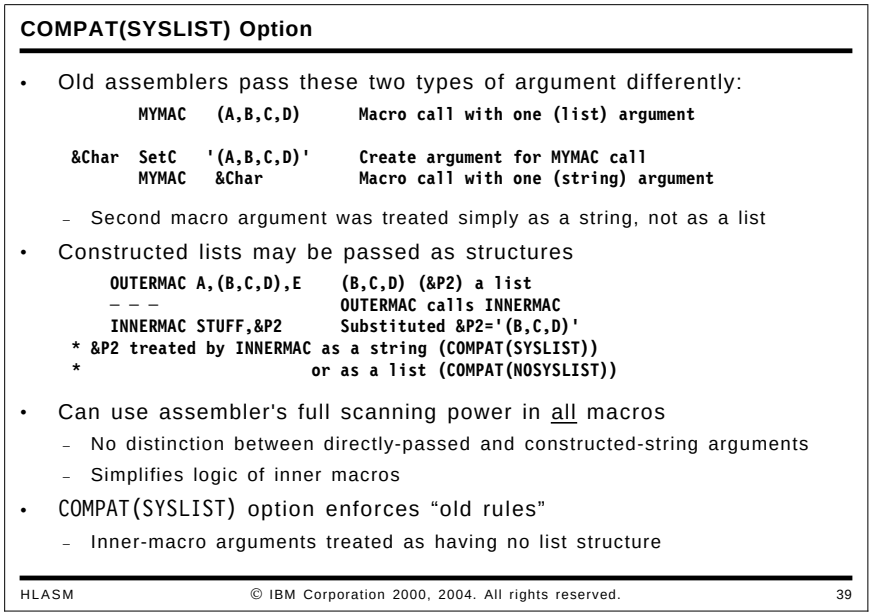
for increased readability.

The problem here is that the ABEND macro was written when only upper-case operands were allowed, so that the internal logic of the macro checks for the presence of a DUMP operand (all capital letters) and does not recognize the mixed-case operand "Dump".

If you specify the COMPAT(MACROCASE) operand, HLASM will automatically convert the mixed-case operand to upper case just before passing it to the internal logic of the macro, which then recognizes DUMP as the operand, even though the original source statement is unchanged.

Things Worth Checking: If familiar, frequently-used macros appear to generate spurious error messages, check whether operands may have been specified in mixed case (while the macro was not written to recognize them). Specifying the COMPAT(MACROCASE) option may be all that's needed to fix the problem.

COMPAT(SYSLIST) Option: Inner-Macro Argument Lists



In older assemblers, character strings substituted as operands of calls to inner macros were treated only as unstructured character strings, independent of their actual structure. This meant that argument scanning techniques might depend on whether the macro was invoked from open code or from another macro; inner macros had to parse the operands one character at a time.

HLASM permits such substituted operands to be treated as having a list structure that is accessible to the assembler through the normal &SYSLIST facilities such as the number and count attributes, as well as the usual ability to designate sublists and sublist elements symbolically or by using a subscript notation. This means that macros need not be written differently depending on whether they are invoked as “outer” or “inner” macros.

If it is desired that HLASM treat such operands and SETC variables as was done in previous assemblers, specify the COMPAT(SYSLIST) option, or with ACONTROL instructions (see page 65). However, if you specify the COMPAT(NOSYSLIST) option, High Level Assembler can recognize substituted sublists as having a list structure. Thus you can construct complex macro operands in an outer macro to be passed as list structures to inner macros. This capability can help remove many unnecessary distinctions between outer and inner macros.

MHELP Instruction

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

1	Trace macro calls
2	Trace macro branches
4	AIF dump
8	Macro exit dump
16	Macro entry dump
32	Global suppression
64	Hex dump
128	MHELP suppression

These values are additive: you may specify any combination.

MHELP is valuable when really difficult macro problems must be resolved. Its output can be large, so you may want to use it only for critical parts of the program.

ACTR Instruction

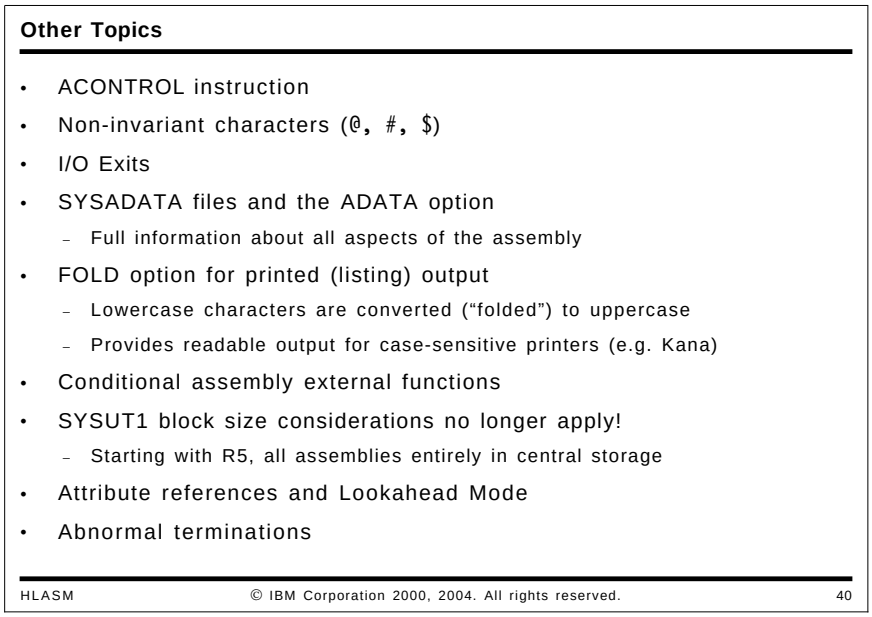
The ACTR instruction can be use the limit the number of conditional assembly branches (AIF and AGO) executed within a macro invocation (or in open code). It is written

```
ACTR  arithmetic_expression
```

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

ACTR is most useful if you suspect a macro may be looping or branching excessively; you can set a lower ACTR value to limit the number of allowed branches.

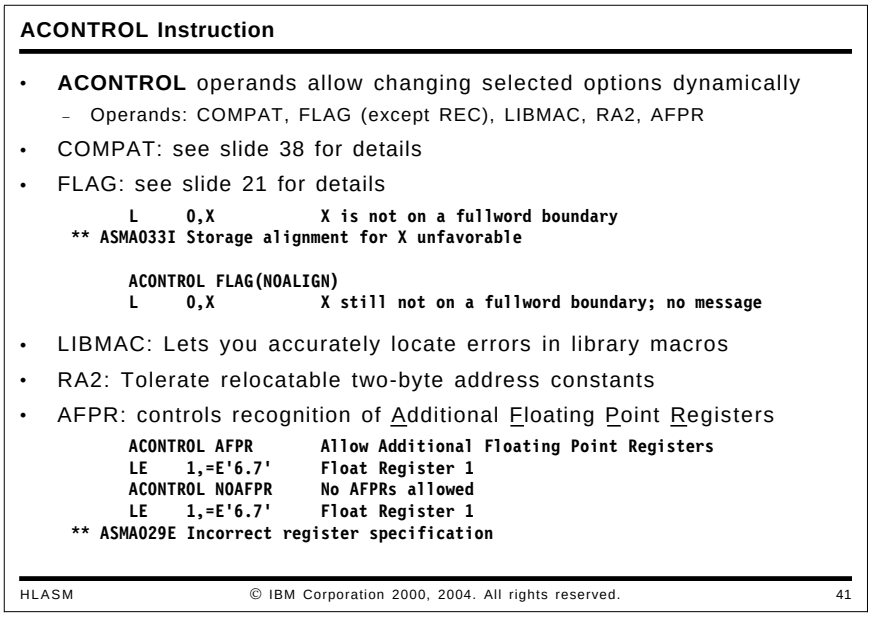
Other Things



Other factors worth knowing about include:

- The ACONTROL instruction (see page 65) allows you to control dynamically the settings of certain options. This gives you more flexibility and precision in selecting ranges of statements for chosen diagnostic checks.
- Certain characters are not “invariant” across all EBCDIC encodings; see page 66 for details.
- I/O exits may modify input and output files, so the presence of exits may mean that the assembler's listing (which itself may be modified!) may not accurately reflect all inputs or outputs. Exits are discussed on page 66.
- The ADATA option causes HLASM to generate a SYSADATA “side file” containing useful information about the assembly. See page 67 for details.
- The FOLD option discussed on page 67 may have changed the appearance of the listing.
- External functions can add capabilities beyond the “native” facilities of the assembler; these are described on page 67.
- Sometimes the behavior of an assembly depends on the order of the statements in the source program. Some considerations are discussed on page 68.
- Some assembler abnormal terminations are due to internal errors, while others may be due to apparently normal coding techniques, as discussed on page 69.

ACONTROL Instruction



The ACONTROL instruction allows you to dynamically change the settings of a subset of assembler options. For example, you can request that High Level Assembler check for possible continuation statement errors at invocation time, and then turn off the checking around a specific set of statements.

The examples of the FLAG(CONT) option in slide 22 on page 36 showed how diagnostics for continuation errors could be suppressed. If you want to retain this valuable checking for the entire program except for a particular statement, you can “bracket” the trusted statement with ACONTROL instructions controlling FLAG(CONT) checking. Normal checking might cause a message like the following:

```
              ELSE  Otherwise, do that and this          ← note comma!
  ** ASMA431W Continuation statement may be in error -
     continuation indicator column is blank.
```

With ACONTROL instructions, the message is suppressed:

```
              AControl FLAG(NOCONT)              Suspend checking
              ELSE  Otherwise, do that and this
              AControl FLAG(NOCONT)              Resume checking
```

The error shown in Figure 44 on page 58 could also have been exposed by placing an ACONTROL instruction before the macro call:

```
        AControl LibMac
        BadMac   65535
```

Things Worth Checking: You might be inclined to avoid certain useful diagnostics because one or two valid statements in a program are flagged by the assembler. Rather than suppress the diagnostics entirely, it is better to bracket the valid statements with ACONTROLs, so that the rest of the program can still be checked.

Non-Invariant Characters

Each EBCDIC character is assigned a specific encoding that defines its numeric value. For example, the letter A is assigned value 193, or X'C1'. This is what makes character self-defining terms equivalent to other forms such as binary and hexadecimal.

Most of the EBCDIC characters recognized by HLASM having syntactic validity in the language have the same encoding across code pages: that is, all but three characters have invariant encodings. The three non-invariant characters are not assigned consistent values, *even though they are valid in symbols*!

These three symbols are the at sign (@), the sharp or pound (US!) sign (#), and the dollar sign ($). For example, a program that scans data for the presence of a dollar sign or other special characters might use CLI instructions such as

```
CLI   0(R4),C'$'     Assumes C'$' = X'5B' =  91
CLI   0(R4),C'#'     Assumes C'#' = X'7B' = 123
CLI   0(R4),C'@'     Assumes C'@' = X'7C' = 124
```

only to find that the encoding of the data does not use the same representation of the dollar sign assumed by the assembler when the program was assembled.

The “Syntactic Character Set” with encodings common to all EBCDIC code pages are the space (blank) and these 81 characters:

- upper and lower case alphabetics A-Z, a-z
- numeric digits 0-9
- the special characters

```
. , : ; ? ( ) ' ″ / - _ & + % * = < >
```

Things Worth Checking: You should avoid using non-invariant characters in any program that is assembled outside the USA, or which might process data that originates elsewhere. Even using non-invariant characters in symbols might cause a program not to assemble correctly if modifications are made on systems using different encodings.

I/O Exits

I/O exits can process all records being read and written by the assembler, and therefore have considerable control over the object file and what you see in the listing. The final page of the assembly listing shows what actions have been taken by each exit (see the example on page 24). While it is possible for a listing exit to hide the presence of all exits, such a situation is very unlikely.

Note that I/O exits can also produce diagnostic messages that may appear in the listing.

Things Worth Checking: The presence of SYSIN or SYSLIB exits can mean that source files have been modified; the presence of SYSLIN or SYSPUNCH exits can mean that the object file has been modified; and the presence of a SYSPRINT exit can mean that the listing has been modified.

SYSADATA File

If you specify the ADATA option, HLASM produces a SYSADATA file of information about all aspects of the assembly, even if parts of the listing are suppressed. The file is intended to be read by programs (unlike the listing, which is formatted for human readers), such as debuggers, program understanders, “bill-of-materials” processors, and so forth. (An example of a SYSADATA exit that creates a list of all library members and the data sets they came from is provided with HLASM as a sample program; it is described in the *High Level Assembler for MVS & VM & VSE Programmer's Guide*.)

A key feature of the SYSADATA file is that it contains all the information in the listing (except for that produced by the INFO and OPTABLE(..,LIST) options), even if the listing is suppressed or if a SYSPRINT exit is active.

HLASM R5 supports ADATA exits that allow record selection and suppression. Also, the layout of the records has been modified, and an optional ASMAXADR exit is provided to reformat R5 ADATA files to R4 format.

FOLD Option

The FOLD option lets you specify that all alphabetic characters in the listing file (whatever their original case) should be produced in upper case only. Character data entered in lower case will of course be converted to the appropriate lower case code points; only the listing file is affected by the FOLD option.

This option is provided so that languages that use the code points of lower case letters for other ideographs or scripts, such as Katakana and Hiragana, can be printed readably.

The case of messages and text sent to the SYSTERM file (normally, the terminal) is not affected by the FOLD option.

Things Worth Checking: Because all lower case letters are forced to upper case on all listing lines, you should check carefully that the contents of DCs, SETC values, ESD aliases, and symbol type attributes in the symbol XREF have not been obscured.

External Conditional Assembly Functions

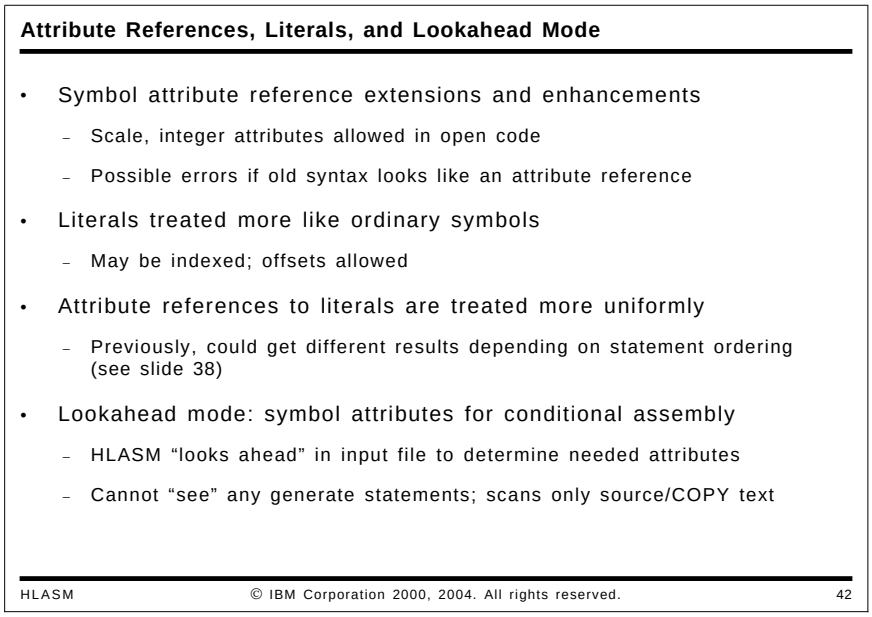
HLASM's Support for external conditional assembly functions allows you to access capabilities not supported directly by the assembler, such as special processing of conditional assembly data and interfaces to the assembler's operating environment.

Your listing may also contain messages produced by external functions (and that are not documented in the HLASM manuals). The final page of the assembly listing summarizes the functions called and the actions they take; see the example in Figure 16 on page 24.

Things Worth Checking: The presence of external-function calls can mean that special code is being generated by macros or other conditional assembly statements.

Things Worth Checking: Some assembler errors such as messages ASMA105U, ASMA170S, or ASMA253C may be correctable by specifying larger block sizes on SYSUT1.

Attribute References, Literals, and Lookahead Mode



High Level Assembler recognizes certain attribute references in contexts where they were not allowed by previous assemblers. The only attribute reference formerly permitted in “open code” was the Length Attribute Reference (L'); High Level Assembler supports Scale (S') and Integer (I') Attribute references in open code. Conditional assembly allows the use of references to the Length, Scale, Integer, Type (T'), Count (K'), Opcode (O'), Number (N'), and Definition (D') attributes of variable symbols.

Things Worth Checking: Some unusual operands using character strings starting with a single letter followed by an apostrophe might be recognized by High Level Assembler as attribute references where previous assemblers had ignored them.

Literal Extensions

High Level Assembler permits literals to be used in wider contexts than previous assemblers. For example, in machine instruction statement operands, a literal may be used as an ordinary relocatable term, or may be indexed; previous assemblers required that the literal be the only term in the operand, and indexing was not allowed. **Note:** Beginning with Release 4, HLASM flags the use of literals as the targets of EX and branch instructions.

Attribute references to literals are allowed in most contexts, whether or not the literals were previously defined.

Literals may also be used as macro instruction operands, and type attribute references to those operands will return a reasonable value for all references rather than the previous “unknown” on the first reference. The COMPAT(LITTYPE) option lets you control this behavior; see page 61 for details.

Lookahead Mode

Whenever an attribute of an unknown symbol is required during a conditional assembly operation, HLASM suspends normal operation and enters “lookahead mode”. The input stream is scanned until the required symbol is found. Its attributes are then entered into the symbol table and normal processing is resumed. Whenever lookahead mode is invoked, the source text is compressed and stored in an internal file, so the input file need not be reread. Attributes of other symbols found during the search are entered into the symbol table, so that attribute references to those symbols will not cause lookahead. Further requests for statements from the input stream will be referred to the lookahead file until it is exhausted, when input will then resume from the primary (SYSIN) file again.

It is possible that conditional assembly might later generate different definitions of symbols from those found during lookahead mode; the attributes are “corrected” when the symbols are generated. Note also that only the *attributes* of a symbol are determined in lookahead mode; the value and relocatability attributes are unknown until conditional assembly and the first ordinary-assembly passes are complete.

Things Worth Checking: Symbol attributes used for conditional assembly might be different from the attributes at the end of the assembly; the symbol cross-reference will show the final values.

Assembler Abnormal Termination

Assembler Abnormal Terminations

Several conditions can cause abnormal/early assembly termination:

- HLASM is unable to load certain modules
  - Main processing module (ASMA93), default options, opcodes, exits, functions, messages, translate table, Unicode table
- A loaded module is found to be invalid
- Missing required file(s)
- Invocation-option errors and the PESTOP install option
- External functions and I/O exits
  - Return codes can request explicit (and orderly) termination
  - ABENDs will kill the assembly
- Insufficient virtual storage
- Internal errors (e.g., messages 950-64, 970-1, 976)
  - Some may be correctable with larger SYSUT1 block size
- COPY loops: excess DASD or CPU time

HLASM 43

Sometimes an abnormal termination of an assembly can be caused by source-program or assembly-environment conditions over which the assembler has little or no control.

Loaded Modules and Required Files

HLASM loads many modules dynamically depending, on the options specified for the assembly. If any of these modules is unavailable, or when loaded is found to have an invalid format, the assembler will terminate immediately.

Similarly, if a file required for the assembly is missing (such as SYSIN, or other files required for the specified options), the assembly is terminated.

Option Errors and PESTOP

If the PESTOP option is specified when HLASM is installed, any error in options processing will terminate the assembly. This can save the time and resources needed to re-run a complete assembly that was discarded because of the errors.

I/O Exits and External Functions

Exits and functions run in same task and space as the assembler itself, so that errors can cause the assembly to fail; there is no error recovery in assembler itself. Assembler failures are rare, so if you are using exits or external functions, check to see if the problem may have originated there; exit and external-function errors may be difficult to detect.

Virtual Storage

While HLASM can write much of its working data to its utility file, some portions must remain in central storage throughout the assembly. Insufficient storage can cause the assembler to terminate in many different ways.

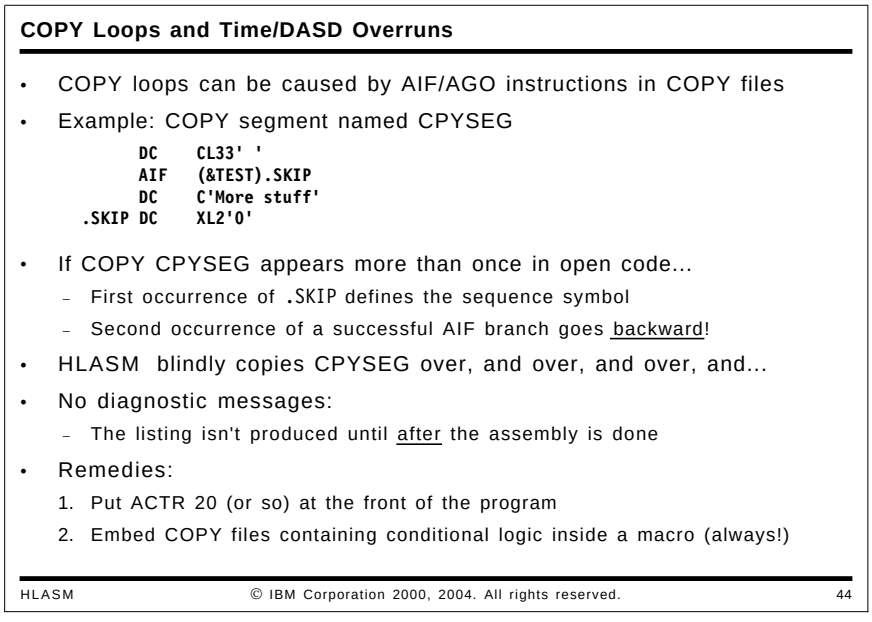
The assembly summary also includes information about the amount of storage used (see Figure 18 on page 24).

Internal and I/O Errors

Occasionally an internal error in the assembler will cause an abnormal termination. Some of these errors are accompanied by a message indicating that HLASM has detected the error itself; others may cause an ABEND condition. If reproducible, these should be reported to IBM Service.

I/O errors may be caused by incorrect JCL, or may be transient conditions that can be corrected by moving or restoring a file.

COPY Loops and Excess DASD or CPU Use



Sometimes HLASM uses unexpectedly large amounts of CPU time or DASD space, and increasing the allotment of either doesn't fix the problem; and, no error messages are produced to help locate the problem!

An apparently normal coding practice might cause this behavior. Suppose you write a COPY file named CPYSEG that contains some conditional assembly logic:

```
        DC    CL33' '
        AIF   (&TEST).SKIP
        DC    C'More stuff'
.SKIP DC    XL2'0'
```

Then, suppose you copy this segment into (say) a DSECT:

```
DATA1 DSECT
      COPY  CPYSEG
```

This causes no problems, whether &TEST is true (1) or false (0). Suppose now that you want to use the same data structure in a second DSECT:

```
DATA2 DSECT
      COPY  CPYSEG
```

HLASM effectively sees a code sequence like this:

```
DATA1 DSECT
*     COPY  CPYSEG
      DC    CL33' '
      AIF   (&TEST).SKIP
      DC    C'More stuff'
.SKIP DC    XL2'0'
DATA2 DSECT
*     COPY  CPYSEG
      DC    CL33' '
      AIF   (&TEST).SKIP     May branch backward!
      DC    C'More stuff'
.SKIP DC    XL2'0'
```

The first COPY causes the sequence symbol **.**SKIP to be defined. The second COPY can cause one of two problems for HLASM. First, if &TEST is false, no conditional assembly branch is taken, statement processing flows sequentially without branching, and HLASM will diagnose a redefinition of the sequence symbol **.**SKIP.

The more serious problem occurs when &TEST is true, because the second AIF will branch back to the *first* occurrence — the definition — of **.**SKIP! But this precedes the COPY instruction following the declaration of the second DSECT, so HLASM executes COPY CPYSEG repeatedly, eventually using up all the space allocated to the SYSUT1 utility file, or exceeding the CPU time limit.

Because HLASM must read the entire source program (and buffer it to DASD if there's not enough central storage) before it produces a listing, the end of the program is never found, and no warnings can be produced.

There are two ways to “expose” the problem. One is to insert an ACTR instruction at the beginning of the program:

```
       ACTR  10     Allow only 10 successful conditional assembly branches
```

This will terminate the COPY loop, and HLASM may then be able to provide meaningful information. (Choose an ACTR value appropriate to the number of successful conditional assembly branches you expect in the program.)

A better approach is to use a macro. If a COPY file must contain any conditional assembly logic, encapsulate that logic in a macro definition. Using the previous example, you could write

```
        MACRO
        CPYSEG
        GBLB   &TEST
        DC     CL33' '
        AIF    (&TEST).SKIP
        DC     C'More stuff'
.SKIP   DC     XL2'0'
        MEND
```

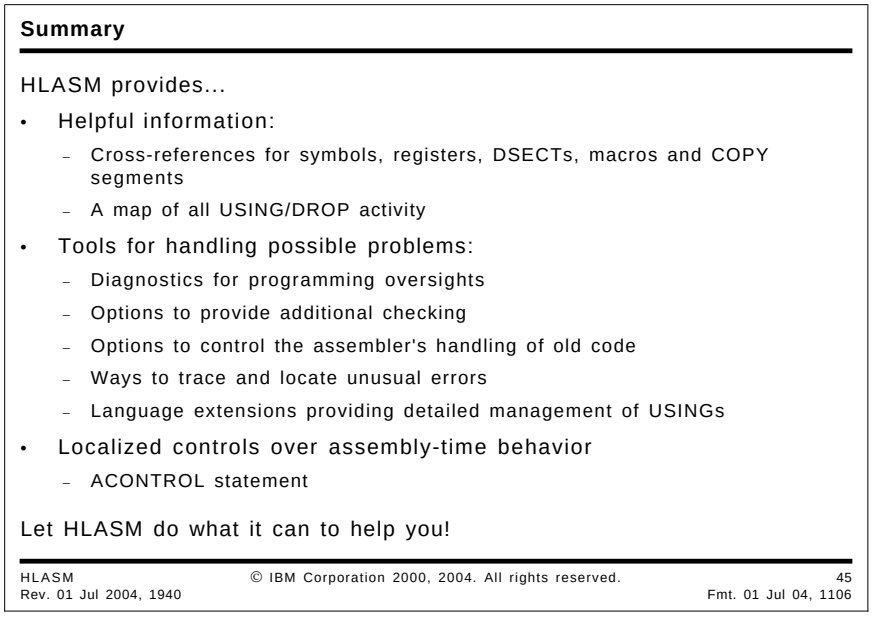
and then write the program with macro calls instead of COPY instructions:

```
DATA1   DSECT
        CPYSEG
DATA2   DSECT
        CPYSEG
```

Looping inside a macro is likely to create only limited damage.

Things Worth Checking: Check for AIF and AGO instructions in all COPY segments, and consider replacing the segments with macros that provide the same function, to limit the scope of such conditional assembly branching.

Summary



HLASM supports many enhanced features that ease the daily chores of finding and fixing problems in Assembler Language programs. While it can't find errors of logic (the HLASM Toolkit Feature can help with this!), many of the features can help reduce the likelihood of error, or can provide information useful in locating and identifying problems.

Index

Special Characters

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

X