IBM High Level Assembler for MVS & VM & VSE:

Benefiting from its Powerful New Features

SHARE 102, February 2004, Session 8165

February, 2004

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

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

Synopsis:

This document describes some of the most powerful and useful features of the IBM High Level Assembler for MVS & VM & VSE, and how they can simplify development and maintenance of Assembler Language programs.

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 102, February 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/Architecture	zSeries	OS/2	OS/2 Warp
DFSMS			

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:

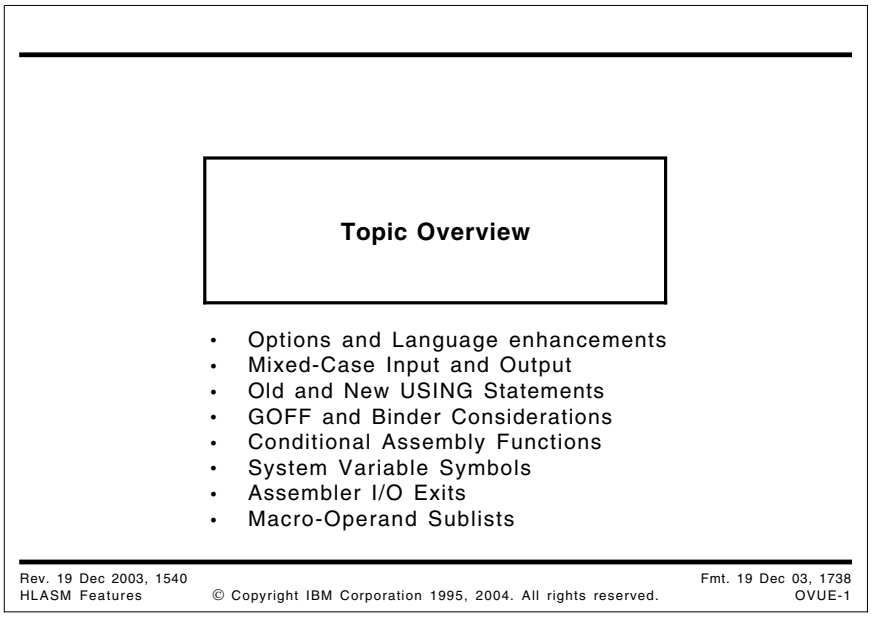
```
http://www.ibm.com/software/ad/hlasm/
```

(Revised 19 Dec 2003, 1540, Formatted 19 Dec 03, 1738.)

Contents

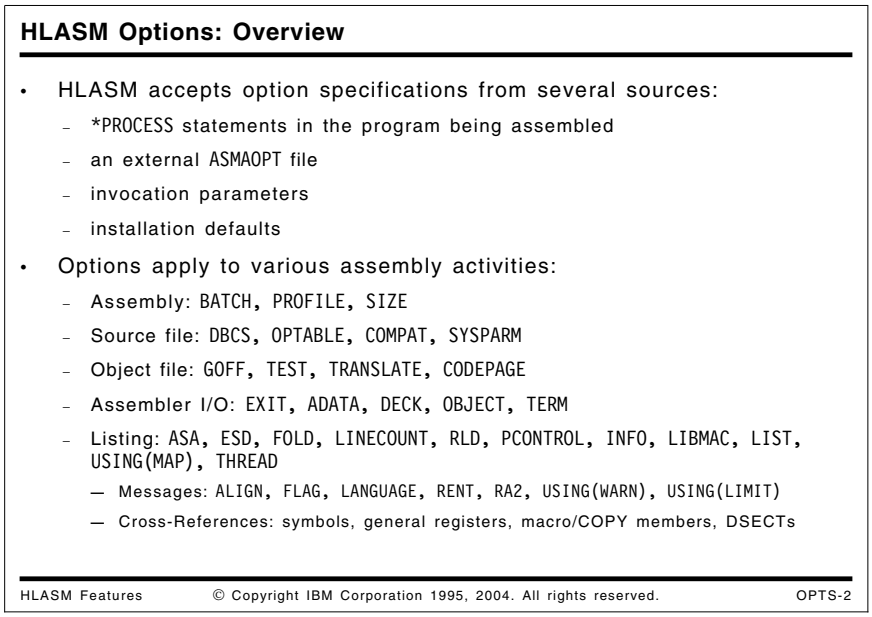
Figures

Overview



We will discuss the following topics:

- useful assembler options (“Assembler Options” on page 2)
- new and enhanced assembler instruction statements (“Useful Language Features” on page 8)
- considerations of mixed-case input and output (“Mixed-Case Input and Output” on page 17)
- a review of ordinary (“old”) USING statements (“Ordinary USING Statements” on page 21)
- new USING statements and their benefits (“New USING Statements” on page 41)
- features of the new object file format (GOFF) (“Generalized Object File Format (GOFF)” on page 91)
- conditional assembly functions (“Conditional-Assembly Functions” on page 93)
- system variable symbols (“System (&SYS) Variable Symbols” on page 103)
- assembler input-output exits (“Input-Output Exits” on page 107).



Many of these new and enhanced options have been mentioned earlier, in the context of the features they control. They are listed here for completeness.

How Options May Be Specified

The High Level Assembler accepts options from four sources:

1. Installation options are set at the time HLASM is placed on your system. They may be fixed — not capable of being overridden — if the installer chooses.
2. Invocation parameters from JCL PARMs or CMS command lines are the traditional and familiar way of providing assembler options.
3. HLASM supports an external ASMDOPT file for options. This can be very helpful if the length of the option string is too long to be conveniently specified by other means.
4. *PROCESS statements at the beginning of the source file to be assembled can “tailor” options to each assembly. The OVERRIDE operand allows you to specify certain options that may not be modified by other sources of options: for example, a program containing EBCDIC data in a national language like French can specify a CODEPAGE option such that that only the French-to-Unicode mapping tables may be used in creating Unicode constants.

Assembly Control Options

These options specify "global" assembly activities:

BATCH Multiple complete assembly files can be processed in a single invocation of the assembler.

PROFILE The assembler will retrieve statements from the member specified in this option, and process them at the head of the source file.

SIZE Controls the amount of storage allocated for the assembly.

Source File Control Options

These options specify how various parts of the source program should be treated:

DBCS If specified, G-type constants and self-defining terms are allowed, and DBCS data in C-type constants is translated correctly.

OPTABLE The assembler provides several tables of mnemonics and operation codes, allowing you to select which should be used for the assembly.

COMPAT For compatibility with code written for older assemblers, HLASM allows you to select either the "old" interpretation or processing, or the newer methods supported by HLASM.

SYSPARM The length of the string passed from the SYSPARM option to the &SYSPARM system variable symbol may be as long as 255 characters. This allows more flexibility in controlling conditional assembly, and is especially useful when supplied from an external options file (see "How Options May Be Specified" on page 2).

Object File Control Options

These options specify how various parts of the object file should be created:

GOFF If specified, the new "Generalized Object File Format" (GOFF) will be produced, rather than the traditional card-image object module (OBJ) format.

TEST The assembler can produce SYM records with old (OBJ) object files containing information about internal symbols used in the assembly. This option is incompatible with the GOFF option.

TRANSLATE The TRANSLATE option lets you specify that C-type character constants (but *not* C-type self-defining terms!) should be translated to a specified code page; the default is ASCII. This can simplify creating data to be passed to systems that do not support EBCDIC data.

CODEPAGE Many EBCDIC code pages have been defined to support national languages. Internationalization may require Unicode data; the CODEPAGE option and CU-type constants support the mapping of single-byte EBCDIC data to the two-byte Unicode format. &SYSPARM system variable symbol may be as long as 255 characters. This allows more flexibility in controlling conditional assembly, and is especially useful when supplied from an external options file (see "How Options May Be Specified" on page 2).

Assembler I/O Control Options

These options specify how the assembler's input/output activities should be handled:

EXIT HLASM allows you to supply I/O exits for all files except its work (utility) file. These exits can supplement or even replace the assembler's I/O, and can add, delete, and modify records, as well as providing messages to be added to the listing. You specify the type of exit, the name of the module, and optionally a “parameter” string to be passed to the exit routine.

ADATA The ADATA option causes HLASM to write information about every aspect of the assembly to the SYSADATA file. The file contains more information than may be present in the listing (because portions of the listing can be suppressed by other options), and the data is in a format intended for processing by other programs, rather than for legibility like the listing.

TERM The TERM option causes HLASM to write messages and erroneous statements to the terminal (or to a file, if specified). The information can be compressed (to avoid line wrap-around) if you specify the TERM(NARROW) option; TERM(WIDE) does not compress spaces.

DECK, OBJECT These two options determine whether and where the object file will be written. DECK specifies the SYSPUNCH file, and OBJECT specifies the SYSLIN file. These options, unlike the GOFF option, do not affect the *format* of the object module, only its destinations.

Listing Control Options

These options let you control various aspects of the assembly listing:

LIST The LIST option controls two things: whether or not the listing is produced (NOLIST suppresses it), and the length of listing lines. The LIST(121) option produces listings appropriate to the OBJ object file format; LIST(133) or LIST(MAX) is required when the GOFF option has been specified.

ASA The ASA option specifies that ANSI carriage-control characters should be used for lines in the listing file. If assembler listings are to be printed as part of a larger set of files (for example, the assembler is invoked dynamically by a program producing listings of its own), compatible carriage controls will help. If not specified, the assembler will provide “machine” carriage controls.

ESD, RLD These options control the listing of the External Symbol Dictionary and the Relocation Dictionary, respectively.

FOLD The FOLD option causes lower-case letters in the listing to be converted to upper case. This is needed only in countries where the code points assigned to lower case letters are used for national characters: if a listing containing lower case Latin letters was displayed on a terminal supporting national characters, the text might be difficult to read.

INFO The INFO option causes HLASM to display information about its current status. Because not all copies of HLASM will apply service at the same time, this option can help users of each copy determine which problems have been fixed, and what enhancements have been added.

THREAD In a multi-section assembly without the (default) NOTHREAD option, HLASM will start the location counter of each section on the next doubleword boundary. This causes extra work in determining section offsets, as the section origin must be subtracted. Specifying NOTHREAD causes HLASM to start each section at a zero origin.

LINECOUNT The number of lines per listing page may be zero (in which case all page ejects are suppressed), or a very large number.

PCONTROL The PCONTROL option lets you override settings of PRINT statements in the source program. Rather than modifying and reassembling a program in order to see otherwise invisible portions of the program, you can simply specify appropriate PCONTROL suboptions:

ON overrides PRINT OFF statements

GEN overrides PRINT NOGEN statements

DATA overrides PRINT NODATA statements

MCALL causes inner macro calls to be displayed

MSOURCE displays source statements generated by macros

UHEAD causes the “Usings in Effect” page heading to be displayed

If the scope of certain PCONTROL actions is too broad, it can be specified “dynamically” using the ACONTROL statement (see “Useful Language Features: New Ordinary-Assembly Statements” on page 8).

LIBMAC Library macros containing previously undetected errors may generate diagnostic messages that don't precisely identify the statements in the macro causing the problem. The LIBMAC option will cause library macros to be placed in the listing just before their first call, as though it had been defined “inline”. Any diagnostic messages will then identify the specific statement in error. The scope of the LIBMAC option is also controllabe with ACONTROL statements.

USING(MAP) The Using Map is a summary of all USING and DROP activity in the program, which can help with determining overlaps and USING ranges. This option causes the Using Map to be printed.

Message Control Options

These options allow you to control some of the assembler's error checking and diagnostic messages.

ALIGN The ALIGN option requests HLASM to check boundary alignment of data. NOALIGN suppresses the checks for non-privileged instructions, and will cause data not to be aligned if the DC/DS duplication factor is nonzero.

FLAG Seven suboptions are supported by the FLAG option to control continuation checking, substring checking, record/file identification for flagged statements, alignment checking, use of implied lengths, un-based references to low-storage addresses, improper substrings, and nonempty PUSH stacks.

- FLAG(ALIGN) checks for possibly incorrect or inefficient operand alignments.
- FLAG(CONT) controls checks for possible errors in coding continuation statements.
- FLAG(PAGE0) controls checks for possible inadvertent references to addresses in the first 4K bytes of storage.
- FLAG(IMPLEN) controls checks for possibly unintentional omission of the length specification in SS-type instructions.
- FLAG(PUSH) checks at the end of the assembly for a nonempty PUSH stack.

- FLAG(RECORD) causes HLASM to identify the name of the source file and the relative record number (from that file) of the statement with which another message is associated.
- FLAG(SUBSTR) controls checks for possible errors in coding conditional assembly substring notation.

LANGUAGE In addition to English-language messages, HLASM supports German, Spanish, and Japanese.

RENT The RENT option asks HLASM to check for possible violation of reentrancy due to apparent stores into CSECTs.

RA2 Sometimes HLASM is used as a cross-assembler for systems supporting 16-bit address constants. The RA2 option permits relocatable 2-byte address constants.

USING(WARN) HLASM can detect several possible errors in specifying USING statement operands. The four sub-options control checking for nullified USINGs, nonzero base addresses based on register zero, overlapping resolutions, and displacements exceeding a supplied limit value.

USING(LIMIT) The USING(LIMIT) option provides a value to be compared to each implicitly calculated displacement: if the displacement is larger and the USING(WARN(8)) value is specified, HLASM will issue a warning.

Cross Reference Control Options

These options specify

XREF This option controls the production of the ordinary symbol cross-reference. Two sub-options allow you to retain only referenced symbols, and to display unreferenced symbols defined in CSECTs and RSECTs.

RXREF The RXREF option causes HLASM to create a cross-reference of all general register usage, including implicitly referenced registers.

MXREF The MXREF option controls cross-referencing of macros and COPY members from library files. The MXREF listing contains information about the sources of each member, and also where each is referenced in the program.

DXREF The DXREF option causes HLASM to list all DSECTs defined in the program, their length and relocation ID, and where their definitions begin.

Installation Options

At the time HLASM is installed, you may choose default options to be used for each assembly. Some options may be “fixed”, so that they may not be overridden at assembly time by other option sources.

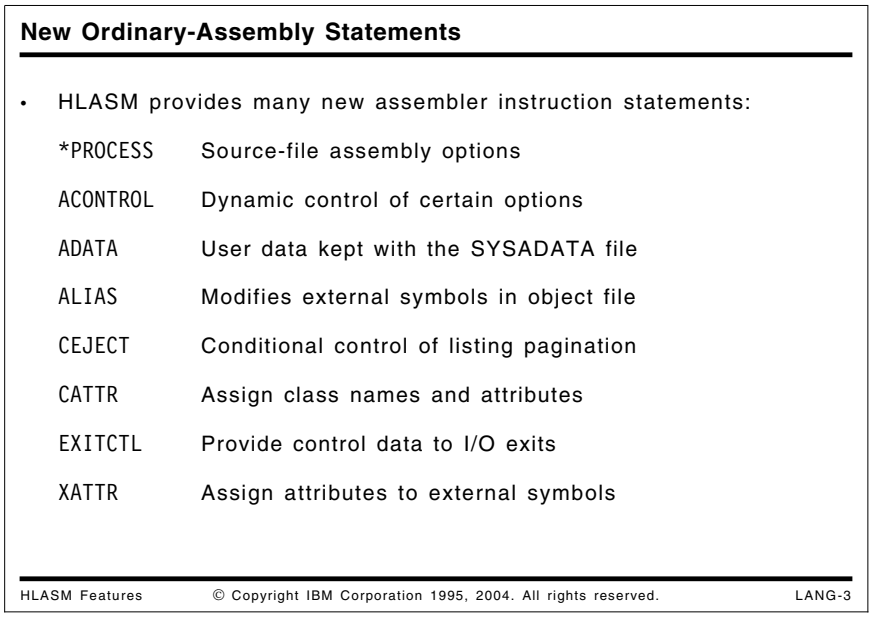
PESTOP Option

By providing the PESTOP option during installation, you may specify that errors in specifying options should cause the assembly to be suppressed. This can help save time and system resources by avoiding the need for reassembling programs with correct options.

Options From Old Assemblers

Some options supported by Assembler H (IEV90), the DOS/VSE Assembler, or Assembler XF (IFOX00) are either not present in HLASM, or are supported in different ways:

- ALOGIC (in XF; not in HLASM)
- EDECK (DOS/VSE only; VSE/ESA provides an I/O exit to support E-decks)
- LINK (in DOS/VSE)
- MCALL (in XF; different form in HLASM)
- MLOGIC (not in HLASM)
- NUM (CMS only; in H only)
- PRINT (CMS only)
- STMT (CMS only; in H only)
- SUBLIB (DOS/VSE and HLASM on VSE)
- SYSPARM (via // `OPTION` JCL statement in DOS/VSE)
- SXREF (in XF; same as XREF(SHORT) option in HLASM)
- WORKSIZE (XF only; CMS only)
- YFLAG (in XF only; same as RA2 option in HLASM)



Useful Language Features: New Ordinary-Assembly Statements

High Level Assembler introduces statements not available in previous assemblers, to provide additional function and flexibility.

*PROCESS — *PROCESS statements are a special form of comment statement at the beginning of a source program, specifying options for the program.

*PROCESS statements may include an OVERRIDE operand: any items specified in the OVERRIDE list may not be modified by invocation options. If your program must be assembled with certain options, you can specify them on *PROCESS OVERRIDE statements.

Note: Programs might contain what appear to be *PROCESS statements, but which are actually comments that might be misinterpreted. High Level Assembler can be prevented from treating such comments as *PROCESS statements by inserting any valid statement (including a comment statement not resembling *PROCESS) ahead of any such “apparent” *PROCESS statement.

ACONTROL — The ACONTROL statement allows you to dynamically change the settings of certain assembler options. For example, you can request that High Level Assembler check for possible continuation statement errors as an invocation option, and then turn checking off and on around specific sets of statements.

ADATA — The ADATA statement allows you (or other creators of source files, such as editors, preprocessors, and the like) to insert information into files to be processed by the assembler. The data specified in these ADATA statements will be captured by the assembler and placed into the SYSADATA output stream for use by other tools and processors.

ALIAS — The ALIAS statement for external symbols permits assembler output modules to be linked with those from other languages whose external symbols contain characters that would otherwise be invalid in the “normal” assembler lan-

guage, or which would have been automatically translated to upper case characters by the assembler.

ALIAS causes a “normal” external symbol already defined by the program to be given a different name (its “alias”) in the External Symbol Dictionary. The ESD listing provides additional information relating to the external symbol substitutions specified by ALIAS statements.

Note: Remember that only one instance of a symbol is allowed; thus, symbols 'aa' and 'AA' are not both supported.

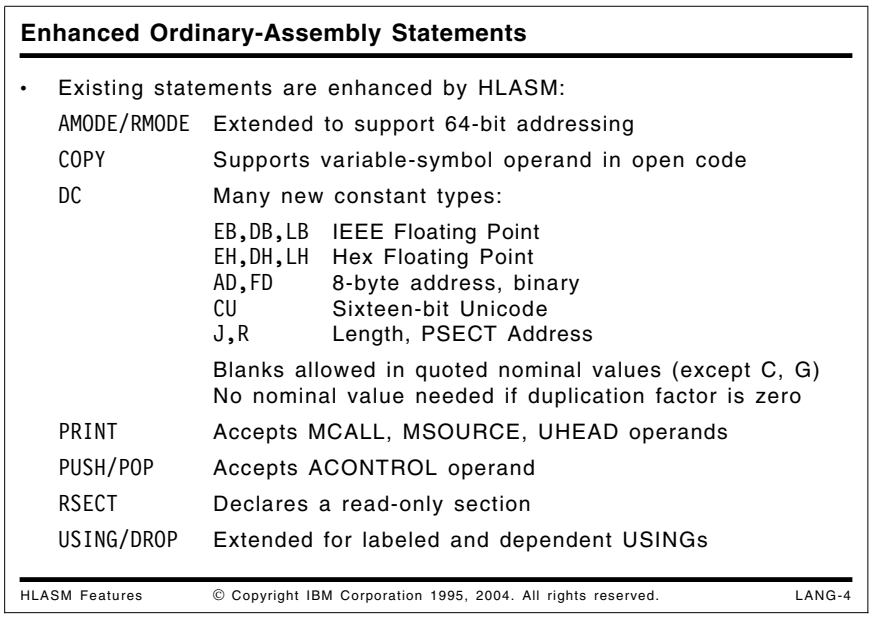
CATTR The CATTR statement defines the “class” into which subsequent text or external symbol definitions will be placed. CATTR requires the GOFF option, and the information will be processed by the DFSMS/MVS* Binder. This topic is discussed further in “Generalized Object File Format (GOFF)” on page 91.

CEJECT The CEJECT “conditional page-EJECT” statement permits automatic determination of the amount of space remaining on a listing page, with a page skip occurring if less than the requested number of lines remains. This relieves you of the necessity of frequently adjusting lines to determine where to put EJECT statements; blocks of statements can be kept together on a page even if preceding statements are added or removed.

EXITCTL The EXITCTL statement passes information from the program being assembled to I/O exit routines, to give you greater flexibility in managing the behavior of I/O exits. (An example of the use of the EXITCTL statement is given at “The EXITCTL Statement” on page 111.)

XATTR The XATTR statement assigns special attributes to external symbols. If you are using special features of the Binder, or establish linkages to C/C++ functions, this statement may be required. XATTR requires the GOFF option. This topic is discussed further in “Generalized Object File Format (GOFF)” on page 91.

Each of these new assembler instructions may also be used as a model statement in macro definitions.



Useful Language Features: Enhanced Statements

Several assembler instruction statements supported by previous assemblers have been enhanced in High Level Assembler:

AMODE/RMODE — The AMODE and RMODE statements have been extended to support 64-bit addressing requirements.

COPY — The COPY statement is enhanced to support a variable symbol in the operand field if the statement is used in open code. For example:

```
&MEMBER  SetC  'MyDefs'          Assign 'MyDefs' to &MEMBER
         - - -
         COPY  &MEMBER
```

causes the assembler to first assign the string MyDefs to the variable symbol &MEMBER, causing the assembler to copy the contents of library member MyDefs into the source program. (Note that this technique doesn't work in macros.)

DC/DS — The DC and DS statements have been extended in many ways:

Floating Point Constants

HLASM converts decimal values to IEEE binary floating point (with five choices of rounding mode) for constants of types EB, ED, and ED.

The introduction of new conversion routines to support IEEE-format data allows HLASM to provide improved and directed-rounding conversion of hexadecimal floating point data for constants of types EH, DH, and LH. In rare cases the improved conversion may cause a one-bit difference in the generated constant.

Floating Point Symbolic Constants

HLASM supports symbolic forms for special floating point values. These three are supported for hex and binary:

MAX Maximum normalized value
MIN Minimum normalized value

DMIN Smallest nonzero denormalized value

These four are supported only for IEEE binary values:
INF Infinity
NAN Not-a-Number (same as QNAN)
SNAN Signaling Not-a-Number
QNAN Quiet Not-a-Number

All forms may be signed.

Unicode Constants
16-bit Unicode values are generated from CU-type constants. The mapping is determined by the CODEPAGE opion.

Address Constants
Two new address constant types are provided:

J Length (generalization of CXD)
R PSECT address

64-bit Constants
Two new 64-bit constant types are provided:

AD 8-byte address
FD 8-byte fixed-point binary

Spaces in Nominal Values
The nominal value of quote-delimited constants may contain extra spaces for improved readability. (This excludes C-type and G-type constants, of course.)

Zero Duplication Factor
In DC statements, a nominal value is not required if the duplication factor is zero.

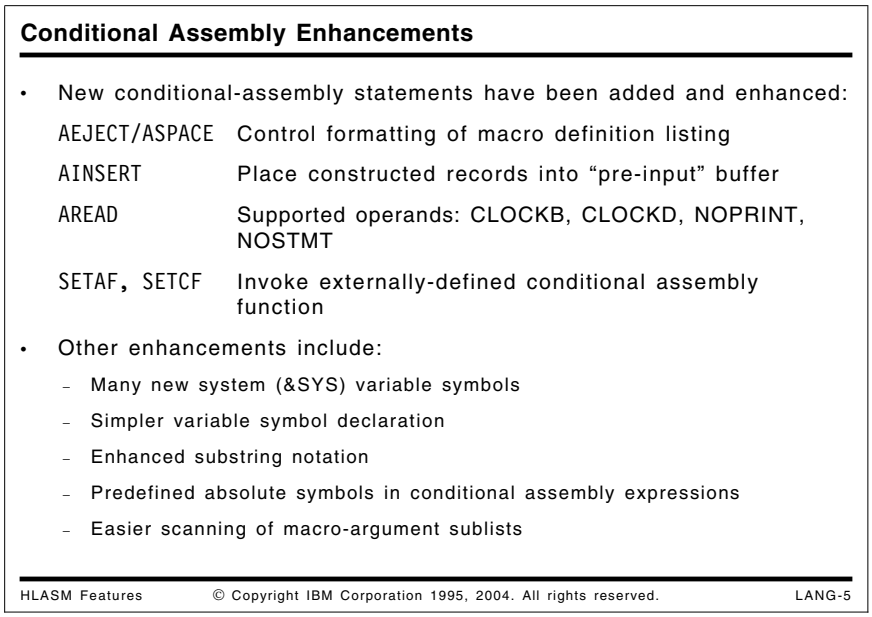
PUSH/POP The PUSH and POP statements have been extended to save and restore the current status of ACONTROL statements.

PRINT The PRINT statement supports three additional operands:

MCALL 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; but inner macro calls are not normally shown. The MCALL operand causes inner calls to be displayed. (This can also be achieved with the PCONTROL(MCALL) option.)
MSOURCE Normally, HLASM displays the source statements and their generated code. To retain the code but suppress the statement, specify the NOMSOURCE operand.
UHEAD To suppress the "Usings In Effect" page heading, specify the NOUHEAD operand.

RSECT The RSECT statement was supported in Assembler H (without documentation), but only to the extent of placing a special flag in the External Symbol Dictionary for the name of the control section. HLASM extends the support by checking the instructions in the designated control section for possible violations of program reentrancy. This is done on a per-section basis, and is independent of the setting of the RENT option.

USING/DROP The USING statement supports several powerful extensions that can greatly clarify and simplify coding that refers to complex data structures. Corresponding extensions were made to DROP. These are discussed in detail at "New USING Statements" on page 41.



Useful Language Features: Conditional Assembly Enhancements

High Level Assembler has added several assembler instruction statements, many new system variable symbols, and other enhancements that add power and flexibility to the conditional assembly language.

New Conditional Assembly Statements

New conditional assembly statements include:

AEJECT This macro-definition-time operation causes a page skip in the assembly listing of the macro definition.

ASPACE This macro-definition-time operation causes spacing of one or more lines in the assembly listing of the macro definition. (ASPACE and AEJECT are not model statements: when the macro is called, the ASPACE and AEJECT statements do not appear in the generated code.)

AINSERT This instruction inserts a record into an internal input buffer that will be read (until the buffer is emptied) in place of the primary input stream; input will then resume from the primary input. Its most powerful applications are in macros.

AREAD An AREAD statement with no operands causes the 80 bytes of the next available record in the primary input stream to be assigned to a character variable symbol, rather than being scanned during normal statement processing.

HLASM supports four AREAD statement operands for controlling statement printing, and for obtaining current time information:

- The CLOCKB and CLOCKD operands return binary and decimal time information, respectively. (They were available but not documented in Assembler H Version 2.1.) These contain *current* time values, unlike the &SYSTIME and &SYSCLOCK system variable symbols. (&SYSTIME returns the time at which the assembly started and does not vary during the assembly; &SYSCLOCK returns the time at which the macro expansion began, and does not vary during the macro expansion.)

  The NOPRINT operand suppresses the printing of the AREAD statement, and the NOSTMT operand suppresses the printing of the source record that was “read” by AREAD.

SETAF This conditional assembly operation causes the assembler to invoke an external arithmetic-valued function, and assign its value to an arithmetic variable symbol.

SETCF This conditional assembly operation causes the assembler to invoke an external character-valued function, and assign its value to a character variable symbol. (SETAF and SETCF are discussed in greater detail in “Conditional-Assembly Functions” on page 93.)

Other Conditional Assembly Enhancements

Other useful enhancements to the conditional assembly language include:

System Variable Symbols
HLASM greatly expands the number of system variable symbols available to your programs. A summary is provided at “System (&SYS) Variable Symbols” on page 103.

Variable Symbol Declarations
No ampersand is required in declarations of variable symbols in LCLx and GBLx statements. For example, these two declarations are equivalent:

```
LCLA  &X,&Y,&ZZZ
LCLA  X,Y,ZZZ
```

Conditional Assembly Substrings
Previous assemblers handled improper substring expressions inconsistently, sometimes providing the expected diagnostics and sometimes not. This occasionally required awkward coding; 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, specify the FLAG(NOSUBSTR) option.

A better approach is to specify the FLAG(SUBSTR) option, and also use the explicit “remainder of string” notation:

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

This allows the assembler to diagnose “true” errors in specifying substrings.

Absolute Symbols in Conditional Assembly Statements
Predefined absolute symbols are permitted in many conditional assembly contexts where they were not allowed by earlier assemblers, such as in arithmetic expressions in SETA and AIF statements.

Substituted (&SYSLIST and SETC) Sublists
In previous assemblers, character strings substituted as arguments of calls to inner macro were treated only as character strings, independent of their actual structure. HLASM permits such substituted operands to be treated as parameters having a list structure accessible through the normal subscripting and &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 subscript notation. This means that macros need not be written differently depending on whether they are invoked as “outer” or “inner” macros.

If HLASM should treat such operands and SETC variables as in previous assemblers, specify the COMPAT(SYSLIST) option. This option determines whether HLASM will or will not match the list-handling behavior of previous assemblers such as Assembler H. (The assembler's handling of macro arguments in list format is rarely a concern, but there are cases where macros can be written much more simply if you can utilize HLASM's ability to handle lists more uniformly than could past assemblers.)

Two types of lists are passed as arguments to macros:

1. a positional argument list, and

2. a parenthesized list of terms passed as a single argument.

For example, a positional argument list of four arguments (A, B, C, and D) appears in the call

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

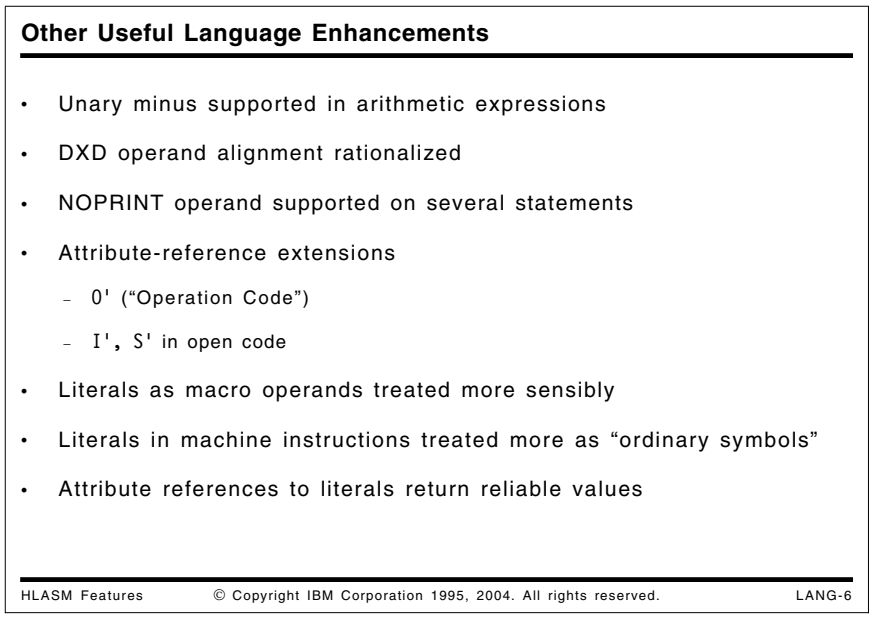
and these may be treated as a list through references in the macro to the &SYSLIST system variable symbol. A list of terms passed as a single argument appears in the call

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

where only one argument is passed (that is, (A,B,C,D) is a list of four elements). If these lists are passed to an inner macro as one argument (A,B,C,D), the inner macro's scanning may be simpler if the NOCOMPAT(SYSLIST) option is specified.

Macro-Call Name Field Operands

The name field (“label”) entry of a macro call need not be a symbol. This allows greater freedom in passing arguments to macros.



Useful Language Features: Other Enhancements

Other potentially useful enhancements to High Level Assembler include:

Unary Minus
HLASM supports arithmetic unary minus operations in both ordinary and conditional assembly arithmetic expressions.

DXD Statement
Previous assemblers assigned alignment requirements to dummy external sections (DXDs) based on the first operand. HLASM corrects this behavior by assigning the most stringent alignment requirement among all the operands.

NOPRINT Operand Extensions
The NOPRINT operand is supported by the AREAD, PRINT, PUSH, and POP statements, and suppresses the appearance of the statement itself on the listing. This 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” of code for other hardware architectures.

Attribute Notation Extensions
HLASM 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'); conditional assembly allowed the use of references to the Length, Type, Scale, Integer, Count, Number, and Definition attributes of variable symbols. High Level Assembler permits the use of Scale (S') and Integer (I') Attribute references in open code.

High Level Assembler also defines a new conditional-assembly “Operation Code” Attribute Reference (0'), which can be used to test for definition of operation code mnemonics.

One effect of these extensions is that code containing character strings that appear to HLASM to be attribute references may not have been treated that way by earlier assemblers; this may cause certain statements to be flagged, or to be interpreted differently.

Literals in Macro Operands

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 provides the actual type if it can be determined. 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.

The Assembler H documentation states that the evaluated operand in an attribute reference must be a symbol (except for type attribute references). However, it actually evaluated attributes of strings containing expressions and other objects, using the first symbol. The High Level Assembler enforces the previously documented rules. One consequence of this enforcement is that attribute evaluations of expressions that previously returned a “valid” type (that is, not 'U') will now return 'U'.

Literals as Relocatable Terms

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. Thus, these two statements are valid:

```
TR   StringToHex,=C'0123456789ABCDEF'-C'0'        Printable Hex
IC   0,=AL1(0,1,1,2,1,2,2,3,1,2,2,3,2,3,3,3,4)(4) No. of 1-bits
```

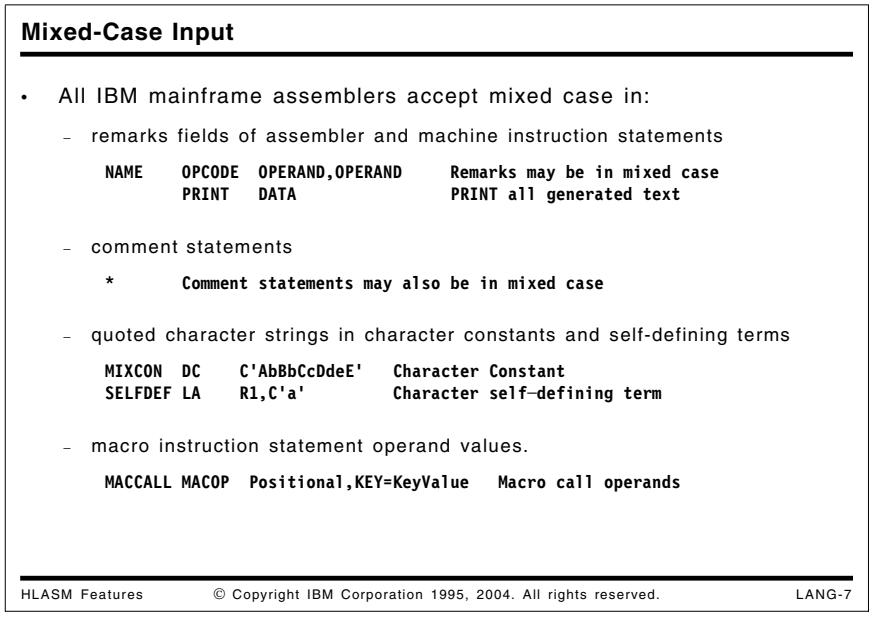
Previous assemblers required that the literal be the only term in the operand, and indexing was not allowed.

Attribute References to Literals

Attribute references to previously-defined literals formerly gave results that were different from later references, after the literal was defined. HLASM now returns a uniform value of the type attribute for all references.

Mixed-Case Input and Output

High Level Assembler can accept its input statements coded either in uppercase characters (for compatibility) or in mixed lower and uppercase characters. Similarly, the assembler's listing file can print records in mixed case, or only as uppercase characters.



Mixed-Case Input

High Level Assembler and all previous IBM assemblers for the System/360/370/390 family of processors accept mixed upper-case and lowercase letters in certain contexts:

- remarks fields of assembler and machine instruction statements

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

- comment statements

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

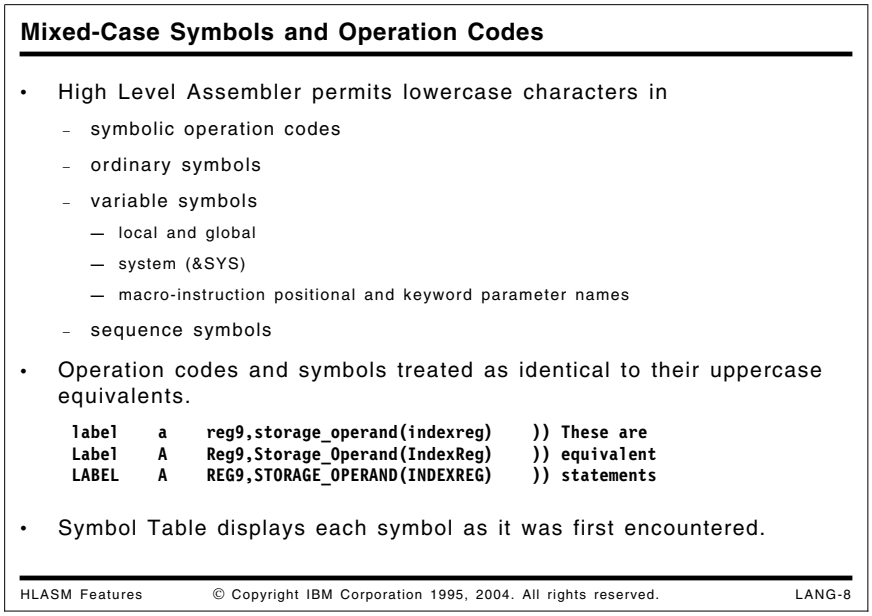
- quoted character strings such as character self-defining terms and C-type operands in DC and DS statements

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

- macro instruction statement operand values.

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

High Level Assembler extends the use of lowercase letters to operation codes and to symbols of all types.



Operation Codes and Symbols

The High Level Assembler permits you to use lowercase characters in contexts where they were not previously allowed:

- symbolic operation codes
- ordinary symbols
- variable symbols
  - local
  - global
  - system (&SYS)
  - macro-instruction positional parameters
  - macro-instruction keyword parameters
- sequence symbols

Internally, all such operation codes and symbols are treated as identical to their uppercase equivalents. Thus, the following three statements are identical:

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

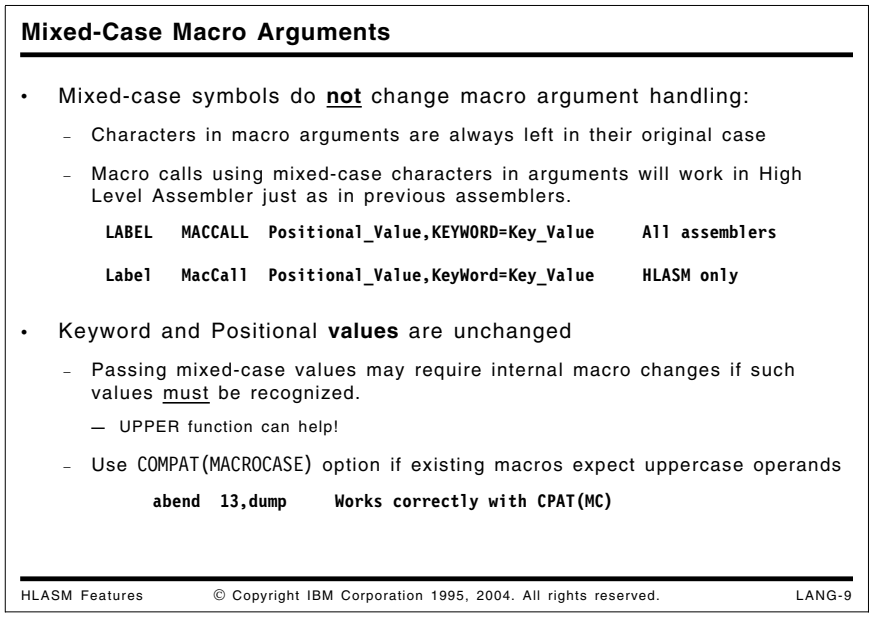
In the Symbol Table listing, each symbol is displayed in the form in which it was first encountered and entered into the symbol table. Thus, if the first recognition of the symbols occurred in the first of the three statements above, the symbols `label`, `reg9`, `storage_operand`, and `indexreg` would appear in the Symbol Table listing in lower case.

The COMPAT(CASE) Option

If you must maintain compatibility with previous assemblers, the option

```
COMPAT(CASE)
```

causes HLASM to recognize symbols and operation codes only when they are entered as uppercase characters.



Macro Arguments

Assemblers for the System/360/370/390 family of processors have always left positional and keyword macro arguments in the form in which they were entered. If the macro must scan the argument characters, either

1. the arguments must be entered in upper case only, or
2. the arguments can be “forced” to upper case before macro expansion by specifying the COMPAT(MACROCASE) option, or
3. the internal conditional-assembly function UPPER may be used inside the macro to convert strings to uppercase letters, or
4. the internal scanning may have to handle case sensitivity.

Because the character-handling capabilities of the conditional assembly language are considerably improved in High Level Assembler, it is not necessary to require that scanned macro arguments be entered in uppercase only. If the argument string is to be substituted without scanning, then no case conversion is required.

The availability of mixed-case symbols in High Level Assembler makes no changes in the ways arguments to macros are handled. Unquoted macro arguments are normally passed to the macro expansion unchanged, which has required (in most cases) that such arguments be written completely in uppercase letters. For example, you might have written

```
File_Error  AbEnd  2,DUMP
```

Because HLASM supports mixed-case symbols and operation codes, it is natural to write other parts of a program using mixed-case text. However, some macros may have been

written to accept only uppercase arguments; to help preserve your investment in such macros, specify the COMPAT(MACROCASE) option. This causes unquoted arguments to be converted internally to uppercase before macro expansion begins. For example, you could specify the COMPAT(MACROCASE) option and write

```
File_Error  AbEnd  2,Dump
```

and the assembler will pass the “uppercased” argument DUMP to the macro expansion.

The FOLD Option

The FOLD option is provided by High Level Assembler to let 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.

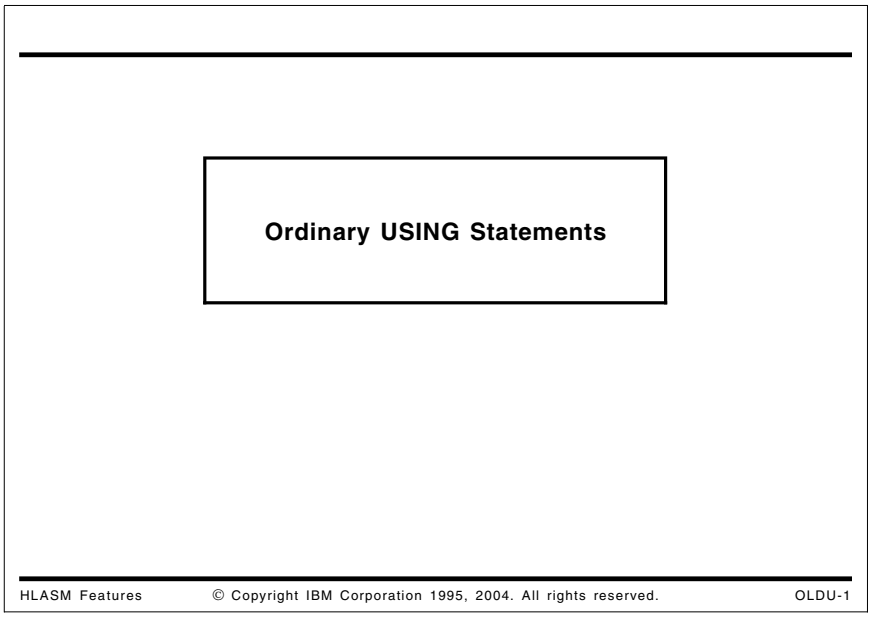
The case of messages and text sent to the SYSTERM file (normally, the terminal) is not affected by the FOLD option.

Because both of these options affect High Level Assembler's treatment of the case of character text, it is worth noting that their effects are independent of each other.

- COMPAT(CASE) affects only the recognition of symbols and operation-code mnemonics in *input* text. If COMPAT(CASE) is specified, High Level Assembler will recognize symbols and operation codes only if they are entered in uppercase.
- FOLD affects only the production of the *output* listing. All alphabetic characters (whatever their case as entered) are converted to uppercase in the listing file, but this has no effect on the assembler's recognition and treatment of character case when the statements are scanned.

  Note: The FOLD option may obscure the visibility of lower-case characters in the program. The object code listing will show that correct values have been generated from lower-case characters.

Ordinary USING Statements



Addressing and USING Statements: A Review

If you are familiar with the way the System/360/370/390 family of machines generates storage addresses, and with the usage rules, purpose, and function of the Assembler Language USING statement, feel free to skip ahead to “New USING Statements” on page 41. (However, you might find that a quick reading of this section will both refresh your knowledge and establish a better familiarity with the terminology to be used in later sections.)

Before discussing the High Level Assembler's treatment of USING statements, we will first review the fundamental mechanisms of address generation used in the System/360/370/390 family of processors.

The addressing technique used in System/360/370/390 processors differs from that found in many earlier computers, where the *actual* storage address (or addresses) of the operand (or operands) was part of the instruction:

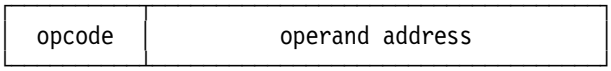


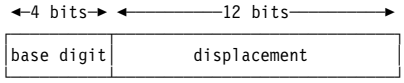
Figure 1. Typical Instruction Format

Because System/370/390 allows very large amounts of addressable central storage (2^{31} bytes, or more), the technique of placing actual addresses into the instructions would require a field at least 31 bits wide for each such address. Since few programs need as much as 2^{31} bytes of memory to execute, many of the bits in the address field would be wasted by such a direct-addressing technique.

In the System/360/370/390 machines, the scheme used for addressing memory operands is more economical in the number of bits allotted to each instruction, but therefore more expensive in terms of the computation needed to determine operand addresses.

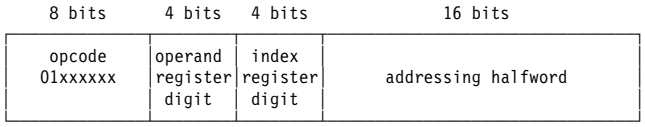
Addressing Halfwords and Effective Addresses

- Many instructions generate addresses from *addressing halfwords*:



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

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



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

The Addressing Halfword

The System/360/370/390 family of processors provides several *modes* of addressing. For the purposes of this review we will discuss only one, in which 31-bit addresses are generated.

To refer to items in processor storage such as data or instructions, the program will almost always make use of one of the general purpose registers. This is due to the way the processor uses the information in a portion of an instruction called an “addressing halfword”, which always occupies a correctly aligned halfword in memory.

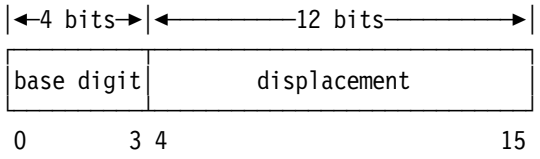


Figure 2. Structure of an Addressing Halfword

The 4-bit field at the left of the addressing halfword contains a single hex digit (called the *base register specification digit,* or *base digit)*. It can take values from 0 to 15, and specifies a general purpose register. The 12-bit field in the rest of the addressing halfword contains an unsigned (and therefore non-negative) number called the *displacement* which can take values from 0 to 4095.

Effective Addresses

To generate the address of an operand, the processor does the following:

Step 1: The 12-bit displacement is put at the right-hand end of an internal register called the Effective Address Register (abbreviated EAR), and the leftmost 19 bits of the EAR are cleared to zeros.

Step 2a: If the base register specification digit is *not* zero, then the rightmost 31 bits of the specified general purpose register are added to the contents of the Effective Address Register, and carries out the left end of the EAR are ignored. (The high-

order bit of the general purpose register is ignored also.) The register used is called the *base register,* and the quantity in its rightmost 31 bits is called the *base address* or *base.*

Step 2b: If the base register specification digit *is* zero, nothing is added to the EAR. Thus, R0 will *never* be used as a base register.

The resulting quantity in the EAR is called the *effective address.* It may be used as the address of an operand in memory, as well as for other purposes such as a shift count. This method of generating addresses is called *base-displacement addressing.*

Examples of Effective Addresses

1. Suppose the addressing halfword of an instruction is 1011 001011010101 in binary (or X'B2D5' in hex) and suppose that the contents of general purpose register 11 is

   1100 0111 0011 1110 1001 0000 1010 1111

   in binary (or X'C73E90AF' in hex). Then the effective address of the instruction is (giving both binary and hex arithmetic):

   ```
   000 0000 0000 0000 0000 0010 1101 0101  000002D5 (displacement)
   100 0111 0011 1110 1001 0000 1010 1111  473E90AF (base)
   100 0111 0011 1110 1001 0011 1000 0100  473E9384 (effective address)
   ```

2. Suppose the addressing halfword of the same instruction is X'0468'. Then the effective address is X'00000468', since R0 cannot be used as a base register.

3. Suppose the addressing halfword of the same instruction is X'B000', and the contents of R11 is as before. Then the effective address is X'473E90AF'; a zero displacement is quite acceptable.

Indexing

After the displacement has been added to the base (if any), the processor checks the type of the instruction. If the instruction is type RX, a further indexing cycle is needed. The second byte of an RX-type instruction contains two four-bit fields, the second of which is called the *index register specification digit,* or *index digit:*

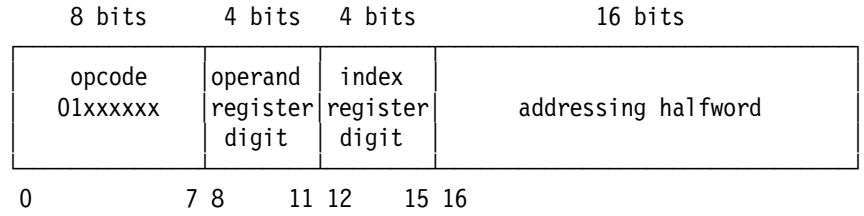


Figure 3. RX Instruction, showing Index Register Specification Digit

Step 3: If the instruction is type RX, *and* the 4-bit index register specification digit is *not* zero, then the rightmost 31 bits of the general purpose register specified by the index register specification digit are added (again ignoring carries out the left end) to the contents of the EAR.

The resulting quantity in the EAR is still called the *effective address.* (Sometimes it is called the *indexed effective address,* but the greater precision of this term is rarely needed.) The index register specification digit is sometimes called the *index digit;* similarly, the specified register is called the *index register,* and the quantity in its rightmost 31 bits is called the *index.*

Examples of Indexing

1. Suppose an RX-type instruction is X'430A7468' and that the contents of R7 is X'12345678' and the contents of R10 is X'FEDCBA98'. (Note that the base register specification digit, namely 7, means that R7 will be used as the source of the base address.) Then the effective address is

   ```
   000 0000 0000 0000 0000 0100 0110 1000  00000468 (displacement)
   001 0010 0011 0100 0101 0110 0111 1000  12345678 (base, from R7)
   001 0010 0011 0100 0101 1010 1110 0000  12345AE0
   111 1110 1101 1100 1011 1010 1001 1000  7EDCBA98 (index, from R10)
   001 0001 0001 0001 0001 0101 0111 1000  11111578 (effective address)
   ```

   The carry off the left end is ignored.

2. Suppose an RX-type instruction is X'43007468' and that the contents of register 7 is again X'12345678'. Then the effective address is

   ```
   000 0000 0000 0000 0000 0100 0110 1000  00000468 (displacement)
   001 0010 0011 0100 0101 0110 0111 1000  12345678 (base)
   001 0010 0011 0100 0101 1010 1110 0000  12345AE0 (effective address)
   ```

   No indexing cycle is needed because the index register specification digit is zero.

3. Suppose an RX-type instruction is X'43070468' and that the contents of register 7 is still X'12345678'. Then the effective address is

   ```
   000 0000 0000 0000 0000 0100 0110 1000  00000468 (displacement)
   000 0000 0000 0000 0000 0000 0000 0000  00000000 (base)
   000 0000 0000 0000 0000 0100 0110 1000  00000468
   001 0010 0011 0100 0101 0110 0111 1000  12345678 (index)
   001 0010 0011 0100 0101 1010 1110 0000  12345AE0 (effective address)
   ```

   In this example the values of the base and index register specification digits were interchanged from those in the preceding example, so that an indexing cycle was required to compute the same effective address.

Addressing Problems

Because the only part of storage which can be addressed *without* the use of a base register is the area with addresses 0 to 4095 = X'FFF', the programmer will almost invariably be required to use a base register to refer to operands in storage. This means that if we want to access a byte at address Q, there must be a base register available (that is, one of registers 1 to 15) which contains a number between Q−4095 and Q, since we can then generate an effective address Q by using a displacement between 0 and 4095. If there is no such number in a register, then the byte at Q is *not addressable.* Thus, if all the general registers contain zero, only the first 4096 bytes of memory are addressable!

Deriving the USING Statement

Understanding the USING statement is fundamental to writing Assembler Language programs for System/360/370/390 systems. In “Addressing and USING Statements: A Review” on page 21 we saw how the processor at *execution* time converts addressing halfwords into effective addresses. In this section, we will see how the Assembler performs the “reverse” process, deriving addressing halfwords from the values of symbolic expressions at *assembly* time.

Rather than give a set of rules and recipes and later explain how they work, we will start with a program that works in a known way.

The BASR Instruction

The Branch and Save (Register) RR-type instruction with mnemonic BASR is central to establishing addressability. For the time being, we will be interested only in the situation where we write

```
BASR   r1,0
```

(so that the second operand register specification digit r2 is zero). The effect of this instruction *when executed* is to replace the contents of the general purpose register specified by r1 by the right half of the Program Status Word (PSW): the rightmost 31 bits contain the value of the Instruction Address (IA). This address will be the address of the instruction *following* the BASR, because the IA is incremented by the instruction length (2 bytes for BASR) during the fetch portion of the instruction cycle.

Suppose the following short sequence of statements is part of a program which has been assembled and placed in storage to be executed. While we are giving the Assembler Language *statements* in Figure 4 below, the actual contents of storage will be hexadecimal data in the form of *instructions,* as illustrated in Figure 5 on page 26. Assume for the moment that the Supervisor has relocated the program so that the first instruction (the BASR) happens to be at storage address X'5000'.

```
Address   Name   Operation  Operand     Comment

          *      Example of a simple program
  5000           BASR       6,0         Establish base address
  5002    BEGIN  L          2,N         Load contents of N into R2
  5006           A          2,ONE       Add contents of ONE
  500A           ST         2,N         Store contents of R2 into N
        --twenty-two additional bytes of instructions, data, etc.--
  5024    N      DC         F'8'        Fullword integer 8
  5028    ONE    DC         F'1'        Fullword integer 1
```

Figure 4. A Simple Program Segment

While the actual functioning of these statements (other than BASR) is irrelevant to this discussion, a brief explanation may be helpful. The instructions L, A, and ST respectively (1) take a copy of the contents of a fullword area of storage and put it into a general register (i.e., **L**oad the register), (2) **A**dd a copy of the contents of a fullword area of storage to the contents of a register, and (3) replace the contents of a fullword area in storage by a copy of the contents of a general register (i.e., **ST**ore the register). The DC (**D**efine **C**onstant) statements are assembler instruction statements that provide two fullword areas of storage with names “N” and “ONE”, and which contain the fullword integer values desired. We have arbitrarily set the contents of the fullword at N to the integer 8, even though any value might be possible in an actual program. All of these instructions will be explained in more detail later.

Computing Displacements

When the program has been allowed to start execution, and after the BASR has been executed, R6 will contain X'00005002'. Remember: BASR places the address of the *next* instruction into the register designated by r1. We can now use the address in R6 as a *base address* for the instructions following the BASR; thus the base register specification digit in subsequent addressing halfwords should be 6. To determine the proper displacement in the L instruction at X'5002', we can use the known contents of R6 (X'00005002'). Since we know the address of the fullword area named N, we can now compute a displacement:

X'00005024' – X'00005002' = X'022'.

Then, the assembled machine instruction (using the operation code X'58' for the mnemonic L) will be X'58206022'. When this instruction is executed, the computation of the effective address yields

X'022' + X'00005002' = X'00005024',

which is the address we want!

If we continue in this fashion for the rest of the statements, we find that the following "assembled" machine language instructions, at the indicated storage addresses, will give the desired results at execution time. That is, after program loading is complete, we want the storage areas starting at address X'5000' to contain the (hexadecimal) data shown under "Assembled Contents".

```
Address   Assembled Contents    Original Statement

  5000    0D60                          BASR     6,0
  5002    58206022              BEGIN   L        2,N
  5006    5A206026                      A        2,ONE
  500A    50206022                      ST       2,N
                                --------------------------------
  5024    00000008              N       DC       F'8'
  5028    00000001              ONE     DC       F'1'
```

Figure 5. Simple Program Segment with Assembled Contents

Remember that when the Assembler processes the BASR statement and produces two bytes of machine language code containing X'0D60', nothing is "in" R6. It is only when this machine language instruction is finally *executed* by the processor that the desired base address will be placed in R6.

So far, so good: we have constructed a sequence of instructions which will give a desired result if it is placed in storage at exactly the right place. It is natural to ask "What would happen if the program is put elsewhere by the Supervisor?"

So, assume now that the same program segment begins at storage address X'84E8', as in the figure below.

```
Address          Statement

  84E8                  BASR     6,0
  84EA           BEGIN  L        2,N
  84EE                  A        2,ONE
  84F2                  ST       2,N
                 --- the same 22 bytes of odds and ends ---
  850C           N      DC       F'8'
  8510           ONE    DC       F'1'
```

Figure 6. Same Program Segment, Different Storage Addresses

In this case, the contents of R6 after the BASR is executed would be X'000084EA'. To access the contents of the fullword at N, using R6 as a base register, the necessary displacement is

X'0000850C' – X'000084EA' = X'022'.

Similarly, the displacement necessary in the "A" instruction is

X'00008510' – X'000084EA' = X'026'.

Thus the assembled program would appear in storage as shown in the figure below.

```
Address         Assembled Contents

  84E8          0D60
  84EA          58206022
  84EE          5A206026
  84F2          50206022
                ----------------
  850C          00000008
  8510          00000001
```

Figure 7. Same Program Segment, with Assembled Contents

The *identical* assembled program is generated in both cases. It therefore appears that so long as the same fixed relationship is maintained among the various parts of the program segment (there must be 22 bytes between the ST instruction and the fullword named N, and that N and ONE name areas that fall on fullword boundaries), then the program segment could be placed *anywhere* in storage and still execute correctly. That is, the program is *relocatable*.

The displacements of the three RX-type instructions were calculated on the assumption that at the time the program is executed there would be an address in R6 (the address of the L instruction named BEGIN) which could be used for a base address. (This is a key observation; we will use it shortly.) Indeed, we could have *assumed* that the program began at storage address zero (even though an actual program would not be placed there) because the contents of R6 after the BASR is executed would then be X'00000002', and the displacements would be calculated exactly as before.

In the first example, the *actual* origin of the program segment was X'5000'. We could by chance have assigned X'5000' as an *assumed* origin in the program, and then the values of the Assembler's Location Counter (LC) would be identical to the actual addresses later assigned by the Supervisor to each instruction. In certain simple operating systems it is possible that someone can tell us the actual origin that will be assigned by the Supervisor to our program; in general, however, this is an unnecessary and occasionally misleading piece of information.

Explicit Base and Displacement

Knowing what we want to obtain for the assembled program (the machine language instructions shown in Figure 5 on page 26 and Figure 7), we will now write the instruction statements with *explicit* addresses in their second operands. Register 6 is the base register, and the displacements are the ones we calculated above. Then we can write the program as shown in the following figure, using an assumed origin of zero for the LC.

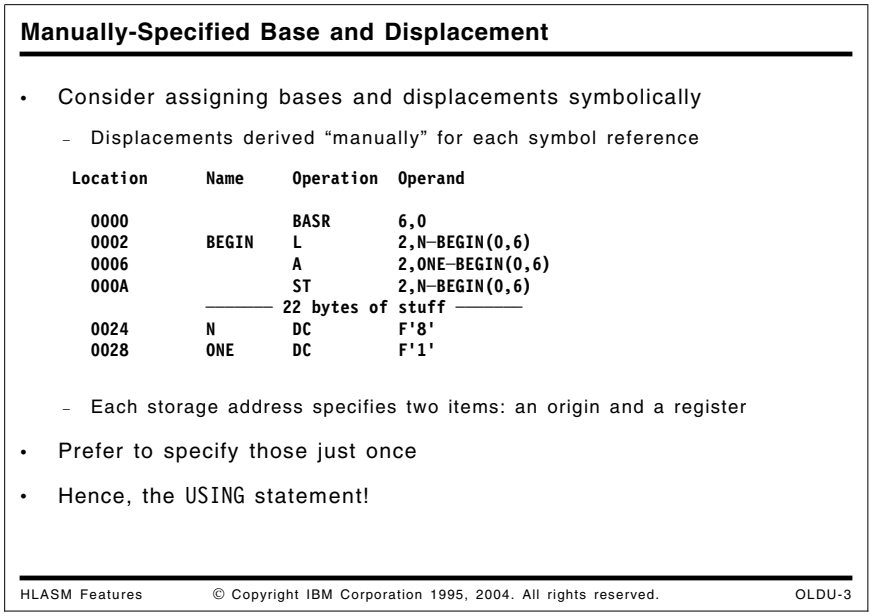
```
Location      Name     Operation  Operand

  0000                 BASR       6,0
  0002        BEGIN    L          2,X'022'(0,6)
  0006                 A          2,X'026'(0,6)
  000A                 ST         2,X'022'(0,6)
              --------- 22 bytes ----------
  0024        N        DC         F'8'
  0028        ONE      DC         F'1'
```

Figure 8. Program Segment with Pre-calculated Explicit Base and Displacements

This example of a program has two major shortcomings. First, calculating all the displacements in advance is a nuisance (especially in large programs), to say nothing of being error-prone. Second, if the relative positions of the parts of the program were to change in any way, we would be forced to recalculate some or all of the displacements.

Thus, our first simplification in this program is to devise a way to make the *Assembler* compute the displacements in the same way we did by hand. Now, however, we will make use of the values assigned by the Assembler to the symbols BEGIN, N, and ONE. (The values of the symbols are the values of the LC when the statement is scanned; thus the values assigned to these three symbols will be the value of the assumed origin plus X'2', X'24', and X'28' respectively.)

The key to this example is the observation made in discussing Figure 7 on page 27 above: at the time the program is *executing,* the base register we have chosen (R6) will contain the address of the instruction named BEGIN. We remember that the difference between assembly-time locations and execution-time addresses in a relocatable program can be only a single constant value, so we can rewrite the program segment as shown below.




```
Location      Name     Operation  Operand

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

Figure 9. Program Segment with Explicit Base and Assembler-Calculated Explicit Displacements

In this example we have eliminated both of the shortcomings of the program segment in Figure 8 on page 27: the values of the displacements were not calculated in advance, and the insertion of (say) four more bytes of instructions or data preceding the DC statements would not require that the rest of the program be rewritten. However, we have generated another nuisance, since *every* instruction containing a reference to a symbol must now specify two extra items: the symbol BEGIN, and the base register (6). It is therefore natural to devise some means that will let the Assembler do the rest of the work for us, after we have specified (1) the base register and (2) the value that will be in it when the program is executed.

The USING Statement and Implied Addresses

The USING assembler instruction statement provides exactly this information. It is written

```
USING   s,r1
```

where “s” is a relocatable expression. (Very infrequently, an absolute expression is used; we will mention this again in “Calculating Displacements: the Assembly Process” on page 32, and in more detail at “Absolute USINGs, Absolute Expressions” on page 38.) The value provided by the expression “s” is sometimes called the *base location.* The operand r1 is an absolute expression of value less than 16, which specifies the register to be used as a base register. Thus, the statement

```
USING   BEGIN,6
```

informs the Assembler that register 6 may be assumed (for purposes of computing displacements at *assembly* time) to be a base register which at *execution* time will contain the relocated value of the symbol BEGIN.

We could rewrite the sample program segment to include the USING statement as in the figure below.

Assembler-Calculated Base and Displacement

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

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

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

HLASM Features OLDU-4

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

Figure 10. Program Segment with USING Instruction

If the initial LC value assigned by the programmer is zero, the value of the symbol BEGIN will be X'2', and the values of the symbols N and ONE will be X'24' and X'28' respectively. To complete its derivation of the addressing halfword of the ST instruction, the Assembler needs only to note that the difference between the value of the symbol N, and the value

(BEGIN) specified in the USING instruction as being present in R6, is X'24'−X'2' = X'22'; this is the required displacement.

Similarly, the implied address in the operand field of the A instruction has value X'28'; when the base location value is subtracted, we obtain a displacement of X'26', as before. We say that the Assembler has *resolved* the implied addresses of the L, A, and ST instructions into base-displacement form. Thus, the machine language generated from this set of statements would appear exactly as in Figure 5 on page 26 and Figure 7 on page 27 above. (A more detailed discussion of the method used by the Assembler to compute displacements and assign bases will be given in “Calculating Displacements: the Assembly Process” on page 32.)

If the attempted calculation

```
displacement = (symbol value) − (base location value)
```

yields a negative value, or a value greater than 4095, the location referred to by the symbol is still not addressable, and some other solution would be needed.

It is clear that the Assembler can make use of the information supplied by the USING statement *only for implied addresses.* If you provide an explicit base and displacement, then the Assembler will simply convert them to their proper binary form.

Two important features of the program segment in Figure 10 on page 29 should be noted.

1. The USING instruction does *absolutely nothing* about actually placing an address into a register; it merely tells the Assembler what to *assume* will be there when the program is executed.

   That is, the USING statement is merely a promise from the programmer to the Assembler that if the Assembler computes displacements in the standard manner, everything will work properly when the program is executed. (Needless to say, it is easy to lie to the Assembler; see “Incorrectly Specified Base Registers” on page 31).

2. If the BASR instruction had been omitted, the contents of R6 is unknown. Thus, there is no guarantee that when the program is *executed,* the correct effective addresses will be computed. The following example will help to illustrate this.

Location Counter Reference

The Assembler provides a very useful notational device for referring to the current value of the Location Counter, the *Location Counter Reference.* The term “*” in an expression is given the current value of the LC; hence it is relocatable.

Thus we can rewrite the first two statements of our sample program as

```
BASR   6,0
USING  *,6
```

and achieve the same results as before. Remember that after the BASR is assembled, the LC will have a value corresponding to the location of the next byte to be assembled. Because the BASR will at execution time place the address of the following instruction into R6, we can use a Location Counter Reference to specify the base location, and not have to use a symbol (such as the symbol BEGIN in Figure 10.3.2) to name the following instruction.

A common technique for specifying base registers in a program is to choose a base register, write the statements

```
BASR   reg,0
USING  *,reg
```

at the beginning of the program, and then carefully avoid modifying that register. Thus, for simple programs, specifying and using base registers is reduced to a very simple procedure.

Incorrectly Specified Base Registers

A careless programmer inverted the order of his BASR and USING statements as follows:

```
USING  *,12
BASR   12,0
```

Why is this wrong? Precisely what would you expect to happen?

The two statements are in the wrong order. The value of the LC *before* the BASR is encountered may be 2 (or even 3) less than the value of the LC after the BASR has been assembled. Thus the value placed in the USING table by the Assembler will cause it to calculate displacements that are 2 (or 3) bytes too large. This will undoubtedly lead to incorrect operand addresses when the program is executed. Stated somewhat differently: the values of the base location specified in the USING expression (at assembly time) and the base address (at execution time), both measured relative to the start of the program, will not be the same.

Destroying Base Registers

Suppose an error had been made in preparing the statement with the L instruction, such that it became

```
BEGIN   L    6,N         Load contents of N into R2
```

(the first operand was incorrectly typed as 6 instead of 2). The assembled program would then appear as in Figure 11, assuming that an assumed origin of zero had been assigned to the Location Counter.

```
Location   Assembled Contents    Statement
  0000     0D60                        BASR   6,0
                                       USING  BEGIN,6
  0002     58606022              BEGIN  L      6,N     WRONG REGISTER!
  0006     5A206026                     A      2,ONE
  000A     50206022                     ST     2,N
                                 -------------------------
  0024     00000008              N      DC     F'8'
  0028     00000001              ONE    DC     F'1'
```

Figure 11. Sample Program Segment with Erroneous Statement

This program will assemble correctly with no diagnostic messages, since all quantities are properly specified according to the rules of the Assembler Language. However, at *execution* time, things go wrong in a hurry.

Suppose again that the program is placed in storage by the Supervisor starting at X'5000', so that when the L instruction is executed, R6 contains X'00005002'. Now, the L instruction is supposed to transmit a fullword from storage (at the address given by the second operand) into the register specified by the first operand. However, the first operand in this case specifies R6, instead of R2 as desired. When the effective address (of N) is being calculated during instruction *decoding,* R6 will contain the correct base address; but when the *execution* of the L instruction is complete, the contents of R6 will have become X'00000008', and *not* X'00005002', because the number at N will have been placed into R6.

Now the fun begins. When the *next* instruction (A) is executed, the effective address calculated is

X'026' + X'00000008' = X'0000002E'

and not X'00005028', which is the address where the desired operand is to be found. In this case, the generated effective address is not only not on a fullword boundary, but it is also somewhere among the old and new PSW's at the bottom end of storage; strange numbers will be added to R2's initial (and unknown) contents. Finally, the ST instruction will attempt

to store a fullword at X'0000002A', which should cause a storage protection exception. At this point, the program should stop.

This does not by any means imply that whenever we have the misfortune to destroy the contents of a base register, the processor will be able to detect the error. Indeed, if the contents of the fullword at N had been the decimal integer 20450 instead of 8, then the effective address would have been computed to be X'4FE2'+ X '26'= X '5008', which is a perfectly acceptable storage address for a fullword (and, besides, it's somewhere inside our program!). The subsequent instructions would thus have gone their merry and oblivious way, adding the contents of the fullword at storage location X'5008' to R2, and storing the result at location X'5004', which is *obviously* not what is intended!

It is partly a matter of chance how much damage such a program error can cause when the program is executed; indeed, when the processor finally (if it ever) detects an error, all evidence pointing to the offending instruction may have been lost (R6 may have been changed several times!), making error tracing difficult. Thus you must take care to guarantee the integrity of the contents of base registers, since the Assembler makes no checks for instructions that might alter the contents of registers designated in USING instructions as base registers.

Calculating Displacements: the Assembly Process

The method used by the Assembler to compute bases and displacements for implied addresses was described earlier in this section; we will now examine the process more closely.

One can visualize assembly as being done by making two *passes* over the program: that is, the Assembler “reads” the program twice. On the first pass, the Symbol Table is built; on the second pass, the data in the Symbol Table is used to help generate the desired instructions and data.

Pass One

We will now describe (in simplified form) the first pass of an assembly.

First, you will remember that values are assigned to symbols by the Assembler as follows:

1. A statement is read and examined to determine its general character. It is also saved in some temporary storage place so that it can be read again during the second pass over the program.
2. If the statement will generate instructions or data, the Assembler adjusts the Location Counter (if necessary) to satisfy alignment requirements, so that instructions begin on halfword boundaries, fullwords begin on fullword boundaries, etc.
3. If a symbol appears in the name field of the statement, it is entered into the Assembler's *Symbol Table,* and (if it is not an EQU statement) is given the value of the Location Counter. That is, the symbol is *defined.* (Of course, it will be an error if the symbol is already in the table with a value; this is called *multiple* or *duplicate* definition.)
4. The rest of the statement is scanned; if any other symbols are encountered, they are entered into the symbol table (if not there already), but numeric values are not assigned to their attributes. That is, if the symbol is not yet defined, it remains “undefined”.
5. The length of the instruction or data to be generated from the statement is then added to the Location Counter. No data or instructions are generated at this time, however.

This process is repeated for each statement, until the end of the program is reached. Because the Assembler has made a complete scan or “pass” over the program's statements, this is called “Pass One” of the assembly. At this point the Symbol Table contains all the symbols in the program, whether or not they are defined.

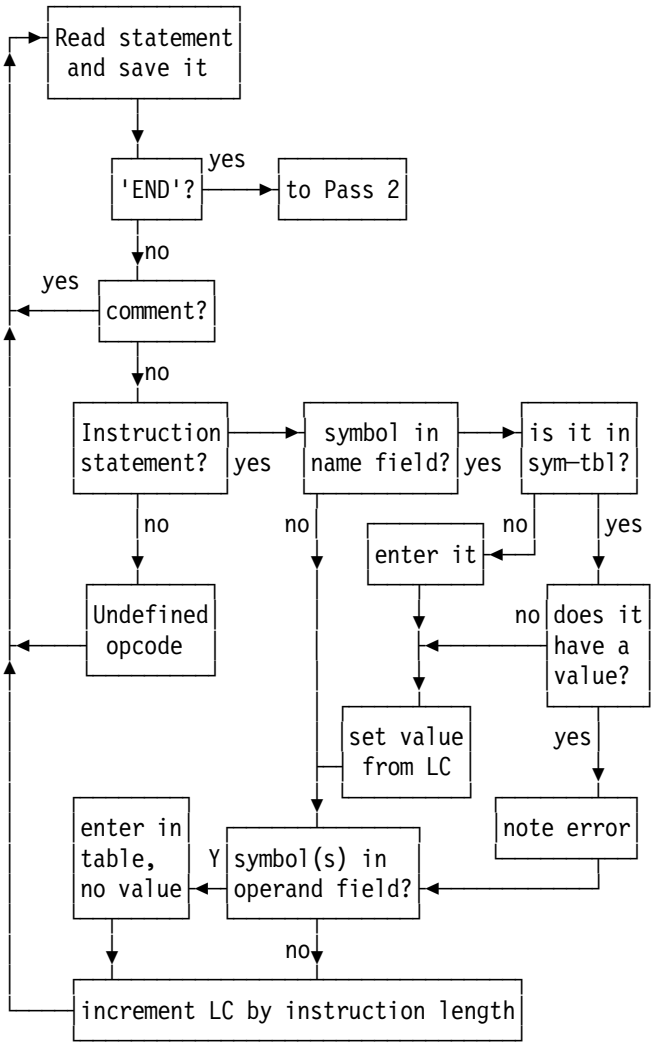


Figure 12. Pass One of Assembly

This figure is incomplete in many ways. For example, you will remember that the EQU statement allows you to assign a value to a symbol, and the value is taken from the expression in the operand field of the EQU statement. The figure above, however, only shows values being assigned to symbols by using the value of the Location Counter. It also omits any description of how erroneous statements are handled.

Pass Two

The Assembler now begins a second pass over the program by retrieving the statements from their temporary storage place. This time, however, the Assembler uses the data in the Symbol Table to evaluate *all* expressions appearing in the statements. When a USING statement is encountered, the Assembler enters the value and relocatability attributes of the first operand expression (the base location), and the value of the second expression (the register number), into a *USING Table.* When a subsequent instruction statement is encountered that contains an *implied* address, the Assembler compares the relocatability attribute and the value of that expression to each entry in the USING Table. If a valid displacement can be calculated from

```
displacement = (implied address value) - (base location value)
```

then the Assembler inserts the computed displacement and the corresponding base register digit into the addressing halfword of the instruction. We say that the Assembler has *resolved* the implied address into base-displacement form, and that the implied address is *addressable*.

For example, consider the second and third statements in Figure 10 on page 29. Assuming that the initial LC value assigned to the program was zero, the USING Table would contain an entry for register 6, with an associated relocatable base location value of X'00000002' (the value of the symbol BEGIN), as illustrated in Figure 13 below. The abbreviations “reg” and “RA” denote respectively the register specified in the second operand of the USING statement, and the relocatability attribute of the base location expression from the first operand of the USING statement. For now, the only importance of the relocatability attribute is that it indicates whether the symbol is relocatable (RA=01) or absolute (RA=00).

reg	base location	RA
6	00000002	01

Figure 13. USING Table with One Entry

The relocatability attribute of any given symbol almost always has a single value (it won't matter if we ignore the special “complex” situations for now, because they don't affect addressability). However, it is not at all unusual for a program to utilize many *different* relocatability attributes to correctly describe all its symbols.

In processing the third statement in Figure 10 on page 29, the value of the implied address is the value of the symbol N, or X'00000024'. The computed displacement is

X'00000024' − X'00000002' = X'022',

as we saw previously. Thus the completed addressing halfword is X'6022'.

We might summarize this description by saying that *the Assembler does at assembly time the opposite of what the processor does at execution time.* That is, the Assembler computes a displacement from the formula

displacement = (operand location) − (base location).

At execution time, the processor reverses this computation:

(operand address) = displacement + (base address).

The importance of giving correct information in a USING statement is now apparent, since it specifies the intimate connection between the base location at assembly time and the base address at execution time.

The overall flow of the second pass of the assembly process is sketched in Figure 14 on page 35 below. As noted following Figure 12 on page 33 (describing the first pass of the assembly), this is a very abbreviated description of the second pass, so that you should not attach great significance to the precise sequence of processing actions implied by the diagram.

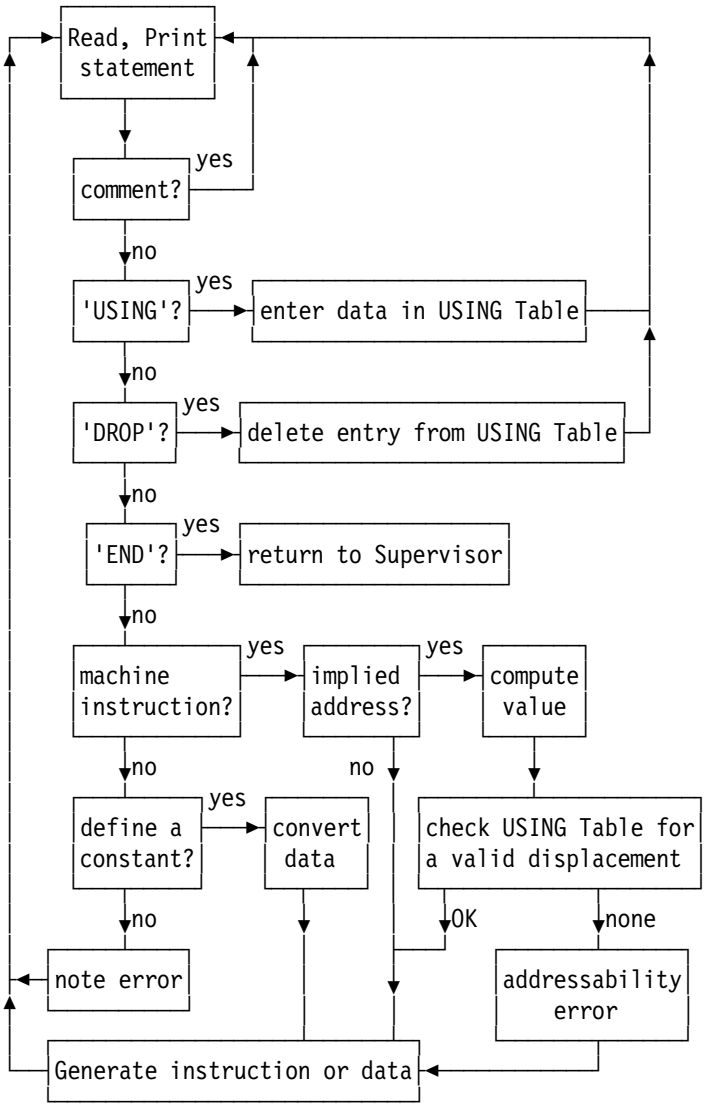


Figure 14. Pass Two of Assembly

An important feature of the High Level Assembler is that it provides an optional summary of all USING Table activity, in the form of a USING Map. If you specify USING(MAP) as part of the parameter string when you invoke the High Level Assembler, it will display all USING and DROP activity for the entire program.

Multiple USING Table Entries

It is possible to have more than one entry in the USING Table, and therefore to have a number of possible correct resolutions of an implied address into base-displacement form.

Suppose we add another USING statement to the program, so that it looks like this:

```
Name    Operation Operand     Remarks

        BASR      6,0
        USING     *,6
BEGIN   L         2,N
        USING     *,7         Added USING statement
        A         2,ONE
        ST        2,N
--------------------------
N       DC        F'8'
ONE     DC        F'1'
```

Figure 15. Program Segment with Second USING Statement

(For the moment, ignore the fact that the contents of register 7 is unknown; we will discuss this point shortly.)

When the second USING is scanned, the value of the Location Counter is X'00000006', so the assembler will make a second entry in the USING Table, as shown in Figure 16.

reg	base location	RA
6	00000002	01
7	00000006	01

Figure 16. USING Table with Multiple Entries

When the *next* statement

```
        A         2,ONE
```

is scanned, there are two possible valid resolutions available for the implied address specified by the symbol ONE:

- If register 6 is used as a base register, the displacement is

  X'00000028' – X'00000002' = X'026'

  and the addressing halfword would be X'6026' (as shown in Figure 11 on page 31).

- If register 7 is used as a base register (again, ignoring the fact that its run-time contents are unknown), the assembler determines that the displacement is

  X'00000028' – X'00000006' = X'022'

  and the addressing halfword would be X'7022'. (Similarly, the ST instruction could have an addressing halfword X'701E'.)

Now the assembler must make a choice: which of the two valid resolutions should be selected for the completed machine language instruction?

The High Level Assembler uses these resolution rules:

1. Find the USING table entries whose relocatability attribute matches that of the implied address to be resolved. If no matching entry is found, HLASM issues message ASMA307W indicating that no USING statement is active for the control section having the relocatability attribute of the implied address. (If the implied address is complexly relocatable, no match will be found.)

2. Choose the base register which leads to the smallest valid displacement. If the displacement exceeds the USING range (usually 4095 bytes), HLASM notes the excess in message ASMA034E.

3. If more than one base register provides the same smallest displacement, choose the corresponding highest-numbered register.

The implications of these choices will be discussed in more detail later.

Thus, the assembled program would appear as shown in Figure 17 below:

```
 Location       Assembled Contents

  00000         0D60
  00002         58206022            Based on register 6
  00006         5A207022            Based on register 7
  0000A         5020701E            Based on register 7
                ----------------
  00024         00000008
  00028         00000001
```

Figure 17. Assembled Contents when Two USINGs Are Active

At this point, you could (correctly) observe that this program is seriously (if not fatally) flawed, because the contents of register 7 at execution time could be “anything”. When the A and ST instructions are executed, their operand addresses are likely to cause errors (whether or not they are detected immediately!).

There is an important lesson in this example: the Assembler has no way of knowing that the information supplied in the statement

```
                USING  *,7
```

may not be valid. It can only proceed on the assumption that *you* have provided correct base-location and register data it can use to resolve implied addresses.

Resolutions With Register Zero

There is one further resolution rule used by the assembler when absolute implied addresses are not resolved according to the three previous resolution rules:

4. If no previous resolution has been completed, *and* the implied address is absolute and has value between 0 and 4095, use General Register 0 as the base register and the value of the implied address expression as the displacement.

Thus, if an implied address happens to be absolute, and has a value between 0 and 4095, the Assembler will assign a base digit of zero and a displacement equal to the value of the implied address. This behavior is used frequently in Assembler Language programs. Thus, if any implied address has a value that is absolute, a valid displacement can be computed only if that value does not exceed 4095.

According to the rules for evaluating expressions, an attempt to compute a displacement for a relocatable symbol using an absolute base location of value zero would require that the displacement be relocatable, which is of course invalid. That is, a valid displacement cannot be calculated from

```
(absolute) displacement ?= (relocatable) - (absolute).
```

Similarly, an absolute implied address cannot be resolved into base-displacement form using a register whose base location is relocatable, since a valid displacement cannot be computed from

```
(absolute) displacement ?= (absolute) - (relocatable).
```

Note: It is possible (but *not* recommended!) to specify USING statements with register zero as the base register, but the assembler will always assign a base address of zero to register zero.

The DROP Statement

It is also possible to *delete* entries from the USING Table. The DROP statement tells the Assembler to remove the information corresponding to a given register. For example, if the statement

```
DROP  6
```

was inserted after the third statement (labeled BEGIN) in Figure 15 on page 36, the initial entry would be deleted, and the USING table would appear as in Figure 18 below.

reg	base location	RA
	empty	
7	00000006	01

Figure 18. USING Table After DROP Statement

Unusable USING Table Entries: Addressability Errors

Suppose a second DROP statement is added after the A instruction in the program shown in Figure 15 on page 36, specifying register 7:

```
DROP  7
```

Then, the remaining entry in the USING Table would be deleted, and the USING table would appear as in Figure 19 below.

reg	base location	RA
	empty	
	empty	

Figure 19. USING Table After Second DROP Statement

Because there are no entries left in the USING Table, there is no way for the Assembler to resolve the implied addresses of any following instructions, and an addressability error condition would be noted for those statements.

Absolute USINGs, Absolute Expressions

While USING statements specifying absolute base addresses are rare, they are allowed; and absolute implied address expressions are subject to the same resolution rules as relocatable expressions. In most cases, there is no entry in the USING Table with an absolute base address, and the assembler proceeds as though an *implicit*

```
USING 0,0          Assembler's implicit USING-Table entry
```

is always present. Thus, an implied address such as

```
LA    3,1000        Implied address = 1000
```

would be resolved to the addressing halfword X'03E8'.

Now, suppose you had provided a USING statement with an absolute base address:

```
USING 400,9         Base Address = 400
LA    3,1000        Implied address = 1000
```

The assembler follows its usual resolution rules, and determines that there are *two* valid resolutions: X'03E8' and X'9258'. Since the latter provides the smallest displacement, the assembler chooses that resolution!

If the original resolution (using base register zero) is required no matter what other USINGs may be active, the terms of the previously implied address should be written explicitly, as

```
LA    3,1000(0,0)   Explicit displacement = 1000, base = 0
```

Ordinary USING Statements: Summary

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

HLASM Features OLDU-5

Summary

In summary, the ordinary USING statement provides two major features:

1. A *base location* relative to which the Assembler can calculate displacements
2. A *base register* to be used in addressing halfwords of those implied addresses whose displacements were calculated as being addressable with this register

It is important to remember that the information conveyed in a USING statement is only, and no more than, a promise that you make to the Assembler. The promise is that if the Assembler uses the base location and base address specified in the USING statement to calculate addressing halfwords at assembly time, then when at execution time the base address in the specified base register is used by the processor to calculate an effective address, the desired (and correct) address will be delivered.

Unfortunately, the rules used by the assembler to resolve implied addresses into base-displacement form are difficult to remember, and their complexity (and sometimes, subtlety) can lead to programming errors that can be quite difficult to correct.

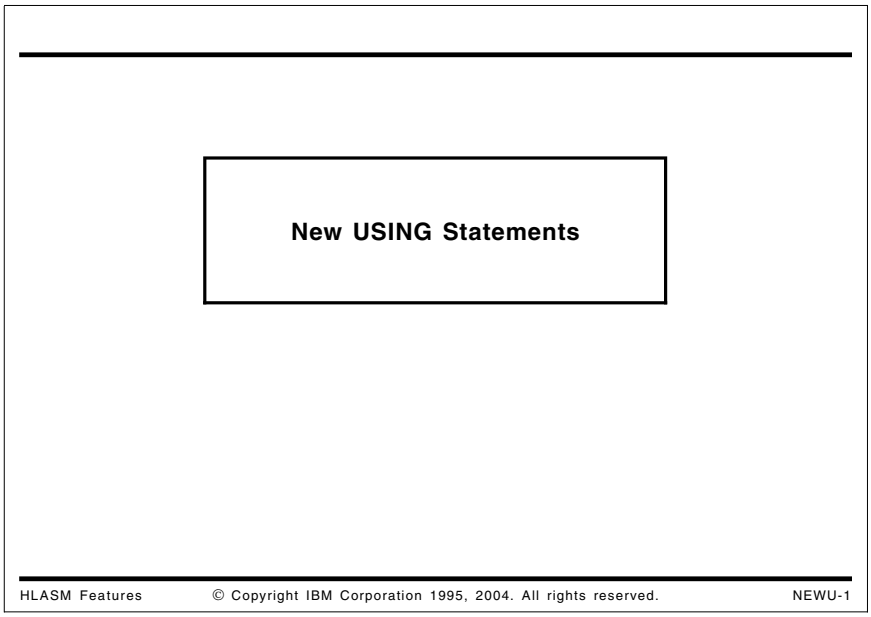
1. The assembler searches the USING Table for entries with a relocatability attribute matching that of the implied address (which will almost always be simply relocatable, but

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.

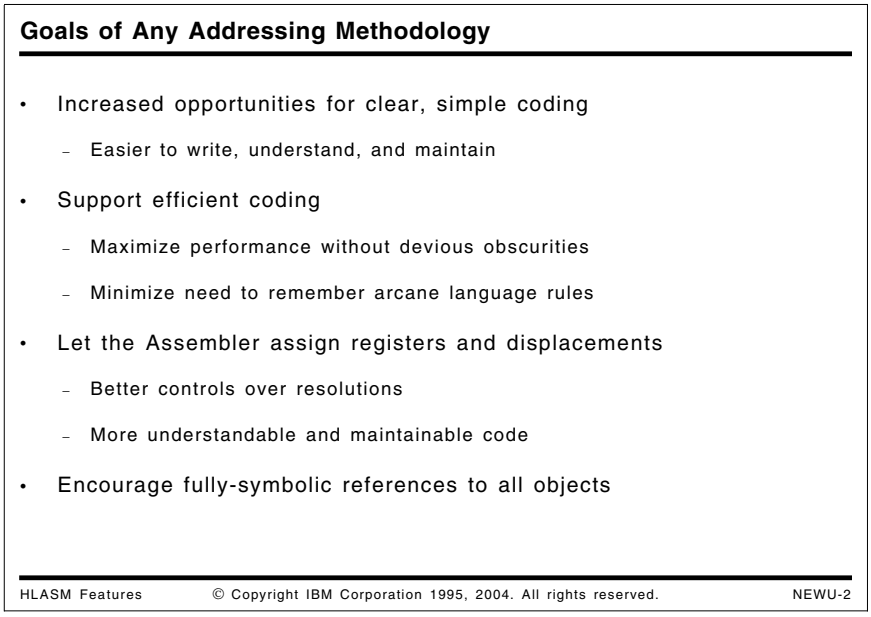
We will see in “New USING Statements” on page 41 that you may achieve even greater control over these resolutions, and that High Level Assembler provides new capabilities to assist in managing and diagnosing address resolutions.

New USING Statements



HLASM provides three powerful new forms of USING statement that can simplify coding, reduce errors, and help you write more efficient code without obscurities. They also help you to achieve the advantages of fully symbolic coding techniques while improving flexibility.

We will illustrate these new USING statements below.

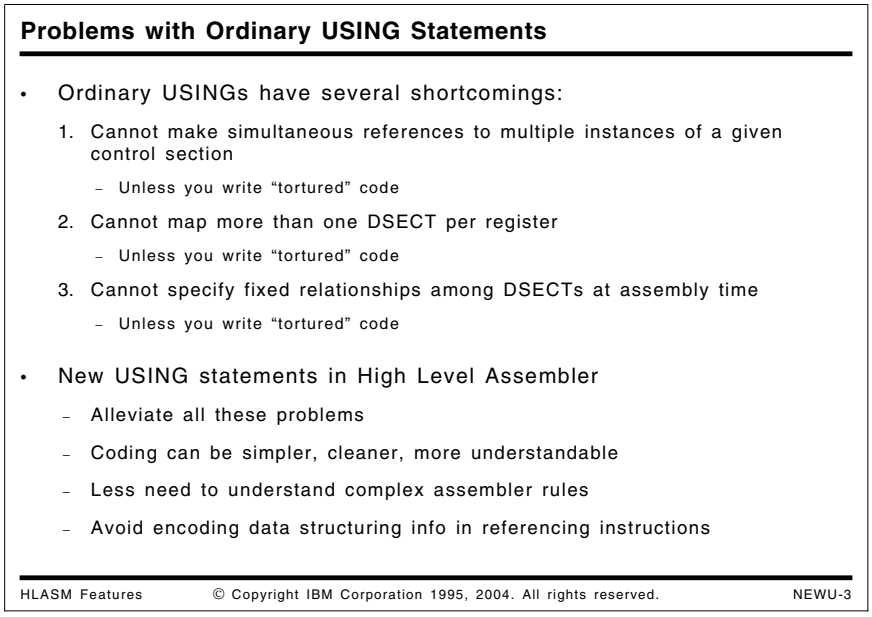


Desirable Properties of Any Addressing Method

Any addressing method should provide as many of the following benefits as possible.

1. Coding should be simple, clear, understandable, and efficient. These help with simplicity, readability, and maintainability.
2. All instructions should use fully symbolic references. These help with readability and maintainability.
3. Base registers and displacements should always be automatically assigned by the assembler from information provided in USING statements, and never be supplied as constants or as manual calculations. These also help with quality, readability, and maintainability.

Ordinary USINGs can easily fail in one or more of these respects, as some of the following illustrations will demonstrate. We will also show how the new USING statements can avoid most of these failures.

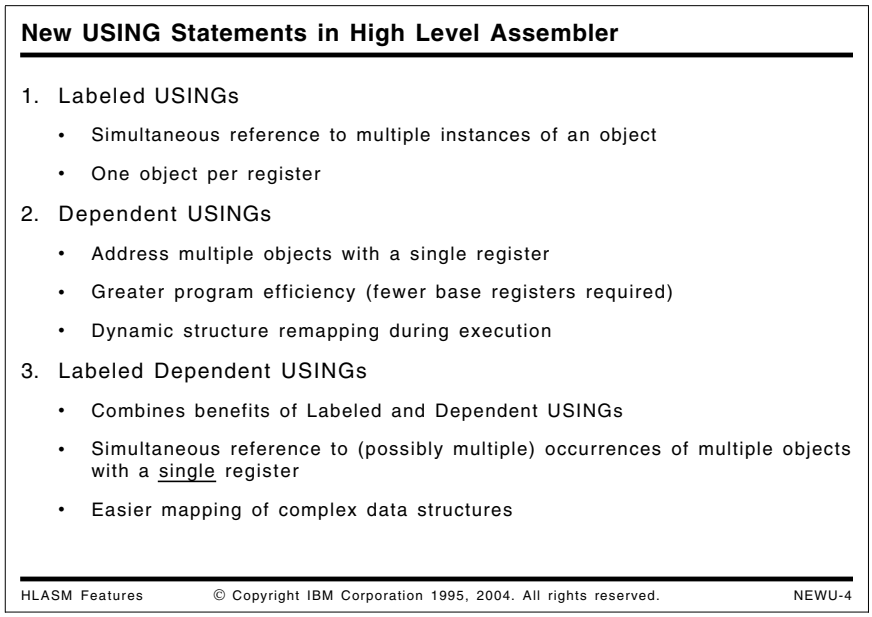


Problems with Ordinary USING Statements

There are three major problems with ordinary USING statements:

1. You cannot make “simultaneous” reference to multiple instances of a given control section (usually, a DSECT).
2. You cannot map more than one DSECT with a single register.
3. You cannot specify fixed relationships among related DSECTs at assembly time. (You can do this at *execution* time, but only at cost of allocating additional base registers).

The new USING statements in High Level Assembler solve all these problems, while still supporting fully-symbolic addressing capabilities.



Three New USING Statements

High Level Assembler provides two major types of extension to the USING statement: *labeled* and *dependent*. They may also be used in combination, as *labeled dependent* USINGs, giving you a repertoire of four different types of USING statement.

This enhancement (adapted from the "SLAC Mods to Assembler H")[1] permits much greater control over the assignment and resolution of base addresses in symbolic expressions and provides a capability that can improve the reliability, maintainability, and efficiency of assembler language applications.

- *Labeled USINGs* permit you simultaneously to address multiple instances of a DSECT (or CSECT) without the usual additional ordinary USING and DROP statements, and without the need to explicitly code offsets and base registers. Thus, you can concurrently manage multiple copies of the same DSECT- or CSECT-defined data structure using the full symbolic capabilities of the assembler language.
- *Dependent USINGs* permit you to address multiple DSECTs that are anchored by a single base register, enabling you to describe adjacent, nested, or overlapping code and data structures. This means that (unlike the symbolic addressing techniques required with all previous assemblers) you can actually reduce the number of general registers required for addressing DSECTs and assign them to other uses. This permits you to write more efficient code while retaining the traditional advantages of fully symbolic addressing for DSECT-mapped data.

  We will see that dependent USINGs have a useful dynamic property, whereby declarations may be changed in the code. This allows different mappings to be used on different code paths. (These relationships are not so dynamic that displacements are calculated at execution time; High Level Assembler still requires that all implied addresses be fully resolved at the end of the assembly. Such dynamic relations are also available with ordinary USINGs, but at cost of additional active base registers.)
- *Labeled dependent USINGs* combine the benefits of both. For example, you can describe record structures containing multiple instances of nested substructures, or of substruc-

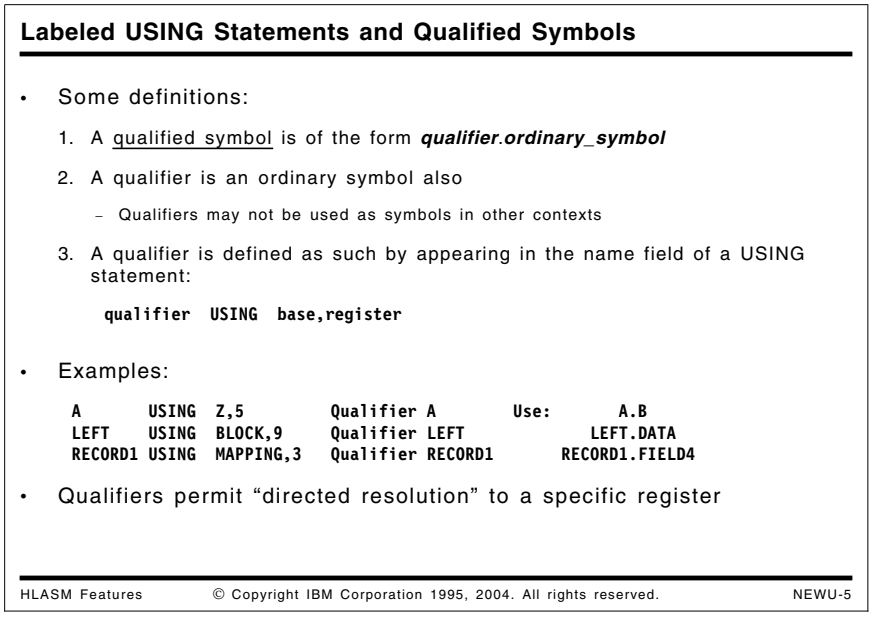
[1] The "Stanford Linear Accelerator Center Mods to Assembler H" achieved widespread use, and many customer and user-group requirements submitted to IBM were based on experiences with those extensions.

tures that depend on a variable elsewhere in the containing structure. Although such complex data structures are commonly used in higher level languages, previous assemblers could describe them only with very complex and difficult coding.

As you will see from the following examples, the possibilities for mapping and addressing complex data structures are much richer and more varied than with previous assemblers. Following the examples, we will summarize the properties of the four types of USING statements supported by High Level Assembler at “Summary of USING Statements” on page 89.

Labeled USING Statements

Labeled USING statements provide you with the capability of symbolically addressing more than one instance of a given control section at the same time. The usual rules for base-displacement resolution of symbolic operands are restricted to operands whose qualifier matches that on a valid USING statement.



Labeled USINGs and Qualified Symbols

A key concept in using labeled USING statements is the “label” or “USING label” or “qualifier” or “qualifying label”; all four terms can be used interchangeably. We will try to use only “qualifier” unless the meaning of the other terms is clear from its context.

A qualifier is a symbol, and follows all the rules for proper form of ordinary symbols. A qualified symbol is a pair of symbols separated by a period, with the first symbol being the qualifier and the second being the ordinary (operand) symbol:

```
qualified_symbol = qualifier.ordinary_symbol

Examples:                    A.B
                         LEFT.DATA
                     RECORD1.FIELD4
```

In the above examples, the qualifiers are A, LEFT, and RECORD1. The ordinary symbols are B, DATA, and FIELD4. Only symbols may be qualified. Even though they might make sense in the proper contexts, qualified terms such as A.*+4 are invalid uses of a qualifier.

A symbol is defined to be a qualifier by its appearance in the name field of a USING statement. The presence of this name field symbol distinguishes labeled USINGs from other USING statements.

```
A       USING  Z,5            Qualifier A
LEFT    USING  BLOCK,9        Qualifier LEFT
RECORD1 USING  MAPPING,3      Qualifier RECORD1
```

If a qualifying symbol is *not* present, the USING statement will be interpreted by High Level Assembler as an ordinary USING. Because qualifiers are maintained by the assembler in the same symbol table as ordinary symbols, they must be distinct. Thus, a qualified symbol like X.X is invalid.

The resolution rule for labeled USINGs is particularly simple: if a symbol is qualified, it may be resolved *only* with respect to the base register(s) specified in the labeled USING statement with the qualifier label. This can help you to avoid errors caused by the possibility of multiple address resolutions with ordinary USING statements.

Remember that correct qualification is no guarantee of addressability! Address resolution still requires that displacements not exceed 4095, and that the relocatability attributes of the addressing expression and the base location in the USING statement must match.

Examples of Labeled USING Statements

We will provide several examples to help illustrate the use and benefits of labeled USING statements, contrasting them with the coding that would be required to obtain similar results with ordinary USING statements. We will begin by showing how the limitations of ordinary USING statements can cause complexities and problems that can be avoided easily with labeled USING statements.

Managing Two Copies of a Data Structure

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

```
New instance (R5)                      Old instance (R7)

A      DSECT                           A      DSECT
F1     DS    - - -                     F1     DS    - - -
F2     DS    CL(FLen)  <-- copy --    F2     DS    CL(FLen)
- - -  etc.  - - -                     - - -  etc.  - - -
```

- We'd like the assembler to understand statements like

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

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

HLASM Features NEWU-6

Example 1: Managing Two Copies of One Structure

Suppose our program must manage two instances of a structure described by a DSECT named A, and that we wish to move a field (say, F2) from one copy of the DSECT to the other.

```
New instance                          Old instance

A      DSECT                          A      DSECT
F1     DS    - - -                    F1     DS    - - -
F2     DS    CL(FLen)  <--- copy ---  F2     DS    CL(FLen)
- - -  etc.  - - -                    - - -  etc.  - - -
```

Figure 20. Sample DSECT fragment, to Illustrate Problems with Ordinary USINGs

We will further suppose that

- R5 and R7 point to the new and old instances of A, respectively
- a simple MVC instruction is the desired efficient solution.

We would be happiest if the assembler could understand statements like

$$\texttt{MVC } F2_{NEW},F2_{OLD}$$

or `MVC NEW_F2,OLD_F2`

because they convey a clear and intuitive sense of what we want to do. We shall see (after examining some of the difficulties imposed by ordinary USING statements) that labeled USINGs let us do this!

We will examine the following approaches:

1. an example of incorrect addressing with ordinary USINGs
2. ordinary USINGs, with manually-specified displacements
3. unusual ordinary USINGs, with manually-specified displacements
4. ordinary USINGs and an intermediate temporary variable
5. duplicated (but differently-named) copies of the DSECT
6. labeled USINGs.

Managing Two Copies of a Structure (The Hard Way)

- Some examples of solutions with ordinary USINGs:

  1. Incorrect usage:

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

  2. With manually-calculated displacements (1):

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

  3. With manually-calculated displacements (2):

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

  4. With manually-calculated displacements (3):

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


Example 1: With Ordinary USINGs

We will illustrate several possible techniques for managing the two copies of DSECT A, using ordinary USING statements. Some of the techniques are clearly incorrect; they are included simply to show how (apparently) obvious and simple solutions can lead to unexpected pitfalls.

Example 1a: Incorrect Usage

First, consider an "obvious but incorrect" solution. Suppose we wrote either of the two following sequences of statements:

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

Figure 21. Incorrect Coding for Simultaneous DSECT Usage

Both of these code sequences fail because only R7 will be used to address the fields of DSECT A. (If two registers are based on the same location, the assembler will choose the higher-numbered register for base-displacement resolutions.) The MVC instructions will effectively move the old field "onto itself", producing no result whatever. (Note that High Level Assembler will provide a diagnostic message warning about the fact that R5 has been nullified as a base register; all other assemblers will not.

In summary, the defects of this technique are

- incorrect code
- no warning message from old assemblers.

Example 1b: Correct (But Not Recommended) Usage: Manually-Specified Displacements and Registers

Suppose now that we now rewrite these simple statements to avoid the previous problems, by specifying the displacement and base to be used:

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

This sequence has the disadvantage that the displacement and base are assigned by the programmer, rather than by the assembler. If there is ever a need to re-allocate base registers (so that, perhaps, R7 is given a different use), then all references to R7 must be located and inspected to see if they need changing.

In summary, the defects of this technique are

- more complex coding
- more difficult maintenance.

Writing these two statements a different, if obvious, way can lead to even more serious difficulties:

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

This sequence, while syntactically correct, will undoubtedly be wrong, because the syntax rules of the Assembler Language dictate that if the first operand of an SS-type instruction is written in the form expression1(expression2), then expression1 provides the implied address and expression2 provides the operand's explicit length value. (A second and possibly more serious flaw is that because expression1 is absolute, the first operand may be resolvable

with base register zero, and therefore refer to the low-addressed end of storage! Fortunately, HLASM will attempt to diagnose such references if you specify the FLAG(PAGE0) option.)

Consider how much more difficult this problem would have been to solve if you had used "proper" register notation:

```
USING  A,R7           map old instance of A with R7
MVC    F2-A(R5),F2    move from old to new (based on R5)
```

The use of the symbol R5 would lead most readers to believe that it was a correct register reference, while in fact it is the *length* expression!

It will be a rare coincidence if the length of the field F2 is the same as the value of the symbol R5, so this statement will only work "partially", and almost always incorrectly, despite the lack of any diagnostics.

The correct form is

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

which requires remembering obscure rules of the assembler language; such usage is not obvious to most programmers. We will see that labeled USINGs can help eliminate these obscurities.

Another potential trap in manually assigning registers is that USINGs may be in effect for both the old and new register numbers, in such a way that a statement may assemble correctly but its operand(s) may be resolved with respect to the wrong register.

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

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

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

- - -                 more statements (forgetting to drop R0)

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

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

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

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

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

HLASM Features NEWU-8

To avoid these syntactic difficulties, a clever programmer might observe that a manually-calculated displacement can be resolved without having to specify a base register explicitly by specifying a zero base address and the desired register:

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

and the MVC instruction will now resolve correctly.

However, if you forget to DROP register 0, later statements that depend on absolute expressions resolving with register 0 may not give the correct object code:

```
- - -                 more statements (forgetting to drop R0)
LA     1,100          Resolved on R7! (X'4110Z064')
```

To avoid this defect, our clever programmer may also observe that setting a large absolute offset in the USING statement and in the manually calculated displacement avoids contaminating later resolutions intended for R0:

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

Again, the code is correct, but at the cost of complexity and coding unlikely to be understood by later maintainers.

Example 1c: Problems with "Manual" Assignment

Suppose the data structure mapped by DSECT A grows to be longer than 4096 bytes. Naturally, you would establish two base registers to map each of the two instances:

```
        LA     6,4095(0,5)     increment R5 by 4095 into R6...
        LA     6,1(0,6)        and by 1 more, for second base
        USING  A,5,6           map new instance of A
        LA     8,4095(0,7)     increment R7 by 4095 into R8...
        LA     8,1(0,8)        and by 1 more, for second base
*       USING  A,7,8           implicit map of old instance of A
```

Then, if you write

```
        MVC    F2,F2-A(7)      F2-A might exceed 4095?
```

the correctness of the second operand depends on whether the manually-assigned displacement F2−A is *less* than 4095. If not, the displacement will be too large, and the manually-assigned register (7) will be incorrect. Thus, you would have to write

```
        MVC    F2,F2-A-4096(8) if F2-A exceeds 4095
```

which is obviously error-prone, since it depends on the current size of DSECT A and the position of field F2 within A.

In summary, the defects of these techniques are

- greater likelihood of undetected error
- deeper understanding required of language details
- more complex coding
- more difficult maintenance.

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

8. With an intermediate temporary (1):

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

9. With an intermediate temporary (2):

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

10. With a duplicated copy of the DSECT:

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

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

HLASM Features NEWU-9

Example 1d: Correct (But Still Not Recommended) Usage: Intermediate Temporary Variable

Correct references to the specific instances of the DSECT A can be obtained (apparently) by using an intermediate temporary storage area:

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

Figure 22. Incorrect Coding for Intermediate Temporary

Unfortunately, this version fails because the programmer forgot the (possibly obscure) rule that if two registers are based at the same location, the higher-numbered register will be used for calculating displacements. Thus, the second MVC instruction will merely move the data from TEMP back to where it started!

The solution for ordinary USINGs is to insert a DROP statement for R7:

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

Figure 23. Corrected Coding for Intermediate Temporary

In summary, the defects of these two techniques are

- greater likelihood of undetected error
- deeper understanding required of language details
- more complex coding
- less efficient instruction sequences
- more difficult maintenance

Example 1e: Correct (But Definitely Not Recommended) Usage: Duplicated DSECTs

A programmer who observes the defects of the above methods of managing two instances of the DSECT A might decide that the best approach will be to make a second copy, with a different name, in order to avoid having to write confusing USING and DROP statements. Thus, he might define an exact copy of A, now named B:

```
B      DSECT
G1     DS    - - -
G2     DS    CL(FLen)
- - -  etc.  - - -
```

Figure 24. The Hard Way: Making a Copy of the DSECT

Then, the desired code sequence takes a much cleaner and simpler form:

```
          USING  B,7           map old instance of A (named B)
          USING  A,5           map new instance of A
          MVC    F2,G2         move from old to new
```

While this is the desired code sequence, the technique can lead to extreme difficulties in maintenance if the maintainer doesn't appreciate that the original coder expected that B must be an *exact* duplicate of A. If changes are made to A, the differences in DSECT and symbol naming make it easy to overlook the requirement to make equivalent and identical changes to B. It is also less obvious that the symbols in this code fragment actually refer to the same things.

In summary, the defects of this technique are

- greater likelihood of maintenance problems
- greater difficulty in understanding the code.

Example 1f: A Simpler Hard Way: Macro-Duplicated DSECTs

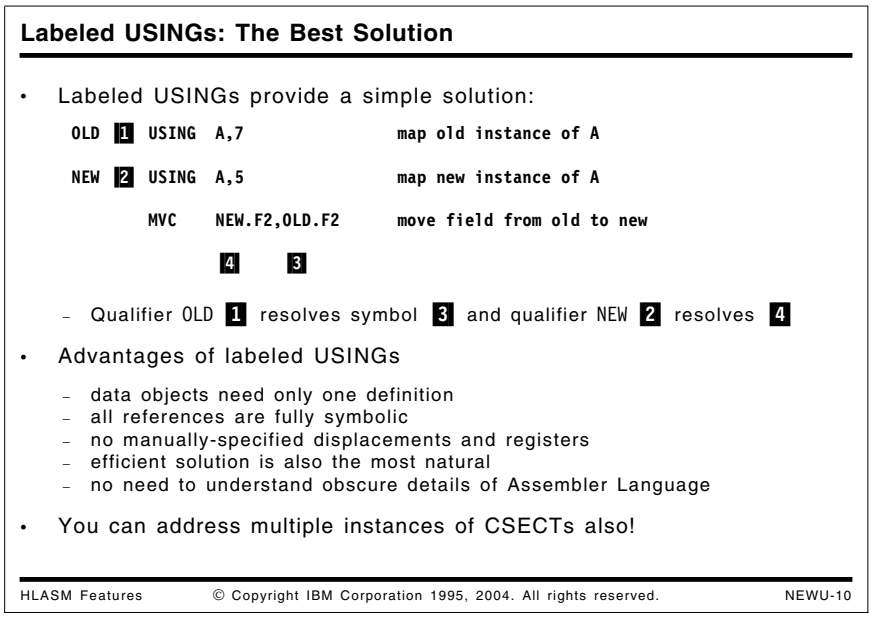
Occasionally, this “duplicate definition” technique is encapsulated in a macro definition. For example, suppose you have written a macro named DDSECTA to define copies of DSECT A. The macro can generate as many copies of the DSECT as needed, adding a specified prefix to each of the generated symbols, as illustrated in Figure 25:

```
          DDSECTA PREFIX=OLD       DSECT A, symbols prefixed 'OLD'
+OLDA     DSECT
+OLDF1    DS    - - -
+OLDF2    DS    CL(FLen)
          - - -
          DDSECTA PREFIX=NEW       DSECT A, symbols prefixed 'NEW'
+NEWA     DSECT
+NEWF1    DS    - - -
+NEWF2    DS    CL(FLen)
          - - -
          USING  NEWA,5
          USING  OLDA,7
          MVC    NEWF2,OLDF2       Move from old F2 to new F2
```

Figure 25. The Simpler Hard Way: a Macro to Copy the DSECT

This technique -- the most satisfactory of all the approaches discussed up to this point -- ensures that only a single source file containing the DSECT's definition is maintained (inside the macro). The defects of this approach are:

- it introduces new symbols and DSECTs into the program, some of which are duplicate names for what is really one object
- it requires that an additional piece of code (the macro definition) be defined and maintained
- all references to the DSECT must use the prefixed names, even when only a single instance of the object is active (unless a third set of names is generated, with no prefix!).



Example 1 Solution: Labeled USINGs

In High Level Assembler, labeled USINGs provide a clean and simple solution to this problem. Suppose the DSECT A has been declared as in Figure 20 on page 46 above. By specifying two labeled USING statements and by specifying appropriately qualified symbols, the resulting code is much cleaner:

```
OLD  1  USING  A,7              map old instance of A
NEW  2  USING  A,5              map new instance of A
        MVC    NEW.F2,OLD.F2    move field from old to new
               4      3
```

Figure 26. The Right Way: Labeled USINGs

The labeled USING with qualifier OLD (at 1) is used to qualify the second occurrence of the symbol F2 (at 3). Similarly, the labeled USING with qualifier NEW (at 2) is used to qualify the first occurrence of the symbol F2 (at 4). Because both occurrences of F2 are qualified, they can only be resolved into base-displacement form using the proper register.

You can see that an appropriate choice of qualifier names also makes the code easier to read and understand!

This example illustrates several advantages of labeled USINGs:

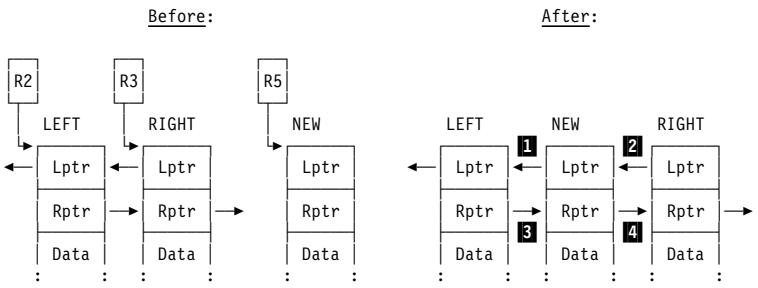
1. Data objects need be defined only once, no matter how many times they may appear to be used concurrently.
2. All references are fully symbolic, and neither explicit base registers nor manually-calculated displacements are required.

3. The desired, efficient solution is obtained in a simple, direct, and readable way.
4. The programmer need not understand the details of instruction syntax or of the address resolution rules for ordinary USING statements.

It is worth noting that the “objects” addressed by labeled USINGs need not always be mapped by dummy control sections (DSECTs). If you have defined a data structure as a CSECT and have made copies, you can reference the copies in exactly the same way as if they are mapped by a DSECT: just set a register to point to each copy, and then issue a labeled USING with the CSECT name as the base address.

Example: Doubly-Linked List Structure

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



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

Example 2: Doubly-Linked List Structure

Suppose we have a data structure requiring a doubly-linked list, in which each structure element points both to its predecessor (called the “left” element) and to its successor (called the “right” element).

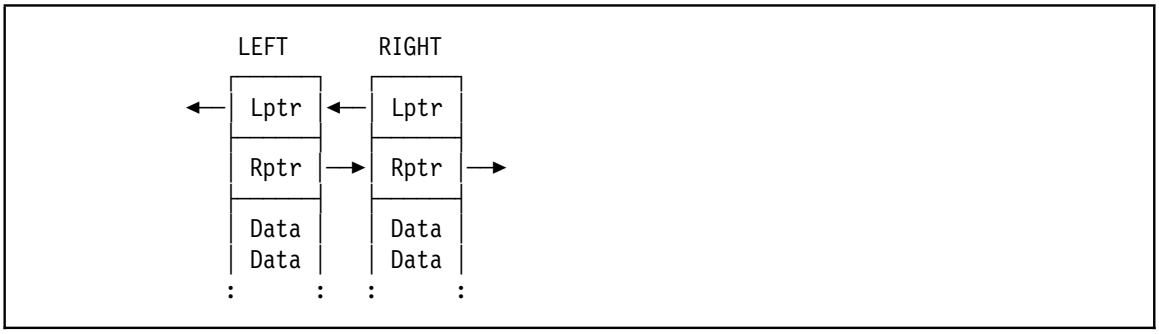


Figure 27. Doubly-linked List Structure

Let the format of a structure element be defined by a DSECT named BLOCK:

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

Figure 28. Labeled USING Example 2: a DSECT Describing a Small Control Block

For this example, we will suppose that we have three distinct instances of the control block structure described by BLOCK: two linked elements LEFT and RIGHT (which are addressable using registers 2 and 3 respectively), and a NEW element addressed by register 5. We wish to insert the new instance of the BLOCK between the two existing instances.

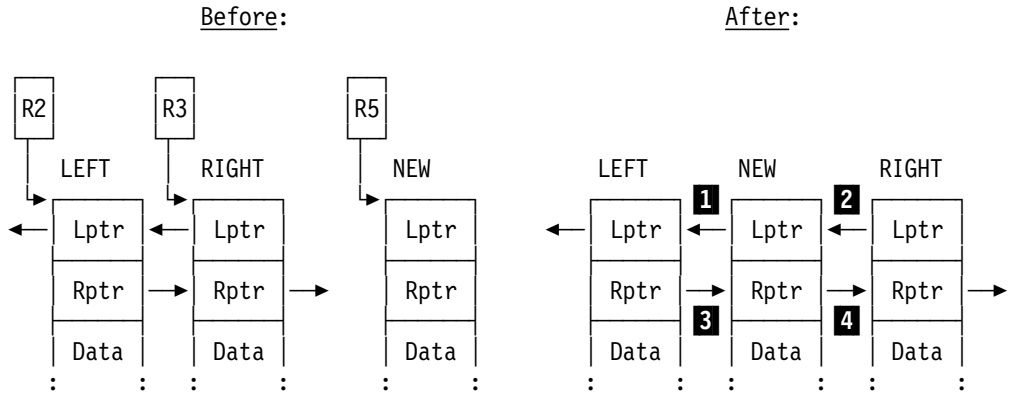


Figure 29. Example 2: Inserting a New Instance of BLOCK

The links that must be changed during the insertion process are indicated in the figure above by the keys **1** and **2** (the “left” links), and by **3** and **4** (the “right” links).

Example 2a: With Multiple Ordinary USINGs

The “cleanest” technique with ordinary USING statements is to refer to the fields in BLOCK with proper symbolic addressing throughout. The following code sequence shows how this might be done:

```
         USING  BLOCK,3        map RIGHT element
         L      0,Lptr         save  old Right.Lptr
         ST     5,Lptr         store new Right.Lptr  2
         DROP   3              unmap RIGHT element

         USING  BLOCK,2        map LEFT element
         L      1,Rptr         save  old Left.Rptr
         ST     5,Rptr         store new Left.Rptr   3
         DROP   2              unmap LEFT element

         USING  BLOCK,5        map NEW element
         ST     0,Lptr         store new New.Lptr    1
         ST     1,Rptr         store new New.Rptr    4
```

Figure 30. Ordinary-USING Code to Insert a New List Element

The statements that establish the links are indicated by keys **1** through **4**, as defined in Figure 29.

The primary shortcomings of this method are

- intermediate temporaries (in this case, registers 0 and 1) are used to hold some of the pointers
- the precise sequence of USING and DROP statements is required to obtain correct address resolutions
- two additional instructions are required (Load and Store via registers, rather than an MVC).

Example 2b: Correct (But Not Recommended) Usage: Manually-Specified Displacements

To eliminate the need for intermediate temporaries, we might wish to use MVC instructions to move the fields (with a presumed gain in efficiency):

```
RNew     Equ    5                    R5 points to New element
         USING  BLOCK,RNew           map NEW element
         MVC    Lptr,Lptr-BLOCK(,3)  move  old Right.Lptr 1
         ST     RNew,Lptr-BLOCK(,3)  store new Right.Lptr 2
         MVC    Rptr,Rptr-BLOCK(,2)  move  old  Left.Rptr 3
         ST     RNew,Rptr-BLOCK(,2)  store new  Left.Rptr 4
```

Figure 31. Ordinary-USING Code to Insert a New List Element

This code sequence contains the desired “efficient” instructions, but its defects are considerable:

- greater difficulty of understanding
- increased likelihood of maintenance problems due to fixed assignments to registers in the instructions themselves

Labeled USINGs: Doubly-Linked List

- Code with labeled USINGs is very simple:

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

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

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

HLASM Features NEWU-12

Example 2c: The Clean and Simple Way: Labeled USINGs

By far the simplest and clearest solution is to use labeled USINGs, with appropriate descriptive qualifiers. In this example, references to the left- and right-pointer fields “LPTR” and “RPTR” are *qualified* through the use of the qualifying symbols LEFT, RIGHT, and NEW. Observe that three instances of the DSECT named BLOCK are concurrently active.

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

Figure 32. Labeled USING Example 2c: Code for Inserting a New Control Block

The advantages in clarity, readability, simplicity, and improved ease of maintenance are obvious. Without labeled USINGs, the code for these operations is much more convoluted and difficult to read, understand, and maintain.

Labeled USING Statements: a Summary

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

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

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

HLASM Features NEWU-13

Labeled USING Statements: a Summary

Labeled USING statements have several interesting properties:

- No symbol without a qualifier that matches the qualifying label can be resolved with that USING. This means that you can actually have several USINGs active against a particular base register at the same time. In general, this practice would not be recommended, because it will be more difficult to understand the code.
- DROP statements for labeled USINGs must be specified by the qualifier, not by the register. (Extensions to the DROP statement will be discussed in “DROP Statement Extensions” on page 90.)

- Normal base-displacement address resolution rules are still in effect:
  - The relocatability attributes of the implied address must match those of candidate entries in the Using Table before displacement calculation will be attempted.
  - Valid displacements still cannot exceed 4095.
- Labeled USINGs and ordinary USINGs specifying the same base register may be active at the same time. For example:

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

Figure 33. Concurrently Active Ordinary and Labeled USINGs

Implied addresses containing symbols without qualifiers will be resolved with the ordinary USING, and qualified symbols will be resolved only with the matching labeled USING. As this example shows, the DROP statement deletes the Using Table entry for the ordinary USING, but the labeled USING remains in effect.

This style of programming should be used with caution, due to the greater potential for confusion.

Dependent USING Statements

- Let you address multiple DSECTs with one base register
  - Provide improved ways to manage data structures
- Syntax is the same as for ordinary USINGs:

  USING symbol,base
- Except that the second operand is interpreted differently:

  ordinary: second operand is absolute, between 0 and 15

  USING symbol,register

  dependent: second operand is relocatable, addressable

  USING symbol,anchor_location
- First operand is “based” or “anchored” at second operand location

HLASM Features NEWU-14

Dependent USING Statements

Dependent USINGs permit addressing multiple DSECTs with a single base register. We will illustrate some typical problems in managing such addressing problems with ordinary USINGs, and then show how dependent USINGs can provide simpler solutions.

Dependent USINGs can provide elegant solutions to problems involving the management of data structures that are adjacent, nested, or overlapping in storage, while maintaining

- addressability with a minimum number of registers
- fully symbolic structure and substructure mappings with independent DSECTs.
- simple mappings of complex data structures
- ability to map "variant records" in which the structure of parts of the record depend on preceding data values
- ability to provide different mappings along different code paths

Definition of Dependent USING Statements

Dependent USING statements allow any object – normally, a DSECT – to be "anchored" or "based" at any location already addressable by an existing USING statement.

Dependent USING statements are written with almost the same syntax as an ordinary USING, but with one key difference:

```
USING  operand1,operand2
```

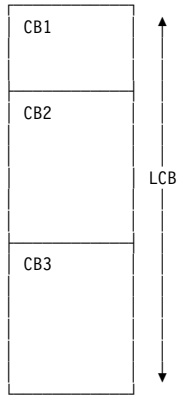
If the second operand of the USING statement (operand2) is absolute, then it must have a value between zero and fifteen, and it then designates the base register of an ordinary USING statement. On the other hand, if the second operand is *relocatable*, then it is understood to be the "supporting base location" at which the first operand is to be "based" or "anchored", and the USING statement will be taken to be a dependent USING statement. Note that this base or anchor location must itself be within the range of an existing ordinary USING statement, because implied operand addresses must still be resolved into base-displacement form with respect to a declared base register and base location.

In summary, we can characterize the difference between an ordinary USING and a dependent USING by the way the first-operand location is "based" or "anchored".

- For *ordinary* USINGs, the first operand is "based" on the *register* specified by the second operand.
- For *dependent* USINGs, the first operand is "based" on the *location* specified by the second operand; this location must already be addressable.

Note that for dependent USINGs, the relative position of the first operand is set at *assembly time* by the assembler, rather by instructions that set the address at *execution time* (as with ordinary USINGs).

Dependent USING Example: Contiguous Control Blocks




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

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

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

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

Contiguous
Control Blocks

Dependent USINGs Example 3: Contiguous Control Blocks

In this example, we assume that a large block of working storage will be acquired, which is to contain several different (and independently named) data structures or control blocks named CB1, CB2, and CB3, which are to reside in adjacent areas of storage. Suppose the control blocks are defined as follows:

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

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

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

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

Figure 34. Dependent USING Example 3: Control Block Definitions

Previous assemblers required using a separate register to address each control block; dependent USING statements allow all of the control blocks to be addressed with a single register.

Contiguous Control Blocks: Ordinary USINGs

- Ordinary USINGs require a register for each DSECT:

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

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

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

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

HLASM Features NEWU-16

Dependent USINGs Example 3a: Contiguous Control Blocks with Ordinary USINGs

To address these three independent control blocks with ordinary USINGs, we must assign one register to hold a base address for each.

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

Figure 35. Dependent USING Example 3a: Control Block Addressing with Ordinary USINGs

There are several defects to this apparently sensible approach:

- Two unnecessary additional base registers are required.
- The extra LA instructions imply a possible loss of efficiency.
- The relationship among the three control blocks is determined by the operands of the LA instructions, and not by any other data declarations.

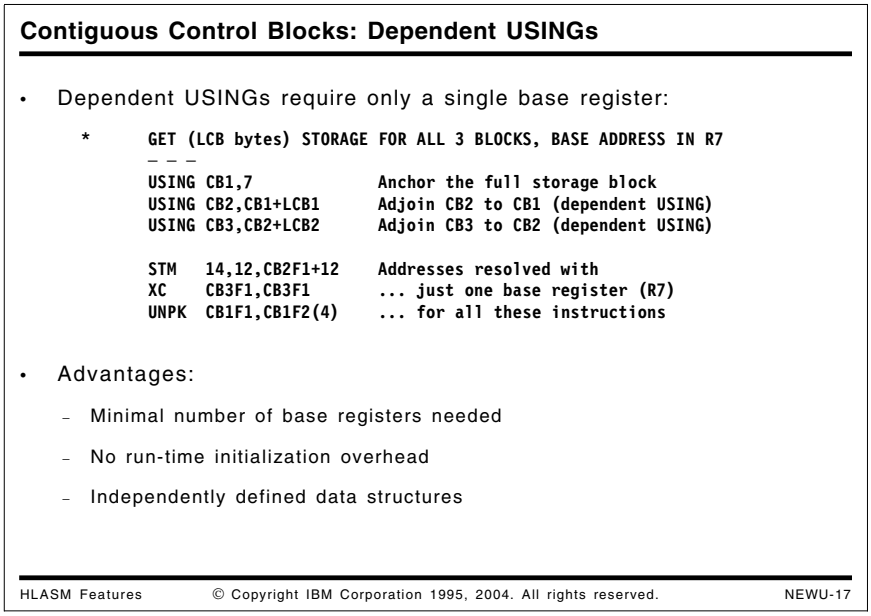
An alternative approach that avoids the need for additional base registers might be to code the offsets of the second and third control blocks into the instructions that reference their fields:

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

Figure 36. Dependent USING Example 3a: Control Block Addressing with Ordinary USINGs

While this method will allow you to refer to all three control blocks using a single base register, it introduces further defects:

- complex coding that is hard to understand and maintain
- the relationships among the control blocks is embedded in each referencing instruction; changing those relationships (for example, interchanging the order of CB2 and CB3) requires modifying every referencing instruction.



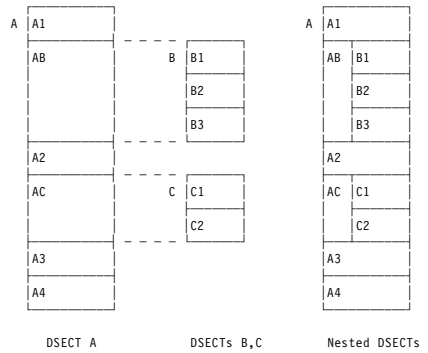
Dependent USINGs Example 3b: Contiguous Control Blocks with Dependent USINGs

Because several such blocks can now be referenced through a single register, registers previously required for addressing can be allocated to other useful purposes, thereby increasing the efficiency of the program.

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

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

Figure 37. Dependent USING Example 3b: Control Block Addressing with Dependent USINGs

Dependent USING Example: Nested Structures




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

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

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


Dependent USINGs Example 4: Nested Structures

Many high-level languages such as C, COBOL, Fortran90, Pascal, and PL/I support “nested” data structures, or “structures of structures”, where data structures may contain one or more sub-structures. Furthermore, there may be more than one instance of each substructure, as in PL/I's “arrays of structures”. The limited facilities of earlier assemblers made it difficult to write straightforward statements to describe these complex data structures, because there was no way to define a DSECT on or within another DSECT without having to use another base register. We will see that High Level Assembler provides facilities that make this much easier.

Suppose we wish to describe three independently-defined “records” A, B, and C. In this example, we will want to use B and C as sub-records of C, as illustrated in the following figure:

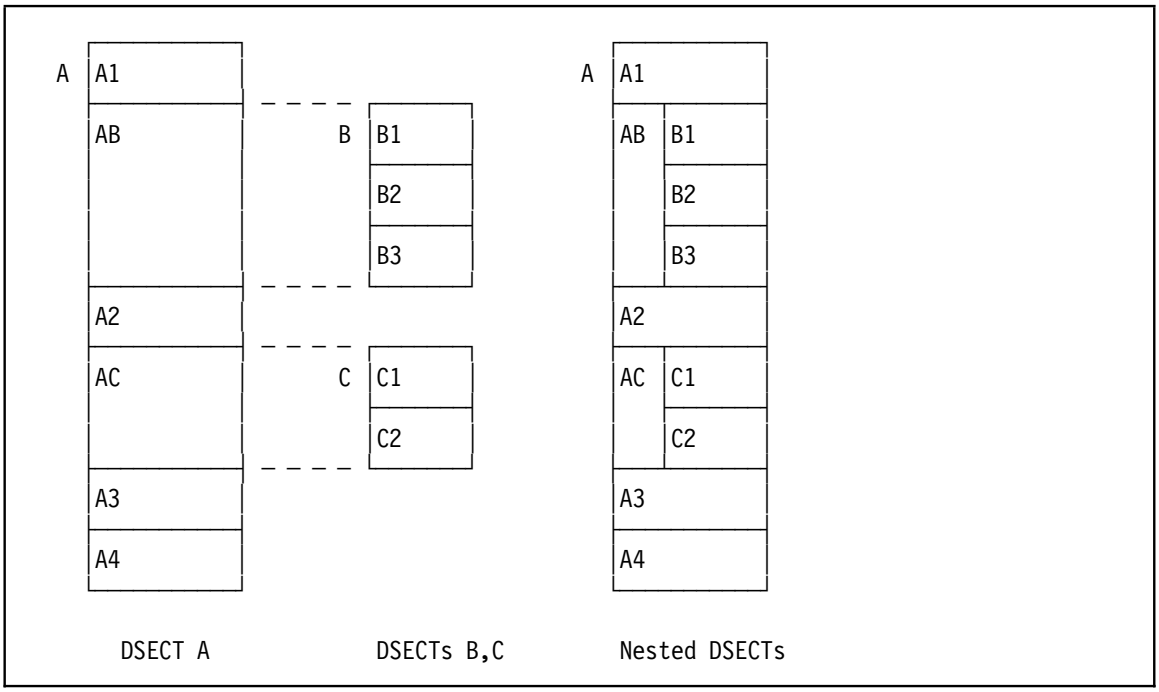


Figure 38. Nested or Overlaid Data Structures

In the figure, the three DSECTs A, B, and C are independently defined. In actual use, we wish to "overlay" or "nest" B and C within A, at the positions labeled AB and AC respectively (at keys **1** and **2**). The DSECTs themselves might be defined with statements like the following:

```
A    DSECT                 | B    DSECT                | C    DSECT
A1   DS    24F             | B1   DS    CL44           | C1   DS    96D
AB   DS    CL(LB)  1       | B2   DS    6D             | C2   DS    4XL20
A2   DS    6CL80           | B3   DS    4A             |
AC   DS    CL(LC)  2       |                           |
A3   DS    XL16            |                           |
A4   DS    CL256           |                           |
```

Figure 39. Defining the DSECTs Which Will Be Nested

We could attempt to solve this problem with ordinary USINGs (in which case three base registers are required), or with techniques like DSECT renaming, where we define the inner DSECTs as being part of the outer. For example, in the above, we could "nest" the components of DSECT B inside DSECT A this way:

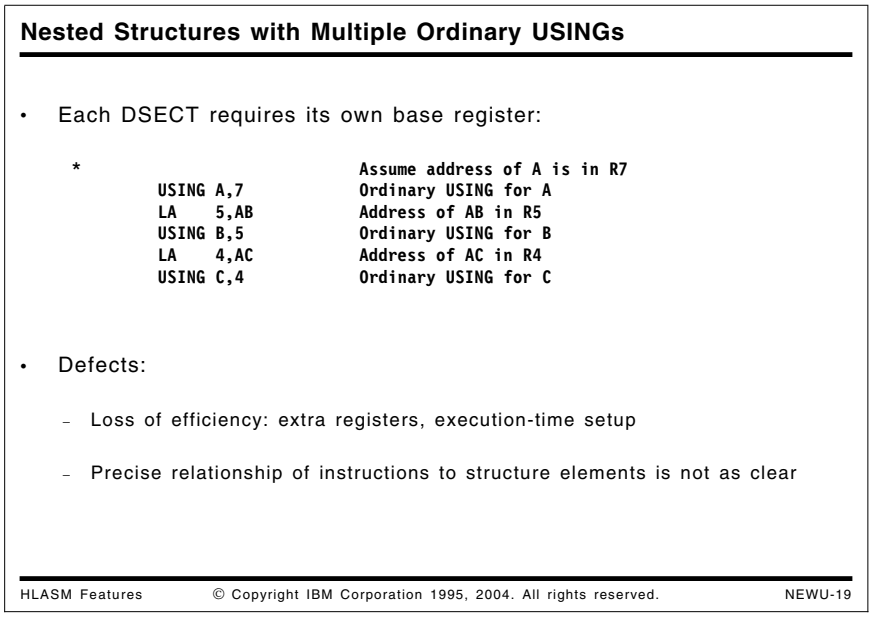
```
A      DSECT
A1     DS    24F
AB     DS    0C
AB_B1  DS    CL44      Nesting of B: field B1
AB_B2  DS    6D        Nesting of B: field B2
AB_B3  DS    4A        Nesting of B: field B3
A2     DS    6CL80
       - - -          etc.
```

It is evident that this technique can lead to maintenance problems, difficulties in understanding the code, and other defects described earlier:

- independence of the three structure definitions is lost
- the structure definition is more complex

- maintainers will have a harder time understanding what to change.

We will now show how to achieve the desired “nesting”, first with ordinary USINGs and then with labeled USINGs.



Example 4a: Structure Nesting with Multiple Ordinary USINGs

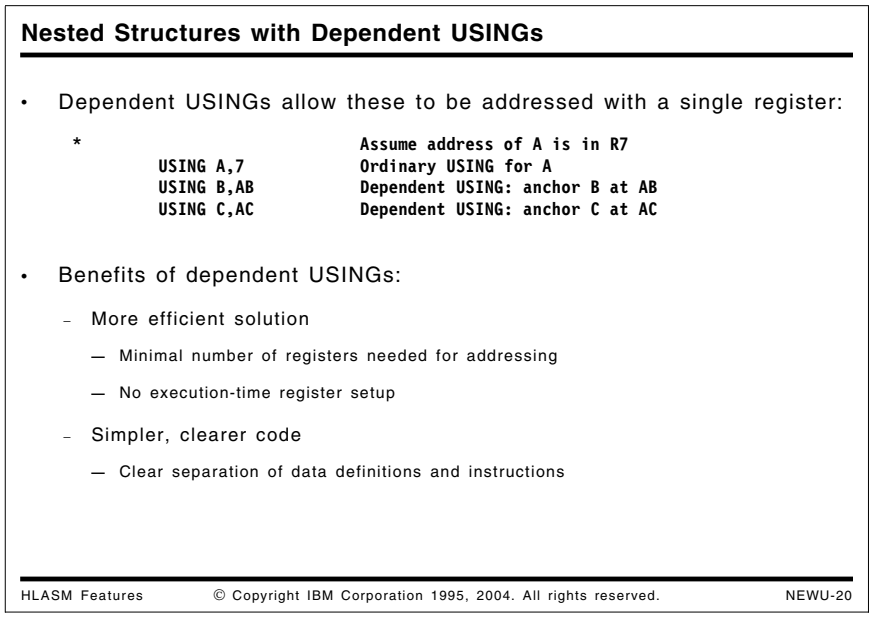
A typical code sequence for establishing addressability to the three DSECTs might look like this:

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

Figure 40. Referencing Nested DSECTs with Ordinary USINGs

While this code sequence will provide the desired addressing and DSECT nesting, it has some shortcomings:

- Two additional registers must be set up and used as base registers, even though it is known that the entire structure can be addressed with a single base register. This means that the two registers cannot be used for other purposes while the structures are being addressed.
- Additional time is required to initialize the extra base registers, leading to a loss in efficiency.
- It is not immediately evident that the desired “nesting” relationship is critically dependent on the instructions that set up the run-time addresses of B and C. It is possible that someone might change those instructions without realizing that the correctness of the nesting might be destroyed.



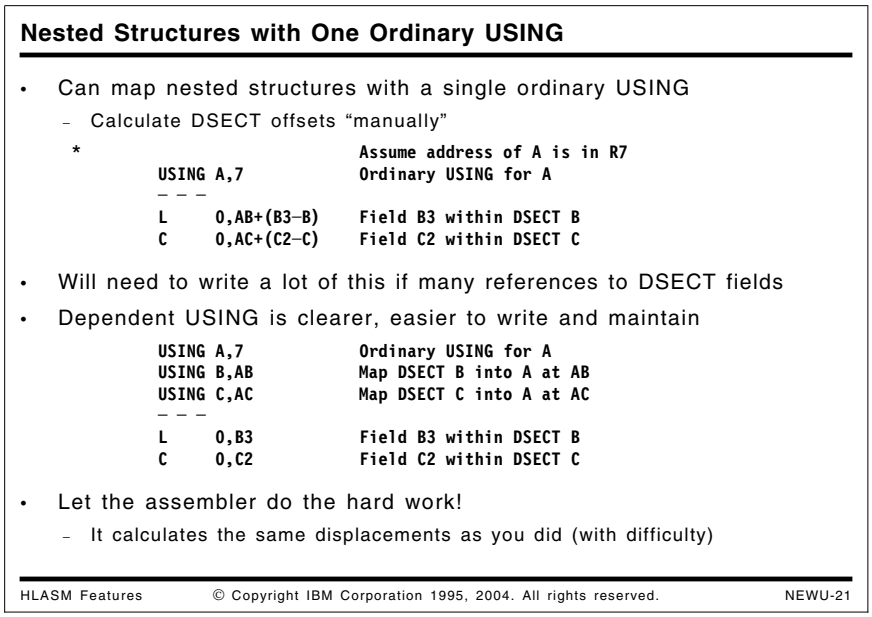
Example 4b: Structure Nesting with Dependent USINGs

The problem of correctly nesting DSECTs is easily solved with dependent USING statements. Rather than calculate the needed base addresses at program execution time, we can "bind" the nested DSECTs (B and C) in Figure 39 on page 63 to the containing DSECT (A) at assembly time:

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

Figure 41. Referencing Nested DSECTs with Dependent USINGs

In addition to saving the two additional base registers that were required when ordinary USING statements were specified (in Figure 40 on page 64), no instructions are needed to initialize those two registers, so the program can now gain other efficiencies by utilizing the registers for other purposes.



Example 4c: Structure Nesting with One Ordinary USING

The structure illustrated in Figure 38 on page 63 can be mapped with a *single* base register, if you are willing to calculate the needed offsets manually. Suppose you want to compare fields in the nested DSECTs B and C. You could write something like the sequence in the following figure.

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

Figure 42. Dependent USING Example 4c: Structure Nesting with One Ordinary USING

where the offsets of the fields B3 and C2 within their respective “owning” DSECTs must be added to the base locations of the DSECTs within DSECT A. While this code conserves base registers, It is likely to be difficult to understand and maintain, especially if many references to the fields within B and C are needed.

With dependent USINGs, the same instructions could be written much more understandably, as shown in this figure:

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

Figure 43. Dependent USING Example 4c: Structure Nesting with Dependent USING

Mapping Message Fields with the Message Itself

- Suppose your message has several fields to fill:

```
Messages CSect ,
Msg1     DC    C'This message for '
Msg1To   DC    C'xxxxxxxx'          Modified field
         DC    C' from '
Msg1From DC    C'yyyyyyyy'          Modified field
Msg1L    Equ   *-Msg1               Length of message
```

- Move the message to a buffer, then map the constant onto the buffer:

```
         Push  Using                Save USING status
         L     10,=A(Messages)      Point to messages
         Using Messages,10
         MVC   Buffer(Msg1L),Msg1   Move to buffer
         Drop  10                   Don't reference original
         Using Msg1,Buffer          Map original onto buffer
         MVC   Msg1To,ToName        Move "To" name
         MVC   Msg1From,FromName    Move "From" name
         Pop   Using                Restore USING status
```

- No need for separate DSects describing the message's fields

HLASM Features NEWU-22

Dependent USINGs Example 5: Message Fields

Dependent USINGs let you do interesting things. For example, suppose you need to construct a message into which several insertions must be made in known positions. First, construct the message (in a separate CSECT – this helps avoid some complexities with addressing):

```
Messages CSect ,
Msg1     DC    C'This message for '
Msg1To   DC    C'xxxxxxxx'          Modified field
         DC    C' from '
Msg1From DC    C'yyyyyyyy'          Modified field
Msg1L    Equ   *-Msg1               Length of message
```

Then, to complete the message, move this string to an output Buffer area, and you can the do this:

```
         Push  Using                Save USING status
         L     10,=A(Messages)      Point to messages
         Using Messages,10
         MVC   Buffer(Msg1L),Msg1   Move to buffer
         Drop  10                   Don't reference original
         Using Msg1,Buffer          Map original onto buffer
         MVC   Msg1To,ToName        Move "To" name
         MVC   Msg1From,FromName    Move "From" name
         Pop   Using                Restore USING status
```

The USING statement that maps the original constant onto the buffer avoids the need to define a DSECT to map each message, and also avoids manually calculating displacements and assigning base registers.

Dependent USINGs Example 6: A Personnel-File Record

The power of dependent USINGs is most evident in handling complex data records, especially when the structure of fields in later parts of the record depends on data values in earlier fields, or where repeated identically-structured fields (mapped by the same DSECT) appear several times within the outer record's structure.

Suppose our application program must reference various fields in records maintained in a “personnel” file. Each record contains information about an employee, and various fields within the record contain different kinds of information. (We will also use these data structures to illustrate the benefits of labeled dependent USINGs, in “Example 10: Personnel-File Record with Labeled Dependent USINGs” on page 83.)

First, let us define the basic layout of the employee record, by defining an Employee DSECT.

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

Figure 44. Dependent USING Example 6: Define a Personnel-File Record

This record contains information about the employee: a description of the person (and of the employee's spouse and first three dependents), work address, date of hire, work telephone, and so forth. Space has been reserved in the Employee DSECT for several other “nested” or “overlaid” DSECTS, to be described below.

The description of each person (employee, spouse, dependents) is similarly defined by a Person DSECT:

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

Figure 45. Dependent USING Example 6: Employee Record Person DSECT

The fields in the Person DSECT describe the person's name, date of birth, home address and telephone, and other items. Again, space has been reserved for three other “nested” DSECTs describing a date, an address, and a telephone number.

The remaining three DSECTs might be defined as follows. First, the Date DSECT:

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

Figure 46. Dependent USING Example 6: Employee Record Date DSECT

The last three statements are used to define the symbol DateF as a single field containing the entire contents of the three Date fields.

The Addr DSECT, describing a postal address, is defined in a similar way:

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

Figure 47. Dependent USING Example 6: Employee Record Address DSECT

Again, the last three statements are used to define the symbol AddrF as a single field containing the entire contents of all the Addr fields.

Finally we define The Phone DSECT, describing a commercial telephone number:

```
Phone    DSECT ,                  Define a Telephone field
PhArea   DS    CL3                Area Code
PhLocal  DS    CL7                Local number
PhExt    DS    CL4                Extension
LPhone   EQU   *-Phone            Length of Phone field
         ORG   Phone
PhoneF   DS   0CL(LPhone)         Full telephone number
         ORG   ,
```

Figure 48. Dependent USING Example 6: Employee Record Phone DSECT

As before, the last three statements define the symbol PhoneF to name a single field containing the entire contents of all the Phone fields.

At this point, it may be worth sketching the nesting of these various DSECTs. It is worth noting the following points:

- The Date DSECT appears at two different levels of nesting: the Date-of-Hire field (EHire) in the Employee DSECT is nested two levels deep, and the Date-of-Birth fields (PDoB) in each Person DSECT are nested three levels deep (because the Person DSECT is nested two levels deep in the Employee DSECT).
- Similarly, the Addr DSECT is nested two levels deep (as the employee's work address), and three levels deep (as the home-address field (PAddr) within each Person DSECT).

- Finally, the Phone DSECT is also nested two levels deep (PPhone, for the employee's home) and three levels deep (EPhoneW, the employee's work number).

  The nesting levels are shown in the upper right corners of the boxes in Figure 49 on page 71.

While this example may seem a bit complex, we will use it again in discussing labeled dependent USINGs, where the full power of those statements can be shown.

To show how dependent USINGs can help with mapping this structure, suppose such an employee record has been placed in main storage and its address has been placed in R10; we now wish to manipulate various fields within the record. The necessary DSECT addressing can be established as follows:

```
         USING Employee,10       R10 points to Employee record
         USING Person,EPerson    Anchor Person DSECT at EPerson field
         USING Date,PDoB         Anchor Date DSECT at PDoB field
         USING Addr,PAddr        Anchor Addr DSECT at PAddr field
         USING Phone,PPhone      Anchor Phone DSECT at PPhone field
```

Figure 50. Dependent USING Example 6: Anchoring DSECTs within Employee Record

These five USING statements provide addressability to five different DSECTS:

- The Employee DSECT is based on an ordinary USING statement with base register 10. All other implied address resolutions within the Employee DSECT will be resolved using R10 as the base register.
- The Person DSECT is anchored by the first dependent USING at the Eperson field in the Employee DSECT.
- Within the Person DSECT, the Date DSECT is anchored by the second dependent USING at the PDoB field in the Person DSECT.
- Within the Person DSECT, the Addr DSECT is anchored by the third dependent USING at the PAddr field in the Person DSECT.
- Finally, within the Person DSECT, the Phone DSECT is anchored by the fourth and last dependent USING at the PPhone field in the Person DSECT.

We can now use these definitions to access and manipulate the fields described by those five DSECTs, as in the following statements:

```
         CLC   PGname,Input_Name  Compare name in record to input value
         - - -
         MVC   PhExt,=CL4' '      Blank out phone extension field
         - - -
         CLC   AZip,=C'95141'     Check for given Zip Code
```

Figure 51. Dependent USING Example 6: Using fields within Employee Record

All of the symbolic references to fields in any of the five DSECTs will be resolved with a single base register, so long as the size of the Employee record does not exceed 4096 bytes. (If it does, the problem is easy to fix: simply add another base register to the ordinary USING statement in Figure 50, and another 4096 bytes will be addressable automatically.)

The primary limitation of the uses of dependent USINGs shown in this example is that only a single instance of each DSECT is addressable at any one time. In many applications this may be entirely adequate; if not, labeled dependent USINGs (as described at “Example 10: Personnel-File Record with Labeled Dependent USINGs” on page 83) will help.

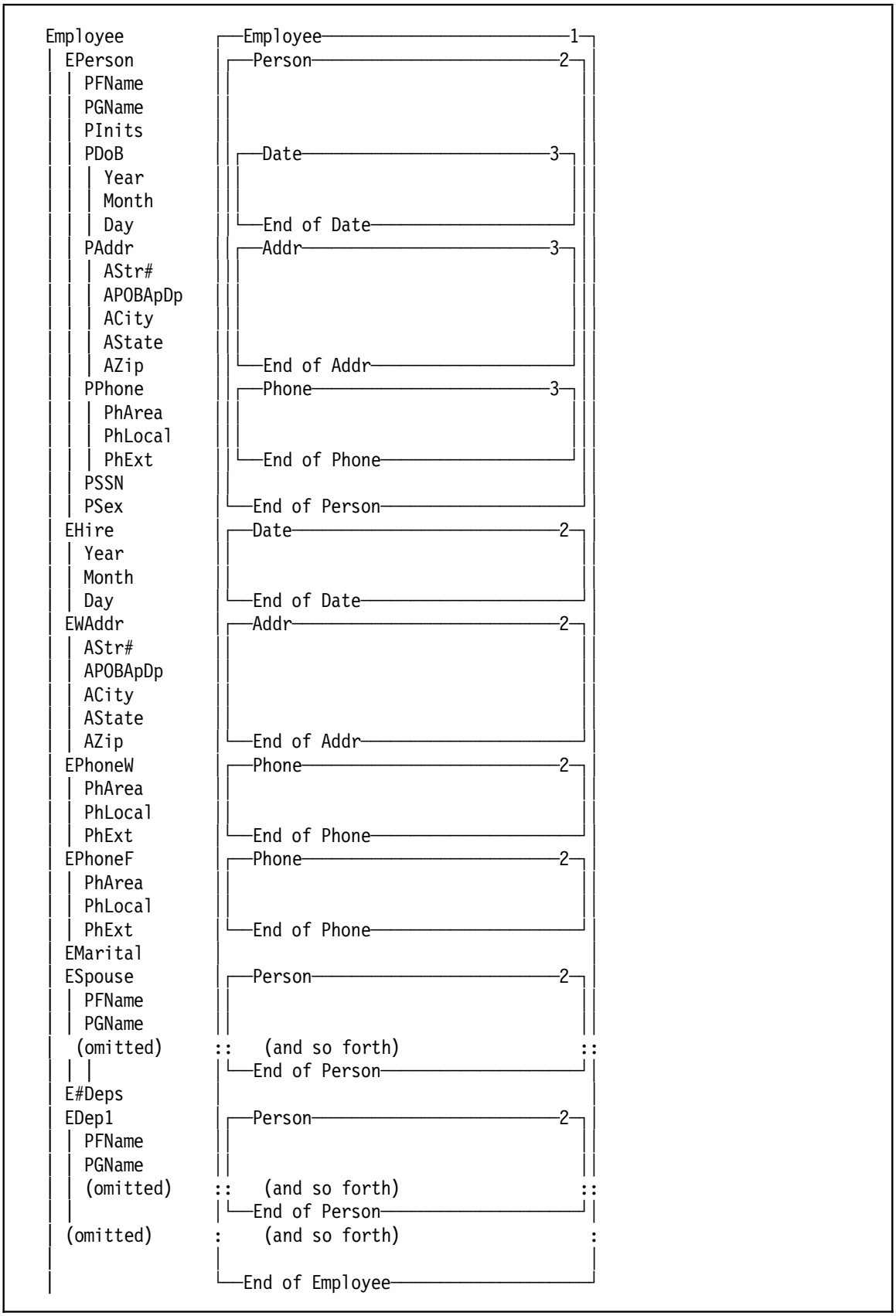


Figure 49. Dependent USING Example 6: DSECT Nesting in Employee Record

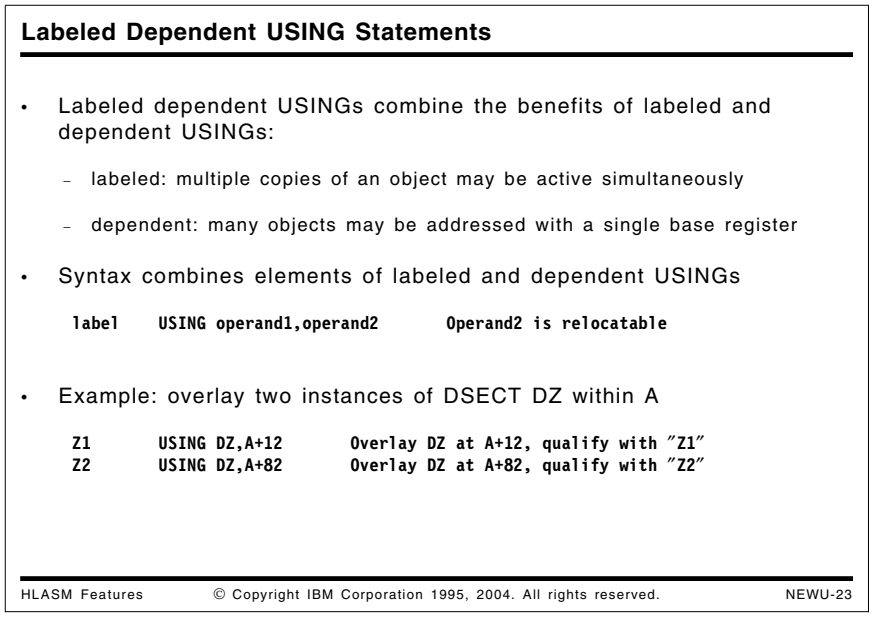
If ordinary USING statements had been required in Figure 50, the resulting burden on the general purpose registers might have been much more severe. Statements such as the following might have been required:

```
         USING Employee,10        R10 points to Employee record
         LA    9,Eperson          Address of EPerson field
         USING Person,9           Anchor Person DSECT at EPerson field
         LA    8,PDoB             Address of PDoB field
         USING Date,8             Anchor Date DSECT at PDoB field
         LA    7,PAddr            Address of PAddr field
         USING Addr,7             Anchor Addr DSECT at PAddr field
         LA    6,PPhone           Address of PPhone field
         USING Phone,6            Anchor Phone DSECT at PPhone field
```

Figure 52. Dependent USING Example 6: DSECTs within Employee Record with Ordinary USINGs

It can be seen that the coding is likely to be less efficient, and also that the number of opportunities for misunderstanding and error has also increased.

Labeled Dependent USING Statements



Labeled dependent USINGs combine the benefits of labeled USINGs and dependent USINGs:

- multiple copies of an object may be active simultaneously (labeled)
- many objects may be addressed with a single base register (dependent).

We will begin this discussion of labeled dependent USINGs with several rather simple examples that are intended to illustrate both the problems encountered with ordinary USINGs, and how High Level Assembler can help you to solve them with labeled dependent USINGs.

Definition of Labeled Dependent USINGs

The syntax of a labeled dependent USING is evident from its name: a qualifying label is required in the name field of the USING statement, and a relocatable second operand is required to indicate where the first operand is to be “anchored” or “based”.

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

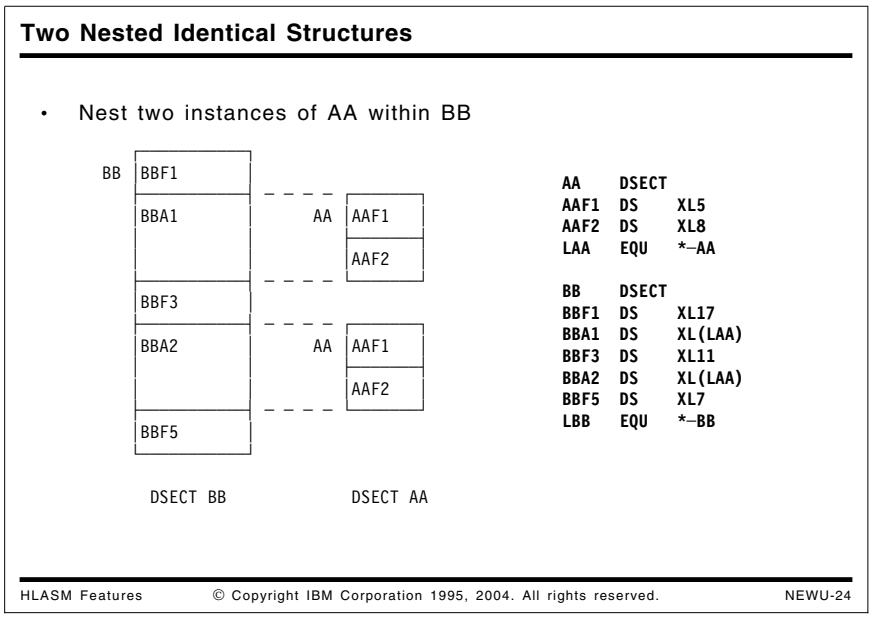
Figure 53. Syntax of a Labeled Dependent USING Statement

As with unlabeled dependent USINGs, the second operand must be addressable with reference to an ordinary USING statement somewhere earlier in the program. (The second operand may itself be a qualified symbol, which allows you to specify multiple levels of dependence.)

We can use such labeled dependent USINGs to map or overlay two different instances of a DSECT on a single area of storage. For example, suppose we have a DSECT named DZ that we wish to “overlay” at two different positions within a third program component named A:

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

Then, references to fields in the two instances of DZ can be distinguished by using the qualifiers Z1 and Z2.



Example 7: Nesting Two Identical Structures Within a Third

Suppose we have a data structure composed of an “outer” structure, whose components contain several data items including two identical sub-structures that we prefer to describe with a single DSECT.

Now, we must manage the “outer” structure described by the DSECT named BB, in which there are two sub-structures described by the DSECT named AA. First, we will define the “inner” DSECT AA:

```
AA    DSECT
AAF1  DS    XL5             Field 1 in AA
AAF2  DS    XL8             Field 2 in AA
LAA   EQU   *-AA            Length of AA
```

Figure 54. Labeled Dependent USINGs Example 7: Nested DSECT Definition (1)

The “outer” DSECT BB can be defined similarly:

```
BB    DSECT
BBF1  DS    XL17            Field 1 in BB
BBA1  DS    XL(LAA)         Field 2 in BB = 1st copy of AA
BBF3  DS    XL11            Field 3 in BB
BBA2  DS    XL(LAA)         Field 4 in BB = 2nd copy of AA
BBF5  DS    XL7             Field 5 in BB
LBB   EQU   *-BB            Length of BB
```

Figure 55. Labeled Dependent USINGs Example 7: Nested DSECT Definition (2)

The positions at which the two sub-structures defined by AA will be located are named BBA1 and BBA2, respectively. We will examine several approaches to managing the description and addressing of the data elements in these structures:

- first, we will consider ordinary USINGs and the problems they present;

- second, we will examine the implications of DSECT “renaming”;
- finally, we will show how labeled dependent USINGs provide a solution free of the defects of the previous approaches.

Example 7a: Nesting Two Identical Structures with Ordinary USINGs

To address these three structures with ordinary USINGs, we would need to provide three base registers and three USING statements. However, we are faced with the problem already discussed in “Example 1: With Ordinary USINGs” on page 47 above: we wish to manage two active instances of the DSECT AA, and only one active instance is allowed by ordinary USINGs.

Example 7b: Nesting Two Identical DSECTs with DSECT Renaming

The limitations of ordinary USINGs can be bypassed by making a second copy of AA, addressing it and the “original” copy with separate USINGs.

```
AB    DSECT
ABF1  DS    XL5             Field 1 in AA (but named AB)
ABF2  DS    XL8             Field 2 in AA (but named AB)
LAB   EQU   *-AB            Length of AA (but named AB)
```

Figure 56. Labeled Dependent USINGs Example 7b: Renamed DSECT Definition

The three DSECTs can now be addressed with statements like the following:

```
        USING BB,10           R10 points to BB
        LA    11,BBA1         R11 points to 1st copy of AA
        USING AA,11           USING for 1st copy of AA
        LA    12,BBA2         R12 points to 2nd copy of AA
        USING AB,12           USING for 2nd copy of AA (named AB)
```

The defects and difficulties involved in attempting to use ordinary USINGs in this context have been thoroughly described; maintenance and readability problems are substantially increased when more than one name is used for the same thing.

Example 7c: Nesting Two Identical DSECTs with Labeled USINGs

A better solution involves labeled USINGs, which allow the two instances of AA to be addressed using a only a single definition of DSECT AA.

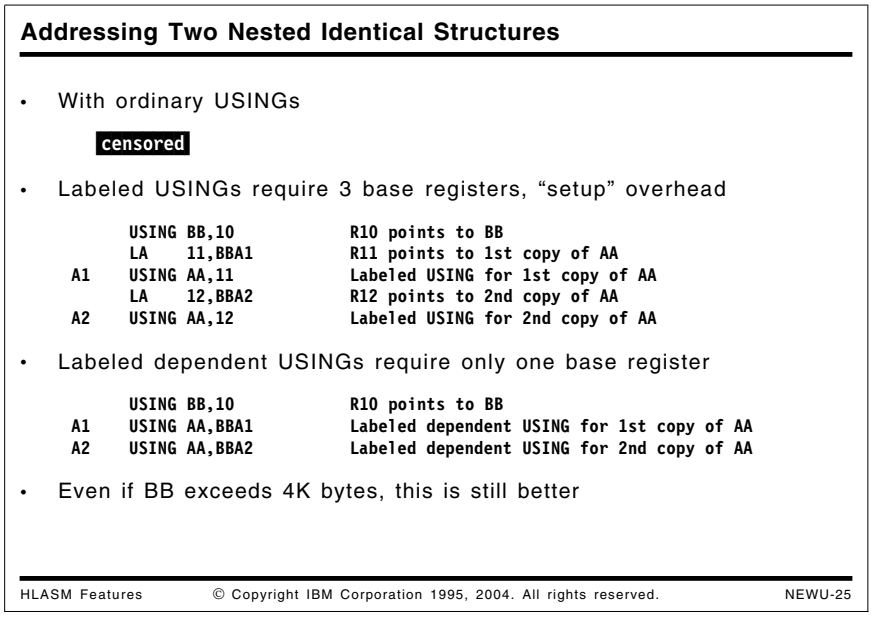
```
        USING BB,10           R10 points to BB
        LA    11,BBA1         R11 points to 1st copy of AA
A1      USING AA,11           Labeled USING for 1st copy of AA
        LA    12,BBA2         R12 points to 2nd copy of AA
A2      USING AA,12           Labeled USING for 2nd copy of AA
```

Figure 57. Labeled Dependent USINGs Example 7c: Nesting with Labeled USINGs

Note that the implied addresses in both LA instructions will be resolved with register 10 as the base register.

The only remaining defect in this example is the requirement to use three addressing registers when only one is actually needed; labeled dependent USINGs provide the desired saving.

Example 7d: Nesting Two Identical DSECTs with Labeled Dependent USINGs



the proper solution involves labeled dependent USINGs, which allow the entire structure and all its components to be addressed with the minimum number of registers, and with proper naming for all components.

Assume again that the address of the containing structure BB has been placed in R10; then the appropriate addressing and addressability statements might be written as follows:

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

Figure 58. Labeled Dependent USINGs Example 7d: Nesting with Labeled USINGs

The first instance of AA is “based” at BBA1, and references to its components can be made using qualifying label A1. Similarly, the second instance of AA is based at BBA2, and its components can be qualified with A2. References to the various fields in the three structures can then be made freely, and only a single register is needed to address the entire structure.

```
        MVC   BBF1,A1.AAF2      Move to BBF1 from first instance of AAF2
        CLC   A2.AAF1,A1.AAF1   Compare AAF1 fields in two instances of AA
        UNPK  A1.AAF2,BBF5      Unpack from BBF5 to first instance of AA
        MVZ   A1.AAF1,A2.AAF1   Move zones from second AAF1 to the first
```

Figure 59. Labeled Dependent USINGs Example 7d: Nesting with Ordinary USINGs

Example 8: Multiple Nesting of Identical Structures

If the number and nesting of data structures increases even slightly, it can be seen that there can be difficult problems to solve in addressing the components. For example, suppose we wish to establish a data structure in which an outermost structure E contains three copies of a structure D, each of which in turn contains three copies of a structure F. This might look somewhat like the following figure.

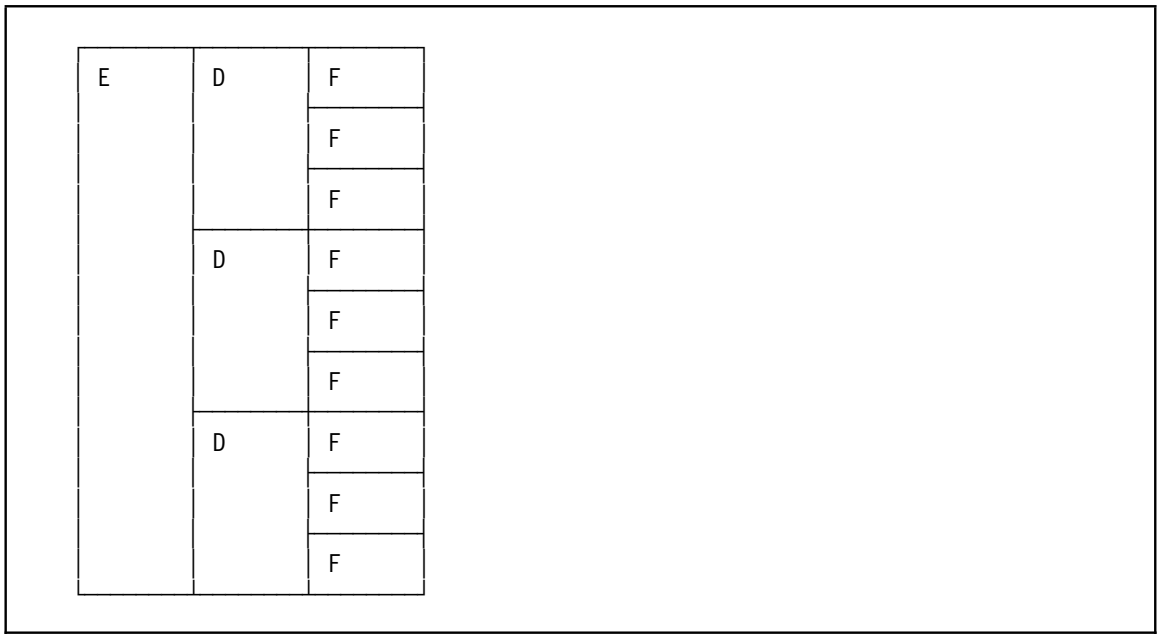
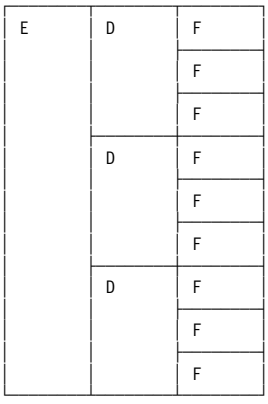


Figure 60. Multiply-Nested Data Structures

If we wish to use ordinary USINGs to address all components of this set of structures, we will have to allocate *thirteen* registers as base registers. This is beyond the capabilities of most programs, so that we will be forced to use “unnatural” solutions if we are restricted to ordinary USING statements. Dependent USINGs will help only a little, because of the high degree of repetition among the inner structures.

Multiple Nested Structures




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

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

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

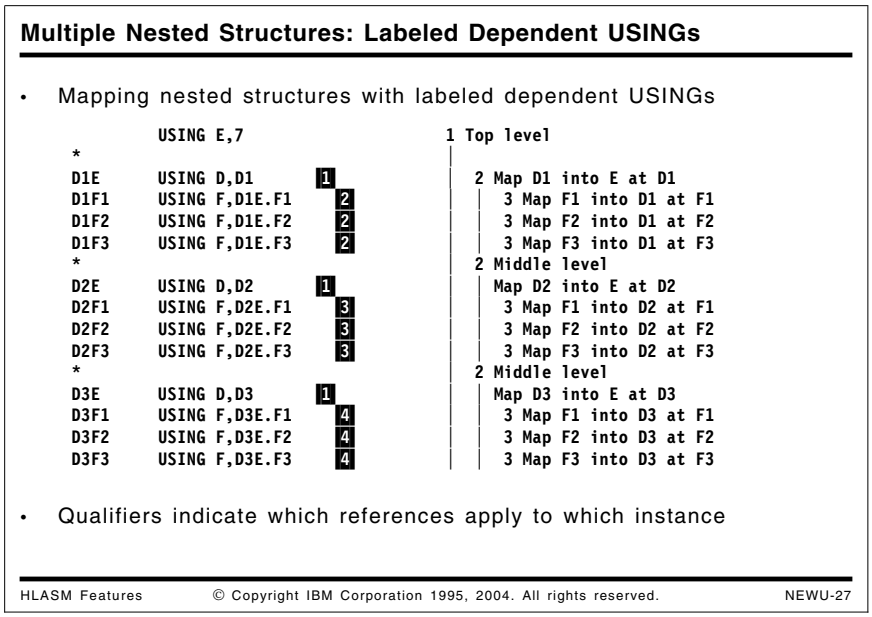
- Problems:
  - Multiple instances of structures D and F
  - Ordinary or labeled USINGs require 13 base registers!

HLASM Features NEWU-26

Suppose the DSECTs are named F, D, and E, and are nested so that three copies of F are to be contained in each D, and three copies of each D are to be contained in E.

```
F       DSECT ,                 Third-level DSECT (bottom level)
X1      DS    XL5                 First data element
X2      DS    XL5                 Second data element
LF      EQU   *-F               Length of F

D       DSECT ,                 Second-level DSECT (middle level)
F1      DS    XL(LF)              First third-level DSECT
F2      DS    XL(LF)              Second third-level DSECT
F3      DS    XL(LF)              Third third-level DSECT
LD      EQU   *-D               Length of D

E       DSECT ,                 First-level DSECT (top level)
D1      DS    XL(LD)              First second-level DSECT
D2      DS    XL(LD)              Second second-level DSECT
D3      DS    XL(LD)              Third second-level DSECT
```

Figure 61. Labeled Dependent USINGs Example 8: Double Nesting DSECT Definitions



It can be seen that addressing this structure with ordinary USINGs is nearly impossible to do cleanly, and that a solution with labeled USINGs also requires *thirteen* registers to address the thirteen different active DSECTs. The only viable solution is to use labeled dependent USINGs.

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

Figure 62. Labeled Dependent USINGs Example 8: Double Nesting DSECT Definitions

While this example looks somewhat complicated, it has a simple basic structure. First, notice the three labeled dependent USING statements (tagged with **1**) that map the middle-level DSECT named D into the outermost DSECT named E. Because there will be three instances of D simultaneously active, the qualifier labels D1E, D2E, and D3E are used to distinguish the first, second, and third instances of D within E. The three instances of D are anchored at the positions within E defined by the fields named D1, D2, and D3, respectively. (See Figure 61 on page 78.)

The three innermost instances of the DSECTs described by F are mapped into the three instances of D in a similar way. For example, the three labeled dependent USING statements for the first instance of D (tagged with **2**) anchor the three instances of F within D at the positions labeled F1, F2, and F3 respectively. (Again, referring to Figure 61 on page 78 may help.) Because there will be three different active instances of those labels, we must use the qualifier label D1E to qualify the references to F1, F2, and F3. Thus the second operand of each of the three labeled dependent USING statements tagged **2** is therefore qualified with D1E. The labels on those three USINGs – D1F2, D1F2, and D1F3 – will be used to qualify references to the first three of the nine possible instances of the fields X1 and X2. (The notation implied by these qualifiers is that D1F3 means “first D, third F”.) Appropriately chosen qualifiers can also help you to understand your program more easily.

It is interesting to observe that qualified symbols may themselves be used in labeled dependent USING statements that themselves define other qualifiers!

The mappings of the second and third sets of instances of the DSECT named F are defined similarly, in the sets of three labeled dependent USINGs tagged **3** and **4** respectively. The qualifying labels D2F1 through D3F3 are then used to qualify references to the fields within the DSECT named F.

Multiple Nested Structures: Referencing Fields

- All symbol references to individual fields are qualified:

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

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

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

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

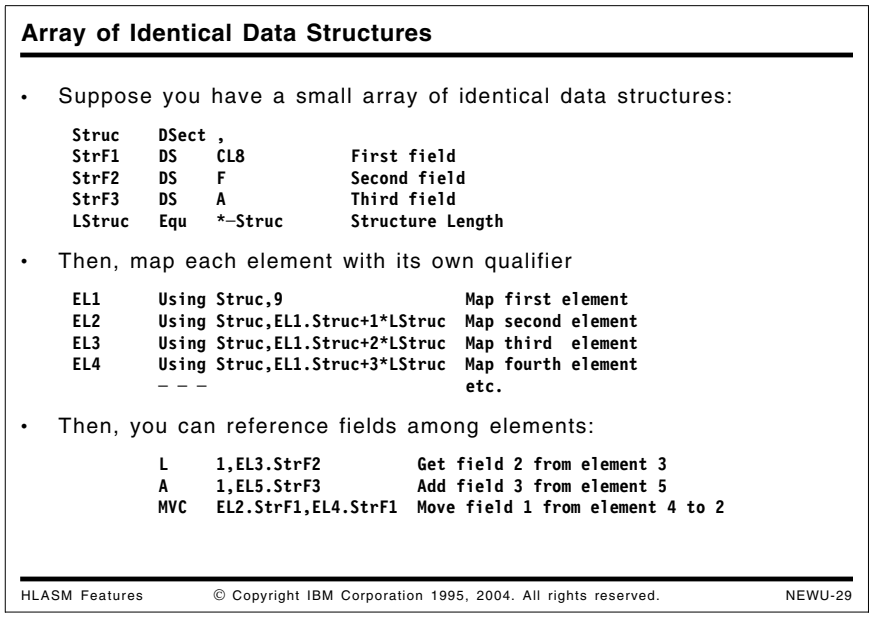
HLASM Features NEWU-28

We could then write instructions to reference these fields, with appropriate qualifiers:

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

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

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

Figure 63. Labeled Dependent USINGs Example 8: Putting the USINGs to Work

As you can appreciate, coding instructions such as these with ordinary USING statements would much more difficult to write and understand.



Mapping an Array of Identical Data Structures

Suppose you have a small array of identical data structures, and need to be able to refer to several fields within different elements. First, you could define a DSECT describing the data structure:

```
Struc    DSect ,
StrF1    DS    CL8          First field
StrF2    DS    F            Second field
StrF3    DS    A            Third field
LStruc   Equ   *-Struc       Structure Length
```

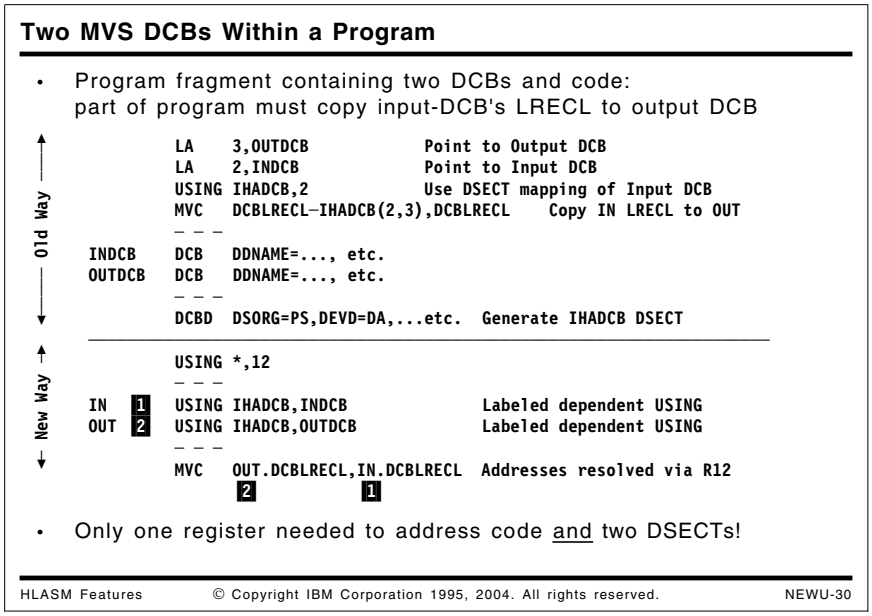
Suppose General Register 9 contains the address of the storage area containing the first element of the array. You can then map each element of the array using Labeled Dependent USINGs:

```
EL1      Using Struc,9                  Map first element
EL2      Using Struc,EL1.Struc+1*LStruc Map second element
EL3      Using Struc,EL1.Struc+2*LStruc Map third  element
EL4      Using Struc,EL1.Struc+3*LStruc Map fourth element
         - - -                          etc.
```

Then, you can refer freely to fields within each element of the array:

```
         L     1,EL3.StrF2          Get field 2 from element 3
         A     1,EL5.StrF3          Add field 3 from element 5
         MVC   EL2.StrF1,EL4.StrF1  Move field 1 from element 4 to 2
```

Example 9: Two MVS Data Control Blocks Within a Program



This small example shows how one might combine the benefits of labeled and dependent USINGs in a single program, by using the program base register to address two embedded structures as well.

Assume that there are two MVS Data Control Blocks (DCBs) addressable in the same program as are its other components. If only ordinary USINGs are available, three registers must be used for addressability: one for the code, and one for each DCB. Furthermore, only one of the DCBs can be mapped with the IHADCB DSECT, because both cannot be mapped simultaneously. Thus, a typical code sequence might look like this:

```
LA    3,OUTDCB            Point to Output DCB
LA    2,INDCB             Point to Input DCB
USING IHADCB,2
MVC   DCBLRECL-IHADCB(2,3),DCBLRECL    Copy IN LRECL to OUT
```

Figure 64. Labeled Dependent USING Example 9: Addressing With Ordinary USINGs

The defects in this technique are that three registers must be assigned, and one of the operands in the MVC instruction must be addressed with explicitly assigned base and displacement.

Suppose we also wish to make symbolic references to fields in both DCBs at the same time. The two labeled dependent USINGs illustrated below permit fully symbolic references to both DCBs at the same time, and without a need for additional registers.

```
         USING *,12
         - - -
IN   1   USING IHADCB,INDCB              Labeled dependent USING
OUT  2   USING IHADCB,OUTDCB             Labeled dependent USING
         - - -
         MVC   OUT.DCBLRECL,IN.DCBLRECL  Addresses resolved via R12
               2           1
         - - -
INDCB    DCB   DDNAME=..., etc.
OUTDCB   DCB   DDNAME=..., etc.
         - - -
         DCBD  ...,etc.                  Generate IHADCB DSECT
```

Figure 65. Labeled Dependent USING Example 9: Addressing Everything with One Register

While most of the previous examples have used data structured defined by DSECTs to illustrate various uses of dependent and labeled dependent USINGs, this example shows that you can map a DSECT “almost anywhere”. The base address (in the first USING operand) may be “anchored” at any addressable location, including the “code” portion of a program.

Example 10: Personnel-File Record with Labeled Dependent USINGs

Personnel-File Employee Record

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

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

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

HLASM Features NEWU-31

We will now return to the example introduced in “Dependent USINGs Example 6: A Personnel-File Record” on page 68, with with labeled dependent USINGs as our primary tool for mapping the complex data structures illustrated in Figure 49 on page 71.

Assume that the Employee, Person, Date, Addr, and Phone structures have been defined as illustrated in Figures 44 through 49 (found on pages 68 through 71).

Personnel-File Employee Record: “Person” Fields

- An individual is described by the Person DSECT:

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

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

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

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

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

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


Now, we will consider examples that require addressing multiple active instances of the inner structures in this Employee data structure. For all of the following examples, we will assume that some other part of the program has placed the Employee record into storage at an address carried in General Register 10.

Personnel-File Record Example 10a: Comparing Birth Dates

Personnel-File Employee Record: Comparing Birth Dates

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

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

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

* Example 1: Compare Employee and Spouse Dates of Birth

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

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

HLASM Features NEWU-34

Suppose our first requirement is to write some code to compare the birth dates of the employee and the employee's spouse. Because the birth date is a component of the Person DSECT, we must establish mappings of the two instances of that DSECT. In the following figure, this is done with two labeled dependent USING statements:

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

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

* Example 1: Compare Employee and Spouse Family Dates of Birth

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

Figure 66. Labeled Dependent USINGs: Comparing Dates of Birth

The first labeled dependent USING statement (indicated by key 1) maps the Person structure onto the Employee record at the position defined by EPerson; this will describe information about the employee. The second labeled dependent USING statement (indicated by key 2) maps the Person structure onto the Employee record at the position defined by ESpouse, and describes information about the employee's spouse.

The actual comparison operation is done with the CLC instruction. Note that both instances of the Person structure are nested at the same level within the Employee structure, so that similar styles of qualification are used for the two occurrences of the symbol PDoB.

Personnel-File Record Example 10b: Comparing Dates

Personnel-File Employee Record: Comparing Dates

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

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

 EHD [3]  USING Date,EHire          Overlay Date DSECT on Date of Hire

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

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

          DROP  EHD,DD1             Remove both date associations
```

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

HLASM Features NEWU-35

Suppose our second requirement is to check the employee record to see of the date of birth of the first dependent is later than the employee's date of hire. In this case, we must deal with two different levels of nesting of the Date structure: one (the employee's date of hire) is nested directly within the Employee DSECT at the position labeled EHire, while the birth date of the first dependent is nested (in the Employee DSECT at position EDep1) within the first-dependent Person DSECT at position PDoB. Thus, we will need additional labeled dependent USINGs to properly establish addressability to the PDoB field.

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

EHD [3]  USING Date,EHire          Overlay Date DSECT on Date of Hire

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

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

         DROP  EHD,DD1             Remove both date associations
```

Figure 67. Labeled Dependent USINGs: Comparing Date Fields

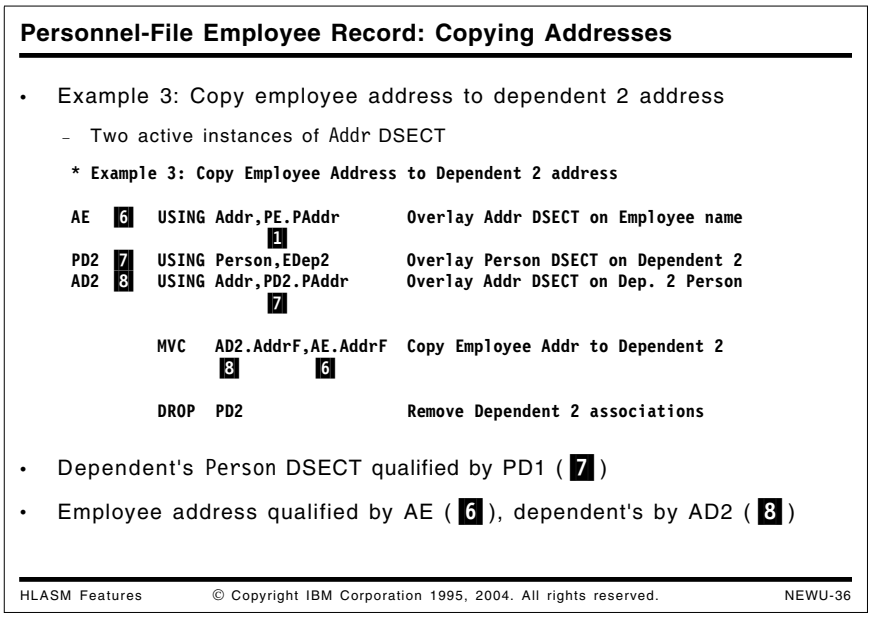
In order to map the two instances of the Date DSECT, we first issue a labeled dependent using with label EHD to describe the employee's date of hire (at EHire, with key 3). Then, to map the first dependent's date of birth, we must first map a Person DSECT onto the employee record (at EDep1, with key 4) with label PD1. Finally, within that Person DSECT, we describe the person's date of birth by mapping the Date DSECT onto the Person structure (at PDoB, key 5) with label DD1.

The comparison instruction CLC refers to two complete date fields DateF, qualified to associate one with the date of hire and the other with the first dependent's date of birth.

This example, while not obvious at first encounter, is worth some study: it shows how you can utilize labeled dependent USINGs to map very complex structures in a natural, readable way that does not require you to understand what pointers may have been established in which registers some pages earlier in the listing.

Note that the DROP statement, by specifying the two qualifiers, removes the mappings of both Date DSECTs.

Personnel-File Record Example 10c: Copying Addresses



Suppose our third requirement is to update the employee record so that the employee's home address is assigned to be the same as that of the second dependent. In this case, the addresses are at the same level of nesting: the Addr structure: the person's home address is nested within the Person DSECT at the position labeled PAddr. This means that we must provide addressability to two different Addr DSECTs.

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

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

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

         DROP  PD2                 Remove Dependent 2 associations
```

Figure 68. Labeled Dependent USINGs: Copying Addresses

The technique used here is like that of the previous example: we establish addressability to the instances of the Addr DSECT within the two instances of the Person DSECT, one for the employee (at EPerson, qualified with PE, at key 1 in Figure 66 on page 85) and one for the second dependent (at EDep2, qualified with PD2, key 7). Within the two instances of the

Person DSECT are the two instances of the Addr DSECT, one for the employee at PE.PAddr, qualified by AE, key 6), and one for the second dependent (at PD2.PAddr, key 8). The move instruction then uses these “address qualifiers” AE and AD2 to qualify the names of the field to be moved, AddrF.

The DROP statement specifies the label PD2. Because the labeled dependent USING with the AD2 qualifier was based (or “anchored”) on that with qualifier PD2, DROPping the latter automatically causes the former to be dropped also.

Summary of USING Statements

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

Summary of USING Statements ...

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

Summary of USING Statements

We can now summarize the properties and behavior of the four types of USING statement in the following table:

Figure 69. Summary of USING Statements

USING Type	Label	Register Usage	Operand 1 Based on	Operand 2	Operand 2 Location in Storage	Number of Instances of Active Objects
Ordinary	no	1 register per object	register	absolute [0,15]	anywhere in storage	only one active instance of an object at a time
Labeled	yes	1 register per object	register	absolute [0,15]	anywhere in storage	as many active instances of an object as registers assigned
Dependent	no	multiple objects per register	operand 2	relocatable, addressable	within addressability range of ordinary USINGs	multiple active objects of different types
Labeled Dependent	yes	multiple objects per register	operand 2	relocatable, addressable	within addressability range of ordinary USINGs	multiple different active objects of the same or different types

As the above table indicates, High Level Assembler provides a rich and complete selection of choices to help you manage addressability concerns in your programs.

DROP Statement Extensions

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

- Examples:

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


DROP Statement Extensions

The DROP statement has been extended in support of the above enhancements to the USING statement.

The properties of the DROP extensions are summarized below:

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

Figure 70. Summary of DROP Statement Behavior

These may be described as follows:

- Ordinary USINGs

  The normal rules for DROP statements apply, and the entry for the specified register is removed from the assembler's Using Table.

- Labeled USINGs

  The qualifying label from a previous labeled USING is specified as the operand of the DROP statement. Only the USING with that qualifier is inactivated; other USINGs specifying the same base register (if any) are still active.

- Dependent USINGs

  The syntax of the ordinary DROP statement is used: a register is specified as the operand. If any further dependent USINGs are based on the same register, they are automatically dropped at the same time. The assembler's Using Table entry for that register is removed.

- Labeled Dependent

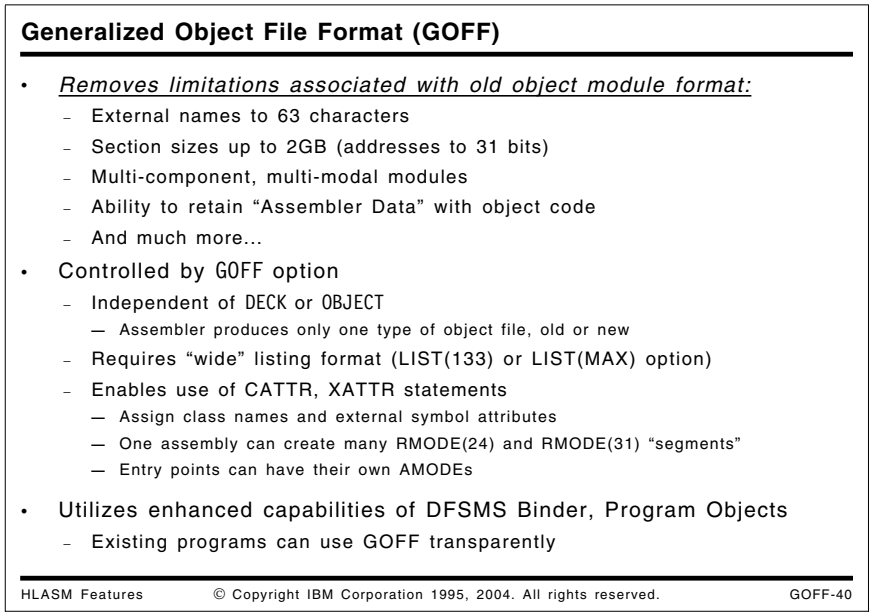
  The qualifying label from a previous labeled or labeled dependent USING is specified as the operand of the DROP statement. Any dependent or labeled dependent USINGs that relied on the qualifying label are also dropped. Other USINGs specifying the same base register (if any) are still active.

Summary

As the examples have illustrated, the capabilities of the new USING statements provided with High Level Assembler for MVS & VM & VSE can support programming techniques of considerably greater power, clarity, expressiveness, and accuracy. They can help you to achieve many of the goals of any programming language.

Generalized Object File Format (GOFF)

When the GOFF option is specified, High Level Assembler will create an object file in a new "extended" or "generalized" format (GOFF). This new format is considerably more flexible than the old, familiar "object module" format (OBJ), which suffers from many limitations.



High Level Assembler and all previous IBM assemblers for the System/360/370/390 family of processors produce the familiar card-image object-module format (OBJ), as requested by the DECK or OBJECT options. The GOFF option lets you take advantage of the capabilities of the new object file format and their support by the DFSMS Binder and its new "Program Object" format for executables. A program object has a two-dimensional structure (whereas load modules and traditional object modules are intended for one-dimensional structures).

GOFF option

This specifies that the object file should be written in the new format. The GOFF option simply determines the *format* of the object file; the DECK and OBJECT options select its destination. The new and old object module formats are mutually exclusive.

The ADATA sub-option requests that "Assembler Data" be included in the object file, so it can be placed in program object classes by the Binder.

GOFF requires a "wide" listing format, specified implicitly or explicitly, either by the LIST(133) option or by LIST(MAX) with a print-line record length of at least 133 characters.

Among the enhancements provided by the GOFF option are:

- External names up to 63 characters long
- Section lengths up to 2GB, and addresses and lengths 31 bits long
- Multi-component, multi-modal modules, with a single assembly capable of producing independently loadable "segments" with different RMODEs
- AMODE attributes may be assigned to ENTRY points (not just to control section names)
- "Assembler Data" (SYSADATA) may be included in the object stream, allowing both the object code and all associated descriptive data to be kept in one place.

The program object format supported by the binder is more complex than the traditional load module format. Load modules are essentially a single loadable segment of code (even when overlay format is used), whereas program objects may contain several loadable segments. This is best visualized by treating load modules as one-dimensional executables, while program objects are two-dimensional:

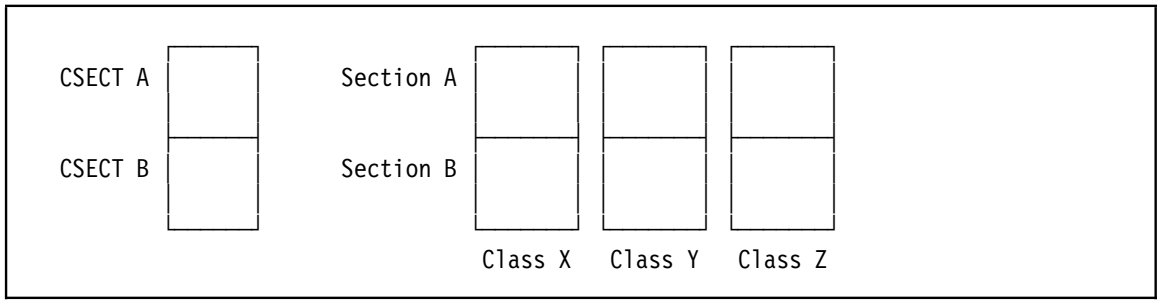


Figure 71. Sketch of Load Module vs. Program Object

Because code and data destined for a program object must specify the class *and* section to which it belongs, the CATTR statement specifies the class name and the attributes of the class.

HLASM provides two statements that describe the properties of symbols and generated text in GOFF files; both require the GOFF option:

CATTR statement
The CATTR statement controls the placement of machine language instructions and data into specified classes with specified attributes.

XATTR statement
The XATTR statement, when used in combination with the GOFF allows you to assign special attributes to external symbols.

The default values assigned to class attributes cause programs without CATTR statements to be bound in the same way as if the GOFF option had not been specified.

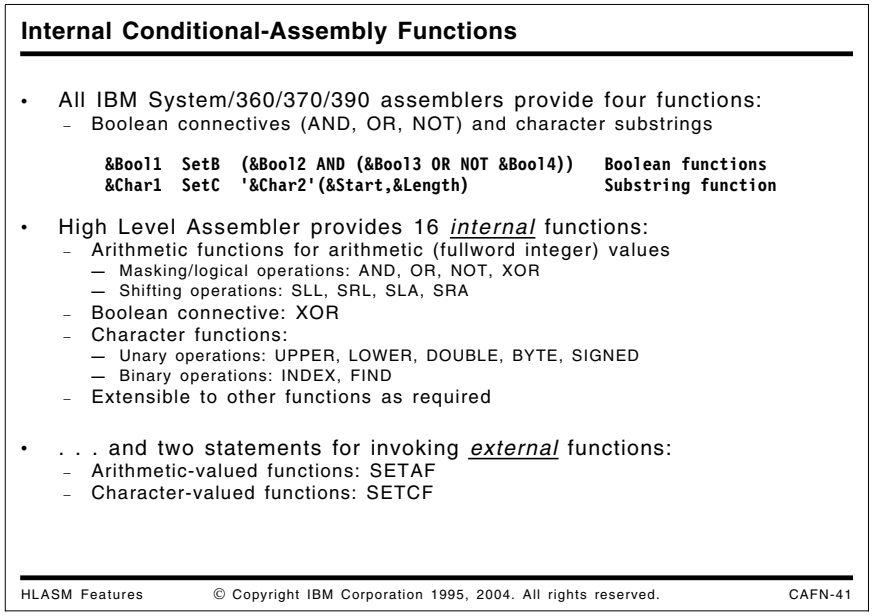
External Symbol Dictionary Listing Enhancements

The listing format is modified when the GOFF option is specified:

- The ESD listing displays extra data.
- The ALIAS statement permits 64-character external names when the GOFF option is active.
- The source and object code listing displays location counter and symbol values using eight hexadecimal digits.
- The RLD listing displays 8-digit address values.

Conditional-Assembly Functions

High Level Assembler provides a large number of enhancements to the functions available for programming the conditional assembly language. Among the functions available in the original (pre-HLASM) assembler language are the Boolean connectives AND, OR, and NOT, and the "substring" function (or, more properly, operator) for character data. These functions are "internal" in the sense that they have no interaction with the assembly environment.

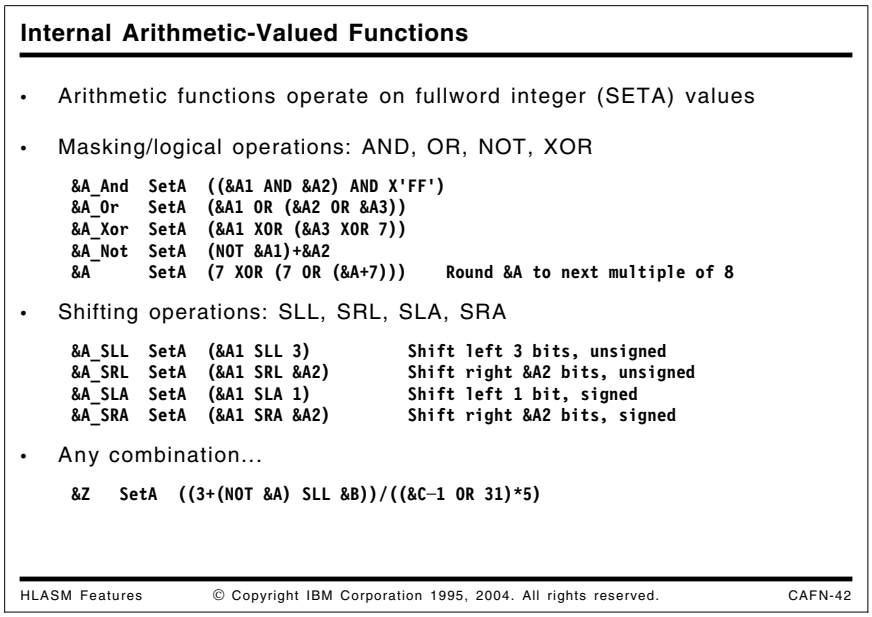


There are two basic classes of new functions: internal and external. The internal functions are written much like the existing Boolean connectives; we will describe examples in "Internal Conditional-Assembly Functions". The external functions supported by High Level Assembler may perform *any* desired action; they are invoked by the SETAF and SETCF statements. We will describe examples of external functions in "External Conditional-Assembly Functions" on page 99.

Internal Conditional-Assembly Functions

There are sixteen internal functions: one for Boolean operations (XOR); eight for arithmetic operations (AND, OR, NOT, XOR, SLA, SLL, SRA, SRL); and seven for character operations (UPPER, LOWER, DOUBLE, INDEX, FIND, BYTE, SIGNED). These functions, like the previously existing Boolean connectives, are a part of the conditional assembly language, not of the "base" language.

With these enhancements, the conditional-assembly language now supports most of the fullword binary operations available in the System/360/370/390 hardware: arithmetic, logic/masking, and shifting.



Internal Arithmetic-Valued Functions

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

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

When used as arithmetic operators, these logical operations act on the fullword binary values of arithmetic operands, in exactly the same way as the corresponding System/360/370/390 instructions N, O, and X. The NOT operator produces the bitwise (or “ones”) complement of its operand, which has the same effect as XORing the operand with a word of all one-bits (−1).

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

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

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

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

The shifting operators for arithmetic operands correspond to the shift instructions provided by the System/360/370/390 hardware: left or right, and arithmetic (signed) or logical (unsigned).

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

These operators may be used in any combination:

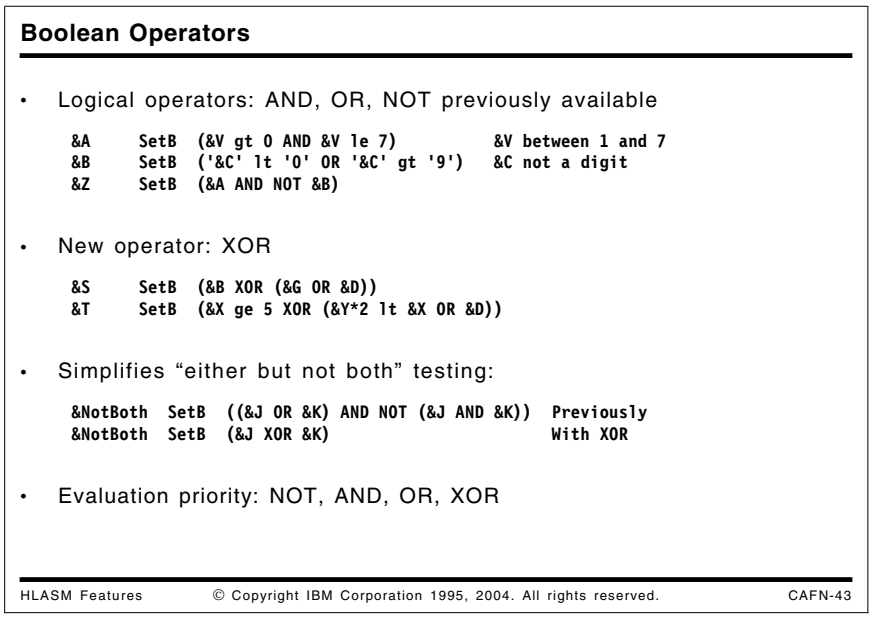
```
&Z   SetA  ((3+(NOT &A) SLL &B))/((&C-1 OR 31)*5)
```

These functions can be used in places where the previously available capabilities of the conditional assembly language led to clumsy constructions. Because the conditional assembly language is interpreted by the assembler, there will not always be significant performance gains in using these new arithmetic operators. However, any simpler expression will almost always be evaluated more rapidly than an equivalent but more complex expression. For example, suppose you must “extract” the value of bit 16 (having numeric weight 2^{15}) from the arithmetic variable &A. Previously, you might have written

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

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

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



Internal Boolean-Valued Functions

The new XOR operator completes the set of Boolean connectives, which previously included the OR, AND, and NOT operators. For example, you can write statements such as

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

XOR can also simplify certain evaluations. Suppose you wish to set the Boolean variable symbol &NotBoth to TRUE if either of &J or &K is TRUE, but not both. Without XOR, you might write

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

but using XOR, the expression is very simple:

```
&NotBoth  SetB  (&J XOR &K)                       With XOR
```

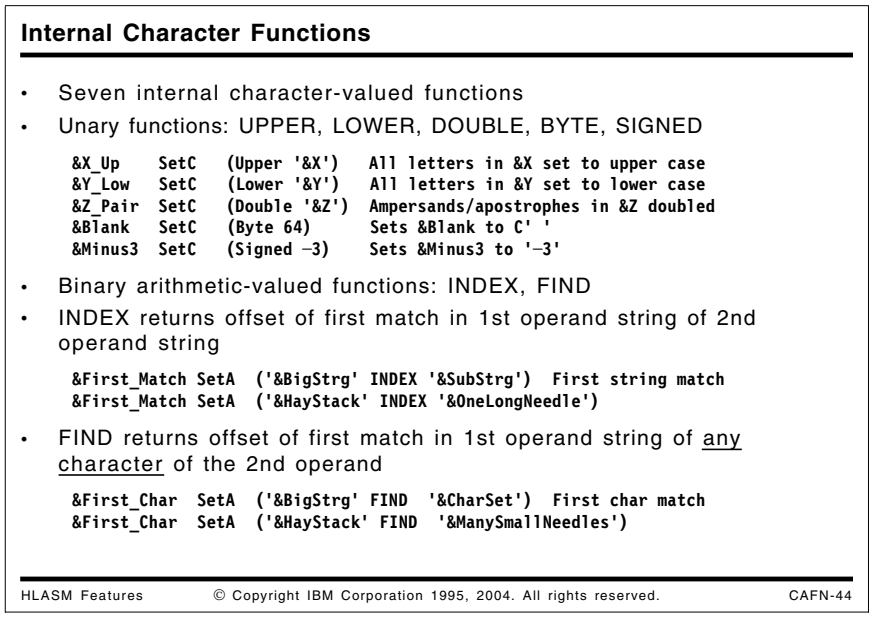
The XOR operator has the lowest priority of all the Boolean operators. Thus, the expression

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

is evaluated as

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

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



Internal Character Functions

High Level Assembler supports seven internal character functions that can greatly simplify handling of character variables and data in conditional assembly expressions. Three of the functions take a single character string argument:

```
&X_Up    SetC   (Upper '&X')   All letters in &X set to upper case
&Y_Low   SetC   (Lower '&Y')   All letters in &Y set to lower case
&Z_Pair  SetC   (Double '&Z')  Ampersands/apostrophes in &Z doubled
```

The UPPER and LOWER functions convert all alphabetic characters in a string to upper case (capital) or lower case letters, respectively. The DOUBLE function scans a string for all occurrences of ampersands and apostrophes, and replaces each such occurrence with pairs of that character; the result may then be substituted safely as the nominal value of a character constant or self-defining term.

```
&S      ARead ,                Read next record into &S
Do's & Dont's
&S      SetC  (Double &S)      Double ampersands and apostrophes
Text    DC    C'&S'            Character constant
```

The remaining two functions, INDEX and FIND, take two arguments. The INDEX function searches its first operand string for a substring that matches the second operand string. If a match is found, the function returns an arithmetic (integer) value giving the character position within the first operand where the match begins; if no match is found, the function returns a zero value. For example:

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

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

Two character-valued functions take a single arithmetic argument: BYTE and SIGNED. The BYTE function creates a string containing a single character whose bit pattern is supplied by the value of the arithmetic argument; the value must lie between 0 and 255. For example:

```
&Blank   SetC  (BYTE 64)         Set &Blank to a space character
```

is equivalent to

```
&Blank   SetC  ' '              Works for representable characters
```

but the BYTE function is more general than SETC in allowing arbitrary bit patterns to be created, rather than just those easily entered as source-statement characters.

The SIGNED function allows you to convert arithmetic expressions to correctly signed character string values. A SETC statement with an arithmetic variable argument creates an unsigned string representing the *magnitude* of the arithmetic variable, whereas the SIGNED function supplies a leading minus sign for negative values.

```
&Val     SetA  -5
&Mag     SetC  '&Val'            &Mag is '5'
&Signed  SetC  (Signed &Val)     &Signed is '-5'
```

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

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

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

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

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

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

The FIND function searches its first operand string for the first occurrence of *any one* character among those in its second operand string. (Unlike INDEX, which requires that every character in the second operand match — in the order given — identical characters in the first operand, FIND only searches for a match of any *single* character.)

The FIND function can greatly simplify string-scanning problems involving searches for one of a set of specific characters. Previously, string scans had to be written to proceed on a character-by-character basis, testing each character in turn for a match. For example, suppose you want to search an “expression string” for the presence of the arithmetic operators +, −, *, and /. Without the FIND function, you might have written a code fragment like this:

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

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

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

Using these techniques, complex string-manipulation problems can be coded much more simply.

External Conditional-Assembly Functions

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

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

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

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

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

HLASM Features CAFN-45

External Conditional-Assembly Functions

High Level Assembler for MVS & VM & VSE supports a powerful capability for invoking externally-defined functions during the assembly. These functions are known as “conditional-assembly functions”, and can perform almost any desired action. They are invoked using the conditional assembly statements SETAF and SETCF, by analogy with the familiar SETA and SETC statements.

The syntax of the statements is similar to that of SETA and SETC: a local or global variable symbol appears in the name field; it will receive the value returned from the function. The operation mnemonic indicates the type of function to be called, and the type of value to be assigned to the “target” variable. The first operand in each case is a character expression (typically a quoted string) giving the name of the function to be called. The remaining operands are optional, and their presence depends on the function: some functions require no

parameters, others may require several. The type of each of these parameters is the same as that of the target variable: arithmetic parameters for SETAF, and character parameters for SETCF.

A compact notational representation of this description is

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

For example, we might invoke the LOG2 and REVERSE functions (to be discussed in detail below) with these two statements:

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

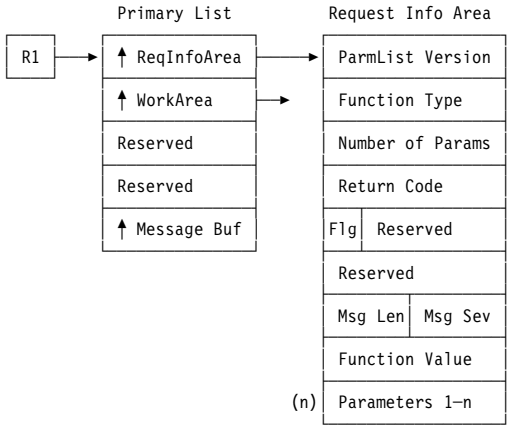
When a function is first invoked, the assembler dynamically loads the module containing the function into working storage, and prepares the necessary control structures for invoking the function. The call to the function uses standard operating system calling conventions; the assembler creates the calling sequence using the parameters and the function name supplied in the SETxF statement.

Following normal parameter-passing conventions, the assembler sets R1 to point to a list of addresses. The first address in this primary list is that of a “Request Information Area”, a list of fullword integer values which describe the type of function (arithmetic or character), the version of the interface, the number of arguments, the return code, and either the returned value and the integer arguments (for SETAF), or the lengths of the respective argument strings (for SETCF). The remaining items in the primary list pointed to by R1 are pointers to a 32-byte work area, and (for SETCF) pointers to the result string and each of the argument strings.

HLASM provides a means whereby an external function can return messages and severity codes; this allows functions to detect and signal error conditions in a way similar to the facility provided by I/O exits.

At the end of the assembly, HLASM will check to see if each called external function wants a final “closing” call so it can free any resources it may have acquired. Finally, the assembler lists for each function the number of SETAF and SETCF calls, the number of messages issued, and the highest severity code returned by the function.

SETAF External Function Interface



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

SETAF External Function Interface

The interface used by High Level Assembler to invoke external arithmetic-valued functions is a standard calling sequence, with an argument list composed of two structures: the layout of the Primary Address List and the Request Information Area is shown in Figure 72.

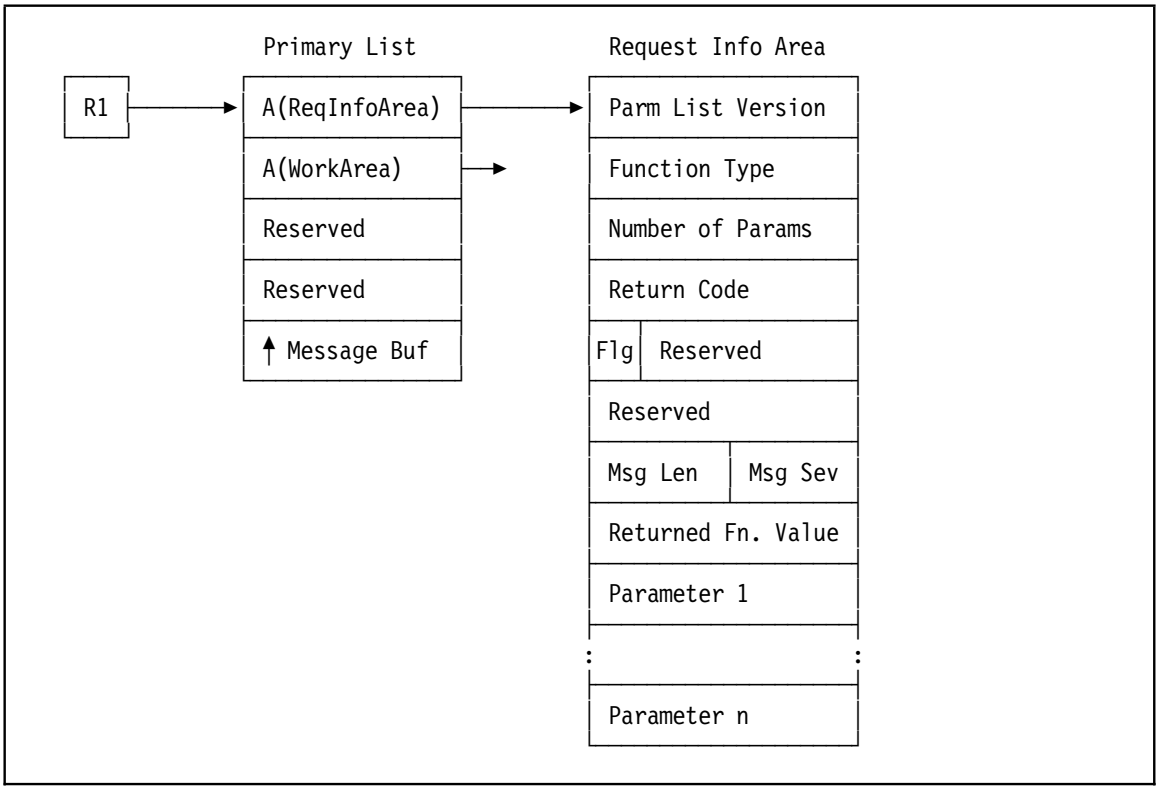
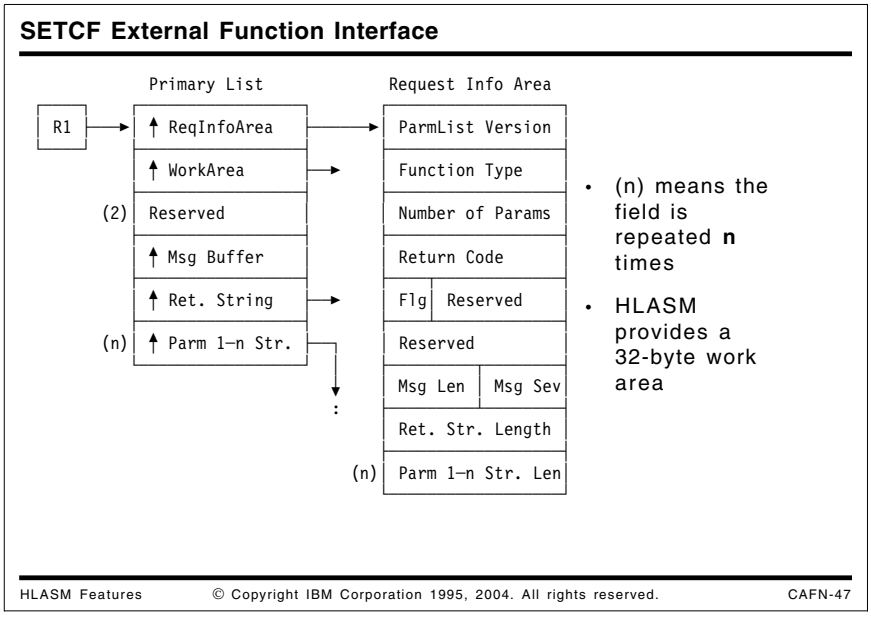


Figure 72. Interface for Arithmetic (SETAF) External Functions



SETCF External Function Interface

The assembler interface for character functions is illustrated in Figure 73, where the layout of the Primary Address List and the Request Information Area are shown.

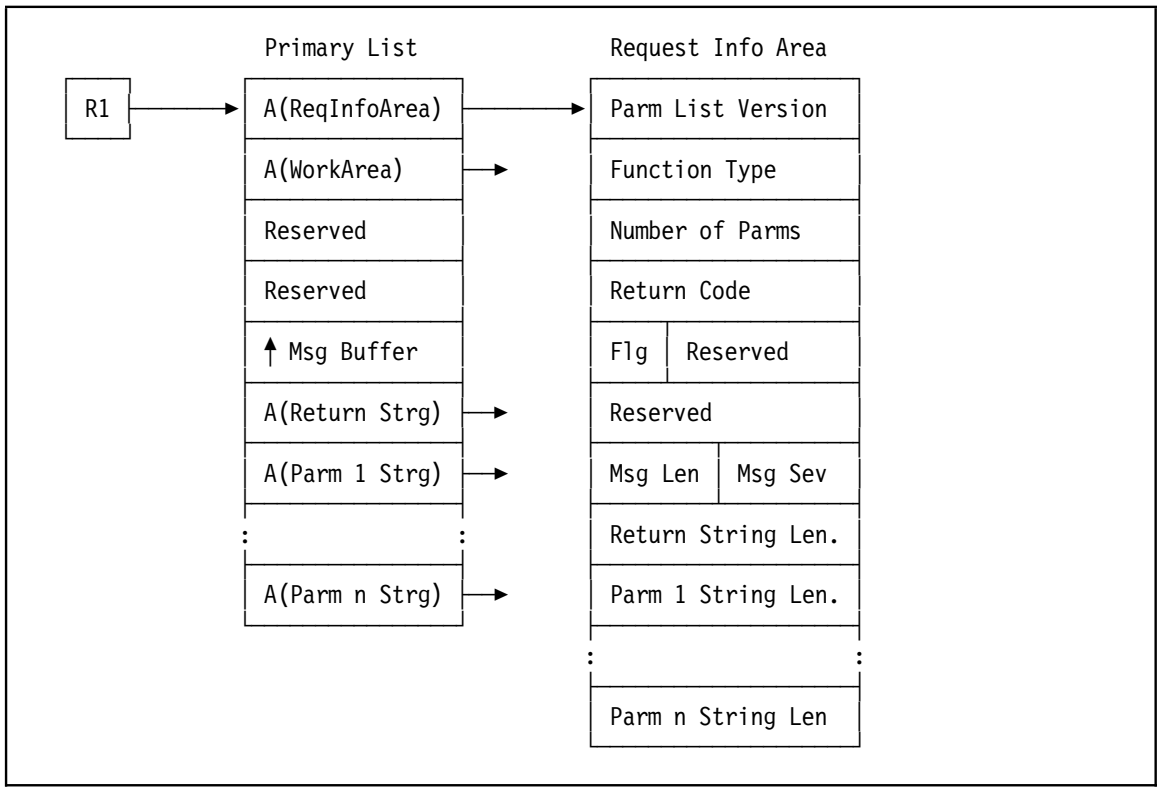
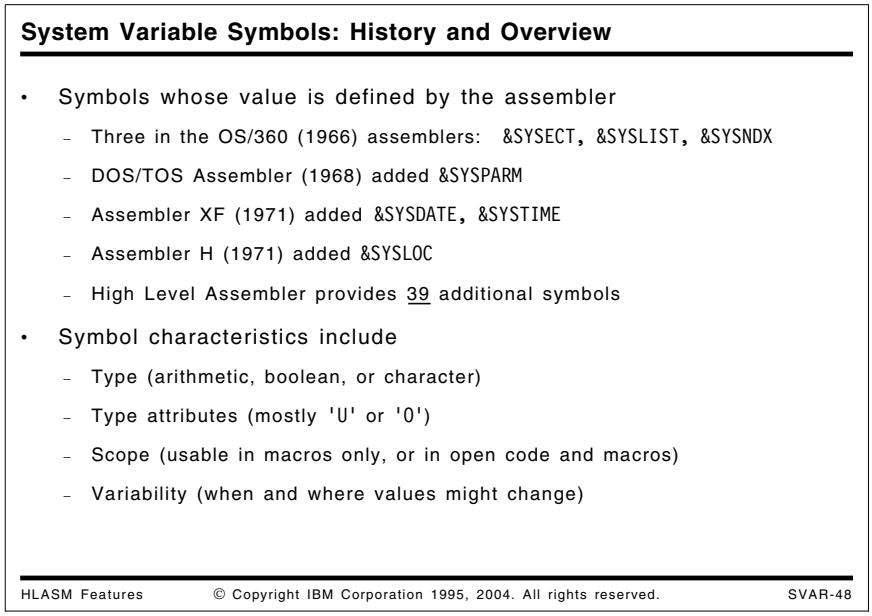


Figure 73. Interface for Character (SETCF) External Functions

System (&SYS) Variable Symbols

System variable symbols are a special class of variable symbols, starting with the characters &SYS. They are “owned” by the assembler: they may not be declared in LCLx or GBLx statements, and may not be used as symbolic parameters. Their values are assigned by the assembler, and never by SETx statements.



High Level Assembler provides 46 system variable symbols that capture useful information about the state of various aspects of the assembly. Figure 74 summarizes their properties.

Figure 74 (Page 1 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	Avail-ability	Type	Type Attr.	Usage Scope	Vari-ability	Content and Use
&SYSADATA_DSN	HLA2	C	U	Local	Fixed	SYSADATA file data set name
&SYSADATA_MEMBER	HLA2	C	U	Local	Fixed	SYSADATA file member name
&SYSADATA_VOLUME	HLA2	C	U	Local	Fixed	SYSADATA file volume identifier
&SYSASM	HLA1	C	U	Global	Fixed	Assembler name
&SYSCLOCK	HLA3	C	U	Local	Constant	Date/time macro was generated
&SYSDATC	HLA1	C,A	N	Global	Fixed	Assembly date, in YYYYMMDD format
&SYSDATE	AsmH	C	U	Global	Fixed	Assembly date in MM/DD/YY format
&SYSECT	AsmH	C	U	Local	Constant	Current control section name
&SYSIN_DSN	HLA1	C	U	Local	Constant	Current primary input data set name
&SYSIN_MEMBER	HLA1	C	U,O	Local	Constant	Current primary input member name
&SYSIN_VOLUME	HLA1	C	U,O	Local	Constant	Current primary input data set name volume identifier
&SYSJOB	HLA1	C	U	Global	Fixed	Assembly job name
&SYSLIB_DSN	HLA1	C	U	Local	Constant	Current library data set name
&SYSLIB_MEMBER	HLA1	C	U,O	Local	Constant	Current library member name
&SYSLIB_VOLUME	HLA1	C	U,O	Local	Constant	Current library data set volume identifier
&SYSLIN_DSN	HLA2	C	U	Local	Fixed	SYSLIN file data set name
&SYSLIN_MEMBER	HLA2	C	U	Local	Fixed	SYSLIN file member name
&SYSLIN_VOLUME	HLA2	C	U	Local	Fixed	SYSLIN file volume identifier
&SYSLIST	AsmH	C	any	Local	Constant	Macro argument list and sublist elements
&SYSLOC	AsmH	C	U	Local	Constant	Current location counter name
&SYSM_HSEV	HLA3	C	N	Global	Variable	Highest MNOTE severity so far in assembly
&SYSM_SEV	HLA3	C	N	Global	Variable	Highest MNOTE severity for most recently called macro
&SYSMAC	HLA3	C	U,O	Local	Constant	Name of current macro and its callers
&SYSNDX	AsmH	C,A	N	Local	Constant	Macro invocation count
&SYSNEST	HLA1	A	N	Local	Constant	Nesting level of the macro call
&SYSOPT_DBCS	HLA1	B	N	Global	Fixed	Setting of DBCS invocation parameter
&SYSOPT_OPTABLE	HLA1	C	U	Global	Fixed	Setting of OPTABLE invocation parameter
&SYSOPT_RENT	HLA1	B	N	Global	Fixed	Setting of RENT invocation parameter

Figure 74 (Page 2 of 2). Properties and Uses of System Variable Symbols

Variable Symbol	Avail-ability	Type	Type Attr.	Usage Scope	Vari-ability	Content and Use
&SYSOPT_XOBJECT	HLA3	B	N	Global	Fixed	Setting of XOBJECT/GOFF invocation parameter
&SYSPARM	AsmH	C	U,O	Global	Fixed	Value provided by SYSPARM invocation parameter
&SYSPRINT_DSN	HLA2	C	U	Local	Fixed	SYSPRINT file data set name
&SYSPRINT_MEMBER	HLA2	C	U	Local	Fixed	SYSPRINT file member name
&SYSPRINT_VOLUME	HLA2	C	U	Local	Fixed	SYSPRINT file volume identifier
&SYSPUNCH_DSN	HLA2	C	U	Local	Fixed	SYSPUNCH file data set name
&SYSPUNCH_MEMBER	HLA2	C	U	Local	Fixed	SYSPUNCH file member name
&SYSPUNCH_VOLUME	HLA2	C	U	Local	Fixed	SYSPUNCH file volume identifier
&SYSSEQF	HLA1	C	U,O	Local	Constant	Sequence field of current open code statement
&SYSSTEP	HLA1	C	U	Global	Fixed	Assembly step name
&SYSSTMT	HLA1	C,A	N	Global	Variable	Number of next statement to be processed
&SYSSTYP	HLA1	C	U,O	Local	Constant	Current control section type
&SYSTEM_ID	HLA1	C	U	Global	Fixed	System on which assembly is done
&SYSTERM_DSN	HLA2	C	U	Local	Fixed	SYSTERM file data set name
&SYSTERM_MEMBER	HLA2	C	U	Local	Fixed	SYSTERM file member name
&SYSTERM_VOLUME	HLA2	C	U	Local	Fixed	SYSTERM file volume identifier
&SYSTIME	AsmH	C	U	Global	Fixed	Assembly start time
&SYSVER	HLA1	C	U	Global	Fixed	Assembler version

System Variable Symbols: Properties

The symbols have a variety of characterizations:

- Availability

  Symbols that were available in Assembler H are designated “AsmH”; High Level Assembler provides a rich set of 39 additional system variable symbols, designated “HLAn” (where “**n**” indicates the release of High Level Assembler in which the symbol first appeared).

- Type

  Most symbols have character values, and are therefore of type C: that is, they would normally be used in SETC statements or in similar contexts. A few, however, have arithmetic values (type A) or boolean values (type B). &SYSDATC and &SYSSTMT are nominally type C, but may also be used as type A.

- Type attributes

  Most system variable symbols have type attribute U (“undefined”) or O (“omitted”, usually indicating a null value); some numeric variables have type N. The exception is &SYSLIST: its type attribute is determined from the designated list item.

- Scope of usage

  Some symbols are usable only within macros (“local” scope), while others are usable both within macros and in open code (“global” scope).

- Variability

  Some symbols have values that do not change as the assembly progresses. Normally, such values are established at the beginning of an assembly. These values are denoted “Fixed”. Note that all have Global scope.

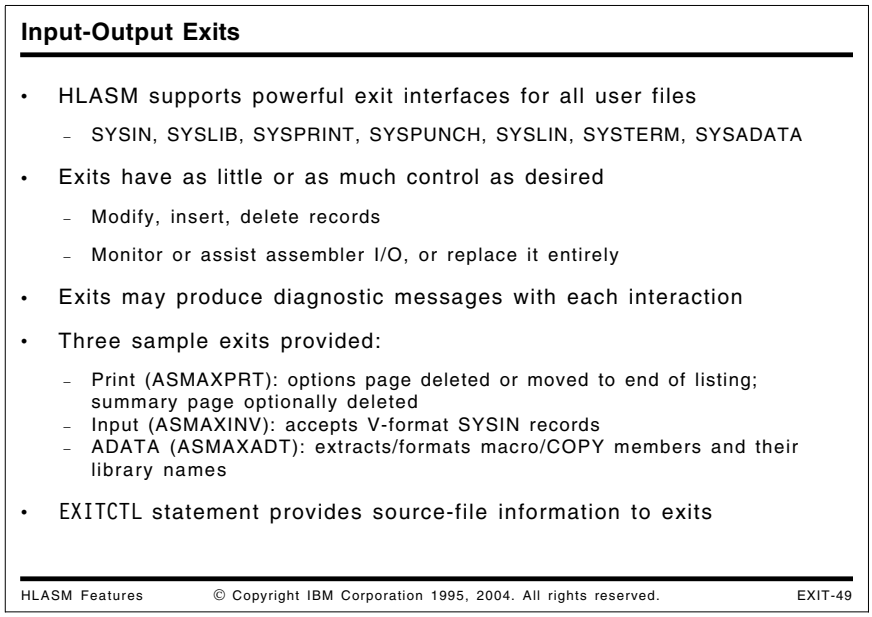
  Other symbols have values that may change during the assembly. These values might be established at the beginning of an assembly or at some point subsequent to the beginning, and may change depending on conditions either internal or external to the assembly process.

  - Variables whose values are established at the beginning of a macro expansion, and for which the values remain unchanged throughout the expansion, are designated “Constant”, even though they may have different values in a later expansion of the same macro, or within “inner macros” invoked by another macro. Note that all have local scope.

  - Variables whose values may change within a single macro expansion are designated “Variable”. Currently, this designation applies only to &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

These symbols have many uses: helping to control conditional assemblies, capturing environmental data for inclusion in the generated object code, providing program debugging data, and more.

Input-Output Exits



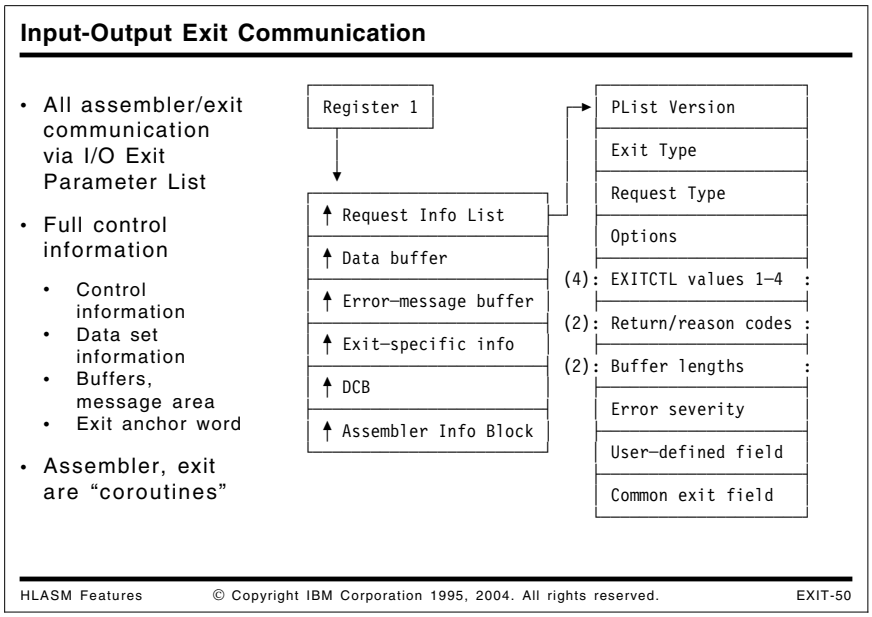
High Level Assembler supports a very powerful and flexible I/O exit interface. An exit routine may modify, add, or delete records as they pass to and from the assembler; it may also share I/O activity with the assembler, or replace that activity entirely by its own. All the exits use the same interface, and a single exit routine may be used to support more than one type of exit. (An example is illustrated in the *High Level Assembler Programmer's Guide*.)

Three sample exits are provided with HLASM (except for the VSE Edition):

- a print exit ASMAXPRT: it will optionally move the list of assembly options from the head to the end of the listing;
- an input exit ASMAXINV: it accepts variable-format (V-format) input records, and converts them to the fixed format required by the assembler;
- an ADATA exit ASMAXADT: it extracts information from the SYSADATA file and produces fixed-format records for each macro or COPY file, indicating the library from which it was read.

Each of these sample exits provides a useful function while illustrating typical exit coding techniques.

In this chapter we will illustrate a sample object-file exit for inserting Linkage Editor or Binder control statements into the object stream, at “Example: A SYSLIN, SYSPUNCH Object-File Exit” on page 112.



Communication and Work Areas

The interface between High Level Assembler and its I/O exits establishes a “coroutine” interaction: both the assembler and the exit routine must cooperate, with neither being fully in control of the other. All interactions take place through the I/O exit parameter list illustrated in Figure 75 on page 109.

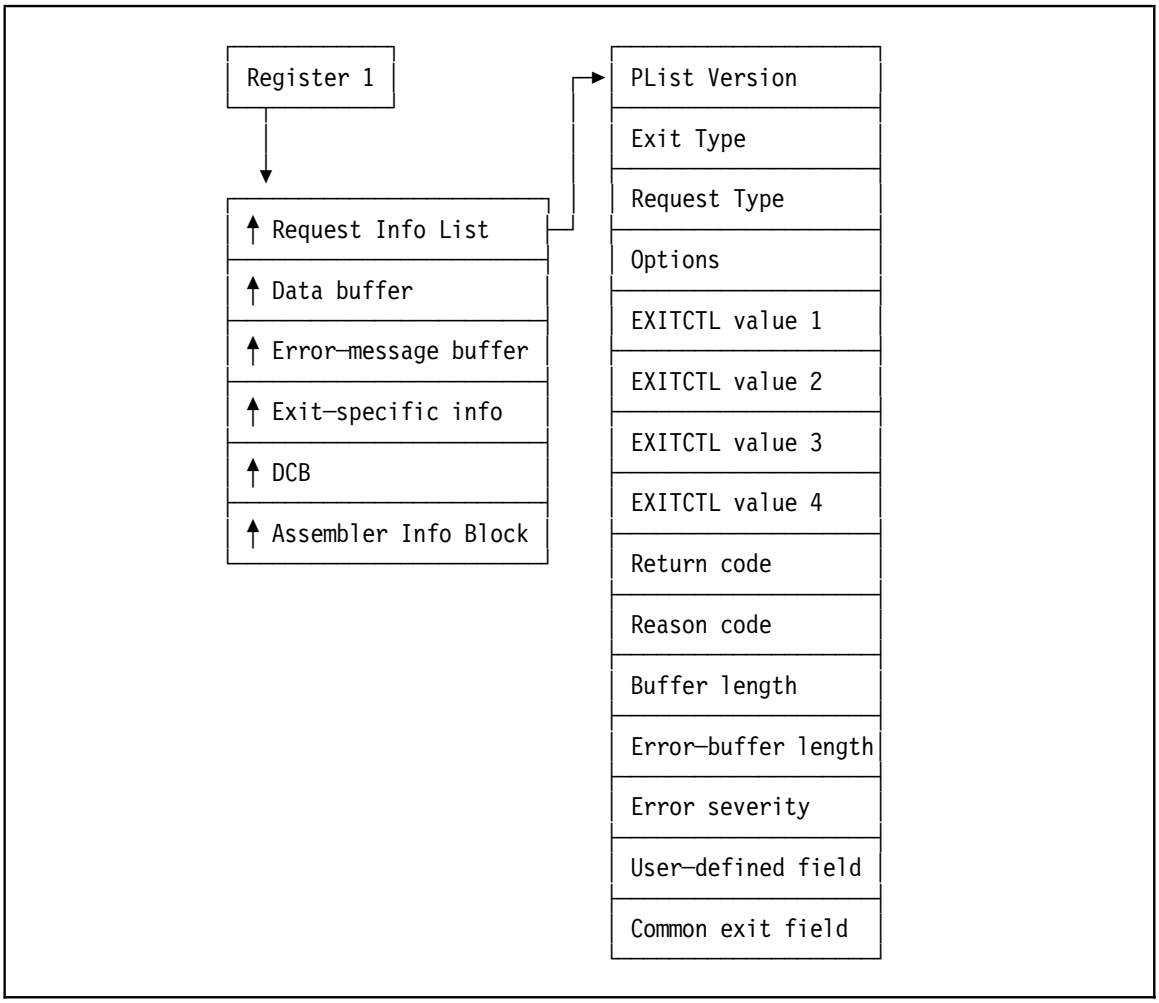


Figure 75. I/O Exit Parameter List

The I/O exit interface uses standard OS linkage conventions, and the parameter list follows standard OS parameter-passing conventions. (In fact, this interface was designed to allow exit routines to be written in most high-level languages.) There are five main elements of this list:

1. The first parameter in the list is a pointer to the Request Information List, an array of fullword integers:
   - The first word indicates the version of the parameter list.
   - The second word indicates the exit type; its value tells the exit routine what file it is expected to work with.
   - The third word indicates the request type, such as opening or closing the exit, processing a record, or performing a read or write.
   - The fourth word indicates optional additional information about the exit's activity, such as changes to data set names or types of output records.
   - The fifth through eighth words contain the EXITCTL values; these are described in “The EXITCTL Statement” on page 111.
   - The ninth and tenth words contain the return and reason codes. These are used by the exit routine to indicate subsequent processing actions to be taken by the assembler.
   - The eleventh word contains the length of the record in the data buffer.

- The twelfth word contains the length of an error message placed in the error message buffer by the exit. The assembler checks this field after each return from an exit, and a nonzero length indicates the presence of a message.
- The thirteenth word is the error severity to be associated with the message.
- The fourteenth word is initialized by the assembler to zero before the initial call to the exit, and is unmodified thereafter. It may be used for any purpose by the exit; a typical use would be to hold the “anchor address” of additional working storage obtained by the exit.
- The fifteenth word is shared by all I/O exits, and provides a common anchor for data to be shared among exits.

2. The second parameter is the address of the data buffer used for passing records between the exit and the assembler.
3. The third parameter is the address of an error-message buffer, where the exit routine can place messages to be displayed in the assembly's source and object code listing. (The length of the message is placed in the twelfth word of the Request Information List.)
4. The fourth parameter is the address of an exit-specific information area; its contents depend on the type of exit being invoked. Typical contents include the data set and member names of the current file.
5. The fifth parameter is the address of the assembler's Data Control Block (DCB) for the file appropriate to the type of exit.
6. The sixth and last parameter is the address of the “Assembler Static Information Block”, which provides information about the assembler itself:
   - an 8-byte field of EBCDIC characters contains the version, release, and modification level of HLASM in V.R.M format. For example, if HLASM Release 4.0 is used, the field will contain

     ```
     1.4.0
     ```

   - an 8-byte field of EBCDIC characters contains the current HLASM PTF level. For example, this field might contain

     ```
     UQ32700
     ```

   - a 16-byte field of EBCDIC characters contains the name and version and release level of the operating system under which HLASM is executing. For example, this field might contain

     ```
     MVS/ESA SP 5.1.0
     ```

The actions to be taken by the exit and the assembler are determined by the values of the request type (when the assembler calls the exit) and the return and reason code (when the exit returns to the assembler).

Mapping the Communication and Work Areas

The I/O exit parameter lists are mapped by DSECTs generated by the ASMAXITP macro, which is supplied by High Level Assembler with the macros used for installing and customizing the assembler.

The EXITCTL Statement

High Level Assembler for MVS & VM & VSE provides the EXITCTL statement to allow greater control (by the source program being assembled) over the actions of an I/O exit. It is written in the form

```
EXITCTL  exit-type,value-1,value-2,value-3,value-4
```

The four operands of the statement are converted into integer values, and are passed to the exit in four contiguous words in the Request Information List, as illustrated in Figure 75 on page 109.

The EXITCTL values are set at the time records are being read or written by the assembler; this means that input exits are active during the earlier phases of the assembly, and the output exits are active during the later phases of the assembly. (Thus, you should not expect to create complex interactions among the exits that depend on a particular sequence of statements!)

The EXITCTL instruction passes four fullword integer values to the designated exit routine. It may sometimes be useful to pass character values to the exit. To illustrate, suppose you are writing a LIBRARY exit routine which is capable of searching multiple sets of libraries (each set having a different DDname), and you wish to specify the “new” library DDname in the source program with an ASYSLIB statement. For example,

```
ASYSLIB  ALTDDN2
```

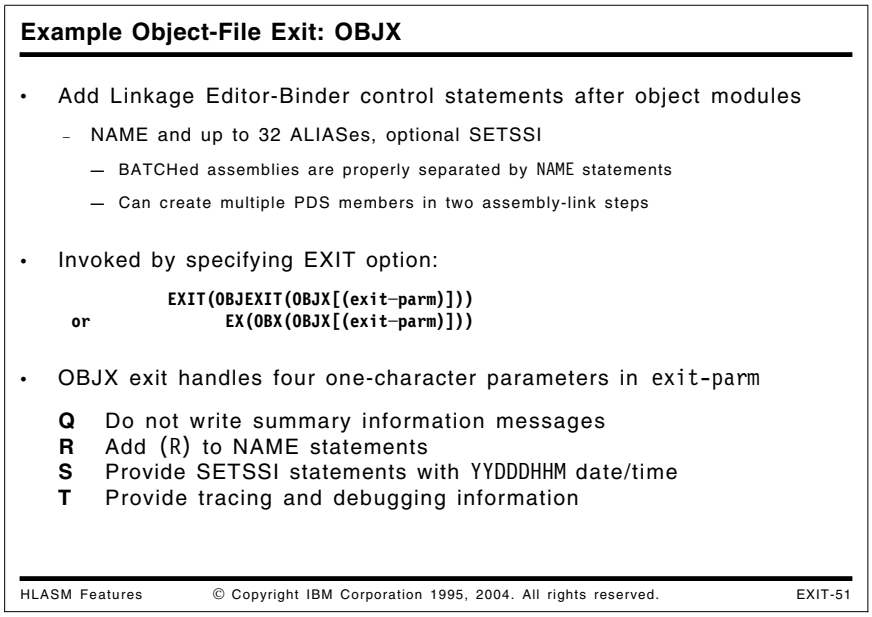
would instruct the exit to switch to the libraries defined by the ALTDDN2 DD statement.

We could define an ASYSLIB macro to convey the new DDname in the third and fourth EXITCTL values as shown in Figure 76. Because EXITCTL accepts only decimal terms in its operands the macro must convert the 8-character DDname to two integers having the same bit representation.

```
         Macro
&L       ASYSLIB  &D,&Z
         AIf      (K'&Z eq 0).Operand
         MNote    8,'Only one operand allowed.'
         MExit
.Operand AIf      (K'&D eq 0).Revert   Null operand => SYSLIB
         AIf      (K'&D le 8).Proceed
         MNote    8,'DDname ''&D'' is too long.'
         MExit
.Revert  ANop
&Name    SetC     'SYSLIB  '
         AGo      .DoName
.Proceed ANop
&Name    SetC     '&D'.'       '       Pad with blanks
&Name    SetC     '&Name'(1,8)         Take first 8 characters
.DoName  ANop
&N(1)    SetC     '&Name'(1,4),'&Name'(5,4)
&N(1)    SetC     'C''&N(1)''','C''&N(2)'''  Make self-defining terms
&V(1)    SetA     &N(1),&N(2)          Arithmetic values
&N(1)    SetC     (Signed &V(1))           Numerics, in character form
&N(2)    SetC     (Signed &V(2))           Numerics, in character form
.A3      ExitCtl  LIBRARY,0,0,&N(1),&N(2)   Data for LIBRARY Exit
         MNote    *,'Switching to DDname ''&Name.'''
         MEnd
```

Figure 76. Passing character data to I/O exits: ASYSLIB macro

Example: A SYSLIN, SYSPUNCH Object-File Exit



The SYSLIN and SYSPUNCH files produce the object-module records assembled from the source program, as well as records produced by PUNCH and REPRO statements. You may monitor these records, adding to them, deleting them, or modifying them, by requesting that High Level Assembler invoke an exit routine.

Creating Linkage Editor Control Statements

When the assembler is used to create one or more output modules for subsequent processing by a Linkage Editor, it is typically necessary to place Linkage Editor control statements following each module — for example, to NAME the member of the load module library, or to assign additional entry points as ALIASes of the member. When several source modules are assembled with the BATCH option, it can be difficult to provide automated procedures to separate the output modules and insert the desired control statements.

There are many different techniques that can be used to write an exit; most exits would emphasize simplicity and small size. The example given here — the object-file exit — is more elaborate than would normally be necessary: it checks interface parameters carefully, instead of assuming that the assembler has set them properly. This is done to help you understand the operation of the assembler interface with its exit routines.

Description of HLASM Object Exit OBJX

This exit, named OBJX, adds Linkage Editor ALIAS and NAME control statements to the object output stream from the High Level Assembler. The exit is invoked by specifying this invocation parameter to HLASM:

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

The allowed values of exit-parm are the lower- or upper-case characters Q, R, S, and T, or any combination of single occurrences of them.

These mean respectively:

Q Do not write summary information messages

R Add the characters (R) to NAME statements

S Provide SETSSI statements

T Provide tracing information

The exit routine will monitor the object stream and extract CSECT names and ENTRY names, up to a total number defined by the value of the variable symbol &MaxAlias (set to 32 in this sample program). When the END record is recognized, it will be followed by up to (&MaxAlias.−1) ALIAS statements, a SETSSI statement (if requested), and a NAME statement for the first non-blank CSECT or entry point name (followed by "(R)" if requested in the exit-parm).

The SETSSI information will be in the form YYDDDHHM, and is initialized at the first invocation of the exit. If more than one object module is processed, all SETSSI statements will provide identical information.

Possible entry names in excess of the number defined by &MaxAlias will be ignored, and (if the Q exit-parm has not been specified) an appropriate message will be printed.

The exit routine will provide a summary of its actions by writing (zero-severity) information messages unless the "Q" exit-parm is specified.

If no object-module records are written (for example, the assembly consists only of PUNCH and REPRO records), then the exit will take no action.

Error Messages

All errors will terminate the assembly. The following error messages will be issued independent of the setting of the "Q" flag:

- Exit not coded at same level (1) as Assembler

  The exit uses version 1 of the High Level Assembler exit definition, but the assembler invoking this exit uses a different version.

- Exit type requested is unrecognizable

  The type of the exit requested is not one of the recognized types (PRINT, LIBRARY, PUNCH, OBJECT, etc.).

- Exit called for other than PUNCH or OBJECT

  The exit was invoked with a valid type, but that type was not one that this exit can handle.

- Exit not initialized, and not entered for OPEN

  The exit has not yet been initialized, but was not entered with an OPEN request. There may be a failure in communication between the assembler and this exit.

- Exit initialized, but was entered for OPEN

  The exit has been initialized, but was unexpectedly entered with an OPEN request. There may be a failure in communication between the assembler and this exit.

- Invalid request-list options value

  This exit should never have any additional options value supplied in the request list for an OPEN exit, but a non-zero value was detected.

- Invalid parameter-string length

  The length of the parm string was not between 0 and 4.

- Invalid character in parameter string

  A character was found in the parameter string that is not one of the permitted values.

- Duplicated valid character in parameter string

A valid parameter character appeared in the parameter string more than once.

- `Invalid action or operation type requested`

  An action was requested that is inconsistent with the type of action the exit was expecting to take.

- `Expecting input record, zero buffer length`

  The exit was expecting an input record to scan, but the buffer length was found to be zero.

Information Messages

If the "Q" option has not been requested, the exit will provide one or more of the following messages at the end of each assembly:

- `nnn Entry names were processed.`
  `nnn Entry names, mmm were ignored.`

  At the end of each object module containing usable control section or entry point names, this message summarizes the number of names recognized and (if there are too many to handle) the number that were not processed.

- `Object Module contained no usable SD or LD names`

  The object module contained no names identifiable as possible module names or aliases. This may be a valid condition; for example, an assembly with only PUNCH statements would contain no names.

Coding the OBJECT Exit OBJX

The code for the object exit will be given in segments that correspond approximately to functional units in the program. Each segment is followed by a description of its function.

```
          Title 'High Level Assembler PUNCH/OBJECT Exit for Linkage Edit*
                or Control Statements'

*         Define the maximum number of ALIAS names to be supported
            LCLA  &MaxAlias

*         Define the environment in which the exit will run
            LCLB  &EnvMVS,&EnvCMS,&EnvVSE
&EnvMVS     SETB 1
&EnvCMS     SETB 1                  Assume CMS can use OS emulation
&EnvVSE     SETB (NOT &EnvMVS)

*         Define the external names of the exit routine
            LCLC  &Csect
&Csect      SETC 'OBJX'

&MaxAlias SETA 32
&Csect    SETC 'OBJX'
```

Figure 77. Object exit OBJX: variable symbol definitions

In Figure 77, five local variable symbols are defined:

- &MaxAlias defines the size of the table that will be used to hold the usable external names scanned from each object module.
- &EnvMVS, &EnvCMS, and &EnvVSE define the environment in which the exit is intended to operate. In this sample program, we assume an MVS environment (or CMS, with OS

emulation). The idea is that future enhancements can use these variables in conditional logic to determine what system interfaces to use.

- &Csect defines the name of the control section containing the exit program, and will define its entry point. By defining it with a variable symbol, the name of the exit routine is easy to change if desired.

```
*        Method of operation:

*        (1) Initial entry
*            (a) validate parameters
*            (b) get and initialize working storage, save pointer
*                in AXPUSER User Field of Request Information List
*            (c) check for and scan parms, set flags
*            (d) initialize batch fields

*        (2) Process entry
*            (a) if not outputting retained names, then
*                1. scan record for ESD or END
*                2. if neither, return to output the record
*                3. if ESD, scan off SD and LD names, keep count
*                4. if END, indicate 'outputting retained names'
*            (b) if outputting retained names, then
*                1. if names count = 0, re-init batch, and exit
*                2. if names count > 1, output ALIAS statements
*                3. if names count = 1, then output SETSSI if wanted;
*                   output NAME record, re-init batch

*        (3) Close entry
*            (a) if outputting, indicate faulty object deck
*            (b) if requested, put out summary message
*            (c) free storage and exit
```

Figure 78. Object exit OBJX: description of method of operation

In Figure 78, the method of operation used by the exit is characterized in simple terms. As we will see, it is necessary for the exit to save information about its state across entries from the assembler.

```
&Csect   Title 'Object-Editing Exit ''&Csect.'''
&Csect   RSect                     Program is re-entrant
&Csect   AMode 24
&Csect   RMode 24

*        Register Equates (R6,R7 not used)

R0       Equ   0
R1       Equ   1
R2       Equ   2
R3       Equ   3
R4       Equ   4
R5       Equ   5
R8       Equ   8                   Work Area Pointer
R9       Equ   9                   Request Information List Pointer
R10      Equ   10                  Buffer Pointer
R11      Equ   11                  Error Buffer Pointer
R12      Equ   12                  Program Base
R13      Equ   13
R14      Equ   14
R15      Equ   15
```

Figure 79. Object exit OBJX: CSECT definition and register EQUates

In Figure 79, the CSECT definition is provided; note that because this exit is re-entrant, the RSect statement is used to request that High Level Assembler check for obvious violations of re-enterability. Various register equates used in the program are defined. (Note that no definitions are provided for registers 6 and 7, as they are not used in the program!)

```
*        Displacements

D0       Equ   0
D1       Equ   1
D2       Equ   2
D4       Equ   4
D5       Equ   5
D8       Equ   8

*        Lengths

L1       Equ   1
L2       Equ   2
L3       Equ   3
L5       Equ   5
L8       Equ   8
L80      Equ   80
L255     Equ   255

*        Shift Counts

S3       Equ   3
         CEject 10                 1

*        Other Equates

No_Reason      Equ   0            Null reason code
Max_Parm_Chars Equ   4            Limit on no. of valid parm chars
MSGSEVC        Equ   12           Severity code for errors
Obj_Ind        Equ   X'02'        Object-record indicator
```

Figure 80. Object exit OBJX: other useful EQUates

In Figure 80, various other EQUates used in the program are defined. One aspect of the programming style used is that absolute (self-defining) terms are avoided wherever their use might limit one's ability to maintain or modify the exit.

Note (at 1) the use of the CEJECT statement to keep the following group of lines together on the listing. Other CEJECT statements were used in the program, but they are omitted elsewhere in this example to help save space.

```
*********************************************************************
*        Entry point for all invocations                            *
*********************************************************************

*        Save caller's registers, establish program base

         Using *,R15
         Save  (14,12),,&Csect.-&SysDatC.-&SysTime  Save registers
         LR    R12,R15             R12 will be program base
         Drop  R15
         Using &Csect.,R12

*        Validate entry type and interface version

         Using AXPXITP,R1          R1 points to primary parm list
         LM    R9,R11,AXPRIP       Addresses of first three items
         Using AXPRIL,R9           R9  --> Request Information List
*        Using Buffer,R10          R10 --> Working Buffer
         Using Err_Buff,R11        R11 --> Error Buffer
         Drop  R1
         SR    R8,R8               Clear work area anchor register
         CLC   AXPLVER,=A(AXPVER1) Check version of exit list
         BL    Bad_Version         Can't continue, version mismatch
         CLC   AXPLVER,=A(AXPVER3) Check version of exit list
         BH    Bad_Version         Can't continue, version mismatch
         L     R1,AXPTYPE          Check exit type
         LTR   R1,R1               Verify value non-negative
         BNP   Bad_Exit_Type       Can't continue, illegal exit type
         C     R1,=A(AXPTAD)       Verify value not too large
         BH    Bad_Exit_Type       Can't continue, illegal exit type
         BNL   Wrong_Exit_Type     Can't continue, wrong exit type
         C     R1,=A(AXPTPUN)      Verify value not too small
         BL    Wrong_Exit_Type     Can't continue, wrong exit type
```

Figure 81. Object exit OBJX: initial entry and interface validation

In Figure 81, control is received from the assembler. After certain registers are initialized, the various interface parameters are validated to ensure that this exit routine has been invoked as an object-file exit. The commented USING statement for R10 will not be needed until later, but is included here to help document base register assignments.

```
*        Have been called as PUNCH or OBJECT exit

         L     R8,AXPUSER          Check User Field for work area
         LTR   R8,R8               Check if anchor is present
         BNE   Started             Branch if already initialized, to
*                                  check which function is desired.

*        Not initialized, validate that entry is for OPEN

         CLC   AXPRTYP,=A(AXPROPN) Should be OPEN request
         BE    Open_Request        Branch to process OPEN
         B     Bad_OPEN_Request1   Branch if not an OPEN request
```

Figure 82. Object exit OBJX: Checking for initial or subsequent entry

In Figure 82, we check whether this is the initial or a subsequent entry to the exit routine. The determination is made by testing the contents of the AXPUSER field of the communication work area: if it is zero, this is an initial entry; if not, that field contains the pointer to the exit's work area as it was created by the OPEN processing (described in Figure 83 on page 119 below).

```
*********************************************************************
*        OPEN Request                                               *
*********************************************************************

*        Obtain and initialize working storage

Open_Request  DS  0H
         L     R0,=A(Work_Size)    Size of work area in R0
         LR    R3,R0               Size of work area in R3 for zeroing
         GETMAIN R,LV=(0)          Obtain the storage
         ST    R1,AXPUSER      1   Save work area anchor
         LR    R8,R1               Work Area base register
         Using WorkArea,R8
         LR    R2,R1               Work Area base for zeroing
         SR    R4,R4               Second operand addr for zeroing
         LR    R5,R4               Second operand length and pad byte
         MVCL  R2,R4               Clear work area to zeros

*        Chain save areas

         ST    R13,Save+D4         Save back pointer to caller
         LA    R0,Save             Point to local save area
         ST    R0,D8(,R13)         Save forward pointer for caller
         LR    R13,R0              Establish local save area pointer
```

Figure 83. Object exit OBJX: OPEN processing: obtain and initialize working storage

In Figure 83, the necessary amount of working storage is requested from the operating system, after which it is initialized to zeros and the exit's save area is chained to that of the assembler. Note (at 1) that the AXPUSER field in the communication area is set to contain the (non-zero!) pointer to the exit's work area; this pointer will be used on subsequent entries, both as the base address for the work area, and to distinguish initial from subsequent entries.

```
*        Check for presence of input parms, validate them if present

         L     R0,AXPBUFL          Pick up buffer length indicator
         LTR   R0,R0               Check sign and value
         BZ    No_Parms            Branch if zero, no parms present
         BM    Bad_Parm_Str        Should not be negative
         LA    R15,Max_Parm_Chars  Set max allowed number of chars
         CR    R0,R15              Compare input length to max
         BH    Bad_Parm_Str        Branch if high, string is too long

*        Prepare to check parm string characters

         LR    R1,R10              Pick up buffer address
         Using Buffer,R1           R1 --> Working Buffer
         LR    R2,R0               Copy character count
         BCTR  R2,0                Decrement by one for Execute
         EX    R2,Upper_Parms      OR blanks to make parms upper case
```

Figure 84. Object exit OBJX: initial checks for exit-parm information

In Figure 84, the availability of exit-parms is tested, and if they are present, the characters are converted to upper case to simplify scanning.

```
Parm_Loop  DS  0H
         CLI   Buffer,C'Q'         Check for 'Q' character: Quiet
         BE    Parm_Has_Q          Branch if present
         CLI   Buffer,C'R'         Check for 'R' character: (R)
         BE    Parm_Has_R          Branch if present
         CLI   Buffer,C'S'         Check for 'S' character: SETSSI
         BE    Parm_Has_S          Branch if present
         CLI   Buffer,C'T'         Check for 'T' character: Trace
         BE    Parm_Has_T          Branch if present
         B     Bad_Parm_Char       Error otherwise

Upper_Parms DS  0H                 Executed instruction
         OC    Buffer(*-*),P_Blanks  Force letters to upper case

         Drop  R1       1          No further buffer addressing
```

Figure 85. Object exit OBJX: scan exit-parm characters

In Figure 85, the characters in the exit-parm string are checked against the four valid values. If one is present, a branch is taken to the appropriate processing code in Figure 86 on page 121; if not, an error condition will be indicated. Note (at **1**) that code following this segment no longer requires addressability to the input buffer.

```
Parm_Has_Q DS  OH
         TM    Parm_Q,L'Parm_Q  1  Check if already specified
         BO    Parm_Dup_Char       Branch if duplicated parm char
         OI    Parm_Q,L'Parm_Q  2  Set indicator
         B     Parm_Loop_End       Step to next character

Parm_Has_R DS  OH
         TM    Parm_R,L'Parm_R     Check if already specified
         BO    Parm_Dup_Char       Branch if duplicated parm char
         OI    Parm_R,L'Parm_R     Set indicator
         B     Parm_Loop_End       Step to next character

Parm_Has_S DS  OH
         TM    Parm_S,L'Parm_S     Check if already specified
         BO    Parm_Dup_Char       Branch if duplicated parm char
         OI    Parm_S,L'Parm_S     Set indicator
         B     Parm_Loop_End       Step to next character

Parm_Has_T DS  OH
         TM    Parm_T,L'Parm_T     Check if already specified
         BO    Parm_Dup_Char       Branch if duplicated parm char
         OI    Parm_T,L'Parm_T     Set indicator
         B     Parm_Loop_End       Step to next character
```

Figure 86. Object exit OBJX: processing each exit-parm option

In Figure 86, two things are done for each valid exit-parm character: first, a check is made to see if that character has already been encountered (if so, a message will be issued unless the “Q” option suppresses it); otherwise, the appropriate flag is set to indicate the presence of that option.

The addressing technique used for referring to the bit flags (at 1 or 2, for example) helps to avoid situations where a correct bit definition is used to test a bit in a different byte that doesn't actually contain the desired flag. The definition of the flag itself will be discussed following Figure 109 on page 133.

```
Parm_Dup_Char  DS  OH
         OI    Dup_Char,L'Dup_Char Set duplicated-character flag

Parm_Loop_End  DS OH
         AL    R1,F1               Increment buffer pointer
         BCT   R0,Parm_Loop        And scan again if needed

*        Done with parm scan, see if SSI information is needed

         TM    Parm_S,L'Parm_S     Check request for SSI parm
         BNO   Open_Done           Branch if not, OPEN completed
```

Figure 87. Object exit OBJX: end of exit-parm scan

In Figure 87, duplicate exit-parm characters are flagged, the scan loop is re-executed if necessary, and then a test is made to see if SETSSI-statement information must be prepared.

```
         TIME  DEC                Get time/date info from system

         ST    R1,FTemp           Store '00yydddF' date temporarily
         UNPK  SSI(L5),FTemp+D1(L3) Unpack 'yydddF' to SSI as 'YYDDD'
         ST    R0,FTemp           Store 'hhmmssth' time temporarily
         UNPK  DTemp(L3),FTemp(L2) Unpack 'hhmm' to Dtemp as 'HHmm'
         OI    DTemp+D2,C'0'      Set zone on high-order minute
         MVC   SSI+D5(L3),DTemp   Move 'HHM' to end of SSI data
```

Figure 88. Object exit OBJX: initializing SETSSI information

In Figure 88, the date and time are requested from the operating system, and the returned data is converted into the format that will be used on SETSSI statements.

```
No_Parms  DS   0H
Open_Done DS   0H

*        Indicate assembler opens object file, and proceed normally

         TM    Dup_Char,L'Dup_Char See if duplicate-char flag set
         BZ    Open_Done_1        Branch if not
         MVC   Err_Msg(L'Dup_Prm_Ch_Msg),Dup_Prm_Ch_Msg  Move message
         MVC   AXPERRL,=A(L'Dup_Prm_Ch_Msg)             Set length
         MVC   AXPSEVC,=A(AXPSEV0) Set severity

Open_Done_1 DS 0H
         MVC   AXPRETC,=A(AXPAOPN) Assembler to open object/punch file
         MVC   AXPREAC,=A(No_Reason) No reason code info

         L     R13,Save+D4        Retrieve caller's save area pointer
         RETURN (14,12)           Return to assembler
```

Figure 89. Object exit OBJX: completion of OPEN processing

In Figure 89, OPEN processing is completed by checking for (and, if needed, issuing) a message about duplicated exit-parm characters, and then setting the return and reason codes to tell the assembler that it should open the object and/or punch files normally. Control is then returned to the assembler to continue processing the program.

```
*********************************************************************
*        Exit has been opened (we believe). Check type of action.   *
*********************************************************************
Started  DS    0H

*        Chain save areas

         ST    R13,Save+D4         Save back pointer to caller
         LA    R0,Save             Point to local save area
         ST    R0,D8(,R13)         Save forward pointer for caller
         LR    R13,R0              Establish local save area pointer

*        Have been initialized, verify type of entry request

         CLC   AXPRTYP,=A(AXPROPN) Should not be OPEN request
         BE    Bad_OPEN_Request2   Branch if an OPEN request
         CLC   AXPOPTS,=A(AXPONA)  Should be no options present
         BNE   Bad_Req_Opts        Branch if options are present
         CLC   AXPRTYP,=A(AXPRCLS) Check if CLOSE request
         BE    Close_Request       Branch to do the CLOSE
         CLC   AXPRTYP,=A(AXPRPRO) Check if PROCESS request
         BE    Process_Request     Branch if a PROCESS request
         B     Bad_Type_Request    Branch if not a PROCESS request
```

Figure 90. Object exit OBJX: processing of subsequent (non-initial) entries

In Figure 90, control will have been passed to the label Started if the exit has already been initialized. (See Figure 82 on page 118 for the test of the AXPUSER field.) The save area of the exit is chained to the assembler's, and then the type of requested operation is checked to see if it is a “process” (intermediate) or a “close” (final) request.

```
*********************************************************************
*        PROCESS Request                                            *
*********************************************************************
Process_Request  DS  0H
         TM    Dumping,L'Dumping   See if outputting names now
         BO    Do_Dumping

         CLC   AXPBUFL,=A(L80)     Check to see if record is in buffer
         BNE   Phase_Error_1       Error if not, we're confused

         Using ESD_Rec,R10         Base descriptive Dsect on buffer
         CLI   ESD_Tag,Obj_Ind     Check for actual object record
         BNE   Return_Rec          May have PUNCHed 'ESD' or 'END'
         CLC   ESD_ESD,=C'ESD'     See if it's an ESD record
         BNE   Check_END           If not ESD, check for END
```

Figure 91. Object exit OBJX: request to process an object record

In Figure 91, the exit has been presented with an object-file record. First, a check is made to see if the end of the object module has been reached; if so, we begin (or continue) the process of “dumping” out the collected names. Otherwise, we will check to see if the input record is “interesting”: if it is other than a valid ESD or END record, a code will simply be returned to the assembler telling it to output this record and continue processing.

```
*        Scan the ESD Record for useful names

         LH    R0,ESD_Amt          Get amount of ESD data
         LA    R1,ESD_Data         Point to first data item
         Using ESD_Item,R1         Base description of ESD item

Scan_Item DS    0H
         CLI   ESD_Type,ESD_Type_LD Check for SD or LD
         BH    Next_Item           No interesting data in this item

*        Have an interesting name

         L     R2,Alias_Count      Pick up count of names in table
         AL    R2,F1               Increment by one
         C     R2,AliasLim         Compare to max allowed number
         BH    Extra_Name          Branch if have too many
         ST    R2,Alias_Count      Store update name count
         SLL   R2,S3               Shift count left by 3 (*8)
         LA    R4,Alias_List-D8(R2) Calculate position in table
         MVC   D0(L8,R4),ESD_Name  Put name into table
         B     Next_Item
```

Figure 92. Object exit OBJX: scan ESD record for usable external names

In Figure 92, the ESD record in the input buffer is scanned for control section definition (SD) or entry point (label definition, or LD) names. If found, they are entered into the table of names at Alias_List. A test is made for table overflow; if this happens, the name is ignored, and the “overflow count” is incremented.

The names in the table will be output in reverse order of their receipt. This ensures that the first name received (typically, the main control section name) will be used on the NAME statement.

```
Extra_Name  DS  0H
         L     R2,Extra_Count      Get overflow counter
         AL    R2,F1               Increment by one
         ST    R2,Extra_Count      Store back

Next_Item DS   0H
         AL    R1,=A(ESD_Item_Len) Increment ESD Item pointer
         S     R0,=A(ESD_Item_Len) Decrement count of data bytes
         BP    Scan_Item           Look for further names on record
         B     Return_Rec          And return the record for output
         Drop  R1                  No ESD-item addressability now
```

Figure 93. Object exit OBJX: finish processing of ESD record

In Figure 93, the number of “overflow” names that cannot be held in the table of names is counted, and then the ESD record is scanned for further names, if any. If none are left, control is returned to the assembler to allow it to output the record.

```
*        Check for END record

Check_END  DS  0H
         CLC   ESD_ESD,=C'END'     See if it's an END record
         BNE   Return_Rec          If not END, return it for output

*        Process END Record: return it for output, request re-entry

         L     R0,Alias_Count      Check number of names
         LTR   R0,R0               If zero, no more to do
         BNZ   Set_Dumping         If not, go set return-to-me info

*        Object module has no entry points for aliasing or naming

         TM    Parm_Q,L'Parm_Q     See if quiet mode (no messages)
         BO    Return_Rec          Return with no message if so
         MVC   AXPSEVC,=A(AXPSEV0) Move severity code zero
         MVC   Err_Msg(L'Null_Name_Msg),Null_Name_Msg  Move message
         MVC   AXPERRL,=A(L'Null_Name_Msg)  Length of message
         BAS   R14,Batch_Init      Re-initialize counters
         B     Return_Rec          And return the END record

Set_Dumping DS  0H
         OI    Dumping,L'Dumping   Set dump flag on
         MVC   Names_To_Do,Alias_Count  Set count of names to output
         B     Exit_Return         Return record to assembler
```

Figure 94. Object exit OBJX: END of object module processing

In Figure 94, a check is first made for an END record. If the record is of some other type, it is returned to the assembler for output. Otherwise, a check is made to see if the object module contained any names usable for Linkage Editor control statements: it is possible that the object program consists entirely of PUNCH or REPRO records, or that all object code belongs to unnamed (blank) control sections (also known as Private Code, section type PC; see Figure 113 on page 136).

```
Do_Dumping  DS  OH

*        Output the names and other info

         Using Buffer,R10          R10 --> Working Buffer
         MVI   Buffer,C' '         Blank at start of output buffer
         MVC   Buffer+D1(Buff_Len-L1),Buffer  Propagate blanks
         L     R2,Names_To_Do      Get count of names left to do
         C     R2,F1               Check for last name
         BNH   Last_Name           Last one, output wrap-up statements

Normal_Dump DS OH
         BCTR  R2,0                Decrement count by one
         ST    R2,Names_To_Do      Save back
         SLL   R2,S3               Shift left 3 (*8)
         LA    R1,Alias_List(R2)   Point to name to be output

         MVC   Buff_Cmd(L'Alias_Cmd),Alias_Cmd  Move 'ALIAS'
         MVC   Buff_Dat,D0(R1)     Move name into buffer
         B     Exit_Return         Output record, and request return
```

Figure 95. Object exit OBJX: prepare an ALIAS statement for output

In Figure 95, pointers to the working buffers are first initialized, and then the number of remaining names is checked. If more than one name remains, an ALIAS statement for the current name is inserted into the output buffer, and control is returned to the assembler requesting output of this record and a return to the exit for further processing.

```
Last_Name  DS  OH

*        Check to see if SETSSI statement desired

         TM    Parm_S,L'Parm_S     See if S flag is set
         BZ    No_SSI              Branch if none
         TM    Flag_S_Done,L'Flag_S_Done  See if done already
         BO    No_SSI              Branch if done

         MVC   Buff_Cmd(L'SSI_Cmd),SSI_Cmd  Set command into buffer
         MVC   Buff_Dat,SSI        Move SSI data into buffer
         OI    Flag_S_Done,L'Flag_S_Done  Set flag for SETSSI done
         B     Exit_Return         Output record, and request return
```

Figure 96. Object exit OBJX: processing of SETSSI statement

In Figure 96, control reaches this point if there is a single name in the table remaining to be output. First, a check is made to see if a SETSSI statement was requested, and if so it is constructed in the output buffer and returned to the assembler with a request to return for the NAME record. A flag is set to indicate processing is completed for the SETSSI record.

```
No_SSI   DS    OH
         MVC   Buff_Cmd(L'Name_Cmd),Name_Cmd  Set command into buffer
         MVC   Buff_Dat,Alias_List Move first external name also
         TM    Parm_R,L'Parm_R     Check if '(R)' wanted
         BZ    Batch_Done          Branch if not, batch is done

         LA    R1,Buff_Dat+L'Buff_Dat-L1  Point to last char of name
         LA    R0,L8               Count of characters in name

Name_Loop DS   OH
         CLI   D0(R1),C' '         Check for trailing blank
         BNE   Put_R               Exit loop if found non-blank
         BCTR  R1,0                Move left 1 place
         BCT   R0,Name_Loop        Count down by 1 and try again
         B     Phase_Error_2       Should not come here

Put_R    DS    OH
         MVC   D1(L'Rep,R1),Rep    Add replace-option indicator
```

Figure 97. Object exit OBJX: output of NAME statement

In Figure 97, the final exit-produced statement is created. The first name from the Alias_List table is inserted into a NAME statement, and then (if the “R” exit-parm was requested, the characters (R) are appended.

```
Batch_Done DS  OH

*        Update counts for this batch

         TM    Parm_Q,L'Parm_Q     See if messages are to be suppressed
         BO    No_Msg              Branch if yes

         MVC   Err_Msg(L'Batch_Msg_1),Batch_Msg_1  Move message text
         MVC   AXPERRL,=A(L'Batch_Msg_1)  Set length of message
         MVC   AXPSEVC,=A(AXPSEV0) Set severity code 0 (Info message)

         L     R0,Alias_Count      Get count of names produced
         A     R0,Extra_Count      Add count of names ignored
         CVD   R0,DTemp            Convert to decimal
         ED    Err_Msg(L'Batch_Msg_1a),DTemp+D5  Edit count into field
         L     R0,Extra_Count      Count of ignored names
         LTR   R0,R0               Check to see if there were any
         BNZ   Bat_Msg_2           Branch if there were some
*                                  Overlay ignored count
         MVC   Err_Msg+Bat_Mg_1_Offb-1(L'Batch_Msg_2),Batch_Msg_2
         MVC   AXPERRL,=A(Bat_Mg_1_Offb-1+L'Batch_Msg_2)  Set length
         B     Bat_Msg_3           Continue
Bat_Msg_2 DS   OH
         CVD   R0,DTemp            Convert to decimal
         ED    Err_Msg+Bat_Mg_1_Offb(L'Batch_Msg_1b),DTemp+D5  Edit
Bat_Msg_3 DS   OH
```

Figure 98. Object exit OBJX: summary message at end of object module

In Figure 98, if the “Q” option has not been specified, a summary message is constructed and placed into the message buffer to tell how many names were processed, and how many (if any) were ignored.

```
No_Msg   DS    0H
         BAS   R14,Batch_Init      Re-initialize counters
         B     Return_Rec          Return record to caller

Batch_Init DS  0H
         L     R0,Batch_Count      Get count of decks
         AL    R0,F1               Increment by 1
         ST    R0,Batch_Count      Save count
         XC    Batch_Start(Batch_Len),Batch_Start  reset flags/counts
         BR    R14                 Return to caller

Exit_Return DS 0H
         MVC   AXPREAC,=A(AXPEEMP) Come back with empty buffer
         MVC   AXPRETC,=A(AXPOREC) Indicate assembler outputs record
         B     Return              Return to assembler

Return_Rec  DS  0H
         MVC   AXPREAC,=A(AXPCONT) Indicate continue normally
         MVC   AXPRETC,=A(AXPOREC) Indicate assembler outputs record
```

Figure 99. Object exit OBJX: re-initialization and return to the assembler

In Figure 99, the work areas for an individual object module are re-initialized, and control is returned to the assembler. At the two main processing return points. the appropriate return and reason codes are set to indicate the disposition of the working buffers and the flow of control between the exit and the assembler.

```
Return   DS    0H
         LTR   R8,R8               Check for local work area set up
         BZ    Return_2            Branch if not, save areas unchained
         TM    Parm_T,L'Parm_T     See if tracing requested
         BZ    Return_1            Branch if not
         OC    AXPERRL,AXPERRL     See if message is already present
         BNZ   Return_1            Branch if yes, no trace record

         MVC   Err_Buff(L80),Buffer     Move the record
         LA    R0,L80                   Set its length
         ST    R0,AXPERRL               Save length for message
         XR    R0,R0                    Clear R0
         ST    R0,AXPSEVC               Set message severity 0

Return_1 DS    0H
         L     R13,Save+D4         Get caller's R13 (if chained)

Return_2 DS    0H
         RETURN  (14,12)           Return to assembler

         Drop  R10
```

Figure 100. Object exit OBJX: return to assembler, possibly with tracing

In Figure 100, control is about to be returned to the assembler. A test is first made to see if the exit's work area has been initialized. If not, return is made directly to the assembler. Otherwise, a test is made for tracing of all input records. If tracing has been requested, the input record is copied to the error buffer for output if no other message is there already. Finally, control is returned to the assembler.

```
*********************************************************************
*        CLOSE Request                                              *
*********************************************************************
Close_Request  DS  0H

*        Create summary message if requested

         TM    Parm_Q,L'Parm_Q     See if running in quiet mode
         BO    Close_Return        Branch if yes
         MVC   Err_Msg(L'Close_Msg),Close_Msg  Move message
         L     R0,Batch_Count      Get number of decks
         CVD   R0,DTemp            Convert to decimal
         ED    Err_Msg(L'Close_Msg_1),DTemp+D5  Edit into message
         MVC   AXPERRL,=A(L'Close_Msg)  Set length
         MVC   AXPSEVC,=A(AXPSEV0) Set severity zero

Close_Return DS  0H
         MVC   AXPRETC,=A(AXPCONT) Set return code for 'normal' end
         MVC   AXPREAC,=A(No_Reason) Set reason code

*        Free storage and return

         LR    R1,R8               R1 points to storage area
         L     R0,=A(Work_Size)    R0 has its length
         L     R13,Save+D4         Restore pointer to caller's savearea

         FREEMAIN R,LV=(0),A=(1)   Free the work area

         RETURN  (14,12)           Return to caller, we're done.
```

Figure 101. Object exit OBJX: CLOSE processing

In Figure 101, a CLOSE request has been made by the assembler. A summary message is prepared if the “Q” option was not specified. Finally, the exit's working storage is returned to the operating system, and control returns to the assembler for the last time.

```
         Title 'Object-Editing Exit ''&Csect.'': Error Handling'

Bad_Version       DS  0H
         LA    R1,Bad_Vers_Msg     Point to error message info
         B     Proc_Error          Branch to process error message
Bad_Exit_Type     DS  0H
         LA    R1,Bad_Exit_Msg     Point to error message info
         B     Proc_Error          Branch to process error message
Wrong_Exit_Type   DS  0H
         LA    R1,Wrong_Exit_Msg   Point to error message info
         B     Proc_Error          Branch to process error message
Bad_OPEN_Request1 DS  0H
         LA    R1,Bad_OPEN_Msg1    Point to error message info
         B     Proc_Error          Branch to process error message
Bad_OPEN_Request2 DS  0H
         LA    R1,Bad_OPEN_Msg2    Point to error message info
         B     Proc_Error          Branch to process error message
Bad_Req_Opts      DS  0H
         LA    R1,Bad_Req_Msg      Point to error message info
         B     Proc_Error          Branch to process error message
```

Figure 102. Object exit OBJX: error processing (1)

In Figure 102, error conditions are handled by setting a pointer to error-message information and branching to the error processing routine. Further details will be explained following Figure 104 on page 131.

```
Bad_Parm_Str      DS  0H
         LA    R1,Bad_Parm_Msg     Point to error message info
         B     Proc_Error          Branch to process error message
Bad_Parm_Char     DS  0H
         LA    R1,Bad_Prm_Ch_Msg   Point to error message info
         B     Proc_Error          Branch to process error message
Bad_Type_Request  DS  0H
         LA    R1,Bad_Type_Msg     Point to error message info
         B     Proc_Error          Branch to process error message
Phase_Error_1     DS  0H
         LA    R1,Phase_1_Msg      Point to error message info
         B     Proc_Error          Branch to process error message
Phase_Error_2     DS  0H
         LA    R1,Phase_2_Msg      Point to error message info
         B     Proc_Error          Branch to process error message
```

Figure 103. Object exit OBJX: error processing (2)

In Figure 103, the remaining error conditions are handled. Further details will be explained following Figure 104 on page 131.

```
*        Handle error conditions and information messages

Proc_Error DS  0H                  Process error messages
          MVC   AXPRETC,=A(AXPCONT) Move return code, assume continuing
          MVC   AXPREAC,=A(No_Reason) Set reason code to zero
          BCTR  R1,0                Back up pointer to severity
          BCTR  R1,0                Back up pointer to length
          SR    R2,R2               Clear register for message length
          IC    R2,D1(,R1)          Get message severity
          ST    R2,AXPSEVC          Store severity code
          C     R2,=A(MSGSEVC)      Compare to max continuable value
          BNH   Proc_Error_1        Leave 'continue' return code alone
          MVC   AXPRETC,=A(AXPCBAD) Otherwise set code to 'fail'

Proc_Error_1 DS  0H
          IC    R2,D0(,R1)          Get message length
          C     R2,=A(Err_Buf_Len)  Compare to buffer size
          BNH   Proc_Text_2         Branch if it will fit
          L     R2,=A(Err_Buf_Len)  Truncate overly wordy messages

Proc_Text_2  DS  0H
          ST    R2,AXPERRL          Store true error message length
          BCTR  R2,0                Decrement length for executed move
          EX    R2,Move_Msg         Move message to buffer

          B     Return              Return to assembler

Move_Msg MVC   Err_Msg(*-*),D2(R1) Move message to error buffer

          Drop  R8,R9               Pointers to work area, req list
          Drop  R11                 End of error-buffer mapping
          Drop  R12                 Program base
```

Figure 104. Object exit OBJX: error message processing and output

In Figure 104, the length and severity code of the error message are extracted, the text of the message is moved to the error buffer, and the message is returned to the assembler.

The final three Drop statements indicate the end of the executable code in the exit; the remaining statements define constants and working storage.

```
         Title 'Object-Editing Exit ''&Csect.'': Messages'

*        Error Messages

                DC  AL1(L'Bad_Vers_Msg,AXPCBAD)   Length and severity
Bad_Vers_Msg    DC  C'Exit not coded at same level as Assembler'
                DC  AL1(L'Bad_Exit_Msg,AXPCBAD)   Length and severity
Bad_Exit_Msg    DC  C'Exit type requested is unrecognizable'
                DC  AL1(L'Wrong_Exit_Msg,AXPCBAD) Length and severity
Wrong_Exit_Msg  DC  C'Exit called for other than PUNCH or OBJECT'
                DC  AL1(L'Bad_OPEN_Msg1,AXPCBAD)  Length and severity
Bad_OPEN_Msg1   DC  C'Exit not initialized, and not entered for OPEN'
                DC  AL1(L'Bad_OPEN_Msg2,AXPCBAD)  Length and severity
Bad_OPEN_Msg2   DC  C'Exit initialized, but was entered for OPEN'
                DC  AL1(L'Bad_Req_Msg,MSGSEVC)    Length and severity
Bad_Req_Msg     DC  C'Invalid request-list options value'
                DC  AL1(L'Bad_Parm_Msg,MSGSEVC)   Length and severity
Bad_Parm_Msg    DC  C'Invalid parm-string length'
                DC  AL1(L'Bad_Prm_Ch_Msg,MSGSEVC) Length and severity
Bad_Prm_Ch_Msg  DC  C'Invalid character in parameter string'
                DC  AL1(L'Bad_Type_Msg,AXPCBAD)   Length and severity
Bad_Type_Msg    DC  C'Invalid action or operation type requested'
                DC  AL1(L'Phase_1_Msg,AXPCBAD)    Length and severity
Phase_1_Msg     DC  C'Expecting input record, zero buffer length'
                DC  AL1(L'Phase_2_Msg,AXPCBAD)    Length and severity
Phase_2_Msg     DC  C'Blank-name condition in (R) processing'
```

Figure 105. Object exit OBJX: error messages

In Figure 105, the error messages are defined. The text of each is preceded by two bytes, the first containing its length and the second its severity.

```
*        Information Messages

Null_Name_Msg   DC  C'Object Module contained no usable SD or LD names'
Batch_Msg_1     DC OC' dddsd Entry Names, dddsd were ignored.'
Batch_Msg_1a    DC  X'402020202120',C' Entry Names,'
Bat_Mg_1_Offb   Equ *-Batch_Msg_1a  Length of first message segment
Batch_Msg_1b    DC  X'402020202120',C' were ignored.'
Batch_Msg_2     DC  C' were processed.'  0-ignored appendage.
Dup_Prm_Ch_Msg  DC  C'Duplicated valid character in parameter string'
Close_Msg       DC OC' dddsd modules processed.'
Close_Msg_1     DC  X'402020202120',C' modules processed.'
```

Figure 106. Object exit OBJX: information messages

In Figure 106, the information messages that may be produced by the assembler are defined. No severity codes are associated with them, as they will all be issued with severity code zero. For messages such as Batch_Msg_1 containing mixed character and hexadecimal text, a readable version of the message is first defined with a zero duplication factor, followed by the actual message-text data.

```
         Title 'Object-Editing Exit ''&Csect.'': Constants'
F1        DC   F'1'                 Integer 1
AliasLim  DC   A(&MaxAlias.)        Maximum number of saved names
Alias_Cmd DC   C'ALIAS'             ALIAS command
Name_Cmd  DC   C'NAME'              NAME command
SSI_Cmd   DC   C'SETSSI'            SETSSI command
Rep       DC   C'(R)'               Replace indicator for NAME
P_Blanks  DC   CL(Max_Parm_Chars)' ' For upper-casing parm string

         LTORG
```

Figure 107. Object exit OBJX: constants

In Figure 107, The constants used by the exit are defined. Extensive use has been made of literals, so a LTORG statement is used to request that the assembler insert them into the program at this point.

```
         Title 'Object-Editing Exit ''&Csect.'': Working Storage'
WorkArea Dsect
Save     DS    18F                Traditional Register Save Area
DTemp    DS    D                  Doubleword temporary work area
DTemp_X  DS    X                  Extra byte for UNPK byte-swapping
         DS    3X                 Padding
FTemp    DS    F                  Fullword temporary work area
FTemp_X  DS    X                  Extra byte for UNPK byte-swapping
         DS    3X                 Padding

*        Areas cleared on initial entry

Init_Start   DS  0D               Start of global work area
Alias_List   DS  (&MaxAlias.)CL8  Table of names
SSI          DS  CL8              SSI info for SETSSI statements
Batch_Count  DS  F                Count of assemblies
```

Figure 108. Object exit OBJX: working storage (1)

In Figure 108, the first portions of the exit's working storage is defined: the save area, some conversion temporaries, and counters and other items not cleared for each object module.

```
*            Following five items must be kept together

Parm_Q       DS  0XL(X'80')       Q Flag
Parm_R       DS  0XL(X'40')       R Flag
Parm_S       DS  0XL(X'20')       S Flag
Parm_T       DS  0XL(X'10')       T Flag
             DS  X                Reserve storage for parm bits

Init_End     DS  0D               End of global work area
Init_Len     EQU Init_End-Init_Start  Length of global area
```

Figure 109. Object exit OBJX: working storage (2)

In Figure 109, further working storage is defined, and the end and length of the “global” work area are marked.

Of interest here is the technique used for defining bit flags, as noted in Figure 86 on page 121. First, observe that each bit flag is is defined by a DS statement that reserves no

storage, but which names whatever the next byte of storage will be. Second, note that the length attribute of each bit definition is explicitly defined as the position of the bit within the byte where it will eventually reside; all references to the bit position will be made using Length Attribute notation. Finally, the byte in which the bit flags will reside is defined without a name.

Then, by referring to the bit flags using statements like

```
          TM   Parm_Q,L'Parm_Q     Test Parm_Q bit
          OI   Parm_T,L'Parm_T     Set Parm_T bit
```

the programmer is assured that bits will never be associated with the wrong byte. It is easy to check in the assembler's symbol cross-reference listing that all references to bit flags are correctly paired; if an unpaired reference is found, it is easy to check the code for the improper reference.

```
*        Areas cleared for every batch

Batch_Start  DS  0D                Start of batch work area
Extra_Count  DS  F                 Names not handled
Alias_Count  DS  F                 Count of names in table
Names_To_Do  DS  F                 Count of names remaining to output
B_Msg_Count  DS  F                 Count of batch messages

*            Following four items must be kept together

Dumping      DS  0XL(X'80')        Dumping-names flag
Flag_S_Done  DS  0XL(X'40')        SETSSI statement has been output
Dup_Char     DS  0XL(X'20')        Duplicate (valid) parm character
             DS  X                 Reserve storage for flag bits

Batch_End    DS  0D                End of global work area
Batch_Len    EQU Batch_End-Batch_Start  Length of batch area

*        End of Work_Area

Work_End DS    0D                  End of Work_Area
Work_Size EQU  Work_End-WorkArea   Size of work area
```

Figure 110. Object exit OBJX: working storage (3)

In Figure 110, the remaining items in the exit's working storage are defined, and the end and length of the area cleared for each object module and for workarea initialization are defined. The same flag-bit-naming technique described above is also used here.

```
         Title 'Object-Editing Exit ''&Csect.'': Dummy Sections'
Buffer   DSect
         DS    CL9' '              Spaces
Buff_Cmd DS    CL8' '              Command name
         DS    CL2' '              Spaces
Buff_Dat DS    CL8' '              Data, names, etc
         DS    CL(L80-(*-Buffer))' '  Spaces
Buff_Len Equ   *-Buffer            Should have value 80

Err_Buff DSect
Err_Msg  DS    CL(L255)' '         Allocated space for messages
Err_Buf_Len Equ *-Err_Buff         Buffer length
```

Figure 111. Object exit OBJX: DSECTs for working buffers

In Figure 111, dummy control sections are defined for the output buffer and for the error-message buffer.

```
*        Object Module record format: ESD Record

ESD_Rec  Dsect
ESD_Tag  DC    AL1(Obj_Ind)        Record indicator
ESD_ESD  DC    C'ESD'              ESD-type record
         DC    CL6' '              Spaces
ESD_Amt  DC    Y(0)                Count of data bytes in ESD_Data
         DC    CL2' '              Spaces
ESD_ID_1 DC    Y(0)                ESD_ID of first SD/XD/CM/PC/ER/WX
ESD_Data DC    CL48' '             ESD data (up to 3 entries)
         DC    CL8' '              Spaces
ESD_Seqf DC    CL8' '              Deck-ID and sequence field
```

Figure 112. Object exit OBJX: object module ESD-record DSECT

In Figure 112, a dummy control section defines the structure of an object module ESD record. This is used to help manage the scanning of the ESD records provided by the assembler.

```
ESD_Item Dsect
ESD_Name DC    CL8' '             Name of ESD item
ESD_Type DC    X'00'              Type associated with the name

ESD_Type_SD Equ   X'00'              .. SD
ESD_Type_LD Equ   X'01'              .. LD
ESD_Type_ER Equ   X'02'              .. ER
ESD_Type_PC Equ   X'04'              .. PC
ESD_Type_CM Equ   X'05'              .. CM
ESD_Type_XD Equ   X'06'              .. XD/PR
ESD_Type_WX Equ   X'0A'              .. WX

ESD_Addr DC    AL3(0)             Address of ESD item
ESD_Flag DC    X'00'              Flag bits

ESD_Align   Equ   X'07'           Alignment for XD/PR items
ESD_Flag_LD Equ   X'40'           LD, ER, and WX
ESD_Flag_RS Equ   X'08'           RSECT
ESD_Flag_RMode Equ   X'04'        Bit 5     for RMode
ESD_Flag_RM24  Equ   X'00'        Bit 5 = 0 for RMode = 24
ESD_Flag_RMAny Equ   X'04'        Bit 5 = 1 for RMode = Any
ESD_Flag_AMode Equ   X'03'        Bits 6, 7 for AMode
ESD_Flag_AM24  Equ   X'00'        Bit 6 = 0 for AMode = 24
ESD_Flag_AM31  Equ   X'02'        Bit 6 = 1 for AMode = 31
ESD_Flag_AMAny Equ   X'03'        Bits 6,7=1 for AMode = Any

ESD_Len  DC    AL3(0)             Length of ESD name
ESD_Item_Len Equ  *-ESD_Item      Length of ESD item on record
```

Figure 113. Object exit OBJX: DSECT for ESD data items

In Figure 113, a description is provided for all the fields in an ESD data item contained in an ESD record. Although the exit does not refer to all the defined items, they are included in case further enhancements to the exit are made that might be able to make use of these other definitions.

```
          Title 'Mapping of Assembler I/O Exit Work Areas'
          ASMAXITP  PRINT=GEN
          End
```

Figure 114. Object exit OBJX: High Level Assembler communication area mapping

In Figure 114, the ASMAXITP macro is invoked to provide a DSECT mapping the communication area used by the assembler to communicate with the exit. (The ASMAXITP macro is provided as one of the sample programs delivered with High Level Assembler.)

Installing the Object Exit OBJX

The statements for the exit are assembled, and the resulting object code is converted into a loadable module:

- on MVS, it is link edited into an appropriate library and given the name OBJX. It may be marked re-entrant if desired.
- on CMS, LOAD the text deck from the assembly with the CLEAR and RLDSAVE options; then GENMOD to obtain a file with name OBJX and filetype MODULE.

Then, when the assembler is invoked, specify the parameters described in “Description of HLASM Object Exit OBJX” on page 112 above.

Glossary of Abbreviations and Terms

absolute symbol. A symbol whose value does not change if *Location Counter* values change in the program; a non-relocatable symbol.

ADATA. See *SYSADATA file*.

address. (1) (*n*) A number used by the processor at *execution time* to locate and reference operands or instructions in central processor storage. In the context of this document, an address is what reference manuals (such as the *Principles of Operation*) would call a virtual address.
(2) (*v*) To reference; to provide an *address* (sense no. 1) that may be used to reference an item in storage.
(3) Sometimes used to mean an *assembly time location*.

address constant. A field in a program containing values calculated at *assembly time*, *bind time*, or *execution time*, typically containing an *address*, an offset, or a length. The operands of an address constant often are expressions involving *internal symbols*, *external symbols*, or both.

address resolution. The process whereby the assembler converts *implied addresses* into *addressing halfwords*, using information in its *USING Table*.

addressable. (1) At *execution time* an operand is addressable if it lies either in the 4096 bytes starting at address zero, or in any 4096-byte region of storage whose lowest address is contained in one of *general purpose registers* 1 through 15.
(2) At *assembly time* an *implied address* is addressable if it can be validly *resolved* by the Assembler into a *base-displacement addressing halfword*, using information contained in the *USING Table* at the time of the resolution.

addressing halfword. A two-byte field in the second and/or third halfwords of a *machine language* instruction, composed of a 4-bit *base digit* and a 12-bit *displacement*. An address expressed in *base-displacement* format.

anchor. (1) The *base location* or *base register* specified in the second operand of a USING statement.
(2) The starting point of a chained list.

Assembler. A program which converts source statements written in *Assembler Language* into *machine language*, providing additional useful information such as diagnostic messages, symbol usage cross-references, and the like.

Assembler Language. The symbolic language accepted by High Level Assembler, in which program statements are written. (Often, these statements describe individual instructions; this is why Assembler Language is frequently characterized as a “low level” language.) The *Assembler* translates these statements to an equivalent representation of the program in *machine language*. Assembler Language is intelligible to human beings trained in the art, but excessive art may render it unintelligible. Compare *machine language*.

In this document, we sometimes distinguish two components: (1) *conditional assembly language* and (2) *ordinary assembly language*. See also Figure 115 on page 143.

assembly language. See *Assembler Language*.

assembly time. The period in the lifetime of a program when its representation as a sequence of symbolic statements is being converted to the desired equivalent *machine language* form.

attribute. A property of a *symbol* known to the *assembler*, typically the characteristics of the item named by the symbol, such as its type, length, etc. A program may request the assembler to provide values of symbol attributes using *attribute references*.

A *variable symbol* may have one attribute specific to the symbol itself (the number attribute), and many attributes specific to the *value* of the *variable symbol*.

attribute reference. A notation used to request the value of a *symbol attribute* from the assembler's *symbol table*, or of a *variable symbol* or its value.

BAL (acronym). Basic Assembler Language. Intended to mean *Assembler Language*. The use of this term is deprecated, due to possible confusions with the BAL (Branch and Link) instruction and the BASIC programming language. The *Assembler Language* implemented by High Level Assembler is neither basic nor BASIC.

base. See *base register*, *base address*.

base address. The *address* in one of *general purpose registers* 1 to 15 to which a *displacement* is added to obtain an *effective address*.

base digit. See *base register specification digit*.

base-displacement addressing. A technique for addressing central storage using a compact *base-displacement* format for representing the derivation of storage addresses.

base location. (1) In *base-displacement address resolution*, the first operand of a USING statement, from which *displacements* are to be calculated. For ordinary USING statements, the base location is assumed to be at a relative offset (*displacement*) of zero from the address contained in the *base register*; for *dependent USING* statements, the base location may be at a positive nonzero offset from the location specified in the *base register* eventually used to resolve an *implied address*.
(2) Informally, this term is sometimes used to mean (a) the origin of a control section, (b) a *base address* in a register at *execution time*, and (c) whatever the speaker likes.

base register. The *General Purpose Register* specified in the second operand of a *labeled USING* or *ordinary USING*.

base register specification digit. The 4-bit field in bit positions 0-3 of an *addressing halfword*.

bind time. The time following *assembly time* during which one or more *object modules* are combined to form an executable module, ready for loading into central storage at *execution time*. Also known as "link time".

COM. A statement declaring the start or resumption of a *common section.*

common section. A special *dummy control section* whose name is an *external symbol*. Common sections receive special treatment during program linking: space is allocated for the greatest length received for all common sections with a given name.

complex relocatability. An *attribute* of a *symbol* indicating that its value is neither constant nor varies in exactly the same way as changes to the origin of its containing section. See *relocatability attribute*.

conditional assembly. A form of assembly whose input is a mixture of *conditional assembly language* and *ordinary assembly language* statements, and whose outputs are statements of the *ordinary assembly language*. Statements of the *ordinary assembly language* are treated only as "text", and are not obeyed during conditional assembly.

conditional assembly language. The "outer" language that controls the sequencing, selection, and tailoring of *ordinary assembly language* statements, through the use of *variable symbols*, *sequence symbols*, *conditional assembly* expressions, and substitutions. See also Figure 115 on page 143.

conditional assembly function. See *external function* and *internal function*.

control section. The smallest independently *relocatable* unit of instructions and/or data. All elements of a given control section maintain the same fixed relative positions to one another at *assembly time*. These fixed relative positions at *assembly time* are usually (but not necessarily) maintained by the program after control sections are placed into storage at *execution time*.

CSECT. See *control section*

dependent USING. A form of USING statement in which the first operand is based or *anchored* at a relocatable address. May also take the form of a labeled dependent USING statement. See also *anchor*, *labeled USING*, and *ordinary USING*.

displacement. The 12-bit field in bit positions 4-15 of an *addressing halfword*. Frequently used to describe the offset (difference) between a given storage address and a *base address* that might be used to *address* (sense no. 2) it.

DSECT. See *dummy control section* and *control section*.

dummy control section. A *control section* with the additional special property that no object code is generated for any of its statements. Most DSECT definitions are used as mappings or templates for data structures. The three types of dummy control sections are (1) ordinary dummy control sections, (2) *common sections*, and (3) *dummy external control sections*.

EAR. See *Effective Address Register*.

effective address. The storage address or similar value calculated at *execution time*

from a *base address* and a *displacement*. See also *indexed effective address*.

Effective Address Register. An internal register used by the processor for calculating an *effective address*.

ESD. See *External Symbol Dictionary*.

execution time. The period in the lifetime of a program when its representation in *machine language* is interpreted by the processor as a sequence of instructions. (2) The time at which programmers whose programs consistently fail to execute correctly are themselves executed.

explicit address. An instruction address in which the *displacement*, and either the *base* or *index* or both, are fully specified in the instruction, and for which no *resolution* into *base-displacement* format is required.

extended object module. A new *generalized object file format* supporting long external names, section sizes up to 1GB, multi-segment modules, and other enhancements. Produced by High Level Assembler when the XOBJECT or *GOFF option* is specified. See also *object module*.

external dummy section. A dummy control section (DSECT) whose name is made part of the *External Symbol Dictionary*. The Binder, Linkage Editor or Loader will resolve the lengths and alignment requirements of external dummy sections in such a way that storage may be allocated to the entire collection of external dummy sections (see the definition of the CXD Assembler Instruction Statement in the Assembler Language Reference), and the offset of each dummy section may be defined to the program using Q-type address constants (again, refer to the Assembler Language Reference).

external function. A function defined by the user and invoked by the assembler by the SETAF and/or SETCF statements during *conditional assembly*. External functions may access the assembler's operating system environment and return either arithmetic or character values, and optional messages to be placed into the listing.

external symbol. A symbol whose name and value are a part of the object module text provided by the Assembler. Such names include (1) *control section* names, (2) referenced names declared in V-type address constants or EXTRN statements, (3) names of *common sections*, (4) names of *Pseudo Registers* or *external dummy sections*, (5) referenced names declared on ENTRY statements, and (6) symbols and character strings renamed through the use of the ALIAS statement. Compare to *internal symbol*.

External Symbol Dictionary. The set of *external symbols* defined or referenced in an assembly, and provided in the *object module* for later use during program linking or binding.

function. See *external function* and *internal function*.

generalized object file format (GOFF). A new form of *object module* produced by High Level Assembler, providing numerous enhancements and extensions not supported by the traditional *object module* format.

GOFF. See *generalized object file format*.

GOFF option. An *option* that causes High Level Assembler to generate an *object module* using the *generalized object file Format*.

General Purpose Registers. A set of 16 32-bit registers used in the System/360/370/390 family of processors for addressing, arithmetic, logic, shifting, and other general purposes. Compare to special purpose registers such as *Access Registers*, *Control Registers*, and *Floating Point Registers*.

GPR. See *General Purpose Register*

HLASM. High Level Assembler/MVS & VM & VSE (Release 1); High Level Assembler for MVS & VM & VSE (Release 2 and later).

High Level Assembler. IBM's most modern and powerful symbolic assembler for the System/370 and System/390 series of computers, running on the MVS, VM, and VSE operating systems. Not necessarily an oxymoron, as High Level Assembler can do much more than ordinary (low-level) assemblers.

implied address. An instruction address requiring *resolution* by the Assembler into *base-displacement* format; an address for which base and displacement are not explicitly specified. Also *implicit address*.

index. (1) The contents of that *index register* specified by the *index register specification digit* in an RX-type instruction. (2) Less frequently, the *index register specification digit* itself.

index digit. See *index register specification digit*.

index register specification digit. In an RX-type instruction, the 4-bit field contained in bit positions 12 through 15 of the instruction; the digit which, if not zero, specifies an *index register* to be used in calculating the *indexed effective address*

indexed effective address. The storage address or similar value calculated during program execution from a *base address*, a *displacement*, and an *index*. The term *effective address* is commonly used whether or not indexing is present.

index register. One of *general purpose registers* 1 through 15 specified by the *index register specification digit* in an RX-type instruction.

internal function. A function defined and executed by the assembler during *conditional assembly*, which acts on arithmetic, boolean, and character expressions to produce arithmetic, boolean, or character values. Compare *external function*.

internal symbol. A symbol naming an element of an *Assembler Language* program, which is assigned a single value by the *assembler*. Internal symbols are normally discarded at the end of the assembly, but may be retained in the *SYSADATA file*. Compare to *external symbol*.

internal symbol dictionary. See *symbol table*.

label. (1) The name field entry of an assembler or machine instruction statement. Normally, the presence of a label in the name field of an instruction statement will *define* the value of that label.
(2) In common parlance, the name of an instruction or data definition. This is more properly called a *name field symbol*.
(3) In High Level Assembler, the name field symbol of a USING statement, designating that statement as a *labeled USING*.

labeled USING. A form of USING statement with a *qualifier* symbol in the name field. Symbolic expressions resolved with respect to a labeled USING must use a *qualified symbol* with the *qualifier* of that labeled USING.

LC. See *location counter*.

Location Counter. A counter used by the Assembler to determine relative positions of all elements of a program as it is assembled.

location. A position within the object code of an assembled program, as determined by assigning values of the *Location Counter* during assembly. An *assembly time* value, sometimes confused with an *execution time address*.

machine language. The binary instructions and data interpreted and manipulated by the processor when the program is executed (at *execution time*). It is not meant to be intelligible to ordinary or normal human beings. Compare *Assembler Language*.

object module. A file produced by the Assembler, containing the *external symbols*, *machine language* instructions and data, and other data produced by assembling the source program. See also *extended object module*.

open code. Statements that are not within a macro definition or expansion. The statements in an assembly source file are typically in open code. See also *ordinary assembly language*.

options. Directives to the *Assembler* specifying various “global” controls over its behavior. For example, the PRINT option specifies that the assembler should produce a listing file. Options are specified by the user as a string of characters, as part of the command or statement that invokes the assembler, or on *PROCESS statements.

ordinary assembly language. The portion of the *Assembler Language* that includes machine instructions, data definitions, and assembler controls, but not including statements involved in *conditional assembly*. See *conditional assembly language*. See also Figure 115 on page 143.

ordinary symbol. See *internal symbol*.

ordinary USING. The oldest form of USING statement, in which (a) no entry is present in the name field, (b) the first operand specifies a *base address*, and (c) the second and successive operands are absolute expressions designating *General Purpose Registers* to be used as *base registers*.

PR. See *Pseudo Register* and *external dummy section*.

Pseudo Register. The name used by other processors such as the Linkage Editor and Loader for what the assembler calls an

external dummy section. See *external dummy section*.

qualified symbol. An ordinary symbol preceded by a *qualifier*, and separated from the *qualifier* by a period.

qualifier. An ordinary symbol, defined as a qualifier by its appearance in the name field of a *labeled USING statement*. It is used only in *qualified symbols* to direct *base-displacement addressing* resolutions to a specified register or *anchor location*.

RA. See *relocatability attribute*.

reenterable. See *reentrant*.

reentrant. (1) Capable of simultaneous execution by two or more asynchronously executing processes or processors, with only a single instance of the code image. Typically, reentrant programs are expected not to modify themselves, but this is neither a necessary nor sufficient condition for reentrancy.
(2) When requested by the RENT option, or in an *RSECT*, simple tests are made by High Level Assembler for conditions of obvious self-modification of the program being assembled.

relocatability attribute. Each independently relocatable element of an *Assembler Language* program (such as a *control section* or *external symbol*) is assigned a distinct relocatability attribute. Each symbol in the *symbol table* is assigned the relocatability attribute of the element to which it belongs. An *absolute symbol* is assigned a zero relocatability attribute. See also *simple relocatability* and *complex relocatability*.

relocatable. (1) Capable of being placed into storage at an arbitrary (possibly properly aligned) address; not requiring placement at a fixed or pre-specified address in order to execute correctly. (2) Independent of the origin address of the section. (3) Having a non-zero *relocatability attribute*, which can mean either *simple relocatability* or *complex relocatability*.

relocation. The assignment of new or different locations or addresses to a set of symbols or addresses, by adding or subtracting constants depending on a module's assigned storage addresses.

relocation ID. Same as *relocatability attribute*. A numeric value assigned by the assembler to each independently relocatable element of a program such as *control sections* and *external symbols*.

resolution. See *address resolution*.

resolved. See *address resolution*.

RSECT. A *reentrant control section*, distinguished from an ordinary *control section* (CSECT) only by (a) the presence of a flag in the *External Symbol Dictionary* and (b) that High Level Assembler will perform *reentrant* checking of instructions within the RSECT.

run time. See *execution time*.

sequence symbol. A *conditional assembly symbol* used to mark positions in a statement stream, typically inside a macro definition.

simple relocatability. An *attribute* of a *symbol* indicating that changes to the value of the origin location of a *control section* will cause the value of the symbol to change by the same amount. See also *absolute symbol* and *complex relocatability*.

symbol table. A table created and maintained by the Assembler, to assign values and attributes to all symbols in the program, including ordinary and variable symbols. Except for symbols named in V-type address constants, the symbol table contains only a single occurrence of an ordinary symbol.

SYSADATA file. A file created by the High Level Assembler when the ADATA *option* is specified, containing machine-readable information about all aspects of the assembled program and the assembly process.

system variable symbol. A *variable symbol* defined by the *assembler*;\, containing information about the assembly process. Its value cannot be changed by the programmer.

USING Table. A table maintained at *assembly time* by the Assembler, used for *resolution* of *implied addresses* into *base-displacement* form. Each entry contains the number of a *base register* and a *base location*.

variable symbol. A symbol prefixed with a single ampersand (&). Used during *conditional assembly* to assist with substitution, expression evaluation, and statement selection and sequencing. Unlike *ordinary symbols*, the values of certain variable symbols may change freely during an assembly.

Ordinary and Conditional Assembly

Comparison	Ordinary Assembly	Conditional Assembly
Generality	the “inner” language of instructions and data definitions	the “outer” language that controls and tailors the inner language
Usage	a language for programming a machine	a language for programming an assembler and its language
Inputs	statements from primary input, library (via COPY or macro call), and generated statements from macros and AINSERT statements	statements from primary input (and records via AREAD), library (via COPY and macro call), external functions
Outputs	generated machine language object code, records (via REPRO, PUNCH)	ordinary assembly statements and macro instructions, messages (via MNOTE), records (via AINSERT)
Symbols	ordinary symbols (internal and external)	variable symbols, sequence symbols
Symbol declaration	ordinary symbols appear in the name field of ordinary assembly statements (except names in V-type address constants); always explicitly declared	sequence symbols appear in the name field of any statement; variable symbols are (a) user-declared (implicit or explicit declaration), (b) system, or (c) macro parameters (both implicit)
Statement labels	ordinary symbols take the values of locations in the ordinary assembly statement stream, and other assigned values, or are positional arguments in macro calls	sequence symbols denote positions in the conditional assembly statement stream
Symbol scope	internal and external; external symbols persist in the object code beyond assembly time	variable symbols have local or global scope; sequence symbols have local scope; both discarded at assembly end
Symbol types and values	ordinary symbols have no types; values are normally assigned from Location Counter values or by EQU statements	variable symbols have arithmetic, boolean, or character types and values
Symbol attributes	ordinary symbols have many attributes	variable symbols have only the property of maximum subscript (if dimensioned), but their *values* may have attributes
Expression evaluation	expressions in ordinary statements, and in A-type and Y-type address constants	expressions in conditional-assembly statements
Expression operators	`+, -, *, /`	`+, -, *, /`; internal arithmetic functions; internal boolean functions; internal character functions; external arithmetic and character functions
Attribute Operators	`L', I', S'`	`T', L', I', S', D', K', N', O'`

Figure 115. Comparison of Ordinary and Conditional Assembly

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

X