High Level Assembler for z/OS & z/VM & z/VSE

Language Reference

Version 1 Release 6

SC26-4940-06

Note
Before using this information and the product it supports, be sure to read the general information under "Notices" on page 379.

This edition applies to IBM High Level Assembler for z/OS & z/VM & z/VSE, Release 6, Program Number 5696-234 and to any subsequent releases until otherwise indicated in new editions. Make sure that you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality.

IBM welcomes your comments. For information on how to send comments, see "How to send your comments to IBM" on page xvii.

Contents

Figures

Tables

About this document

This manual describes the syntax of assembler language statements, and provides information about writing source programs that are to be assembled by IBM High Level Assembler for z/OS & z/VM & z/VSE, Licensed Program 5696-234, from here on referred to as "High Level Assembler", or "the assembler". It is meant to be used with the HLASM Programmer's Guide.

Detailed definitions of machine instructions are not included in this manual. See "Bibliography" on page 381 for a list of manuals that provide this information.

Throughout this book, we use these indicators to identify platform-specific information:

- Prefix the text with platform-specific text (for example, "Under CMS...")
- Add parenthetical qualifications (for example, "(CMS)")
- A definition list, for example:

  z/OS Informs you of information specific to z/OS®.

  z/VM Informs you of information specific to z/VM®.

  z/VSE Informs you of information specific to z/VSE®.

CMS is used in this manual to refer to Conversational Monitor System on z/VM.

Who should use this manual

HLASM Language Reference is for application programmers coding in the High Level Assembler language. It is not intended to be used for tutorial purposes, but is for reference only. If you are interested in learning more about assemblers, most libraries have tutorial books on the subject. It assumes that you are familiar with the functional details of the Enterprise Systems Architecture, and the role of machine-language instructions in program execution.

Programming interface information

This manual is intended to help the customer create application programs. This manual documents General-Use Programming Interface and Associated Guidance Information provided by IBM High Level Assembler for z/OS & z/VM & z/VSE.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM High Level Assembler for z/OS & z/VM & z/VSE.

Organization of this manual

This manual is organized as follows:

Assembler language structure and concepts

Chapter 1, "Introduction," on page 1 describes the assembler language and how the assembler processes assembler language source statements. It also describes the relationship between the assembler and the operating system, and suggests ways to make the task of coding easier.

Chapter 2, "Coding and structure," on page 9 describes the coding rules for and the structure of the assembler language. It also describes the language elements in a program.

Chapter 3, "Program structures and addressing," on page 43 describes the concepts of addressability and symbolic addressing. It also describes control sections and the linkage between control sections.

Machine and assembler instruction statements

Chapter 4, "Machine instruction statements," on page 65 describes the machine instruction types and their formats.

Chapter 5, "Assembler instruction statements," on page 83 describes the assembler instructions in alphabetical order.

Macro language

Chapter 6, "Introduction to macro language," on page 207 describes the macro facility concepts including macro definitions, macro instruction statements, source and library macro definitions, and conditional assembly language.

Chapter 7, "How to specify macro definitions," on page 213 describes the components of a macro definition.

Chapter 8, "How to write macro instructions," on page 259 describes how to call macro definitions using macro instructions.

Chapter 9, "How to write conditional assembly instructions," on page 279 describes the conditional assembly language including SET symbols, sequence symbols, data attributes, branching, and the conditional assembly instructions.

Chapter 10, "MHELP instruction," on page 351 describes the MHELP instruction that you can use to control a set of macro trace and dump facilities.

Appendixes

Appendix A, "Assembler instructions," on page 355 summarizes the assembler instructions and assembler statements, and the related name and operand entries.

Appendix B, "Summary of constants," on page 359 summarizes the types of constants and related information.

Appendix C, "Macro and conditional assembly language summary," on page 363 summarizes the macro language described in Part 3. This summary also includes a summary table of the system variable symbols.

Appendix D, "Standard character set code table," on page 375 shows the code table for the assembler's standard character set.

High Level Assembler documents

High Level Assembler runs under z/OS, z/VM, and z/VSE. Its publications for the z/OS, z/VM, and z/VSE operating systems are described in this section.

Documents

Here are the High Level Assembler documents. The table shows tasks, and which document can help you with that particular task. Then there is a list showing each document and a summary of its contents.

Table 1. IBM High Level Assembler for z/OS & z/VM & z/VSE documents

Task	Document	Order Number
Installation and customization	HLASM V1R6 Installation and Customization Guide	SC26-3494
	HLASM V1R6 Programmer's Guide	SC26-4941
	HLASM V1R6 Toolkit Feature Installation Guide	GC26-8711
Application Programming	HLASM V1R6 Programmer's Guide	SC26-4941
	HLASM V1R6 Language Reference	SC26-4940
	HLASM V1R6 Toolkit Feature User's Guide	GC26-8710
	HLASM V1R6 Toolkit Feature Interactive Debug Facility User's Guide	GC26-8709
Diagnosis	HLASM V1R6 Installation and Customization Guide	SC26-3494

HLASM Installation and Customization Guide
Contains the information you need to install and customize, and diagnose failures in, the High Level Assembler product.

The diagnosis section of the book helps users determine if a correction for a similar failure has been documented previously. For problems not documented previously, the book helps users to prepare an APAR. This section is for users who suspect that High Level Assembler is not working correctly because of some defect.

HLASM Language Reference
Presents the rules for writing assembler language source programs to be assembled using High Level Assembler.

HLASM Programmer's Guide
Describes how to assemble, debug, and run High Level Assembler programs.

HLASM Toolkit Feature Installation and Customization Guide
Contains the information you need to install and customize, and diagnose failures in, the High Level Assembler Toolkit Feature.

HLASM Toolkit Feature User's Guide
Describes how to use the High Level Assembler Toolkit Feature.

HLASM Toolkit Feature Debug Reference Summary
Contains a reference summary of the High Level Assembler Interactive Debug Facility.

HLASM Toolkit Feature Interactive Debug Facility User's Guide
Describes how to use the High Level Assembler Interactive Debug Facility.

Collection kits

The High Level Assembler publications are available in these collection kits:
- *z/OS V1Rx and Software Products DVD Collection*, SK3T-4271 (Book and PDF)
- *z/OS V1Rx Information Center DVD*, SK5T-7089
- *z/VM Collection on DVD*, SK5T-7054
- *z/VM VxRy Information Center DVD*, SK5T-7098
- *z/VSE Collection*, SK3T-8348

For more information about High Level Assembler, see the High Level Assembler Web site, at

`http://www-306.ibm.com/software/awdtools/hlasm/`

Related publications

See "Bibliography" on page 381 for a list of publications that supply information you might need while you are using High Level Assembler.

Syntax notation

Throughout this book, syntax descriptions use this structure:
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►— symbol indicates the beginning of a statement.

  The —► symbol indicates that the statement syntax is continued on the next line.

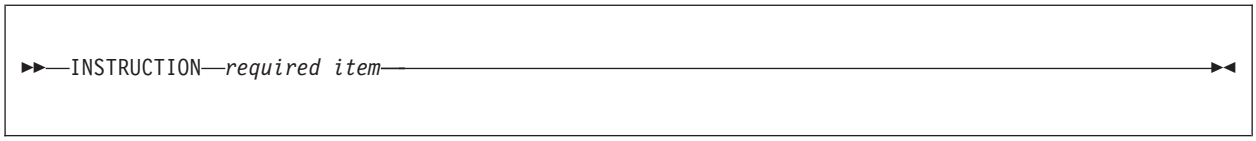
  The ►— symbol indicates that a statement is continued from the previous line.

  The —►◄ indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►— symbol and end with the —► symbol.

- **Keywords** appear in uppercase letters (for example, ASPACE) or uppercase and lowercase (for example, PATHFile). They must be spelled exactly as shown. Lowercase letters are optional (for example, you could enter the PATHFile keyword as PATHF, PATHFI, PATHFIL, or PATHFILE).

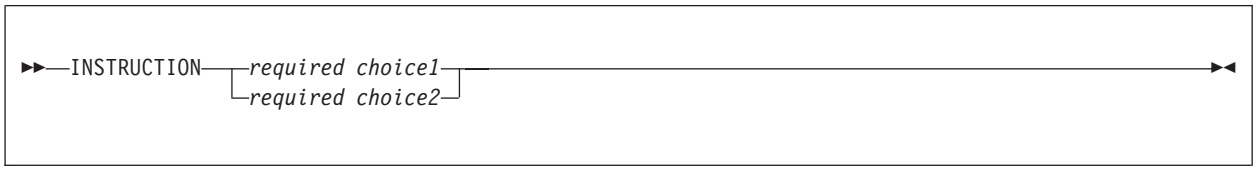
  Variables appear in all lowercase letters in a special typeface (for example, *integer*). They represent user-supplied names or values.
- If punctuation marks, parentheses, or such symbols are shown, they must be entered as part of the syntax.
- Required items appear on the horizontal line (the main path).



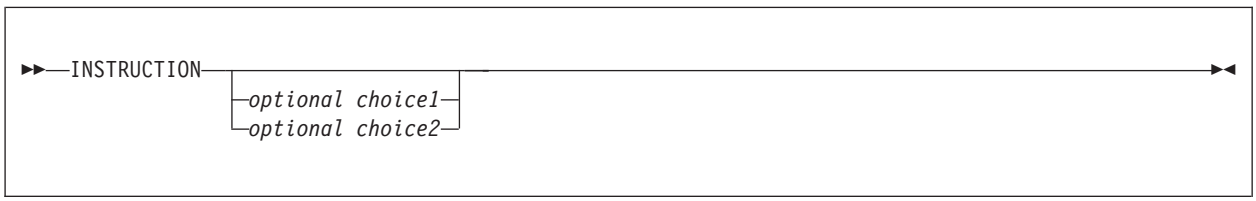
- Optional items appear below the main path. If the item is optional and is the default, the item appears above the main path.

- When you can choose from two or more items, they appear vertically in a stack.

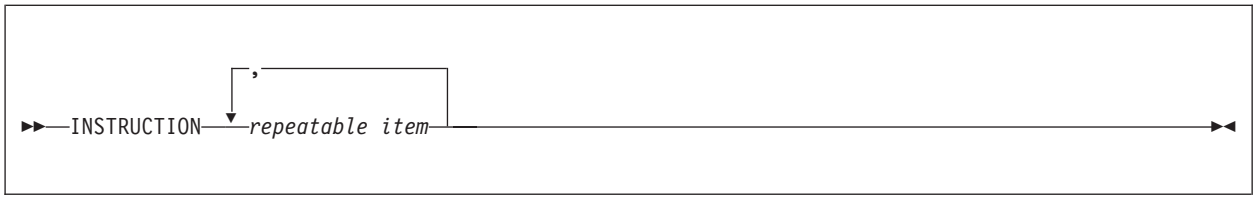
  If you **must** choose one of the items, one item of the stack appears on the main path.



  If choosing one of the items is optional, the whole stack appears below the main path.



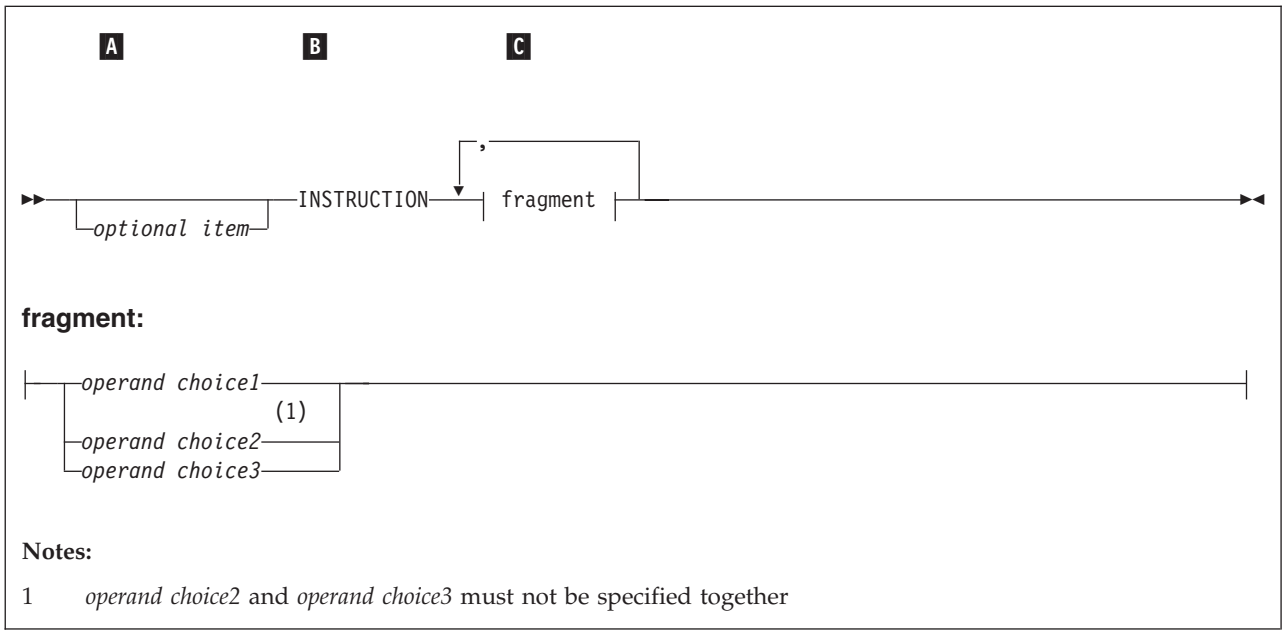
- An arrow returning to the left above the main line indicates an item that can be repeated. When the repeat arrow contains a separator character, such as a comma, you must separate items with the separator character.



  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

Format

The following example shows how the syntax is used.



A The item is optional, and can be coded or not.

B The INSTRUCTION key word must be specified and coded as shown.

C The item referred to by "fragment" is a required operand. Allowable choices for this operand are given in the fragment of the syntax diagram shown below "fragment" at the bottom of the diagram. The operand can also be repeated. That is, more than one choice can be specified, with each choice separated by a comma.

How to send your comments to IBM

If you especially like or dislike anything about this book, feel free to send us your comments.

You can comment on what you regard as specific errors or omissions, and on the accuracy, organization, subject matter, or completeness of this book. Please limit your comments to the information that is in this book and to the way in which the information is presented. Speak to your IBM representative if you have suggestions about the product itself.

When you send us comments, you grant to IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

You can get your comments to us quickly by sending an e-mail to **idrcf@hursley.ibm.com**. Alternatively, you can mail your comments to:

User Technologies,
IBM United Kingdom Laboratories,
Mail Point 095, Hursley Park,
Winchester, Hampshire,
SO21 2JN, United Kingdom

Please ensure that you include the book title, order number, and edition date.

If you have a technical problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative
- Call IBM technical support
- Visit the IBM support web page

Summary of changes

Date of Publication
August 2013

Form of Publication
Seventh Edition, SC26-4940-HLASM Language Reference

Here is a list of the changes to HLASM that are explained in this document.

Changed Assembler instructions

- New QY-type and SY-type address constants provide resolution into long displacement.
- Support for three decimal floating-point data types, increasing instruction addressability and reducing the need for additional instructions.

Unified Opcode table

- OPTABLE option
  - The OPTABLE option is permitted on the *PROCESS statement.
- **Mnemonic tagging**
  - Suffix tags for instruction mnemonics let you use identically named macro instructions and machine instructions in the same source program.

Chapter 1. Introduction

A computer can understand and interpret only machine language. Machine language is in binary form and, thus, difficult to write. The assembler language is a symbolic programming language that you can use to code instructions instead of coding in machine language.

Because the assembler language lets you use meaningful symbols made up of alphabetic and numeric characters, instead of just the binary digits 0 and 1 used in machine language, you can make your coding easier to read, understand, and change. The assembler must translate the symbolic assembler language into machine language before the computer can run your program. The specific procedures followed to do this vary according to the system you are using. However, the method is basically the same for all systems:

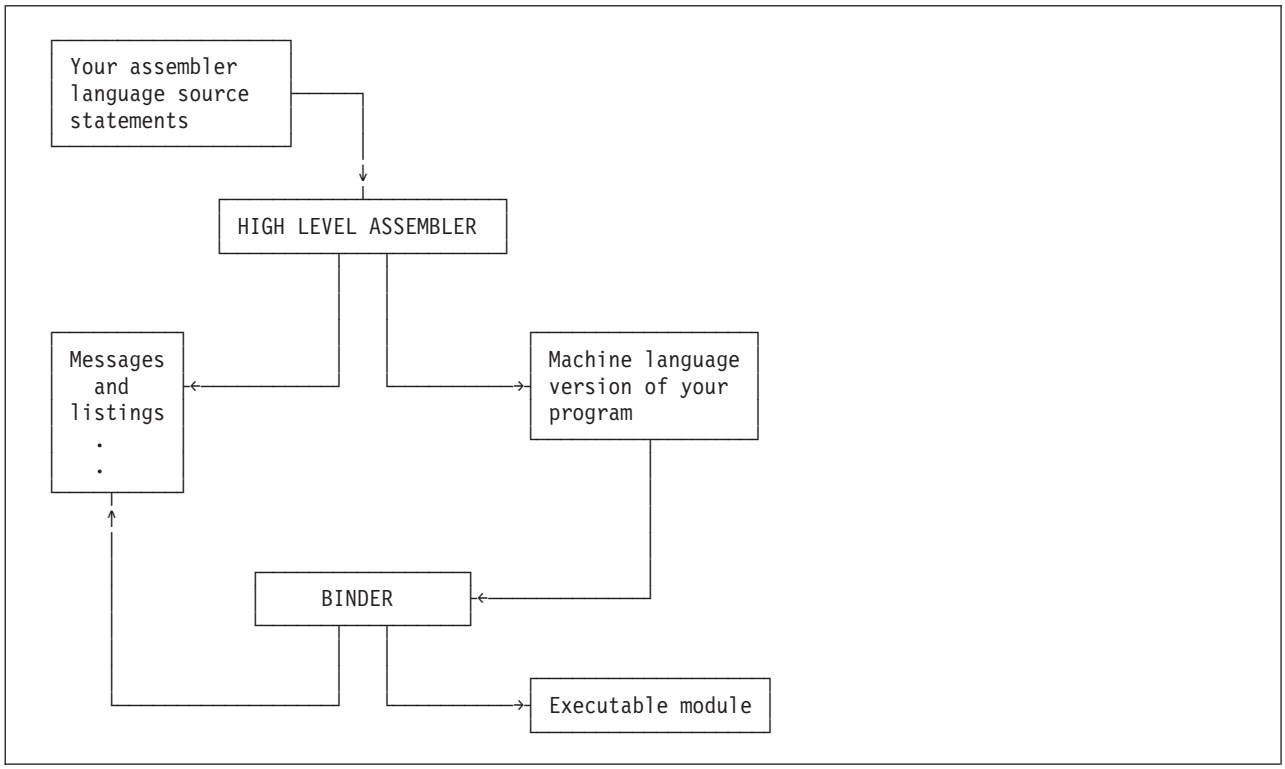


Figure 1. Assembling and link-editing your assembler language program

Your program, written in the assembler language, becomes the source module that is input to the assembler. The assembler processes your source module and produces an object module in machine language (called object code). The object module can be used as input to be processed by the linker or the binder. The linker or binder produces a load module (z/OS and CMS), or a phase (z/VSE), that can be loaded later into the main storage of the computer. When your program is loaded, it can then be run. Your source module and the object code produced are printed, along with other information, on a program listing.

Language compatibility

The assembler language supported by High Level Assembler has functional extensions to the languages supported by Assembler H Version 2 and DOS/VSE Assembler. High Level Assembler uses the same language syntax, function, operation, and structure as Assembler H Version 2. Similarly, the functions provided by the Assembler H Version 2 macro facility are all provided by High Level Assembler.

Migration from Assembler H Version 2 or DOS/VSE Assembler to High Level Assembler requires an analysis of existing assembler language programs to ensure that they do not contain:

- Macro instructions with names that conflict with High Level Assembler symbolic operation codes
- SET symbols with names that conflict with the names of High Level Assembler system variable symbols
- Dependencies on the type attribute values of certain variable symbols or macro instruction operands

Except for these possible conflicts, and with the appropriate High Level Assembler option values, source language source programs written for Assembler H Version 2 or DOS/VSE Assembler, that assemble without warning or error diagnostic messages, should assemble correctly using High Level Assembler.

z/VSE An E-Deck refers to a macro source book of type E that can be used as the name of a macro definition to process in a macro instruction. E-Decks are stored in edited format, and High Level Assembler requires that library macros be stored in source statement format. A library input exit can be used to analyze a macro definition, and, in the case of an E-Deck, call the z/VSE ESERV program to change, the E-Deck definition, line by line, back into source format required by the assembler, without modifying the original library file.

See the section titled *Using the High Level Assembler Library Exit for Processing E-Decks* in the *z/VSE: Guide to System Functions*. This section describes how to set up the exit and how to use it.

Assembler language

The assembler language is the symbolic programming language that lies closest to the machine language in form and content. The assembler language is useful when:

- You need to control your program closely, down to the byte and even the bit level.
- You must write subroutines for functions that are not provided by other symbolic programming languages, such as COBOL, Fortran, or PL/I.

The assembler language is made up of statements that represent either instructions or comments. The instruction statements are the working part of the language and are divided into the following three groups:

- Machine instructions
- Assembler instructions
- Macro instructions

Machine instructions

A machine instruction is the symbolic representation of a machine language instruction of the following instruction sets:

- IBM System/370
- IBM System/370 Extended Architecture (370-XA)
- Enterprise Systems Architecture/370 (ESA/370)
- Enterprise Systems Architecture/390 (ESA/390)
- z/Architecture®

It is called a machine instruction because the assembler translates it into the machine language code that the computer can run. Machine instructions are described in Chapter 4, “Machine instruction statements,” on page 65.

Assembler instructions

An assembler instruction is a request to the assembler to do certain operations during the assembly of a source module; for example, defining data constants, reserving storage areas, and defining the end of the source module. Except for the instructions that define constants, and the instruction used to generate no-operation instructions for alignment, the assembler does not translate assembler instructions into

object code. The assembler instructions are described in Chapter 3, “Program structures and addressing,” on page 43, Chapter 5, “Assembler instruction statements,” on page 83, and Chapter 9, “How to write conditional assembly instructions,” on page 279.

Macro instructions

A macro instruction is a request to the assembler program to process a predefined sequence of instructions called a *macro definition*. From this definition, the assembler generates machine and assembler instructions, which it then processes as if they were part of the original input in the source module.

IBM supplies macro definitions for input/output, data management, and supervisor operations that you can call for processing by coding the required macro instruction. (These IBM-supplied macro instructions are described in the applicable *Macro instructions* manual.)

You can also prepare your own macro definitions, and call them by coding the corresponding macro instructions. Rather than code all this sequence each time it is needed, you can create a macro instruction to represent the sequence and then, each time the sequence is needed, code the macro instruction statement. During assembly, the sequence of instructions represented by the macro instruction is inserted into the source program.

A complete description of the macro facility, including the macro definition, the macro instruction, and the conditional assembly language, is given in Chapter 6, “Introduction to macro language,” on page 207 through Chapter 10, “MHELP instruction,” on page 351.

Assembler program

The *assembler program*, also referred to as the *assembler*, processes the machine, assembler, and macro instructions you have coded (source statements) in the assembler language, and produces an object module in machine language.

Basic functions

Processing involves the translation of source statements into machine language, assignment of storage locations to instructions and other elements of the program, and performance of auxiliary assembler functions you have designated. The output of the assembler program is the object program, a machine language translation of the source program. The assembler produces a printed listing of the source statements and object program statements and additional information, such as error messages, that are useful in analyzing the program. The object program is in the format required by the binder.

Associated data

The assembler can produce an associated data file that contains information about the source program and the assembly environment. The ADATA information includes information such as:

- Data sets used by the assembler
- Program source statements
- Macros used by the assembler
- Program symbols
- Program object code
- Assembly error messages

Different subsets of this information are needed by various consumers, such as configuration managers, debuggers, librarians, metrics collectors, and many more.

Controlling the assembly

You can control the way the assembler produces the output from an assembly, using assembler options and assembler language instructions.

Assembler options are described in the chapter “Controlling Your Assembly with Options” in the HLASM Programmer's Guide. A subset of assembler options can be specified in your source program using the *PROCESS statement described in “*PROCESS statement” on page 84.

Assembler language instructions are assembler language source statements that cause the assembler to perform a specific operation. Some assembler language instructions, such as the DC instruction, generate object code. Assembler language instructions are categorized as follows:

Assembler Instructions

These include instructions for:

- Producing associated data
- Assigning base registers
- Defining data constants
- Controlling listing output
- Redefining operation codes
- Sectioning and linking programs
- Defining symbols

These instructions are described in Chapter 5, “Assembler instruction statements,” on page 83.

Macro Instructions

These instructions let you define macros for generating a sequence of assembler language statements from a single instruction. These instructions are described in Chapter 6, “Introduction to macro language,” on page 207 through Chapter 10, “MHELP instruction,” on page 351.

Conditional Assembly Instructions

These instructions let you perform general arithmetic and logical computations, and condition tests that can vary the output generated by the assembler. These instructions are described under “Conditional assembly instructions” on page 302.

Processing sequence

The assembler processes the machine and assembler language instructions at different times during its processing sequence. You should be aware of the assembler's processing sequence in order to code your program correctly.

The assembler processes most instructions twice, first during conditional assembly and, later, at assembly time. Some processing is done only during conditional assembly.

Conditional assembly and macro instructions

The assembler processes conditional assembly instructions and macro processing instructions during conditional assembly. During this processing the assembler evaluates arithmetic, logical, and character conditional assembly expressions. Conditional assembly takes place before assembly time.

The assembler processes the machine and ordinary assembler instructions generated from a macro definition called by a macro instruction at assembly time.

Machine instructions

The assembler processes all machine instructions, and translates them into object code at assembly time.

Assembler instructions

The assembler processes ordinary assembler instructions at assembly time. During this processing:

- The assembler evaluates absolute and relocatable expressions (sometimes called assembly-time expressions)
- Some instructions, such as ADATA, ALIAS, CATTR and XATTR (z/OS and CMS), DC, DS, ENTRY, EXTRN, PUNCH, and REPRO, produce output for later processing by programs such as the binder.

The assembler prints in a program listing all the information it produces at the various processing times discussed above. The assembler also produces information for other processors. The binder uses such information at link-edit time to combine object modules into load modules. At program fetch time, the load module produced by the binder is loaded into virtual storage. Finally, at execution time, the computer runs the load module.

Relationship of assembler to operating system

High Level Assembler operates under the z/OS operating system, the CMS component of the z/VM operating system, the z/VSE operating system, and Linux for System z®. These operating systems provide the assembler with services for:

- Assembling a source module
- Running the assembled object module as a program

In writing a source module, you must include instructions that request any required service functions from the operating system.

z/OS provides the following services:

- For assembling the source module:
  - A control program
  - Sequential data sets to contain source code
  - Libraries to contain source code and macro definitions
  - Utilities
- For preparing for the execution of the assembler program as represented by the object module:
  - A control program
  - Storage allocation
  - Input and output facilities
  - Binder
  - Loader

CMS provides the following services:

- For assembling the source module:
  - An interactive control program
  - Files to contain source code
  - Libraries to contain source code and macro definitions
  - Utilities
- For preparing for the execution of the assembler program as represented by the object modules:
  - An interactive control program
  - Storage allocation
  - Input and output facilities
  - Linker
  - A loader

z/VSE provides the following services:

- For assembling the source module:
  - A control program
  - Sequential data sets to contain source code
  - Libraries to contain source code and macro definitions
  - Utilities
- For preparing for the execution of the assembler program as represented by the object module:
  - A control program
  - Storage allocation
  - Input and output facilities
  - Linker

Linux for System z provides the following services:

- For assembling the source module:
  - An interactive control program
  - Files to contain source code
  - Utilities
- For preparing for the execution of the assembler program as represented by the object modules:
  - An interactive control program
  - Storage allocation
  - Input and output facilities
  - Linker
  - A loader

Coding made easier

It can be difficult to write an assembler language program using only machine instructions. The assembler provides additional functions that make this task easier. Here is a summary of these additional functions:

Symbolic representation of program elements

Symbols greatly reduce programming effort and errors. You can define symbols to represent storage addresses, displacements, constants, registers, and almost any element that makes up the assembler language. These elements include operands, operand subfields, terms, and expressions. Symbols are easier to remember and code than numbers; moreover, they are listed in a symbol cross reference table, which is printed in the program listings. Thus, you can easily find a symbol when searching for an error in your code. See page "Symbols" on page 25 for details about symbols, and how you can use them in your program.

Variety in data representation

You can use decimal, binary, hexadecimal, or character representation of machine language binary values in writing source statements. You select the representation best suited to the purpose. The assembler converts your representations into the binary values required by the machine language.

Controlling address assignment

If you code the correct assembler instruction, the assembler computes the relative offset, or displacement from a base address, of any symbolic addresses you specify in a machine instruction. It inserts this displacement, along with the base register assigned by the assembler instruction, into the object code of the machine instruction.

At execution time, the object code of address references must be in relative-immediate or base-displacement form. The computer obtains the required address by adding the displacement to the base address contained in the base register, or from the relative-immediate offset of the instruction.

Relocatability

The assembler produces an object module that is independent of the location it is initially assigned in virtual storage. That is, it can be loaded into any suitable virtual storage area without affecting program execution. This is made easier because most addresses are assembled in their base-displacement form.

Sectioning a program

You can divide a source module into one or more control sections. After assembly, you can include or delete individual control sections from the resulting object module before you load it for execution. Control sections can be loaded separately into storage areas that are not contiguous. A discussion of sectioning is contained in "Source program structures" on page 44.

Linkage between source modules

You can create symbolic linkages between separately assembled source modules. This lets you refer symbolically from one source module to data and instructions defined in another source module. You can also use symbolic addresses to branch between modules.

A discussion of sectioning and linking is contained in "Source program structures" on page 44.

Program listings

The assembler produces a listing of your source module, including any generated statements, and the object code assembled from the source module. You can control the form and content of the listing using assembler listing control instructions, assembler options, and user I/O exits. The listing control instructions are described in Chapter 5, "Assembler instruction statements," on page 83, and in "Processing statements" on page 224. Assembler options and user I/O exits are discussed in the chapters "Controlling Your Assembly with Options" and "Providing User Exits" in the *HLASM Programmer's Guide*.

The assembler also prints messages about actual errors and warnings about potential errors in your source module.

Multiple source modules

The assembler can assemble more than one source module in a single input stream, if the BATCH option is specified. For more information about the BATCH option, see the section "BATCH" in the HLASM Programmer's Guide.

An "input stream" may contain one or more "source modules", and may also consist of one or more data sets if the host operating system supports data set or file concatenation. A "source module" is a single assembly.

Double-byte character set notation

Double-byte character set (DBCS) characters in terms, expressions, character strings, and comments are delimited by shift-out and shift-in characters. In this manual, the shift-out delimiter is represented pictorially by the < character, and the shift-in delimiter is represented pictorially by the > character. The EBCDIC codes for the shift-out and shift-in delimiters are X'0E' and X'0F'.

The following figure summarizes the DBCS notation used throughout this manual.

Characters	Represents
<	Shift-out (SO)
>	Shift-in (SI)
D1D2D3...	Double-byte characters
DaDbDc...	Double-byte characters
.A.B.C.'.&.,	EBCDIC characters in double-byte form: A, B, C, apostrophe, ampersand, and comma. The dots separating the letters represent the hexadecimal value X'42'. A double-byte character that contains the value of an EBCDIC ampersand or apostrophes in either byte is not recognized as a delimiter when enclosed by SO and SI.
eeeeeee	Single-byte (EBCDIC) characters
abcd...	Single-byte (EBCDIC) characters
XXX	Extended continuation indicator for macro-generated statements
+++	Alternative extended continuation indicator for macro-generated statements

Chapter 2. Coding and structure

This chapter provides information about assembler language coding conventions and assembler language structure.

Character set

High Level Assembler provides support for both standard single-byte characters and double-byte characters.

Standard character set

The standard (default) character set used by High Level Assembler is a subset of the EBCDIC character set. This subset consists of letters of the alphabet, national characters, the underscore character, digits, and special characters. The complete set of characters that make up the standard assembler language character set is shown in Table 2.

Table 2. Standard character set

Character Type	Character Set
Alphabetic characters	a through z A through Z national characters @, $, and # underscore character _
Digits	0 through 9
Special characters	+ - , = . * ( ) ' / & space

For a description of the binary and hexadecimal representations of the characters that make up the standard character set, see Appendix D, "Standard character set code table," on page 375.

When you code terms and expressions (see "Terms, literals, and expressions" on page 24) in assembler language statements, you can only use the set of characters described above. However, when you code remarks, comments, or character strings between paired apostrophes, you can use any character in the EBCDIC character set.

The term *alphanumeric characters* includes both alphabetic characters and digits, but not special characters. Normally, you use strings of alphanumeric characters to represent terms, and special characters as:

- Arithmetic operators in expressions
- Data or field delimiters
- Indicators to the assembler for specific handling

Whenever a lowercase letter (a through z) is used, the assembler considers it to be identical to the corresponding uppercase character (A through Z), except when it is used within a character string enclosed in apostrophes, or within the positional and keyword operands of macro instructions.

Compatibility with Earlier Assemblers: You can specify the COMPAT(MACROCASE) assembler option to instruct the assembler to maintain uppercase alphabetic character set compatibility with earlier assemblers for unquoted macro operands. The assembler converts lowercase alphabetic characters (a through z) in unquoted macro operands to uppercase alphabetic characters (A through Z).

Double-byte character set

In addition to the standard EBCDIC set of characters, High Level Assembler accepts double-byte character set (DBCS) data. The double-byte character set consists of the following:

Table 3. Double-byte character set (DBCS)

Character or code	Description
Double-byte space	X'4040'
Double-byte characters	Each double-byte character contains 2 bytes, each of which must be in the range X'41' to X'FE'. The first byte of a double-byte character is known as the *ward* byte. For example, the ward byte for the double-byte representation of EBCDIC characters is X'42'.
Shift codes	Shift-out (SO) - X'0E' Shift-in (SI) - X'0F'

Note:

1. SO and SI delimit DBCS data only when the DBCS assembler option is specified. The DBCS assembler option is described in the section "DBCS" in the *HLASM Programmer's Guide*.
2. When the DBCS assembler option is specified, double-byte characters can be used anywhere that EBCDIC characters enclosed by apostrophes can be used.
3. Regardless of the invocation option, double-byte characters can be used in remarks, comments, and the statements processed by AREAD and REPRO statements.

Examples showing the use of EBCDIC characters and double-byte characters are given in Table 4. For a description of the DBCS notation used in the examples, see "Double-byte character set notation" on page 7.

Table 4. Examples using character set

Characters	Usage	Example	Constituting

Alphanumeric	In ordinary symbols	`Label`
		`FIELD#01`
		`Save_Total`
		`&EASY_TO_READ`
			Terms

	In variable symbols	`&EASY_TO_READ`	
Digits	As decimal self-defining terms	`1` `9`	Terms
Special Characters	As operators		
+	Addition	`NINE+FIVE`	Expressions
-	Subtraction	`NINE-5`	Expressions
*	Multiplication	`9*FIVE`	Expressions
/	Division	`TEN/3`	Expressions
+ or -	(Unary)	`+NINE  -FIVE`	Terms[1]
	As delimiters		
Spaces	Between fields	`LABEL  AR  3,4`	Statement
Comma	Between operands	`OPND1,OPND2`	Operand field
Apostrophes	Enclosing character strings	`'STRING'`	String
	Attribute operator	`L'OPND1`	Term

Table 4. Examples using character set (continued)

Characters	Usage	Example	Constituting
Parentheses	Enclosing subfields or subexpressions	`MOVE MVC TO(80),FROM(A+B*(C-D))`	Statement expression
SO and SI	Enclosing double-byte data	`C'<.A.B.C>abc'` `G'<D1D2D3D4>'`	Mixed string Pure DBCS
	As indicators for		
Ampersand	Variable symbol	`&VAR`	Term
Period	Symbol qualifier	`QUAL.SYMBOL`	Term
	Sequence symbol	`.SEQ`	(label)
	Comment statement in macro definition	`.*THIS IS A COMMENT`	Statement
	Concatenation	`&VAR.A`	Term
	Bit-length specification	`DC CL.7'AB'`	Operand
	Decimal point	`DC F'1.7E4'`	Operand
Asterisk	Location counter reference	`*+72`	Expression
	Comment statement	`*THIS IS A COMMENT`	Operand
Equal sign	Literal reference	`L 6,=F'2'`	Operand
	Keyword	`&KEY=D`	Keyword parameter

Note:
1. If these are passed as macro arguments, they are treated as expressions, not terms. Expressions cannot be substituted into SETA expressions.

Translation table

In addition to the standard EBCDIC set of characters, High Level Assembler can use a user-specified translation table to convert the characters contained in character (C-type) data constants (DCs) and literals. High Level Assembler provides a translation table to convert the EBCDIC character set to the ASCII character set. You can supply a translation table using the TRANSLATE assembler option, described in the section "TRANSLATE" in the *HLASM Programmer's Guide*.

Self-defining Terms: Self-defining terms are not translated when a translation table is used, except for C-type character self-defining terms where the COMPAT(TRANSDT) assembler suboption in effect.

Assembler language coding conventions

Figure 2 shows the standard format used to code an assembler language statement.

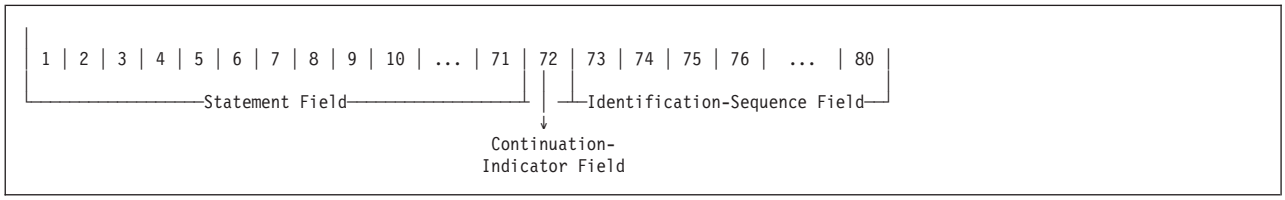


Figure 2. Standard assembler coding format

Field boundaries

Assembler language statements normally occupy one 80-character record, or line. For information about statements that occupy more than 80 characters, see "Continuation lines" on page 13. Each line is divided into three main fields:

- Statement field
- Continuation-indicator field
- Identification-sequence field

If it can be printed, any character coded into any column of a line, or otherwise entered as a position in a source statement, is reproduced in the listing printed by the assembler. Whether it can be printed or not depends on the printer.

Uppercase Printing: Use the FOLD assembler option to instruct the assembler to convert lowercase alphabetic characters to uppercase alphabetic characters before they are printed.

Statement field

The instructions and comment statements must be written in the statement field. The statement field starts in the *begin* column and ends in the *end* column. The continuation-indicator field always lies in the column after the *end* column, unless the *end* column is column 80, in which case no continuation is possible. The identification-sequence field normally lies in the field after the continuation-indicator field. Any continuation lines needed must start in the *continue* column and end in the *end* column.

Blank lines are acceptable. For more information, see "Blank lines" on page 15.

The assembler assumes the following standard values for these columns:

- The *begin* column is column 1
- The *end* column is column 71
- The *continue* column is column 16

These standard values can be changed by using the Input Format Control (ICTL) assembler instruction. The ICTL instruction can, for example, be used to reverse the order of the statement field and the identification-sequence field by changing the standard begin, end, and continue columns. However, all references to the *begin*, *end*, and *continue* columns in this manual refer to the standard values described above.

Continuation-indicator field

The continuation-indicator field occupies the column after the end column. Therefore, the standard position for this field is column 72. A non-space character in this column indicates that the current statement is continued on the next line. This column must be a space character on the last (or only) line of a statement. If this column is not a space, the assembler treats the statement that follows on the next line as a continuation line of the current statement.

If the DBCS assembler option is specified, then:

- When an SI is placed in the end column of a continued line, and an SO is placed in the continue column of the next line, the SI and SO are considered redundant and are removed from the statement before statement analysis is done.
- An extended continuation-indicator provides the ability to extend the end column to the left on a line-by-line basis, so that any alignment of double-byte data in a source statement can be supported.
- The double-byte delimiters SO and SI cannot be used as continuation-indicators.

Identification-sequence field

The identification-sequence field can contain identification characters or sequence numbers or both. If the ISEQ instruction has been specified to check this field, the assembler verifies whether the source statements are in the correct sequence.

The columns that are checked by the ISEQ function are not restricted to columns 73 through 80, or by the boundaries determined by any ICTL instruction. The columns that are specified in the ISEQ instruction can be anywhere on the input statement, including columns that are occupied by the statement field.

Continuation lines

To continue a statement on another line, follow these rules:

1. Enter a non-space character in the continuation-indicator field (column 72). This non-space character must not be part of the statement coding. When more than one continuation line is needed, enter a non-space character in column 72 of each line that is to be continued.
2. Continue the statement on the next line, starting in the continue column (column 16). Columns to the left of the continue column must be spaces. Comment statements can be continued after column 16.

If an operand is continued after column 16, it is taken to be a comment. Also, if the continuation-indicator field is filled in on one line and you try to start a new statement after column 16 on the next line, this statement is taken as a comment belonging to the previous statement.

Specify the FLAG(CONT) assembler option to instruct the assembler to issue warning messages when it suspects a continuation error in a macro call instruction. Refer to the FLAG option description in the section "FLAG" in the *HLASM Programmer's Guide* for details about the situations that might be flagged as continuation errors.

Unless it is one of the statement types listed in "Alternative statement format," nine continuation lines are allowed for a single assembler language statement.

Alternative statement format

The alternative statement format, which allows as many continuation lines as are needed, can be used for the following instructions:

- AGO conditional assembly statement, see "Alternative format for AGO instruction" on page 348
- AIF conditional assembly statement, see "Alternative format for AIF instruction" on page 346
- GBLA, GBLB, and GBLC conditional assembly statements, see "Alternative format for GBLx statements" on page 304
- LCLA, LCLB, and LCLC conditional assembly statements, see "Alternative format for LCLx statements" on page 305
- Macro instruction statement, see "Alternative formats for a macro instruction" on page 260
- Prototype statement of a macro definition, see "Alternative formats for the prototype statement" on page 216
- SETA, SETB, SETAF, SETCF, and SETC conditional assembly statements, see "Alternative statement format" on page 343

Examples of the alternative statement format for each of these instructions are given with the description of the individual instruction.

Continuation of double-byte data

No special considerations apply to continuation:

- Where double-byte data is created by a code-generation program, and
- There is no requirement for double-byte data to be readable on a device capable of presenting DBCS characters

A double-byte character string can be continued at any point, and SO and SI must be balanced within a field, but not within a statement line.

Where double-byte data is created by a workstation that has the capability of presenting DBCS characters, such as the IBM 5550 multistation, or where readability of double-byte data in High Level Assembler source input or listings is required, special features of the High Level Assembler language might be used.

When the DBCS assembler option is specified, High Level Assembler provides the flexibility to cater for any combination of double-byte data and single-byte data. The special features provided are:

- Removal of redundant SI/SO at continuation points. When an SI is placed in the end column of a continued line, and an SO is placed in the continue column of the next line, the SI and SO are considered redundant and are removed from the statement before statement analysis.
- An extended continuation-indicator provides a flexible end column on a line-by-line basis to support any alignment of double-byte data in a source statement. The end column of continued lines can be shifted to the left by extending the continuation-indicator.
- To guard against accidental continuation caused by double-byte data ending in the continuation-indicator column, SO and SI are not continuation indicators. If either falls in the continuation-indicator column, this warning message is issued:

  ```
  ASMA201W SO or SI in continuation column - no continuation
  assumed
  ```

The use of these features is shown in "Examples." The examples below show the use of these features. Refer to "Double-byte character set notation" on page 7 for the notation used in the examples.

Source input considerations:

- Extended continuation-indicators can be used in any source statement, including macro statements and statements included by the COPY instruction. This feature is intended for source lines containing double-byte data, however it becomes available to all lines when the DBCS option is set.
- On a line with a non-space continuation-indicator, the end column is the first column to the left of the continuation-indicator which has a value different from the continuation-indicator.
- When converting existing programs for assembly with the DBCS option, ensure that continuation-indicators are different from the adjacent data in the end column.
- The extended continuation-indicators must not be extended into the continue column, otherwise the extended continuation-indicators are treated as data, and the assembler issues the following error message:

  ```
  ASMA205E Extended continuation column must not extend into continue
  column
  ```

- For SI and SO to be removed at continuation points, the SI must be in the end column, and the SO must be in the continue column of the next line.

Examples:

```
Name     Operation       Operand                              Continuation
                                                                    |
                                                                    ↓
DBCS1    DC              C'<D1D2D3D4D5D6D7D8D9>XXXXXXXXXXXXXXXXXXXX
                         <DaDb>'

DBCS2    DC              C'abcdefghijklmnopqrstuvwxyz0123456789XXXX
                         <DaDb>'

DBCS3    DC              C'abcdefghijklmnopqrstuv<D1D2D3D4D5D6D7>XX
                         <DaDb>'
```

DBCS1

The DBCS1 constant contains 11 double-byte characters bracketed by SO and SI. The SI and SO at the continuation point are not assembled into the operand. The assembled value of DBCS1 is:

```
<D1D2D3D4D5D6D7D8D9DaDb>
```

DBCS2

The DBCS2 constant contains an EBCDIC string which is followed by a double-byte string. Because there is no space for any double-byte data on the first line, the end column is extended three columns to the left and the double-byte data started on the next line. The assembled value of DBCS2 is:

```
abcdefghijklmnopqrstuvwxyz0123456789<DaDb>
```

DBCS3

The DBCS3 constant contains 22 EBCDIC characters followed by nine double-byte characters. Alignment of the double-byte data requires that the end column is extended one column to the left. The SI and SO at the continuation point are not assembled into the operand. The assembled value of DBCS3 is:

```
abcdefghijklmnopqrstuv<D1D2D3D4D5D6D7DaDb>
```

Source listing considerations:

- For source that does not contain substituted variable symbols, the listing exactly reflects the source input.
- Double-byte data input from code-generation programs, that contain no substituted variables, are not readable in the listing if the source input was not displayable on a device capable of presenting DBCS characters.
- Refer to "Listing of generated fields containing double-byte data" on page 218 for details of extended continuation and macro-generated statements.

Blank lines

Blank lines are accepted in source programs. In **open code**, each blank line is treated as equivalent to a SPACE 1 statement. In the body of a **macro definition**, each blank line is treated as equivalent to an ASPACE 1 statement.

Comment statement format

Comment statements are not assembled as part of the object module, but are only printed in the assembly listing. You can write as many comment statements as you need, provided you follow these rules:

- Comment statements require an asterisk in the begin column. Internal macro definition comment statements require a period in the begin column, followed by an asterisk. Internal macro comments are accepted as comment statements in open code.
- Any characters of the EBCDIC character set, or double-byte character set can be used (see "Character set" on page 9).
- Comment statements must lie within the statement field. If the comment extends into the continuation-indicator field, the statement following the comment statement is considered a continuation line of that comment statement.
- Comment statements must not appear between an instruction statement and its continuation lines.

Instruction statement format

Instruction statements must consist of one to four entries in the statement field. They are:

- A name entry
- An operation entry
- An operand entry
- A remarks entry

These entries must be separated by one or more spaces, and must be written in the order stated.

Statement coding rules

The following general rules apply to the coding of an instruction statement:

- The entries must be written in the following order: name, operation, operand, and remarks.
- The entries must be contained in the begin column (1) through the end column (71) of the first line and, if needed, in the continue column (16) through the end column (71) of any continuation lines.
- The entries must be separated from each other by one or more spaces.
- If used, a name entry must start in the begin column.

- The name and operation entries, each followed by at least one space, must be contained in the first line of an instruction statement.
- The operation entry must begin at least one column to the right of the begin column.

Statement example: The following example shows the use of name, operation, operand, and remarks entries. The symbol `COMP` names a compare instruction, the operation entry (`CR`) is the mnemonic operation code for a register-to-register compare operation, and the two operands (`5,6`) designate the two general registers whose contents are to be compared. The remarks entry reminds readers that this instruction compares `NEW SUM` to `OLD`.

```
COMP     CR              5,6                NEW SUM TO OLD
```

Descriptions of the name, operation, operand, and remarks entries follow:

Name entry: The name entry is a symbol created by you to identify an instruction statement. A name entry is generally optional. Except for two instances, the name entry, when provided, must be a valid symbol at assembly time (after substituting variable symbols, if specified). For a discussion of the exceptions to this rule, see "TITLE instruction" on page 190 and "Macro instruction format" on page 259.

The symbol must consist of 63 or fewer alphanumeric characters, the first of which must be alphabetic. It must be entered with the first character appearing in the begin column. If the begin column is a space, the assembler program assumes that no name has been entered. No spaces or double-byte data can appear in the symbol.

Operation entry: The operation entry is the symbolic operation code specifying the machine, assembler, or macro instruction operation. The following rules apply to the operation entry:

- An operation entry is mandatory, and it must appear on the same line as any name entry.
- For machine and assembler instructions, it must be a valid symbol at assembly time (after substitution for variable symbols, if specified), consisting of 63 or fewer alphanumeric characters, the first which must be alphabetic. Most standard symbolic operation codes are five characters or fewer. For a description of machine instructions, refer to the *z/Architecture Principles of Operation* information. For a summary of assembler instructions, see Appendix A, "Assembler instructions," on page 355.

  The standard set of codes can be changed by OPSYN instructions (see "OPSYN instruction" on page 175).
- For macro instructions, the operation entry can be any valid symbol.
- An operation entry cannot be continued on the next statement.

Operand entries: Operand entries contain zero or more operands that identify and describe data to be acted upon by the instruction, by indicating such information as storage locations, masks, storage area lengths, or types of data. The following rules apply to operands:

- One or more operands are typically required, depending on the instruction.
- Operands must be separated by commas. No spaces are allowed between the operands and the commas that separate them.
- A space normally indicates the end of the operand entry, unless the operand is in apostrophes. This applies to machine, assembler, and macro instructions.
- A space does not end the operand in some types of SET statement. Spaces that do not end operands are discussed further at:
  - "Arithmetic (SETA) expressions" on page 311
  - "Logical (SETB) expressions" on page 323
  - "Character (SETC) expressions" on page 328

  There are two examples of operands containing spaces in Figure 6 on page 22; the last box in Row 3, and the middle box in Row 4.

- The alternative statement format uses slightly different rules. For more information, see "Alternative formats for a macro instruction" on page 260.

The following instruction is correctly coded:

```
LA              R1,4+5              No space
```

The following instruction appears to be the same, but is not:

```
LA              R1,4 + 5            Spaces included
```

In this example, the embedded space means that the operand finishes after "4". There is no assembler error, but the result is an `LA R1,4`, which is possibly not what you intended.

A space inside unquoted parentheses is an error, and leads to a diagnostic. The following instruction is correctly coded:

```
DC              CL(L'STRLEN)' '     Space within quotes
```

The following instruction, with an extra space, is not correct:

```
DC              CL(L'STRLEN )' '    Space not within quotes
```

The following example shows a space enclosed in quotes, as part of a string. This space is properly accounted for:

```
MVC             AREA1,=C'This Area' Space inside quotes
```

In quotes, spaces and parentheses can occur in any quantity and in any order:

```
LA              R1,=C'This is OK (isn''t it)'
```

Remarks entries: Remarks are used to describe the current instruction. The following rules apply to remarks:

- Remarks are optional.
- They can contain any character from the EBCDIC character set, or the double-byte characters set.
- They can follow any operand entry.
- In statements in which an optional operand entry is omitted, but you want to code a comment, indicate the absence of the operand by a comma preceded and followed by one or more spaces. For example:

```
END             ,                   End of Program
```

Assembler language structure

This section describes the structure of the assembler language, including the statements that are allowed in the language, and the elements that make up those statements.

"Statement coding rules" on page 15 describes the composition of an assembler language source statement.

The figures in this section show the overall structure of the statements that represent the assembler language instructions, and are not specifications for these instructions. The individual instructions, their purposes, and their specifications are described in other sections of this manual.

Model statements, used to generate assembler language statements, are described in Chapter 7, "How to specify macro definitions," on page 213.

The remarks entry in a source statement is not processed by the assembler, but it is printed in the assembler listing. For this reason, it is only shown in the overview of the assembler language structure in Figure 3 on page 19, and not in the other figures.

The machine instruction statements are described in Figure 4 on page 20, discussed in Chapter 4, "Machine instruction statements," on page 65, and summarized in the *z/Architecture Principles of Operation* information.

Assembler instruction statements are described in Figure 5 on page 21, discussed in Chapter 3, "Program structures and addressing," on page 43 and Chapter 5, "Assembler instruction statements," on page 83, and are summarized in Appendix A, "Assembler instructions," on page 355.

Conditional assembly instruction statements and the macro processing statements (MACRO, MEND, MEXIT, MNOTE, AREAD, ASPACE, and AEJECT) are described in Figure 6 on page 22. The conditional assembly instructions are discussed in Chapter 9, "How to write conditional assembly instructions," on page 279, and macro processing instructions in Chapter 7, "How to specify macro definitions," on page 213. Both types are summarized in Appendix A, "Assembler instructions," on page 355.

Macro instruction statements are described in Figure 7 on page 23, and discussed in Chapter 8, "How to write macro instructions," on page 259.

Overview of assembler language structure

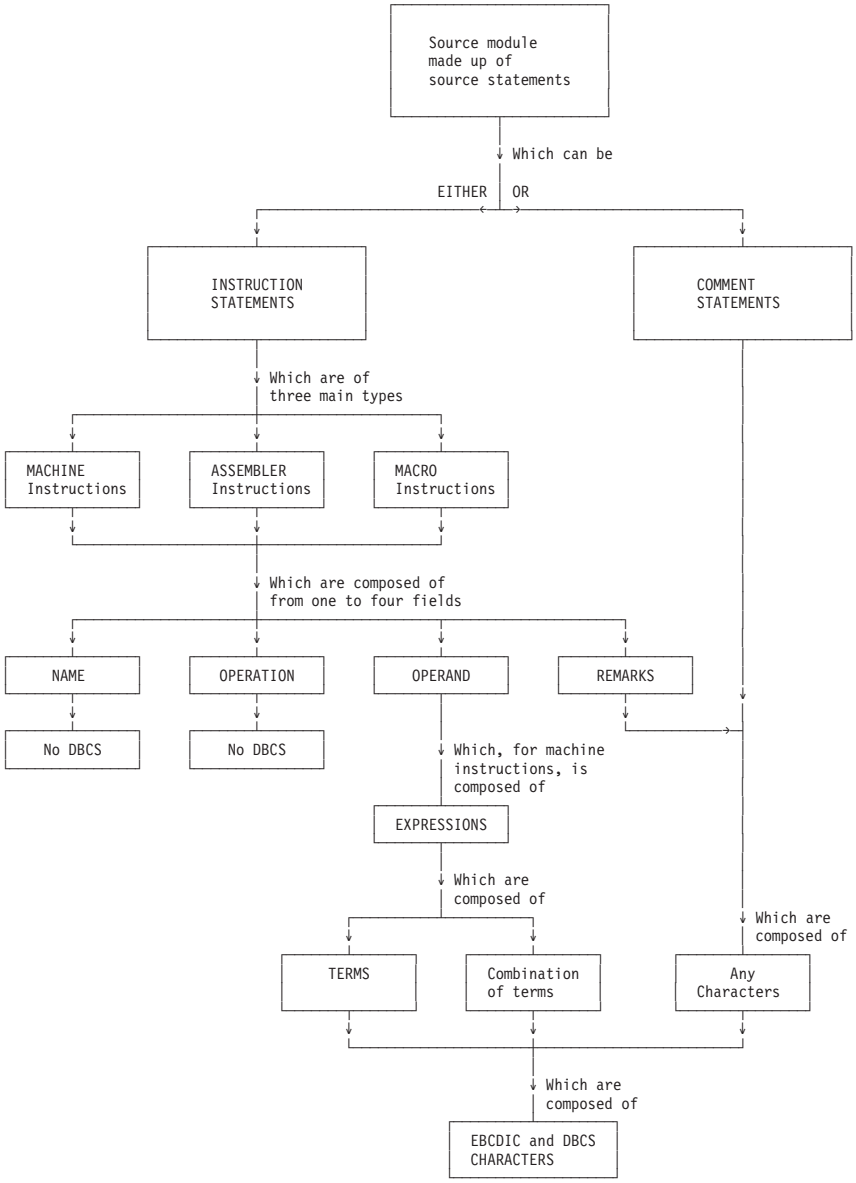
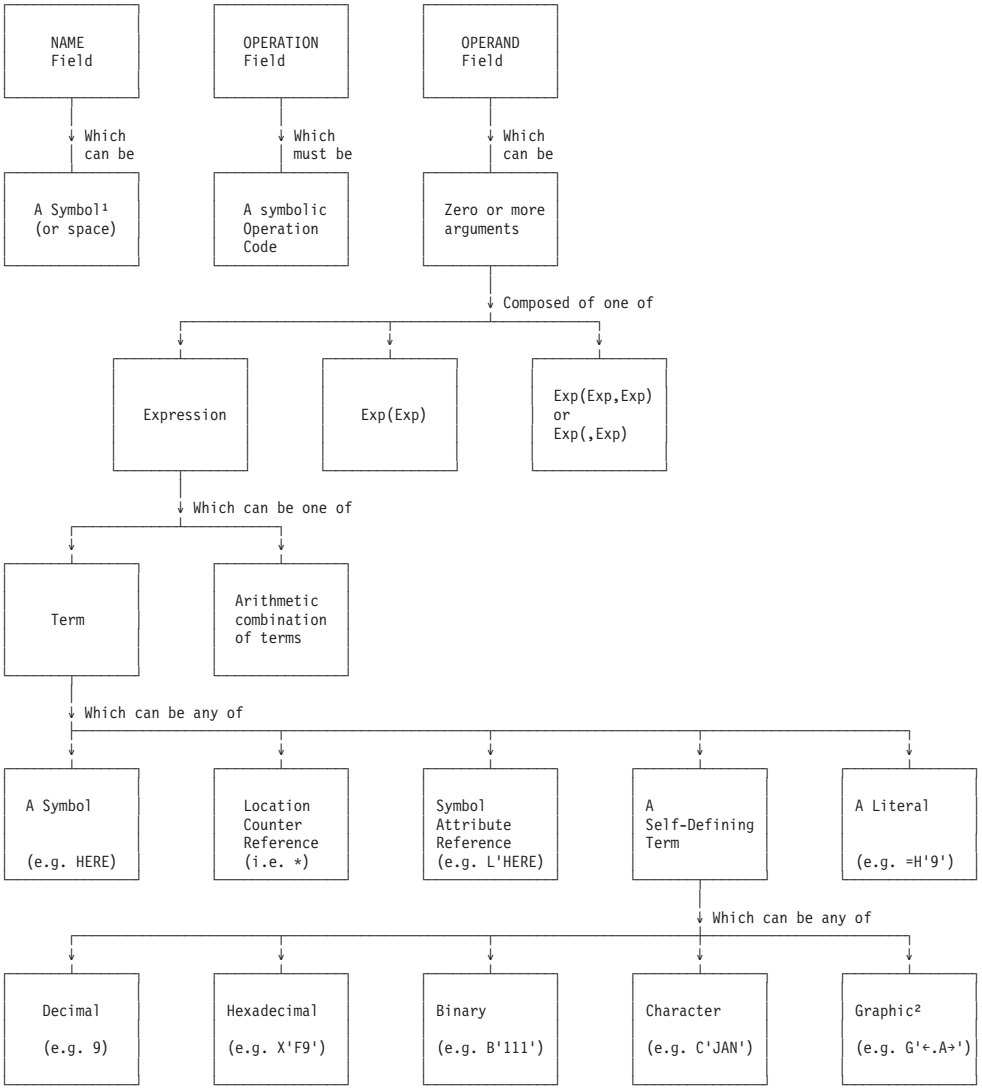


Figure 3. Overview of assembler language structure

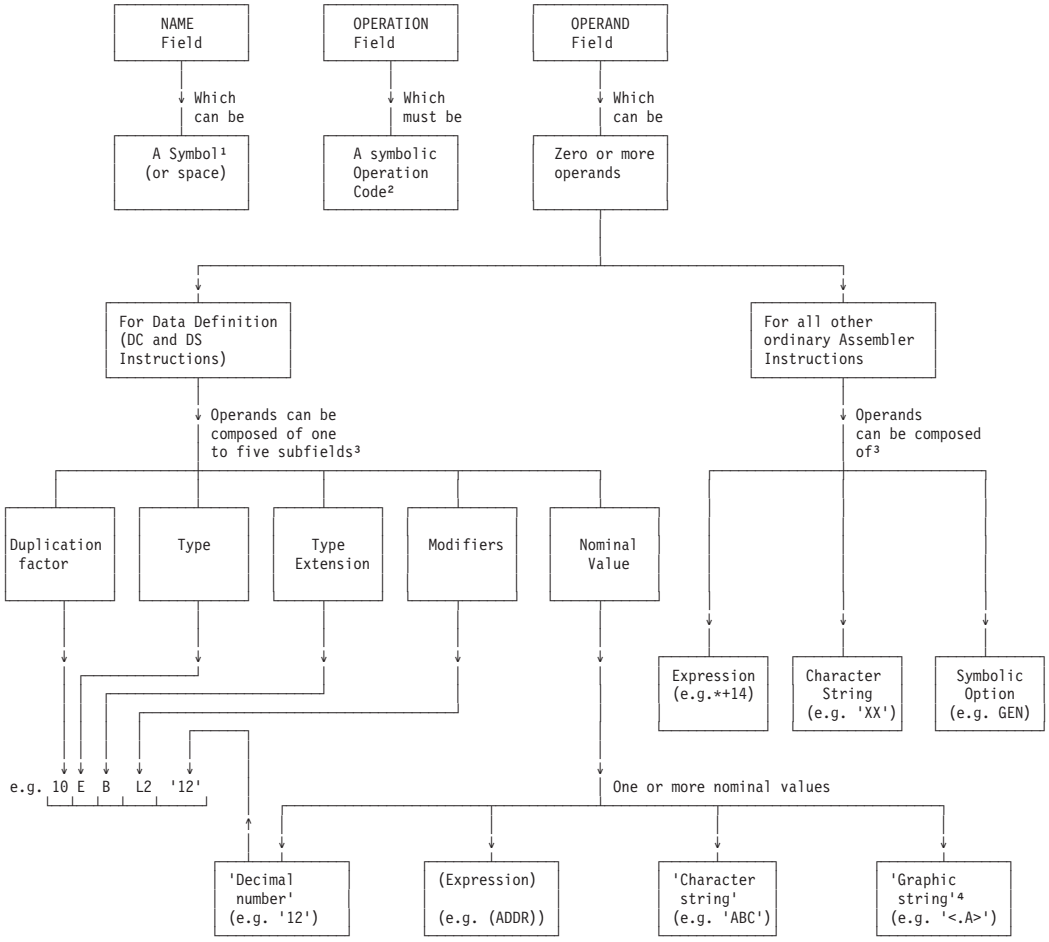
Machine instructions



¹ Can be an ordinary symbol, a variable symbol, or a sequence symbol
² With DBCS option only

Figure 4. Machine instructions

Assembler instructions



¹ Can be an ordinary symbol, a variable symbol, or a sequence symbol
² Includes symbolic operation codes of macro definitions
³ Discussed more fully where individual instructions are described
⁴ With DBCS option only

Figure 5. Ordinary assembler instruction statements

Conditional assembly instructions

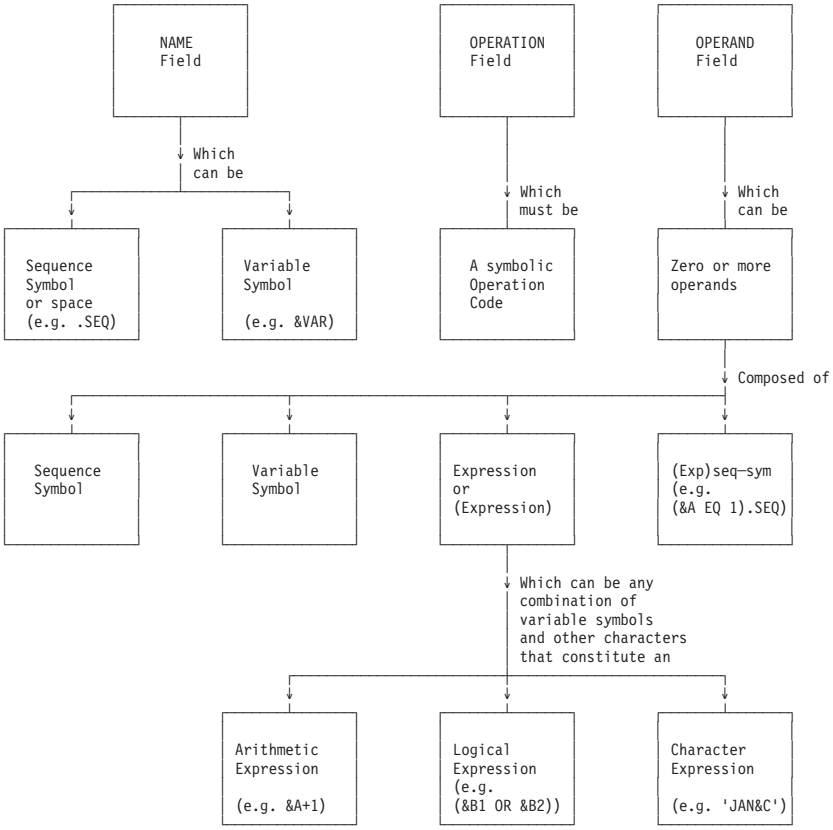


Figure 6. Conditional assembly instructions

Macro instruction statements are described in Figure 7 on page 23.

Macro instructions

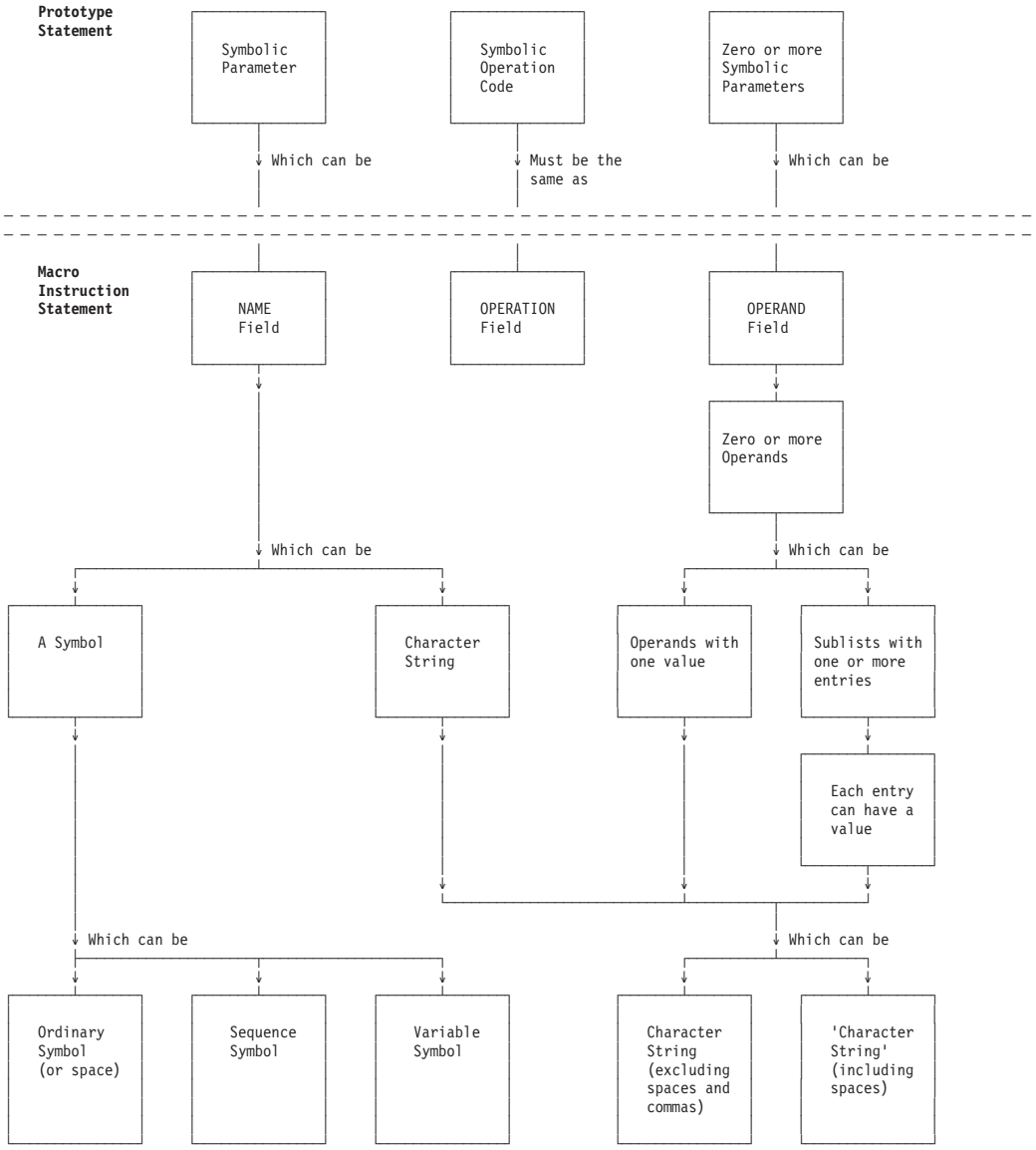


Figure 7. Macro instructions

Mnemonic tags

With mnemonic tagging, you can add a ":ASM" or ":MAC" suffix to an operation code. The mnemonic directs the assembler in this way:

:ASM The assembler searches for machine or assembler instructions only. Macros of the same name are ignored. If the operation code is not found, then the search ends.

:MAC The assembler searches for macro instructions only. Machine and assembler instructions are ignored. If the entry is not found in the current table, then a search for a definition of opcode is done on SYSLIB (the normal search).

For example, say there is a machine code entry named AR. Then `AR` finds that entry (no change). `AR:ASM` is the same; the assembler looks for a machine or assembler instruction. With `AR:MAC`, the assembler looks

for a macro named AR. If it is not found in the internal table then the assembler searches on SYSLIB. Assuming AR is found, the assembler adds a macro entry for AR to the table, and this entry is used for this instruction.

Note: Library macros are added after any existing entry of the same name, while an inline macro is added before any existing entry. This is done to preserve the current behavior when mnemonic tags are not used.

Terms, literals, and expressions

The most basic element of the assembler language is the *term*. Terms can be used alone, or in combination with other terms in expressions. This section describes the different types of terms used in the assembler language, and how they can be used.

Terms

A term is the smallest element of the assembler language that represents a distinct and separate value. It can, therefore, be used alone or in combination with other terms to form expressions. Terms are classified as absolute or relocatable, depending on the effect of program relocation upon them. *Program relocation* is the loading of the object program into storage locations other than those originally assigned by the assembler. Terms have absolute or relocatable values that are assigned by the assembler or that are inherent in the terms themselves.

A term is absolute if its value does not change upon program relocation. A term is relocatable if its value changes by *n* if the origin of the control section in which it appears is relocated by *n* bytes.

Terms in parentheses

Terms in parentheses are reduced to a single value; thus the terms in parentheses, in effect, become a single term.

You can use arithmetically combined terms, enclosed in parentheses, in combination with terms outside the parentheses, as follows:

```
14+BETA-(GAMMA-LAMBDA)
```

When the assembler encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value, which can be absolute or relocatable, depending on the combination of terms. This value is then used in reducing the rest of the combination to another single value.

You can include terms in parentheses within a set of terms in parentheses:

```
A+B-(C+D-(E+F)+10)
```

The innermost set of terms in parentheses is evaluated first. Any number of levels of parentheses are allowed. A *level of parentheses* is a left parenthesis and its corresponding right parenthesis. An arithmetic combination of terms is evaluated as described in "Expressions" on page 38. Table 5 summarizes the various types of terms, and gives a reference to the page number where the term is discussed and the rules for using it are described.

Table 5. Summary of terms

Terms	Term can be absolute	Term can be relocatable	Value is assigned by assembler	Value is inherent in term	Page reference
Symbols	X	X	X		25
Literals	X	X	X		35
Self-defining terms	X			X	29

Table 5. Summary of terms (continued)

Terms	Term can be absolute	Term can be relocatable	Value is assigned by assembler	Value is inherent in term	Page reference
Location counter reference		X	X		32
Symbol length attribute	X		X		33
Other data attributes[1]	X		X		35

Notes:
1. Other valid data attributes are scale and integer.

For more information about absolute and relocatable expressions, see "Absolute and relocatable expressions" on page 41.

Symbols

You can use a symbol to represent storage locations or arbitrary values. If you write a symbol in the name field of an instruction, you can then specify this symbol in the operands of other instructions and thus refer to the former instruction symbolically. This symbol represents a relocatable address.

You can also assign an absolute value to a symbol by coding it in the name field of an EQU instruction with an operand whose value is absolute. This lets you use this symbol in instruction operands to represent:

- Registers
- Displacements in explicit addresses
- Immediate data
- Lengths
- Implicit addresses with absolute values

For details of these program elements, see "Operand entries" on page 71.

The advantages of symbolic over numeric representation are:

- Symbols are easier to remember and use than numeric values, thus reducing programming errors and increasing programming efficiency.
- You can use meaningful symbols to describe the program elements they represent. For example, `INPUT` can name a field that is to contain input data, or `INDEX` can name a register to be used for indexing.
- You can change the value of one symbol that is used in many instructions (through an EQU instruction) more easily than you can change several numeric values in many instructions.
- If the symbols are relocatable, the assembler can calculate displacements and assign base registers for you.
- Symbols are entered into a cross reference table that is printed in the *Ordinary Symbol and Literal Cross Reference* section of the assembler listing. The cross reference helps you find a symbol in the *source and object* section of the listing because it shows:
  - The number of the statement that defines the symbol. A symbol is defined when it appears in the name entry of a statement.
  - The number of all the statements in which the symbol is used as an operand.

Symbol table: When the assembler processes your source statements for the first time, it assigns an absolute or relocatable value to every symbol that appears in the name field of an instruction. The assembler enters this value, which normally reflects the setting of the location counter, into the symbol table. It also enters the attributes associated with the data represented by the symbol. The values of the symbol and its attributes are available later when the assembler finds this symbol or attribute reference

used as a term in an operand or expression. See ""Symbol length attribute reference" on page 33" and ""Self-defining terms" on page 29" in this chapter for more details. The three types of symbols recognized by the assembler are:

- Ordinary symbols
- Variable symbols
- Sequence symbols

Ordinary symbols: Ordinary symbols can be used in the name and operand fields of machine and assembler instruction statements. There are two types of ordinary symbol, internal and external. Code them to conform to these rules:

- The symbol must not consist of more than 63 alphanumeric characters. The first character must be an alphabetic character. An *alphabetic character* is a letter from A through Z, or from a through z, or $, _, #, or @. The other characters in the symbol can be alphabetic characters, digits, or a combination of the two.
  - The assembler does not distinguish between upper-case and lower-case letters used in symbols.
  - If the GOFF option is not specified, external symbols must not consist of more than eight characters.
  - No other special characters can be included in an ordinary symbol.
  - No spaces are allowed in an ordinary symbol.
  - No double-byte data is allowed in an ordinary symbol.

External symbols are placed in the External Symbol Dictionary of the object module, where they are available to link editors and binders for linking with other separately translated programs. Internal symbols are normally discarded at the end of the assembly, but might be placed in the SYSADATA file (see "Input and output files" in the *HLASM Programmer's Guide*) for use by other programs such as debuggers.

In the following sections, the term *symbol* refers to the ordinary symbol.

The following examples are valid ordinary symbols:

```
ORDSYM#435A         HERE          $OPEN
K4                  #0123         X
B49467LITTLENAIL    @33           _TOTAL_SAVED
```

Variable symbols: Variable symbols must begin with an & followed by an alphabetic character and, optionally, up to 61 alphanumeric characters. Variable symbols can be used in macro processing and conditional assembly instructions, and to provide substitution in machine and assembler instructions. They allow different values to be assigned to one symbol. A complete discussion of variable symbols appears in Chapter 7, "How to specify macro definitions," on page 213.

The following examples are valid variable symbols:

```
&VARYINGSYMABC      &@ME
&F346944            &A
&EASY_TO_READ
```

System variable symbol prefix: Do not begin a variable symbol with the characters &SYS, as these characters are used to prefix System Variable Symbols. See "System variable symbols" on page 229 for a list of the System Variable Symbols provided with High Level Assembler.

Sequence symbols: Sequence symbols consist of a period (.) followed by an alphabetic character, and up to 61 additional alphanumeric characters. Sequence symbols can be used in macro processing and conditional assembly instructions. They indicate the position of statements within the source program or macro definition. They are used in AIF and AGO statements to vary the sequence in which statements are processed by the assembler program. (See the complete discussion in Chapter 9, "How to write conditional assembly instructions," on page 279.)

The following examples are valid sequence symbols:

```
.BLABEL04           .#359
.BRANCHTOMEFIRST    .A
```

Symbol definition: An ordinary symbol is defined in:

- The name entry in a machine or assembler instruction of the assembler language
- One of the operands of an EXTRN or WXTRN instruction

Ordinary symbols can also be defined in instructions generated from model statements during conditional assembly.

In Figure 8 on page 28, the assembler assigns a value to the ordinary symbol in the name entry according to the following rules:

1. The symbol is assigned a relocatable address value if the first byte of the storage field contains one of the following:
   - Any machine or assembler instruction, except the EQU or OPSYN instruction (see **1** in Figure 8 on page 28)
   - A storage area defined by the DS instruction (see **2** in Figure 8 on page 28)
   - Any constant defined by the DC instruction (see **3** in Figure 8 on page 28)
   - A channel command word defined by the CCW, CCW0, or CCW1 instruction

   The address value assigned is relocatable, because the object code assembled from these items is relocatable. The relocatability of addresses is described in "Addresses" on page 73.
2. The symbol is assigned the value of the first or only expression specified in the operand of an EQU instruction. This expression can have a *relocatable* (see **4** in Figure 8 on page 28) or *absolute* (see **5** in Figure 8 on page 28) value, which is then assigned to the ordinary symbol.

   The value of an ordinary absolute symbol must lie in the range -2^{31} through $+2^{31}-1$. Relocatable symbols have unsigned address values in the range 0 - $2^{24}-1$, or 0 - $2^{31}-1$ if the GOFF option is specified.

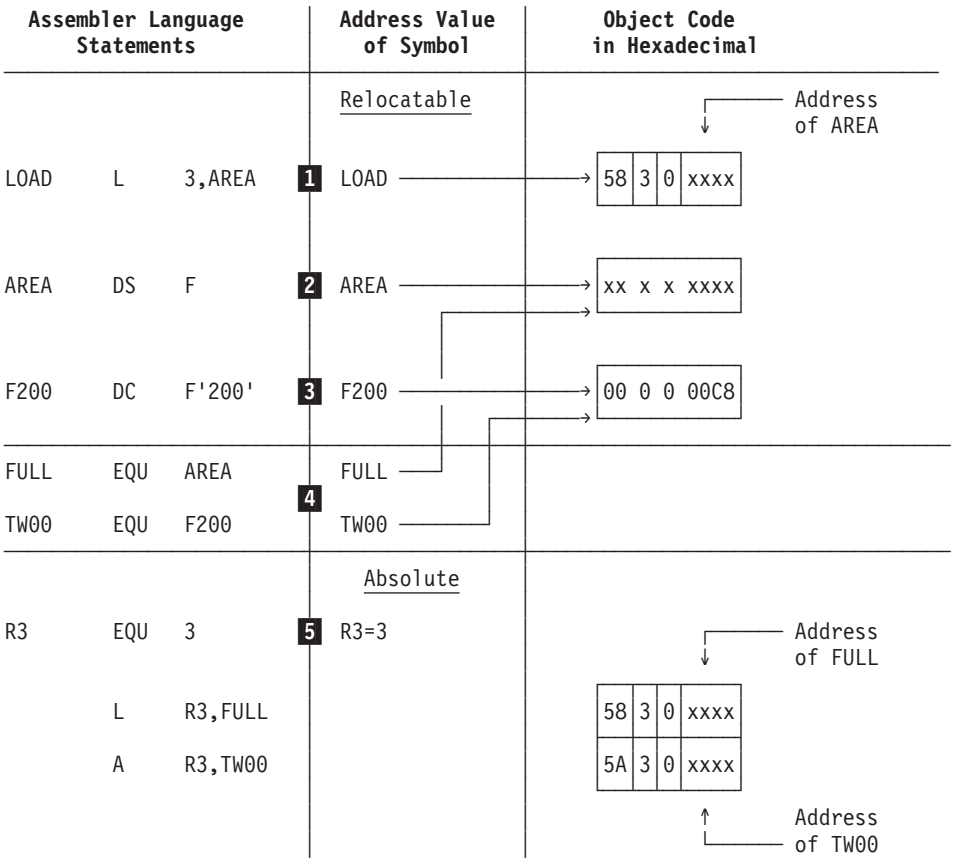


Figure 8. Transition from assembler language statement to object code

Restrictions on symbols: A symbol must be defined only once in a source module with one or more control sections, with the following exceptions:

- The symbol in the name field of a CSECT, RSECT, DSECT, or COM instruction can be the same as the name of previous CSECT, RSECT, DSECT, or COM instruction. It identifies the resumption of the control section specified by the name field.
- **z/VM and z/OS**
  The symbol in the name field of a CATTR instruction can be the same as the name of a previous CATTR instruction. It identifies the resumption of the class specified by the name field.
- The symbol in the name field of a LOCTR instruction can be the same as the name of a previous START, CSECT, RSECT, DSECT, COM, or LOCTR instruction. It identifies the resumption of the location counter specified by the name field.
- The symbol in the name field of a labeled USING instruction can be the same as the name of a previous labeled USING instruction. It identifies the termination of the domain of the previous labeled USING instruction with the specified name.
- A symbol can be used as an operand of a V-type constant and as an ordinary label, without duplication, because the operand of a V-type constant does not define the symbol in the symbol table.

An ordinary symbol is not defined when:

- It is used in the name field of an OPSYN or TITLE instruction. It can, therefore, be used in the name field of any other statement in a source module.

- It is used as the operand of a V-type address constant.
- It is only used in the name field of a macro instruction and does not appear in the name field of a macro-generated assembler statement. It can, therefore, be used in the name field of any other statement in a source module.
- It is only used in the name field of an ALIAS instruction and does not appear in one of the following:
  - The name field of a START, CSECT, RSECT, COM, or DXD instruction.
  - The name field of a DSECT instruction and the nominal value of a Q-type address constant.
  - The operand of an ENTRY, EXTRN, or WXTRN instruction.

Previously defined symbols: An ordinary symbol is *previously defined* if the statement that defines it is processed before the statement in which the symbol appears in an operand.

An ordinary symbol must be defined by the time the END statement is reached, however, it need not be previously defined when it is used as follows:

- In operand expressions of certain instructions such as CNOP instructions and some ORG instructions
- In modifier expressions of DC, DS, and DXD instructions
- In the first operand of an EQU instruction
- In Q-type constants

When using the forward-reference capability of the assembler, avoid the following types of errors:

- Circular definition of symbols, such as:

```
X       EQU             Y
Y       EQU             X
```

- Circular location-counter dependency, as in this example:

```
A       DS              (B-A)C
B       LR              1,2
```

  The first statement in this example cannot be resolved because the value of the duplication factor is dependent on the location of `B`, which is, in turn, dependent upon the length and duplication factor of `A`.

Literals can contain symbolic expressions in modifiers, but any ordinary symbols used must have been previously defined.

Self-defining terms

A self-defining term lets you specify a value explicitly. With self-defining terms, you can also specify decimal, binary, hexadecimal, or character data. If the DBCS assembler option is specified, you can specify a graphic self-defining term that contains pure double-byte data, or include double-byte data in character self-defining terms. These terms have absolute values and can be used as absolute terms in expressions to represent bit configurations, absolute addresses, displacements, length or other modifiers, or duplication factors.

Using self-defining terms: Self-defining terms represent machine language *binary values* and are absolute terms. Their values do not change upon program relocation. Here are some examples of self-defining terms and the binary values they represent:

Self-Defining Term	Decimal Value	Binary Value
`15`	15	1111
`241`	241	1111 0001
`B'1111'`	15	1111
`B'11110001'`	241	1111 0001
`B'100000001'`	257	0001 0000 0001

Self-Defining Term	Decimal Value	Binary Value
X'F'	15	1111
X'F1'	241	1111 0001
X'101'	257	0001 0000 0001
C'1'	241	1111 0001
C'A'	193	1100 0001
C'AB'	49,602	1100 0001 1100 0010
G'<.A>'	17,089	0100 0010 1100 0001

The assembler carries the values represented by self-defining terms to 4 bytes or 32 bits, the high-order bit of which is the sign bit. (A '1' in the sign bit indicates a negative value; a '0' indicates a positive value.)

The use of a self-defining term is distinct from the use of data constants or literals. When you use a self-defining term in a machine instruction statement, its value is used to determine the binary value that is assembled into the instruction. When a data constant is referred to or a literal is specified in the operand of an instruction, its address is assembled into the instruction. Self-defining terms are always right-aligned. Truncation or padding with zeros, if necessary, occurs on the left.

Decimal self-defining term: A decimal self-defining term is an unsigned decimal number written as a sequence of decimal digits. High-order zeros can be used (for example, 007). Limitations on the value of the term depend on its use. For example, a decimal term that designates a general register must have a value 0 - 15. A decimal term that represents an address must not exceed the size of storage. In any case, a decimal term must not exceed 2,147,483,647 (2^{31}-1). A decimal self-defining term is assembled as its binary equivalent. Some examples of decimal self-defining terms are: 8, 147, 4092, and 00021.

Hexadecimal self-defining term: A hexadecimal self-defining term consists of hexadecimal digits enclosed in apostrophes and preceded by the letter X; for example, X'C49' and X'00FF00FF00'.

Each hexadecimal digit is assembled as its 4 bit binary equivalent. Thus, a hexadecimal term used to represent an 8 bit mask consists of two hexadecimal digits. The maximum value of a hexadecimal term is X'FFFFFFFF'; this allows a range of values -2,147,483,648 - 2,147,483,647.

The hexadecimal digits and their bit patterns are as follows:

```
0 - 0000   4 - 0100   8 - 1000   C - 1100
1 - 0001   5 - 0101   9 - 1001   D - 1101
2 - 0010   6 - 0110   A - 1010   E - 1110
3 - 0011   7 - 0111   B - 1011   F - 1111
```

When used as an absolute term in an expression, a hexadecimal self-defining term has a negative value if the high-order bit is 1.

Binary self-defining term: A binary self-defining term is written as an unsigned sequence of 1s and 0s enclosed in apostrophes and preceded by the letter B; for example, B'10001101'. A binary term can have up to 32 bits, not counting leading zero bits. This allows a range of values from -2,147,483,648 through 2,147,483,647.

When used as an absolute term in an expression, a binary self-defining term has a negative value if the term is 32 bits long and the high-order bit is 1.

Binary representation is used primarily in designating bit patterns of masks or in logical operations.

The following shows a binary term used as a mask in a Test Under Mask (TM) instruction. The contents of GAMMA are to be tested, bit by bit, against the pattern of bits represented by the binary term.

```
ALPHA    TM             GAMMA,B'10101101'
```

Character self-defining term: A character self-defining term consists of 1-to-4 characters enclosed in apostrophes, and must be preceded by the letter C. All letters, decimal digits, and special characters can be used in a character self-defining term. In addition, any of the remaining EBCDIC characters can be designated in a character self-defining term. Examples of character self-defining terms are:

```
C'/'
C' ' (space)
C'ABC'
C'13'
```

Because of the use of apostrophes in the assembler language and ampersands in the macro language as syntactic characters, the following rule must be observed when using these characters in a character self-defining term:

> For each apostrophe or ampersand you want in a character self-defining term, two apostrophes or ampersands must be written. For example, the character value A'# is written as 'A''#', while a single apostrophe followed by a space and another apostrophe is written as ''' '''.

For C-type character self-defining terms, each character in the character sequence is assembled as its 8 bit code equivalent (see Appendix D, "Standard character set code table," on page 375). The two apostrophes or ampersands that must be used to represent an apostrophe or ampersand within the character sequence are assembled as an apostrophe or ampersand. Double-byte data can appear in a character self-defining term, if the DBCS assembler option is specified. The assembled value includes the SO and SI delimiters. Hence a character self-defining term containing double-byte data is limited to one double-byte character delimited by SO and SI. For example, C'<.A>'.

Since the SO and SI are stored, the null double-byte character string, C'<>', is also a valid character self-defining term.

Note: The assembler does not support character self-defining terms of the form CU'x' because self-defining terms are required by definition of the Assembler Language to have fixed values.

There are many EBCDIC code pages; some characters have different encodings in different code pages. To be sure that your character constants and self-defining terms have the representations and value, use just these 82 invariant characters:

- space
- decimal digits
- upper-case and lower-case letters A through Z
- these special characters:

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

These characters have the same binary representation across all single-byte EBCDIC code pages. If you use any other character, the Assembler uses its bit pattern as its value, so it might not display as the same character in environments where a different code page is used by default.

Graphic self-defining term: If the DBCS assembler option is specified, a graphic self-defining term can be specified. A graphic self-defining term consists of 1 or 2 double-byte characters delimited by SO and SI, enclosed in apostrophes and preceded by the letter G. Any valid double-byte characters can be used. Examples of graphic self-defining terms are:

```
G'<.A>'
G'<.A.B>'
G'<Da>'
G'<.A><.B>'
```

The SO and SI are not represented in the assembled value of the self-defining term, hence the assembled value is pure double-byte data. A redundant SI/SO pair can be present between two double-byte characters, as shown in the last of the above examples. However, if SO and SI are used without an intervening double-byte character, this error is issued:

```
ASMA148E Self-defining term lacks ending quote or has bad character
```

Location counter

The assembler maintains a location counter to assign storage addresses to your program statements. It is the assembler's equivalent of the execution-time instruction counter in the computer. You can refer to the current value of the location counter at any place in a source module by specifying an asterisk (*) as a term in an operand.

As the instructions and constants of a source module are being assembled, the location counter has a value that indicates a location in the program. The assembler increments the location counter according to the following:

1. After an instruction or constant has been assembled, the location counter indicates the *next available location*.
2. Before assembling the current instruction or constant, the assembler checks the boundary alignment required for it and *adjusts the location counter*, if necessary, to the correct boundary.
3. While the instruction or constant is being assembled, the location counter value does not change. It indicates the location of the current data after boundary alignment and is the *value assigned to the symbol*, if present, in the name field of the statement.
4. After assembling the instruction or constant, the assembler increments the location counter by the length of the assembled data to *indicate the next available location*.

Here is an example of the application of these rules:

```
Location in              Source
Hexadecimal              Statements

000004         DONE      DC  CL3'ABC'
000007         BEFORE    EQU *
000008         DURING    DC  F'200'
00000C         AFTER     EQU *
000010         NEXT      DS  D
```

You can specify multiple location counters for each control section in a source module; for more details about the location counter setting in control sections, see “Location counter setting” on page 52.

Maximum location counter value: The assembler carries internal location counter values as 4 byte (31 bit unsigned) values. When you specify the NOGOFF assembler option, the assembler uses only the low-order 3 bytes for the location counter, and prints only the low-order 3 bytes in the assembly source and object code listing if the LIST(121) option is active. All 4 bytes are displayed if the LIST(133) option is active. In this case the maximum valid location counter value is 2^{24}-1.

z/VM and z/OS

When you specify the GOFF assembler option, the assembler requires the LIST(133) option, and uses the entire 4 byte value for the location counter and prints the 4 byte value in the assembly listings. In this case the maximum valid location counter value is 2^{31}-1.

If the location counter exceeds its valid maximum value the assembler issues error message

```
ASMA039S Location counter error
```

Controlling the location counter value: You can control the setting of the location counter in a particular control section by using the START or ORG instruction, described in Chapter 3, “Program

structures and addressing," on page 43 and Chapter 5, "Assembler instruction statements," on page 83. The counter affected by either of these assembler instructions is the counter for the control section in which they appear.

Location counter reference: You can refer to the current value of the location counter at any place in a program by using an asterisk as a term in an operand. The asterisk is a relocatable term, specified according to the following rules:

- The asterisk can be specified only in the operands of:
  - Machine instructions
  - DC and DS instructions
  - EQU, ORG, and USING instructions
- It can also be specified in *literal constants*. See "Literals" on page 35. For example:

  ```
  THERE    L   =3A(*)
  ```

  generates three identical address constants, each with value A(THERE).

  The value of the location counter reference (*) is the same as the value of the symbol THERE, the current value of the location counter of the control section in which the asterisk (*) is specified as a term. The asterisk has the same value as the *address of the first byte of the instruction* in which it appears. For example:

  ```
  HERE     B   *+8
  ```

  where the value of `*` is the value of `HERE`.

For the value of the asterisk in address constants with duplication factors, see "Subfield 1: Duplication Factor" on page 114 of "DC instruction" on page 109, and "Address constants—A and Y" on page 133. For a discussion of location counter references in literals, see "Subfield 1: Duplication Factor" on page 114.

Symbol length attribute reference

The length attribute of a symbol can be used as a term. Reference to the attribute is made by coding `L'` followed by the symbol, as in:

```
L'BETA
```

The length attribute of `BETA` is substituted for the term. When you specify a symbol length attribute reference, you obtain the length of the instruction or data named by a symbol. You can use this reference as a term in instruction operands to:

- Specify assembler-determined storage area lengths
- Cause the assembler to compute length specifications for you
- Build expressions to be evaluated by the assembler

The symbol length attribute reference must be specified according to the following rules:

- The format must be `L'` immediately followed by a valid symbol (`L'SYMBOL`), an expression (`L'SYMBOL+SYMBOL2-SYMBOL7`), or the location counter reference (`L'*`). If the operand is an expression, the length attribute of its leftmost term is used.
- Symbols must be defined in the same source module in which the symbol length attribute reference is specified.
- The symbol length attribute reference can be used in the operand of any instruction that requires an absolute term. However, it cannot be used in the form `L'*` in any instruction or expression that requires a previously defined symbol.

The value of the length attribute is normally the length in bytes of the storage area required by an instruction, constant, or field represented by a symbol. The assembler stores the value of the length attribute in the symbol table along with the address value assigned to the symbol.

When the assembler encounters a symbol length attribute reference, it substitutes the value of the attribute from the symbol table entry for the symbol specified.

The assembler assigns the length attribute values to symbols in the name field of instructions as follows:

- For machine instructions (see **1** in Table 6), it assigns 2, 4, or 6, depending on the format of the instruction.
- For the DC and DS instructions (see **2** in Table 6), it assigns either the implicitly or explicitly specified length of the first or only operand. The length attribute is not affected by a duplication factor.
- For the EQU instruction, it assigns the length attribute value of the first or only term (see **3** in Table 6) of the first expression in the first operand, unless a specific length attribute is supplied in a second operand.

  Note the length attribute values of the following terms in an EQU instruction:
  - Self-defining terms (see **4** in Table 6)
  - Location counter reference (see **5** in Table 6)
  - L'* (see **6** in Table 6)

For assembler instructions such as DC, DS, and EQU, the length attribute of the location counter reference (L'* — see **6** in Table 6) is equal to 1. For machine instructions, the length attribute of the location counter reference (L'* — see **7** in Table 6) is equal to the length attribute of the instruction in which the L'* appears.

Table 6. Assignment of length attribute values to symbols in name fields

Source Module				Length Attribute Reference	Value of Symbol Length Attribute At Assembly Time	
MACHA	MVC	TO,FROM		L'MACHA	6	1
MACHB	L	3,ADCON		L'MACHB	4	1
MACHC	LR	3,4		L'MACHC	2	1
TO	DS	CL80		L'TO	80	2
FROM	DS	CL240		L'FROM	240	2
ADCON	DC	A(OTHER)		L'ADCON	4	2
CHAR	DC	C'YUKON'		L'CHAR	5	2
DUPL	DC	3F'200'		L'DUPL	4	2
RELOC1	EQU	TO	3	L'RELOC1	80	
RELOC2	EQU	TO+80	3	L'RELOC2	80	
RELOC3	EQU	TO,44	3	L'RELOC3	44	
ABSOL1	EQU	FROM-TO	3	L'ABSOL1	240	
ABSOL2	EQU	ABSOL1	3	L'ABSOL2	240	
SDT1	EQU	102	3	L'SDT1	1	4
SDT2	EQU	X'FF'+A-B	3	L'SDT2	1	4
SDT3	EQU	C'YUK'		L'SDT3	1	4
ASTERISK	EQU	*+10	3	L'ASTERISK	1	5
LOCTREF	EQU	L'*	3	L'LOCTREF	1	6
LENGTH1	DC	A(L'*)		L'*	1	6
				L'LENGTH1	4	
LENGTH2	MVC	TO(L'*),FROM		L'*	6	7
LENGTH3	MVC	TO(L'TO-20),FROM		L'TO	80	

Note: Instructions that contain length attribute references L'SDT1, L'SDT2, L'SDT3, L'ASTERISK, and L'LOCTREF as shown in this figure might generate ASMA019W.

The following example shows how to use the length attribute to move a character constant into either the high-order or low-order end of a storage field.

```
A1       DS             CL8
B2       DC             CL2'AB'
HIORD    MVC            A1(L'B2),B2
LOORD    MVC            A1+L'A1-L'B2(L'B2),B2
```

A1 names a storage field eight bytes in length and is assigned a length attribute of 8. B2 names a character constant 2 bytes in length and is assigned a length attribute of 2. The statement named HIORD moves the contents of B2 into the first 2 bytes of A1. The term L'B2 in parentheses provides the length specification required by the instruction.

The statement named LOORD moves the contents of B2 into the rightmost 2 bytes of A1. The combination of terms A1+L'A1-L'B2 adds the length of A1 to the beginning address of A1, and subtracts the length of B2 from this value. The result is the address of the seventh byte in field A1. The constant represented by B2 is moved into A1 starting at this address. L'B2 in parentheses provides the length specification in both instructions.

For ease in following the preceding example, the length attributes of A1 and B2 are specified explicitly in the DS and DC statements that define them. However, keep in mind that the L'symbol term makes coding such as this possible in situations where lengths are unknown. For example:

```
C3       DC             C'This is too long a string to be worth counting'
STRING   MVC            BUF(L'C3),C3
```

Other attribute references

Other attributes describe the characteristics and structure of the data you define in a program; for example, the kind of constant you specify or the number of characters you need to represent a value. These other attributes are:

- Count (K')
- Defined (D')
- Integer (I')
- Number (N')
- Operation code (O')
- Scale (S')
- Type (T')

You can refer to the count (K'), defined (D'), number (N'), and operation code (O') attributes only in conditional assembly instructions and expressions. For full details, see “Data attributes” on page 284.

Literals

You can use literals as operands in order to introduce data into your program. The literal is a special type of relocatable term. It behaves like a symbol in that it represents data. However, it is a special kind of term because it also is used to define the constant specified by the literal. This is convenient because:

- The data you enter as numbers for computation, addresses, or messages to be printed is visible in the instruction in which the literal appears.
- You avoid the added effort of defining constants elsewhere in your source module and then using their symbolic names in machine instruction operands.

The assembler assembles the data item specified in a literal into a *literal pool* (See “Literal pool” on page 38). It then assembles the address of this literal data item in the pool into the object code of the instruction that contains the literal specification. Thus, the assembler saves you a programming step by storing your literal data for you. The assembler also organizes literal pools efficiently, so that the literal data is aligned on the correct boundary alignment and occupies a minimum amount of space.

Literals, constants, and self-defining terms

Literals, constants, and self-defining terms differ in three important ways:

- Where you can specify them in machine instructions, that is, whether they represent data or an address of data

- Whether they have relocatable or absolute values
- What is assembled into the object code of the machine instruction in which they appear

Figure 9 shows examples of the differences between literals, constants, and self-defining terms.

---

1. A literal with a relocatable address:

```
          L     3,=F'33'       Register 3 set to 33.  See note 1
          L     3,F33          Register 3 set to 33.  See note 2
          .
          .
          .
F33       DC    F'33'
```

2. A literal with a self-defining term and a symbol with an absolute value

```
          MVC   FLAG,=X'00'    FLAG set to X'00'.  See note 1
          MVI   FLAG,X'00'     FLAG set to X'00'.  See note 3
          MVI   FLAG,ZERO      FLAG set to X'00'.  See note 4
          .
          .
          .
FLAG      DS    X
ZERO      EQU   X'00'
```

3. A symbol having an absolute address value specified by a self-defining term

```
          LA    4,LOCORE       Register 4 set to 1000.  See note 4
          LA    4,1000         Register 4 set to 1000.  See note 3
          .
          .
          .
LOCORE    EQU   1000
```

Notes:

1. A literal both defines data and represents data. The address of the literal is assembled into the object code of the instruction in which it is used. The constant specified by the literal is assembled into the object code, in the literal pool.
2. A constant is represented by a symbol with a relocatable value. The address of a constant is assembled into the object code.
3. A self-defining term has an absolute value. In this example, the absolute value of the self-defining term is assembled into the object code.
4. A symbol with an absolute value does not represent the address of a constant, but represents either immediate data or an absolute address. When a symbol with an absolute value represents immediate data, it is the absolute value that is assembled into the object code.

Figure 9. Differences between literals, constants, and self-defining terms

General rules for using literals

You can specify a literal as either a complete operand in a machine instruction, or as part of an expression in the operand of a machine instruction. A literal can also be specified as the name field on a macro call instruction.

Because literals define *read-only* data, they must not be used in operands that represent the receiving field of an instruction that modifies storage.

The assembler requires a description of the type of literal being specified as well as the literal itself. This descriptive information assists the assembler in assembling the literal correctly. The descriptive portion of the literal must indicate the format of the constant. It can also specify the length of the constant.

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in a single operand of a DC assembler instruction. The only difference is that the literal must start with an equal sign (=), which indicates to the assembler that a literal follows. The length of the literal, including the equal sign, constant type and modifiers, delimiters, and nominal values is limited to a maximum of 256 characters.

A literal can be coded as indicated here:

```
=10XL5'F3'
```

where the subfields are:

```
Duplication factor 10
Type                X
Modifiers          L5
Nominal value      'F3'
```

The following instruction shows one use of a literal:

```
GAMMA    L              10,=F'274'
```

The statement `GAMMA` is a load instruction using a literal as the second operand. When assembled, the second operand of the instruction refers to the relative address at which the value `F'274'` is stored.

You cannot rely on the ordering of literals in the literal pool remaining the same. For this reason, referring to a point that extends beyond the bounds of a literal is flagged with warning message ASMA015W. Here is an example of such a reference:

```
BETA     L              10,=F'274'+4
```

In general, literals can be used wherever a storage address is permitted as an operand, including with an index register in instructions with the RX format. For example:

```
DELTA    LH             5,=H'11,23,39,48,64'(6)
```

is equivalent to:

```
DELTA    LH             5,LENGTHS(6)
         .
         .
         .
LENGTHS  DC             H'11,23,39,48,64'
```

See “DC instruction” on page 109 for a description of how to specify the subfields in a literal.

Literals cannot be used in any assembler instruction where a previously defined symbol is required, but length attribute references to previously defined literals are allowed. Literals are relocatable terms because the address of the literal, rather than the literal-generated constant itself, is assembled in the statement that references a literal. The assembler generates the literals, collects them, and places them in a specific area of storage, as explained under “Literal pool” on page 38. Because the assembler determines the order in which literals are placed in the literal pool, the effect of using two literals as paired relocatable terms (see “Paired relocatable terms” on page 41) is unpredictable.

“Location counter reference” on page 33 describes how you can use the current location counter in a literal.

The rules for determining whether two literals are identical are:

1. A literal which contains a location counter reference is not identical to any other literal.
2. Otherwise, two literals are identical (and are generated only once), if their source forms are identical.

Summary of literal rules:

1. S-type address constants can be used in literals.
2. Location counter references (*) can be used in address constants.
3. When a literal address constant contains *, each use of that literal is assigned a separate location in the literal pool.
4. When a literal address constant contains *, the value used for * is the address of the (single) instruction in which the literal is used.

   When not in a literal, * in an address constant refers to the first byte of the constant.
5. When a literal address constant containing * also has a duplication factor, the value of * does not change for each duplication, but remains equal to the address if the first byte of the instruction in which the literal was used.

   When not in a literal, if an address constant containing * is duplicated, the value of * is updated for each duplication to refer to the address of that duplication.
6. When an S-type address constant is used in a literal, regardless of whether it contains *, the base register that is used to compute the base and displacement that are parts of the S-type address constant is determined by the USING statements that are in effect at the place that the literal is assembled, not the USING statements in effect at the place where the literal is referenced in an instruction. There are two different base-displacement calculations: one in the instruction referring to the S-type address constant, and one in the S-type address constant to determine how to address the object of the constant.

Contrast with immediate data: Do not confuse a literal with the *immediate data*. Immediate data is assembled into the instruction.

Literal pool

The literals processed by the assembler are collected and placed in a special area called the literal pool. You can control the positioning of the literal pool. Unless otherwise specified, the literal pool is placed at the end of the first control section.

You can also specify that multiple literal pools be created. However, the assembler controls the sequence in which literals are ordered within the pool. Further information about positioning literal pools is in "LTORG instruction" on page 171.

Expressions

This section discusses the expressions used in coding operand entries for source statements. You can use an expression to specify:

- An address
- An explicit length
- A modifier
- A duplication factor
- A complete operand

Expressions have absolute and relocatable values. Whether an expression is absolute or relocatable depends on the value of the terms it contains. The assembler evaluates relocatable and absolute expressions at assembly time. Figure 10 on page 39 shows examples of valid expressions.

In addition to expressions used in coding operand entries, there are three types of expression that you can use only in conditional assembly instructions: arithmetic, logical, and character expressions. They are evaluated during conditional assembly. For more information, see "Assigning values to SET symbols" on page 305.

An expression is composed of a single term or an arithmetic combination of terms. The assembler reduces multiterm expressions to single values. Thus, you do not have to compute these values yourself. Here are

examples of valid expressions:

```
*                                   BETA*10
AREA1+X'2D'                         B'101'
*+32                                C'ABC'
N-25                                29
FIELD+332                           L'FIELD
FIELD                               LAMBDA+GAMMA
(EXIT-ENTRY+1)+GO                   TEN/TWO
ALPHA-BETA/(10+AREA*L'FIELD)-100    =F'1234'
=A(100,133,175,221)+8
```

Figure 10. Examples of valid expressions

Rules for coding expressions

The rules for coding an absolute or relocatable expression are:

- Unary (operating on one value) operators and binary (operating on two values) operators are allowed in expressions.
- An expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression.
- An expression must not begin with a binary operator, nor can it contain two binary operators in succession. When + and - are used as prefix operators, then they are unary, and not binary, operators.
- An expression starting with * is interpreted as a location counter reference, and not a multiplication operator.
- An expression must not contain two terms in succession.
- No spaces are allowed between an operator and a term, nor between two successive operators.
- An expression can contain any number of unary and binary operators, and any number of levels of parentheses.
- A single relocatable term is not allowed in a multiply or divide operation. Paired relocatable terms have absolute values and can be multiplied and divided if they are enclosed in parentheses. See "Paired relocatable terms" on page 41.

Figure 11 on page 40 shows the definitions of absolute and relocatable expressions.

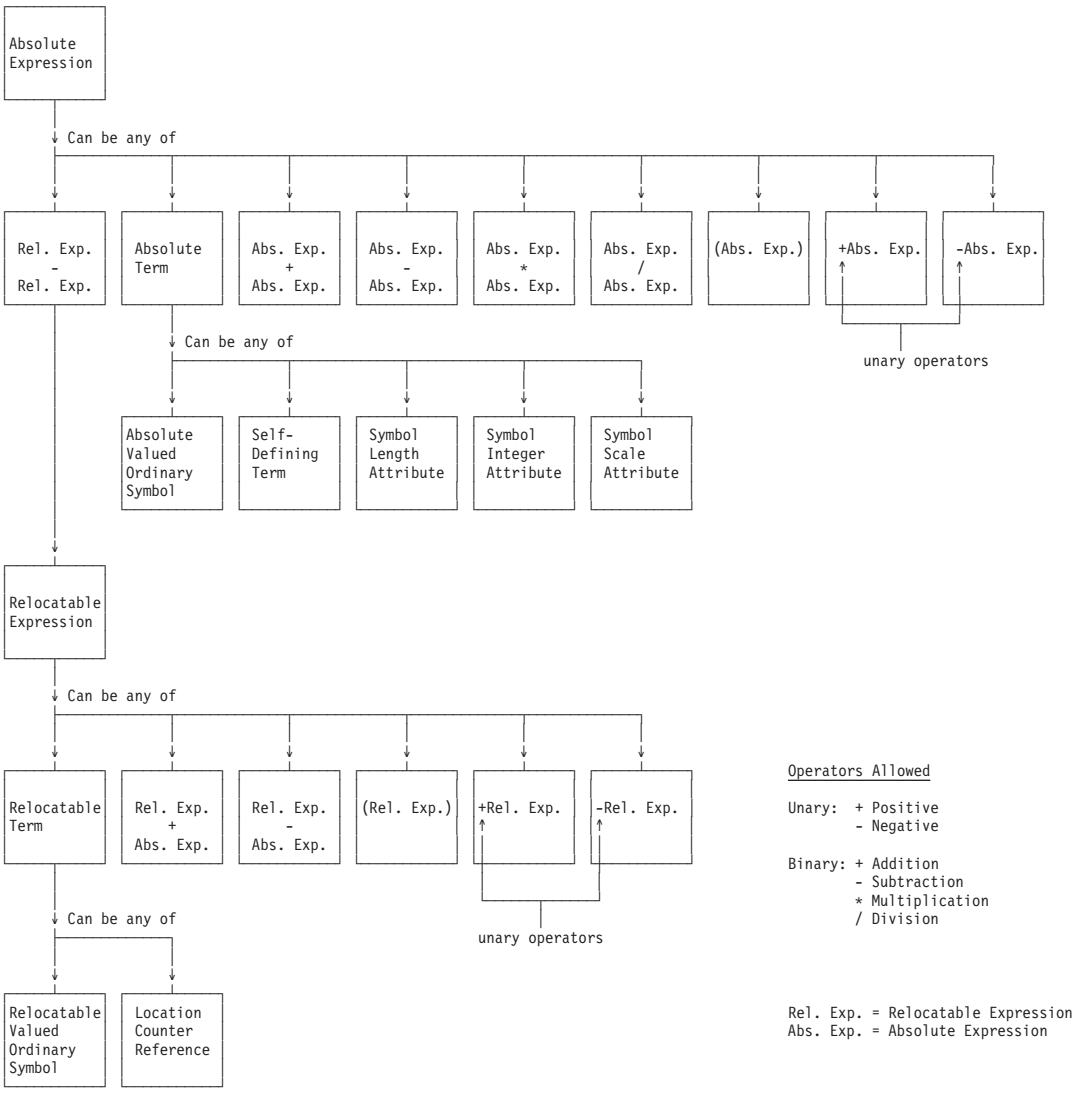


Figure 11. Definitions of absolute and relocatable expressions

Evaluation of expressions

A single-term expression, like `29` or `BETA`, has the value of the term involved. The assembler reduces a multiterm expression, like `25*10+A/B` or `BETA+10`, to a single value, as follows:

1. It evaluates each term.
2. It does arithmetic operations from left to right. However:
   a. It does unary operations before binary operations.
   b. It does binary operations of multiplication and division before the binary operations of addition and subtraction.
3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives 0.
4. In parenthesized expressions, the assembler evaluates the innermost expressions first and then considers them as terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated.
5. A term or expression's intermediate value and computed result must lie in the range of -2^{31} through $+2^{31}-1$.

The assembler evaluates paired relocatable terms at each level of expression nesting.

Absolute and relocatable expressions

An expression is *absolute* if its value is unaffected by program relocation. An expression is *relocatable* if its value depends upon program relocation. The two types of expressions, absolute and relocatable, take on these characteristics from the term or terms composing them. A description of the factors that determine whether an expression is absolute or relocatable follows.

Absolute expression: An absolute expression is one whose value remains the same after program relocation. The value of an absolute expression is called an *absolute value*.

An expression is absolute, and is reduced to a single absolute value if the expression:

1. Comprises a symbol with an absolute value, a self-defining term, or a symbol length attribute reference, or any arithmetic combination of absolute terms.

   The absolute terms can include Integer and Scale attributes, but not Type attributes.
2. Contains relocatable terms alone or in combination with absolute terms, and if all these relocatable terms are paired.

Relocatability attribute: The relocatability attribute describes the attribute of a relocatable term. If two terms are defined in the same control section, they are characterized as having the same relocatability attribute.

If the terms are defined in different control sections, or have different relocatability attributes, the expression is said to be "complex relocatable".

The relocatability attribute is the same as the ESDID for external symbols, and the "Relocation ID" in the listing.

Paired relocatable terms: An expression can be absolute even though it contains relocatable terms, if all the relocatable terms are paired. The pairing of relocatable terms cancels the effect of relocation.

The assembler reduces paired terms to single absolute terms in the intermediate stages of evaluation. The assembler considers relocatable terms as paired under the following conditions:

- The paired terms must have the same relocatability attribute.
- The paired terms must have opposite signs after all unary operators are resolved. In an expression, the paired terms do not have to be contiguous (that is, other terms can come between the paired terms).

The following examples show absolute expressions. `A` is an absolute term; `X` and `Y` are relocatable terms with the same relocatability:

```
A-Y+X
A
A*A
X-Y+A
(***)-(***)
*-*
```

A reference to the location counter must be paired with another relocatable term from the same control section; that is, with the same relocatability. For example:

```
*-Y
```

Relocatable expression: A relocatable expression is one whose value changes by *n* if the origin of the control section in which it appears is relocated *n* bytes.

A relocatable expression can be a single relocatable term. The assembler reduces a relocatable expression to a single relocatable value if the expression:

1. Is composed of a single relocatable term, or
2. Contains relocatable terms, alone or in combination with absolute terms, and
   a. All the relocatable terms but one are paired. The unpaired term gives the expression a relocatable value; the paired relocatable terms and other absolute terms constitute increments or decrements to the value of the unpaired term.
   b. The relocatability attribute of the whole expression is that of the unpaired term.
   c. The sign preceding the unpaired relocatable term must be positive, after all unary operators have resolved.

The following examples show relocatable expressions. A is an absolute term, W and X are relocatable terms with the same relocatability attribute, and Y is a relocatable term with a different relocatability attribute.

```
Y-32*A                    W-X+*     =F'1234' (literal)
* (reference to           W-X+W     Y
  location counter)        W-X+Y     A*A+W-W+Y
```

Complex relocatable expressions: Complex relocatable expressions, unlike relocatable expressions, can contain:
- Two or more unpaired relocatable terms
- An unpaired relocatable term preceded by a negative sign

Using the same symbols, here are examples of complex relocatable expressions:

```
W+X                 *+*
X-Y                 A-W+Y
```

Complex relocatable expressions are used in A-type and Y-type address constants to generate address constant values. For more details, refer to ""Complex relocatable expressions" on page 133", and "Address constants—A and Y" on page 133. V-type and S-type constants cannot contain complex relocatable expressions. You can assign a complex relocatable value to a symbol using the EQU instruction, as described in "EQU instruction" on page 162.

Chapter 3. Program structures and addressing

This chapter describes:

- How you use symbolic addresses to refer to data in your assembler language program.
- How you divide a large program into smaller parts and use symbolic addresses in one part to refer to data in another part.

Object program structures

High Level Assembler supports two object-program models. The older "load module" model generally involves one or more independently relocatable control sections combined into a single block of machine language text, which is loaded into a single contiguous portion of memory. Addresses within this block of text are resolved to locations within the block, or are left unresolved. Such programs are considered one-dimensional structures. Examples include z/OS load modules, CMS modules, and z/VSE phases.

z/VM and z/OS

The second object-program model supports a two-dimensional structure called a *program object*. The loaded program can consist of one or more contiguous blocks of machine language text grouped in *classes* and placed in different portions of memory. Each contribution of machine language text to a class is provided by an owning *section*, and the independently relocatable text from a section that contributes to a *class* is an *element*. For certain types of class, an element can contain *parts*. Unlike a control section, a program object section can specify more than one independently relocatable block of text. Addresses within each class can be resolved to addresses in the same or different classes. A class in a program object has behavior properties like those of a load module.

Section names are specified with the CSECT, RSECT, and START statements, and class and part names are specified with the CATTR statement. Additional attributes can be assigned to external symbols with the XATTR statement.

The program object model can be created only when the GOFF option is specified. The "load module" model can be created when either the NOGOFF or GOFF option is specified, but there are limitations on source program statements if GOFF is specified.

Note: The term "section" is used in different senses for each object-program model. In the load module model, a section is a control section. In the program object model, a section is a one-dimensional cross-section of program object data containing contributions to one or more classes.

z/VM and z/OS

Note: Features supported by High Level Assembler when you specify the GOFF option might not be supported by the system linker/binder or run-time environment where the assembled program is processed. You should check the relevant product documentation before utilizing the assembler's features.

The following figure illustrates the differences between the object-program models.

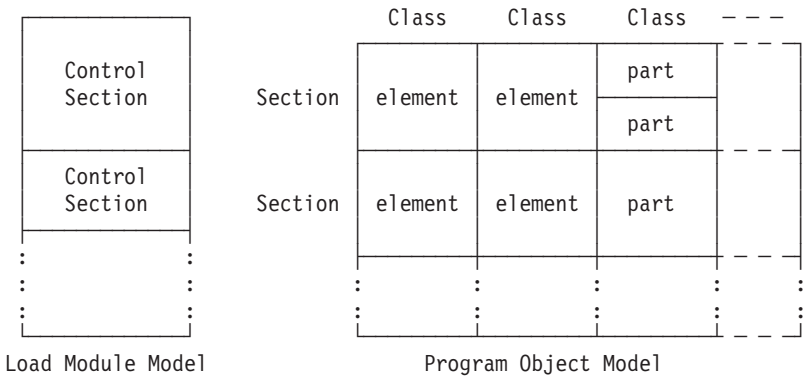


Figure 12. Load module and Program Object structures

Source program structures

This part of the chapter explains how to subdivide a large program into smaller parts that are easier to understand and maintain. It also explains how to divide these smaller parts such as one section or element to contain executable instructions, and another to contain data constants and work areas.

You should consider two different subdivisions when writing an assembler language program:
- The source module
- The control section (load module model), or sections, elements, and parts (program object model)

You can divide a program into two or more source modules. Each source module is assembled into a separate object module. The object modules can then be combined to form an executable program.

You can also divide a source module into two or more sections, or (in the program object model) into sections containing multiple classes. Each section is assembled as part of the same object module. By writing the correct linker control statements, you can select a complete object module or any individual section of the object module to be linked and later loaded as an executable program.

Size of Program Components

If a source module becomes so large that its logic is not easily understood, divide it into smaller modules. For some instructions, at most 4096 bytes can be addressed by one base register. Long-displacement instructions allow you to address 1048576 bytes with one base register.

Communication between Program Components

You must be able to communicate between the components of your program; that is, be able to refer to data in a different component or branch to an instruction in another component.

To communicate between two or more source modules, you must link them together with applicable symbolic references.

To communicate between two or more sections or elements within a source module, you must correctly establish the addressability of each to the others.

Source module

A source module is composed of source statements in the assembler language. You can include these statements in the source module in two ways:

- You can enter them directly into the file that contains your source program.

- You specify one or more COPY instructions among the source statements being entered. When High Level Assembler encounters a COPY instruction, it replaces the COPY instruction with a predetermined set of source statements from a library. These statements then become a part of the source module. See "COPY instruction" on page 105 for more details.

Beginning of a source module

The first statement of a source module can be any assembler language statement, except MEXIT and MEND. You can initiate the first control section of a source module by using the START instruction. However, you can write some source statements before the beginning of the first control statement. See "First section" on page 46 for more details.

End of a source module

The END instruction marks the end of a source module. However, you can code several END instructions; conditional assembly processing can determine which of several coded or substituted END instructions is to be processed. Also, specifying the BATCH option allows you to supply more than one source module in a single input stream. When BATCH is specified, the assembler completes assembling a source module when an END statement is encountered. If further statements are found in the input stream, assembly of a new source module is begun. See "END instruction" on page 160 for more details, and the section "BATCH" in the *HLASM Programmer's Guide* for information about the BATCH option.

Conditional Assembly: Conditional assembly processing can determine which of several coded or substituted END instructions is to be processed.

Sections, elements, and parts

In the load module model, a *control section* is the smallest subdivision of a program that can be relocated as a unit. The assembled control sections contain the object code for machine instructions, data constants, and areas.

In the program object model, elements and parts are the smallest subdivisions of a program that can be relocated as a unit. Sections allow grouping all element and part contributions under a single name. The assembled sections, elements, and parts contain the object code for the machine instructions, data, and areas.

Consider the concept of a control section at different processing times:

At coding time
: You create a control section or an element or part when you write the instructions it contains. In addition, you establish the addressability of each component within the source module, and provide any symbolic linkages between components that lie in different source modules. You also write the linker control statements to combine sections into a load module or program object, and to provide an entry point address for the beginning of program execution.

At assembly time
: High Level Assembler translates the source statements into object code. Each source module is assembled into one object module. The contents of the object module are relocatable.

At linking time
: As specified by linker or binder control statements, the linker or binder combines the object code of one or more sections into one load module or program object. It also calculates the addresses needed to accommodate common sections and external dummy sections from different object modules. In addition, it calculates the space needed to accommodate external dummy sections.

  You can specify the relocatable address of the starting point for program execution in a linker control statement or request a starting address in the operand field of an assembler END instruction.

At program fetch time
The control program loads the load module or program object into virtual storage. All the relocatable addresses are converted to fixed locations in storage.

At execution time
The control program passes control to the loaded program now in virtual storage, and your program is run.

Sections

In the load module model, control sections might generate machine language text containing instructions and data, or define mappings of storage areas to be referenced at execution time. Control sections that generate machine language text are called *executable control sections*, even though they might contain only data. Control sections that create only mappings are called *reference control sections*.

z/VM and z/OS

In the program object model, sections can define *classes* containing *elements*. (Classes are described in "Classes (z/OS and CMS)" on page 50.) Elements can contain machine language text or define mappings, or both. Elements can in turn contain one or more *parts*, which are described at "Parts (z/OS and CMS)" on page 52.

Elements containing machine language text are usually linked in a class comprising other elements containing machine language text, and elements defining mappings are usually linked in a class with other elements defining mappings.

The section name is used in binder operations to refer to its entire collection of elements and parts, but a program object section is not the same as a load module control section. A section name can be referenced as an external name only if defined as an entry point in an element belonging to that section. (By default, the assembler generates an entry point in class `B_TEXT` with the section's name. See "Classes (z/OS and CMS)" on page 50 for more information.)

The term "executable" is used to describe executable control sections in the load module model, or elements in the program, or sections in the program object model, or elements in the program object model.

You initiate an executable section by using the START, CSECT, or RSECT instruction, as described below:

- The START instruction can be used to initiate the first or only section of a source module. For more information about the START instruction, see "START instruction" on page 189.
- The CSECT instruction can be used anywhere in a source module to initiate or continue a section. For more information about the CSECT instruction, see "CSECT instruction" on page 106.
- Like the CSECT instruction, the RSECT instruction can be used anywhere in a source module to initiate or continue a section. Unlike the CSECT instruction, however, the RSECT instruction causes the assembler to check the coding in the section for possible violations of reenterability. For more information about the RSECT instruction, see "RSECT instruction" on page 188.

A section can also be initiated as an unnamed section, or *private code*, without using the START, CSECT, or RSECT instruction. For more information, see "Unnamed section" on page 47.

First section

Before you initiate the first section in your source module, you can code only certain instructions. The following information lists those instructions that initiate the first section, and those instructions that can precede the first section.

What must appear before the first section: The ICTL instruction, if specified, must be the first statement in a source module.

*PROCESS statements must precede all other statements in a source module, except the ICTL instruction. There is a limit of 10 *PROCESS statements allowed in a source module. Additional *PROCESS statements are treated as assembler comment statements. See page "*PROCESS statement" on page 84 for a description of the *PROCESS statement.

What can optionally appear before the first executable control section: The instructions or groups of instructions that can optionally be specified before the first executable control section are:

- The following assembler instructions:

```
ACONTROL ADATA    AINSERT  ALIAS    CEJECT   COPY
DXD      EJECT    ENTRY    EXITCTL  EXTRN    ISEQ
MACRO    MEND     MEXIT    POP      PRINT    PUNCH
PUSH     REPRO    SPACE    TITLE    WXTRN    XATTR
```

- Comment statements, including macro format comment statements
- Any statement which is part of an inline macro definition, between MACRO and MEND statements, with the possible exception of *PROCESS and ICTL.
- Common control sections
- Dummy control sections
- Any conditional assembly instruction
- Macro instructions that do not generate statements that establish the first section

These instructions or groups of instructions belong to a source module, but are not considered part of an executable section.

Instructions that establish the first section: Any instruction that affects the location counter, or uses its current value, establishes the beginning of the first executable section. The instructions that establish the first section include any machine instruction and the following assembler instructions:

```
CCW      CCW0     CCW1     CNOP     COM      CSECT
CXD      DC       DS       DSECT    EQU      LOCTR
LTORG    ORG      RSECT    START    USING
```

COM, CSECT, DSECT, RSECT, and START start a possibly named control section. The other statements start an unnamed control section.

These instructions are always considered a part of the control section in which they appear.

The DSECT, COM, and DXD instructions initiate reference control sections and do not establish the first executable section.

The statements copied into a source module by a COPY instruction determine whether it initiates the first control section.

Any instructions copied by a COPY instruction, or generated by the processing of a macro instruction before the first section, must belong to one of the groups of instructions shown above. Any other instructions cause the assembler to establish the first section.

All the instructions or groups of instructions listed above can also appear as part of a section.

If you specify the PROFILE assembler option the assembler generates a COPY statement as the first statement in the assembly after any ICTL or *PROCESS statements. The copy member should not contain any ICTL or *PROCESS statements.

Unnamed section

The *unnamed section* is an executable section that can be initiated in one of the following two ways:

- By coding a START, CSECT, RSECT, or COM instruction without a name entry
- By coding any instruction, other than the START, CSECT, or RSECT instruction, that initiates the first executable section

An unnamed control section is sometimes referred to as *private code*. Private code sections are sometimes difficult to manage with other system components such as linkers and configuration management tools. Avoiding their use is recommended. (Zero-length private code sections are sometimes ignored or discarded by system linkers.)

All sections should be given names so they can be referred to symbolically:

- Within a source module
- In EXTRN and WXTRN instructions
- In linker control statements for section ordering and replacement, and for linkage between source modules

Unnamed common control sections or dummy control sections can be defined if the name entry is omitted from a COM or DSECT instruction.

If you include an AMODE or RMODE instruction in the assembly and leave the name field blank, you must provide an unnamed control section.

Reference control sections

A *reference control section* is one you initiate by using the DSECT, COM, or DXD instruction, as follows:

- You can use the DSECT instruction to initiate or continue a dummy control section. For more information about dummy sections, see "Dummy control sections."
- You can use the COM instruction to initiate or continue a common control section. For more information about common sections, see "Common control sections" on page 49.
- You can use the DXD instructions to define an external dummy section. For more information about external dummy sections, see "External dummy sections" on page 49.

At assembly time, reference control sections are not assembled into object code. You can use a reference control section either to reserve storage areas or to describe data to which you can refer from executable control sections. These reference control sections are considered empty at assembly time, and the actual binary data to which they refer is not available until execution time.

Dummy control sections

A *dummy control section* is a reference control section that describes the layout of data in a storage area without reserving any virtual storage.

You might want to describe the format of an area whose storage location is not determined until the program is run. You can do so by describing the format of the area in a dummy section, and using symbols defined in the dummy section in the operands of machine instructions.

The DSECT instruction initiates a dummy control section or indicates its continuation. For more information about the DSECT instruction, see "DSECT instruction" on page 157.

How to use a dummy control section: A dummy control section (dummy section) lets you write a sequence of assembler language statements to describe the layout of data located elsewhere in your source module. The assembler produces no object code for statements in a dummy control section, and it reserves no storage in the object module for it. Rather, the dummy section provides a symbolic template or mapping that is empty of data. However, the assembler assigns location values to the symbols you define in a dummy section, relative to its beginning.

Therefore, to use a dummy section, you must:

- Have access to a storage area for the data
- Ensure that the locations of the symbols in the dummy section correspond to the locations of the data being described

- Establish the addressability of the dummy section in combination with the storage area

You can then refer to the data symbolically by using the symbols defined in the dummy section.

Common control sections

A *common control section* is a reference control section that lets you reserve a storage area that can be used by one or more source modules. One or more common sections can be defined in a source module.

The COM instruction initiates a common control section, or indicates its continuation. For more information about the COM instruction, see "COM instruction" on page 104.

How to use a common control section: A common control section (common section) lets you describe a common storage area in one or more source modules.

When the separately assembled object modules are linked as one program, the required storage space is reserved for the common control section. Thus, two or more modules can share the common area.

Only the storage area is provided; the assembler does not assemble the source statements that make up a common control section into object code. You must provide the data for the common area at execution time.

The assembler assigns locations to the symbols you define in a common section relative to the beginning of that common section. This lets you refer symbolically to the data that is placed in the common section at execution time. If you want to refer to data in a common control section, you must establish the addressability of the common control section in each source module that contains references to it. If you code identical common sections in two or more source modules, you can communicate data symbolically between these modules through this common section.

Communicating with modules in other languages: Some high-level languages such as COBOL, PL/I, C, and Fortran use common control sections. This lets you communicate between assembler language modules and modules written in those languages.

External dummy sections

An *external dummy section* is a reference control section that lets you describe storage areas for one or more source modules, to be used as:
- Work areas for each source module
- Communication areas between two or more source modules

Note: External dummy sections are also called "pseudo-registers" in other contexts.

When the assembled object modules are linked and loaded, you can dynamically allocate the storage required for all your external dummy sections at one time from one source module (for example, by using the z/OS GETMAIN macro instruction). This is not only convenient, but it saves space and reduces fragmentation of virtual storage.

Typical bind-time processing of external dummy sections involves "merging" the attributes of identically named external dummy sections, retaining the longest length and strictest alignment among all identically-named external dummy sections. In particular, the lengths of identically named external dummy sections are not additive.

To generate and use the external dummy sections, you need to specify a combination of the following:
- DXD or DSECT instruction
- Q-type address constant
- CXD instruction

For more information about the DXD and CXD instructions, see “DXD instruction” on page 159 and “CXD instruction” on page 108.

Note: The names of dummy external control sections might match the names of other external symbols that are not names of dummy control sections, without conflict.

Generating an external dummy section: An external dummy section is generated when you specify a DXD instruction, or when you specify a DSECT instruction whose name appears in a Q-type address constant.

When a DSECT name is used as an operand of a Q-type address constant, that name becomes an external symbol with type XD in the External Symbol Dictionary portion of the listing. The name must satisfy the name-length requirements of the object file format specified in the assembler options.

Use the Q-type address constant to reserve storage for the offset to the external dummy section whose name is specified in the operand. This offset is the distance in bytes from the beginning of the area allocated for all the external dummy sections to the beginning of the external dummy section specified. You can use this offset value to address the external dummy section.

Using external dummy sections: To use an external dummy section, you must do the following:

1. Identify and define the external dummy section. The assembler computes the length and alignment required. The linker merges this definition with other definitions of the same name, assigning the longest length and strictest alignment.
2. Provide a Q-type constant for each external dummy section defined.
3. Use the CXD instruction to reserve a fullword area into which the linker or loader inserts the total length of all the external dummy sections that are specified in the source modules of your program. The linker computes this length from the accumulated lengths of the individual external dummy sections supplied by the assembler.
4. Allocate a storage area using this computed total length.
5. Load the address of the allocated area into a register.
6. Add to the address in the register the offset into the allocated area of the applicable external dummy section. The linker inserts this offset into the area reserved by the associated Q-type address constant.
7. Establish the addressability of the external dummy section in combination with the portion of the allocated area reserved for the external dummy section.

You can now refer symbolically to the locations in the external dummy section. The source statements in an external dummy section are not assembled into object code. Thus, you must create the data described by external dummy sections at execution time.

Note: During linking, external dummy sections might be arranged in any order. Do not assume any ordering relationship among external dummy sections.

Classes (z/OS and CMS)

Each section's contributions to a program object are assigned to one or more classes, according to their desired binding and loading properties. Class names are assigned either by default (see “Default class assignments” on page 51) or explicitly. You define a *class* with the CATTR instruction, which must follow the initiation of an executable section. The class name is provided in the name entry of the CATTR instruction, and attributes of the class are provided by the operands of the first CATTR instruction declaring the class. (See “CATTR instruction (z/OS and CMS)” on page 96 for further information.) The element containing subsequent machine language text or storage definitions is defined by the combination of the section and class names, as illustrated in Figure 12 on page 44.

For example, suppose you define two classes, `CLASS_X` and `CLASS_Y`:

```
SECT_A   CSECT ,             Define section SECT_A
CLASS_X  CATTR RMODE(ANY)    Define class CLASS_X
         - - -               Statements for CLASS_X
CLASS_Y  CATTR RMODE(24)     Define class CLASS_Y
         - - -               Statements for CLASS_Y
```

The statements following the first CATTR instruction are assigned to an element defined by the section name SECT_A and the class name CLASS_X. Similarly, the statements following the second CATTR instruction are assigned to an element defined by the section name SECT_A and the class name CLASS_Y. CLASS_Y is loaded below 16 Mb, and CLASS_X might be loaded anywhere below 2 Gb.

Class names are rarely referenced, because the attributes of the class, such as RMODE, are much more important.

You can resume a class by providing additional CATTR statements with the class name in the name entry. No attributes of the class can be specified after the first CATTR statement declaring the class.

Resuming a section causes subsequent text to be placed in the B_TEXT class if there is no intervening CATTR statement defining or resuming a different class:

```
SECT_A   CSECT ,             Define section SECT_A
CLASS_X  CATTR RMODE(ANY)    Define class CLASS_X
         - - -               Statements for CLASS_X
CLASS_Y  CATTR RMODE(24)     Define class CLASS_Y
         - - -               Statements for CLASS_Y
SECT_A   CSECT ,             Resume section SECT_A
         - - -               Statements for class B_TEXT
CLASS_X  CATTR ,             Resume class CLASS_X
         - - -               More statements for CLASS_X
```

Class binding and loading attributes

Each class is bound into a separately relocatable loadable *segment*, using one of two binding attributes.

- Classes containing parts use *merge* binding (described at "Parts (z/OS and CMS)" on page 52). Parts are the smallest independently relocatable components of a merge class.
- Classes not containing parts use *concatenation* binding, in which elements, after suitable alignment, are placed one after another. Zero-length elements are retained but take no space in the program object. Elements are the smallest independently relocatable components of a concatenation class

Each class must have uniform binding and loading attributes. More than one class can have identical attributes, and the binder can put such classes into one segment. The most usual class attributes are RMODE, alignment, and Loadability; see "CATTR instruction (z/OS and CMS)" on page 96 for further information.

Class loading attributes determine the load-time placement of segments in virtual storage. Loadable segments are loaded as separately relocated non-contiguous entities at different origin addresses.

Default class assignments

High Level Assembler provides compatible behavior with "load module" model object files generated when the NOGOFF option is active. When the GOFF option is specified, the assembler automatically follows each CSECT, RSECT, and START statement by defining two classes: B_TEXT and B_PRV.

- B_TEXT contains the machine language text associated with the section name, and is assigned the RMODE of the section name. The section name is assigned to an entry point at the origin of the class. If a subsequent CATTR statement declares a class name before any other statements have defined storage, the element defined by the section name and the B_TEXT class name is empty.
- B_PRV contains any external dummy sections defined by DXD instructions, or by DSECTs named in Q-type address constants. If none are defined, the elements in this class are empty. ("PRV" is the binder's term for a "Pseudo Register Vector", the cumulative collection of external dummy sections.)

- High Level Assembler assigns the name of the section as an entry name at the initial byte of B_TEXT, and assigns to it the AMODE of the section name.

These two classes are bound in the same way as ordinary control sections. Dummy external sections are bound in the load module model. They can be used to generate a load module if certain restrictions are satisfied.

You can declare other classes in addition to the defaults, but the resulting program object is not convertible to a load module.

Parts (z/OS and CMS)

Parts are the smallest externally named and independently relocatable subdivisions of elements in a merge class. A class containing parts can contain only parts, and a class containing anything other than parts cannot contain any parts.

ENTRY statements cannot define an entry point in a part.

You define a *part* with the CATTR instruction, which must follow the initiation of an executable section. The name of the class to which the part belongs is provided in the name entry of the CATTR instruction, and the name of the part is specified as an operand. The first definition of a class name can also specify the attributes of the class. (See "CATTR instruction (z/OS and CMS)" on page 96 for further information.)

For example, suppose you define two parts in a class:

```
SECT_B   CSECT ,             Define section SECT_B
PClass   CATTR Part(Part_R),RMODE(ANY)  Define class PClass, part Part_R
         - - -               Statements included in Part_R
PClass   CATTR Part(Part_S)  Define part Part_S in class PClass
         - - -               Statements included in Part_S
PClass   CATTR Part(Part_R)  Resume class PClass and part Part_R
         - - -               More statements included in Part_R
```

These statements define a "merge" class PClass containing two parts, Part_R and Part_S. If other classes or other object files declare parts with the same names in the same class, the binder merges their contents to determine the final part definition in the program object.

You can provide additional statements for a part by specifying a CATTR statement with the class name in the name entry and the part name specified as the operand. No other class attributes can be specified following the first CATTR statement declaring the class.

Parts are automatically assigned a "merge" attribute, meaning that more than one identically named part might appear in a class defined in other assemblies or compilations. The binder assigns the longest length and strictest alignment of all such identically named parts, and merges the machine language text contributions of each to form the final text belonging to that part. The order of text merging depends on the sequence of parts processing by the binder.

Note: During linking, parts might be arranged in any order, depending on their priority attribute. Do not assume any ordering relationship among parts.

Location counter setting

The assembler maintains a separate *location counter* for each section, element, and part. The location counter setting for the first section starts at 0, except when an initial section is started with a START instruction that specifies a nonzero location counter value. The location values assigned to the instructions and other data in a section, element, or part are, therefore, relative to the location counter setting at the beginning of that section, element, or part.

For executable sections, the location values that appear in the listings depend on the THREAD option:

- If you specify NOTHREAD, the location counter values for each section, element, or part restart at 0, except possibly those associated with a first section initiated by a START instruction with a nonzero address.
- If you specify THREAD, location counter values do not restart at 0 for each subsequent section, element, or part. They continue, after suitable alignment, from the end of the previous section, element, or part.

For reference control sections, the location values that appear in the listings always start from 0.

You can continue a control section, element, or part that has been discontinued and thus intersperse code sequences from different control sections, elements, or parts. The location values that appear in the listings for such discontiguous sequences are divided into segments that follow from the end of one segment to the beginning of the subsequent segment.

The location values, listed for the next defined control section, element, or part, begin after the last location value assigned to the preceding such item.

On z/VSE, or when you specify the NOGOFF assembler option on z/OS and CMS, the maximum value of the location counter and the maximum length of a control section is 2^{24}-1, or X'FFFFFF' bytes. If LIST(133) is in force, then the high-order byte is shown as zero.

z/VM and z/OS

When you specify the GOFF assembler option, the maximum value of the location counter and the maximum length of an element or part is 2^{31}-1, or X'7FFFFFFF' bytes.

Location counter and length limits

The assembler also maintains a *length counter* for each individually relocatable component of the program: executable and reference control sections, elements, and parts.

If any location counter overflows its maximum value, High Level Assembler issues the severe error message:

```
ASMA039S Location counter error
```

and continues assembling with the location counter value “wrapping” around to zero.

The length of a section, element, or part cannot exceed the maximum allowed length described above. If the length counter reaches this maximum value, it stays fixed at that value without an error condition or error message. Exceeding the length counter causes overflow of the location counter, producing the `ASMA039S` message.

The location counter setting is relative to the beginning of the location it represents, and the length counter represents the cumulative length of the control section. This means that the length counter is nearly always greater than the location counter, and can exceed its maximum value before the location counter. Even if the location counter overflows, the length counter value might be correct, and reassembling with the NOTHREAD option might avoid the location counter overflow condition.

Use of multiple location counters

High Level Assembler lets you use multiple location counters for each individual control section. Use the LOCTR instruction (see “LOCTR instruction” on page 169) to assign different location counters to different parts of a control section. The assembler then rearranges and assembles the coding together, according to the different location counters you have specified:

- All coding using the first location counter is assembled together.
- Then the coding using the second location counter is assembled together.
- And so on, for futher location counters.

An example of the use of multiple location counters is shown in Figure 13. In the example, executable instructions and data areas have been interspersed throughout the coding in their logical sequence. Each group of instructions is preceded by a LOCTR instruction that identifies the location counter under which it is to be assembled. The assembler rearranges the control section so that the executable instructions are grouped together and the data areas are grouped together. Symbols are not resolved in the order they appear in the source program, but in location counter sequence.

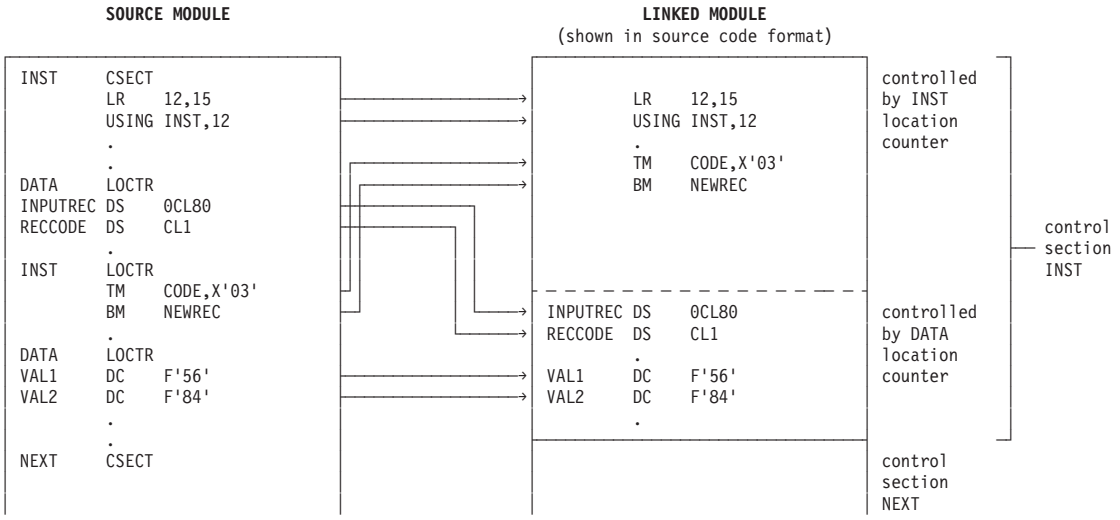


Figure 13. Use of multiple location counters

The interactions of the LOCTR instruction with sections, classes, and parts is described at "LOCTR instruction" on page 169.

Addressing

This part of the chapter describes the techniques and introduces the instructions that let you use symbolic addresses when referring to instructions and data. You can address code and data that is defined within the same source module, or code and data that is defined in another source module. Symbolic addresses are more meaningful and easier to use than the corresponding object code addresses required for machine instructions. The assembler can convert the symbolic addresses you specify into their object code form.

The z/Architecture architecture has two ways of resolving addresses in your program, depending on the machine instruction type:

- **base displacement**, where the address is computed by adding the displacement to the contents of a base register.
- **relative immediate**, where the address is computed by adding 2 × the signed immediate operand field to the instruction's address (refer to "RI format" on page 76 and "RSI format" on page 79).

Addressing within source modules: establishing addressability

You can use symbolic addresses in machine instructions and certain assembler instructions. This is much easier than explicitly coding the addresses in the form required by the hardware. Symbolic addresses you code in the instruction operands are *implicit addresses*, and addresses in which you specify the base-displacement or intermediate form are *explicit addresses*.

The assembler converts your implicit addresses into the explicit addresses required for the assembled object code of the machine instruction. However, for base-displacement operands, you must first establish addressability, as described below.

Base Address Definition: The term *base address* is used throughout this manual to mean the location counter value within a control section, element, or part from which the assembler can compute displacements to locations, or *addresses*. The base address need not always be the storage address of a control section, element, or part when it is loaded into storage at execution time.

How to establish addressability

To establish the addressability of a control section, element, or part (see "Sections, elements, and parts" on page 45), you must:

- Specify a base address from which the assembler can compute displacements to the addresses within the control section, element, or part.
- Assign the base registers to contain the base addresses.
- Write the instructions that load the base registers with the base addresses.

The following example shows the base address at MYPROG, that is assigned by register 12. Register 12 is loaded with the value in register 15. By convention, register 15 contains the storage address (set by the operating system) of the control section (CSECT) when the program is loaded into storage at execution time.

```
MYPROG   CSECT                    The base address
         USING MYPROG,12          Assign the base register
         LR    12,15              Load the base address
```

Similarly, you can use a BASR or similar instruction to put the address of the following instruction into register 12.

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

The USING instruction indicates that register 12 can be used as a base register containing that address.

During assembly, the implicit addresses you code are converted into their explicit base-displacement form; then, they are assembled into the object code of the machine instructions in which they have been coded.

During execution, the base address is loaded into the base register.

z/VM and z/OS

If you specify multiple classes, you must provide addressability for each element. For example, suppose you define two classes that must reference positions in the other:

```
MYPROG   CSECT ,
CLASS_A  CATTR RMODE(24)     Define class CLASS_A
         BASR  12,0          Local base register
         USING *,12          Addressability for this element
         - - -
         L     1,Addr_B      Address of BDATA in CLASS_B
         USING BDATA,1
         - - -
ADATA    DS    F             Data in CLASS_A
Addr_B   DC    A(BDATA)
         - - -
CLASS_B  CATTR RMODE(31)     Define class CLASS_B
         BASR  11,0          Local base register
         USING *,11          Addressability for this element
         - - -
         L     2,Addr_A      Address of ADATA in CLASS_A
         USING ADATA,2
         - - -
BDATA    DS    D             Data in CLASS_B
Addr_A   DC    A(ADATA)
```

A class specifying the "deferred load" (DEFLOAD) attribute on its defining CATTR statement cannot be referenced from other classes using A-type or V-type address constants. However, A-type and V-type address constants can be used within a deferred-load class to refer to locations within that class or within any default_load (LOAD) class.

The loading service for deferred-load classes provides the origin address of the deferred-load segment containing the classes. You can then use Q-type address constants in other classes to calculate the addresses of items in the loaded classes. For example:

```
MYPROG   CSECT ,
CLASS_A  CATTR RMODE(31)
         BASR  12,0          Set base register
         USING *,12          Addressability for this element
         - - -
* Address of CLASS_B segment assumed to be returned in register 8
         - - -
         A     8,BDATAOff    Add offset of BDATA in CLASS_B
         USING BDATA,8
         - - -
BDATAOff DC    Q(BDATA)      Offset of BDATA
         - - -
CLASS_B  CATTR DEFLOAD,RMODE(ANY)  Define deferred-load class
         - - -
BDATA    DS    F             Data in deferred-load class
```

Parts must always be referenced from LOAD classes using Q-type address constants using the techniques shown in this example, whether or not they reside in deferred load classes. This is because parts are subject to reordering during binding. As noted above, parts can reference other parts in the same class using A-type and V-type address constants.

Base register instructions

The USING and DROP assembler instructions enable you to use expressions representing implicit addresses as operands of machine instruction statements, leaving the assignment of base registers and the calculation of displacements to the assembler.

In order to use symbols in implicit addresses in the operand field of machine instruction statements, you must:

- Code a USING instruction to assign one or more base registers to a base address or sequence of base addresses
- Code machine instructions to load each base register with the base address

Having the assembler determine base registers and displacements relieves you of the need to separate each address into an explicit displacement value and an explicit base register value. This feature of the assembler eliminates a likely source of programming errors, thus reducing the time required to write and test programs. You use the USING and DROP instructions to take advantage of this feature. For information about how to use these instructions, see "USING instruction" on page 193 and "DROP instruction" on page 152.

Qualified addressing

Qualified addressing lets you use the same symbol to refer to data in different storage locations. Qualified symbols are ordinary symbols prefixed by a symbol qualifier and a period. A symbol qualifier is used to specify which base register the assembler should use when converting an implicit address into its explicit base-displacement form. Before you use a symbol qualifier, you must have previously defined it in the name entry of a labeled USING instruction. For information about labeled USING instructions, see "USING instruction" on page 193. When defined, you can use a symbol qualifier to qualify any symbol that names a storage location within the range of the labeled USING. Qualified symbols can be used anywhere a relocatable term can be used.

The following examples show the use of qualified symbols. SOURCE and TARGET are both symbol qualifiers previously defined in two labeled USING instructions. X and Y are both symbols that name storage locations within the range of both labeled USING instructions.

```
         MVC             TARGET.X,SOURCE.X
         MVC             TARGET.Y+5(3),SOURCE.Y+5
         XC              TARGET.X+10(L'X-10),TARGET.X+10
         LA              2,SOURCE.Y
```

Dependent addressing

Dependent addressing lets you minimize the number of base registers required to refer to data by making greater use of established addressability. For example, you might want to describe the format of a table of data defined in your source module with a dummy control section (see "Dummy control sections" on page 48). To refer to the data in the table using the symbols defined in the dummy section, you need to establish the addressability of the dummy section. To do this you must:

- Code a USING instruction to assign one or more base registers to a base address.
- Code machine instructions to load each base register with the base address.

However, dependent addressing offers an alternative means of establishing addressability of the dummy section.

Establish addressability of the control section in which the table is defined. Then you can establish addressability of the dummy section by coding a USING statement which specifies the name of the dummy section and the address of the table. When you refer to the symbols in the dummy section, the assembler uses the already established addressability of the control section when converting the symbolic addresses into their base-displacement form.

For example, suppose addressability has been established for a control section containing a data structure that is mapped by a dummy control section:

```
DATAMAP  DSECT ,                 DSECT describing data structure
FIELD1   DS    F
FIELD2   DS    CL32
FIELD3   DS    CL24
         - - -
CODE     CSECT ,                 Program code
         BASR  12,0              Set base register
         USING *,12              Provide addressability
         - - -
         USING DATAMAP,REALDATA  Map DSECT onto REALDATA
         L     2,FIELD1          Register 12 is base register
         LA    3,FIELD3          Address of DATA3
         - - -
REALDATA DS    0F                Data mapped by DATAMAP
DATA1    DC    F'32'
DATA2    DC    CL32'Actual Data'
DATA3    DC    CL24'Additional Data'
```

Relative addressing

Relative addressing is the technique of addressing instructions and data areas by designating their location in relation to the location counter or to some symbolic location. This type of addressing is always in bytes—never in bits, words, or instructions. Thus, the expression *+4 specifies an address that is 4 bytes greater than the current value of the location counter. In the sequence of instructions in the following example, the location of the CR machine instruction can be expressed in two ways, ALPHA+2, or BETA-4, because all the machine instructions in the example are for 2 byte instructions.

```
ALPHA    LR              3,4
         CR              4,6
         BCR             1,14
BETA     AR              2,3
```

Literal pools

Literals, collected into pools by the assembler, are assembled as part of the executable control section to which the pools belong. If an LTORG instruction is specified at the end of each control section or element, the literals specified for that section or element are assembled into the pool starting at the LTORG instruction. If no LTORG instruction is specified, a literal pool containing all the literals used in the whole source module is assembled at one of:

- The end of the first control section.
- The end of the B_TEXT class belonging to the first section.

This literal pool appears in the listings after the END instruction. For more information about the LTORG instruction, see "LTORG instruction" on page 171.

Independently Addressed Segments: If any control section is divided into independently addressed segments, an LTORG instruction should be specified at the end of each segment to create a separate literal pool for that segment.

Establishing residence and addressing mode

The AMODE and RMODE instructions specify the addressing mode (AMODE) and the residence mode (RMODE) to be associated with control sections in the object deck. You can specify AMODE for ENTRY, EXTRN, and WXTRN instruction operands if the GOFF option is specified. If OBJ format is used, then AMODE is not valid for ENTRY, EXTRN, or WXTRN instruction operands. AMODE and RMODE can be specified for any CSECT or START operand with either OBJ or GOFF and without restriction on the xMODE operands. These modes can be specified for these types of control sections:

- Control section (for example START, CSECT)
- Unnamed control section
- Common control section (COM instruction)

The assembler sets the AMODE and RMODE indicators in the ESD record for each applicable external symbol in an assembly. The linker stores the AMODE and RMODE values in the bound program. They are later used by the loader program that brings the load module into storage. The loader program uses the RMODE value to determine where it loads the load module, and passes the AMODE value of the executable program's main entry point to the operating system to establish the addressing mode.

z/VM and z/OS

When you specify the GOFF option:

- The RMODE value specified for a section is by default assigned to the B_TEXT class.
- The AMODE specified for the section is assigned to an entry point having the section name and the location of the first byte of class B_TEXT.

If the source program defines additional classes, each class might be assigned its own RMODE, and an entry point in any class might be assigned its own AMODE.

For more information about the AMODE and RMODE instructions, see "AMODE instruction" on page 95 and "RMODE instruction" on page 187.

Symbolic linkages

Symbols can be defined in one module and referred to in another, which results in symbolic linkages between independently assembled program sections. These linkages can be made only if the assembler can provide information about the linkage symbols to the linker, which resolves the linkage references at link-edit time.

Establishing symbolic linkage

You must establish symbolic linkage between source modules so that you can refer to or branch to symbolic locations defined in the control sections of external source modules. You do this by using external symbol definitions, and external symbol references. To establish symbolic linkage with an external source module, you must do the following:

- In the current source module, you must identify the symbols that are not defined in that source module, if you want to use them in instruction operands. These symbols are called external symbols, because they are defined in another (external) source module. You identify external symbols in the EXTRN or WXTRN instruction, or the V-type address constant. For more information about the EXTRN and WXTRN instructions, see "EXTRN instruction" on page 167 and "WXTRN instruction" on page 202.
- In the external source modules, you must identify the symbols that are defined in those source modules, and that you refer to from the current source module. The two types of definitions that you can use are control section names (defined by the CSECT, RSECT, and START instructions), and entry symbols. Entry symbols are so called because they provide points of entry to a control section in a source module. You identify entry symbols with the ENTRY instruction. For more information about the ENTRY instruction, see "ENTRY instruction" on page 162.
- Your reference external symbols using one of these methods:
  - Provide the A-type or V-type address constants needed by the assembler to reserve storage for the addresses represented by the external symbols.
  - Reference an external symbol in the same class in a relative branch instruction.

The assembler places information about entry and external symbols in the external symbol dictionary. The linker uses this information to resolve the linkage addresses identified by the entry and external symbols.

Referring to external data

Use the EXTRN instruction to identify the external symbol that represents data in an external source module, if you want to refer to this data symbolically.

For example, you can identify the address of a data area as an external symbol and load the A-type address constant specifying this symbol into a base register. Then, you use this base register when establishing the addressability of a dummy section that describes this external data. You can now refer symbolically to the data that the external area contains.

You must also identify, in the source module that contains the data area, the address of the data as an entry symbol.

Branching to an external address

Use the V-type address constant to identify the external symbol that represents the address in an external source module that you want to branch to.

For example, you can load into a register the V-type address constant that identifies the external symbol. Using this register, you can then branch to the external address represented by the symbol.

If the symbol is the name entry of a START, CSECT, or RSECT instruction in the other source module, and thus names an executable control section, it is automatically identified as an entry symbol. If the symbol represents an address in the middle of a control section, you must identify it as an entry symbol for the external source module.

You can also use a combination of an EXTRN instruction to identify, and an A-type address constant to contain, the external branch address. However, the V-type address constant is more convenient because:

- You do not have to use an EXTRN instruction.
- The external symbol you specify, can be used in the name entry of any other statement in the same source program.
- It works correctly even if the program is linked as an overlay module, so long as the reference is not to a symbol in an exclusive segment. See *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643 for further information.

The following example shows how you use an A-type address constant to contain the address of an external symbol that you identify in an EXTRN instruction. You cannot use the external symbol name EXMOD1 in the name entry of any other statement in the source program.

```
         .
         .
         L     15,EX_SYM        Load address of external symbol
         BASR  14,15            Branch to it
         .
         .
EX_SYM   DC    A(EXMOD1)        Address of external symbol
         EXTRN EXMOD1           Identify EXMOD1 as external symbol
         .
         .
```

The following example shows how you use the symbol EXMOD1 as both the name of an external symbol and a name entry on another statement.

```
         .
         .
         L     15,EX_SYM        Load address of external symbol
         BASR  14,15            Branch to it
         .
         .
EXMOD1   DS    0H               Using EXMOD1 as a name entry
         .
         .
EX_SYM   DC    V(EXMOD1)        Address of external symbol
         .
         .
```

If the external symbol that represents the address to which you want to branch is part of an overlay-structured module, identify it with a V-type address constant. Do not use an EXTRN instruction and an A-type address constant. You can use the supervisor CALL macro instruction to branch to the address represented by the external symbol. The CALL macro instruction generates the necessary V-type address constant.

z/VM and z/OS

You can branch to external symbols in the same class using relative branch instructions.

```
MYPROG   CSECT ,              Define section MYPROG
CLASS_A  CATTR RMODE(31)      Define class CLASS_A
         - - -
         BRAS  14,ENTRYB      Branch to external symbol
         - - -
HISPROG  CSECT ,              Define section HISPROG
CLASS_A  CATTR RMODE(31)      Define class CLASS_A
         - - -
ENTRYB   STM   14,12,12(13)   Entry point referenced externally
         - - -
         END
```

You can also use a relative branch instruction to branch to an externally defined symbol:

```
MYPROG   CSECT ,              Define section MYPROG
MYCLASS  CATTR RMODE(31)      Define class MYCLASS
         EXTRN TARGET         Declare external symbol TARGET
         - - -
         BRAS  14,TARGET      Branch to external symbol
         - - -
         END
```

A separate source module must define the entry point TARGET in class MYCLASS.

Establishing an external symbol alias

You can instruct the assembler to use an alias for an external symbol in place of the external symbol itself, when it generates the object module. To do this you must code an ALIAS instruction which specifies the external symbol and the alias you want the assembler to use. The external symbol must be defined in a START, CSECT, RSECT, ENTRY, COM, DXD, external DSECT, EXTRN, or WXTRN instruction, or in a V-type address constant.

The following example shows how you use the ALIAS instruction to specify an alias for the external symbol `EXMOD1`.

```
         .
         .
         L     15,EX_SYM        Load address of external symbol
         BASR  14,15            Branch to it
         .
         .
EXMOD1   DS    0H               Using EXMOD1 as a name entry
         .
         .
EX_SYM   DC    V(EXMOD1)        Address of external symbol
EXMOD1   ALIAS C'XMD1PGM'       XMD1PGM is the real external name
         .
         .
```

See "ALIAS instruction" on page 93 for information about the ALIAS instruction.

External symbol dictionary entries

For each section, class, part, entry, external symbol, and dummy external control section, the assembler keeps a record of the following external symbol dictionary (ESD) information:

- Symbolic name, if one is specified
- Type code
- Individual identification number (ESDID)
- Starting address
- Length
- Owning ESDID, if any
- Symbol attributes
- Alias, if one is specified

Table 7 lists the assembler instructions that define control sections and dummy control sections, classes and parts, or identify entry and external symbols, and tells their associated type codes. You can define up to 65535 individual control sections and external symbols in a source module if the NOGOFF option is specified, or up to 999999 external symbols if the GOFF option is specified.

Table 7. Defining external symbols

Name Entry	Instruction	Coding Entered into External Symbol Dictionary	
		NOGOFF option	GOFF option
If present	START, CSECT, or RSECT	SD	SD, ED, LD
If omitted	START, CSECT, or RSECT	PC	SD,ED
Instruction-dependent	Any instruction that initiates the unnamed section	PC	SD

Table 7. Defining external symbols (continued)

Name Entry	Instruction	Coding Entered into External Symbol Dictionary	
If present	COM	CM	SD,ED,CM
If omitted	COM	CM	SD,ED
Optional	DSECT	None	None
Mandatory	DXD or external DSECT	XD	XD
Mandatory	CATTR	Not applicable	ED
Mandatory	CATTR PART(name)	Not applicable	PD
Not applicable	ENTRY	LD	LD
Not applicable	EXTRN	ER	ER
Not applicable	DC (V-type address constant)	ER	ER
Not applicable	WXTRN	WX	WX

See the appendix "Object Deck Output" in the *HLASM Programmer's Guide* for details about the ESD entries produced when you specify the NOGOFF assembler option.

z/VM and z/OS

Refer to *z/OS MVS Program Management: Advanced Facilities*, SA22-7644 for details about the ESD entries produced when you specify the GOFF assembler option.

Summary of source and object program structures

The differences between the load module model and the program object model, and their interactions with assembler language instructions, are summarized in the following table:

Table 8. Object program structure comparison

Property	"Load Module" Model	"Program Object" Model
Form of object program	One-dimensional module	Two-dimensional module
Smallest indivisible independently relocatable component	Control section	Element and part
Residence Mode	Only one	One per class
Addressing Mode	Only one	One per entry point
Compatibility	Can be converted to program object	Can be converted to load module with limitations
Assembler Option	NOGOFF or GOFF	GOFF only
Assembler statements	CSECT, RSECT, START	CSECT, RSECT, START, CATTR, XATTR
Assignable loadable-program attributes	RMODE	RMODE, alignment, load type
External symbol types	SD/CM, LD, ER/WX, PR	SD, ED, LD, ER/WX, PR, PD
External symbol maximum length	8 characters	256 characters

Table 8. Object program structure comparison (continued)

Property	"Load Module" Model	"Program Object" Model
External symbol scope	Module (WX), Library (ER)	Section, Module, Library, Import/Export
External symbol attributes	AMode, RMode	AMode, RMode, scope, PSect name, linkage type, reference type, extended attributes
Object module record types	ESD, TXT, RLD, END, SYM	HDR, ESD, TXT, RLD, END, LEN
Address constant types	A, V, Q, CXD	A, V, Q, J, R, CXD
Binding attributes	Catenate (SD),Merge-like (CM,PR)	Catenate (non-Merge classes), Merge classes (Parts, Pseudo-Registers)
Text types	Byte stream	Byte stream, records (structured and unstructured)
Maximum contiguous text length	16 MB	1 GB

Chapter 4. Machine instruction statements

This chapter introduces a sample of the more common instruction formats and provides general rules for coding them in their symbolic assembler language format.

For the complete specifications of machine instructions, their object code format, their coding specifications, and their use of registers and virtual storage areas, see the applicable *z/Architecture Principles of Operation* manual for your processor. If your program requires vector facility instructions, see the applicable *Vector Operations* manual for the complete specifications of vector-facility instructions.

At assembly time, the assembler converts the symbolic assembler language representation of the machine instructions to the corresponding object code. The computer processes this object code at execution time. Thus, the functions described in this section can be called execution-time functions.

Also at assembly time, the assembler creates the object code of the data constants and reserves storage for the areas you specify in your data definition assembler instructions, such as DC and DS (see Chapter 5, "Assembler instruction statements," on page 83). At execution time, the machine instructions can refer to these constants and areas, but the constants themselves are not normally processed.

As defined in the *z/Architecture Principles of Operation* information. there are five categories of machine instructions:
- General instructions
- Decimal instructions
- Floating-Point instructions
- Control instructions
- Input/Output operations

Each is discussed in the following sections.

General instructions

Use general instructions to manipulate data that resides in general registers or in storage, or that is introduced from the instruction stream. General instructions include fixed-point, logical, and branching instructions. In addition, they include unprivileged status-switching instructions. Some general instructions operate on data that resides in the PSW or the TOD clock.

The general instructions treat data as four types: signed binary integers, unsigned binary integers, unstructured logical data, and decimal data. Data is treated as decimal by the conversion, packing, and unpacking instructions.

For further information, see "General Instructions" in the *z/Architecture Principles of Operation* information.

Decimal instructions

Use the decimal instructions when you want to do arithmetic and editing operations on data that has the binary equivalent of decimal representation.

Decimal data is represented in either zoned or packed format. In the *zoned format*, the rightmost four bits of a byte are called the numeric bits and normally consist of a code representing a decimal digit. The leftmost four bits of a byte are called the zone bits, except for the rightmost byte of a decimal operand; these bits are treated as a zone or a sign.

In the *packed format*, each byte contains two decimal digits, except for the rightmost byte, which contains a sign to the right of a decimal digit.

Decimal instructions treat all numbers as integers. For example, `3.14`, `31.4`, and `314` are all processed as `314`. You must keep track of the decimal point yourself. The integer and scale attributes discussed in "Data attributes" on page 284 can help you do this.

Additional operations on decimal data are provided by several of the instructions in "General Instructions" in the *z/Architecture Principles of Operation* information. Decimal operands always reside in storage.

For further information, see "Decimal Instructions" in the applicable *z/Architecture Principles of Operation* manual.

Floating-point instructions

Use floating-point instructions when you want to do arithmetic operations on data in the floating-point representation. Thus, you do not have to keep track of the decimal point in your computations. Floating-point instructions also let you do arithmetic operations on both large numbers and small numbers, normally providing greater precision than fixed-point decimal instructions.

For further information, see "Floating-Point Instructions" in the *z/Architecture Principles of Operation* information.

Control instructions

Control instructions include all privileged and semiprivileged machine instructions, except the input/output instructions described in "Input/output operations."

Privileged instructions are processed only when the processor is in the supervisor state. An attempt to process an installed privileged instruction in the problem state generates a privileged-operation exception.

Semiprivileged instructions are those instructions that can be processed in the problem state when certain authority requirements are met. An attempt to process an installed semiprivileged instruction in the problem state when the authority requirements are not met generates a privileged-operation exception or some other program-interruption condition depending on the particular requirement that is violated.

For further details, see "Control Instructions" in the *z/Architecture Principles of Operation* information.

Input/output operations

Use the input/output instructions (instead of the IBM-supplied system macro instructions) when you want to control your input and output operations more closely.

The input or output instructions let you identify the channel or the device on which the input or output operation is to be done. For information about how and when you can use these instructions, see the applicable system manual.

For more information, see "Input/Output Operations" in the applicable *z/Architecture Principles of Operation* manual and the applicable system manuals.

Branching with extended mnemonic codes

Branch instructions let you specify an *extended mnemonic code* for the condition on which a branch is to occur. Thus, you avoid having to specify the mask value, that represents the condition code, required by the BC, BCR, and BRC machine instructions. The assembler translates the extended mnemonic code into the mask value, and then assembles it into the object code of the BC, BCR, or BRC machine instruction.

The extended branch mnemonics for the BC instruction require a base register; the extended mnemonics for the BCR and BRC instructions do not. The extended mnemonics for the BRC instruction begin with the letter "J", and are sometimes called "Jump" instructions, as indicated in Figure 14.

Some typical extended mnemonic codes are given in Figure 14. They can be used as operation codes for branching instructions, replacing the BC, BCR, and BRC machine instruction codes (see **1** in Figure 14). The first operand (see **2** in Figure 14) of the BC, BCR, and BRC instructions must not be present in the operand field (see **3** in Figure 14) of the extended mnemonic branching instructions.

For the complete list of branch mnemonics, see the latest edition of *z/Architecture Reference Summary* (SA22-7871).

Extended Code		Meaning	Format	(Symbolic) Machine Instruction Equivalent	
	3 **4**			**1**	**2**
B	$D_2(X_2,B_2)$	Unconditional Branch	RX	BC	$15,D_2(X_2,B_2)$
BR	R_2		RR	BCR	$15,R_2$
J	label	Unconditional Jump	RI	BRC	15,label
NOP	$D_2(X_2,B_2)$	No Operation	RX	BC	$0,D_2(X_2,B_2)$
NOPR	R_2		RR	BCR	$0,R_2$
JNOP	label		RI	BRC	0,label
Used After Compare Instructions					
BH	$D_2(X_2,B_2)$	Branch on High	RX	BC	$2,D_2(X_2,B_2)$
BHR	R_2		RR	BCR	$2,R_2$
JH	label	Jump on High	RI	BRC	2,label
BL	$D_2(X_2,B_2)$	Branch on Low	RX	BC	$4,D_2(X_2,B_2)$
BLR	R_2		RR	BCR	$4,R_2$
JL	label	Jump on Low	RI	BRC	4,label
BE	$D_2(X_2,B_2)$	Branch on Equal	RX	BC	$8,D_2(X_2,B_2)$
BER	R_2		RR	BCR	$8,R_2$
JE	label	Jump on Equal	RI	BRC	8,label
BNH	$D_2(X_2,B_2)$	Branch on Not High	RX	BC	$13,D_2(X_2,B_2)$
BNHR	R_2		RR	BCR	$13,R_2$
JNH	label	Jump on Not High	RI	BRC	13,label
BNL	$D_2(X_2,B_2)$	Branch on Not Low	RX	BC	$11,D_2(X_2,B_2)$
BNLR	R_2		RR	BCR	$11,R_2$
JNL	label	Jump on Not Low	RI	BRC	11,label
BNE	$D_2(X_2,B_2)$	Branch on Not Equal	RX	BC	$7,D_2(X_2,B_2)$
BNER	R_2		RR	BCR	$7,R_2$
JNE	label	Jump on Not Equal	RI	BRC	7,label

Figure 14. Extended mnemonic codes (part 1 of 5)

Used After Arithmetic Instructions

BP	$D_2(X_2,B_2)$	Branch on Plus	RX	BC	$2,D_2(X_2,B_2)$
BPR	R_2		RR	BCR	$2,R_2$
JP	label	Jump on Plus	RI	BRC	2,label
BM	$D_2(X_2,B_2)$	Branch on Minus	RX	BC	$4,D_2(X_2,B_2)$
BMR	R_2		RR	BCR	$4,R_2$
JM	label	Jump on Minus	RI	BRC	4,label
BZ	$D_2(X_2,B_2)$	Branch on Zero	RX	BC	$8,D_2(X_2,B_2)$
BZR	R_2		RR	BCR	$8,R_2$
JZ	label	Jump on Zero	RI	BRC	8,label
BO	$D_2(X_2,B_2)$	Branch on Overflow	RX	BC	$1,D_2(X_2,B_2)$
BOR	R_2		RR	BCR	$1,R_2$
JO	label	Jump on Overflow	RI	BRC	1,label
BNP	$D_2(X_2,B_2)$	Branch on Not Plus	RX	BC	$13,D_2(X_2,B_2)$
BNPR	R_2		RR	BCR	$13,R_2$
JNP	label	Jump on Not Plus	RI	BRC	13,label
BNM	$D_2(X_2,B_2)$	Branch on Not Minus	RX	BC	$11,D_2(X_2,B_2)$
BNMR	R_2		RR	BCR	$11,R_2$
JNM	label	Jump on Not Minus	RI	BRC	11,label
BNZ	$D_2(X_2,B_2)$	Branch on Not Zero	RX	BC	$7,D_2(X_2,B_2)$
BNZR	R_2		RR	BCR	$7,R_2$
JNZ	label	Jump on Not Minus	RI	BRC	7,label
BNO	$D_2(X_2,B_2)$	Branch on No Overflow	RX	BC	$14,D_2(X_2,B_2)$
BNOR	R_2		RR	BCR	$14,R_2$
JNO	label	Jump on No Overflow	RI	BRC	14,label

Figure 15. Extended mnemonic codes (part 2 of 5)

Used After Test Under Mask Instructions

BO	$D_2(X_2,B_2)$	Branch if Ones	RX	BC	$1,D_2(X_2,B_2)$
BOR	R_2		RR	BCR	$1,R_2$
BM	$D_2(X_2,B_2)$	Branch if Mixed	RX	BC	$4,D_2(X_2,B_2)$
BMR	R_2		RR	BCR	$4,R_2$
BZ	$D_2(X_2,B_2)$	Branch if Zero	RX	BC	$8,D_2(X_2,B_2)$
BZR	R_2		RR	BCR	$8,R_2$
BNO	$D_2(X_2,B_2)$	Branch if Not Ones	RX	BC	$14,D_2(X_2,B_2)$
BNOR	R_2		RR	BCR	$14,R_2$
BNM	$D_2(X_2,B_2)$	Branch if Not Mixed	RX	BC	$11,D_2(X_2,B_2)$
BNMR	R_2		RR	BCR	$11,R_2$
BNZ	$D_2(X_2,B_2)$	Branch if Not Zero	RX	BC	$7,D_2(X_2,B_2)$
BNZR	R_2		RR	BCR	$7,R_2$

Branch Relative on Condition Long

BRUL	label	Unconditional Br Rel Long	RIL	BRCL	15,label
BRHL	label	Br Rel Long on High	RIL	BRCL	2,label
BRLL	label	Br Rel Long on Low	RIL	BRCL	4,label
BREL	label	Br Rel Long on Equal	RIL	BRCL	8,label
BRNHL	label	Br Rel Long on Not High	RIL	BRCL	13,label
BRNLL	label	Br Rel Long on Not Low	RIL	BRCL	11,label
BRNEL	label	Br Rel Long on Not Equal	RIL	BRCL	7,label
BRPL	label	Br Rel Long on Plus	RIL	BRCL	2,label
BRML	label	Br Rel Long on Minus	RIL	BRCL	4,label
BRZL	label	Br Rel Long on Zero	RIL	BRCL	8,label
BROL	label	Br Rel Long on Overflow	RIL	BRCL	1,label
BRNPL	label	Br Rel Long on Not Plus	RIL	BRCL	13,label
BRNML	label	Br Rel Long on Not Minus	RIL	BRCL	11,label
BRNZL	label	Br Rel Long on Not Zero	RIL	BRCL	7,label
BRNOL	label	Br Rel Long on Not Overflow	RIL	BRCL	14,label

Figure 16. Extended mnemonic codes (part 3 of 5)

```
Branch Relative on Condition
BRO    label              Branch on Overflow          RI          BRC   1,label
BRP    label              Branch on Plus              RI          BRC   2,label
BRH    label              Branch on High              RI          BRC   2,label
BRL    label              Branch on Low               RI          BRC   4,label
BRM    label              Branch on Minus             RI          BRC   4,label
BRNE   label              Branch on Not Equal         RI          BRC   7,label
BRNZ   label              Branch on Not Minus         RI          BRC   7,label
BRE    label              Branch on Equal             RI          BRC   8,label
BRZ    label              Branch on Zero              RI          BRC   8,label
BRNL   label              Branch on Not Low           RI          BRC   11,label
BRNM   label              Branch on Not Minus         RI          BRC   11,label
BRNH   label              Branch on Not High          RI          BRC   13,label
BRNP   label              Branch on Not Plus          RI          BRC   13,label
BRNO   label              Branch on No Overflow       RI          BRC   14,label
BRU    label              Unconditional Branch        RI          BRC   15,label
```

Figure 17. Extended mnemonic codes (part 4 of 5)

```
Jump on Condition Long

JLU    label              Unconditional Jump Long     RIL         BRCL  15,label
JLNOP  label              No operation                RIL         BRCL  0,label

JLH    label              Jump Long on High           RIL         BRCL  2,label
JLL    label              Jump Long on Low            RIL         BRCL  4,label
JLE    label              Jump Long on Equal          RIL         BRCL  8,label
JLNH   label              Jump Long on Not High       RIL         BRCL  13,label
JLNL   label              Jump Long on Not Low        RIL         BRCL  11,label
JLNE   label              Jump Long on Not Equal      RIL         BRCL  7,label

JLP    label              Jump Long on Plus           RIL         BRCL  2,label
JLM    label              Jump Long on Minus          RIL         BRCL  4,label
JLZ    label              Jump Long on Zero           RIL         BRCL  8,label
JLO    label              Jump Long on Overflow       RIL         BRCL  1,label
JLNP   label              Jump Long on Not Plus       RIL         BRCL  13,label
JLNM   label              Jump Long on Not Minus      RIL         BRCL  11,label
JLNZ   label              Jump Long on Not Zero       RIL         BRCL  7,label
JLNO   label              Jump Long on Not Overflow   RIL         BRCL  14,label
```

Notes:

1. D_2=displacement, X_2=index register, B_2=base register, R_2=register containing branch address
2. The addresses represented are explicit address (see **4**). However, implicit addresses can also be used in this type of instruction.
3. Avoid using BM, BNM, JM, and JNM after the TMH, TML,TMHH, TMHL, TMLH or TMLL instruction.

Figure 18. Extended mnemonic codes (part 5 of 5)

Alternative mnemonics for some branch relative instructions

For some branch relative statements, there are alternative mnemonics. These are:

Table 9. Alternative mnemonics for some branch relative instructions

Instruction	Alternative	Description
BRAS	JAS	Branch Relative and Save
BRASL	JASL	Branch Relative and Save Long
BRCT	JCT	Branch Relative on Count
BRCTG	JCTG	Branch Relative on Count
BRXH	JXH	Branch Relative on Index High
BRXHG	JXHG	Branch Relative on Index High
BRXLE	JXLE	Branch Rel. on Index Low or Equal

Table 9. Alternative mnemonics for some branch relative instructions (continued)

Instruction	Alternative	Description
BRXLG	JXLEG	Branch Rel. on Index Low or Equal

Statement formats

Machine instructions are assembled into 2, 4, or 6 bytes of object code according to the format of each instruction. Machine instruction formats include the following (ordered by length attribute):

Length Attribute Basic Formats
2 RR
4 RI, RS, RSI, RX, SI
6 SS

See the *z/Architecture Principles of Operation* information. for complete details about machine instruction formats. See also "Examples of coded machine instructions" on page 76.

When you code machine instructions, you use symbolic formats that correspond to the actual machine language formats. Within each basic format, you can also code variations of the symbolic representation, divided into groups according to the basic formats shown in "Examples of coded machine instructions" on page 76.

The assembler converts only the operation code and the operand entries of the assembler language statement into object code. The assembler assigns to a name entry symbol the value of the address of the first byte of the assembled instruction. When you use this same symbol in the operand of an assembler language statement, the assembler uses this address value in converting the symbolic operand into its object code form. The length attribute assigned to the symbol depends on the basic machine language format of the instruction in which the symbol appears as a name entry.

A remarks entry is not converted into object code.

An example of a typical assembler language statement follows:

```
LABEL    L              4,256(5,10)         LOAD INTO REG4
```

where:
LABEL Is the name entry
L Is the operation code mnemonic (converted to hex 58)
4 Is the register operand (converted to hex 4)
256(5,10)
Are the storage operand entries (converted to hex 5A100)
LOAD INTO REG4
Are remarks not converted into object code

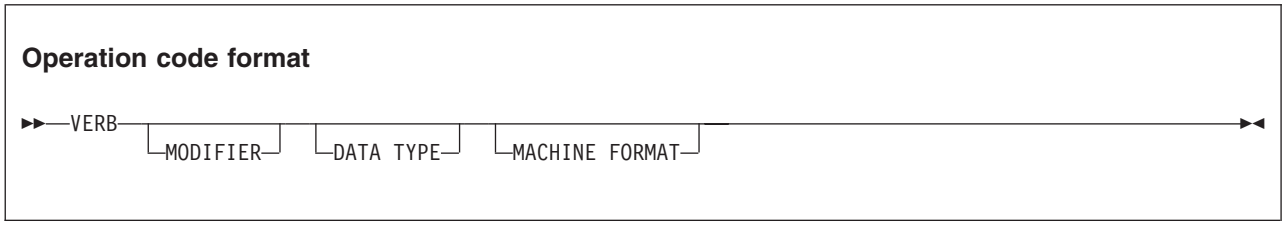
The object code of the assembled instruction, in hexadecimal, is:

```
5845A100 (4 bytes in RX format)
```

Symbolic operation codes

You must specify an operation code for each machine instruction statement. The symbolic operation code, or mnemonic code as it is also called, indicates the type of operation to be done; for example, A indicates the addition operation. Refer to the *z/Architecture Principles of Operation* information for a complete list of symbolic operation codes and the formats of the corresponding machine instructions.

The general format of the machine instruction operation code is:



Verb The verb must always be present. It typically consists of one or two characters and specifies the operation to be done. The verb is underscored in the following examples:

```
A    3,AREA         A indicates an add operation
MVC  TO,FROM        MV indicates a move operation
```

The other items in the operation code are not always present. They include the following (underscores are used to indicate modifiers, data types, and machine formats in the following examples):

Modifier

Modifier, which further defines the operation:

```
AL  3,AREA          L indicates a logical operation
```

Data Type

Type qualifier, which indicates the type of data used by the instruction in its operation:

```
CVB 3,BINAREA       B indicates binary data

MVC TO,FROM         C indicates character data

AE  2,FLTSHRT       E indicates normalized short
                    floating-point data

AD  2,FLTLONG       D indicates normalized long
                    floating-point data
```

Machine Format

Format qualifier, R indicating a register operand, or I indicating an immediate operand. For example:

```
ADR 2,4             R indicates a register operand

MVI FIELD,X'A1'     I indicates an immediate operand
AHI 7,123
```

Operand entries

Specify one or more operands in each machine instruction statement to provide the data or the location of the data upon which the machine operation is to be done. The operand entries consist of one or more fields or subfields, depending on the format of the instruction being coded. They can specify a register, an address, a length, or immediate data. You can omit length fields or subfields, which the assembler computes for you from the other operand entries. You can code an operand entry either with symbols or with self-defining terms.

The rules for coding operand entries are:

- A comma must separate operands.
- Parentheses must enclose subfields.
- A comma must separate subfields enclosed in parentheses.
- If a subfield is omitted because it is implicit in a symbolic address, the parentheses that enclosed the subfield must be omitted.
- If two subfields are enclosed in parentheses and separated by commas, the following applies:

  - If both subfields are omitted because they are implicit in a symbolic entry, the separating comma and the parentheses that were needed must also be omitted.
  - If the first subfield is omitted, the comma that separates it from the second subfield must be written, as well as the enclosing parentheses.
  - If the second subfield is omitted, the comma that separates it from the first subfield must be omitted; however, the enclosing parentheses must be written.
- Spaces must not appear within the operand field, except as part of a character self-defining term, or in the specification of a character literal.

Registers

You can specify a register in an operand for use as an arithmetic accumulator, a base register, an index register, and as a general depository for data to which you want to refer repeatedly.

You must be careful when specifying a register whose contents have been affected by the execution of another machine instruction, the control program, or an IBM-supplied system macro instruction.

For some machine instructions, you are limited in which registers you can specify in an operand.

The expressions used to specify registers must have absolute values; in general, registers 0 through 15 can be specified for machine instructions. However, the following restrictions on register usage apply:

- The even-numbered registers must be specified for the following groups of instructions:
  - The double-shift instructions
  - Most multiply and divide instructions
  - The move long and compare logical long instructions
- If the NOAFPR ACONTROL operand is specified, then only the floating-point registers (0, 2, 4, or 6) can be specified for floating-point instructions.
- If the AFPR ACONTROL operand is specified, then one of the floating-point registers 0, 1, 4, 5, 8, 9, 12, or 13 can be specified for the instructions that use extended floating-point data in pairs of registers, such as AXR, SXR, LTXBR, and SQEBR.
- If the NOAFPR ACONTROL operand is specified, then either floating-point register 0 or 4 must be specified for these instructions.
- For a processor with a vector facility, the even-numbered vector registers (0, 2, 4, 6, 8, 10, 12, 14) must be specified in vector-facility instructions that are used to manipulate long floating-point data or 64 bit signed binary data in vector registers.

The assembler checks the registers specified in the instruction statements of the above groups. If the specified register does not comply with the stated restrictions, the assembler issues a diagnostic message and does not assemble the instruction. Binary zeros are generated in place of the machine code.

Register usage by machine instructions

Registers that are not explicitly coded in symbolic assembler language representation of machine instructions, but are nevertheless used by assembled machine instructions, are divided into two categories:

- Base registers that are implicit in the symbolic addresses specified. (See “Addresses” on page 73.) The registers can be identified by examining the object code or the USING instructions that assign base registers for the source module.
- Registers that are used by machine instructions, but do not appear in assembled object code.
  - For double shift and fullword multiply and divide instructions, the odd-numbered register, whose number is one greater than the even-numbered register specified as the first operand.
  - For Move Long and Compare Logical Long instructions, the odd-numbered registers, whose number is one greater than even-numbered registers specified in the two operands.

- For Branch on Index High (BXH) and the Branch on Index Low or Equal (BXLE) instructions, if the register specified for the second operand is an even-numbered register, the next higher odd-numbered register is used to contain the value to be used for comparison.
- For Load Multiple (LM, LAM) and Store Multiple (STM, STAM) instructions, the registers that lie between the registers specified in the first two operands.
- For extended-precision floating point instructions, the second register of the register pair.
- For Compare and Form Codeword (CFC) instruction, registers 1, 2, and 3 are used.
- For Translate and Test (TRT) instruction, registers 1 and 2 are used.
- For Update Tree (UPT) instruction, registers 0-5 are used.
- For Edit and Mark (EDMK) instruction, register 1 is used.
- For certain control instructions, one or more of registers 0-4 and register 14 are used. See "Control Instructions" in the applicable *z/Architecture Principles of Operation* manual.
- For certain input and output instructions, either or both registers 1 and 2 are used. See "Input/Output Instructions" in the applicable *z/Architecture Principles of Operation* manual.
- On a processor with a vector facility:
  1. For instructions that manipulate long floating-point data in vector registers, the odd-numbered vector registers, whose number is one greater than the even-numbered vector registers specified in each operand.
  2. For instructions that manipulate 64 bit signed binary data in vector registers, the odd-numbered vector registers, whose number is one greater than the even-numbered vector registers specified in each operand.

Register usage by system

The programming interface of the system control programs uses registers 0, 1, 13, 14, and 15.

Addresses

You can code a symbol in the name field of a machine instruction statement to represent the address of that instruction. You can then refer to the symbol in the operands of other machine instruction statements. The object code requires that addresses be assembled in a numeric relative-offset or base-displacement format. This format lets you specify addresses that are relocatable or absolute. Chapter 3, "Program structures and addressing," on page 43 describes how you use symbolic addresses to refer to data in your assembler language program.

Defining Symbolic Addresses: Define relocatable addresses by either using a symbol as the label in the name field of an assembler language statement, or equating a symbol to a relocatable expression.

Define absolute addresses (or values) by equating a symbol to an absolute expression.

Referring to Addresses: You can refer to relocatable and absolute addresses in the operands of machine instruction statements. (Such address references are also called addresses in this manual.) The two ways of coding addresses are:

- Implicitly—in a form that the assembler must first convert into an explicit relative-offset or base-displacement form before it can be assembled into object code.
- Explicitly—in a form that can be directly assembled into object code.

Implicit address

An implicit address is specified by coding one expression. The expression can be relocatable or absolute. The assembler converts all implicit addresses into their relative-offset or base-displacement form before it assembles them into object code. The assembler converts implicit addresses into explicit base-displacement addresses only if a USING instruction has been specified, or for small absolute expressions, where the address is resolved without a USING. The USING instruction assigns both a base address, from which the assembler computes displacements, and a base register, which is assumed to

contain the base address. The base register must be loaded with the correct base address at execution time. For more information, refer to “Addressing” on page 54.

Explicit address

An explicit address is specified by coding two absolute expressions as follows:

- The first is an absolute expression for the displacement, whose value must lie in the range 0 through 4095 (4095 is the maximum value that can be represented by the 12 binary bits available for the displacement in the object code), or in the range -524,288 to 524,287 for long-displacement instructions.
- The second (enclosed in parentheses) is an absolute expression for the base register, whose value must lie in the range 0 through 15.

An explicit base register designation must not accompany an implicit address. However, in RX-format instructions, an index register can be coded with an implicit address as well as with an explicit address. When two addresses are required, each address can be coded as an explicit address or as an implicit address.

Relative address

A relative address is specified by coding one expression. The expression is relocatable or absolute. If a relocatable expression is used, then the assembler converts the value to a signed number of halfwords relative to the current location counter, and then uses that value in the object code. An absolute value can be used for a relative address, but the assembler issues a warning message, as it uses the supplied value, and this might cause unpredictable results.

Relocatability of addresses

If the value of an address expression changes when the assumed origin of the program is changed, and changes by the same amount, then the address is “simply relocatable”. If the addressing expression does not change when the assumed origin of the program is changed, then that address is “absolute”. If the addressing expression changes by some other amount, the address is “complexly relocatable”.

Addresses in the relative-offset or base-displacement form are relocatable, because:

- Each relocatable address is assembled as a signed relative offset from the instruction, or as a displacement from a base address and a base register.
- The base register contains the base address.
- If the object module assembled from your source module is relocated, only the contents of the base register need reflect this relocation. This means that the location in virtual storage of your base has changed, and that your base register must contain this new base address.
- Addresses in your program have been assembled as relative to the base address; therefore, the sum of the displacement and the contents of the base register point to the correct address after relocation.

Absolute addresses are also assembled in the base-displacement form, but always indicate a fixed location in virtual storage. This means that the contents of the base register must always be a fixed absolute address value regardless of relocation.

Machine or object code format

Addresses assembled into the object code of machine instructions have the format given in Figure 19 on page 75. Not all the instruction formats are shown in Figure 19 on page 75.

The addresses represented have a value that is the sum of a displacement (see **1** in Figure 19 on page 75) and the contents of a base register (see **2** in Figure 19 on page 75).

Index register: In RX-format instructions, the address represented has a value that is the sum of a displacement, the contents of a base register, and the contents of an index register (see **3** in Figure 19 on page 75).

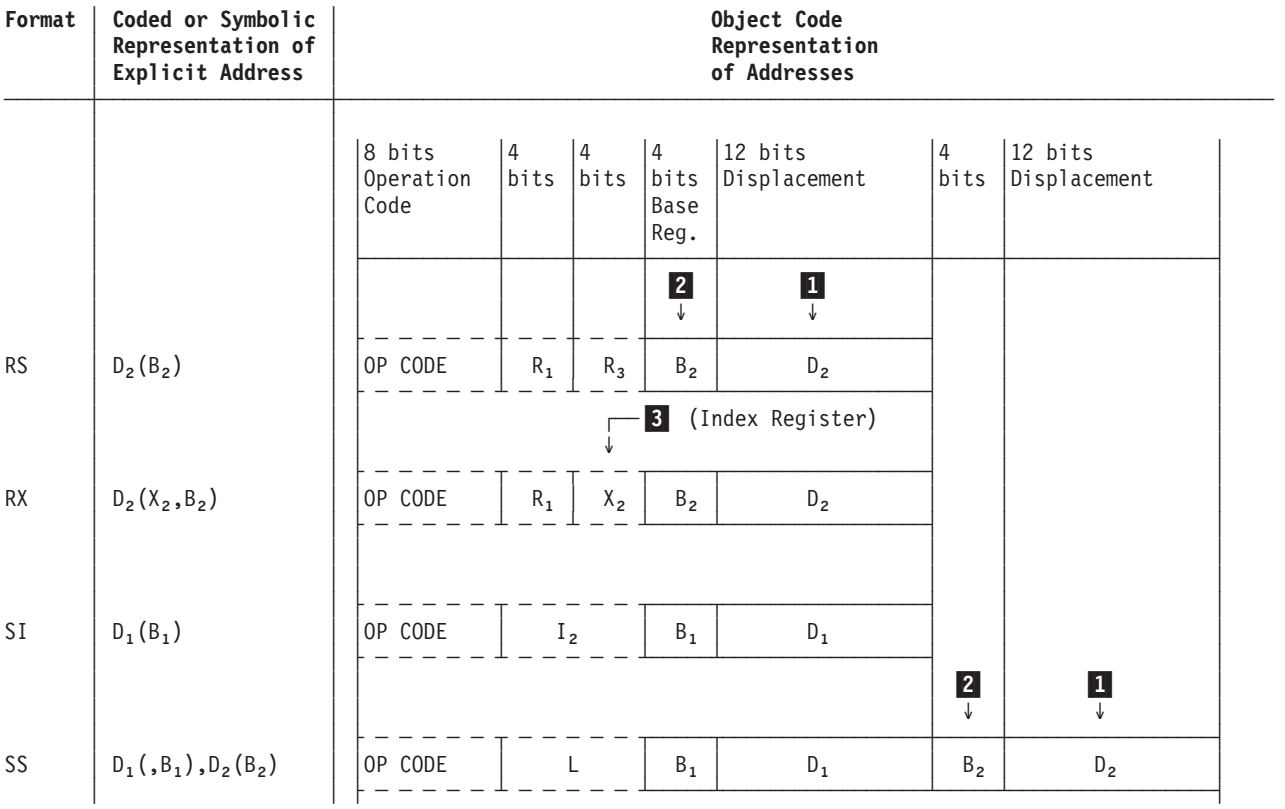


Figure 19. Format of addresses in object code

Lengths

You can specify the length field in an SS-format instruction. This lets you indicate explicitly the number of bytes of data at a virtual storage location that is to be used by the instruction. However, you can omit the length specification, because the assembler computes the number of bytes of data to be used from the expression that represents the address of the data.

See page "SS format" on page 81 for more information about SS-format instructions.

Implicit Length

When a length subfield is omitted from an SS-format machine instruction, an implicit length is assembled into the object code of the instruction. The implicit length is either of the following:

- For an *implicit address*, it is the length attribute of the first or only term in the expression representing the *implicit address*.
- For an *explicit address*, it is the length attribute of the first or only term in the expression representing the *displacement*.

Explicit Length

When a length subfield is specified in an SS-format machine instruction, the explicit length always overrides the implicit length.

An implicit or explicit length is the *effective length*. The length value assembled is always one less than the effective length. If you want an assembled length value of 0, an explicit length of 0 or 1 can be specified.

In the SS-format instructions requiring one length value, the allowable range for explicit lengths is 0 through 256. In the SS-format instructions requiring two length values, the allowable range for explicit lengths is 0 through 16.

Immediate data

In addition to registers, numeric values, relative addresses, and lengths, some machine instruction operands require immediate data. Such data is assembled directly into the object code of the machine instructions. Use immediate data to specify the bit patterns for masks or other absolute values you need.

Specify immediate data only where it is required. Do not confuse it with address references to constants and areas, or with any literals you specify as the operands of machine instructions.

Immediate data must be specified as absolute expressions whose range of values depends on the machine instruction for which the data is required. The immediate data is assembled into its binary representation.

Examples of coded machine instructions

The examples that follow are grouped according to machine instruction format. They show the various ways in which you can code the operands of machine instructions. Both symbolic and numeric representation of fields and subfields are shown in the examples. Therefore, assume that all symbols used are defined elsewhere in the same source module.

The object code assembled from at least one coded statement per group is also included. A complete summary of machine instruction formats with the coded assembler language variants can be found in the *z/Architecture Principles of Operation*, SA22-7832 (and also in the *z/Architecture Reference Summary*, SA22-7871). These two documents provide the definitive reference to machine instruction formats.

The examples that follow show the various instruction formats.

RI format

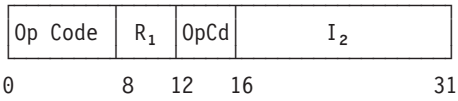
The operand fields of RI-format instructions designate a register and an immediate operand, with the following exception:

- In BRC branching instructions, a 4 bit branching mask with a value 0 - 15 replaces the register designation.

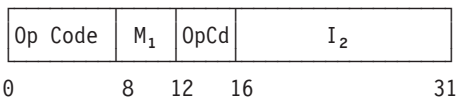
Symbols used to represent registers (such as REG1 in the example) are assumed to be equated to absolute values 0 - 15. The 16 bit immediate operand has two different interpretations, depending on whether the instruction is a branching instruction or not.

There are two types of non-branching RI-format instructions.

- For most, the immediate value is treated as a signed binary integer (-32768 - +32767). This value can be specified by any absolute expression.



- For logical instructions such as TMH, the immediate field is a 16 bit mask.



Examples:

```
ALPHA1   AHI            REG1,2000
ALPHA2   MHI            3,1234
BETA1    TMH            7,X'8001'
```

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

```
A7708001
```

where:

A7.0 Is the operation code
7 Is register R_1
8001 Is the immediate data I2

For branching RI-format instructions, the immediate value is treated as a signed binary integer representing the number of halfwords to branch relative to the current location.

The branch target can be specified as a relocatable expression, in which case the assembler performs some checking, and calculates the immediate value.

The branch target can also be specified as an absolute value, in which case the assembler issues a warning before it assembles the instruction.

Examples:

```
ALPHA1   BRAS           1,BETA1
ALPHA2   BRC            3,ALPHA1
BETA1    BRCT           7,ALPHA1
```

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

```
A776FFFC
```

where:

A7.6 Is the operation code
7 Is register R_1
FFFC Is the immediate data I2; a value of -4 decimal

If the GOFF assembler option is active, then it is possible to specify the target address as one or more external symbols (with or without offsets).

If an offset is specified it can be specified as an absolute or relocatable expression. If the offset is specified as a relocatable expression, the assembler performs some checking and calculates the immediate value. If the offset is an absolute expression the assembler issues warning message ASMA056W.

Examples:

```
ALPHA1 BRAS    14,A-B+C+10  where A, B and C are external symbols
ALPHA2 BRASL   14,A-B+C+10
BETA1  BRC     15,A-B+C+10
```

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is

```
A7F40005
```

where:

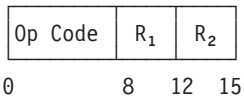
```
A7.4 is the operation code
F    is the condition code
0005 is the immediate data I2; a value of 5 decimal.
```

In addition GOFF Relocation Dictionary Data Items are generated for the external symbols A, B, and C.

RR format

The operand fields of RR-format instructions designate two registers, with the following exceptions:

- In BCR branching instructions, when a 4 bit branching mask replaces the first register specification (see 8 in the instruction labeled GAMMA1 in the examples).
- In SVC instructions, where an immediate value (0 - 255) replaces both registers (see 200 in the instruction labeled DELTA1 in the examples).



Symbols used to represent registers in RR-format instructions (see INDEX and REG2 in the instruction labeled ALPHA2 in the examples) are assumed to be equated to absolute values 0 - 15.

Symbols used to represent immediate values in SVC instructions (see TEN in the instruction labeled DELTA2 in the examples) are assumed to be equated to absolute values 0 - 255.

Examples:

```
ALPHA1   LR              1,2
ALPHA2   LR              INDEX,REG2
GAMMA1   BCR             8,12
DELTA1   SVC             200
DELTA2   SVC             TEN
```

When assembled, the object code of the instruction labeled ALPHA1, in hexadecimal, is:

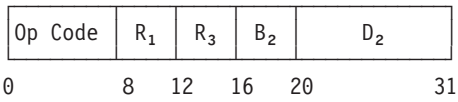
```
1812
```

where:

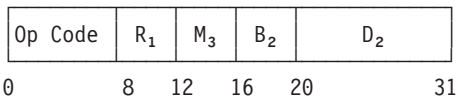
18 Is the operation code
1 Is register R_1
2 Is register R_2

RS format

The operand fields of RS-format instructions designate two registers, and a virtual storage address (coded as an implicit address or an explicit address).



In the Insert Characters under Mask (ICM) and the Store Characters under Mask (STCM) instructions, a 4 bit mask (see X'E' and MASK in the instructions labeled DELTA1 and DELTA2 in the examples), with a value 0 - 15, replaces the second register specifications.



Symbols used to represent registers (see REG4, REG6, and BASE in the instruction labeled ALPHA2 in the examples) are assumed to be equated to absolute values 0 - 15.

Symbols used to represent implicit addresses (see AREA and IMPLICIT in the instructions labeled BETA1 and DELTA2 in the examples) can be either relocatable or absolute.

Symbols used to represent displacements (see DISPL in the instruction labeled BETA2 in the examples) in explicit addresses are assumed to be equated to absolute values 0 - 4095.

Many other instruction formats are supported by the High Level Assembler. For complete information see the latest editions of *z/Architecture Principles of Operation*, SA22-7832 and the *z/Architecture Reference Summary*, SA22-7871.

Examples:

```
ALPHA1   LM              4,6,20(12)
ALPHA2   LM              REG4,REG6,20(BASE)
BETA1    STM             4,6,AREA
BETA2    STM             4,6,DISPL(BASE)
GAMMA1   SLL             2,15
DELTA1   ICM             3,X'E',1024(10)
DELTA2   ICM             REG3,MASK,IMPLICIT
```

When assembled, the object code for the instruction labeled `ALPHA1`, in hexadecimal, is:

```
9846C014
```

where:

98 Is the operation code
4 Is register R_1
6 Is register R_3
C Is base register B_1
014 Is displacement D_1 from base register B_1

When assembled, the object code for the instruction labeled `DELTA1`, in hexadecimal, is:

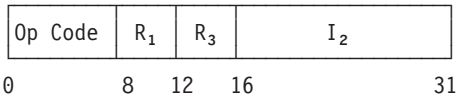
```
BF3EA400
```

where:

BF Is the operation code
3 Is register R_1
E Is mask M_3
A Is base register B_1
400 Is displacement D_1 from base register B_1

RSI format

The operand fields of RSI-format instructions designate two registers and a 16 bit immediate operand.



Symbols used to represent registers (see REG1 in the examples) are assumed to be equated to absolute values 0 - 15.

The immediate value is treated as a signed binary integer representing the number of halfwords to branch relative to the current location.

The branch target can be specified as a label in which case the assembler calculates the immediate value and performs some checking of the value.

The branch target can also be specified as an absolute value in which case the assembler issues a warning before it assembles the instruction.

Examples:

```
ALPHA1   BRXH            REG1,REG3,BETA1
BETA1    BRXLE           1,2,ALPHA1
```

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is

```
84130002
```

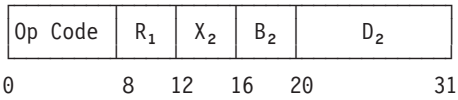
where:

84 Is the operation code
1 Is register REG1
3 Is register REG3
0002 Is the immediate data I2

RX format

The operand fields of RX-format instructions designate one or two registers, including an index register, and a virtual storage address (coded as an implicit address or an explicit address), with the following exception:

> In BC branching instructions, a 4 bit branching mask (see 7 and TEN in the instructions labeled LAMBDAn in the examples) with a value 0 - 15, replaces the first register specification.



Symbols used to represent registers (see REG1, INDEX, and BASE in the ALPHA2 instruction in the examples) are assumed to be equated to absolute values 0 - 15.

Symbols used to represent implicit addresses (see IMPLICIT in the instructions labeled GAMMAn in the examples) can be either relocatable or absolute.

Symbols used to represent displacements (see DISPL in the instructions labeled BETA2 and LAMBDA1 in the examples) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

Examples:

```
ALPHA1   L               1,200(4,10)
ALPHA2   L               REG1,200(INDEX,BASE)
BETA1    L               2,200(,10)
BETA2    L               REG2,DISPL(,BASE)
GAMMA1   L               3,IMPLICIT
GAMMA2   L               3,IMPLICIT(INDEX)
DELTA1   L               4,=F'33'
LAMBDA1  BC              7,DISPL(,BASE)
LAMBDA2  BC              TEN,ADDRESS
```

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:

```
5814A0C8
```

where:

58 Is the operation code
1 Is register R_1
4 Is index register X_2
A Is base register B_2
0C8 Is displacement D_2 from base register B_2

When assembled, the object code for the instruction labeled GAMMA1, in hexadecimal, is:

```
5824xyyy
```

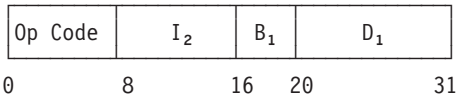
where:

58 Is the operation code

2 Is register R_1
4 Is the index register X_2
x Is base register B_2
yyy Is displacement D_2 from base register B_2

SI format

The operand fields of SI-format instructions designate immediate data and a virtual storage address.



Symbols used to represent immediate data (see `HEX40` and `TEN` in the instructions labeled `ALPHA2` and `BETA1` in the examples) are assumed to be equated to absolute values 0 - 255.

Symbols used to represent implicit addresses (see `IMPLICIT` and `KEY` in the instructions labeled `BETA1` and `BETA2`) can be either relocatable or absolute.

Symbols used to represent displacements (see `DISPL40` in the instruction labeled `ALPHA2` in the examples) in explicit addresses are assumed to be equated to absolute values 0 - 4095.

Examples:

```
ALPHA1   CLI             40(9),X'40'
ALPHA2   CLI             DISPL40(NINE),HEX40
BETA1    CLI             IMPLICIT,TEN
BETA2    CLI             KEY,C'E'
```

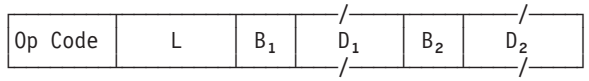
When assembled, the object code for the instruction labeled `ALPHA1`, in hexadecimal, is:

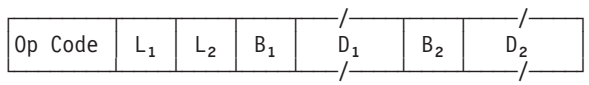
```
95409028
```

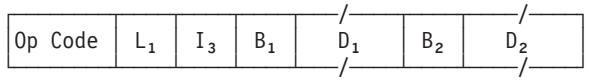
where
95 Is the operation code.
40 Is the immediate data.
9 Is the base register.
028 Is the displacement from the base register

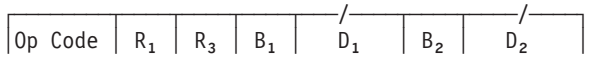
SS format

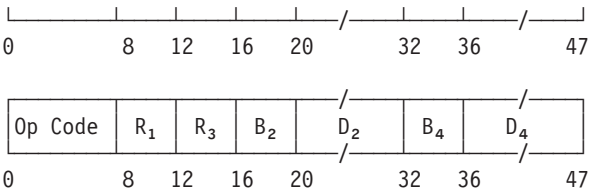
The operand fields and subfields of SS-format instructions designate two virtual storage addresses (coded as implicit addresses or explicit addresses) and, optionally, the explicit data lengths you want to include. However, in the Shift and Round Decimal (SRP) instruction, a 4 bit immediate data field (see the operand 3 in the example of an SRP instruction), with a value 0 - 9, is specified as a third operand.











Symbols used to represent base registers (see BASE8 and BASE7 in the instruction labeled ALPHA2 in the examples) in explicit addresses are assumed to be equated to absolute values 0 - 15.

Symbols used to represent explicit lengths (see NINE and SIX in the instruction labeled ALPHA2 in the examples) are assumed to be equated to absolute values 0 - 256 for SS-format instructions with one length specification, and 0 - 16 for SS-format instructions with two length specifications.

Symbols used to represent implicit addresses (see FIELD1 and FIELD2 in the instruction labeled ALPHA3, and FIELD1,X'8' in the SRP instructions in the examples) can be either relocatable or absolute.

Symbols used to represent displacements (see DISP40 and DISP30 in the instruction labeled ALPHA5 in the examples) in explicit addresses are assumed to be equated to absolute values 0 - 4095.

See page "Lengths" on page 75 for more information about the lengths of SS-format instructions.

Examples:

```
ALPHA1   AP             40(9,8),30(6,7)
ALPHA2   AP             40(NINE,BASE8),30(SIX,BASE7)
ALPHA3   AP             FIELD1,FIELD2
ALPHA4   AP             AREA(9),AREA2(6)
ALPHA5   AP             DISP40(,8),DISP30(,7)
BETA1    MVC            0(80,8),0(7)
BETA2    MVC            DISP0(,8),DISP0(7)
BETA3    MVC            TO,FROM
         SRP            FIELD1,X'8',3
```

When assembled, the object code for the instruction labeled ALPHA1, in hexadecimal, is:

```
FA858028701E
```

where:

FA Is the operation code.
8 Is length L_1
5 Is length L_2
8 Is base register B_1
028 Is displacement D_1 from base register B_1
7 Is base register B_2
01E Is displacement D_2 from base register B_2

When assembled, the object code for the instruction labeled BETA1, in hexadecimal, is:

```
D24F80007000
```

where:

D2 Is the operation code
4F Is length L
8 Is base register B_1
000 Is displacement D_1 from base register B_1
7 Is base register B_2
000 Is displacement D_2 from base register B_2

Chapter 5. Assembler instruction statements

This chapter describes, in detail, the syntax and usage rules of each assembler instruction. There is also information about assembly instructions on "Conditional assembly instructions" on page 224. The following table lists the assembler instructions by type, and provides the number of the page where the instruction is described.

Table 10. Assembler instructions

Type of Instruction	Instruction	Page No.
Program Control	AINSERT	92
	CNOP	102
	COPY	105
	END	160
	EXITCTL	166
	ICTL	168
	ISEQ	168
	LTORG	171
	ORG	177
	POP	180
	PUNCH	184
	PUSH	185
	REPRO	186
Listing Control	CEJECT	101
	EJECT	160
	PRINT	181
	SPACE	189
	TITLE	190
Operation Code Definition	OPSYN	175

Table 10. Assembler instructions (continued)

Type of Instruction	Instruction	Page No.
Program Section and Linking	ALIAS	93
	AMODE	95
	CATTR (z/OS and CMS)	96
	COM	104
	CSECT	106
	CXD	108
	DSECT	157
	DXD	159
	ENTRY	162
	EXTRN	167
	LOCTR	169
	RMODE	187
	RSECT	188
	START	189
	WXTRN	202
	XATTR (z/OS and CMS)	203
Base Register	DROP	152
	USING	193
Data Definition	CCW	99
	CCW0	99
	CCW1	100
	DC	109
	DS	154
Symbol Definition	EQU	162
Associated Data	ADATA	92
Assembler Options	*PROCESS	84
	ACONTROL	85

64 bit addressing mode

Some instructions have an operand or operands that pertain to 64 bit addressing mode (for example, `64` for `AMODE`). This operand is accepted and processed by the assembler. However, other operating system components and utility programs might not be able to accept and process information related to this operand.

*PROCESS statement

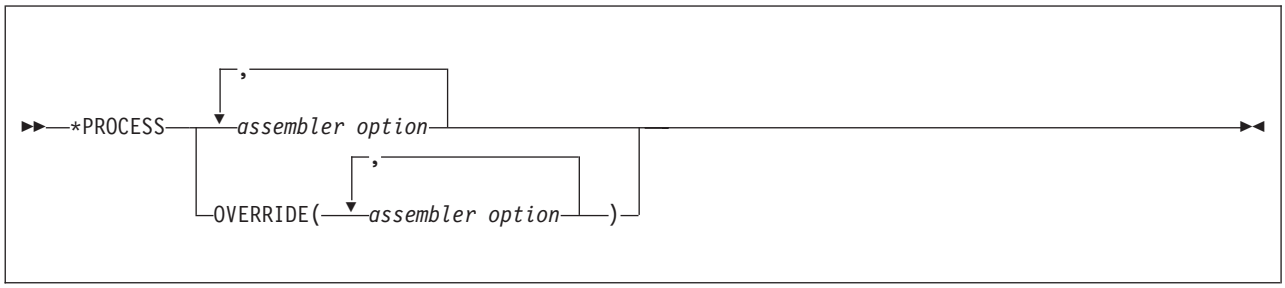
Process (*PROCESS) statements specify assembler options in an assembler source program. You can include them in the primary input data set or provide them from a SOURCE user exit.

To ensure that certain assembler options cannot be changed for a given source file, put the OVERRIDE keyword as the first and only keyword on the process statement, followed by a list of options. This means that default and invocation options cannot override the specified options.

You can specify up to 10 process statements in each source program. Except for the ICTL instruction, process statements must be the first statements in your source program. If you include process statements anywhere else in your source program the assembler treats them as comments.

A process statement has a special coding format, unlike any other assembler instruction, although it is affected by the column settings of the ICTL instruction. You must code the characters *PROCESS starting in the begin column of the source statement, followed by one or more spaces. You can code as many assembler options that can fit in the remaining columns up to, and including the end column of the source statement. Options scanning on a *PROCESS record ends at the first space not enclosed in apostrophes.

You cannot continue a process statement on to the next source record.



assembler_option
Is any assembler option.

These options are not accepted from a process statement

ADATA	LANGUAGE	SYSPARM
ASA	LINECOUNT	TERM
DECK	LIST	TRANSLATE
EXIT	OBJECT	XOBJECT
GOFF	SIZE	

If the option is specified on a process override statement and differs from the option in effect at the time of processing the statement, the assembler issues a warning message.

When the assembler detects an error in a process statement, it produces an error message in the *High Level Assembler Option Summary* section of the assembler listing. If the installation default option PESTOP is set then the assembler stops after it finishes processing any remaining process statements.

The assembler lists the options from process statements in the *High Level Assembler Option Summary* section of the assembler listing. The process statements are also shown as comment lines in the *Source and Object* section of the assembler listing.

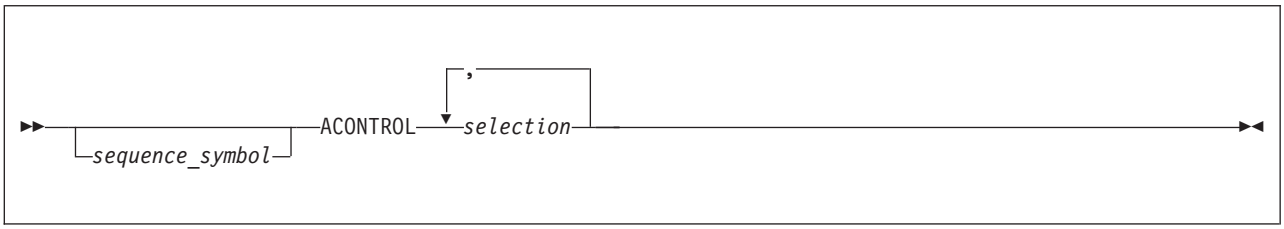
ACONTROL instruction

The ACONTROL instruction can change these HLASM options and controls within a program:

- AFPR

  Note: The AFPR option is not available as an assembler option at invocation of the assembler. It can only be used on ACONTROL instructions.
- COMPAT
- FLAG (except the RECORD/NORECORD and the PUSH/NOPUSH suboptions)
- LIBMAC
- RA2
- TYPECHECK

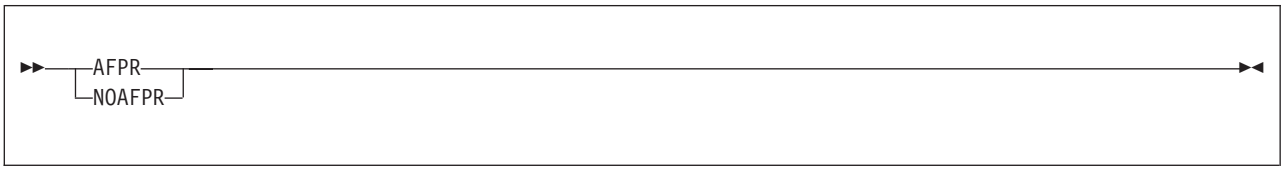
The selections which can be specified are documented here for completeness.



sequence_symbol
Is a sequence symbol.

selection
Is one or more selections from the following options.

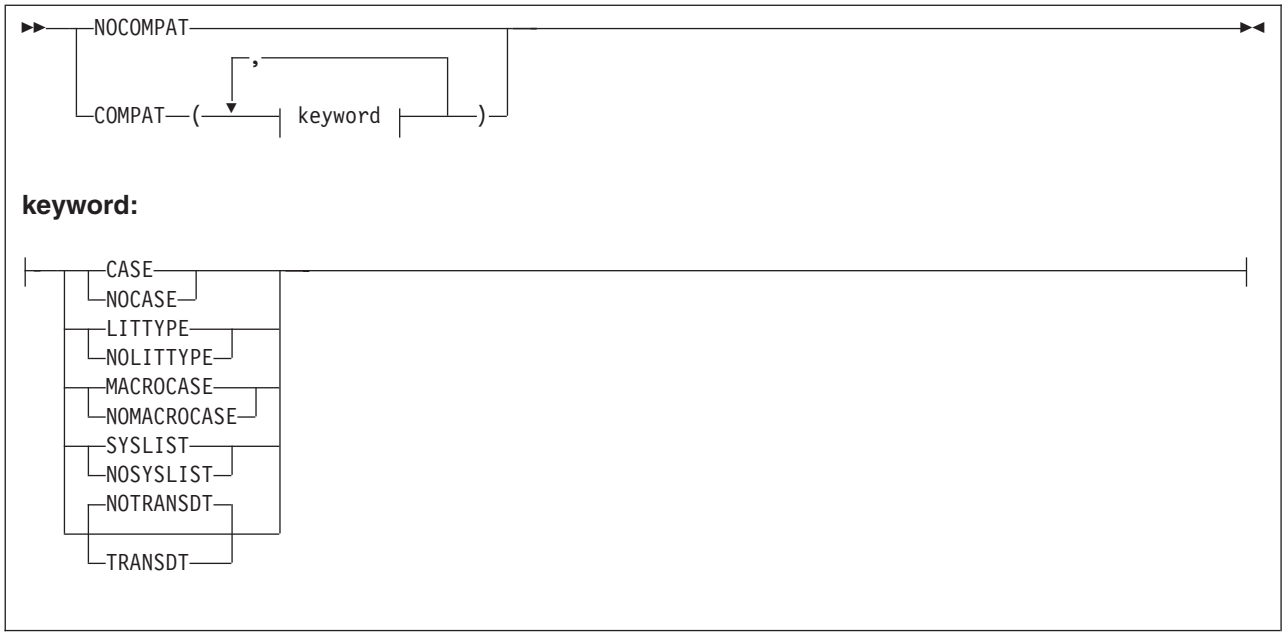
Because ACONTROL changes existing values, there are no default values for the ACONTROL instruction.



AFPR
Instructs the assembler that the additional floating point registers 1, 3, 5, and 7 through 15 can be specified in the program.

Note: The assembler starts with AFPR enabled.

NOAFPR
Instructs the assembler that no additional floating point registers, that is, only floating point registers 0, 2, 4, and 6 can be specified in the program.



COMPAT(CASE), abbreviation CPAT(CASE)
Instructs the assembler to maintain uppercase alphabetic character set compatibility with earlier assemblers.

COMPAT(NOCASE), abbreviation CPAT(NOCASE)
Instructs the assembler to allow mixed case alphabetic character set.

COMPAT(LITTYPE), abbreviation CPAT(LIT)
Instructs the assembler to return 'U' as the type attribute for all literals.

COMPAT(NOLITTYPE), abbreviation CPAT(NOLIT)
Instructs the assembler to return the correct type attribute for literals.

COMPAT(MACROCASE), abbreviation CPAT(MC)
Instructs the assembler to convert internally lowercase alphabetic characters in unquoted macro operands to uppercase alphabetic characters prior to macro expansion. (The source statement is unchanged).

COMPAT(NOMACROCASE), abbreviation CPAT(NOMC)
Instructs the assembler not to convert lowercase alphabetic characters (a through z) in unquoted macro operands.

COMPAT(SYSLIST), abbreviation CPAT(SYSL)
Instructs the assembler to treat sublists in SETC symbols as compatible with earlier assemblers.

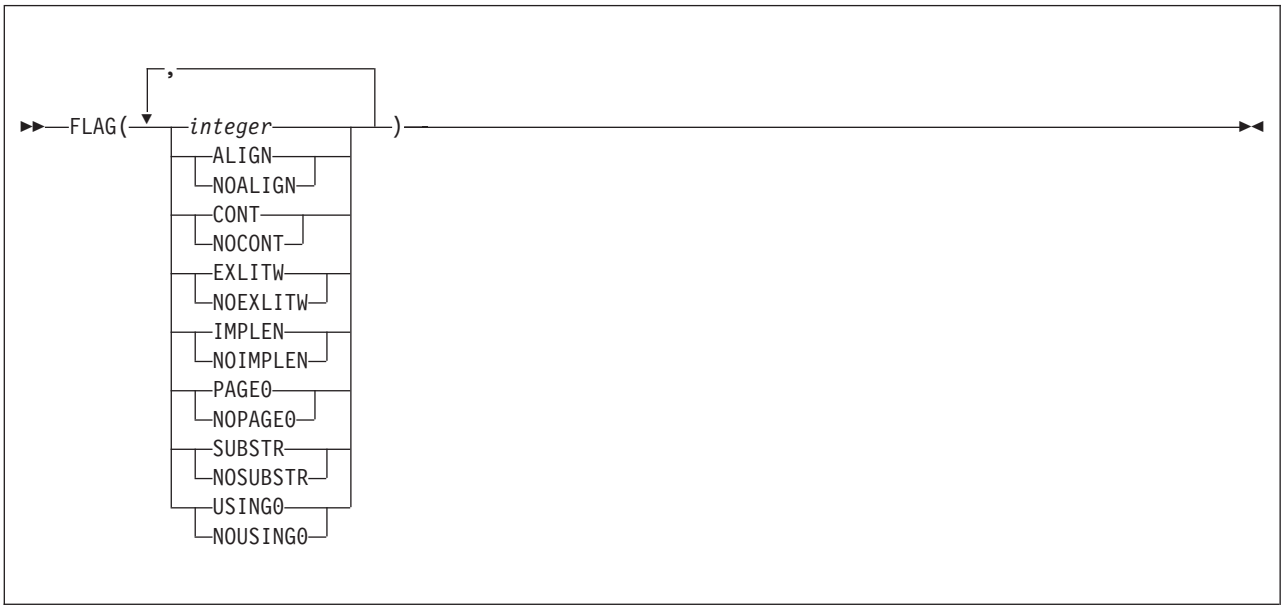
COMPAT(NOSYSLIST), abbreviation CPAT(NOSYSL)
Instructs the assembler not to treat sublists in SETC symbols as character strings, when passed to a macro definition in an operand of a macro instruction.

| **COMPAT(TRANSDT),abbreviation CPAT(TRS)**
| Instructs the assembler to extend use of the translation table, as specified by the TRANSLATE
| assembler option, to any C-type character Self-Defining Terms.

| **COMPAT(NOTRANSDT),abbreviation CPAT(NOTRS)**
| Instructs the assembler not to translate any C-type character Self-Defining Terms.

NOCOMPAT, abbreviation NOCPAT
Instructs the assembler to:
- Allow lowercase alphabetic characters in all language elements.

- Treat sublists in SETC symbols as sublists when passed to a macro definition in the operand of a macro instruction.
- Return the correct type attribute for literals.



integer
Specifies that error diagnostic messages with this or a higher severity code are printed in the source and object section of the assembly listing.

FLAG(ALIGN), abbreviation FLAG(AL)
instructs the assembler to issue diagnostic message ASMA033I, ASMA212W, or ASMA213W when an inconsistency is detected between the operation code and the alignment of addresses in machine instructions.

FLAG(NOALIGN), abbreviation FLAG(NOAL)
instructs the assembler not to issue diagnostic message ASMA033I, ASMA212W, or ASMA213W when an inconsistency is detected between the operation code and the alignment of addresses in machine instructions.

FLAG(CONT)
specifies that the assembler is to issue diagnostic messages ASMA430W through ASMA433W when an inconsistent continuation is encountered in a statement.

FLAG(NOCONT)
specifies that the assembler is not to issue diagnostic messages ASMA430W through ASMA433W when an inconsistent continuation is encountered in a statement.

FLAG(EXLITW)
instructs the assembler to issue diagnostic warning ASMA016W when a literal is specified as the object of an EX instruction.

FLAG(NOEXLITW)
instructs the assembler to suppress diagnostic warning message ASMA016W when a literal is specified as the object of an EX instruction.

FLAG(IMPLEN)
instructs the assembler to issue diagnostic message ASMA169I when an explicit length subfield is omitted from an SS-format machine instruction.

FLAG(NOIMPLEN)
instructs the assembler not to issue diagnostic message ASMA169I when an explicit length subfield is omitted from an SS-format machine instruction.

FLAG(PAGE0)
instructs the assembler to issue diagnostic message ASMA309W when an operand is resolved to a baseless address and a base and displacement is expected.

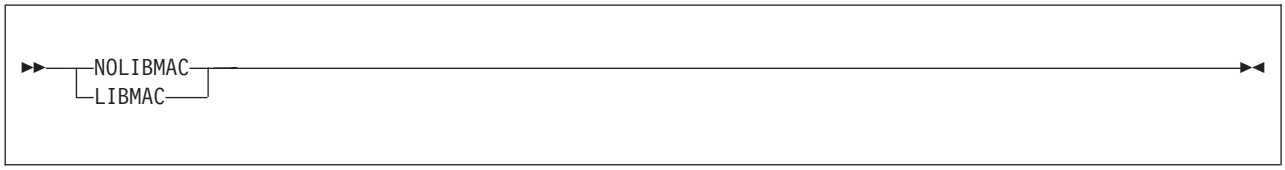
FLAG(NOPAGE0)
instructs the assembler not to issue diagnostic message ASMA309W when an operand is resolved to a baseless address and a base and displacement is expected.

FLAG(SUBSTR), abbreviation FLAG(SUB)
instructs the assembler to issue warning diagnostic message ASMA094I when the second subscript value of the substring notation indexes past the end of the character expression.

FLAG(NOSUBSTR), abbreviation FLAG(NOSUB)
instructs the assembler not to issue warning diagnostic message ASMA094I when the second subscript value of the substring notation indexes past the end of the character expression.

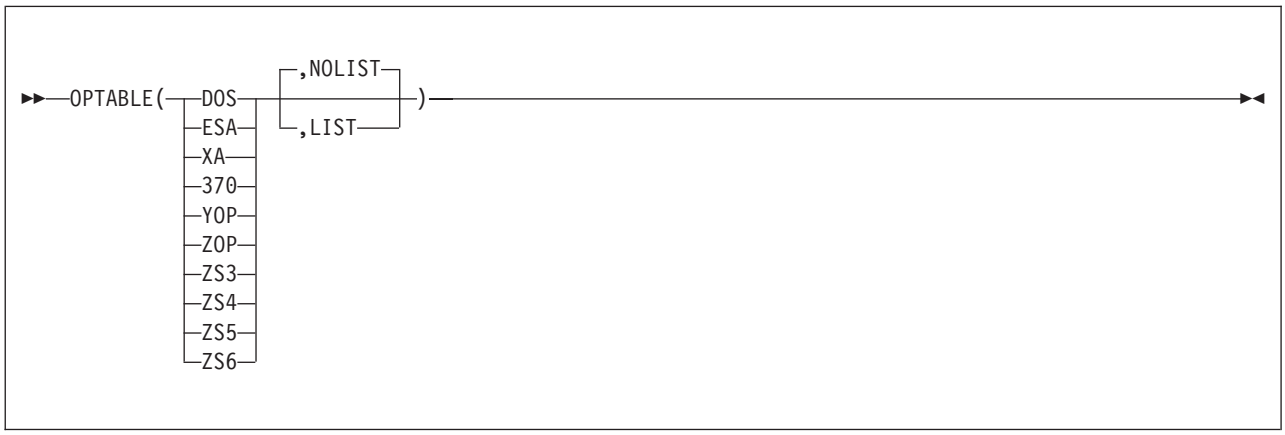
FLAG(USING0), abbreviation FLAG(US0)
instructs the assembler to issue diagnostic warning message ASMA306W for a USING that is coincident with or overlaps the implied USING 0,0, when the USING(WARN) suboption includes the condition numbers 1 and 4.

FLAG(NOUSING0), abbreviation FLAG(NOUS0)
instructs the assembler to suppress diagnostic warning message ASMA306W



LIBMAC, abbreviation LMAC
Specifies that, for each macro, macro definition statements read from a macro library are to be embedded in the input source program immediately preceding the first invocation of that macro.

NOLIBMAC, abbreviation NOLMAC
Specifies that macro definition statements read from a macro library are not to be included in the input source program.



OPTABLE
Lets you switch to a different opcode table. This table is then used to resolve any opcodes after the ACONTROL statement.

DOS
Instructs the assembler to load and use the DOS operation code table. The DOS operation code is designed specifically for assembling programs previously assembled using the DOS/VSE assembler. The operation code table contains the System/370 machine instructions, excluding those with a vector facility.

ESA
Instructs the assembler to load and use the operation code table that contains the ESA/370 and ESA/390 architecture machine instructions, including those with a vector facility. Equivalent to MACHINE(S390E).

UNI
Instructs the assembler to load and use the operation code table that contains the System/370 and System/390 architecture machine instructions, including those with a vector facility, and Z/Architecture machine instructions.

XA Instructs the assembler to load and use the operation code table that contains the System/370 extended architecture machine instructions, including those with a vector facility. Equivalent to MACHINE(S370XA).

370
Instructs the assembler to load and use the operation code table that contains the System/370 machine instructions, including those with a vector facility. Equivalent to MACHINE(S370).

YOP
Same as OPTABLE(ZOP) but with the addition of the long displacement facility. Equivalent to MACHINE(ZSERIES-2).

ZOP
Instructs the assembler to load and use the operation code table that contains the symbolic operation codes for the machine instructions specific to Z/Architecture systems. Equivalent to MACHINE(ZSERIES).

ZS3
Same as OPTABLE(YOP) but with the addition of support for the z9-109 instructions. Equivalent to MACHINE(ZSERIES-3).

ZS4
Same as OPTABLE(ZS3) but with the addition of support for the z10 instructions. Equivalent to MACHINE(ZSERIES-4).

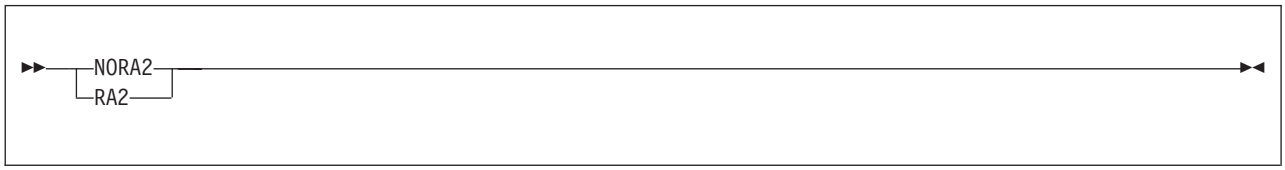
ZS5
Same as OPTABLE(ZS4) but with the addition of support for the z196 instructions. Equivalent to MACHINE(ZSERIES-5).

ZS6
Same as OPTABLE(ZS5) but with the addition of support for the zEnterprise EC12 (zEC12) instructions. Equivalent to MACHINE(ZSERIES-6).

LIST
Instructs the assembler to produce the Operation Code Table Contents section in the listing. Equivalent to MACHINE(LIST).

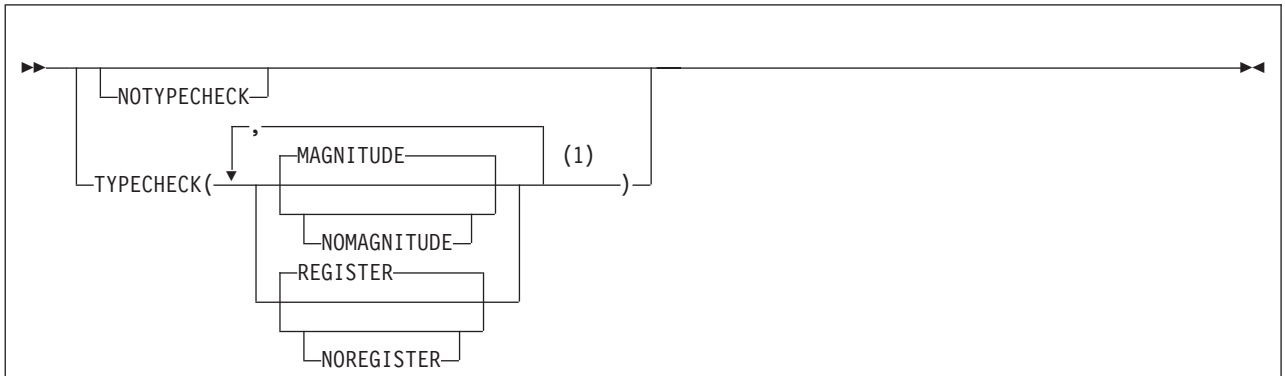
NOLIST
Instructs the assembler not to produce the Operation Code Table Contents section in the listing. Equivalent to MACHINE(NOLIST).

Note: Any macros fetched from SYSLIB receive the current optable setting. If a switch is made to a different table, then any previously resolved macros might be fetched again. Conversely if a switch is made back to a previously used table then any macros that were fetched earlier are available again.



RA2
Instructs the assembler to suppress error diagnostic message ASMA066W when 2 byte relocatable address constants are defined in the source

NORA2
Instructs the assembler to issue error diagnostic message ASMA066W when 2 byte relocatable address constants are defined in the source



Notes:

1 Choose at least one option.

TYPECHECK(MAGNITUDE)
Specifies that the assembler performs magnitude validation of signed immediate-data fields of machine instruction operands.

TYPECHECK(NOMAGNITUDE)
Specifies that the assembler not perform magnitude validation of signed immediate-data fields of machine instruction operands.

TYPECHECK(REGISTER)
Specifies that the assembler performs type checking of register fields of machine instruction operands.

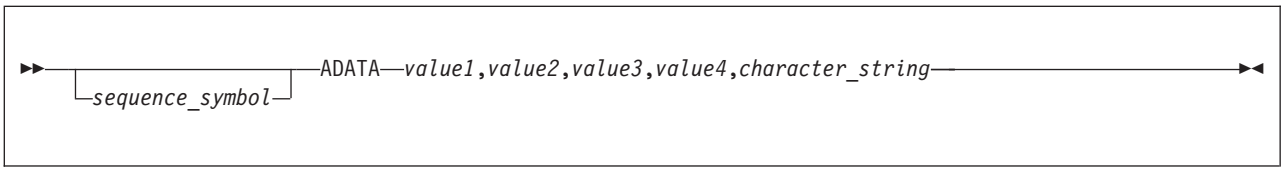
TYPECHECK(NOREGISTER)
Specifies that the assembler not perform type checking of register fields of machine instruction operands.

NOTYPECHECK
Specifies that the assembler not perform any type checking of machine instruction operands.

For further details of the TYPECHECK option, refer to the appendix "TYPECHECK Assembler Option" in the *HLASM Programmer's Guide*.

ADATA instruction

The ADATA instruction writes records to the associated data file.



sequence_symbol
Is a sequence symbol.

value1-value4
Up to four values can be specified, separated by commas. If a value is omitted, the field written to the associated data file contains binary zeros. You must code a comma in the operand for each omitted value. If specified, *value1* through *value4* must be a decimal self-defining term with a value in the range -2^{31} to $+2^{31}-1$.

character_string
Is a character string up to 255 bytes long, enclosed in single quotes. If omitted, the length of the user data field in the associated data file is set to zero.

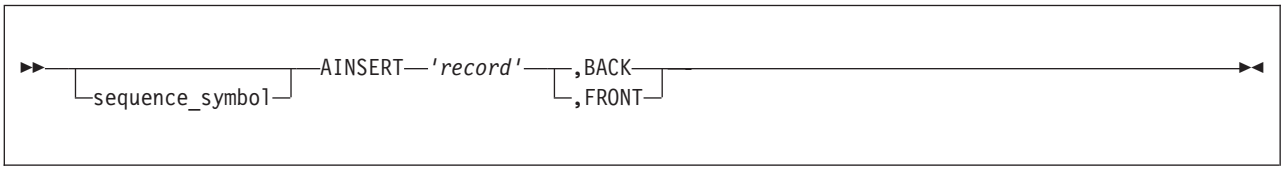
Notes:

1. All operands can be omitted to produce a record containing binary zeros in all fields except the user data field.
2. The record written to the associated data file is described in the section "User-Supplied Information Record X'0070'" in the *HLASM Programmer's Guide*.
3. If you do not specify the ADATA assembler option, or the GOFF(ADATA) or the XOBJECT(ADATA) assembler option (z/OS or CMS), the assembler only checks the syntax of an ADATA instruction, and prints it in the assembler listing.
4. The assembler writes associated data records to the SYSADATA (z/OS or CMS), or the SYSADAT (z/VSE) file if the ADATA assembler option has been specified.

AINSERT instruction

The AINSERT instruction inserts records into the input stream. These records are queued in an internal buffer until the macro generator has completed expanding the current outermost macro instruction. At that point the internal buffer queue provides the next record or records. An operand controls the sequence of the records within the internal buffer queue.

Note: You can place inserted records at either end of the buffer queue, the records are removed only from the front of the buffer queue.



sequence_symbol
Is a sequence symbol.

record
Is the record stored in the internal buffer. It can be any characters enclosed in apostrophes.

The rules that apply to this character string are:

- Variable symbols are allowed.
- The string can be up to 80 characters in length. If the string is longer than 80 characters, only the first 80 characters are used, the rest of the string is ignored.

BACK
The record is placed at the back of the internal buffer.

FRONT
The record is placed at the front of the internal buffer.

Notes:

1. The ICTL instruction does not affect the format of the stored records. The assembler processes these records according to the standard begin, end, and continue columns.
2. The assembler does not check the sequence field of the stored records, even when the ISEQ instruction is active.
3. Continuation is ignored for the last record in the AINSERT buffer but is active for all other records.

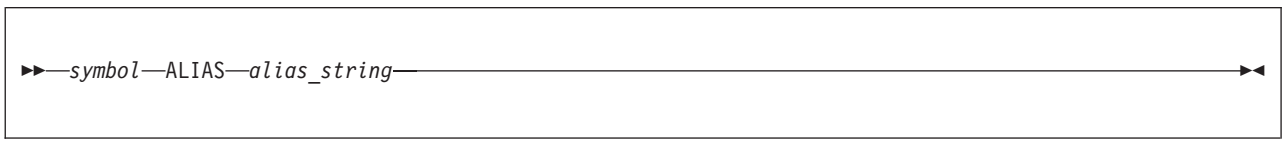
Example:

```
        MACRO
        MAC1
        .
.A      AINSERT 'INSERT RECORD NUMBER ONE',FRONT         Insert record into the input stream
.B      AINSERT 'INSERT RECORD NUMBER TWO',FRONT         Insert record at the top of the input stream
.C      AINSERT 'INSERT RECORD NUMBER THREE',BACK        Insert record at the bottom of the input stream
        .
        .
        .
&FIRST  AREAD                                               Retrieve record TWO from the top of the input stream
        .
.D      AINSERT 'INSERT RECORD NUMBER FOUR',FRONT        Insert record at the top of the input stream
        .
&SECOND AREAD                                               Retrieve record FOUR from the top of the input stream
        .
        MEND
        CSECT
        .
        MAC1
        .
        END
```

In this example, the variable &FIRST receives the operand of the AINSERT statement created at .B. &SECOND receives the operand of the AINSERT statement created at .D. The operand of the AINSERT statements at .A and .C are in the internal buffer in the sequence .A followed by .C and are the next statements processed when the macro generator has finished processing.

ALIAS instruction

The ALIAS instruction specifies alternate names for the external symbols that identify control sections, entry points, and external references. The instruction has nothing to do with the link-time aliases in libraries.



symbol
Is an external symbol that is represented by one of the following:

- An ordinary symbol

- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

alias_string
Is the alternate name for the external symbol, represented by one of the following:

- A character constant in the form C'*aaaaaaaa*', where *aaaaaaaa* is a string of characters each of which has a hexadecimal value of X'42' - X'FE'.
- A hexadecimal constant in the form X'*xxxxxxxx*', where *xxxxxxxx* is a string of hexadecimal digits, each pair of which is in the range X'42' - X'FE'.

The ordinary symbol denoted by *symbol* must also appear in one of the following in this assembly:

- The name entry field of a START, CSECT, RSECT, COM, or DXD instruction
- The name entry field of a DSECT instruction and the nominal value of a Q-type offset constant
- The operand of an ENTRY, EXTRN, or WXTRN instruction
- The nominal value of a V-type address constant

Note: All external symbols identified by EXTRN, WXTRN, ENTRY and V-cons statements must belong to an owning section definition.

The assembler uses the string denoted by *alias_string* to replace the external symbol denoted by *symbol* in the external symbol dictionary records in the object module. Because the change is made only in the external symbol dictionary, references to the ALIASed symbol in the source program must use the original symbol. If the string is shorter than eight characters, or 16 hexadecimal digits, it is padded on the right with EBCDIC spaces (X'40'). If the string is longer than eight characters, it is truncated. Some programs that process object modules do not support external symbols longer than 8 characters.

z/VM and z/OS
If the extended object format is being generated (GOFF assembler option), the *alias_string* can be up to 256 characters, or 512 hexadecimal digits.

The following examples are of the ALIAS instruction, and show both formats of the alternate name denoted by *alias_string*.

```
EXTSYM1  ALIAS           C'lower1'
EXTSYM2  ALIAS           X'9396A68599F2'
```

The *alias_string* must not match any external symbol, regardless of case. References to an ALIASed symbol must be made using the original name; the original symbol is changed only in the external symbol dictionary. For example, you write

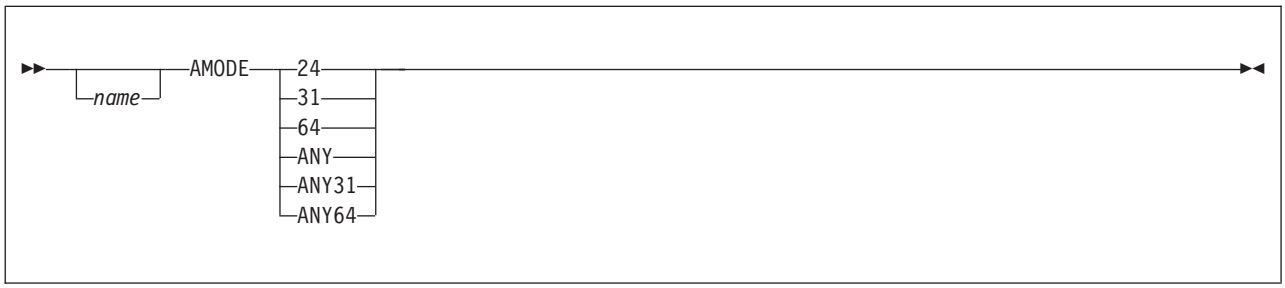
```
        EXTRN    EXTSYM1
```

to refer to the external symbol 'lower1'.

Aliased names are not checked against ALIASes for possible duplicates or conflicts.

AMODE instruction

The AMODE instruction specifies the addressing mode associated with control sections in the object deck.



name
Is the name field that associates the addressing mode with a control section, or, if GOFF is specified, an ENTRY symbol or EXTRN symbol. If GOFF is specified, AMODE can also associate an addressing mode with an entry name. If there is a symbol in the name field, it must also appear in the name field of a START, CSECT, RSECT, or COM instruction. If GOFF is specified, the symbol can also appear as the operand of an ENTRY, EXTRN, or WXTRN instruction, in this assembly. If the name field is space-filled, there must be an unnamed control section in this assembly. If the name field contains a sequence symbol (see "Symbols" on page 25 for details), it is treated as a blank name field.

z/VM and z/OS
If the extended object format is being generated (GOFF assembler option), *name* is a relocatable symbol that names an entry point specified on an ENTRY instruction, or on an external symbol specified on an EXTRN instruction.

24
Specifies that 24 bit addressing mode is to be associated with a control section, or entry point.

31
Specifies that 31 bit addressing mode is to be associated with a control section, or entry point.

64
Specifies that 64 bit addressing mode is to be associated with a control section, or entry point (see "64 bit addressing mode" on page 84).

ANY
The same as ANY31.

ANY31
The control section or entry point is not sensitive to whether it is entered in AMODE 24 or AMODE 31.

ANY64
The control section or entry point is not sensitive to whether it is entered in AMODE 24, AMODE 31, or AMODE 64.

Any field of this instruction can be generated by a macro, or by substitution in open code.

If *symbol* denotes an ordinary symbol, the ordinary symbol associates the addressing mode with an external symbol. The ordinary symbol must also appear in the name field of a START, CSECT, RSECT, or COM instruction or, if GOFF is specified, the operand of an ENTRY, EXTRN, or WXTRN instruction, in this assembly.

If *symbol* is not specified, or if *name* is a sequence symbol, there must be an unnamed control section in this assembly.

Notes:

1. AMODE can be specified anywhere in the assembly. It does not initiate an unnamed control section.
2. AMODE is permitted on external labels (EXTRNs) and Entry labels for both GOFF formats and Parts for GOFF formats.
3. An assembly can have multiple AMODE instructions; however, two AMODE instructions cannot have the same name field.
4. The valid and invalid combinations of AMODE and RMODE are shown in the following table. Combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64 bit addressing mode" on page 84.

Table 11. AMODE/RMODE combinations

	RMODE 24	RMODE 31	RMODE 64
AMODE 24	OK	invalid	invalid
AMODE 31	OK	OK	invalid
AMODE ANY | ANY31	OK	OK	invalid
AMODE 64 | ANY64	OK	OK	OK

5. AMODE or RMODE can only be specified for an unnamed control section by specifying a sequence symbol or a space in the name field.
6. The defaults used when there is no mode or one MODE is specified are shown in the following table. Combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64 bit addressing mode" on page 84.

Table 12. AMODE/RMODE defaults

Specified	Default
Neither	AMODE 24, RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY | ANY31	RMODE 24
RMODE 24	AMODE 24
RMODE 31 (was ANY)	AMODE 31
AMODE 64	RMODE 31
AMODE ANY64	RMODE 31
RMODE 64	AMODE 64

CATTR instruction (z/OS and CMS)

The CATTR instruction establishes a program object external class name, and assigns binder attributes for the class. This instruction is valid only when you specify the GOFF or XOBJECT assembler option.

class_name
Is a valid program object external class name. The class name must follow the rules for naming external symbols, except that:
- Class names are restricted to a maximum of 16 characters
- Class names with an underscore (_) in the second character are reserved for IBM use; for example *B_TEXT*. If you use a class name of this format, it might conflict with an IBM-defined binder class.

attribute
Is one or more binder attributes that are assigned to the text in this class:

ALIGN(*n*)
Aligns the text on a 2^n boundary. *n* is an integer with value 0, 1, 2, 3, 4, or 12. If not specified, then the SECTALGN option value (8 is the default, corresponding to ALIGN(3)) is used (see the section "SECTALGN" in the *HLASM Programmer's Guide* for more information).

Note: Execution-time support of the desired alignment depends on its being respected by other operating system components such as linkers and loaders.

EXECUTABLE
The text can be branched to or executed—it is instructions, not data.

DEFLOAD
The text is not loaded when the program object is brought into storage, but is probably requested, and therefore partially loaded, for fast access.

MOVABLE
The text can be moved, and is reenterable (that is, it is free of location-dependent data such as address constants, and executes normally if moved to a properly aligned boundary).

NOLOAD
The text for this class is not loaded when the program object is brought into storage. An external dummy section is an example of a class which is defined in the source program but not loaded.

NOTEXECUTABLE
The text cannot be branched to or executed (that is, it is data, not instructions).

NOTREUS
The text is marked not reusable.

PART(*part-name*)
Identifies or continues the part with the name *part-name*. The *part-name* can be up to 63 characters in length. An invalid *part-name* is ignored and diagnostic message 'ASMA062E Illegal operand format xxxxxx' is issued.

Binding attributes assigned to the class are also assigned to the part. Both the class and the part are assigned to Name Space 3 and are assigned the **merge** attribute.

Text within a part cannot contain an entry point. If an entry point is found within the part it is ignored and diagnostic message 'ASMA048E Entry error - *xxxxxxxx*' is issued.

The following rules apply to the validation of the PART attribute on the CATTR instruction:
- If the PART attribute has not been specified on the first CATTR statement for the class, but is specified on subsequent CATTR statements for the class, the attribute is ignored and diagnostic message ASMA191W is issued.
- If the PART attribute has been specified on the first CATTR statement for the class, but is not specified on subsequent CATTR statements for the class, the diagnostic message ASMA155S is issued.
- Multiple parts can be defined within a class.

PRIORITY(*nnnnn*)
The binding priority to be attached to this part. The value must be specified as an unsigned

decimal number and must lie between 0 and 2^{31}-1. An invalid *priority* is ignored and diagnostic message 'ASMA062E Illegal operand format xxxxxx' is issued.

The PRIORITY attribute can be specified on the first CATTR instruction for the part. If the PRIORITY attribute is specified on second or subsequent CATTR instructions for the part it is ignored and the diagnostic message ASMA191W is issued.

The PRIORITY attribute is ignored if there is no PART attribute on the CATTR instruction and the diagnostic message 'ASMA062E Illegal operand format xxxxxx' is issued.

READONLY
The text is storage-protected.

REFR
The text is marked refreshable.

REMOVABLE
The content of this class can be discarded from the program object at bind time if the user specifies an appropriate binder option. This might help reduce the size of the program object.

RENT
The text is marked reenterable.

REUS
The text is marked reusable.

RMODE(24)
The text has a residence mode of 24.

RMODE(31)
The text has a residence mode of 31.

RMODE(ANY)
The text can be placed in any addressable storage.

These attributes are accepted by the assembler and encoded in the GOFF object file, but some are not processed by the binder.

Refer to the z/OS MVS Program Management: User's Guide and Reference, SA22-7643 for details about the binder attributes.

Default Attributes: When you do not specify attributes on the CATTR instruction the defaults are: ALIGN(3),EXECUTABLE,NOTREUS,RMODE(24) The LOAD attribute is the default if DEFLOAD or NOLOAD are not specified.

Where to Use the CATTR Instruction: Use the CATTR instruction anywhere in a source module after any ICTL or *PROCESS statements. The CATTR instruction must be preceded by a START, CSECT, or RSECT statement, otherwise the assembler issues diagnostic message ASMA190E.

A section can contain any number of classes. Any machine language instructions or data appearing after a CATTR instruction are components of the element defined by the section and class names. An element is a separately relocatable component of the resulting program object, and is typically bound with other elements having the same attributes.

If several CATTR instructions within a source module have the same class name, the first occurrence establishes the class and its attributes, and the rest indicate the continuation of the text for the class. If you specify attributes on subsequent CATTR instructions having the same class name as a previous CATTR instruction, the assembler ignores the attributes and issues diagnostic message ASMA191W.

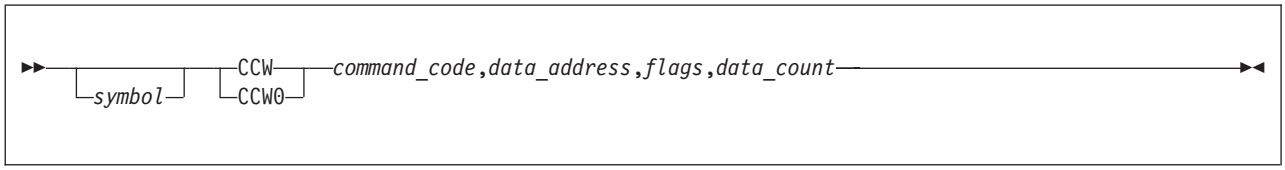
If you specify conflicting attributes on the same instruction, the assembler uses the last one specified. In the following example, the assembler uses RMODE(ANY):

```
MYCLASS  CATTR RMODE(24),RMODE(ANY)
```

Syntax Checking Only: If you code a CATTR instruction but do not specify the GOFF or XOBJECT option, the assembler checks the syntax of the instruction statement and does not process the attributes.

CCW and CCW0 instructions

The CCW and CCW0 instructions define and generate an 8 byte, format-0 channel command word for input/output operations. A format-0 channel command word allows a 24 bit data address. The CCW and CCW0 instructions have identical functions. If a control section has not been established, CCW and CCW0 initiate an unnamed (private) control section.



symbol
Is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

command_code
Is an absolute expression that specifies the command code. This expression's value is right-aligned in byte 0 of the generated channel command word.

data_address
Is a relocatable or absolute expression that specifies the address of the data to operate upon. This value is treated as a 3 byte, A-type address constant. The value of this expression is right-aligned in bytes 1 to 3 of the generated channel command word.

flags
Is an absolute expression that specifies the flags for bits 32 to 39, and is right-aligned, of the generated channel command word. The value of this expression is right-aligned in byte 4 of the generated channel command word. Byte 5 is set to zero by the assembler.

data_count
Is an absolute expression that specifies the byte count or length of data. The value of this expression is right-aligned in bytes 6 and 7 of the generated channel command word.

The generated channel command word is aligned at a doubleword boundary. Any skipped bytes are set to zero.

The internal machine format of a channel command word is shown in Table 13.

Table 13. Channel command word, format 0

Byte	Bits	Usage
0	0-7	Command code
1-3	8-31	Address of data to operate upon
4	32-39	Flags
	38-39	Must be specified as zeros
5	40-47	Set to zeros by assembler

Table 13. Channel command word, format 0 (continued)

Byte	Bits	Usage
6-7	48-63	Byte count or length of data

If *symbol* is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the generated channel command word. The length attribute value of the symbol is 8.

Here is an example of a channel program:

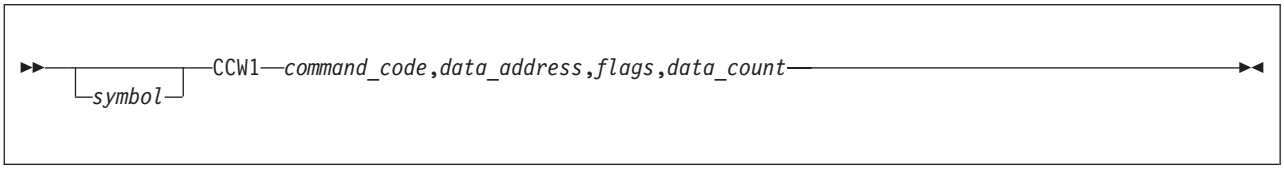
```
LocRcd   CCW   X'47',LocData,X'48',L'LocData   Locate record
         CCW0  X'06',MyData,X'40',MyBlkSize    Read Data
         CCW0  X'06',MyData+MyBlkSize,0,80     Read Data
LocData  DC    XL16'0'      Locate Record data, set at run time
```

z/OS ***Using EXCP or EXCPVR access methods:*** If you use the EXCP or EXCPVR access method, you must use CCW or CCW0, because EXCP and EXCPVR do not support 31-bit data addresses in channel command words.

Specifying RMODE: Use RMODE 24 with CCW or CCW0 if you wish to ensure that valid data addresses are generated. If you use RMODE ANY with CCW or CCW0, an invalid data address in the channel command word can result at execution time. If your program has an RMODE value other than 24, you might choose to code 0 or an absolute expression for the data addresses. When your program runs, it can copy the channel program to 24-bit storage for execution and set or relocate the address fields.

CCW1 instruction

The CCW1 instruction defines and generates an 8 byte format-1 channel command word for input/output operations. A format-1 channel command word allows 31 bit data addresses. A format-0 channel command word generated by a CCW or CCW0 instruction allows only a 24 bit data address. If a control section has not been established, CCW1 initiates an unnamed (private) control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

command_code
Is an absolute expression that specifies the command code. This expression's value is right-aligned in byte 0 of the generated channel command word.

data_address
Is a relocatable or absolute expression that specifies the address of the data to operate upon. This value is treated as a 4 byte, A-type address constant. The value of this expression is right-aligned in bytes 4 to 7 of the generated channel command word.

flags
Is an absolute expression that specifies the flags for bits 8 to 15 of the generated channel command word. The value of this expression is right-aligned in byte 1 of the generated channel command word.

data_count
Is an absolute expression that specifies the byte count or length of data. The value of this expression is right-aligned in bytes 2 and 3 of the generated channel command word.

The generated channel command word is aligned at a doubleword boundary. Any skipped bytes are set to zero.

The internal machine format of a channel command word is shown in Table 14.

Table 14. Channel command word, format 1

Byte	Bits	Usage
0	0-7	Command code
1	8-15	Flags
2-3	16-31	Count
4	32	Must be zero
4-7	33-63	Data address

The expression for the data address should be such that the address is 0 - 2^{31}-1, after possible relocation. This is the case if the expression refers to a location within one of the control sections that are link-edited together. An expression such as `*-1000000000` yields an acceptable value only when the value of the location counter (*) is 1000000000 or higher at assembly time.

If *symbol* is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the generated channel command word. The length attribute value of the symbol is 8.

Here is an example of a CCW1 statement:

```
A       CCW1            X'0C',BUF1,X'00',L'BUF1
```

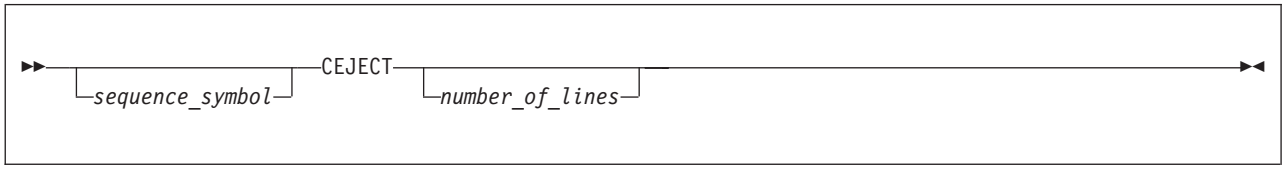
The object code generated (in hexadecimal) for the above examples is:

```
0C 00 yyyy xxxxxxxx
```

where *yyyy* is the length of `BUF1` and *xxxxxxxx* is the address of `BUF1`. `BUF1` can reside anywhere in virtual storage.

CEJECT instruction

The CEJECT instruction conditionally stops the printing of the assembler listing on the current page, and continues the printing on the next page.



sequence_symbol
Is a sequence symbol.

number_of_lines
Is an absolute value that specifies the minimum number of lines that must be remaining on the current page to prevent a page eject. If the number of lines remaining on the current page is less than the value specified by *number_of_lines*, the next line of the assembler listing is printed at the top of a new page.

You can use any absolute expression to specify *number_of_lines*.

If number of lines is omitted, the CEJECT instruction behaves as an EJECT instruction.

If zero, a page is ejected unless the current line is at the top of a page.

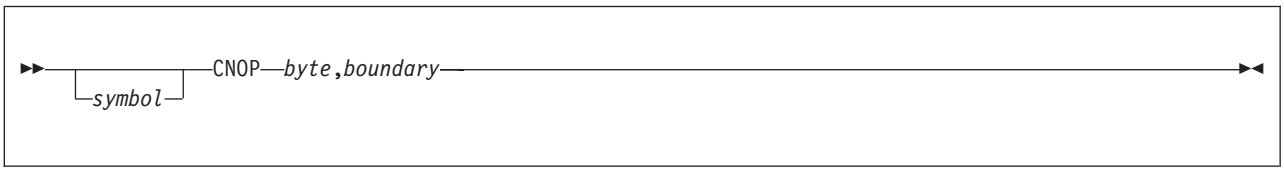
If the line before the CEJECT statement appears at the bottom of a page, the CEJECT statement has no effect. A CEJECT instruction without an operand immediately following another CEJECT instruction or an EJECT instruction is ignored.

Notes:

1. The CEJECT statement itself is not printed in the listing unless a variable symbol is specified as a point of substitution in the statement, in which case the statement is printed before substitution occurs.
2. The PRINT DATA and PRINT NODATA instructions can alter the effect of the CEJECT instruction, depending on the number of assembler listing lines that are required to print the generated object code for each instruction.

CNOP instruction

The CNOP instruction aligns any instruction or other data on a specific halfword boundary. This ensures an unbroken flow of executable instructions, since the CNOP instruction generates no-operation instructions to fill the bytes skipped to achieve specified alignment. If a control section has not been established, CNOP initiates an unnamed (private) control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The name is assigned to the next halfword aligned location. If there is a single byte before that location, it is skipped for alignment. Zero or more NOP(R)s might be generated at or after that location.

byte
Is an absolute expression that specifies at which even-numbered byte in a fullword, doubleword, or quadword the location counter is set. The value of the expression must be *0* to *boundary*-2.

boundary
Is an absolute expression that specifies the byte specified by *boundary* is in a fullword, doubleword, or quadword. A value of 4 indicates the byte is in a fullword, a value of 8 indicates the byte is in a doubleword, and a value of 16 indicates the byte is in a quadword.

Table 15 shows valid pairs of *byte* and *word*.

Table 15. Valid CNOP values

Values	Specify
0,4	Beginning of a word
2,4	Middle of a word
0,8	Beginning of a doubleword
2,8	Second halfword of a doubleword
4,8	Middle (third halfword) of a doubleword
6,8	Fourth halfword of a doubleword
0,16	Beginning of a quadword
2,16	Second halfword of a quadword
4,16	Third halfword of a quadword
6,16	Fourth halfword of a quadword
8,16	Fifth halfword of a quadword
10,16	Sixth halfword of a quadword
12,16	Seventh halfword of a quadword
14,16	Eighth halfword of a quadword

Figure 20 shows the position in a doubleword that each of these pairs specifies. Both 0,4 and 2,4 specify two locations in a doubleword.

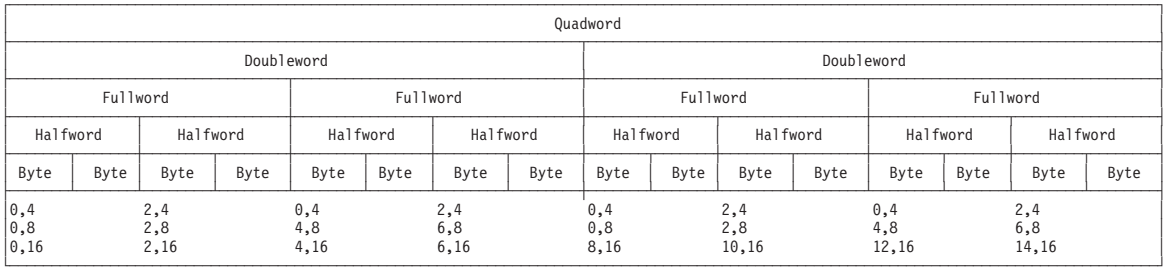


Figure 20. CNOP alignment

Use the CNOP instruction, for example, when you code the linkage to a subroutine, and you want to pass parameters to the subroutine in fields immediately following the branch and link instructions. These parameters—for example, channel command words—can require alignment on a specific boundary. The subroutine can then address the parameters you pass through the register with the return address, as in the following example:

```
         CNOP            6,8
LINK     BALR            2,10
         CCW             1,DATADR,X'48',X'50'
```

Assume that the location counter is aligned at a doubleword boundary. Then the CNOP instruction causes the following no-operations to be generated, thus aligning the BALR instruction at the last halfword in a doubleword as follows:

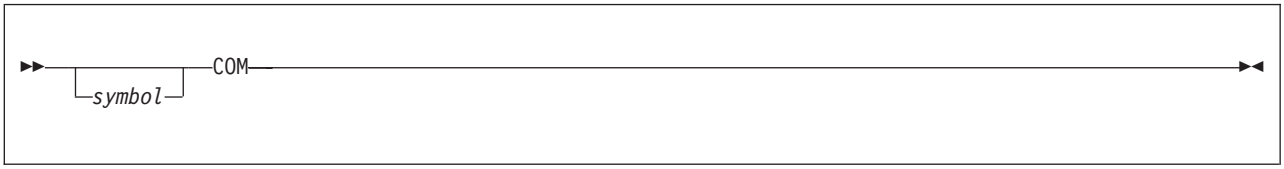
```
         BCR             0,0
         BC              0,X'700'
         BALR            2,10
LINK     CCW             1,DATADR,X'48',X'50'
```

After the BALR instruction is generated, the location counter is at a doubleword boundary, thus ensuring that the CCW instruction immediately follows the branch and link instruction.

The CNOP instruction forces the alignment of the location counter to a halfword, fullword, doubleword, or quadword boundary. It does not affect the location counter if the counter is already correctly aligned. If the specified alignment requires the location counter to be incremented, no-operation instructions are generated to fill the skipped bytes. Any single byte skipped to achieve alignment to the first no-operation instruction is filled with zeros, even if the preceding byte contains no machine language object code. A length attribute reference to the name of a CNOP instruction is always invalid. Message ASMA042E is issued, and a default value of 1 is assigned.

COM instruction

The COM instruction identifies the beginning or continuation of a common control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The COM instruction can be used anywhere in a source module after the ICTL instruction.

If *symbol* denotes an ordinary symbol, the ordinary symbol identifies the common control section. If several COM instructions within a source module have the same symbol in the name field, the first occurrence initiates the common section and the rest indicate the continuation of the common section. The ordinary symbol denoted by *symbol* represents the address of the first byte in the common section, and has a length attribute value of 1.

If *symbol* is not specified, or if *name* is a sequence symbol, the COM instruction initiates, or indicates the continuation of, the unnamed common section.

See “CSECT instruction” on page 106 for a discussion on the interaction between COM and the GOFF assembler option.

The location counter for a common section is always set to an initial value of 0. However, when an interrupted common control section is continued using the COM instruction, the location counter last specified in that control section is continued.

If a common section with the same name (or unnamed) is specified in two or more source modules, the amount of storage reserved for this common section is equal to that required by the longest common section specified.

The source statements that follow a COM instruction belong to the common section identified by that COM instruction.

Note:

1. The assembler language statements that appear in a common control section are not assembled into object code.
2. When establishing the addressability of a common section, the symbol in the name field of the COM instruction, or any symbol defined in the common section, can be specified in a USING instruction.
3. An AMODE cannot be assigned to a common section.

In the following example, addressability to the common area of storage is established relative to the named statement XYZ.

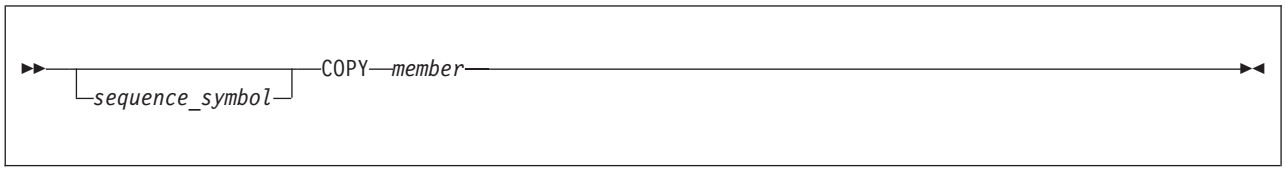
```
         .
         .
         L               1,=A(XYZ)
         USING           XYZ,1
         MVC             PDQ(16),=4C'ABCD'
         .
         .
         COM
XYZ      DS              16F
PDQ      DS              16C
         .
         .
```

A common control section can include any assembler language instructions, but no object code is generated by the assembly of instructions or constants appearing in a common control section. Data can only be placed in a common control section through execution of the program.

If the common storage is assigned in the same manner by each independent assembly, reference to a location in common by any assembly results in the same location being referenced.

COPY instruction

Use the COPY instruction to obtain source statements from a source language library and include them in the program being assembled. You can thus avoid writing the same, often-used sequence of code over and over.



sequence_symbol
Is a sequence symbol.

member
Is an ordinary symbol that identifies a source language library member to be copied from either a system macro library or a user macro library. In open code, it can also be a variable symbol that has been assigned a valid ordinary symbol.

The source statements that are copied into a source module:

- Are inserted immediately after the COPY instruction.
- Are inserted and processed according to the standard instruction statement coding format, even if an ICTL instruction has been specified.
- Must not contain either an ICTL or ISEQ instruction.
- Can contain other COPY statements. There are no restrictions on the number of levels of nested copy instructions. However, the COPY nesting must not be recursive. For example, assume that the source program contains the statement:

```
COPY  A
```

and library member A contains the statement:

```
COPY  B
```

In this case, the library member B must not contain a `COPY A` or `COPY B` statement.

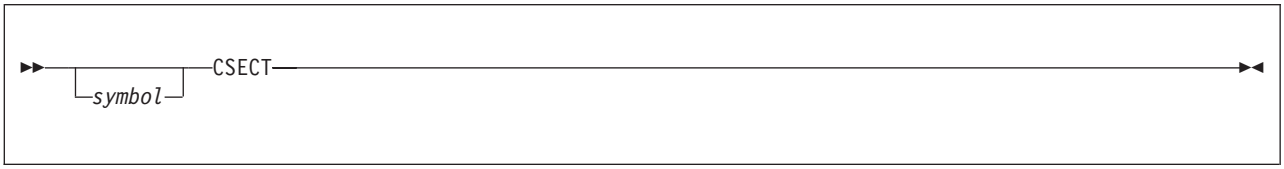
- Can contain macro definitions. Note, however, that if a source macro definition is copied into a source module, both the MACRO and MEND statements that delimit the definition must be contained in the same level of copied code.
- The scope of any sequence symbols defined by the statements within the COPY member are the same as that of the COPY statement itself. That is, if the COPY statement appears in open code then any sequence symbols defined by statements within the member also have open code scope. Take care to define symbols only once, because COPYing the same member more than once can cause looping due to backward AGO or AIF branches in the source file.

Notes:

1. The COPY instruction can also be used to copy statements into source macro definitions.
2. The rules that govern the occurrence of assembler language statements in a source module also govern the statements copied into the source module.
3. Whenever the assembler processes a COPY statement, whether it is in open code or in a macro definition, the assembler attempts to read the source language library member specified in the COPY statement. This means that all source language library members specified by COPY statements in a source program, including those specified in macro definitions, must be available during the assembly. The HLASM Programmer's Guide describes how to specify the libraries when you run the assembler, in these sections:
   - CMS: “Specifying macro and copy code libraries: SYSLIB”
   - z/OS: “Specifying macro and copy code libraries: SYSLIB”
   - z/VSE: “Specifying macro and copy code libraries: LIBDEF job control statement”
4. If an END instruction is encountered in a member during COPY processing, the assembly is ended. Any remaining statements in the COPY member are discarded.

CSECT instruction

The CSECT instruction initiates an executable control section or indicates the continuation of an executable control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The CSECT instruction can be used anywhere in a source module after any ICTL or *PROCESS statements. If it is used to initiate the first executable control section, it must not be preceded by any instruction that affects the location counter and thus causes a control section to be initiated.

If *symbol* denotes an ordinary symbol, the ordinary symbol identifies the control section. If several CSECT instructions within a source module have the same symbol in the name field, the first occurrence initiates the control section and the rest indicate the continuation of the control section. The ordinary symbol denoted by *symbol* represents the address of the first byte in the control section, and has a length attribute value of 1.

If *symbol* is not specified, or if *name* is a sequence symbol, the CSECT instruction initiates, or indicates the continuation of the unnamed control section.

If the first control section is initiated by a START instruction, the CSECT instruction which continues the section must have the same name as the START instruction.

z/VM and z/OS

When the GOFF option is not specified a control section is initiated or resumed by the CSECT, RSECT, and COM statements. Any machine language text created by statements that follow such control section declarations belongs to the control section, and is manipulated during program linking and binding as an indivisible unit.

When the GOFF option is specified, the behavior of CSECT, RSECT, and COM statements is different. By default, the assembler creates a definition of a text class named B_TEXT, to which subsequent machine language text belongs if no other classes are declared. If you specify other class names using the CATTR statement, machine language text following such CATTR statements belongs to that class.

The combination of a section name and a class name defines an *element*, which is the indivisible unit manipulated during linking and binding. All elements with the same section name are "owned" by that section, and binding actions (such as section replacement) act on all elements owned by a section.

When the GOFF option is specified, and if no CATTR statements are present, then all machine language text is placed in the default class B_TEXT. The behavior of the elements in the bound module is essentially the same as the behavior of control sections when the OBJECT option is specified. However, if additional classes are declared, a section name can best be thought of as a "handle" by which elements within declared classes are owned.

The beginning of a control section is aligned on a boundary determined by the SECTALGN option. However, when an interrupted control section is continued using the CSECT instruction, the location counter last specified in that control section is continued. Consider the coding in Figure 21:

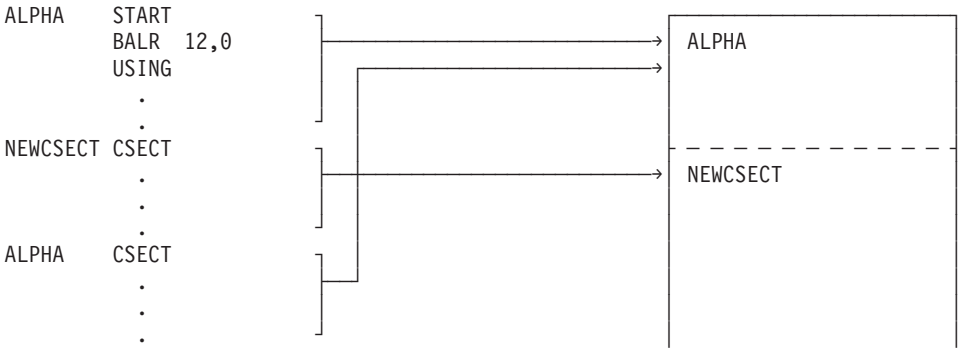


Figure 21. How the location counter works

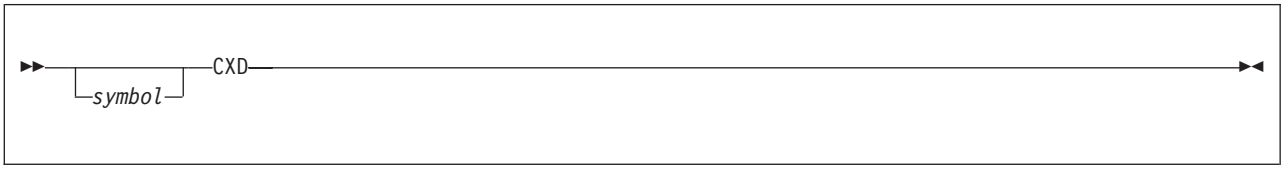
The source statements following a CSECT instruction that either initiate or indicate the continuation of a control section are assembled into the object code of the control section identified by that CSECT instruction.

The end of a control section or portion of a control section is marked by:
- Any instruction that defines a new or continued control section
- The END instruction

The CSECT instruction can interact with any LOCTR instructions that are present. For more information about this interaction, see "LOCTR instruction" on page 169.

CXD instruction

The CXD instruction reserves a fullword area in storage. The linker or loader inserts into this area the total length of all external dummy sections specified in the source modules that are assembled and linked into one program. If a control section has not previously been established, CXD initiates an unnamed (private) control section.



symbol
Is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The linker or loader inserts into the fullword-aligned fullword area reserved by the CXD instruction the total length of storage required for all the external dummy sections specified in a program. If the GOFF assembler option is specified, CXD returns the length of the B_PRV class. If *symbol* denotes an ordinary symbol, the ordinary symbol represents the address of the fullword area. The ordinary symbol denoted by *symbol* has a length attribute value of 4.

These examples show how external dummy sections can be used:

ROUTINE A

```
ALPHA    DXD     2DL8
BETA     DXD     4FL4
OMEGA    CXD
         .
         .
         DC      Q(ALPHA)
         DC      Q(BETA)
         .
         .
```

ROUTINE B

```
GAMMA    DXD     5D
DELTA    DXD     10F
ZETA     DXD     XL22
         .
         .
         DC      Q(GAMMA)
         DC      Q(DELTA)
         DC      Q(ZETA)
         .
         .
```

ROUTINE C

```
EPSILON  DXD     4H
ZETA     DXD     4F
         .
         .
         DC      Q(EPSILON,ZETA)
         .
         .
```

Each of the three routines is requesting an amount of work area. Routine A wants 2 doublewords and 4 fullwords; Routine B wants 5 doublewords, 10 fullwords, and 22 bytes; Routine C wants 4 halfwords and 4 fullwords. During program linking, identically named dummy sections are combined, retaining their strictest alignment and longest length. For example, Routines B and C both request storage named ZETA: the resulting allocation is 22 bytes on a fullword boundary. When program linking is complete, the sum of these individual dummy external section lengths is placed in the location of the CXD instruction labeled OMEGA. Routine A can then allocate the amount of storage that is specified in the CXD location, and each dummy external section's offset within the allocated storage is found in the Q-type offset constant referencing its name. Q-type offset constants are described at “Offset constant—Q” on page 139.

DC instruction

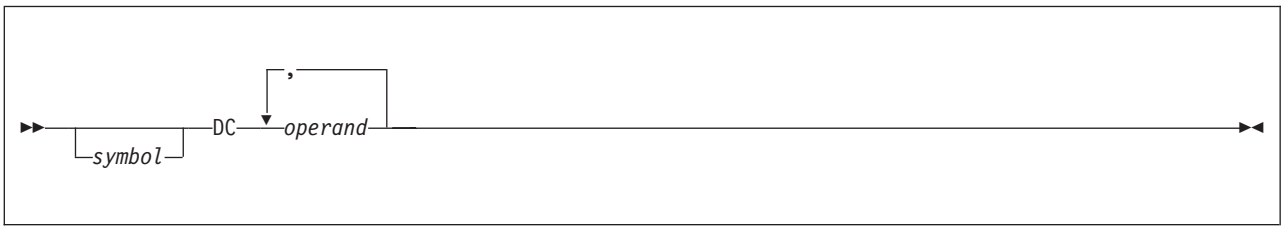
You specify the DC instruction to define the data constants you need for program execution. The DC instruction causes the assembler to generate the binary representation of the data constant you specify into a particular location in the assembled source module; this is done at assembly time.

The DC instruction's name — Define Constant — is misleading: DC simply creates initial data in an area of the program. The contents of that area might be modified during program execution, so the original data is not truly “constant”. If you want to declare values that are more likely to behave like constants, use literals (“Literals” on page 35); the assembler attempts to detect and diagnose instructions that might change the contents of a field defined by a literal. If a control section has not been established previously, DC initiates an unnamed (private) control section.

The DC instruction can generate the following types of constants:

Table 16. Types of data constants

Type of Constant	Function	Example		
Address	Defines address mainly for the use of fixed-point and other instructions		L	5,ADCON
		ADCON	DC	A(SOMWHERE)
Binary	Defines bit patterns	FLAG	DC	B'00010000'
Character	Defines character strings or messages	CHAR	DC	C'string of characters'
Decimal	Used by decimal instructions		ZAP	AREA,PCON
		PCON	DC	P'100'
		AREA	DS	PL3
Fixed-point	Used by the fixed-point and other instructions		L	3,FCON
		FCON	DC	F'100'
Floating-point	Used by floating-point instructions		LE	2,ECON
		ECON	DC	E'100.50'
Graphic	Defines character strings or messages that contain pure double-byte data	DBCS	DC	G'<.D.B.C.S. .S.T.R.I.N.G>'
Hexadecimal	Defines large bit patterns	PATTERN	DC	X'FF00FF00'
Zoned	Defines numeric characters	ZONEVVAL	DC	Z'-123'



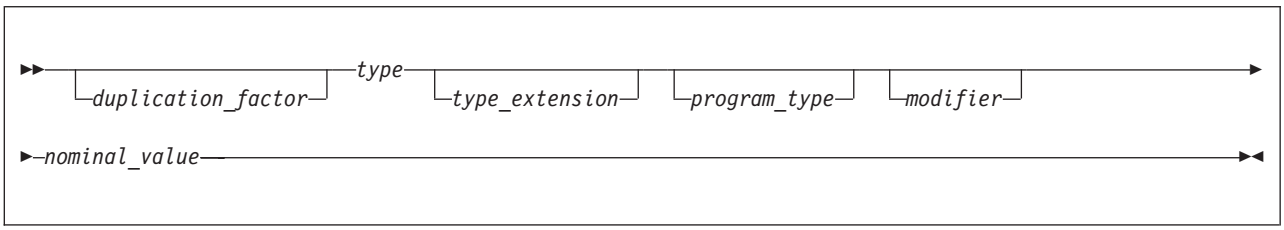
symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If *symbol* denotes an ordinary symbol, the ordinary symbol represents the address of the first byte of the assembled constant. If several operands are specified, the first constant defined is addressable by the ordinary symbol. The other constants can be reached by relative addressing.

operands
An operand of six subfields. The first five subfields describe the constant. The sixth subfield provides the nominal values for the constants.

A DC operand has this format:



duplication_factor
Causes the *nominal_value* to be generated the number of times indicated by this factor. See "Subfield 1: Duplication Factor" on page 114.

type
Further determines the type of constant the *nominal_value* represents. See "Subfield 2: Type" on page 115.

type_extension
Determines some of the characteristics of the constant. See "Subfield 3: Type Extension" on page 116.

program_type
assign a programmer determined 32 bit value to the symbol naming the DC instruction, if a symbol was present. See "Subfield 4: Program type" on page 117.

modifier
Describes the length, the scaling, and the exponent of the *nominal_value*. See "Subfield 5: Modifier" on page 118.

nominal_value
Defines the value of the constant. See "Subfield 6: Nominal Value" on page 121.

For example, in:

```
10EBP(7)L2'12'
```

the six subfields are:
- Duplication factor is `10`
- Type is `E`
- Type extension is `B`
- Program type is `P(7)`
- Modifier is `L2`
- Nominal value is `12`

  If all subfields are specified, the order given above is required. The first, third, fourth, and fifth subfields can be omitted, but the second and sixth must be specified in that order.

Rules for DC operands

1. The type subfield and the nominal value must always be specified unless the duplication factor is zero. If the duplication factor is zero, only the type must be specified.
2. The duplication factor, type extension, program type, and modifier subfields are optional.
3. When multiple operands are specified, they can be of different types.
4. When multiple nominal values are specified in the sixth subfield, they must be separated by commas and be of the same type. Multiple nominal values are not allowed for character or graphic constants, because a comma is part of the nominal value of the constant; it is not possible to specify multiple nominal values.
5. The descriptive subfields, apart from the program type, apply to all the nominal values. The program type applies to only the symbol naming the DC instruction, if a symbol was present. Separate constants are generated for each separate operand and nominal value specified.
6. No spaces are allowed:
   - Between subfields
   - Between multiple operands

General information about constants

Constants defined by the DC instruction are assembled into an object module at the location at which the instruction is specified. However, the type of constant being defined, and the presence or absence of a length modifier, determines whether the constant is to be aligned on a particular storage boundary or not (see "Alignment of constants").

Symbolic Addresses of Constants: The value of the symbol that names the DC instruction is the address of the first byte (after alignment) of the first or only constant.

Length attribute value of symbols naming constants

The length attribute value assigned to the symbols in the name field of the constants is equal to one of:
- The implicit length (see "Implicit Length" in Table 17 on page 112) of the constant when no explicit length is specified in the operand of the constant.
- The explicit length (see "Value of Length Attribute" in Table 17 on page 112) of the constant.

If more than one operand is present, the length attribute value of the symbol is the length in bytes of the first constant specified, according to its implicit or explicit length.

Alignment of constants

The assembler aligns constants on different boundaries according to the following:
- On boundaries implicit to the type of constant (see "Implicit Boundary Alignment" in Table 18 on page 112) when no length is specified.
- On byte boundaries (see "Boundary Alignment" in Table 18 on page 112) when an explicit length is specified.

Bytes that are skipped to align a constant at the correct boundary are not considered part of the constant. They are filled with binary zeros.

Notes:

1. The automatic alignment of constants and areas does not occur if the NOALIGN assembler option has been specified.
2. Alignment can be forced to any boundary by a preceding DS or DC instruction with a zero duplication factor. This occurs whether or not the ALIGN option is set.

Table 17. Length attribute value of symbol naming constants

Type of constant	Implicit Length	Examples	Value of Length Attribute[1]
B	as needed	`DC B'10010000'`	1
C	as needed	`DC C'ABC'`	3
		`DC CL8'WOW'`	8
CU	as needed	`DC CU'ABC'`	6
		`DC CUL4'XX'`	4
G	as needed	`DC G'<DaDb>'`	4
		`DC GL8'<DaDb>'`	8
X	as needed	`DC X'COFFEE'`	3
		`DC XL2'FFEE'`	2
H	2	`DC H'32'`	2
F	4	`DC FL3'32'`	3
FD	8	`DC FD'32'`	8
P	as needed	`DC P'123'`	2
		`DC PL4'123'`	4
Z	as needed	`DC Z'123'`	3
		`DC ZL10'123'`	10
E	4	`DC E'565.40'`	4
D	8	`DC DL6'565.40'`	6
L	16	`DC LL12'565.40'`	12
LQ	16	`DC LQ'565.40'`	16
Y	2	`DC Y(HERE)`	2
A	4	`DC AL1(THERE)`	1
AD	8	`DC AD(WHERE)`	8
S	2	`DC S(THERE)`	2
V	4	`DC VL3(OTHER)`	3
VD	8	`DC VD(BIGOTHER)`	8
J	4	`DC J(CLASS)`	4
JD	4	`DC JD(LARGECLASS)`	8
Q	8	`DC QL1(LITTLE)`	1
QD	4	`DC QD(BIGLITTLE)`	8
RD	8	`DC R(APSECT)`	4
		`DC RD(BPSECT)`	8

Note:
1. Depends on whether an explicit length is specified in the constant.

Table 18. Alignment of constants

Type of constant	Implicit Boundary Alignment	Examples	Boundary Alignment[1]
B	byte	`DC B'1011'`	byte

Table 18. Alignment of constants (continued)

Type of constant	Implicit Boundary Alignment	Examples	Boundary Alignment[1]
C	byte	`DC C'Character string'`	byte
CU	byte	`DC CU'Character string'`	byte
G	byte	`DC G'<.D.B.C.S .S.T.R.I.N.G>'`	byte
X	byte	`DC X'20202021202020'`	byte
H	halfword	`DC H'25'`	halfword
		`DC HL3'25'`	byte
F	fullword	`DC F'225'`	fullword
		`DC FL7'225'`	byte
FD	doubleword	`DC FD'225'`	doubleword
P	byte	`DC P'2934'`	byte
Z	byte	`DC Z'1235'`	byte
		`DC ZL2'1235'`	byte
E	fullword	`DC E'1.25'`	
		`DC EL5'1.25'`	fullword
D	doubleword	`DC 8D'95'`	byte
		`DC 8DL7'95'`	doubleword
L	doubleword	`DC L'2.57E65'`	byte
LQ	quadword	`DC LQ'0.1'`	doubleword
			quadword
		`DC Y(HERE)`	
Y	halfword	`DC AL1(THERE)`	halfword
A	fullword	`DC AD(WHERE)`	byte
AD	doubleword	`DC S(LABEL)`	doubleword
S	halfword	`DC SL2(LABEL)`	halfword
		`DC V(EXTERNAL)`	byte
V	fullword	`DC VL3(EXTERNAL)`	fullword
		`DC VD(BIGOTHER)`	byte
VD	doubleword	`DC J(CLASS)`	doubleword
J	fullword	`DC JD(LARGECLASS)`	fullword
JD	doubleword	`DC QL1(DUMMY)`	doubleword
Q	fullword	`DC QD(BIGDUMMY)`	byte
QD	doubleword	`DC R(APSECT)`	doubleword
R	fullword	`DC RD(BPSECT)`	fullword
RD	doubleword		doubleword

Note:
1. Depends on whether an explicit length is specified in the constant.

Padding and truncation of values

The nominal values specified for constants are assembled into storage. The amount of space available for the nominal value of a constant is determined:

- By the explicit length specified in the length modifier, or
- If no explicit length is specified, by the implicit length according to the type of constant defined (see Appendix B, "Summary of constants," on page 359).

The padding and truncation rules apply to **single** nominal values.

Padding

If more space is specified than is needed to accommodate the binary representation of the nominal value, the extra space is padded:

- With binary zeros on the left for the binary (B), hexadecimal (X), fixed-point (H,F), packed decimal (P), and all address (A,Y,S,V,J,Q,R) constants having relocatable arguments.
- With sign extension for constants having constant arguments that support sign extension of the nominal value (H, F, Y, A), as described in Table 26 on page 129.
- With ASCII spaces on the right (X'20') for CA-type character constants.
- With EBCDIC zeros on the left (X'F0') for the zoned decimal (Z) constants.
- With EBCDIC spaces on the right (X'40') for the character (C and CE-type) constants.
- With EBCDIC spaces on the right (X'40') for the Unicode character (CU) constant prior to translation.
- With double-byte spaces on the right (X'4040') for the graphic (G) constants.

Notes:

1. In floating-point constants (E,D,L), the fraction is extended to occupy the extra space available.
2. Padding is on the left for all constants except the character constant and the graphic constant.

Truncation

If less space is available than is needed to accommodate the nominal value, the nominal value is truncated and part of the constant is lost. Truncation of the nominal value is:

- On the left for the binary (B), hexadecimal (X), fixed-point (H and F), and decimal (P and Z)
- On the right for the character (C) constant, the Unicode character (CU) constant, and the graphic (G) constant
- On the left for absolute or relocatable address (A and Y), the external address (V), offset (Q), length (J) and PSECT address (R) constants. The actual value stored and any possible truncation is dependent on the values inserted by the linker/binder and the length of the constant.

Notes:

1. If significant bits are lost in the truncation of fixed-point constants, error diagnostic message `ASMA072E Data item too large` is issued.
2. Floating-point constants (E, D, L) are not truncated. They are rounded to fit the space available—see Figure 22 on page 141 for rounding modes.
3. The above rules for padding and truncation also apply when using the bit-length modifier (see "Subfield 5: Modifier" on page 118).
4. Double-byte data in C-type constants cannot be truncated because truncation creates incorrect double-byte data. Error `ASMA208E Truncation into double-byte data is not permitted` is issued if such truncation is attempted.
5. Truncation of double-byte data in CU-type and G-type constants is permitted because the length modifier restrictions (see "Subfield 5: Modifier" on page 118) ensure that incorrect double-byte data cannot be created by truncation. However, truncating bit-length constants might create incorrect double-byte data.

Subfield 1: Duplication Factor

The syntax for coding the *duplication factor* is shown in the subfield format `operands` in "DC instruction" on page 109.

You can omit the duplication factor. If specified, it causes the nominal value or multiple nominal values specified in a constant to be generated the number of times indicated by the factor. It is applied after the nominal value or values are assembled into the constant. Symbols used in subfield 1 need not be previously defined. This does not apply to literals.

The duplication factor can be specified by an unsigned decimal self-defining term or by an absolute expression enclosed in parentheses.

The factor must have a positive value or be equal to zero.

Notes:

1. A duplication factor of zero is permitted, except for literals, with the following results:
   - No value is assembled.
   - Alignment is forced according to the type of constant specified, if no length attribute is present (see "Alignment of constants" on page 111).
   - The length attribute of the symbol naming the constant is established according to the implicitly or explicitly specified length.

   When the duplication factor is zero, the nominal value can be omitted. The alignment is forced, even if the NOALIGN option is specified.

   When the duplication factor is zero for a literal, the assembler issues message `ASMA067S Illegal duplication factor`.
2. If duplication is specified for an address constant whose nominal value contains a location counter reference, the value of the location counter reference is incremented by the length of the constant before each duplication is done (see "Address constants—A and Y" on page 133). If the duplication factor is zero, the value of the location counter reference is not incremented by the length of each constant that was generated for a non-zero duplication factor. Thus, in the following two statements, the first generates an ASMA072E error message for "Data item too large", but the second does not:

   ```
   A        DC 0Y(0,32768-(*-A))
   B        DC  Y(0,32768-(*-B))
   ```

   However, if duplication is specified for an address-type literal constant containing a location counter reference, the value of the location counter reference is not incremented by the length of the literal before each duplication is done. The value of the location counter reference is the location of the first byte of the literal in the literal pool, and is the same for each duplication.

   The location counter value is that of the instruction in which the literal appears for A-type constants, but for S-type constants it is the location where the literal appears.

   When a bit-length constant of type A, B, F, H, P, X, Y, or Z is specified with a duplication factor:
   - Each nominal value is right-aligned in the specified field.
   - Each nominal value is padded on the left with zeros or sign bits, according to the type.

   If unfilled bits remain after each constant is generated, any remaining bits in the last byte are filled with zero bits. That is, padding within a constant is different from filling after a group of constants.
3. If a bit-length constant is specified with a duplication factor, each nominal value is right-aligned in the specified field and padded on the left with zeros or sign bits, according to the type. If unfilled bits remain after each constant is generated, any remaining bits in the last byte are filled with zero bits. Thus, padding within a constant is different from padding after a group of constants.
4. The maximum value for the duplication factor is 2^{24}-1, or X'FFFFFF' for OBJ object files, 2^{32}-1, or X'7FFFFFFF' for GOFF object files. If the maximum value for the duplication factor is exceeded, the assembler issues a message. Possibilities include `ASMA067S Illegal duplication factor` and `ASMA068S Length error`.

Subfield 2: Type

The syntax for coding the *type* is shown in the subfield format `operands` in "DC instruction" on page 109.

You must specify the type subfield. From the type specification, the assembler determines how to interpret the constant and translate it into the correct format. The type is specified by a single-letter code as shown in Table 19 on page 116, the type extension as shown in Table 20 on page 116.

Further information about these constants is provided in the discussion of the constants themselves under "Subfield 6: Nominal Value" on page 121.

Table 19. Type codes for constants

Code	Constant Type	Machine Format
C	Character	8 bit code for each character
G	Graphic	16 bit code for each character
X	Hexadecimal	4 bit code for each hexadecimal digit
B	Binary	Binary format
F	Fixed-point	Signed, fixed-point binary format; normally a fullword
H	Fixed-point	Signed, fixed-point binary format; normally a halfword
E	Floating-point	Short floating-point format; normally a fullword
D	Floating-point	Long floating-point format; normally a doubleword
L	Floating-point	Extended floating-point format; normally two doublewords
P	Decimal	Packed decimal format
Z	Decimal	Zoned decimal format
A	Address	Value of address; normally a fullword
Y	Address	Value of address; normally a halfword
S	Address	Base register and displacement value; a halfword
V	Address	Space reserved for external symbol addresses; normally a fullword
J	Address	Space reserved for length of class or DXD; normally a fullword
Q	Address	Space reserved for external dummy section offset
R	Address	Space reserved for PSECT addresses; normally a fullword

The type, with an optional type extension specification, indicates to the assembler:

1. How to assemble the nominal values specified in subfield 6; that is, which binary representation or machine format the object code of the constant must have.
2. At what boundary the assembler aligns the constant, if no length modifier is present.
3. How much storage the constant occupies, according to the implicit length of the constant, if no explicit length modifier is present (for details, see "Padding and truncation of values" on page 113).

Subfield 3: Type Extension

The syntax for coding the *type extension* is shown in the subfield format `operands` in "DC instruction" on page 109.

You can omit the type extension subfield. If specified, the assembler, using this field with the type subfield, determines how to interpret the constant and translate it into the correct format. The type extension is specified by a single-letter code as shown in Table 20.

Table 20. Type extension codes for constants

Type	Type Extension	Description
C	A	ASCII character constant
	E	EBCDIC character constant
	U	Unicode UTF-16 character constant
E	H	Hexadecimal floating-point constant
	B	Binary floating-point constant
	D	Decimal floating-point constant
D	H	Hexadecimal floating-point constant

Table 20. Type extension codes for constants (continued)

Type	Type Extension	Description
	B	Binary floating-point constant
	D	Decimal floating-point constant
L	H	Hexadecimal floating-point constant
	B	Binary floating-point constant
	D	Decimal floating-point constant
	Q	Hexadecimal floating-point, quadword alignment
F	D	Doubleword fixed-point constant
A	D	Doubleword address constant
V	D	Doubleword address constant
J	D	Doubleword address constant
Q	D	Doubleword address constant
Q	Y	20 bit address constant (GOFF only)
R	D	Doubleword address constant
S	Y	20 bit address constant

The type extension specification, with the type subfield, indicates to the assembler:

1. How to assemble the nominal values specified in subfield 6; that is, which binary representation or machine format the object code of the constant must have.
2. At what boundary the assembler aligns the constant, if no length modifier is present.
3. How much storage the constant occupies, according to the implicit length of the constant, if no explicit length modifier is present (for details, see "Padding and truncation of values" on page 113).

Subfield 4: Program type

The syntax for coding the *program type* is shown in the subfield format `operands` in "DC instruction" on page 109.

You can omit the program type subfield. If specified, the assembler assigns the value to the symbol naming the DC instruction, if a symbol was present. It can be specified as a decimal, character, hex, or binary self-defining term and is stored as a 32 bit value. The value is not used in any way by the assembler, and can be queried by using the SYSATTRP built-in function.

The program type is specified within a P prefixed set of parenthesis - P(). For example:

```
Prog1   DC  CP(7)'Perth'      Program type is 7
Prog2   DC  3XP(C'APC')'FF'   Program type is C'APC'
```

Symbols used in subfield 4 need not be previously defined, except in literals. For example:

```
PV       EQU   240
         LA    1,=FP(PV)'99'    Literal
SYM      DC    FP(Rate5)'35.92'
Rate5    EQU   5
```

All expressions in program type must be evaluatable when the DC is processed.

If program type is omitted, the assembler assigns a null to the program type, and querying the value using the SYSATTRP built-in function returns a null value.

If there are multiple operands and the first has no P-type, but one of the subsequent operands does have a P-type, then the program type is assigned from the first operand specifying a program type value. For example:

```
alabel   dc   fp(1)'1',hp(33)'32760'
```

results in a program type of 33 being assigned to `alabel`.

Subfield 5: Modifier

The syntax for coding the *modifier* is shown in the subfield format `operands` in "DC instruction" on page 109.

You can omit the modifier subfield. Modifiers describe the length in bits or bytes you want for a constant (in contrast to an implied length), and the scaling and exponent for the constant.

The three modifiers are:

- The length modifier (L), that explicitly defines the length in bytes you want for a constant. For example:

  ```
  LENGTH   DC              XL10'FF'
  ```

- The scale modifier (S), that is only used with the fixed-point or floating-point constants (for details, see "Scale modifier" on page 120). For example:

  ```
  SCALE    DC              FS8'35.92'
  ```

- The exponent modifier (E), that is only used with fixed-point or floating-point constants, and indicates the power of 10 by which the constant is to be multiplied before conversion to its internal binary format. For example:

  ```
  EXPON    DC              EE3'3.414'
  ```

If multiple modifiers are used, they must appear in this sequence: length, scale, and exponent. For example:

```
ALL3     DC              DL7S3E50'2.7182'
```

Symbols used in subfield 5 need not be previously defined, except in literals. For example:

```
SYM      DC              FS(X)'35.92'
X        EQU             7
```

Length modifier

The length modifier indicates the number of bytes of storage into which the constant is to be assembled. It is written as L*n*, where *n* is either a decimal self-defining term or an absolute expression enclosed by parentheses. It must have a positive value.

When the length modifier is specified:

- Its value determines the number of bytes of storage allocated to a constant. It therefore determines whether the nominal value of a constant must be padded or truncated to fit into the space allocated (see "Padding and truncation of values" on page 113).
- No boundary alignment, according to constant type, is provided (see "Alignment of constants" on page 111).
- Its value must not exceed the maximum length allowed for the various types of constant defined.
- The length modifier must not truncate double-byte data in a C-type constant, except for bit-length modifiers.
- The length modifier must be a multiple of 2 in a G-type or CU-type constant.

When no length is specified, for character and graphic constants (C and G), hexadecimal constants (X), binary constants (B), and decimal constants (P and Z), the whole constant is assembled into its implicit length.

Bit-length modifier: The length modifier can be specified to indicate the number of bits into which a constant is to be assembled. The bit-length modifier is written as L.*n* where *n* is either a decimal self-defining term, or an absolute expression enclosed in parentheses. It must have a positive value. Such a modifier is sometimes called a "bit-length" modifier, to distinguish it from a "byte-length" modifier. You cannot combine byte-length and bit-length modifiers. For example, a 12 bit field must be written L.12, not L1.4.

The value of n is 1 - the number of bits (a multiple of 8) that are required to make up the maximum number of bytes allowed in the type of constant being defined. The bit-length modifier can never be used with the CU-, G-, S-, V-, R-, J- and Q-type constants, and cannot be used with the A-type or Y-type constant if the operand is simply or complexly relocatable.

When only one operand and one nominal value are specified in a DC instruction, the following rules apply:

1. The bit-length modifier allocates a field into which a constant is to be assembled. The field starts at a byte boundary and can run over one or more byte boundaries, if the bit length is greater than 8.

   If the field does not end at a byte boundary and if the bit length is not a multiple of 8, the remainder of the last byte is filled with binary zeros. For example, `DC FL.12'-1'` generates `X'FFF0'`.
2. The nominal value of the constant is assembled into the field:
   a. Starting at the high-order end for the C-, E-, D-, and L-type constants
   b. Starting at the low-order end for the remaining types of constants that support a bit-length modifier
3. The nominal value is padded or truncated to fit the field (see "Padding and truncation of values" on page 113).

   "padding" is not the same as "filling". In padding, the designated bit field is completed according to the rules for the constant type. Filling is always binary zeros placed at the right end of an incomplete byte.

   C-type character constants are padded with EBCDIC spaces (hexadecimal X'40', and CA-type character constants are padded with ASCII spaces (hexadecimal X'20'). Other constant types are padded either by sign extension or with zeros, according to the type of the constant.

The length attribute value of the symbol naming a DC instruction with a specified bit length is equal to the minimum number of integral bytes needed to contain the bit length specified for the constant. Consider the following example:

```
TRUNCF   DC              FL.12'276'
```

`L'TRUNCF` is equal to 2. Thus, a reference to `TRUNCF` addresses both the 2 bytes that are assembled.

When more than one operand is specified in a DC instruction, or more than one nominal value in a DC operand, the above rules about bit-length modifiers also apply, except:

1. The first field allocated starts at a byte boundary, but the succeeding fields start at the next available bit. For example, `BL1 DC FL.12'-1,1000'` generates `X'FFF3E8'`.
2. After all the constants have been assembled into their respective fields, the bits remaining to make up the last byte are filled with zeros. For example, `BL2 DC FL.12'-1,1000,-2'` generates `X'FFF3E8FFE0'`

   If duplication is specified, filling with zeros occurs once at the end of all the fields occupied by the duplicated constants. For example, `BL3 DC 3FL.12'-2'` generates `X'FFEFFEFFE0'`.
3. The length attribute value of the symbol naming the DC instruction is equal to the number of integral bytes needed to contain the bit length specified for the first constant to be assembled. For example, the symbols BL1, BL2, and BL3 in the preceding examples each have length attribute 2.

For double-byte data in C-type constants: If bit-lengths are specified, with a duplication factor greater than 1, and a bit-length which is not a multiple of 8, then the double-byte data is no longer valid for devices capable of presenting DBCS characters. No error message is issued.

Storage requirement for constants: The total amount of storage required to assemble a DC instruction is the sum of:

1. The requirements for the individual DC operands specified in the instruction. The requirement of a DC operand is the product of:
   - The sum of the lengths (implicit or explicit) of each nominal value
   - The duplication factor, if specified
2. The number of bytes skipped for the boundary alignment between different operands; such skipped bytes are filled with binary zeros.

Scale modifier

The scale modifier specifies the amount of internal scaling that you want for:

- Binary digits for fixed-point constants (H, F)
- Hexadecimal digits for floating-point constants (E, D, L)

The scale modifier can be used only with the above types of constants. It cannot be used with EB, DB, and LB floating point constants.

The range for each type of constant is:

Fixed-point constant H
: -187 to +346

Fixed-point constant F
: -187 to +346

Floating-point constant E, EH
: 0 to 14

Floating-point constant D, DH
: 0 to 14

Floating-point constant L, LH
: 0 to 28

The scale modifier is written as S*n*, where *n* is either a decimal self-defining term, or an absolute expression enclosed in parentheses. Both forms of the modifier's value n can be preceded by a sign; if no sign is present, a plus sign is assumed.

Scale modifier for fixed-point constants: The scale modifier for fixed-point constants specifies the power of two by which the fixed-point constant must be multiplied after its nominal value has been converted to its binary representation, but before it is assembled in its final *scaled* form. Scaling causes the binary point to move from its assumed fixed position at the right of the rightmost bit position.

Notes:

1. When the scale modifier has a positive value, it indicates the number of binary positions occupied by the fractional portion of the binary number.
2. When the scale modifier has a negative value, it indicates the number of binary positions deleted from the integer portion of the binary number.
3. When low-order positions are lost because of scaling (or lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position saved.

Scale modifier for hexadecimal floating-point constants: The scale modifier for hexadecimal floating-point constants must have a positive value. It specifies the number of hexadecimal positions that the fractional portion of the binary representation of a floating-point constant is shifted to the right. The hexadecimal point is assumed to be fixed at the left of the leftmost position in the fractional field. When scaling is specified, it causes an unnormalized hexadecimal fraction to be assembled (unnormalized means that the leftmost positions of the fraction contain hexadecimal zeros). The magnitude of the constant is retained, because the exponent in the characteristic portion of the constant is adjusted upward accordingly. When non-zero hexadecimal positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost position saved.

Exponent modifier

The exponent modifier specifies the power of 10 by which the nominal value of a constant is to be multiplied before it is converted to its internal binary representation. It can only be used with the fixed-point (H and F) and floating-point (E, D, and L) constants. The exponent modifier is written as E*n*, where *n* can be either a decimal self-defining term, or an absolute expression enclosed in parentheses.

The decimal self-defining term or the expression can be preceded by a sign. If no sign is present, a plus sign is assumed. The range for the exponent modifier is -85 to +75. If a type extension is used to define a floating-point constant, the exponent modifier can be in the range -2^{31} to $2^{31}-1$. If the nominal value cannot be represented exactly, a warning message is issued.

Notes:

1. Do not confuse the exponent modifier with the exponent that can be specified in the nominal value subfield of fixed-point and floating-point constants.

   The exponent modifier affects each nominal value specified in the operand, whereas the exponent written as part of the nominal value subfield only affects the nominal value it follows. If both types of exponent are specified in a DC operand, their values are added together before the nominal value is converted to binary form. However, this sum must lie within the permissible range of -85 to +75, unless a type extension is specified.
2. The value of the constant, after any exponents have been applied, must be contained in the implicitly or explicitly specified length of the constant to be assembled.

Subfield 6: Nominal Value

The syntax for coding the *nominal value* is shown in the subfield format `operands` in "DC instruction" on page 109.

You must specify the nominal value subfield unless a duplication value of zero is specified. It defines the value of the constant (or constants) described and affected by the subfields that precede it. It is this value that is assembled into the internal binary representation of the constant. Table 21 shows the formats for specifying constants.

Table 21. Specifying constant values

Constant Type	Single Nominal Value	Multiple Nominal Value	Page No.
C	'value'	not allowed	123
G	'<.v.a.l.u.e>'	not allowed	126
	'value'	'value,value,...value'	
B			122
X			127
H			128
F			128
P			131
Z			131
E			141, 145
D			141, 145
L			141, 145
	(value)	(value,value,...value)	
A			132
Y			132
S			132
V			132
R			132
Q	(value)	(value,value,...value)	139

Table 21. Specifying constant values (continued)

Constant Type	Single Nominal Value	Multiple Nominal Value	Page No.
J	(value)	(value,value,...value)	140

As the above list shows:

- A data constant value (any type except A, Y, S, Q, J, R, and V) is enclosed by apostrophes.
- An address constant value (type A, Y, S, V, R) or an offset constant (type Q) or a length constant (type J) is enclosed by parentheses.
- To specify two or more values in the subfield, the values must be separated by commas, and the whole sequence of values must be enclosed by the correct delimiters; that is, apostrophes or parentheses.
- Multiple values are not permitted for character constants.

Spaces are allowed and ignored in nominal values for the quoted constant types (BDEFHLPXZ). Spaces are significant for C and G constant types.

How nominal values are specified and interpreted by the assembler is explained in each of the subsections that follow. There is a subsection for each of the following types of constant:

Binary
Character
Graphic
Hexadecimal
Fixed-Point
Decimal
Packed Decimal
Zoned Decimal
Address
Floating-Point

Literal constants are described in "Literal constants" on page 151.

Binary constant—B

The binary constant specifies the precise bit pattern assembled into storage. Each binary constant is assembled into the integral number of bytes (see **1** in Table 22) required to contain the bits specified, unless a bit-length modifier is specified.

The following example shows the coding used to designate a binary constant. `BCON` has a length attribute of 1.

```
BCON     DC             B'11011101'
BTRUNC   DC             BL1'100100011'
BPAD     DC             BL1'101'
BFOUR    DC             B'1111 0100 1111 0100'
```

`BTRUNC` assembles with the leftmost bit truncated, as follows:

```
00100011
```

`BPAD` assembles with five zeros as padding, as follows:

```
00000101
```

Table 22. Binary constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		

Table 22. Binary constants (continued)

Subfield	Value	Example	Result
2. **Type**	B		
3. **Type Extension**	Not Allowed		
4. **Program type**	Allowed		
5. **Modifiers**			
Implicit length: (length modifier not present)	As needed	`B DC B'10101111'` `C DC B'101'`	`L'B = 1` **1** `L'C = 1` **1**
Alignment:	Byte		
Range for length:	1 to 256 (byte length) .1 to .2048 (bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	Binary digits (0 or 1)		
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	At left		

Character constant—C

The character constant specifies character strings, such as error messages, identifiers, or other text, that the assembler converts into binary representations. If no type extension is provided, then the constant might be changed, depending on the value of the TRANSLATE option. If the type extension of "E" is provided, then the representation is also EBCDIC, but it cannot be changed by the TRANSLATE option. For information about type extension "A" see "ASCII data in character constants" on page 125, and for information about type extension "U" see "Unicode UTF-16 data from character constants" on page 125.

Any of the 256 characters from the EBCDIC character set can be designated in a character constant. Each character specified in the nominal value subfield is assembled into one byte (see **1** in Table 23 on page 124). For more information, see the discussion about the 82 invariant characters in "Character self-defining term" on page 31.

A null nominal value is permitted if a length is specified. For example:

```
DC              CL3''
```

is assembled as three EBCDIC spaces with object code X'404040', whereas

```
DC              CAL3''
```

is assembled as three ASCII spaces with object code X'202020'.

Multiple nominal values are not allowed because a comma in the nominal value is considered a valid character (see **2** in Table 23) and is assembled into its binary (EBCDIC) representation (see Appendix D, "Standard character set code table," on page 375). For example:

```
DC              C'A,B'
```

is assembled as `A,B` with object code X'C16BC2'.

Give special consideration to representing apostrophes and ampersands as characters. Each apostrophe or ampersand you want as a character in the constant must be represented by a pair of apostrophes or ampersands. Each pair of apostrophes is assembled as one apostrophe, and each pair of ampersands is assembled as one ampersand (see **3** in Table 23).

Table 23. Character constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	C		

3. **Type Extension**	U
	A
	E
		`U  DC  CU'UNICODE'`
		`A  DC  CA'ASCII'`
		`E  DC  CE'EBCDIC'`
			`L'U = 14`
			`L'A = 5`
			`L'E = 6`

4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	Evaluate as an even number, if Type Extension of U is specified	`C  DC  C'LENGTH'`	`L'C = 6` **1**
Alignment:	Byte		
Range for length:	1 to 256 (byte length) Must be a multiple of 2 when the Type Extension is U .1 to .2048 (bit length) (Not permitted if Type Extension of U is specified.)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		

6. **Nominal value**
Represented by:
	Characters (all 256 eight bit combinations)	`DC C'A''B'`
		`DC CU'AA'`
		`DC CA'AB'`
			Object code
			`X'C17DC2'` **3**
			`X'00410041'`
			`X'4142'`

Enclosed by:	Apostrophes		
Exponent allowed:	No (is interpreted as character data)		
Number of values per operand:	One	`DC C'A,B'`	Object code `X'C16BC2'` **2**
Padding:	With spaces at right (X'40' EBCDIC, X'20' ASCII)		

Table 23. Character constants (continued)

Subfield	Value	Example	Result
Truncation of assembled value:	At right		

In the following example, the length attribute of FIELD is 12:

```
FIELD    DC             C'TOTAL IS 110'
```

However, in this next example, the length attribute is 15, and three spaces appear in storage to the right of the zero:

```
FIELD    DC             CL15'TOTAL IS 110'
```

In the next example, the length attribute of FIELD is 12, although 13 characters appear in the operand. The two ampersands are paired, and so count as only one byte.

```
FIELD    DC             C'TOTAL IS &&10'
```

In the next example, a length of 4 has been specified, but there are five characters in the constant.

```
FIELD    DC             3CL4'ABCDE'
```

The generated constant is:

```
ABCDABCDABCD
```

The same constant can be specified as a literal.

```
         MVC            AREA(12),=3CL4'ABCDE'
```

On the other hand, if the length modifier is specified as 6 instead of 4, the generated constant is:

```
ABCDE ABCDE ABCDE (with one trailing space)
```

ASCII data in character constants: For Character ASCII (CA) constants the character string is converted to ASCII (code page 819), assuming that the characters in the nominal value are represented in code page 37. Any paired occurrences of ampersands and apostrophes are converted to a single occurrence of such a character prior to conversion. The assembler then maps each EBCDIC character into its ASCII equivalent. This constant is not modified by the TRANSLATE option.

Unicode UTF-16 data from character constants: For Character Unicode (CU) constants the value is converted to Unicode UTF-16 using the code page identified by the CODEPAGE assembler option. Any paired occurrences of ampersands and apostrophes are converted to a single occurrence of such a character prior to conversion. If necessary the value is padded with EBCDIC spaces on the right (X'40'). The assembler then maps each EBCDIC character into its 2 byte Unicode UTF-16 equivalent.

For example:

```
UA     DC       CU'UTF-16'        object code X' 005500540046002D00310036'
UB     DC       CUL4'L'           object code X' 004C0020'
UC     DC       CUL2'XYZ'         object code X' 0058'
```

Double-byte data in character constants: When the DBCS assembler option is specified, double-byte data can be used in a character constant. The start of double-byte data is delimited by SO, and the end by SI. All characters between SO and SI must be valid double-byte characters. No single-byte meaning is drawn from the double-byte data. Hence, special characters such as the apostrophe and ampersand are not recognized between SO and SI. The SO and SI are included in the assembled representation of a character constant containing double-byte data.

If a duplication factor is used, SI/SO pairs at the duplication points are not removed. For example, the statement:

```
DBCS     DC              3C'<D1>'
```

results in the assembled character string value of:

```
<D1><D1><D1>
```

Null double-byte data (SO followed immediately by SI) is acceptable and is assembled into the constant value.

The following examples of character constants contain double-byte data:

```
DBCS0    DC              C'<>'
DBCS1    DC              C'<.D.B.C.S>'
DBCS2    DC              C'abc<.A.B.C>'
DBCS3    DC              C'abc<.A.B.C>def'
```

The length attribute includes the SO and SI. For example, the length attribute of DBCS0 is 2, and the length attribute of DBCS2 is 11. No truncation of double-byte character strings within C-type constants is allowed, since incorrect double-byte data is created.

It is possible to generate invalid DBCS data in these situations:

- You specify a bit-length modifier that causes truncation of the DBCS data or the shift-out and shift-in characters.
- You specify the TRANSLATE option.

Graphic constant—G

When the DBCS assembler option is specified, the graphic (G-type) constant is supported. This constant type allows the assembly of pure double-byte data. The graphic constant differs from a character constant containing only double-byte data in that the SO and SI delimiting the start and end of double-byte data are not present in the assembled value of the graphic constant. Because SO and SI are not assembled, if a duplication factor is used, no redundant SI/SO characters are created. For example, the statement:

```
DBCS     DC              3G'<D1>'
```

results in the assembled character string value of:

```
D1D1D1
```

Examples of graphic constants are:

```
DBCS1    DC              G'<.A.B.C>'
DBCS2    DC              GL10'<.A.B.C>'
DBCS3    DC              GL4'<.A.B.C>'
```

Because the length attribute does not include the SO and SI, the length attribute of DBCS1 is 6. The length modifier of 10 for DBCS2 causes padding of 2 double-byte spaces at the right of the nominal value. The length modifier of 4 for DBCS3 causes truncation after the first 2 double-byte characters. The length attribute of a graphic constant must be a multiple of 2.

Type attribute of G-type constant: Do not confuse the G-type constant character with the type (data) attribute of a graphic constant. The type attribute of a graphic constant is @, **not** G. See the general discussion about data attributes in “Data attributes” on page 284, and “Type attribute (T')” on page 289.

Table 24. Graphic constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed	`DC 3G'<.A>'`	Object code X'42C142C142C1'
2. **Type**	G		

Table 24. Graphic constants (continued)

Subfield	Value	Example	Result
3. **Type Extension**	Not allowed		
4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	As needed(twice the number of DBCS characters)	`GC DC G'<.A.B>'`	L'GC = 4
Alignment:	Byte		
Range for length:	2 to 256, must be multiple of 2(byte length)bit length not allowed		
6. **Nominal value** Represented by:	DBCS characters delimited by SO and SI	`DC G'<.&.'>'` `DC G'<.A><.B>'`	Object code X'4250427D' X'42C142C2'
Enclosed by:	Apostrophes		
Number of values per operand:	One	`DC G'<.A.,.B>'`	Object code X'42C1426B42C2'
Padding:	With DBCS spaces at right (X'4040')	`DC GL6'<.A>'`	Object code X'42C140404040'
Truncation of assembled value:	At right	`DC GL2'<.A.B>'`	Object code X'42C1'

Hexadecimal constant—X

Hexadecimal constants generate large bit patterns more conveniently than binary constants. Also, the hexadecimal values you specify in a source module let you compare them directly with the hexadecimal values generated for the object code and address locations printed in the program listing.

Each hexadecimal digit (see **1** in Table 25 on page 128) specified in the nominal value subfield is assembled into four bits (their binary patterns can be found in "Self-defining terms" on page 29). The implicit length in bytes of a hexadecimal constant is then half the number of hexadecimal digits specified (assuming that a high-order hexadecimal zero is added to an odd number of digits). See **2** and **3** in Table 25 on page 128.

An 8-digit hexadecimal constant provides a convenient way to set the bit pattern of a full binary word. The constant in the following example sets the first and third bytes of a word with all 1 bits.

```
        DS              0F
TEST    DC              X'FF00FF00'
```

The DS instruction sets the location counter to a fullword boundary. (See "DS instruction" on page 154.)

The next example uses a hexadecimal constant as a literal and inserts a byte of all 1 bits into the rightmost 8 bits of register 5.

```
        IC              5,=X'FF'
```

In the following example, the digit A is dropped, because 5 hexadecimal digits are specified for a length of 2 bytes:

```
ALPHACON DC             3XL2'A6F4E'         Generates 6F4E 3 times
```

The resulting constant is 6F4E, which occupies the specified 2 bytes. It is duplicated three times, as requested by the duplication factor. If it is specified as:

```
ALPHACON DC             3X'A6F4E'           Generates 0A6F4E 3 times
```

the resulting constant has a hexadecimal zero in the leftmost position.

```
0A6F4E0A6F4E0A6F4E
```

Table 25. Hexadecimal constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	X		
3. **Type Extension**	Not allowed		
4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	As needed	`X DC X'FF00A2'` `Y DC X'F00A2'`	`L'X = 3` **2** `L'Y = 3` **2**
Alignment:	Byte		
Range for length:	1 to 256 (byte length) .1 to .2048 (bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	Hexadecimal digits (0 to 9 and A to F)	`DC X'1F'` `DC X'91F'`	Object code `X'1F'` **1** `X'091F'` **3**
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	At left		

Fixed-point constants—F and H

Fixed-point constants let you introduce data that is in a form suitable for the arithmetic operations of the binary fixed-point machine instructions. The constants you define can also be automatically aligned to the correct doubleword, fullword, or halfword boundary for the instructions that refer to addresses on these boundaries (unless the NOALIGN option has been specified; see "General information about constants" on page 111). You can do algebraic operations using this type of constant because they can have positive or negative values.

A fixed-point constant is written as a decimal number, which can be followed by a decimal exponent. The format of the constant is as follows:

1. The nominal value can be a signed or unsigned (see 1 in Table 26) integer, fraction, or mixed number (see 2 Table 26) followed by a signed exponent (see 3 in Table 26). If a sign is not specified for either the number or exponent, + is assumed. To generate unsigned data, precede the numeric value with the letter U; signs are not allowed.
2. The exponent must lie within the permissible range (see 4 in Table 26). If an exponent modifier is also specified, the algebraic sum (see 5 in Table 26) of the exponent and the exponent modifier must lie within the permissible range.

Here are some examples of the range of values that can be assembled into fixed-point constants:

Length	Range of signed values that can be assembled	Range of unsigned values that can be assembled
8	-2^{63} to $2^{63}-1$	0 to $2^{64}-1$
4	-2^{31} to $2^{31}-1$	0 to $2^{32}-1$
2	-2^{15} to $2^{15}-1$	0 to $2^{16}-1$
1	-2^{7} to $2^{7}-1$	0 to $2^{8}-1$

The range of values depends on the implicitly or explicitly specified length (if scaling is disregarded). If the value specified for a particular constant does not lie within the allowable range for a given length, the constant is not assembled, but flagged as an error.

A fixed-point constant is assembled as follows:

1. The specified number, multiplied by any exponents, is converted to a binary number.
2. Scaling is done, if specified. If a scale modifier is not provided, the fractional portion of the number is lost.
3. The binary value is rounded, if necessary. The resulting number does not differ from the exact number specified by more than one in the least significant bit position at the right.
4. A negative number is carried in two's-complement form.
5. Duplication is applied after the constant has been assembled.

The example statement generates 3 fullwords of data. The location attribute of CONWRD is the address of the first byte of the first word, and the length attribute is 4, the implied length for a fullword fixed-point constant. The expression CONWRD+4 can be used to address the second constant (second word) in the field.

```
CONWRD   DC              3F'658474'
```

Table 26. Fixed-point constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	F and H		
3. **Type Extension**	D permitted with type F		
4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	 Doubleword: 8 bytes Fullword: 4 bytes Halfword: 2 bytes		
Alignment: (Length modifier not present)	Doubleword, fullword, or halfword		

Table 26. Fixed-point constants (continued)

Subfield	Value	Example	Result
Range for length:	1 to 8 (byte length) .1 to .64 (bit length)		
Range for scale:	F: -187 to +346 H: -187 to +346		
Range for exponent:	-85 to +75 **4**	`DC HE+75'2E-73'` **5**	value=2x10²
6. **Nominal value**			

Represented by:	Decimal
	digits
	(0 to 9)
		Doubleword
		`DC FD'-200'` **1**
		`DC FD'U7890123456'`
		
		Fullword
		`DC FS4'2.25'` **2**
		`DC FS4'U2.25'`
		
		Halfword:
		`DC H'+200'`
		`DC HS4'.25'`
		`DC H'U200'`
		`DC HS4'U0.25'`	

Enclosed by:	Apostrophes		

Exponent allowed:	Yes	Doubleword:
		`DC FD'2E6'`
		`DC FD'U2E6'`
		
		Fullword:
		`DC F'2E6'` **3**
		`DC F'U2E6'`
		
		Halfword:
		`DC H'2E+1'`
		`DC H'U2E+1'`	

Number of values per operand:	Multiple		
Padding:	With sign bits at left		
Truncation of assembled value:	At left (error message issued)		

In the following example, the DC statement generates a 2 byte field containing a negative constant. Scaling has been specified in order to reserve 6 bits for the fractional portion of the constant.

```
HALFCON  DC              HS6'-25.46'
```

In the following example, the constant (3.50) is multiplied by 10 to the power -2 before being converted to its binary format. The scale modifier reserves 12 bits for the fractional portion.

```
FULLCON  DC              HS12'3.50E-2'
```

The same constant can be specified as a literal:

```
         AH              7,=HS12'3.50E-2'
```

The final example specifies three constants. The scale modifier requests 4 bits for the fractional portion of each constant. The 4 bits are provided whether or not the fraction exists.

```
THREECON DC             FS4'-10,25.3,U268435455'
```

Remember that commas separate operands. For readability, use spaces instead, as shown in this example:

```
TWOCONS  DC             F'123,445'          Two constants
ONECON   DC             F'123 456'          One constant
```

Decimal constants—P and Z

The decimal constants let you introduce data in a form suitable for operations on decimal data. The packed decimal constants (P-type) are used for processing by the decimal instructions. The zoned decimal constants (Z-type) are in the form (EBCDIC representation) you can use as a print image, except for the digit in the rightmost byte.

The nominal value can be a signed (plus is assumed if the number is unsigned) decimal number. A decimal point can be written anywhere in the number, or it can be omitted. The placement of a decimal point in the definition does not affect the assembly of the constant in any way, because the decimal point is not assembled into the constant. It it only affects the integer and scaling attributes of the symbol that names the constant.

The specified digits are assumed to constitute an integer (see **1** in Table 27). You can determine correct decimal point alignment either by defining data so that the point is aligned or by selecting machine instructions that operate on the data correctly (that is, shift it for purposes of decimal point alignment).

Decimal constants are assembled as follows:

Packed decimal constants: Each digit is converted into its 4 bit binary coded decimal equivalent (see **2** in Table 27). The sign indicator (see **3** in Table 27) is assembled into the rightmost 4 bits of the constant.

Zoned decimal constants: Each digit is converted into its 8 bit EBCDIC representation (see **4** in Table 27). The sign indicator (see **5** in Table 27) replaces the first four bits of the low-order byte of the constant.

Here are the range of values that can be assembled into a decimal constant:

Type of decimal constant	Range of values that can be specified
Packed	10^{31}-1 to -10^{31}
Zoned	10^{16}-1 to -10^{16}

For both packed and zoned decimals, a plus sign is translated into the hexadecimal digit C, a minus sign into the digit D. The packed decimal constants (P-type) are used for processing by the decimal instructions.

If, in a constant with an implicit length, an even number of packed decimal digits is specified, one digit is left unpaired because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits are set to zeros and the rightmost four bits contain the unpaired (first) digit.

Table 27. Decimal constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	P and Z		
3. **Type Extension**	Not allowed		

Table 27. Decimal constants (continued)

Subfield	Value	Example	Result
4. **Program type**	Allowed		

5. **Modifiers**
Implicit length:
(length modifier not present)
	As needed	Packed:
		`P DC P'+593'`
		
		Zoned:
		`Z DC Z'-593'`
			
			L'P = 2
			
			L'Z= 3

Alignment:	Byte		
Range for length:	1 to 16 (byte length) .1 to .128 (bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	Decimal digits (0 to 9)	Packed: `DC P'5.5'` 1 `DC P'55'` 1 `DC P'+555'` 2 `DC P'-777'` Zoned: `DC Z'-555'` 4	Object code X'055C' X'055C' X'555C' 3 X'777D' 3 Object code X'F5F5D5' 5
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	Packed: with binary zeros at left Zoned: with EBCDIC zeros (X'F0') at left		
Truncation of assembled value:	At left		

In the following example, the DC statement specifies both packed and zoned decimal constants. The length modifier applies to each constant in the first operand (that is, to each packed decimal constant). A literal cannot specify both operands.

```
DECIMALS DC    PL8'+25.8,-3874,+2.3',Z'+80,-3.72'
```

The last example shows the use of a packed decimal literal.

```
         UNPK          OUTAREA,=PL2'+25'
```

Address constants

An address constant is an absolute or relocatable expression, such as a storage address, that is translated into a constant. Address constants can be used for initializing base registers to facilitate the addressing of storage. Furthermore, they provide a means of communicating between control sections of a multisection

program. However, storage addressing and control section communication also depends on the USING assembler instruction and the loading of registers. See “USING instruction” on page 193.

The nominal value of an address constant, unlike other types of constants, is enclosed in parentheses. If two or more address constants are specified in an operand, they are separated by commas, and the whole sequence is enclosed by parentheses. There are seven types of address constants: A, Y, S, R, Q, J, and V. A relocatable address constant cannot be specified with bit lengths.

Complex relocatable expressions: A complex relocatable expression can only specify an A- or Y-type address constant. These expressions contain two or more unpaired relocatable terms, or two or more negative relocatable terms in addition to any absolute or paired relocatable terms. A complex relocatable expression might consist of external symbols and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the address constant.

The following example shows how, and why, a complex relocatable expression might be used for an A or Y address constant:

```
         EXTRN           X
B        DC              A(X-*)       Offset from B to X
```

Address constants—A and Y: The following sections describe how the different types of address constants are assembled from expressions that normally represent storage addresses, and how the constants are used for addressing within and between source modules.

In the A-type and Y-type address constants, you can specify any of the three following types of assembly-time expressions whose values the assembler then computes and assembles into object code. Use this expression computation as follows:

- Relocatable expressions for addressing
- Absolute expressions for addressing and value computation
- Complex relocatable expressions to relate addresses in different source modules

Literals, which are relocatable forms, are not allowed as operands, but length, scale, and integer attribute references to literals are allowed.

Here are some examples:

```
DC   A(L'=F'1.23')
DC   A(I'=F'3.45')
DC   A(S'=FS6'7.89)
```

Notes:

1. No bit-length modifier (see **1** in Table 28 on page 134) is allowed when a relocatable or complex relocatable expression (see **2** in Table 28 on page 134) is specified. The only explicit lengths that can be specified with relocatable or complex relocatable address constants are:
   - 2-8 bytes for AD-type constants
   - 2, 3, or 4 bytes for A-type constants
   - 2 bytes for Y-type constants

   The linkage editor or binder or loader you use determines which lengths are supported. Please see the appropriate product manual for more information.

   For absolute operands, you can specify byte or bit lengths:

   - Byte lengths 1 through 8, or bit lengths .1 through .128, for A-type constants
   - Byte lengths 1 or 2, or bit lengths .1 through .16, for Y-type constants
2. The value of the location counter reference (*) when specified in an address constant varies from constant to constant, if any of the following, or a combination of the following, are specified:
   - Multiple operands
   - Multiple nominal values (see **3** in Table 28 on page 134)

- A duplication factor (see 4 in Table 28)

The location counter is incremented with the length of the previously assembled constant.

3. When the location counter reference occurs in a literal address constant, the value of the location counter is the address of the first byte of the instruction.

   The behavior of location counter references in A-type address constants is different from that in S-type address constants ("Address constant—S" on page 136).

Table 28. A and Y address constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed	`A DC 5AL1(*-A)` 4	Object code X'0001020304'
2. **Type**	A and Y		
3. **Type Extension**	D permitted for A type only		
4. **Program type**	Allowed		
5. **Modifiers**			
Implicit length: (length modifier not present)	A-type: 4 bytes AD-type: 8 bytes Y-type: 2 bytes		
Alignment: (Length modifier not present)	A-type: fullword AD-type: doubleword Y-type: halfword		
Range for length:	A-type: 2 to 4 1 (byte length) AD-type: 1 to 8 (byte length) Y-type: 2 only (byte length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value**			

Represented by:	Absolute, relocatable, or complex relocatable expressions 2	A-type:
		`DC A(ABSOL+10)`
		
		Y-type:
		`DC Y(RELOC+32)`
		`A DC Y(*-A,*+4)` 3
			values=0,A+6

Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	If an absolute term is present, by sign extension. Otherwise, with zeros at left.		
Truncation of assembled value:	At left		

Take care when using Y-type address constants and 2 byte A-type address constants for relocatable addresses, as they can only address a maximum of 65,536 bytes of storage. Using these types of address constants for relocatable addresses results in message ASMA066W being issued unless the assembler option RA2 is specified.

Here is how the A-type and Y-type address constants are processed:
- If the nominal value is an absolute expression, it is computed to its 32 bit value> it is then truncated or sign-extended on the left to fit the implicit or explicit length of the constant.
- If the nominal value is a relocatable or complex relocatable expression, it is not completely evaluated until linkage edit time. The relocated address values are then placed in the fields set aside for them at assembly time by the A-type and Y-type constants.

In the following examples, the field generated from the statement named `ACON` contains four constants, each of which occupies 4 bytes. The statement containing the `LM` instruction shows the same set of constants specified as literals (that is, address constant literals).

```
ACON     DC             A(108,LOP,END-STRT,*+4096)
         LM             4,7,=A(108,LOP,END-STRT,*+4096)
```

A location counter reference (`*`) appears in the fourth constant (`*+4096`). The value of the location counter is the address of the first byte of the fourth constant. When the location counter reference occurs in a literal, as in the `LM` instruction, the value of the location counter is the address of the first byte of the instruction.

Note: It is important to remember that expression evaluation for address constants is restricted to using 32 bit internal arithmetic. The result is then sign-extended to the length of the constant. This means that certain expressions in AD-type constants might not yield expected results, especially if the resulting value is negative.

PSECT reference—R: The R-type constant reserves storage for the address of the PSECT of *symbol1* as specified in the associated XATTR statement ("XATTR instruction (z/OS and CMS)" on page 203). It is the caller's responsibility to establish the definition of the R-type address constant referencing the called routine's PSECT, and to pass that address to the called routine. This constant is only available if the GOFF option is specified.

Note: If a program is to be reentrant, R-type address constants must not appear in shared (read-only) text. They should be in the caller's PSECT, and be provided to the called routine using an appropriate convention. That is, R-type address constants referring to PSECTs should themselves reside in PSECTs. If not, there can be only a single instantiation of the PSECT work area, and the program cannot be reentrant.

Table 29. R address constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	R		
3. **Type Extension**	D		
4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	R-type: 4 bytes RD-type: 8 bytes		
Alignment: (Length modifier not present)	R-type: Fullword RD-type: Doubleword		

Table 29. R address constants (continued)

Subfield	Value	Example	Result
Range for length:	R-type: 3 or 4 only RD-type: 3, 4, or 8 (no bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	An ordinary symbol	`DC R(PSECT1)`	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	Not applicable		

Address constant—S: Use the S-type address constant to assemble an explicit address in base-displacement form. You can specify the explicit address yourself or let the assembler compute it from an implicit address, using the current base register and address in its computation.

The nominal values can be specified in two ways:

1. As one absolute or relocatable expression (see **1** in Table 30 on page 137) representing an implicit address.
2. As two absolute expressions (see **2** in Table 30 on page 137) the first of which represents the displacement and the second, enclosed in parentheses, represents the base register.

The address value represented by the expression in **1** in Table 30 on page 137, is converted by the assembler into the correct base register and displacement value. An S-type constant is assembled as a halfword and aligned on a halfword boundary. An SY-type constant is assembled as 3 bytes and aligned on a halfword boundary. The leftmost four bits of the assembled constant represent the base register designation; the remaining 12 bits (S-type) or 20 bits (SY-type), the displacement value.

Notes:

1. The value of the location counter (*) when specified in an S-type address constant varies from constant to constant if one or more the following is specified:
   - Multiple operands
   - Multiple nominal values
   - A duplication factor

   In each case the location counter is incremented with the length of the previously assembled constant, except when multiple S-type address constants are specified in a literal. In a literal, the same location counter value is used for each of the multiple values.
2. If a length modifier is used, only 2 bytes for an S-type constant, or only 3 bytes for an SY-type constant, can be specified.
3. S-type address constants can be specified as literals. The USING instructions used to resolve them are those in effect at the place where the literal pool is assembled, and not where the literal is used.
4. The location counter value used in the literal is the value at the point where the literal is used, not where it is defined.

   For example:

```
USING *,15
DC    2S(*)        generates F000F002
LA    1,=2S(*)     generated constants are F004F004
```

This behavior is different from that in A-type address constants and Y-type address constants.

Table 30. S address constants

Subfield	Value	Example	Result	
1. **Duplication factor**	Allowed			
2. **Type**	S			
3. **Type Extension**	Y			
4. **Program type**	Allowed			
5. **Modifiers** Implicit length: (length modifier not present)	2 bytes (S-type) 3 bytes (SY-type)			
Alignment: (Length modifier not present)	Halfword			
Range for length:	2 (S) or 3(SY) only (no bit length)			
Range for scale:	Not allowed			
Range for exponent:	Not allowed			
6. **Nominal value**			Base	Disp
Represented by:	Absolute or relocatable expression **1**	`DC S(RELOC)` `DC S(1024)`	X 0	YYY 400
	Two absolute expressions **2**	`DC S(512(12))` `DC SY(-2(3))`	C 3	200 FFEFF
Enclosed by:	Parentheses			
Exponent allowed:	No			
Number of values per operand:	Multiple			
Padding:	Not applicable			
Truncation of assembled value:	Not applicable			

Address constant—V: The V-type constant reserves storage for the address of a location in a control section that is defined in another source module. Use the V-type address constant only to branch to an external address, because link-time processing might cause the branch to be *indirect* (for example, an assisted linkage in an overlay module). That is, the resolved address in a V-type address constant might *not* contain the address of the referenced symbol. In contrast, to refer to external data, use an A-type address constant whose nominal value specifies an external symbol identified by an EXTRN instruction.

Because you specify a symbol in a V-type address constant, the assembler assumes that it is an external symbol. A value of zero is assembled into the space reserved for the V-type constant; the correct relocated value of the address is inserted into this space by the linkage editor before your object program is loaded.

The symbol specified (see **1** in Table 31) in the nominal value subfield does not constitute a definition of the symbol for the source module in which the V-type address constant appears.

The symbol specified in a V-type constant must not represent external data in an overlay program.

Table 31. V address constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	V		
3. **Type Extension**	D		
4. **Program type**	Allowed		
		VL4(ExtSym)	
5. **Modifiers**			
Implicit length: (length modifier not present)	V-type: 4 bytes VD-type: 8 bytes		
Alignment: (Length modifier not present)	V-type: Fullword VD-type: Doubleword		
Range for length:			
	V-type: 4 or 3 only VD-type: 3, 4, or 8 (no bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value**			
Represented by:	A single external symbol	`DC V(MODA)` **1** `DC V(EXTADR)` **1**	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	None		
Truncation of assembled value:	Not applicable		

In the following example, 12 bytes are reserved, because there are three symbols. The value of each assembled constant is zero until the program is link-edited.

```
VCONST   DC              V(SORT,MERGE,CALC)
```

z/OS only: To specify a list of conditional external symbols to be resolved by the Binder, the following syntax is used:

```
VCONST DC V(FUNCA:FUNCB:FUNCC)
```

The Binder will attempt to resolve the external reference to FUNCA. If FUNCA is not available, the Binder attempts to resolve the reference to FUNCB. If FUNCB is not available, FUNCC. Finally, if FUNCC is not available, the external references are flagged as unresolved.

Offset and length constants

This section describes the offset constant ("Offset constant—Q") and length constant ("Length constant—J" on page 140).

Offset constant—Q: Use this constant to reserve storage for the offset into a storage area of an external dummy section, or the offset to a part or label in a class. The offset is entered into this space by the binder. The binder inserts the offset into a QY-type constant in 20 bit signed long-displacement format. When the offset is added to the address of an overall block of storage set aside for external dummy sections, it addresses the applicable section.

For a description of the use of the Q-type offset constant in combination with an external dummy section, see "External dummy sections" on page 49. See also Table 32 for details.

In the following example, to access the external dummy section named `VALUE`, the value of the constant labeled `A` is added to the base address of the block of storage allocated for external dummy sections.

```
A        DC              Q(VALUE)
```

The DXD, DSECT, or part names referenced in the Q-type offset constant need not be previously defined.

Table 32. Q offset constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	Q		
3. **Type Extension**	D, Y		
4. **Program type**	Allowed		
		Q(DXDEXT)	
5. **Modifiers** Implicit length: (length modifier not present)	Q-type: 4 bytes QD-type: 8 bytes QY-type: 3 bytes		
Alignment: (Length modifier not present)	Q-type: Fullword QD-type: Quadword QY-type: Halfword		
Range for length:	Q-type: 1-4 bytes QD-type: 1-8 bytes QY-type: 3 bytes only (no bit length)	QL2(DXDEXT)	
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	A DXD, DSECT, or part name (an external symbol)	`DC Q(DUMMYEXT)` `DC Q(DXDEXT)`	
Enclosed by:	Parentheses		
Exponent allowed:	No		

Table 32. Q offset constants (continued)

Subfield	Value	Example	Result
Number of values per operand:	Multiple		
Padding:	None		
Truncation of assembled value:	Not applicable		

Length constant—J: Use this constant to reserve storage for the length of a DXD, class, or DSECT. The assembler fills the field with binary zeros, and the length is entered into this space by the linker. This constant is only available if the GOFF option is specified.

In the following example, the value at A is the length of CLASS.

```
A        DC              J(CLASS)
```

The DXD or DSECT names referenced in the J-type length constant need not be previously defined.

Table 33. J length constants

Subfield	Value	Example	Result
1. **Duplication factor**	Allowed		
2. **Type**	J		
3. **Type Extension**	D		
4. **Program type**	Allowed		
5. **Modifiers** Implicit length: (length modifier not present)	J-type: 4 bytes JD-type: 8 bytes		
Alignment: (Length modifier not present)	J-type: Fullword JD-type: Doubleword		
Range for length:	J-type: 2 to 4 bytes, or 8 JD-type: 2 to 4 bytes, or 8 (no bit length)		
Range for scale:	Not allowed		
Range for exponent:	Not allowed		
6. **Nominal value** Represented by:	A single DXD, class, or DSECT name	DC J(CLASS)	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	None.		
Truncation of assembled value:	At left		

Hexadecimal floating-point constants—E, EH, D, DH, L, LH, LQ

Floating-point constants let you introduce data that is in the form suitable for the operations of the floating-point feature instructions. These constants have the following advantages over fixed-point constants:

- You do not have to consider the fractional portion of a value you specify, nor worry about the position of the decimal point when algebraic operations are to be done.
- You can specify both much larger and much smaller values.
- You retain greater processing precision; that is, your values are carried in more significant figures.

The nominal value can be a signed (see 1 in Table 34) integer, fraction, or mixed number (see 2 Table 34) followed by a signed exponent (see 3 in Table 34). If a sign is not specified for either the number or exponent, a plus sign is assumed.

If you specify the 'H' type extension you can also specify a rounding mode that is used when the nominal value is converted from decimal to its hexadecimal form. The syntax for nominal values (including the binary floating-point constants) is shown in Figure 24 on page 148. The valid rounding mode values are:
See 4 in Table 34.

1	Round by adding one in the first lost bit position
4	Unbiased round to nearest, with tie-breaking rule
5	Round towards zero (that is, truncate)
6	Round up towards the maximum positive value
7	Round down towards the minimum negative value

Figure 22. Rounding mode values

The exponent must lie within the permissible range. If an exponent modifier is also specified, the algebraic sum of the exponent and the exponent modifier must lie within the permissible range.

Table 34. Hexadecimal floating-point constants

Subfield	Value	Example
1. **Duplication factor**	Allowed	
2. **Type**	E, D, and L	
3. **Type Extension**	Omitted or H or Q	
4. **Program type**	Allowed	
5. **Modifiers**		
Implicit length: (length modifier not present)	E-type: 4 bytes D-type: 8 bytes L-type: 16 bytes	
Alignment: (Length modifier not present)	E-type: Fullword D-type: Doubleword L-type: Doubleword LQ-type: Quadword	

Table 34. Hexadecimal floating-point constants (continued)

Subfield	Value	Example
Range for length:	E-type: 1 to 8 (byte length) .1 to .64 (bit length) EH-type: .12 to .64 (bit length) D-type: 1 to 8 (byte length) .1 to .64 (bit length) DH-type: .12 to .64 (bit length) L-type: 1 to 16 (byte length) .1 to .128 (bit length) LH-type: .12 to .128 (bit length) LQ-type: .12 to .128 (bit length)	
Range for scale:	E-type: 0 to 5 D-type: 0 to 13 L-type: 0 to 27	
Range for exponent:	-85 to +75	
6. **Nominal value** Represented by:	Decimal digits	E-type: `DC E'+525'` 1 `DC E'5.25'` 2 D-type: `DC D'-525'` 1 `DC D'+.001'` 2 L-type: `DC L'525'` `DC L'3.414'` 2
Enclosed by:	Apostrophes	
Exponent allowed:	Yes	E-type: `DC E'1E+60'` 3 D-type: `DC D'-2.5E10'` 3 L-type: `DC L'3.712E-3'` 3
Rounding mode allowed if type extension specified:	Yes (see Figure 22 on page 141 for values)	E-type: `DC EH'1E+60R1'` 4 D-type: `DC DH'-2.5E10R4'` 4 L-type: `DC LH'3.712E-3R5'` 4
Number of values per operand:	Multiple	

Table 34. Hexadecimal floating-point constants (continued)

Subfield	Value	Example
Padding:	Correct fraction is extended to the right and rounded	
Truncation of assembled value:	Only if rounding mode 5; rounded otherwise.	

The format of the constant is shown in Figure 23 on page 144.

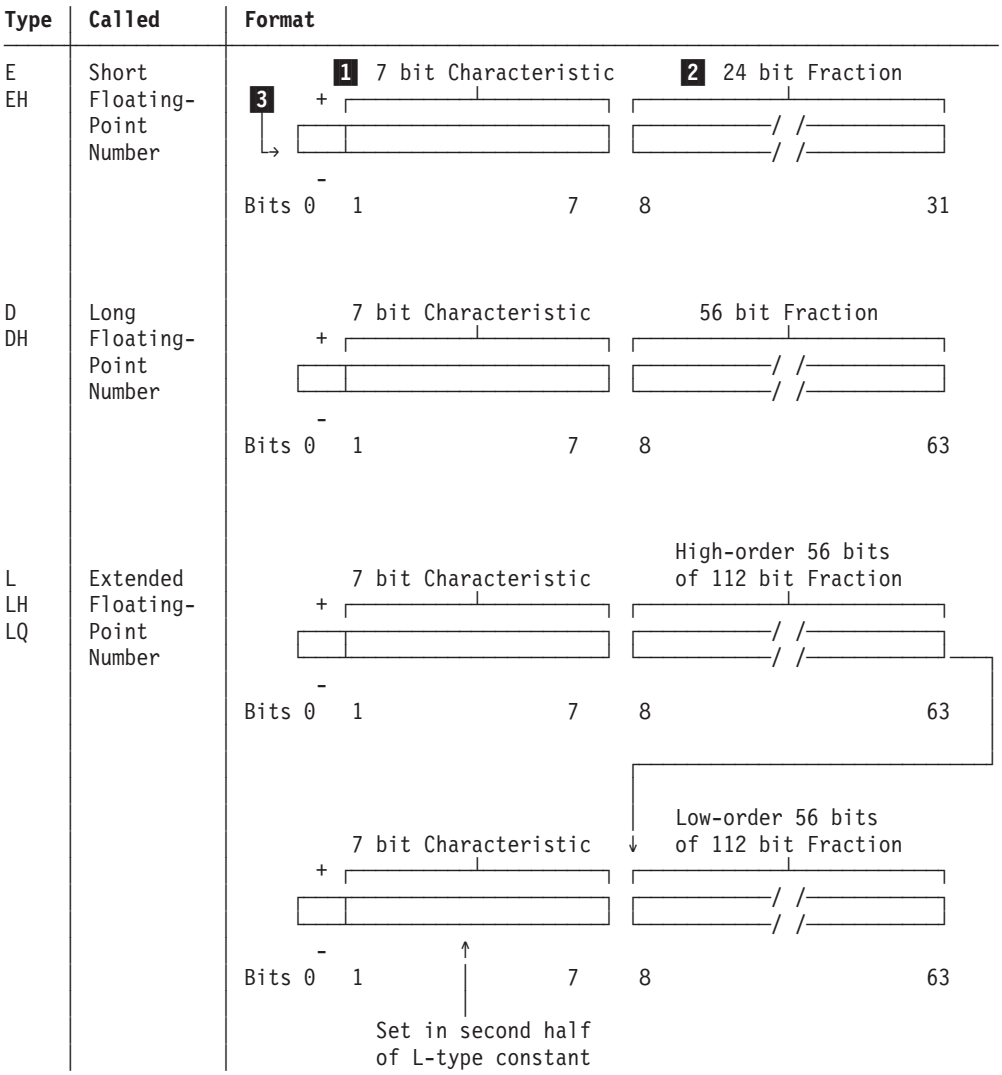
The value of the constant is represented by two parts:
- An exponent portion (see **1** in Figure 23 on page 144), followed by
- A fractional portion (see **2** in Figure 23 on page 144)

A sign bit (see **3** in Figure 23 on page 144) indicates whether a positive or negative number has been specified. The number specified must first be converted into a hexadecimal fraction before it can be assembled into the correct internal format. The quantity expressed is the product of the fraction (see **4** in Figure 23 on page 144) and the number 16 raised to a power (see **5** in Figure 23 on page 144). Figure 23 on page 144 shows the external format of the three types of floating-point constants.

Here is the range of values that can be assembled into hexadecimal floating-point constants:

Type of Constant	Range of Magnitude (M) of Values (Positive and Negative)
E	$16^{-65} \le M \le (1-16^{-6}) \times 16^{63}$
D	$16^{-65} \le M \le (1-16^{-14}) \times 16^{63}$
L	$16^{-65} \le M \le (1-16^{-28}) \times 16^{63}$
E, D, L	$5.4 \times 10^{-79} \le M \le 7.2 \times 10^{75}$ (approximate)

If the value specified for a particular constant does not lie within these ranges, the assembled value then depends on these factors:
- With type extension H, overflows assemble to the largest magnitude for the specified type, underflows denormalize the value or return zero, depending on the value and rounding mode.
- Without type extension H, certain combinations of exponents (modifier and nominal value) might produce invalid results (message ASMA071E). If the exponent is too large it is ignored, and the nominal value of the constant preceding the exponent is assembled instead.



Characteristic		Hexadecimal Fraction
5	16^E	$X\left[\frac{a}{16} + \frac{b}{16^2} + \frac{c}{16^3} + \ldots\right]$ (4)

where a,b,c ... are hexadecimal digits, and E is an exponent that has a positive or negative value indicated by the characteristic

Figure 23. Hexadecimal floating-point external formats

Representation of hexadecimal floating point: The assembler assembles a floating-point constant into its binary representation as follows: The specified number, multiplied by any exponents, is converted to the required two-part format. The value is translated into:

- A fractional portion represented by hexadecimal digits and the sign indicator. The fraction is then entered into the leftmost part of the fraction field of the constant (after rounding).
- An exponent portion represented by the excess-64 binary notation, which is then entered into the characteristic field of the constant.

The excess-64 binary notation is obtained by adding +64 to the value of the exponent (-64 - +63) to yield the characteristic (0 - 127).

Notes:

1. The L-type floating-point constant resembles two contiguous D-type constants. The sign of the second doubleword is assumed to be the same as the sign of the first.

   The characteristic for the second doubleword is equal to the characteristic for the first minus 14 (the number of hexadecimal digits in the fractional portion of the first doubleword). No indication is given if the characteristic of the second doubleword is zero.

   The L-type and LH-type floating-point constants are doubleword aligned. The LQ-type is quadword aligned. A `DC 0LQ` forces the alignment to a quadword boundary.
2. If scaling has been specified, hexadecimal zeros are added to the left of the normalized fraction (causing it to become unnormalized), and the exponent in the characteristic field is adjusted accordingly. (For further details on scaling, see "Subfield 5: Modifier" on page 118.)
3. The fraction is rounded according to the implied or explicit length of the constant. The resulting number does not differ from the exact value specified by more than one in the last place.

   Note: You can control rounding by using the 'H' type extension and specifying the rounding mode.
4. Negative fractions are carried in true representation, not in the two's-complement form.
5. Duplication is applied after the constant has been assembled.
6. An implied length of 4 bytes is assumed for a short (E) constant and eight bytes for a long (D) constant. An implied length of 16 bytes is assumed for an extended (L) constant. The constant is aligned at the correct word (E) or doubleword (D and L) boundary if a length is not specified. However, any length up to and including eight bytes (E and D) or 16 bytes (L) can be specified by a length modifier. In this case, no boundary alignment occurs.
7. Signed zero values are correctly generated for type extensions H and B. Without a type extension, zero values of either sign are assembled with positive sign.

Any of the following statements can be used to specify 46.415 as a positive, fullword, floating-point constant; the last is a machine instruction statement with a literal operand. Each of the last two constants contains an exponent modifier.

```
        DC              E'46.415'
        DC              E'46415E-3'
        DC              E'+464.15E-1'
        DC              E'+.46415E+2'
        DC              EE2'.46415'
        AE              6,=EE2'.46415'
```

The following generates 3 doubleword floating-point constants.

```
FLOAT   DC              DE(+4)'+46,-3.729,+473'
```

Binary floating-point constants—EB, DB, LB

Binary floating-point numbers are represented in three formats: short, long, or extended.

- The short format is 4 bytes with a sign of 1 bit, an exponent of 8 bits and a fraction of 23 bits.
- The long format is 8 bytes with a sign of 1 bit, an exponent of 11 bits and a fraction of 52 bits.
- The extended format is 16 bytes with a sign of 1 bit, an exponent of 15 bits and a fraction of 112 bits.

There are five classes of binary floating-point data, including numeric and related nonnumeric entities. Each data item consists of a sign, an exponent, and a significand. The exponent is biased such that all exponents are nonnegative unsigned numbers, and the minimum biased exponent is zero. The significand consists of an explicit fraction and an implicit unit bit to the left of the binary point. The sign bit is zero for plus and one for minus values.

All finite nonzero numbers within the range permitted by a given format are normalized and have a unique representation. There are no unnormalized numbers, which might allow multiple representations for the same value, and there are no unnormalized arithmetic operations. Tiny numbers of a magnitude below the minimum normalized number in a given format are represented as *denormalized* numbers, because they imply a leading zero bit, but those values are also represented uniquely.

The classes are:

1. *Zeros* have a biased exponent of zero, a zero fraction and a sign. The implied unit bit is zero.
2. *Denormalized numbers* have a biased exponent of zero and a nonzero fraction. The implied unit bit is zero.

   The smallest denormalized numbers have approximate magnitudes 1.4 10**-45 (short format), 4.94 10**-324 (long format) and 6.5 10**-4966 (extended format).
3. *Normalized numbers* have a biased exponent greater than zero but less than all ones. The implied unit bit is one and the fraction can have any value. The largest normalized numbers have approximate magnitudes 3.4 10**38 (short format), 1.8 10**308 (long format), and 1.2 10**4932 (extended format). The smallest normalized numbers have approximate magnitudes 1.18 10**-38 (short format), 2.23 10**-308 (long format), and 3.4 10**-4392 (extended format).
4. An *infinity* is represented by a biased exponent of all ones and a zero fraction.
5. A *NaN (Not-a-Number)* entity is represented by a biased exponent of all ones and a nonzero fraction. NaNs are produced in place of a numeric result after an invalid operation when there is no interruption. NaNs can also be used by the program to flag special operands, such as the contents of an uninitialized storage area. There are two types of NaNs, signaling and quiet. A signaling NaN (SNaN) is distinguished from the corresponding quiet NaN (QNaN) by the leftmost fraction bit: zero for the SNaN and one for QNaN. A special QNaN is supplied as the default result for an invalid-operation condition; it has a plus sign and a leftmost fraction bit of one, with the remaining fraction bits being set to zeros. Normally, QNaNs are just propagated during computations, so that they remain visible at the end. An SNaN operand causes an invalid operation exception.

Decimal floating-point constants—ED, DD, LD

Decimal floating-point numbers are represented in three formats: short, long, or extended.

- Short: 1 sign bit, 11 combination field bits, 20 significand continuation field bits
- Long: 1 sign bit, 13 combination field bits, 50 significand continuation field bits
- Extended: 1 sign bit, 17 combination field bits, 110 significand continuation field bits

There are four classes of decimal floating-point data, including numeric and related nonnumeric entities. Each data item consists of a sign, an exponent, and a significand. The exponent is biased such that all exponents are nonnegative unsigned numbers, and the minimum biased exponent is zero. The significand consists of an explicit fraction and an implicit unit bit to the left of the decimal point. The sign bit is zero for plus and one for minus values.

The classes are:

1. *Zeros* have a biased exponent of zero, a zero fraction and a sign. The implied unit bit is zero.
2. *Numbers* have a biased exponent greater than zero but less than all ones. The largest numbers have approximate values 10^{97} (short format), 10^{385} (long format), and 10^{1145} (extended format). The smallest numbers have approximate values 10^{-101} (short format), 10^{-398} (long format), and 10^{-6176} (extended format).
3. An *infinity* is represented if the first 5 bits of the combination field are 11110 (binary).
4. A *NaN (Not-a-Number)* is represented if the first 5 bits of the combination field are 11111 (binary). If the following bit is 1, the NaN is a Signaling NaN; otherwise it is a Quiet NaN.

The rounding modes used for decimal floating point are:

R8 Decimal floating point equivalent of binary floating point R4 (round-half-even)

R9 Decimal floating point equivalent of binary floating point R5 (truncate; round towards zero)

R10 Decimal floating point equivalent of binary floating point R6 (-> +<inf>; ceiling)

R11 Decimal floating point equivalent of binary floating point R7 (-> -<inf>; floor)

R12 Decimal floating point equivalent of binary floating point R1 (round-half-up)

R13 Round-half-down (no binary floating point equivalent)

R14 Round-up (away-from-zero, no binary floating point equivalent)

R15 Decimal floating point: round-for-reround, or 'prepare for shorter precision'.

Syntax of binary, decimal, and hexadecimal floating-point constants

The syntax for coding binary, decimal, and hexadecimal floating-point constants is:

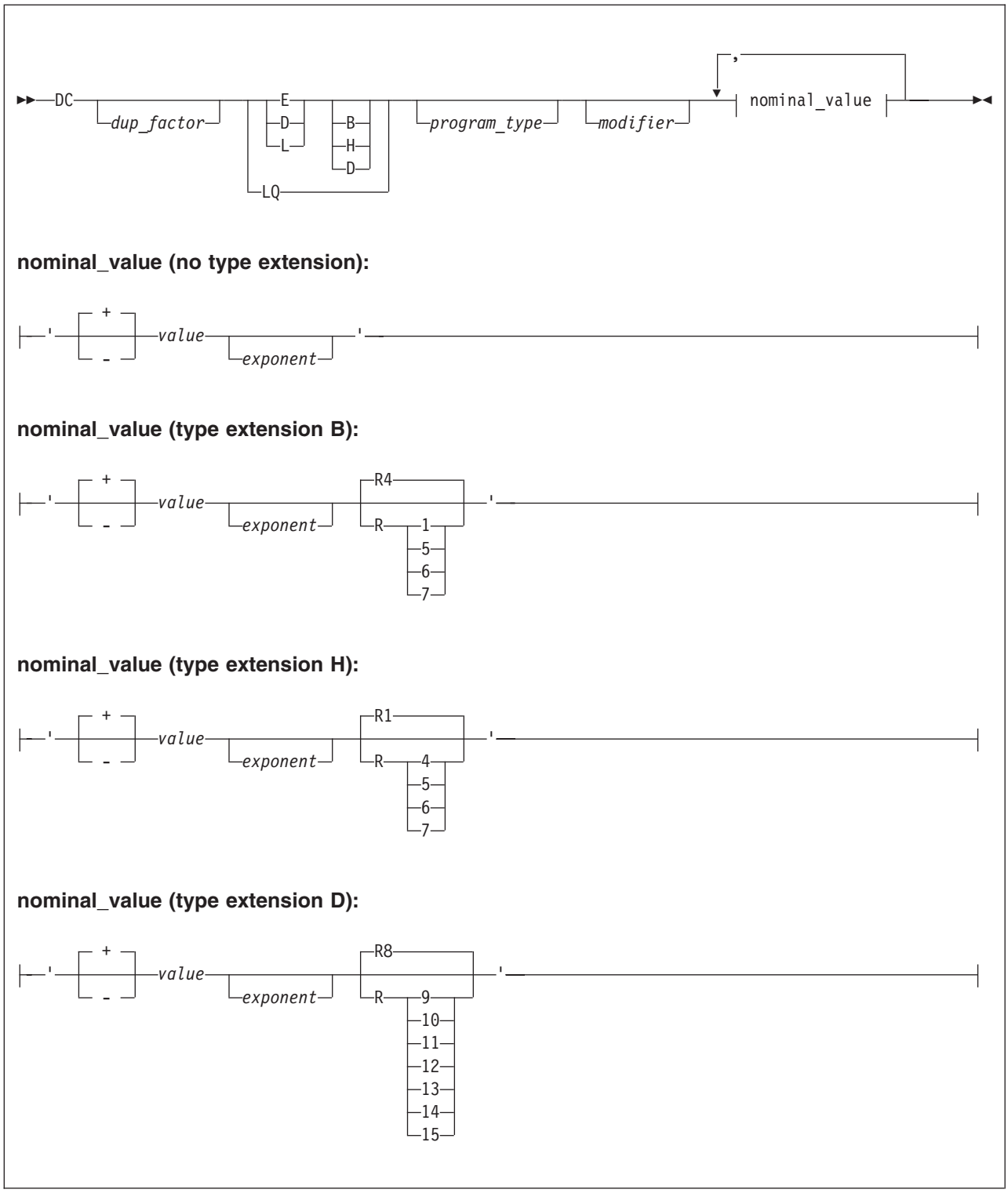


Figure 24. DC instruction syntax for floating point constants

dup_factor
Causes the constant to be generated the number of times indicated by the factor.

type
Indicates that the constant is short, long, or extended floating point.

type extension
The type of conversion required to assemble the constant. Valid values are:

null Hexadecimal floating-point constant which is converted using the conversion logic of rounding mode 1 and slightly less precise algorithms

B Binary floating-point constant which is converted allowing all rounding modes

D Decimal floating-point constant which is converted allowing all rounding modes

H Hexadecimal floating-point constant which is converted allowing all rounding modes

Q For extended-precision hexadecimal constants of type L, the Q-type extension requests alignment on a quadword boundary.

program_type
Assign a programmer determined 32 bit value to the symbol naming the DC instruction, if a symbol was present.

modifier
Describes the length, the scaling, and the exponent of the *nominal_value*. The minimum length of the 'H' hexadecimal constant is 12 bits. The minimum length in bits of the binary constant is:

9 Short floating-point constant

12 Long floating-point constant

16 Extended floating-point constant

This minimum length allows for the sign, exponent, the implied unit bit which is considered to be one for normalized numbers and zero for zeros and denormalized numbers.

The exponent modifier can be in the range from -2^{31} to $2^{31}-1$ if either B or H is specified as a type extension. The only valid length modifiers for decimal floating point constants are 4 bytes (short format), 8 bytes (long format), and 16 bytes (extended format).

nominal_value
Defines the value of the constant and can include the integer, fraction, or mixed number followed by an optional signed exponent and an optional explicit rounding mode.

The assembler imposes no limits on the exponent values that can be specified. The BFP architecture limits the actual values that can be represented; a warning message is issued whenever a specified value cannot be represented exactly.

The rounding mode identifies the rounding required when defining a floating-point constant. The valid values are those displayed in Figure 22 on page 141.

Note: As binary floating-point does not support scaling, the scale modifier is ignored and a warning message issued if the scaling modifier is specified when defining a binary floating-point constant. The H type extension causes HLASM to use a different conversion algorithm for hexadecimal floating-point data. The results are correctly rounded for all values. Without the H type extension, some rare values are in error by one unit in the last place.

Conversion to binary floating-point

For decimal to binary floating-point conversion, the assembler conforms to ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic, dated August 12, 1985, with the following differences: exception status flags are not provided and traps are not supported.

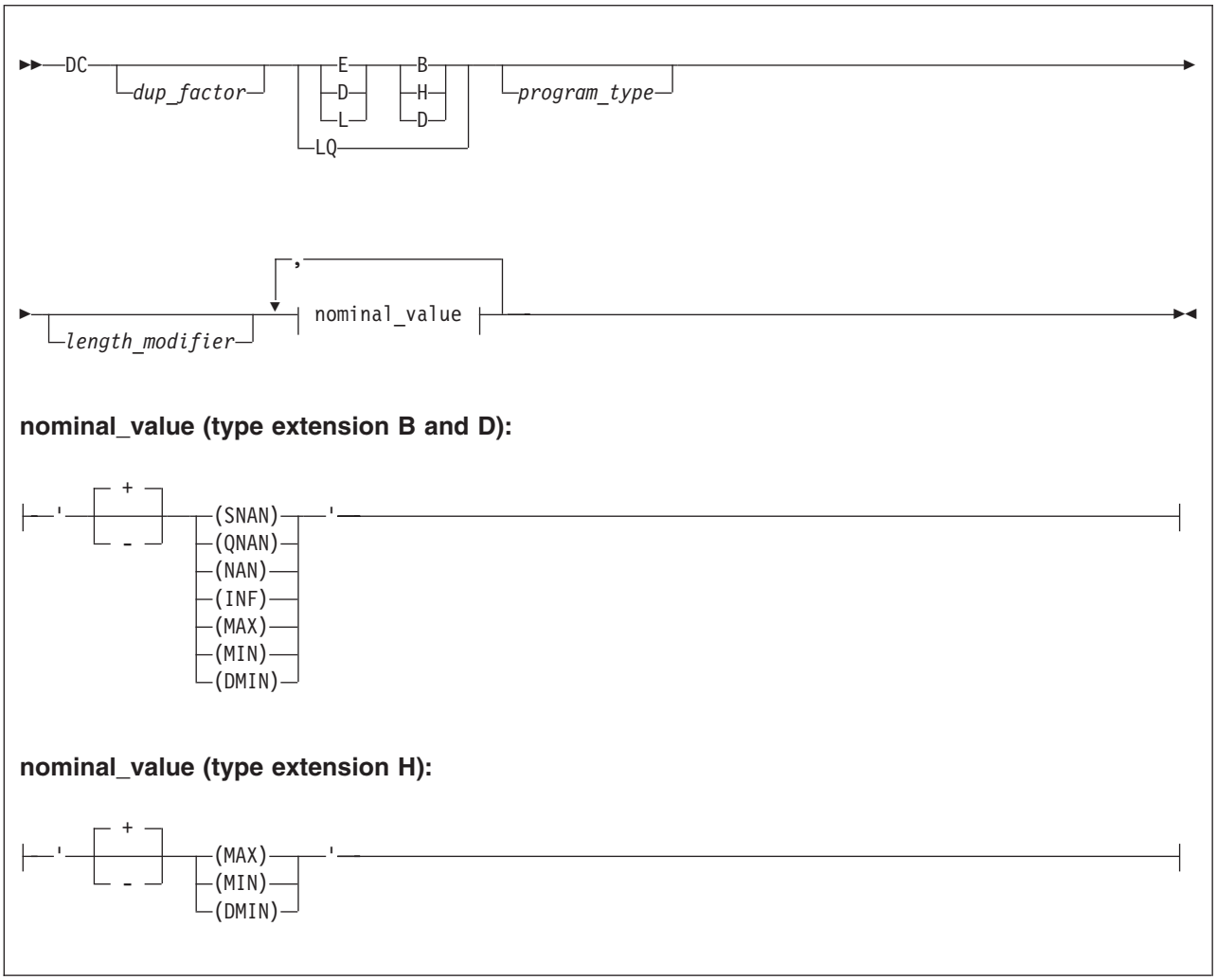
Conversion of values within the represented range is correctly rounded.

Conversion of values outside the represented range is as follows. If the resultant value before rounding is larger in magnitude than MAX (the maximum allowed value) as represented in the specified length, then,

depending on the rounding mode, either MAX or infinity is generated, along with a warning message. If the resultant nonzero value is less than Dmin (the minimum allowed value) as represented in the specified length, then, depending on the rounding mode, either Dmin or zero is generated, along with a warning message.

Floating-point special values

For special values, the syntax of the DC statement is:



dup_factor
Causes the constant to be generated the number of times indicated by the factor.

type
Indicates that the constant is short, long, or extended floating point.

type extension
The type of conversion required to assemble the constant.

program_type
Assign a programmer determined 32 bit value to the symbol naming the DC instruction, if a symbol was present.

length_modifier
Describes the length in bytes or bits into which the constant is to be assembled. For binary floating-point constants (type extension B), the minimum length in bits for INF and NAN is:

11 Short floating-point constant

14 Long floating-point constant

18 Extended floating-point constant

This minimum length allows for the sign, exponent and two fraction bits.

No length modifiers other than 4, 8, and 16 are allowed for decimal (type extension D) floating-point constants.

nominal_value
Defines the special value to be generated.

Notes:

1. The nominal value can be in mixed case.
2. SNAN assembles with an exponent of ones and 01 in the high-order fraction bits with the remainder of the fraction containing zeros.
3. QNAN assembles with an exponent of ones and 11 in the high-order fraction bits with the remainder of the fraction containing zeros.
4. NAN assembles with an exponent of one and 10 in the high-order fraction bits with the remainder of the fraction containing zeros.
5. MIN assembles as a normalized minimum value, that is an exponent of one and a fraction of zeros for binary constants, and a fraction with a leading hexadecimal digit 1 followed by zeros for hexadecimal constants.
6. DMIN assembles as a denormalized minimum value with an exponent of zeros and a fraction of all zeros except for a low-order bit of one.
7. INF assembles with an exponent of ones and a fraction of zeros.
8. MAX assembles with a fraction of all ones and an exponent of all ones for hexadecimal constants, and an exponent of all ones except for the low bit for binary constants.

Literal constants

Literal constants let you define and refer to data directly in machine instruction operands. You do not need to define a constant separately in another part of your source module. The differences between a literal, a data constant, and a self-defining term are described in “Literals” on page 35.

A literal constant is specified in the same way as the operand of a DC instruction. The general rules for the operand subfields of a DC instruction also apply to the subfield of a literal constant. Moreover, the rules that apply to the individual types of constants apply to literal constants as well.

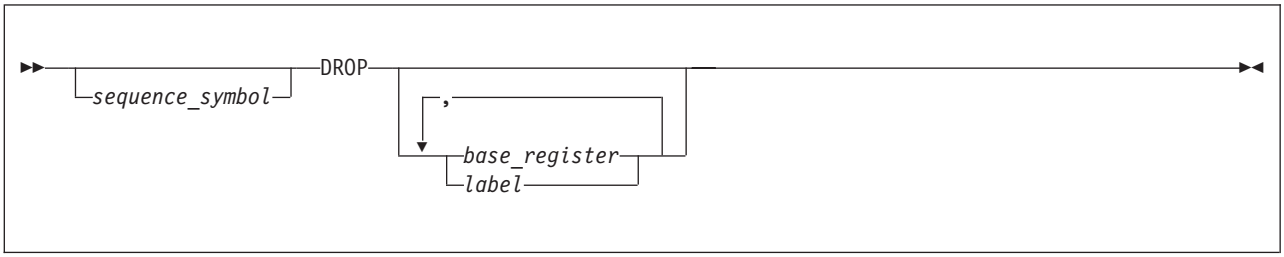
However, literal constants differ from DC operands in the following ways:

- Literals must be preceded by an equal sign.
- Multiple operands are not allowed.
- The duplication factor must not be zero.
- Symbols used in the duplication factor or length modifier must be previously defined. Scale and Exponent modifiers do not need pre-definition.
- If an address-type literal constant specifies a duplication factor greater than one and a nominal value containing the location counter reference, the value of the location counter reference is not incremented, but remains the same for each duplication.
- The assembler groups literals together by size. If you use a literal constant, the alignment of the constant can be different from that for an explicit constant. See “Literal pool” on page 38.

DROP instruction

The DROP instruction ends the domain of a USING instruction. This:

- Frees base registers previously assigned by the USING instruction for other programming purposes
- Ensures that the assembler uses the base register you want in a particular coding situation, for example, when two USING ranges overlap or coincide
- If a control section has not been established, DROP initiates an unnamed (private) control section



sequence_symbol
Is a sequence symbol.

base_register
Is an absolute expression whose value represents one of the general registers 0 through 15. The expression in *base_register* indicates a general register, previously specified in the operand of an ordinary USING statement, that is no longer to be used for base addressing.

label
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

The ordinary symbol denoted by *label* must be a symbol previously used in the name field of a labeled USING statement or a labeled dependent USING statement.

If *base_register* or *label* are not specified in the operand of a DROP instruction, all active base registers assigned by ordinary, labeled, and labeled dependent USING instructions are dropped.

After a DROP instruction:

- The assembler does not use the register or registers specified in the DROP instruction as base registers. A register made unavailable as a base register by a DROP instruction can be reassigned as a base register by a subsequent USING instruction.
- The label or labels specified in the DROP instruction are no longer available as symbol qualifiers. A label made unavailable as a symbol qualifier by a DROP instruction can be reassigned as a symbol qualifier by a subsequent labeled USING instruction.

The following statements, for example, stop the assembler using registers 7 and 11 as base registers, and the label `FIRST` as a symbol qualifier:

```
DROP            7,11
DROP            FIRST
```

Labeled USING

You cannot end the domain of a labeled USING instruction by coding a DROP instruction that specifies the same registers as were specified in the labeled USING instruction. If you want to end the domain of a labeled USING instruction, you must code a DROP instruction with an operand that specifies the label of the labeled USING instruction.

Dependent USING

To end the domain of a dependent USING instruction, you must end the domain of the corresponding ordinary USING instruction. In the following example, the DROP instruction prevents the assembler from using register 12 as a base register. The DROP instruction causes the assembler to end the domain of the ordinary USING instruction and the domains of the two dependent USING instructions. The storage areas represented by INREC and OUTREC are both within the range of the ordinary USING instruction (register 12).

```
         USING           *,12
         USING           RECMAP,INREC
         USING           RECMAP,OUTREC
         .
         .
         DROP            12
         .
         .
INREC    DS              CL156
OUTREC   DS              CL156
```

To end the domain of a labeled dependent USING instruction, you can code a DROP instruction with the USING label in the operand. The following example shows this:

```
         USING           *,12
PRIOR    USING           RECMAP,INREC
POST     USING           RECMAP,OUTREC
         .
         .
         DROP            PRIOR,POST
         .
         .
INREC    DS              CL156
OUTREC   DS              CL156
```

In the above example, the DROP instruction makes the labels PRIOR and POST unavailable as symbol qualifiers.

When a labeled dependent USING domain is dropped, none of any subordinate USING domains are dropped. In the following example the labeled dependent USING BLBL1 is not dropped, even though it is dependent on the USING ALBL2 that is dropped:

```
         USING           DSECTA,14
ALBL1    USING           DSECTA,14
         USING           DSECTB,ALBL1.A
         .
         .
ALBL2    USING           DSECTA,ALBL1.A
         .
BLBL1    USING           DSECTA,ALBL2.A+4
         .
         DROP            ALBL2
         .
DSECTA   DSECT
A        DS              A
DSECTB   DSECT
B        DS              A
```

A DROP instruction is not needed:

- If the base address is being changed by a new ordinary USING instruction, and the same base register is assigned. However, the new base address must be loaded into the base register by an appropriate sequence of instructions.
- If the base address is being changed by a new labeled USING instruction or a new labeled dependent USING instruction, and the same USING label is assigned. The correct base address must be loaded into the base register specified in the USING instruction by an appropriate sequence of instructions.

- At the end of a source module

DS instruction

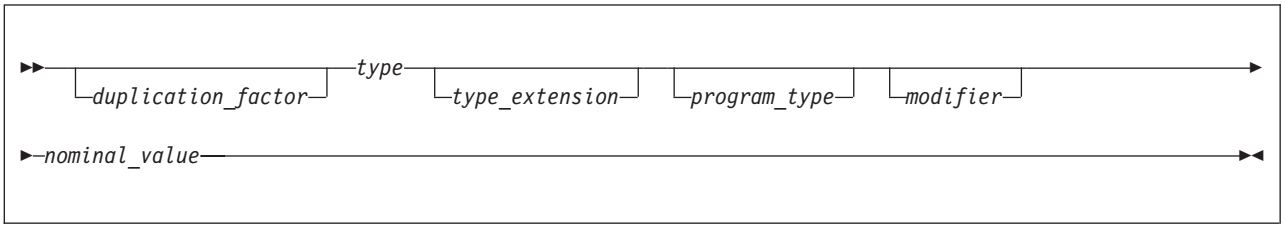
The DS instruction:
- Reserves areas of storage
- Provides labels for these areas
- Uses these areas by referring to the symbols defined as labels
- If a control section has not previously been established, DS initiates an unnamed (private) control section

symbol
Is one of the following:
- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If *symbol* denotes an ordinary symbol, the ordinary symbol represents the address of the first byte of the storage area reserved. If several operands are specified, the first storage area defined is addressable by the ordinary symbol. The other storage areas can be reached by relative addressing.

operand
Is an operand of six subfields. The first five subfields describe the attributes of the symbol. The sixth subfield provides the nominal values that determine the implicit lengths; however no constants are generated.

A DS operand has this format:



The format of the DS operand is identical to that of the DC operand; exactly the same subfields are used and are written in exactly the same sequence as they are in the DC operand. For more information about the subfields of the DC instruction, see "DC instruction" on page 109.

Unlike the DC instruction, the DS instruction causes no data to be assembled. Therefore, you do not have to specify the nominal value (sixth subfield) of a DS instruction operand. The DS instruction is the best way of symbolically defining storage for work areas, input and output buffers, and so on.

Although the formats are identical, there are two differences in the specification of subfields. They are:

- The nominal value subfield is optional in a DS operand, but it is mandatory in a DC operand. If a nominal value is specified in a DS operand, it must be valid.
- The maximum length that can be specified for the character (C) and hexadecimal (X) type areas is 65,535 bytes rather than 256 bytes for the same DC operands. The maximum length for the graphic (G) type is 65,534 bytes.

If *symbol* denotes an ordinary symbol, the ordinary symbol, as with the DC instruction:

- Has an address value of the first byte of the area reserved, after any boundary alignment is done
- Has a length attribute value, depending on the implicit or explicit length of the type of area reserved

If the DS instruction is specified with more than one operand or more than one nominal value in the operand, the label addresses the area reserved for the field that corresponds to the first nominal value of the first operand. The length attribute value is equal to the length explicitly specified or implicit in the first operand.

Bytes skipped for alignment

Unlike the DC instruction, bytes skipped for alignment are not set to zero. Also, nothing is assembled into the storage area reserved by a DS instruction. No assumption should be made as to the contents of the skipped bytes or the reserved area.

The size of a storage area that can be reserved by a DS instruction is limited only by the size of virtual storage or by the maximum value of the location counter, whichever is smaller.

How to use the DS instruction

Use the DS instruction to:

- Reserve storage
- Force alignment of the location counter so that the data that follows is on a particular storage boundary
- Name fields in a storage area.

To reserve storage

If you want to take advantage of automatic boundary alignment (if the ALIGN option is specified) and implicit length calculation, do not supply a length modifier in your operand specifications. Instead, specify a type subfield that corresponds to the type of area you need for your instructions.

Using a length modifier can give you the advantage of explicitly specifying the length attribute value assigned to the label naming the area reserved. However, your areas are not aligned automatically according to their type. If you omit the nominal value in the operand, use a length modifier for the binary (B), character (C), graphic (G), hexadecimal (X), and decimal (P and Z) type areas. If you do not, their labels are given a length attribute value of 1 (2 for G and CU type).

When you need to reserve large areas, you can use a duplication factor. However, in this case, you can only refer to the first area by its label. You can also use the character (C) and hexadecimal (X) field types to specify large areas using the length modifier. Duplication has no effect on implicit length.

Although the nominal value is optional for a DS instruction, you can put it to good use by letting the assembler compute the length for areas of the B, C, G, X, and decimal (P or Z) type areas. You achieve this by specifying the general format of the nominal value that is placed in the area at execution time.

If a nominal value and no length modifier are specified for a Unicode character string, the length of the storage reserved is derived by multiplying by two the number of characters specified in the nominal value (after pairing).

To force alignment

Use the DS instruction to align the instruction or data that follows, on a specific boundary. You can align the location counter to a doubleword, a fullword, or a halfword boundary by using the correct constant type (for example, D, F, or H) and a duplication factor of zero. No space is reserved for such an instruction, yet the data that follows is aligned on the correct boundary. For example, the following statements set the location counter to the next doubleword boundary and reserve storage space for a 128 byte field (whose first byte is on a doubleword boundary).

```
        DS              0D
AREA    DS              CL128
```

Alignment is forced whether or not the ALIGN assembler option is set.

To name fields within an area

Using a duplication factor of zero in a DS instruction also provides a label for an area of storage without reserving the area. Use DS or DC instructions to reserve storage for, and assign labels to, fields within the area. These fields can then be addressed symbolically. (Another way of accomplishing this is described in "DSECT instruction" on page 157.) The whole area is addressable by its label. In addition, the symbolic label has the length attribute value of the whole area. Within the area, each field is addressable by its label.

For example, assume that 80-character records are to be read into an area for processing and that each record has the following format:

Positions 5-10
: Payroll Number

Positions 11-30
: Employee Name

Positions 31-36
: Date

Positions 47-54
: Gross Wages

Positions 55-62
: Withholding Tax

The following example shows how DS instructions might be used to assign a name to the record area, then define the fields of the area and allocate storage for them. The first statement names the whole area by defining the symbol `RDAREA`; this statement gives `RDAREA` a length attribute of 80 bytes, but does not reserve any storage. Similarly, the fifth statement names a 6 byte area by defining the symbol `DATE`; the three subsequent statements define the fields of `DATE` and allocate storage for them. The second, ninth, and last statements are used for spacing purposes and, therefore, are not named.

```
RDAREA  DS      0CL80
        DS      CL4
PAYNO   DS      CL6
NAME    DS      CL20
DATE    DS      0CL6
DAY     DS      CL2
MONTH   DS      CL2
YEAR    DS      CL2
        DS      CL10
GROSS   DS      CL8
FEDTAX  DS      CL8
        DS      CL18
```

Here are some more examples of DS statements:

```
ONE     DS      CL80        One 80 byte field, length attribute of 80
TWO     DS      80C         Eighty 1 byte fields, length attribute of 1
THREE   DS      6F          6 fullwords, length attribute of 4
FOUR    DS      D           1 doubleword, length attribute of 8
```

```
FIVE     DS         4H           4 halfwords, length attribute of 2
SIX      DS         GL80         One 80 byte field, length attribute of 80
SEVEN    DS         80G          Eighty 2 byte fields, length attribute of 2
```

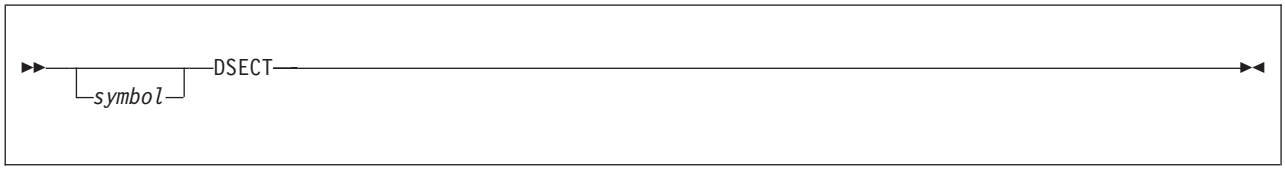
To define four 10 byte fields and one 100 byte field, the respective DS statements might be as follows:

```
FIELD    DS             4CL10
AREA     DS             CL100
```

Although FIELD might have been specified as one 40 byte field, the preceding definition has the advantage of providing FIELD with a length attribute of 10. This is pertinent when using FIELD as an SS machine instruction operand.

DSECT instruction

The DSECT instruction identifies the beginning or continuation of a dummy control section. One or more dummy sections can be defined in a source module.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

The DSECT instruction can be used anywhere in a source module after the ICTL instruction.

If *symbol* denotes an ordinary symbol, the ordinary symbol identifies the dummy section. If several DSECT instructions within a source module have the same symbol in the name field, the first occurrence initiates the dummy section and the rest indicate the continuation of the dummy section. The ordinary symbol denoted by *symbol* represents the address of the first byte in the dummy section, and has a length attribute value of 1.

If *symbol* is not specified, or if *name* is a sequence symbol, the DSECT instruction initiates or indicates the continuation of the unnamed control section.

The location counter for a dummy section is always set to an initial value of 0. However, when an interrupted dummy control section is continued using the DSECT instruction, the location counter last specified in that control section is continued.

The source statements that follow a DSECT instruction belong to the dummy section identified by that DSECT instruction.

Notes:

1. The assembler language statements that appear in a dummy section are not assembled into object code.
2. When establishing the addressability of a dummy section, the symbol in the name field of the DSECT instruction, or any symbol defined in the dummy section can be specified in a USING instruction.

3. A symbol defined in a dummy section can be specified in an address constant only if the symbol is paired with another symbol from the same dummy section, and if the symbols have opposite signs.

To effect references to the storage area defined by a dummy section, do the following:

- Provide one of:
  - An ordinary or labeled USING statement that specifies:
    - A general register that the assembler can use as a base register for the dummy section.
    - A value from the dummy section that the assembler can assume is contained by the register.
  - A dependent or labeled dependent USING statement that specifies:
    - A supporting base address (for which there is a corresponding ordinary USING statement) that lets the assembler determine a base register and displacement for the dummy section.
    - A value from the dummy section that the assembler can assume is the same as the supporting base address.
- Ensure that the base register is loaded with one of:
  - The actual address of the storage area if an ordinary USING statement or a labeled USING statement was specified.
  - The base address specified in the corresponding ordinary USING statement if a dependent or labeled dependent USING statement was specified.

The values assigned to symbols defined in a dummy section are relative to the initial statement of the section. Thus, all machine instructions that refer to names defined in the dummy section refer, at execution time, to storage locations relative to the address loaded into the register.

Figure 25 on page 159 shows an example of how to code the DSECT instruction. The sample code is referred to as "Assembly-2".

Assume that two independent assemblies (Assembly-1 and Assembly-2) have been loaded and are to be run as a single overall program. Assembly-1 is a routine that

1. Places a record in an area of storage
2. Places the address of the storage area in general register 3
3. Branches to Assembly-2 to process the record

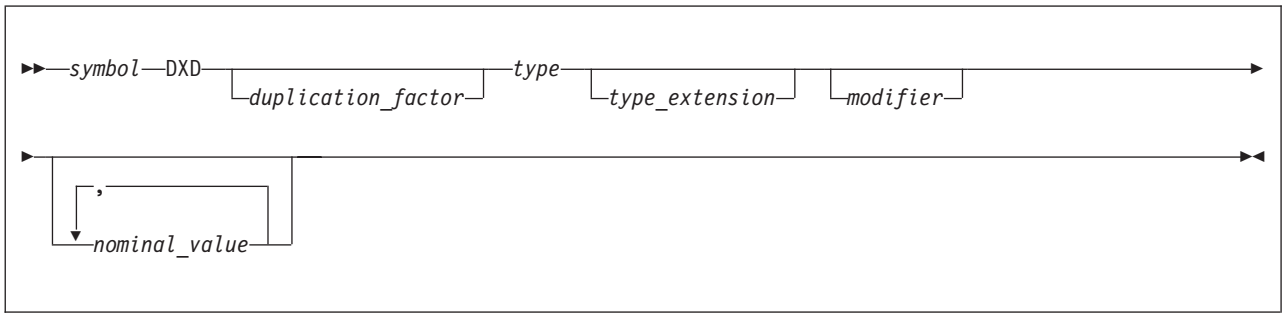
The storage area from Assembly-1 is identified in Assembly-2 by the dummy control section (DSECT) named INAREA. Parts of the storage area that you want to work with are named `INCODE`, `OUTPUTA`, and `OUTPUTB`. The statement `USING INAREA,3` assigns general register 3 as the base register for the `INAREA` DSECT. General register 3 contains the address of the storage area. The symbols in the DSECT are defined relative to the beginning of the DSECT. This means that the address values they represent are, at the time of program execution, the actual storage locations of the storage area that general register 3 addresses.

```
ASEMBLY2 CSECT
         USING           *,15
         USING           INAREA,3
         CLI             INCODE,C'A'
         BE              ATYPE
         MVC             OUTPUTA,DATA_B
         MVC             OUTPUTB,DATA_A
         B               FINISH
ATYPE    DS              0H
         MVC             OUTPUTA,DATA_A
         MVC             OUTPUTB,DATA_B
FINISH   BR              14
DATA_A   DC              CL8'ADATA'
DATA_B   DC              CL8'BDATA'
INAREA   DSECT
INCODE   DS              CL1
OUTPUTA  DS              CL8
OUTPUTB  DS              CL8
         END
```

Figure 25. Sample code using the DSECT instruction (Assembly-2)

DXD instruction

The DXD instruction identifies and defines an external dummy section.



symbol
Is an external symbol which is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

duplication_factor
Is the duplication factor subfield equivalent to the duplication factor subfield of the DS instruction.

type
Is the type subfield equivalent to the type subfield of the DS instruction.

type_extension
Is the type extension subfield equivalent to the type extension subfield of the DS instruction.

modifiers
Is the modifiers subfield equivalent to the modifiers subfield of the DS instruction.

nominal_value
Is the nominal-value subfield equivalent to the nominal-value subfield of the DS instruction. The nominal value is optional. If specified, it is not generated.

The DXD instruction can be used anywhere in a source module, after the ICTL instruction.

In order to reference the storage defined by the external dummy section, the ordinary symbol denoted by *symbol* must appear in the operand of a Q-type constant. This symbol represents the address of the first byte of the external dummy section defined, and has a length attribute value of 1.

The subfields in the operand field (duplication factor, type, type extension, modifier, and nominal value) are specified in the same way as in a DS instruction. The assembler computes the amount of storage and the alignment required for an external dummy section from the area specified in the operand field. For more information about how to specify the subfields, see "DS instruction" on page 154.

For example:

```
A    DXD     CL20          20 bytes, byte alignment
B    DXD     3F,XL4        20 bytes, fullword alignment
C    DXD     LQ            16 bytes, quadword alignment
```

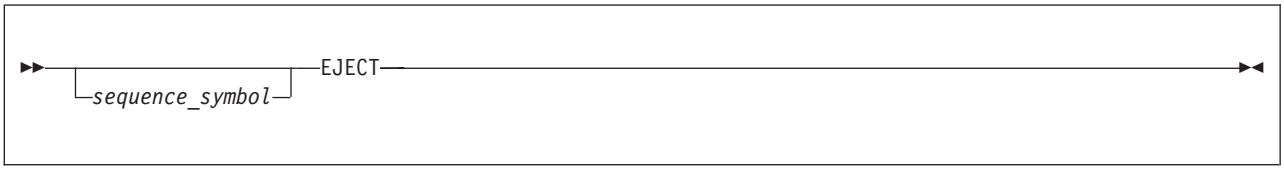
The linker uses the information provided by the assembler to compute the total length of storage required for all external dummy sections specified in a program.

Notes:

1. The DSECT instruction also defines an external dummy section, but only if the symbol in the name field appears in a Q-type offset constant in the same source module. Otherwise, a DSECT instruction defines a dummy section.
2. If two or more external dummy sections for different source modules have the same name, the linker uses the most restrictive alignment, and the largest section to compute the total length.

EJECT instruction

The EJECT instruction stops the printing of the assembler listing on the current page, and continues the printing on the next page.



sequence_symbol
Is a sequence symbol.

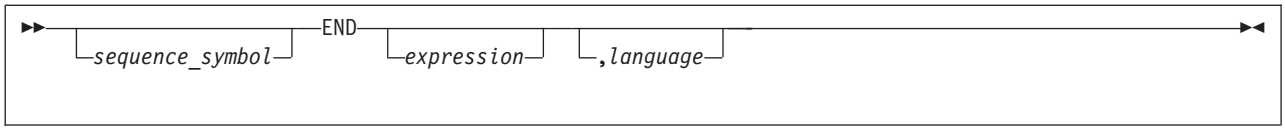
The EJECT instruction causes the next line of the assembler listing to be printed at the top of a new page. If the line before the EJECT statement appears at the bottom of a page, the EJECT statement has no effect.

An EJECT instruction immediately following another EJECT instruction is ignored. A TITLE instruction immediately following an EJECT instruction causes the title to change but no additional page eject is performed. (The TITLE instruction normally forces a page eject.)

The EJECT instruction statement itself is not printed in the listing.

END instruction

Use the END instruction to end the assembly of a program. You can also supply an address in the operand field to which control can be transferred after the program is loaded. The END instruction must always be the last statement in the source program.



sequence_symbol
Is a sequence symbol.

expression
Specifies the point to which control can be transferred when loading of the object program completes. If the GOFF option is in effect this parameter is ignored. This point is normally the address of the first executable instruction in the program, as shown in the following sequence:

```
NAME     CSECT
AREA     DS             50F
BEGIN    BALR           2,0
         USING          *,2
         .
         .
         .
         END            BEGIN
```

If specified, *expression* can be generated by substitution into variable symbols. It must not be a literal. It must also satisfy one of these conditions:

- It is a simply relocatable expression representing an address in the source module delimited by the END instruction.
- If it contains an external symbol, the external symbol must be the only term in the expression, or the remaining terms in the expression must reduce to zero.

language
A marker for use by language translators that produce assembly code. The operand has three suboperands. The values in the operand are copied into the END record in the object deck if the NOGOFF option is specified, or in a B_IDRL record if the GOFF option is specified.

The syntax of the operand is

```
(char10,char4,char5)
```

where all three suboperands and the commas and parentheses are required.

`char10` is a one to ten character code. It is intended to be a language translator identifier. `char4` must be exactly four characters long. It is intended to be a version and release code. `char5` must be exactly five characters long, and should be a date in the format "YYDDD". It is intended to be the compile date. For example:

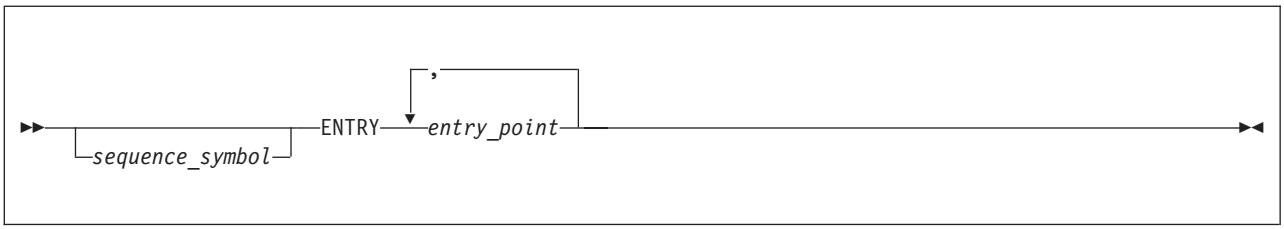
```
END    ENTRYPT,(MYCOMPILER,0101,00273)
```

Notes:

1. If the END instruction is omitted, one is generated by the assembler, and message `ASMA140W END record missing` is issued.
2. Refer to the text in "Generating END statements" on page 300 about lookahead processing, and the effect it has on generated END statements.
3. If the END statement is not the last statement in the input stream, and the BATCH option has been specified, the assembler initiates assembly of a new source module when the current assembly is completed. (For more information about the BATCH option, see the section "BATCH" in the *HLASM Programmer's Guide*)

ENTRY instruction

The ENTRY instruction identifies symbols defined in this source module as “external” so that they can be referred to by another source module. These symbols are entry symbols.



sequence_symbol
Is a sequence symbol.

entry_point
Is a relocatable symbol that:

- Is a valid symbol
- Is defined in an executable control section
- Is not defined in a dummy control section, a common control section, or an external control section

Up to 65535 individual control sections, external symbols, and external dummy sections can be defined in a source module. However, the practical maximum number depends on the amount of table storage available to the program that links the object module.

The assembler lists each entry symbol of a source module in an external symbol dictionary, along with entries for external symbols, common control sections, parts, and external control sections.

A symbol used as the name entry of a START or CSECT instruction is also automatically considered an entry symbol, and does not have to be identified by an ENTRY instruction.

A symbol identified by an ENTRY instruction should not also be declared by an EXTRN instruction, but it can be referenced in the nominal value of a V-type address constant in the same source module.

The length attribute value of entry symbols is the same as the length attribute value of the symbol at its point of definition.

EQU instruction

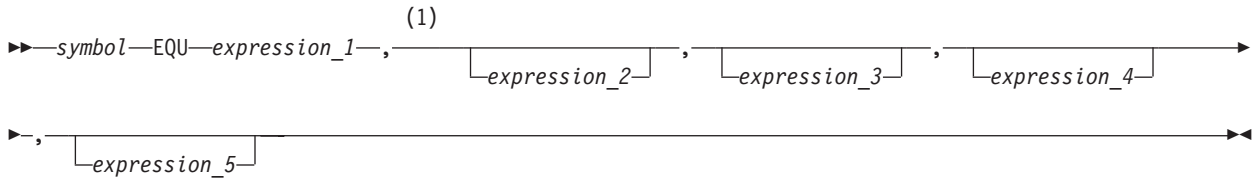
The EQU instruction assigns absolute or relocatable values to symbols. Use it to:

- Assign single absolute values to symbols.
- Assign the values of previously defined symbols or expressions to new symbols, thus letting you use different mnemonics for different purposes.
- Compute expressions whose values are unknown at coding time or difficult to calculate. The value of the expressions is then assigned to a symbol.
- Assign length and type attributes to symbols, either implicitly or explicitly.
- Assign program type and assembler type values to symbols.

EQU also assigns attributes. It takes the value, relocation, and length attributes of the operand and assigns them to the name field symbol, and sets the integer and scale attributes to zero. The type attributes of an absolute expression is always 'U', and its length attribute is always 1 (unless the second and third operands are specified.

When there is a symbol naming a complex relocatable expression, or a complex relocatable expression is eventually "reduced" to an absolute or simply relocatable expression, the first symbol is used for attribute assignment.

The program type is always null, and the assembler type is always null, except when the appropriate operand is specified.



Notes:

1 Use commas as placeholders when there is an expression following

symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

expression_1
Represents a value and attributes that the assembler assigns to the symbol in the name field. *expression_1* can have any value allowed for an assembly expression: absolute (including negative), relocatable, or complexly relocatable. The assembler carries this value as a signed 4 byte (32 bit) number; all 4 bytes are printed in the program listings opposite the symbol. Implicitly, the relocation and length attributes are also assigned for certain types of expressions.

Any symbols used in *expression_1* need not be previously defined. However, if any symbol is not previously defined, the value of *expression_1* is not assigned to the symbol in the name field until assembly time and therefore cannot be used during conditional assembly (see "Using conditional assembly values" on page 165).

If *expression_1* is a complexly relocatable expression, the whole expression, rather than its value, is assigned to the symbol. During the evaluation of any expression that includes a complexly relocatable symbol, that symbol is replaced by its own defining expression. Consider the following example, in which `A1` and `A2` are defined in one control section, and `B1` and `B2` in another:

```
X        EQU             A1+B1
Y        EQU             X-A2-B2
```

The first EQU statement assigns a complexly relocatable expression (`A1+B1`) to `X`. During the evaluation of the expression in the second EQU statement, `X` is replaced by its defining relocatable expression (`A1+B1`). The assembler evaluates the resulting expression (`A1+B1-A2-B2`) and assigns an absolute value to `Y`, because the relocatable terms in the expression are paired. The expression must not contain literals.

expression_2
Represents a value that the assembler assigns as a *length attribute value* to the symbol in the name field. It is optional, but, if specified, must be an absolute value in the range 0 to 65,535. This value overrides the normal length attribute value implicitly assigned from *expression_1*. For example:

```
A    DS    CL121          Define a print line buffer
ACC  Equ   A,1            Define first character, length 1
```

The symbol ACC has the same location value as A, but length attribute 1.

All symbols appearing in *expression_2* must have been previously defined, and all expressions in *expression_2* must be evaluatable when the EQU statement is processed. For example, the second operand in the statements defining the symbol X cannot be evaluated when the last statement has been processed, because the value of the symbol X is unknown until the symbol A has been defined.

```
Z    DS   XL(L'A)          Z    DS XL(A)
Y    DS   XL7              Y    DS XL7
X    EQU  Z,*-Z            X    EQU Z,*-Z
A    DS   XL5              A    EQU 5
```

If *expression_2* is omitted, the assembler assigns a length attribute value to the symbol in the name field according to the length attribute value of the leftmost (or only) term of *expression_1*, as follows:

1. If the leftmost term of *expression_1* is a location counter reference (*), a self-defining term, or a symbol length attribute value reference, the length attribute is 1. This also applies if the leftmost term is a symbol that is equated to any of these values.
2. If the leftmost term of *expression_1* is a symbol that is used in the name field of a DC or DS instruction, the length attribute value is equal to the implicit or explicit length of the first (or only) constant specified in the DC or DS operand field.
3. If the leftmost term is a symbol that is used in the name field of a machine instruction, the length attribute value is equal to the length of the assembled instruction.
4. Symbols that name assembler instructions, except the DC, DS, CCW, CCW0, and CCW1 instructions, have a length attribute value of 1. Symbols that name a CCW, CCW0, or CCW1 instruction have a length attribute value of 8.
5. The length attribute value described in cases 2, 3, and 4 above is the assembly-time value of the attribute.

For more information about the length attribute value, see "Symbol length attribute reference" on page 33.

For example:

```
X   DS    CL80           X has length attribute 80
Y   EQU   X,40           Y has length attribute 40
```

expression_3

Represents a value that the assembler assigns as a *type attribute value* to the symbol in the name field. It is optional, but, if specified, it must be an absolute value in the range 0 to 255.

All symbols appearing in *expression_3* must have been previously defined, and all expressions in *expression_3* must be evaluatable when the EQU statement is processed.

If *expression_3* is omitted, the assembler assigns a type attribute value of U to the symbol, which means the symbol in the name field has an undefined (or unknown or unassigned) type attribute. See the general discussion about data attributes in "Data attributes" on page 284, and "Type attribute (T')" on page 289.

For example:

```
A    DS    D            A has type attribute D
B    EQU   A,,C'X'      B has type attribute X
```

expression_4

Represents a value (any absolute expression) that the assembler assigns as a *program type value* to the symbol in the name field. It is optional. It can be specified as a decimal, character, hex, or binary self-defining term and is stored as a 4 byte (32 bit) number; all 4 bytes are printed in the program listings opposite the symbol. The value is not used in any way by the assembler, and can be queried by using the SYSATTRP built-in function.

All symbols appearing in *expression_4* must have been previously defined, and all expressions in *expression_4* must be evaluatable when the EQU statement is processed.

If *expression_4* is omitted, the assembler assigns a null to the program type, and querying the value using the SYSATTRP built-in function returns a null value.

expression_5
Represents 2 to 4 characters that the assembler assigns as an *assembler type value* to the symbol in the name field. It is optional. It is stored as a 4 byte string; all 4 bytes are printed in the program listings opposite the symbol. The value is used by the assembler when type-checking has been activated, and can be queried by using the SYSATTRA built-in function.

Valid values for this operand are:

```
AR       Register - Access
CR       Register - Control
CR32     Register - Control 32 bit
CR64     Register - Control 64 bit
FPR      Register - Floating-Point
GR       Register - General
GR32     Register - General 32 bit
GR64     Register - General 64 bit
```

If *expression_5* is omitted, the assembler assigns a null value to the assembler type, and querying the value using the SYSATTRA built-in function returns a null value.

The EQU instruction can be used anywhere in a source module after the ICTL instruction. Note, however, that the EQU instruction initiates an unnamed control section (private code) if it is specified before the first control section.

Using conditional assembly values

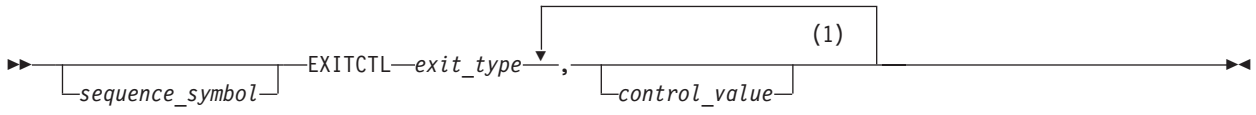
The following rules describe when you can use the value, length attribute value, or type attribute value of an equated symbol in conditional assembly statements:

- If you want to use the value of the symbol in conditional assembly statements, then:
  - The EQU statement that defines the symbol must be processed by the assembler before the conditional assembly statement that refers to the symbol.
  - The symbol in the name field of the EQU statement must be an ordinary symbol.
  - *Expression_1* must be an absolute expression, and must contain only self-defining terms or previously defined symbols.
- If only *expression_1* is specified:
  - The assembler assigns a type attribute value of U.
  - If the EQU statement that defines the symbol is processed by the assembler before the conditional assembly statement that refers to the symbol, the assembler assigns the length attribute of expression_1. Otherwise, the assembler assigns a length attribute value 1.

  You can use these values in conditional assembly statements, although references to the length attribute might be flagged.
- If you specify *expression_2* or *expression_3* and you want to use the explicit attribute value during conditional assembly processing, then:
  - The symbol in the name field must be an ordinary symbol.
  - The expression must contain only self-defining terms.

EXITCTL instruction

The EXITCTL instruction sets or modifies the contents of the four signed fullword exit-control parameters that the assembler maintains for each type of exit.



Notes:

1 From one to four values to be supplied.

sequence_symbol
Is a sequence symbol.

exit_type
Identifies the type of exit to which this EXITCTL instruction applies. *Exit_type* must have one of the following values:

SOURCE
Sets the exit-control parameters for the user-supplied exit module specified in the INEXIT suboption of the EXIT assembler option.

LIBRARY
Sets the exit-control parameters for the user-supplied exit module specified in the LIBEXIT suboption of the EXIT assembler option.

LISTING
Sets the exit-control parameters for the user-supplied exit module specified in the PRTEXIT suboption of the EXIT assembler option.

PUNCH
Sets the exit-control parameters for the user-supplied exit module specified in the OBJEXIT suboption of the EXIT assembler option when it is called to process the object module records generated when the DECK assembler option is specified.

OBJECT (z/OS and CMS)
Sets the exit-control parameters for the user-supplied exit module specified in the OBJEXIT suboption of the EXIT assembler option when it is called to process the object module records generated when the OBJECT or GOFF assembler option is specified.

ADATA
Sets the exit-control parameters for the user-supplied exit module specified in the ADEXIT suboption of the EXIT assembler option.

TERM Sets the exit-control parameters for the user-supplied exit module specified in the TRMEXIT suboption of the EXIT assembler option.

control_value
Is the value to which the corresponding exit-control parameter should be set. For each exit type, the assembler maintains four exit-control parameters known as EXITCTL_1, EXITCTL_2, EXITCTL_3, and EXITCTL_4. Therefore, up to four values can be specified. Which exit-control parameter is set is determined by the position of the value in the operand of the instruction. You must code a comma in the operand for each omitted value. If specified, *control_value* must be either:

- A decimal self-defining term with a value in the range -2^{31} to $+2^{31}-1$.
- An expression in the form $*\pm n$, where * is the current value of the corresponding exit-control parameter to which *n*, a decimal self-defining term, is added or from which *n* is subtracted. The value of the result of adding *n* to or subtracting *n* from the current exit-control parameter value must be in the range -2^{31} to $+2^{31}-1$.

If *control_value* is omitted, the corresponding exit-control parameter retains its current value.

The following example shows how to set the exit-control parameters EXITCTL_1 and EXITCTL_3 for the LISTING exit without affecting the contents of the other exit-control parameters:

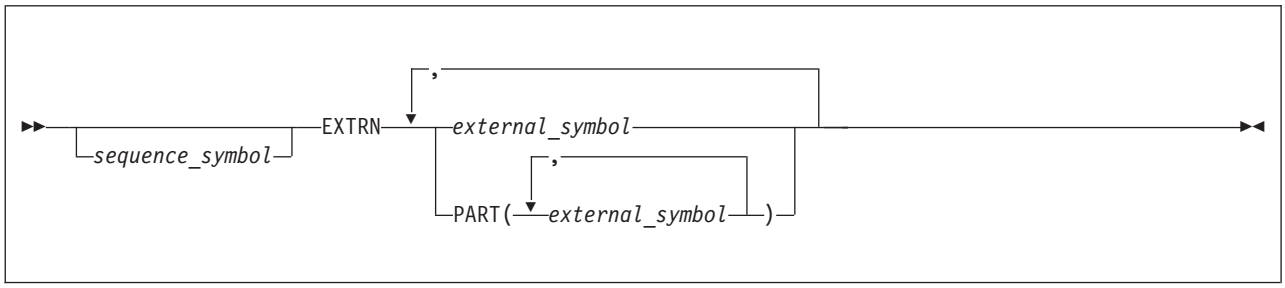
```
EXITCTL LISTING,256,,*+128
```

See the section "EXITCTL*n*" in the *HLASM Programmer's Guide* for information about how EXITCTL values are passed to each type of exit.

The assembler initializes all exit-control parameters to binary zeros.

EXTRN instruction

The EXTRN instruction identifies "external" symbols referred to in this source module but defined in another source module. These symbols are external symbols.



sequence_symbol
Is a sequence symbol.

external_symbol
Is a relocatable symbol that:

- Is a valid symbol
- Is not used as the name entry of a source statement in the source module in which it is defined

PART(*external_symbol***)**
external_symbol is a relocatable symbol as described above, that also:

- Is a reference to a part as defined on the CATTR instruction.

Up to 65535 individual control sections, external symbols, and external dummy sections can be defined in a source module. However, the practical maximum number depends on the amount of table storage available during link-editing.

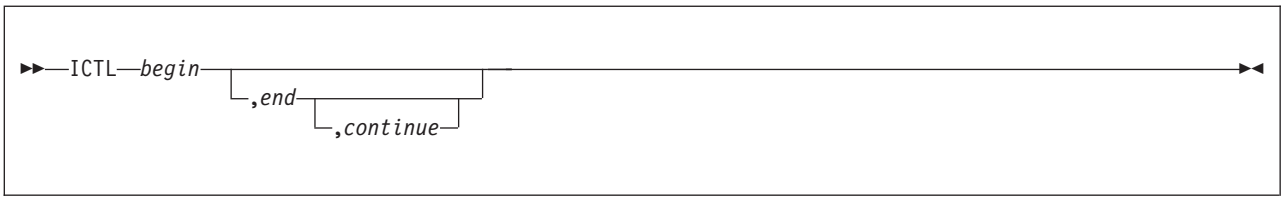
The assembler lists each external symbol identified in a source module in the external symbol dictionary, along with entries for entry symbols, common control sections, parts, and external control sections.

A symbol identified by an EXTRN instruction should not also be declared by an ENTRY instruction.

External symbols have a length attribute of 1. See also "WXTRN instruction" on page 202.

ICTL instruction

The ICTL instruction changes the begin, end, and continue columns that establish the coding format of the assembler language source statements.



begin
Specifies the begin column of the source statement. It must be a decimal self-defining term with value 1 - 40.

end
Specifies the end column of the source statement. When *end* is specified it must be a decimal self-defining term with 41 - 80. It must be not less than *begin* +5, and must be greater than *continue*. If *end* is not specified, it is assumed to be 71.

continue
Specifies the continue column of the source statement. When specified, *continue* must be a decimal self-defining term within the range of 2 to 40, and it must be greater than *begin*. If *continue* is not specified, or if column 80 is specified as the end column, the assembler assumes that continuation lines are not allowed.

Default
1,71,16

Use the ICTL instruction only once, at the beginning of a source program. If no ICTL statement is used in the source program, the assembler assumes that 1, 71, and 16 are the begin, end, and continue columns.

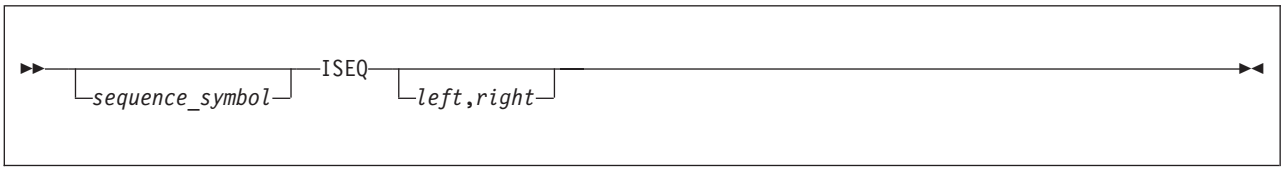
With the ICTL instruction, you can, for example, increase the number of columns to be used for the identification or sequence checking of your source statements. By changing the begin column, you can even create a field before the begin column to contain identification or sequence numbers. For example, the following instruction designates the begin column as 9 and the end column as 80. Since the end column is specified as 80, no continuation records are recognized.

```
        ICTL          9,80
```

COPY Instruction: The ICTL instruction does not affect the format of statements brought in by a COPY instruction or generated from a library macro definition. The assembler processes these statements according to the standard begin, end, and continue columns described in "Field boundaries" on page 12.

ISEQ instruction

The ISEQ instruction forces the assembler to check if the statements in a source module are in sequential order. In the ISEQ instruction, you specify the columns between which the assembler is to check for sequence numbers.



sequence_symbol
Is a sequence symbol.

left
Specifies the first column of the field to be sequence-checked. If specified, *left* must be a decimal self-defining term with value 1 - 80.

right
Specifies the rightmost column of the field to be sequence checked. If specified, *right* must be a decimal self-defining term with value 1 - 80, and must be greater than or equal to *left*.

If *left* and *right* are omitted, sequence checking is ended. Sequence checking can be restarted with another ISEQ statement. An ISEQ statement that is used to end sequence checking is itself sequence-checked.

The assembler begins sequence checking with the first statement line following the ISEQ instruction. The assembler also checks continuation lines.

Sequence numbers on adjacent statements or lines are compared according to the 8 bit internal EBCDIC collating sequence. When the sequence number on one line is not greater than the sequence number on the preceding line, a sequence error is flagged, and a warning message is issued, but the assembly is not ended.

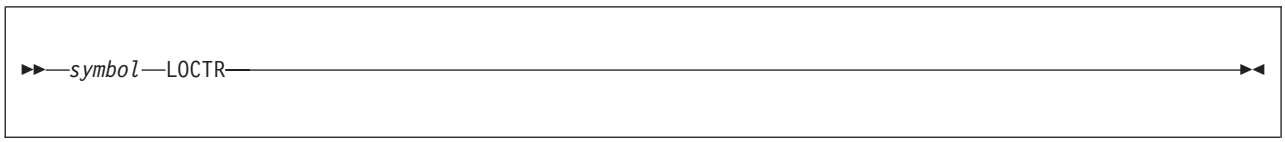
If the sequence field in the preceding line is spaces, the assembler uses the last preceding line with a non-space sequence field to make its comparison.

The assembler checks only those statements that are specified in the coding of a source module. This includes any COPY instruction statement or macro instruction. The assembler does not check:

- Statements inserted by a COPY instruction
- Statements generated from model statements inside macro definitions or from model statements in open code (statement generation is discussed in detail in Chapter 7, "How to specify macro definitions," on page 213)
- Statements in library macro definitions

LOCTR instruction

The LOCTR instruction specifies multiple location counters within a control section. The assembler assigns consecutive addresses to the segments of code using one location counter before it assigns addresses to segments of coding using the next location counter.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

By using the LOCTR instruction, you can code your control section in a logical order. For example, you can code work areas and data constants within the section of code, using them without having to branch around them:

```
A      CSECT ,          See note 1
       LR    12,15
       USING A,12
       .
B      LOCTR ,          See note 2
       .
C      LOCTR ,
       .
B      LOCTR ,          See note 3
       .
A      LOCTR ,          See note 4
       .
DUM    DSECT ,          See note 1
C      LOCTR ,          See note 5
       .
       END
```

LOCTRs are ordered by their definition order. So in the previous example, the ordering is A, B, and C. When there are statements in LOCTR groups, the code is generated using currently active USINGs and then moved to the final location.

Notes:

1. The first location counter of a section, class, or part is defined by the name of the START, CSECT, DSECT, RSECT, CATTR, or COM instruction defining the section.
2. The LOCTR instruction defines a location counter.
3. The LOCTR continues a previously defined location counter. A location counter remains in use until it is interrupted by a LOCTR, CSECT, DSECT, RSECT, or COM instruction.
4. A LOCTR instruction with the same name as a control section continues the first location counter of that section. However, an unnamed LOCTR cannot be used to continue an unnamed (private code) control section.
5. A LOCTR instruction with the same name as a LOCTR instruction in a previous control section causes that control section to be continued using the location counter specified, even though the LOCTR instruction might follow the definition (or resumption) of a different section.
6. To continue a location counter in an unnamed section, a named location counter must first be specified for the section by a LOCTR in the unnamed section.

A control section cannot have the same name as a previous LOCTR instruction. A LOCTR instruction placed before the first control section definition initiates an unnamed control section before the LOCTR instruction is processed.

The length attribute of a LOCTR name is 1.

LOCTR instructions do not force alignment; code assembled under a location counter other than the first location counter of a control section is assembled starting at the next available byte after the previous segment.

A LOCTR name can be referenced as an ordinary symbol. If the LOCTR name does not match a section name, its value is the location counter value assigned to its first appearance, and it might have arbitrary alignment and other attributes. If the LOCTR name is also a control section name, the value assigned is that of the origin of the control section. So a LOCTR with the same name as the CSECT resumes the first location counter within the CSECT. A CSECT instruction resumes the last location counter used.

Table 35. LOCTR behavior with NOGOFF option

LOCTR name	Effect
Section	Resumes assembling with the first location counter of that section

Table 35. LOCTR behavior with NOGOFF option (continued)

LOCTR name	Effect
Other	• If the LOCTR name was previously declared, resumes assembling with the location counter of that LOCTR group • If the LOCTR name was not previously declared, begins processing a new LOCTR group of statements to be assembled following the most recently processed section or LOCTR group

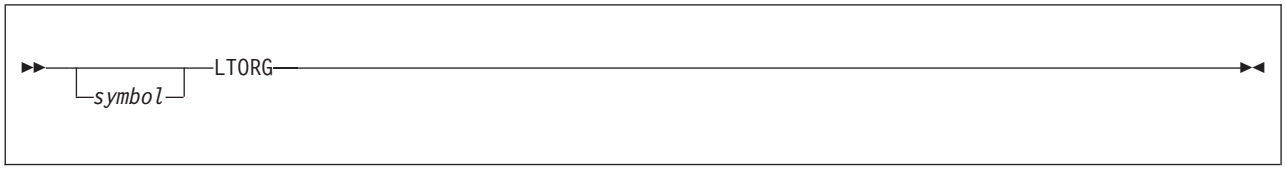
Table 36. LOCTR behavior with GOFF option

LOCTR name	Effect
Section	Resumes assembling with the first location counter of the element in the B_TEXT class of that section
Class	Not allowed
Part	Resumes assembling with the first location counter of the part
Other	• If the LOCTR name was previously declared, resumes assembling with the location counter of that LOCTR group • If the LOCTR name was not previously declared, begins processing statements in a new LOCTR group to be assembled following the most recently processed class, part, or LOCTR group.

LTORG instruction

Use the LTORG instruction so that the assembler can collect and assemble literals into a literal pool. A literal pool contains the literals you specify in a source module either after the preceding LTORG instruction, or after the beginning of the source module.

If a control section has not been established, LTORG initiates an unnamed (private) control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If *symbol* is an ordinary symbol or a variable symbol that has been assigned an ordinary symbol, the ordinary symbol is assigned the value of the address of the first byte of the literal pool. This symbol is aligned on a boundary specified by the SECTALGN option, and has a length attribute of 1. If bytes are skipped after the end of a literal pool to achieve alignment for the next instruction, constant, or area, the bytes are not filled with zeros. If the literal pool does not reside in a DSECT, and includes any items that require quadword alignment, and the SECTALGN value defaults to 8, the assemble of the literal causes the issue of an ASMA500E message.

The assembler ignores the borders between control sections when it collects literals into pools. Therefore, you must be careful to include the literal pools in the control sections to which they belong (for details, see "Addressing considerations" on page 172).

The creation of a literal pool gives the following advantages:

- Automatic organization of the literal data into sections that are correctly aligned and arranged so that minimal space is wasted in the literal pool.
- Assembling of duplicate data into the same area.
- Because all literals are cross-referenced, you can find the literal constant in the pool into which it has been assembled.

Literal pool

A literal pool is created under the following conditions:

- Immediately after an LTORG instruction.
- If no LTORG instruction is specified, and no LOCTRs are used in the first control section, a literal pool generated after the END statement is created at the end of the first control section, and appears in the listing after the END statement.
- If no LTORG instruction is specified, and LOCTRs are used in the first control section, a literal pool generated after the END statement is created at the end of the most recent LOCTR segment of the first section, and appears in the listing after the END statement.
- To force the literal pool to the end of the control section when using LOCTRs, you must resume the last LOCTR of the CSECT before the LTORG statement (or before the END statement if no LTORG statement is specified).

Each literal pool has five segments into which the literals are stored (a) in the order that the literals are specified, and (b) according to their assembled lengths, which, for each literal, is the total explicit or implied length:

- The *first segment* contains all literal constants whose assembled lengths are a multiple of 16.
- The *second segment* contains those whose assembled lengths are a multiple of 8, but not of 16.
- The *third segment* contains those whose assembled lengths are a multiple of 4, but not a multiple of 8.
- The *fourth segment* contains those whose assembled lengths are even, but not a multiple of 4.
- The *fifth segment* contains all the remaining literal constants whose assembled lengths are odd.

Since each literal pool is aligned on a SECTALGN alignment, this guarantees that all literals in the second segment are doubleword aligned; in the third segment, fullword aligned; and, in the fourth, halfword aligned. The minimum value of SECALGN is doubleword, so quadword alignment is not guaranteed. No space is wasted except, possibly, at the origin of the pool, and in aligning to the start of the statement following the literal pool.

Literals from the following statements are in the pool, in the segments indicated by the parenthesized numbers:

```
FIRST    START           0
         .
         MVC             TO,=3F'9'       (3)
         AD              2,=D'7'         (2)
         IC              2,=XL1'8'       (5)
         MVC             MTH,=CL3'JAN'   (5)
         LM              4,5,=2F'1,2'    (2)
         AH              5,=H'33'        (4)
         L               2,=A(ADDR)      (3)
         MVC             FIVES,=XL16'05' (1)
```

Addressing considerations

If you specify literals in source modules with multiple control sections, then:

- Write an LTORG instruction at the end of each control section, so that all the literals specified in the section are assembled into the one literal pool for that section. If a control section is divided and interspersed among other control sections, write an LTORG instruction at the end of each segment of the interspersed control section.

- When establishing the addressability of each control section, make sure that:
  - All the literal pool for that section is also addressable, by including it within a USING range.
  - The literal specifications are within the corresponding USING domain.

  The USING range and domain are described in "USING instruction" on page 193.

All the literals specified after the last LTORG instruction, or, if no LTORG instruction is specified, all the literals in a source module are assembled into a literal pool at the end of the first control section. You must then make this literal pool addressable, along with the addresses in the first control section. This literal pool is printed in the program listing after the END instruction.

Duplicate literals

If you specify duplicate literals within the part of the source module that is controlled by an LTORG instruction, only one literal constant is assembled into the pertinent literal pool. This also applies to literals assembled into the literal pool at the end of the first or only control section of a source module that contains no LTORG instructions.

Literals are duplicates only if their specifications are identical, not if the object code assembled happens to be identical.

When two literals specifying identical A-type, Y-type or S-type address constants contain a reference to the value of the location counter (*), both literals are assembled into the literal pool. This is because the value of the location counter might be different in the two literals. Even if the location counter value is the same for both, they are still both assembled into the literal pool.

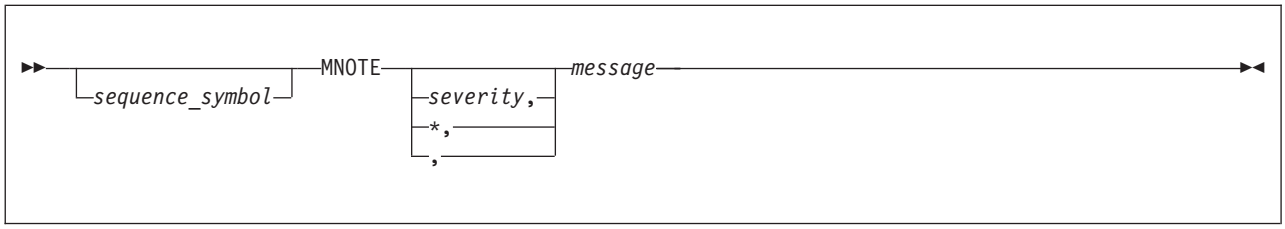
The following examples show how the assembler stores pairs of literals, if the placement of each pair is controlled by the same LTORG statement.

```
=X'F0'          Both are
=C'0'           stored

=XL3'0'         Both are
=HL3'0'         stored

=A(*+4)         Both are
=A(*+4)         stored

=X'FFFF'        Identical,
=X'FFFF'        only one copy is stored
```

MNOTE instruction

The MNOTE instruction generates your own error messages or displays intermediate values of variable symbols computed during conditional assembly.

The MNOTE instruction can be used inside macro definitions or in open code, and its operation code can be created by substitution. The MNOTE instruction causes the generation of a message that is given a statement number in the printed listing.



sequence_symbol
Is a sequence symbol.

severity
Is a severity code. The *severity* operand can be any decimal self-defining term, or a SETA variable symbol. The term must have a value in the range 0 through 255. The severity code is used to determine the return code issued by the assembler when it returns control to the operating system. The severity can also change the value of the system variable symbols &SYSM_HSEV and &SYSM_SEV (see "&SYSM_HSEV System Variable Symbol" on page 244 and "&SYSM_SEV System Variable Symbol" on page 244).

message
Is the message text. It can be any combination of characters enclosed in apostrophes. The rules that apply to this character string are as follows:

- Variable symbols are allowed. The apostrophes that enclose the message can be generated from variable symbols.
- Two ampersands or two apostrophes are needed to generate an ampersand or an apostrophe. If variable symbols have ampersands or apostrophes as values, the values must be coded as two ampersands or two apostrophes.
- If the number of characters in the character string plus the rest of the MNOTE operand exceeds 1024 bytes the assembler issues diagnostic message

  ```
  ASMA062E Illegal operand format
  ```

  Note: The maximum length of the second operand is three less than the maximum supported length of SETC character string.
- Double-byte data is permissible in the operand field when the DBCS assembler option is specified. The double-byte data must be valid.
- The DBCS ampersand and apostrophe are not recognized as delimiters.
- A double-byte character that contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.

Remarks

Any remarks for the MNOTE instruction statement must be separated by one or more spaces from the apostrophe that ends the message.

If *severity* is provided, or *severity* is omitted but the comma separating it from *message* is present, the message is treated as an error message; otherwise the message is treated as comments. The rules for specifying the contents of *severity* are:

- The severity code can be specified as any arithmetic expression allowed in the operand field of a SETA instruction. The expression must have a value in the range 0 through 255.

  Example:

  ```
  MNOTE 2,'ERROR IN SYNTAX'
  ```

  The generated result is:

  ```
  2,ERROR IN SYNTAX
  ```
- If the severity code is omitted, but the comma separating it from the message is present, the assembler assigns a default value of 1 as the severity code.

  Example:

  ```
  MNOTE ,'ERROR, SEV 1'
  ```

  The generated result is:

  ```
  ,ERROR, SEV 1
  ```
- An asterisk in the severity code subfield causes the message and the asterisk to be generated as a comment statement.

Example:

```
MNOTE *,'NO ERROR'
```

The generated result is:

```
*,NO ERROR
```

Here is an example taken from a CICS® macro:

```
MNOTE 8,'FIELD IS DEFINED OUTSIDE OF THE SIZE OPERAND'
MNOTE *,'PARAMETERS SPECIFIED IN THE DFHMDI MACRO,'
MNOTE *,'MACRO REQUEST IS IGNORED.'
```

A further advantage of this approach is that only one severity 8 error is seen instead of three.

- If the severity code subfield is omitted, including the comma separating it from the message, the assembler generates the message as a comment statement.

  Example:

  ```
  MNOTE 'NO ERROR'
  ```

  The generated result is:

  ```
  NO ERROR
  ```

Notes:

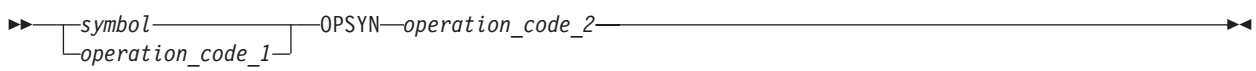
1. An MNOTE instruction causes a message to be printed, if the current PRINT option is ON, even if the PRINT NOGEN option is specified.
2. The statement number of the message generated from an MNOTE instruction with a severity code is listed among any other error messages for the current source module. However, the message is printed only if the severity code specified is greater than or equal to the severity code *nnn* specified in the FLAG(*nnn*) assembler option.
3. The statement number of the comments generated from an MNOTE instruction without a severity code is not listed among other error messages.

OPSYN instruction

The OPSYN instruction defines or deletes symbolic operation codes.

The OPSYN instruction has two formats. The first format defines a new operation code to represent an existing operation code, or to redefine an existing operation code for:
- Machine and extended mnemonic branch instructions
- Assembler instructions, including conditional assembly instructions
- Macro instructions

Define operation code



If *operation_code_2* has been previously defined as both a machine instruction and as a macro, both are copied to the definition of *operation_code_1*.

The second format deletes an existing operation code for:
- Machine and extended mnemonic branch instructions
- Assembler instructions, including conditional assembly instructions
- Macro instructions

Delete operation code

symbol
Is one of the following:

- An ordinary symbol that is not the same as an existing operation code
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol and is not the same as an existing operation code

operation_code_1
Is one of the following:

- An operation code described in this chapter, or any machine instruction (such as those described in Chapter 4, "Machine instruction statements," on page 65), or Chapter 9, "How to write conditional assembly instructions," on page 279
- The operation code defined by a previous OPSYN instruction
- The name of a previously defined macro.

operation_code_2
Is one of these:

- An operation code described in this chapter, or any machine instruction (such as those described in Chapter 4, "Machine instruction statements," on page 65), or Chapter 9, "How to write conditional assembly instructions," on page 279
- The operation code defined by a previous OPSYN instruction
- The name of a previously defined macro.

In the first format, the OPSYN instruction assigns the properties of the operation code denoted by *operation_code_2* to the ordinary symbol denoted by *symbol* or the operation code denoted by *operation_code_1*.

In the second format, the OPSYN instruction causes the operation code specified in *operation_code_1* to lose its properties as an operation code.

The OPSYN instruction can be coded anywhere in the program to redefine an operation code, following an ICTL instruction, if any.

The symbol in the name field can represent a valid operation code. It loses its current properties as if it had been defined in an OPSYN instruction with a space-filled operand field. In the following example, `L` and `LR` both possess the properties of the LR machine instruction operation code:

```
L        OPSYN           LR
```

When the same symbol appears in the name field of two OPSYN instructions, the latest definition takes precedence. In this example, `STORE` now represents the STH machine operation:

```
STORE    OPSYN           ST
STORE    OPSYN           STH
```

Note: OPSYN is not processed during lookahead mode (see "Lookahead" on page 299). Therefore it cannot be used during lookahead to replace an opcode that must be processed during lookahead, such as COPY. For example, assuming AFTER is defined in COPYBOOK, the following code gives an ASMA042E error (Length attribute of symbol is unavailable):

```
         AIF  (L'AFTER LT 2).BEYOND
 OPCOPY  OPSYN COPY              OPSYN not processed during look ahead
         OPCOPY COPYBOOK         OPCOPY fails
 .BEYOND ANOP ,
```

Redefining conditional assembly instructions

A redefinition of a conditional assembly instruction only comes into effect in macro definitions occurring after the OPSYN instruction. The original definition is always used when a macro instruction calls a macro that was defined and edited before the OPSYN instruction.

An OPSYN instruction that redefines the operation code of an assembler or machine instruction generated from a macro instruction is, however, effective immediately, even if the definition of the macro was made prior to the OPSYN instruction. Consider the following example:

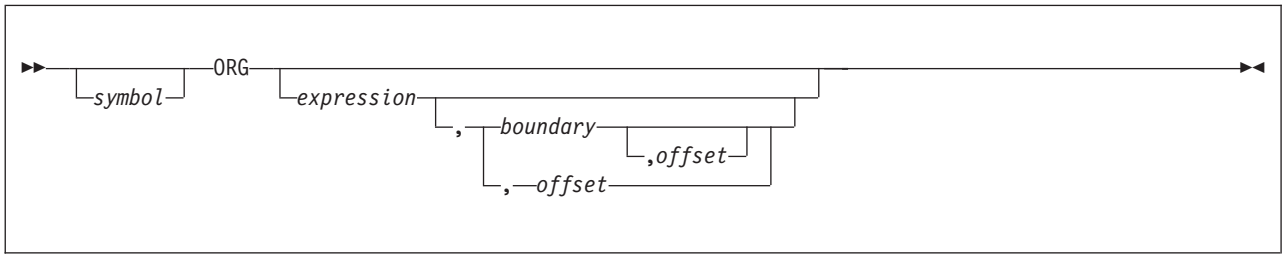
```
         MACRO                    Macro header
         MAC           ...        Macro prototype
         AIF           ...
         MVC           ...
         .
         MEND                     Macro trailer
         .
 AIF     OPSYN         AGO        Assign AGO properties to AIF
 MVC     OPSYN         MVI        Assign MVI properties to MVC
         .
         MAC           ...        Macro call
                                  (AIF interpreted as AIF instruct-
                                  ion; generated AIFs not printed)
+        MVC           ...        Interpreted as MVI instruction
         .
         .                        Open code started at this point
         AIF           ...        Interpreted as AGO instruction
         MVC           ...        Interpreted as MVI instruction
```

In this example, AIF and MVC instructions are used in a macro definition. AIF is a conditional assembly instruction, and MVC is a machine instruction. OPSYN instructions are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC. In subsequent calls of the macro MAC, AIF is still defined, and used, as an AIF operation, but the generated MVC is treated as an MVI operation. In open code following the macro call, the operations of both instructions are derived from their new definitions assigned by the OPSYN instructions. If the macro is redefined (by another macro definition), the new definitions of AIF and MVC (that is, AGO and MVI) are used for further generations.

ORG instruction

The ORG instruction alters the setting of the location counter and thus controls the structure of the current control section. This redefines portions of a control section.

If a control section has not been previously established, ORG initiates an unnamed (private) control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

If *symbol* denotes an ordinary symbol, the ordinary symbol is defined with the value that the location counter had before the ORG statement is processed.

expression
Is a relocatable expression, the value of which is used to set the location counter. If *expression* is omitted, the location counter is set to the next available location for the current control section.

boundary
Is an absolute expression that must be a number that is a power of 2 with a range from 2 (halfword) to 4096 (page). If *boundary* exceeds the SECTALGN value, message ASMA500E is issued. This message is not issued if the section being processed is a Reference Control Section (DSECT, DXD, or COM).

boundary must be a predefined absolute expression whose value is known at the time the ORG statement is processed.

If the *boundary* operand is greater than 16, the GOFF option must be specified in addition to the SECTALGN option.

offset
Any absolute expression

If *boundary* or *offset* are provided, then the resultant location counter is calculated by rounding the expression up to the next higher **boundary** and then adding the **offset** value.

ORG emits no "fill" bytes for bytes skipped in any direction.

In general, symbols used in *expression* need not have been previously defined. However, the relocatable component of *expression* (that is, the unpaired relocatable term) must have been previously defined in the same control section in which the ORG statement appears, or be equated to a previously defined value.

A length attribute reference to the name of an ORG instruction is always invalid. Message ASMS042E is issued, and a default value of 1 is assigned.

An ORG statement cannot be used to specify a location below the beginning of the control section in which it appears. For example, the following statement is not correct if it appears less than 500 bytes from the beginning of the current control section.

```
ORG             *-500
```

This is because the expression specified is negative, and sets the location counter to a value larger than the assembler can process. The location counter *wraps around* (the location counter is discussed in detail in "Location counter" on page 32).

If you specify multiple location counters with the LOCTR instruction, the ORG instruction can alter only the location counter in use when the instruction appears. Thus, you cannot control the structure of the whole control section using ORG, but only the part that is controlled by the current location counter.

An ORG statement cannot be used to change sections or LOCTR segments. For example:

```
AA              CSECT
X               DS          D
Y               DS          F
BB              CSECT
                ORG         Y
```

is invalid, because the section containing the ORG statement (BB) is not the same as the section in AA in which the ORG operand expression Y is defined.

With the ORG statement, you can give two instructions the same location counter values. In such a case, the second instruction does not always eliminate the effects of the first instruction. Consider the following example:

```
ADDR     DC              A(ADDR)
         ORG             *-4
B        DC              C'BETA'
```

In this example, the value of `B ('BETA')` is destroyed by the relocation of `ADDR` during linkage editing.

The following example shows some examples of ORG using the boundary and offset operands:

```
origin   csect
         ds    235x               Define 235 bytes
         org   origin,,3          Move location counter back to start + 3
         org   *,8                Align on 8 byte boundary
         org   *,8,-2             Align to 8 byte boundary -2 bytes
translate dc   cl256' '           Define aligned translate table
         org   translate+c'a'
         dc    c'ABCDEFGHI'
         org   translate+c'j'
         dc    c'JKLMNOPQR'
         org   translate+c's'
         dc    c'STUVWXYZ'
         org   translate+c'A'
         dc    c'ABCDEFGHI'
         org   translate+c'J'
         dc    c'JKLMNOPQR'
         org   translate+c'S'
         dc    c'STUVWXYZ'
         org   ,
         end
```

Using Figure 26 on page 180 as an example, to build a translate table (for example, to convert EBCDIC character code into some other internal code):

1. Define the table (see **1** in Figure 26 on page 180) as being filled with zeros.
2. Use the ORG instruction to alter the location counter so that its counter value indicates a specific location (see **2** in Figure 26 on page 180) within the table.
3. Redefine the data (see **3** in Figure 26 on page 180) to be assembled into that location.
4. After repeating the first three steps (see **4** in Figure 26 on page 180) until your translate table is complete, use an ORG instruction with a null operand field to alter the location counter. The counter value then indicates the next available location (see **5** in Figure 26 on page 180) in the current control section (after the end of the translate table).

Both the assembled object code for the whole table filled with zeros, and the object code for the portions of the table you redefined, are printed in the program listings. However, the data defined later is loaded over the previously defined zeros and becomes part of your object program, instead of the zeros.

That is, the ORG instruction can cause the location counter to be set to any part of a control section, even the middle of an instruction, into which you can assemble data. It can also cause the location counter to be set to the next available location so that your program can be assembled sequentially.

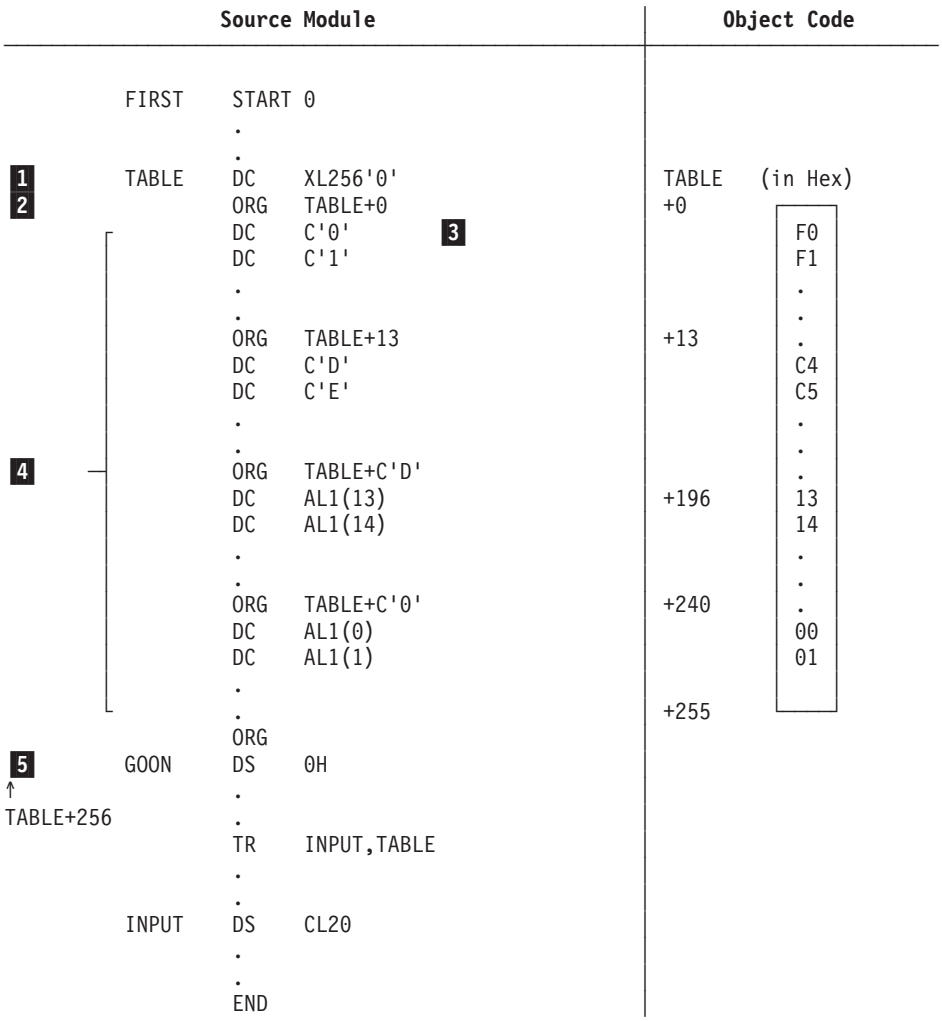
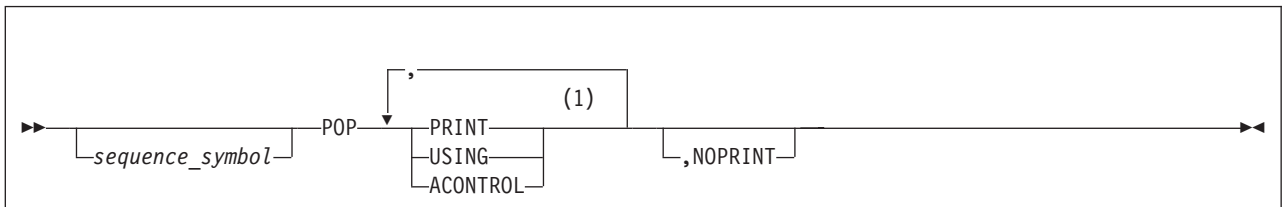


Figure 26. Building a translate table

POP instruction

The POP instruction restores the PRINT, USING, or ACONTROL status saved by the most recent PUSH instruction.



Notes:

1 Each keyword from this group can be selected only once.

sequence_symbol
Is a sequence symbol.

PRINT
Instructs the assembler to restore the PRINT status to the status saved by the most recent PUSH instruction.

USING
Instructs the assembler to restore the USING status to the status saved by the most recent PUSH instruction.

ACONTROL
Instructs the assembler to restore the ACONTROL status to the status saved by the most recent PUSH instruction.

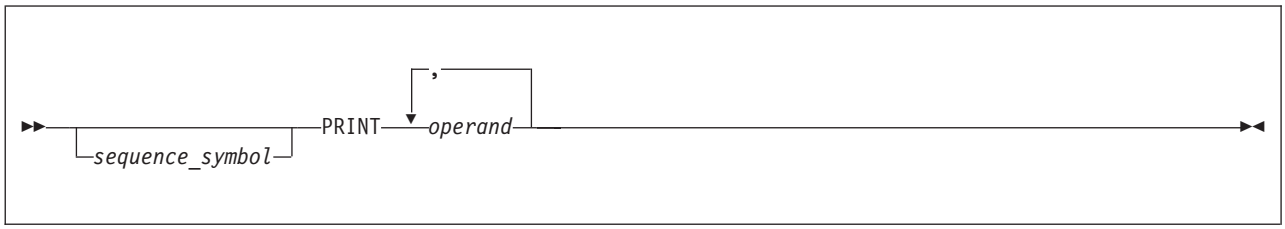
NOPRINT
Instructs the assembler to suppress the printing of the POP statement in which it is specified.

The POP instruction causes the status of the current PRINT, USING or ACONTROL instruction to be overridden by the PRINT, USING or ACONTROL status saved by the last PUSH instruction. For example:

```
          PRINT  GEN                  Printed macro generated code
          DCMAC  X,27                 Call macro to generate DC
+         DC     X'27'                ... Generated statement
          PUSH   PRINT                Save PRINT status
          PRINT  NOGEN                Suppress macro generated code
          DCMAC  X,33                 Call macro to generate DC
          POP    PRINT                Restore PRINT status
          DCMAC  X,42                 Call macro to generate DC
+         DC     X'42'                ... Generated statement
```

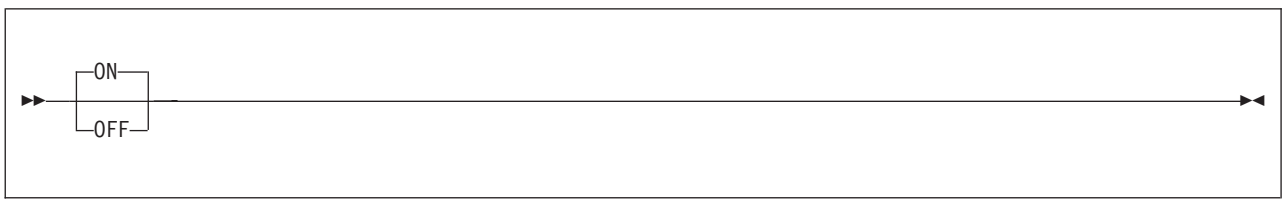
PRINT instruction

The PRINT instruction controls the amount of detail printed in the listing of programs.



sequence_symbol
Is a sequence symbol.

operand
Is an operand from one of the groups of operands described below. If a null operand is supplied, it is accepted by the assembler with no effect on the other operands specified. The operands are listed in hierarchic order. The effect, if any, of one operand on other operands is also described.



ON
Instructs the assembler to print, or resume printing, the *source and object* section of the assembler listing.

OFF
Instructs the assembler to stop printing the *source and object* section of the assembler listing. A subsequent PRINT ON instruction resumes printing.

When this operand is specified the printing actions requested by the GEN, DATA, MCALL, and MSOURCE operands do not apply.

GEN
Instructs the assembler to print all statements generated by the processing of a macro. This operand does not apply if PRINT OFF has been specified.

NOGEN
Instructs the assembler not to print statements generated by conditional assembly or the processing of a macro. This applies to all levels of macro nesting; no generated code is displayed while PRINT NOGEN is in effect. If this operand is specified, the DATA operand does not apply to constants that are generated during macro processing. Also, if this operand is specified, the MSOURCE operand does not apply. When the PRINT NOGEN instruction is in effect, the assembler prints one of the following on the same line as the macro call or model statement:

- The object code for the first instruction generated. The object code includes the data that is shown under the ADDR1 and ADDR2 columns of the assembler listing.
- The first eight bytes of generated data from a DC instruction

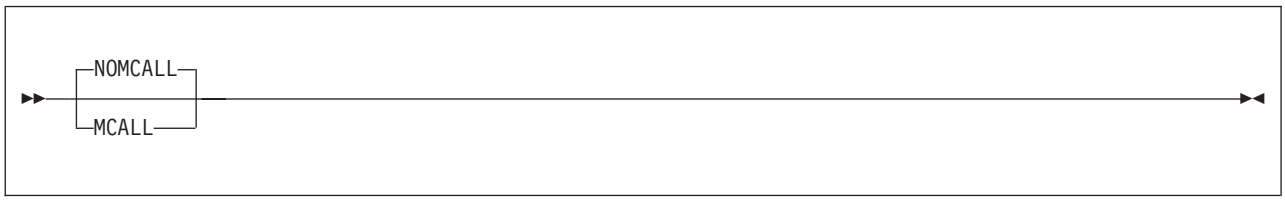
When the assembler forces alignment of an instruction or data constant, it generates zeros in the object code and prints the generated object code in the listing. When you use the PRINT NOGEN instruction the generated zeros are not printed.

Note: If the next line to print after macro call or model statement is a diagnostic message, the object code or generated data is not shown in the assembler listing.

The MNOTE instruction always causes a message to be printed.

NODATA
Instructs the assembler to print only the first eight bytes of the object code of constants. This operand does not apply if PRINT OFF has been specified. If PRINT NOGEN has been specified, this operand does not apply to constants generated during macro processing.

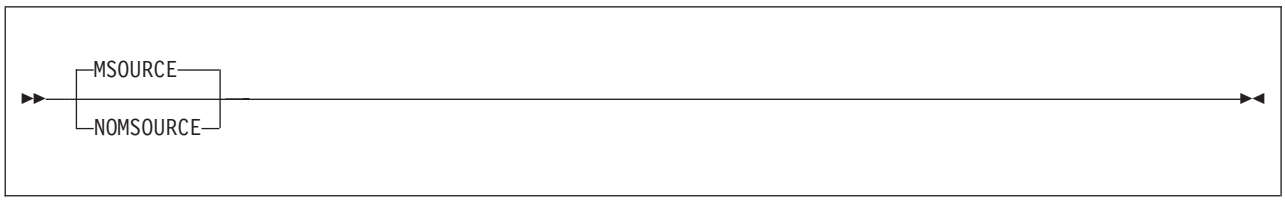
DATA
Instructs the assembler to print the object code of all constants in full. This operand does not apply if PRINT OFF has been specified. If PRINT NOGEN has been specified, this operand does not apply to constants generated during macro processing.



NOMCALL
Instructs the assembler to suppress the printing of nested macro call instructions.

MCALL
Instructs the assembler to print nested macro call instructions, including the name of the macro definition to be processed and the operands and values passed to the macro definition. The assembler only prints the operands and comments up to the size of its internal processing buffer. If this size is exceeded the macro call instruction is truncated, and the characters `... MORE` are added to the end of the printed macro call. This does not affect the processing of the macro call.

This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.

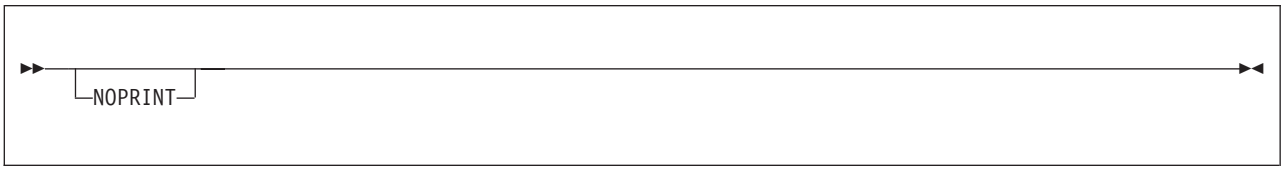


MSOURCE
Instructs the assembler to print the source statements generated during macro processing, as well as the assembled addresses and generated object code of the statements. This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.

NOMSOURCE
Instructs the assembler to suppress the printing of source statements generated during macro processing, without suppressing the printing of the assembled addresses and generated object code of the statements. This operand does not apply if either PRINT OFF or PRINT NOGEN has been specified.

UHEAD
Instructs the assembler to print a summary of active USINGs following the TITLE line on each page of the *source and object program* section of the assembler listing. This operand does not apply if PRINT OFF has been specified.

NOUHEAD
Instructs the assembler not to print a summary of active USINGs.



NOPRINT
Instructs the assembler to suppress the printing of the PRINT statement in which it is specified. The NOPRINT operand must only be specified with one or more other operands.

The PRINT instruction can be specified any number of times in a source module, but only those operands specified in the instruction change the current print status.

PRINT options can be generated by macro processing during conditional assembly. However, at assembly time, all options are in force until the assembler encounters a new and opposite option in a PRINT instruction.

The PUSH and POP instructions, described in "PUSH instruction" on page 185 and "POP instruction" on page 180, also influence the PRINT options by saving and restoring the PRINT status.

You can override the effect of the operands of the PRINT instruction by using the PCONTROL assembler option. For more information about this option, see the section "PCONTROL" in the *HLASM Programmer's Guide*.

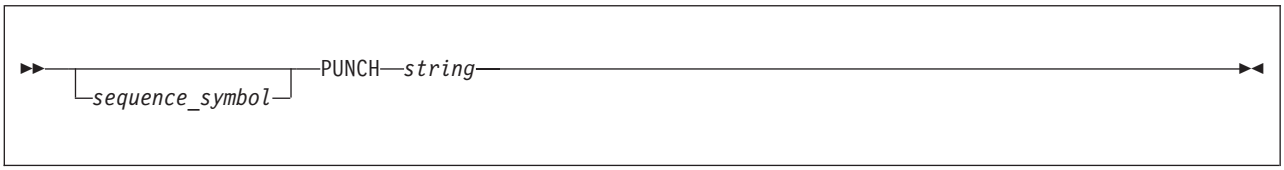
Unless the NOPRINT operand is specified, or the assembler listing is suppressed by the NOLIST assembler option, the PRINT instruction itself is printed.

Process statement

The process statement is described under "*PROCESS statement" on page 84.

PUNCH instruction

The PUNCH instruction creates a record containing a source or other statement, or an object record, to be written to the object file.



sequence_symbol
Is a sequence symbol.

string
Is a character string of up to 80 characters, enclosed in apostrophes. All 256 characters in the EBCDIC character set are allowed in the character string. Variable symbols are also allowed.

Double-byte data is permissible in the operand field when the DBCS assembler option is specified. However, the following rules apply to double-byte data:

- The DBCS ampersand and the apostrophe are not recognized as delimiters.

- A double-byte character that contains the value of an EBCDIC ampersand or an apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.

The position of each character specified in the PUNCH statement corresponds to a column in the record to be punched. However, the following rules apply to ampersands and apostrophes:

- A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.
- A pair of ampersands is punched as one ampersand.
- A pair of apostrophes is punched as one apostrophe.
- An unpaired apostrophe followed by one or more spaces ends the string of characters punched. If a non-space character follows an unpaired apostrophe, an error message is issued and nothing is punched.

Only the characters punched, including spaces, count toward the maximum of 80 allowed.

The PUNCH instruction causes the data in its operand to be punched (copied) into a record. One PUNCH instruction produces one record, but as many PUNCH instructions as necessary can be used.

You can code PUNCH statements in:

- A source module to produce control statements for the linker. The linker uses these control statements to process the object module.
- Macro definitions to produce, for example, source statements in other computer languages or for other processing phases.

The assembler writes the record produced by a PUNCH statement when it writes the object deck. The ordering of this record in the object deck is determined by the order in which the PUNCH statement is processed by the assembler. The record appears after any object deck records produced by previous statements, and before any other object deck records produced by subsequent statements.

The PUNCH instruction statement can appear anywhere in a source module. If a PUNCH instruction occurs before the first control section, the resultant record punched precedes all other records in the object deck.

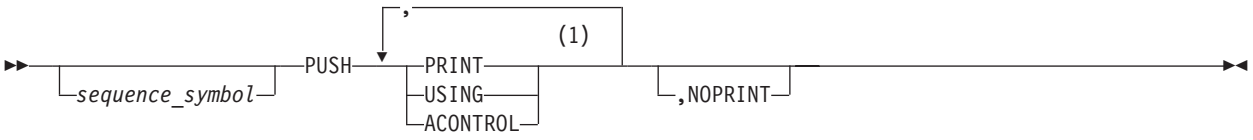
The record punched as a result of a PUNCH instruction is not a logical part of the object deck, even though it can be physically interspersed in the object deck.

Notes:

1. The identification and sequence number field generated as part of other object deck records is not generated for the record punched by the PUNCH instruction.
2. If the NODECK and NOOBJECT assembler options are specified, no records are punched for the PUNCH instruction.
3. Do not use the PUNCH instruction if the GOFF option is specified, as the resulting file might be unusable.

PUSH instruction

The PUSH instruction saves the current PRINT, USING, or ACONTROL status in push-down storage on a last-in, first-out basis. You restore this PRINT, USING, or ACONTROL status later, also on a last-in, first-out basis, by using a POP instruction.



Notes:

1 Each keyword from this group can be selected only once.

sequence_symbol
Is a sequence symbol.

PRINT
Instructs the assembler to save the PRINT status in a push-down stack.

USING
Instructs the assembler to save the USING status in a push-down stack.

ACONTROL
Instructs the assembler to save the ACONTROL status in a push-down stack.

NOPRINT
Instructs the assembler to suppress the printing of the PUSH statement in which it is specified.

The PUSH instruction only causes the status of the current PRINT, USING, or ACONTROL instructions to be saved. The PUSH instruction does not:

- Change the status of the current PRINT or ACONTROL instructions
- Imply a DROP instruction, or change the status of the current USING instructions

REPRO instruction

The REPRO instruction causes the data specified in the record that follows to be copied unchanged into the object file.

sequence_symbol
Is a sequence symbol.

The REPRO instruction can appear anywhere in a source module. One REPRO instruction produces one punched record, but as many REPRO instructions as necessary can be used. Records are created as the object file is being created, so records might be interspersed among object code. These records are part of the object file, but are not intended to contain normal object code or symbols.

The statement to be reproduced can contain any of the 256 characters in the EBCDIC character set, including spaces, ampersands, and apostrophes. Unlike the PUNCH instruction, the REPRO instruction does not allow values to be substituted into variable symbols before the record is punched.

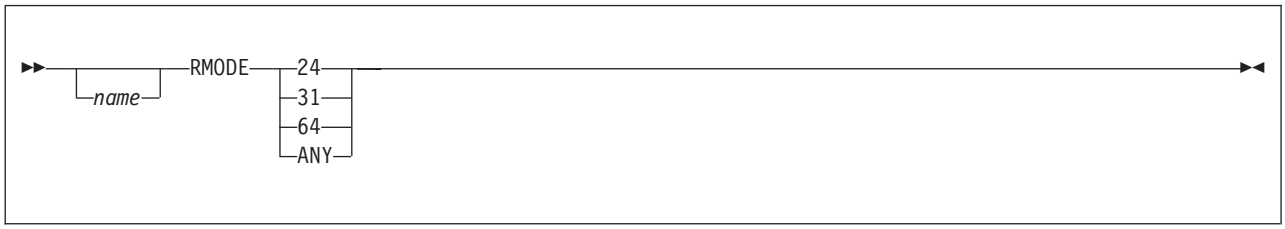
Notes:

1. The identification and sequence numbers generated as part of other object deck records are not generated for records punched by the REPRO instruction.

2. The sequence and continuation fields of the record to be REPROed are not checked, even if the ISEQ instruction was specified.
3. If the NODECK and NOOBJECT assembler options are specified, no records are punched for the REPRO instruction, or for the object deck of the assembly.
4. Since the text of the line following a REPRO statement is not validated or changed in any way, it can contain double-byte data, but this data is not validated.
5. Do not use the REPRO instruction if the GOFF option is specified, as the resulting file might be unusable.

RMODE instruction

The RMODE instruction specifies the residence mode to be associated with control sections in the object deck.



name
Is the name field that associates the residence mode with a control section. If there is a symbol in the name field, it must also appear in the name field of a START, CSECT, RSECT, or COM instruction in this assembly. If the name field is space-filled, there must be an unnamed control section in this assembly. If the name field contains a sequence symbol (see "Symbols" on page 25 for details), it is treated as a blank name field.

24
Specifies that a residence mode of 24 is to be associated with the control section; that is, the control section must be resident below 16 MB.

31 Specifies that a residence mode of either 24 or 31 is to be associated with the control section; that is, the control section can be resident above or below 16 MB.

64
Specifies that a residence mode of 64 is to be associated with the control section (see "64 bit addressing mode" on page 84).

ANY
Is understood to mean RMODE 31.

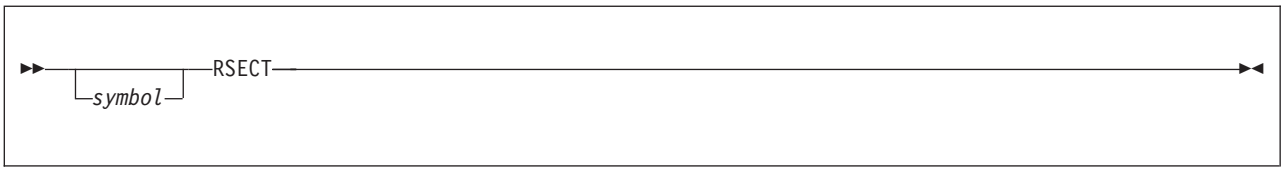
Any field of this instruction can be generated by a macro, or by substitution in open code.

Notes:

1. RMODE can be specified anywhere in the assembly. It does not initiate an unnamed control section.
2. An assembly can have multiple RMODE instructions; however, two RMODE instructions cannot have the same name field.
3. The valid and invalid combinations of AMODE and RMODE are shown in Table 11 on page 96. Combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64 bit addressing mode" on page 84.
4. AMODE or RMODE cannot be specified for an unnamed *common* control section.
5. The defaults used when zero or one MODE is specified are shown in Table 12 on page 96. Combinations involving AMODE 64 and RMODE 64 are subject to the support outlined in "64 bit addressing mode" on page 84.

RSECT instruction

The RSECT instruction initiates a read-only executable control section or indicates the continuation of a read-only executable control section.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

When an executable control section is initiated by the RSECT instruction, the assembler automatically checks the control section for possible coding violations of program reenterability, regardless of the setting of the RENT assembler option. As the assembler cannot check program logic, the checking is not exhaustive. Non-reentrant code is diagnosed by a warning message.

The RSECT instruction can be used anywhere in a source module after the ICTL instruction. If it is used to initiate the first executable control section, it must not be preceded by any instruction that affects the location counter and thus causes the first control section to be initiated.

If *symbol* denotes an ordinary symbol, the ordinary symbol identifies the control section. If several RSECT instructions within a source module have the same symbol in the name field, the first occurrence initiates the control section and the rest indicate the continuation of the control section. The ordinary symbol denoted by *symbol* represents the address of the first byte in the control section, and has a length attribute value of 1.

If *symbol* is not specified, or if *name* is a sequence symbol, the RSECT instruction initiates or indicates the continuation of the unnamed control section.

See "CSECT instruction" on page 106 for a discussion on the interaction between RSECT and the GOFF assembler option.

The beginning of a control section is aligned on a boundary determined by the SECTALGN option. However, when an interrupted control section is continued using the RSECT instruction, the location counter last specified in that control section is continued.

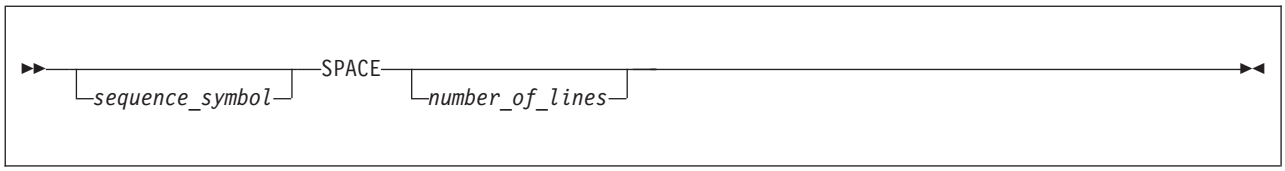
The source statements following an RSECT instruction that either initiate or indicate the continuation of a control section are assembled into the object code of the control section identified by that RSECT instruction.

Notes:

1. The assembler indicates that a control section is read-only by setting the read-only attribute in the object module.
2. The end of a control section or portion of a control section is marked by (a) any instruction that defines a new or continued control section, or (b) the END instruction.

SPACE instruction

The SPACE instruction inserts one or more blank lines in the listing of a source module, thus separating sections of code on the listing page.



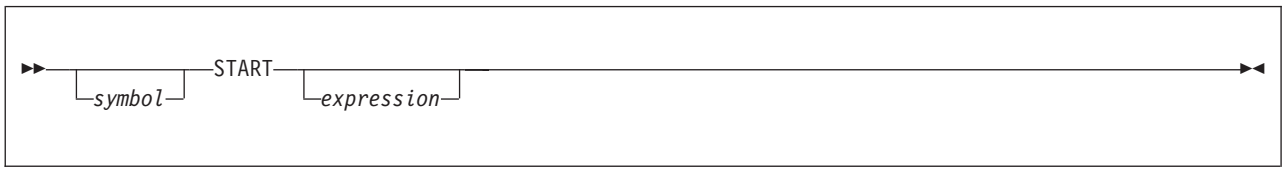
sequence_symbol
Is a sequence symbol.

number_of_lines
Is an absolute expression that specifies the number of lines to be left blank. You can use any absolute expression to specify *number_of_lines*. If *number_of_lines* is omitted, one line is left blank. If *number_of_lines* has a value greater than the number of lines remaining on the listing page, the instruction has the same effect as an EJECT statement.

The SPACE statement itself is not printed in the listing unless a variable symbol is specified as a point of substitution in the statement, in which case the statement is printed before substitution occurs. A blank line is equivalent to a `SPACE 1` statement.

START instruction

The START instruction can be used to initiate the first or only executable control section of a source module, and optionally to set an initial location counter value.



symbol
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol
- A sequence symbol

expression
Is an absolute expression, the value of which the assembler uses to set the location counter to an initial value for the source module.

Any symbols referenced in *expression* must have been previously defined.

The START instruction must be the first instruction of the first executable control section of a source module. It must not be preceded by any instruction that affects the location counter for an executable control section (that is, not a reference section such as COM, DXD, or DSECT), and thus causes the first executable control section to be initiated.

Use the START instruction to initiate the first or only control section of a source module, because it:

- Determines exactly where the first control section is to begin, thus avoiding the accidental initiation of the first control section by some other instruction.

- Gives a symbolic name to the first control section, which can then be distinguished from the other control sections listed in the external symbol dictionary.
- Specifies the initial setting of the location counter for the first or only control section.

If *symbol* denotes an ordinary symbol, the ordinary symbol identifies the first control section. It must be used in the name field of any CSECT instruction that indicates the continuation of the first control section. The ordinary symbol denoted by *symbol* represents the address of the first byte in the control section, and has a length attribute value of 1.

If *symbol* is not specified, or if *name* is a sequence symbol, the START instruction initiates an unnamed control section.

The assembler uses the value *expression* in the operand field, if specified, to set the location counter to an initial value for the source module. All control sections are aligned on the boundary specified by the SECTALGN option. Therefore, if the value specified in *expression* is not divisible by the SECTALGN value, the assembler sets the initial value of the location counter to the next higher required boundary. If *expression* is omitted, the assembler sets the initial value to 0.

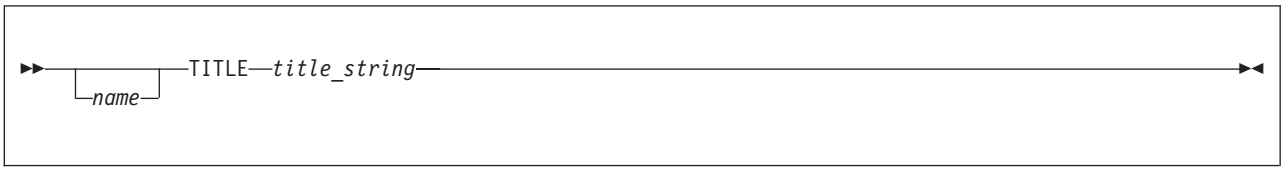
The source statements that follow the START instruction are assembled into the first control section. If a CSECT instruction indicates the continuation of the first control section, the source statements that follow this CSECT instruction are also assembled into the first control section.

Any instruction that defines a new or continued control section marks the end of the preceding control section. The END instruction marks the end of the control section in effect.

TITLE instruction

The TITLE instruction:

- Provides headings for each page of the *source and object* section of the assembler listing. If the first statement in your source program is an ICTL instruction or a *PROCESS statement then the title is not printed on the first page of the *Source and Object* section, because each of these instructions must precede all other instructions.
- Identifies the assembly output records of your object modules. You can specify up to 8 identification characters that the assembler includes as a *deck ID* in all object records, beginning at byte 73. If the deck ID is less than eight characters, the assembler puts sequence numbers in the remaining bytes up to byte 80.



name

You can specify *name* only once in the source module. It is one of the following:

- A string of printable characters
- A variable symbol that has been assigned a string of printable characters
- A combination of the above
- A sequence symbol

Except when the *name* is a sequence symbol, the assembler uses the first eight characters you specify, and discards the remaining characters without warning.

title_string
Is a string of 1 to 100 characters enclosed in apostrophes

If two or more TITLE instructions are together, the title provided by the last instruction is printed as the heading.

Deck ID in object records

When you specify the *name*, and it is not a sequence symbol, it has a special significance. The assembler uses the *name* value to generate the *deck ID* in object records. The deck ID is placed in the object records starting at byte 73. It is not generated for records produced by the PUNCH and REPRO instructions. The *name* value does not need to be on the first TITLE instruction.

The *name* value is not defined as a symbol, so it can be used in the name entry of any other statement in the same source module, provided it is a valid ordinary symbol.

GOFF Assembler Option (z/OS and CMS): When you specify the GOFF assembler option the deck ID is not generated.

Printing the heading

The character string denoted by *title_string* is printed as a heading at the top of each page of the *source and object* section of the assembler listing. The heading is printed beginning on the page in the listing that follows the page on which the TITLE instruction is specified. A new heading is printed each time a new TITLE instruction occurs in the source module. If the TITLE instruction is the first instruction in the source module the heading is printed on the first page of the listing.

When a TITLE instruction immediately follows an EJECT instruction, the assembler changes the title but does not perform an additional page-eject.

Printing the TITLE statement

The TITLE statement is printed in the listing when you specify a variable symbol in the *name*, or in the *title_string*, in which case the statement is printed before substitution occurs.

Sample program using the TITLE instruction

The following example shows three TITLE instructions:

```
PGM1     TITLE 'The First Heading'
PGM1     CSECT
         USING PGM1,12             Assign the base register
         TITLE 'The Next Heading'
         LR    12,15               Load the base address
&VARSYM  SETC 'Value from Variable Symbol'
         TITLE 'The &VARSYM'
         BR    14                  Return
         END
```

After the program is assembled, the characters PGM1 are placed in bytes 73 to 76 of all object records, and the heading appears at the top of each page in the listing as shown in Figure 27 on page 192. The TITLE instruction at statement 7 is printed because it contains a variable symbol.

```
PGM1     The First Heading                                                                          Page    3
  Active Usings: None
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0  2008/07/11 17.48
000000                00000 00004     2 PGM1     CSECT                                                 LRM00020
                R:C  00000           3          USING PGM1,12             Assign the base register
PGM1     The Next Heading                                                                           Page    4
  Active Usings: PGM1,R12
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0  2008/07/11 17.48
000000 18CF                          5          LR    12,15               Load the base address
                                     6 &VARSYM  SETC 'Value from Variable Symbol'
                                     7          TITLE 'The &VARSYM'
PGM1     The Value from Variable Symbol                                                             Page    5
  Active Usings: PGM1,R12
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0  2008/07/11 17.48
000002 07FE                          8          BR    14                  Return
                                     9          END                                                   LRM00090
```

Figure 27. Sample program using TITLE instruction

Page ejects

Each inline TITLE statement causes the listing to be advanced to a new page before the heading is printed unless it is preceded immediately by one of the following:

- A CEJECT instruction
- An EJECT instruction
- A SPACE instruction that positions the current print line at the start of a new page
- A TITLE instruction

If the TITLE statement appears in a macro or contains a variable symbol *and* PRINT NOGEN is specified, the listing is not advanced to a new page.

Valid characters

Any printable character specified appears in the heading, including spaces. Double-byte data can be used when the DBCS assembler option is specified. The double-byte data must be valid. Variable symbols are allowed. However, the following rules apply to ampersands and apostrophes:

- The DBCS ampersand and apostrophe are not recognized as delimiters.
- A double-byte character that contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.
- A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.
- A pair of ampersands is printed as one ampersand.
- A pair of apostrophes is printed as one apostrophe.
- An unpaired apostrophe followed by one or more spaces ends the string of characters printed. If a non-space character follows an unpaired apostrophe, the assembler issues an error message and prints no heading.

Only the characters printed in the heading count toward the maximum of 100 characters allowed. If the count of characters to be printed exceeds 100, the heading that is printed is truncated and error diagnostic message

```
ASMA062E Illegal operand format
```

is issued.

USING instruction

The USING instruction specifies a base address and range and assigns one or more base registers. If you also load the base register with the base address, you have established addressability in a control section. If a control section has not been established, USING initiates an unnamed (private) control section.

To use the USING instruction correctly, you should know:

- Which locations in a control section are made addressable by the USING instruction
- Where in a source module you can use implicit addresses in instruction operands to refer to these addressable locations

Base address

The term *base address* is used throughout this manual to mean the location counter value within a control section from which the assembler can compute displacements to locations, or *addresses*, within the control section. Do not confuse this with the storage address of a control section when it is loaded into storage at execution time.

The USING instruction has three formats:

- The first format specifies a base address, an optional range, and one or more base registers. This format of the USING instruction is called an *ordinary USING instruction*, and is described under "Ordinary USING instruction" on page 194.
- The second format specifies a base address, an optional range, one or more base registers, and a USING label which can be used as a symbol qualifier. This format of the USING instruction is called a *labeled USING instruction*, and is described under "Labeled USING instruction" on page 197.
- The third format specifies a base address, an optional range, and a relocatable expression instead of one or more base registers. This format of a USING instruction is called a *dependent USING instruction*, and is described under "Dependent USING instruction" on page 200. If a USING label is also specified, this format of the USING instruction is called a *labeled dependent USING instruction*.

Note: The assembler identifies and warns about statements where the implied alignment of an operand does not match the requirements of the instruction. However, if the base for a USING is not aligned on the required boundary, the assembler cannot diagnose a problem. For example:

```
DS1      DSECT
         DS    H
REGPAIR  DS    2ADL8              Halfword alignment
DS2      DSECT
REGPAIR_ALIGN  DS 2ADL8           Doubleword alignment
         CSECT
         ...
         USING DS1,R1             Ordinary USING
         USING DS2,REGPAIR        Dependent USING
         STPQ  R0,REGPAIR         REGPAIR is not a quadword
         STPQ  R0,REGPAIR_ALIGN   But REGPAIR_ALIGN is
```

The first STPQ instruction is diagnosed as an alignment error. The second STPQ instruction is not, even though the same storage location is implied by the code.

You must take care to ensure base addresses match the alignment requirements of storage mapped by a USING. For a description of the alignment requirements of instructions, see the relevant *z/Architecture Principles of Operation*.

How to use the USING instruction

Specify the USING instruction so that:

- All the required implicit addresses in each control section lie within a USING range.
- All the references for these addresses lie within the corresponding USING domain.

You could, therefore, place all ordinary USING instructions at the beginning of the control section and specify a base address in each USING instruction that lies at the beginning of each control section.

For Executable Control Sections: To establish the addressability of an executable control section defined by a START or CSECT instruction, specify a base address and assign a base register in the USING instruction. At execution time, the base register must be loaded with the correct base address.

If a control section requires addressability to more than 4096 bytes, you must assign more than one base register, or make implicit references using only instructions supporting 20 bit displacements ("long displacements"). This establishes the addressability of the entire control section with one USING instruction.

For Reference Control Sections: A dummy section is a reference control section defined by the DSECT instructions. To establish the addressability of a dummy section, specify the address of the first byte of the dummy section as the base address, so that all its addresses lie within the pertinent USING range. The address you load into the base register must be the address of the storage area being described by the dummy section. However, if all references to fields within the DSECT are made with instructions supporting long displacements, the base address need not be the first byte of the dummy section.

When you refer to symbolic addresses in the dummy section, the assembler computes displacements accordingly. However, at execution time, the assembled addresses refer to the location of real data in the storage area.

Base registers for absolute addresses

Absolute addresses used in a source module must also be made addressable. Absolute addresses require a base register other than the base register assigned to relocatable addresses (as described above).

However, the assembler does not need a USING instruction to convert absolute implicit addresses in the range 0 through 4095 to their explicit form. The assembler uses register 0 as a base register. Displacements are computed from the base address 0, because the assembler assumes that a base or index of 0 implies that a zero quantity is to be used in forming the address, regardless of the contents of register 0. The USING domain for this automatic base register assignment is the entire source module.

If a register is specified with base address zero, the assembler uses it in preference to the default use of register zero. For example:

```
USING 3,0
LA    7,5
```

generates the instruction X'41703005'; in the absence of the USING statement, the generated instruction is X'41700005'.

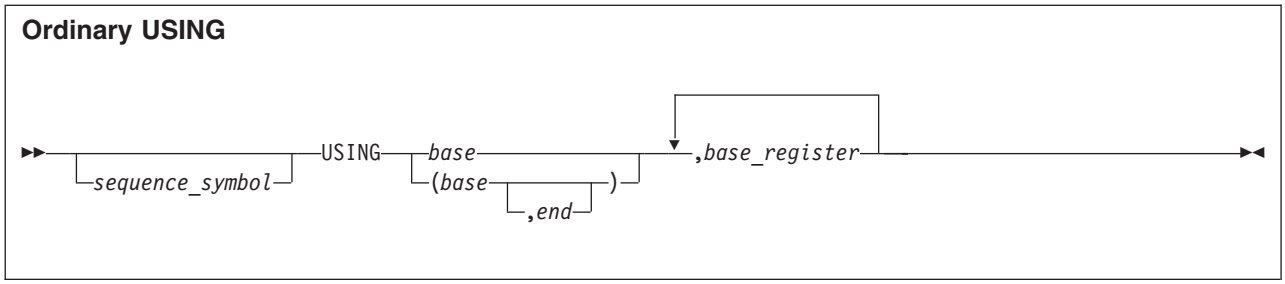
For absolute implicit addresses greater than 4095 and in the absence of long-displacement instructions, a USING instruction must be specified according to the following:

- With a base address representing an absolute expression
- With a base register that has not been assigned by a USING instruction in which a relocatable base address is specified

This base register must be loaded with the base address specified.

Ordinary USING instruction

The ordinary USING instruction format specifies a base address and one or more base registers.



sequence_symbol
Is a sequence symbol.

base
Specifies a base address, which can be a relocatable or an absolute expression. The value of the expression is 0 - 2^{31}-1.

end
Specifies the end address, which can be a relocatable or an absolute expression. The value of the expression is 0 - 2^{31}-1. The end address can exceed the (base address + default range) without error. The *end* address must be greater than the *base* and must have the same relocatability attribute.

The resolvable range of a USING with an 'end' operand is

```
base,MIN(4095,end-1)
```

Thus `USING base,reg` is equivalent to `USING (base,base+4096),reg`.

base_register
Is an absolute expression whose value represents general registers 0 through 15.

The default range is 4096 per base register.

The assembler assumes that the base register denoted by the first *base_register* operand contains the base address *base* at execution time. If present, the subsequent *base_register* operands represent registers that the assembler assumes contain the address values *base*+4096, *base*+8192, and so on.

For example:

```
        USING           BASE,9,10,11
```

has the logical equivalent of:

```
        USING           BASE,9
        USING           BASE+4096,10
        USING           BASE+8192,11
```

In another example, the following statement:

```
        USING           *,12,13
```

tells the assembler to assume that the current value of the location counter is in general register 12 at execution time, and that the current value of the location counter, incremented by 4096, is in general register 13 at execution time.

Computing displacement

If you change the value in a base register being used, and want the assembler to compute displacements from this value, you must tell the assembler the new value with another USING statement. In the following sequence, the assembler first assumes that the value of `ALPHA` is in register 9. The second statement then causes the assembler to assume that `ALPHA+1000` is the value in register 9.

```
USING           ALPHA,9
.
.
USING           ALPHA+1000,9
```

Using General Register Zero

You can refer to the first 4096 bytes of storage using general register 0, subject to the following conditions:

- The value of operand *base* must be either absolute or relocatable zero.
- Register 0 must be specified as the first *base_register* operand.

The assembler assumes that register 0 contains zero. Therefore, regardless of the value of operand *base*, it calculates displacements as if operand *base* were absolute or relocatable zero. The assembler also assumes that subsequent registers specified in the same USING statement contain 4096, 8192, and so on.

If register 0 is used as a base register, the referenced control section (or dummy section) is not relocatable, despite the fact that operand *base* might be relocatable. The control section can be made relocatable by:

- Replacing register 0 in the USING statement
- Loading the new register with a relocatable value
- Reassembling the program

Range of an ordinary USING instruction

The range of an ordinary USING instruction (called the “ordinary USING range”, or the “USING range”) is the 4096 bytes beginning at the base address specified in the USING instruction, or the range as specified by the range end, whichever is the lesser. For long-displacement instructions, the range is the addresses between (*base_address*-524288) and (*base_address*+524287). Addresses that lie within the USING range can be converted from their implicit to their explicit base-displacement form using the designated base registers; those outside the USING range cannot be converted.

The USING range does not depend upon the position of the USING instruction in the source module; rather, it depends upon the location of the base address specified in the USING instruction.

The USING range is the range of addresses in a control section that is associated with the base register specified in the USING instruction. If the USING instruction assigns more than one base register, the composite USING range is the union of the USING ranges that applies if the base registers were specified in separate USING instructions.

USING ranges need not be contiguous. For example, you can specify

```
USING X,4
USING X+6000,5
```

and implicit addresses with values X+4096 - X+5999 are not addressable by instructions with unsigned 12 bit displacements.

Two USING ranges coincide when the same base address is specified in two different USING instructions, even though the base registers used are different. When two USING ranges coincide, the assembler uses the higher-numbered register for assembling the addresses within the common USING range. In effect, the domain of the USING instruction that specifies the lower-numbered register is ended by the other USING instruction. If the domain of the USING instruction that specifies the higher-number register is terminated, the domain of the other USING instruction is resumed.

Two USING ranges overlap when the base address of one USING instruction lies within the range of another USING instruction. You can use the WARN suboption of the USING assembler option to find out if you have any overlapping USING ranges. When an overlap occurs the assembler issues a diagnostic message. However, the assembler does allow an overlap of one byte in USING ranges so that you do not receive a diagnostic message if you code the following statements:

```
PSTART   CSECT
         LR     R12,R15
         LA     R11,4095(,R12)
         USING  PSTART,R12
         USING  PSTART+4095,R11
```

In the above example, the second USING instruction begins the base address of the second base register (R11) in the 4096th byte of the first base register (R12) USING range. If you do not want the USING ranges to overlap, you can code the following statements:

```
PSTART   CSECT
         LR     R12,R15
         LA     R11,4095(,R12)
         LA     R11,1(,R11)
         USING  PSTART,R12
         USING  PSTART+4096,R11
```

When two ranges overlap, the assembler computes displacements from the base address that gives the smallest non-negative displacement; or if no non-negative displacement can be found, for long-displacement instructions, the base register giving the smallest negative displacement; it uses the corresponding base register when it assembles the addresses within the range overlap. This applies only to implicit addresses that appear after the second USING instruction.

LOCTR does not affect the USING domain.

Domain of an ordinary USING instruction

The domain of an ordinary USING instruction (called the "ordinary USING domain", or the "USING domain") begins where the USING instruction appears in a source module. It continues until the end of a source module, except when:

- A subsequent DROP instruction specifies the same base register or registers assigned by a preceding USING instruction.
- A subsequent USING instruction specifies the same register or registers assigned by a preceding USING instruction.

The assembler converts implicit address references into their explicit form when the following conditions are met:

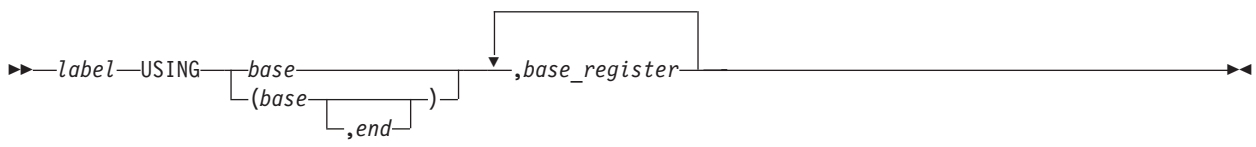
- The address reference appears in the domain of a USING instruction.
- The addresses referred to lie within the range of some USING instruction.

The assembler does not convert implicit address references that are outside the USING domain. The USING domain depends on the position of the USING instruction in the source module after conditional assembly, if any, has been done.

Labeled USING instruction

The labeled USING instruction specifies a base address, one or more base registers, and a USING label which can be used as a symbol qualifier.

Labeled USING



label
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

base
Specifies a base address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and 2^{31}-1.

end
Specifies the end address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and 2^{31}-1. The end address can exceed the (base address + default range) without error. The *end* address must be greater than the *base* and must have the same relocatability attributes.

base_register
Is an absolute expression whose value represents general registers 0 through 15.

The default range is 4096 per base register.

The essential difference between a labeled USING instruction and an ordinary USING instruction is the label placed on the USING statement. To indicate to the assembler that the USING established with the label is to provide resolution of base and displacement for a symbol, the label must be used to qualify the symbol. Qualifying a symbol consists of preceding the symbol with the label on the USING followed by a period. The only symbols resolved by the labeled USING are those symbols qualified with the label. This label cannot be used for any other purpose in the program, except possibly as a label on other USING instructions.

The following examples show how labeled USINGs are used:

```
PRIOR    USING  IHADCB,R10
NEXT     USING  IHADCB,R2
         MVC    PRIOR.DCBLRECL,NEXT.DCBLRECL
```

The same code without labeled USINGs can be written like this:

```
         USING  IHADCB,R10
         MVC    DCBLRECL,DCBLRECL-IHADCB(R2)
```

In the following example, a new element, NEW, is inserted into a doubly linked list between two existing elements LEFT and RIGHT, where the links are stored as pointers LPTR and RPTR:

```
LEFT     USING  ELEMENT,R3
RIGHT    USING  ELEMENT,R6
NEW      USING  ELEMENT,R1
         .
         .
         MVC    NEW.RPTR,LEFT.RPTR     Move previous Right pointer
         MVC    NEW.LPTR,RIGHT.LPTR    Move previous Left pointer
         ST     R1,LEFT.RPTR           Chain new element from Left
         ST     R1,RIGHT.LPTR          Chain new element from Right
         .
         .
ELEMENT  DSECT
LPTR     DS     A                      Link to left element
RPTR     DS     A                      Link to right element
         .
         .
```

Range of a labeled USING instruction

The range of a labeled USING instruction (called the *labeled USING range*) is the 4096 bytes beginning at the base address specified in the labeled USING instruction, or the range as specified by the range end,

whichever is the lesser. Addresses that lie within the labeled USING range can be converted from their implicit form (qualified symbols) to their explicit form; those outside the USING range cannot be converted.

Like the ordinary USING range, the labeled USING range is the range of addresses in a control section that is associated with the base register specified in the labeled USING instruction. If the labeled USING instruction assigns more than one base register, the composite labeled USING range is the product of the number of registers specified in the labeled USING instruction and 4096 bytes. The composite labeled USING range begins at the base address specified in the labeled USING instruction. Unlike the ordinary USING range, however, you cannot specify separate labeled USING instructions to establish the same labeled USING range. For example,

```
IN      USING BASE,10,11
```

specifies a range of 8192 bytes beginning at BASE, but

```
IN      USING BASE,10
IN      USING BASE+4096,11
```

specifies a single labeled USING range of 4096 bytes beginning at BASE+4096.

You can specify the same base address in any number of labeled USING instructions. You can also specify the same base address in an ordinary USING and a labeled USING. However, unlike ordinary USING instructions that have the same base address, if you specify the same base address in an ordinary USING instruction and a labeled USING instruction, High Level Assembler does not treat the USING ranges as coinciding. When you specify an unqualified symbol in an assembler instruction, the base register specified in the ordinary USING is used by the assembler to resolve the address into base-displacement form. Here is an example of coexistent ordinary USINGs and labeled USINGs:

```
         USING  IHADCB,R10
SAMPLE   USING  IHADCB,R2
         MVC    DCBLRECL,SAMPLE.DCBLRECL
```

In this MVC instruction, the (unqualified) first operand is resolved with the ordinary USING, and the (qualified) second operand is resolved with the labeled USING.

Domain of a labeled USING instruction

The domain of a labeled USING instruction (called the *labeled USING domain*) begins where the USING instruction appears in a source module. It continues to the end of the source module, except when:

- A subsequent DROP instruction specifies the label used in the preceding labeled USING instruction.
- A subsequent USING instruction specifies the same label used in the preceding labeled USING instruction. The second specification of the label causes the assembler to end the domain of the prior USING with the same label.

You can specify the same base register or registers in any number of labeled USING instructions. However, unlike ordinary USING instructions, as long as all the labeled USINGs have unique labels, the assembler considers the domains of all the labeled USINGs to be active and their labels eligible to be used as symbol qualifiers. With ordinary USINGs, when you specify the same base register in a subsequent USING instruction, the domain of the prior USING is ended.

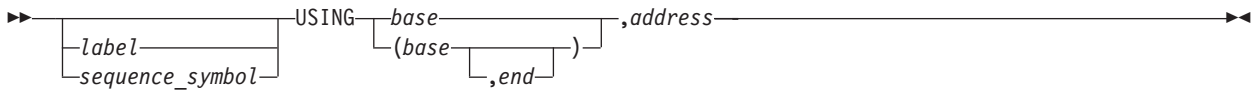
The assembler converts implicit address references into their explicit form using the base register or registers specified in a labeled USING instruction when the following conditions are met:

- The address reference appears in the domain of the labeled USING instruction.
- The address reference takes the form of a qualified symbol and the qualifier is the label of the labeled USING instruction.
- The address lies within the range of the labeled USING instruction.

Dependent USING instruction

The dependent USING instruction format specifies a base address and a relocatable expression instead of one or more base registers. If a USING label is also specified, this format USING instruction is called a *labeled dependent USING* instruction.

Dependent USING



label
Is one of the following:

- An ordinary symbol
- A variable symbol that has been assigned a character string with a value that is valid for an ordinary symbol

sequence_symbol
Is a sequence symbol.

base
Specifies a base address, which must be a relocatable expression. The value of the expression must lie between 0 and 2^{31}-1.

address
Is a simply relocatable expression that represents an implicit address within the range of an active USING instruction. The range of an active USING is considered to be that which is valid for generating 12 bit or 20 bit displacements.

end
Specifies the end address, which can be a relocatable or an absolute expression. The value of the expression must lie between 0 and 2^{31}-1. The end address can exceed the (base address + default range) without error. The *end* address must be greater than the *base* and must have the same relocatability attributes.

The implicit address denoted by *address* specifies the address where *base* is to be based, and is known as the *supporting base address*. As *address* is a relocatable expression, it distinguishes a dependent USING from an ordinary USING. The assembler converts the implicit address denoted by *address* into its explicit base-displacement form. It then assigns the base register from this explicit address as the base register for *base*. The assembler assumes that the base register contains the base address *base* minus the displacement determined in the explicit address. The assembler also assumes that *address* is appropriately aligned for the code based on *base*. Warnings are not issued for potential alignment problems in the dependent USING *address*.

A dependent USING depends on the presence of one or more corresponding labeled or ordinary USINGs being in effect to resolve the symbolic expressions in the range of the dependent USING.

The following example shows the use of an unlabeled dependent USING:

```
EXAMPLE  CSECT
         USING  EXAMPLE,R10,R11               Ordinary USING
         .
         .
         USING  IHADCB,DCBUT2                 Unlabeled dependent USING
```

```
         LH     R0,DCBBLKSI                    Uses R10 or R11 for BASE
         .
         .
DCBUT2   DCB    DDNAME=SYSUT2,...
```

The following example shows the use of two labeled dependent USINGs:

```
EXAMPLE  CSECT
         USING  EXAMPLE,R10,R11                Ordinary USING
         .
         .
DCB1     USING  IHADCB,DCBUT1                  Labeled dependent USING
DCB2     USING  IHADCB,DCBUT2                  Labeled dependent USING
         MVC    DCB2.DCBBLKSI,DCB1.DCBBLKSI    Uses R10 or R11 for BASE
         .
         .
DCBUT1   DCB    DDNAME=SYSUT1,...
DCBUT2   DCB    DDNAME=SYSUT2,...
```

Range of a dependent USING instruction

The range of a dependent USING instruction (called the dependent USING range) is either the range as specified by the range end, or the range of the corresponding USING minus the offset of *address* within that range, whichever is the lesser. If the corresponding labeled or ordinary USING assigns more than one base register, the maximum dependent USING range is the composite USING range of the labeled or ordinary USING.

If the dependent USING instruction specifies a supporting base address that is within the range of more than one ordinary USING, the assembler determines which base register to use during base-displacement resolution as follows:

- The assembler computes displacements from the ordinary USING base address that gives the smallest displacement, and uses the corresponding base register.
- If more than one ordinary USING gives the smallest displacement, the assembler uses the higher-numbered register for assembling addresses within the coinciding USING ranges.

Domain of a dependent USING instruction

The domain of a dependent USING instruction (called the dependent USING domain) begins where the dependent USING appears in the source module and continues until the end of the source module, except when:

- You end the domain of the corresponding ordinary USING by specifying the base register or registers from the ordinary USING instruction in a subsequent DROP instruction.
- You end the domain of the corresponding ordinary USING by specifying the same base register or registers from the ordinary USING instruction in a subsequent ordinary USING instruction.
- You end the domain of a labeled dependent USING by specifying the label of the labeled dependent USING in the operand of a subsequent DROP instruction.
- You end the domain of a labeled dependent USING by specifying the label of the labeled dependent USING in the operand of a subsequent labeled USING instruction.

When a labeled dependent USING domain is dropped, none of any subordinate USING domains are dropped. In the following example the labeled dependent USING BLBL1 is not dropped, even though it appears to be dependent on the USING ALBL2 that is being dropped:

```
ALBL1    USING          DSECTA,14
         USING          DSECTB,ALBL1.A
         .
         .
ALBL2    USING          DSECTA,ALBL1.A
         .
BLBL1    USING          DSECTA,ALBL2.A
         .
         DROP           ALBL2
```

```
         .
DSECTA   DSECT
A        DS              A
DSECTB   DSECT
B        DS              A
```

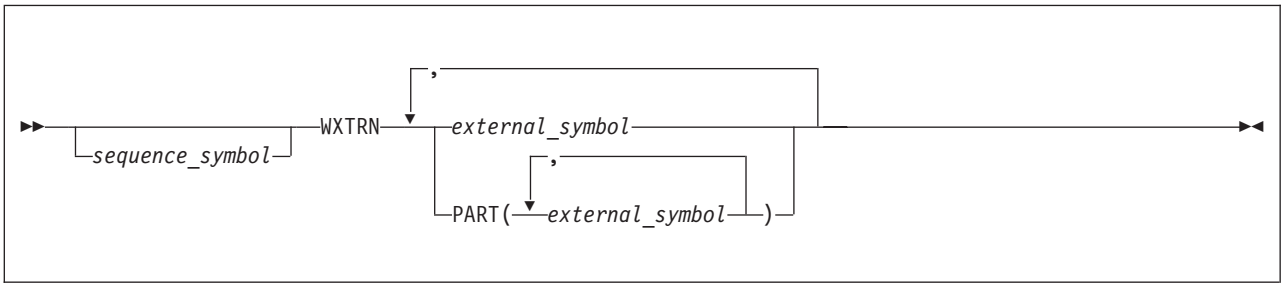
A dependent USING is *not* dependent on another dependent USING. It *is* dependent on the ordinary or labeled USING that is finally used to resolve the address. For example, the USING at BLBL1 is dependent on the ALBL1 USING.

Remember that all dependent USINGs must eventually be based on an ordinary or labeled USING that provides the base register used for base-displacement resolutions.

WXTRN instruction

The WXTRN statement identifies "weak external" symbols referred to in a source module but defined in another source module. The WXTRN instruction differs from the EXTRN instruction (see "EXTRN instruction" on page 167) as follows:

- The EXTRN instruction causes the linker to automatically search libraries (if automatic library call is in effect) to find the module that contains the external symbols that you identify in its operand field. If the module is found, linkage addresses are resolved; the module is then linked to your module, which contains the EXTRN instruction.
- The WXTRN instruction suppresses automatic search of libraries. The linker only resolves the linkage addresses if the external symbols that you identify in the WXTRN operand field are defined in one of these ways:
  - In a module that is linked and loaded along with the object module assembled from your source module.
  - In a module brought in from a library because of the presence of an EXTRN instruction in another module linked and loaded with yours.



sequence_symbol
Is a sequence symbol.

external_symbol
Is a relocatable symbol that is not:

- Used as the name entry of a source statement in the source module in which it is defined

PART(*external_symbol***)**
external_symbol is a relocatable symbol as described above, that also:

- Is a reference to a part as defined on the CATTR instruction.

The external symbols identified by a WXTRN instruction have the same properties as the external symbols identified by the EXTRN instruction. However, the type code assigned to these external symbols differs.

V-Type Address Constant: If a symbol, specified in a V-type address constant, is also identified by a WXTRN instruction, it is assigned the same ESD type code as the symbol in the WXTRN instruction, and is treated by the linkage editor as a *weak external* symbol.

If an external symbol is identified by both an EXTRN and WXTRN instruction in the same source module, the first declaration takes precedence, and subsequent declarations are flagged with diagnostic messages.

XATTR instruction (z/OS and CMS)

The XATTR instruction enables attributes to be assigned to an external symbol. This instruction is only valid when you specify the GOFF assembler option.

symbol
Is a symbol which has been declared implicitly or explicitly as an external symbol. Further, if the PSECT attribute is specified, must be an RSECT, CSECT, or START name or an ENTRY name (where the entry is in one of the preceding types of section)

attribute
Is one or more attributes from the group of attributes described below. The assembler sets the appropriate attribute flags in the GOFF External Symbol Directory record.

Notes:

1. If more than one value is specified for a given attribute, no diagnostic is issued and only the last value is used.
2. All attributes of an external symbol must be specified in a single XATTR statement (which can be continued).

ATTRIBUTES

ATTRIBUTES(*label*), abbreviation ATTR(*label*)
Is a symbol (internal or external) known in the declaring program. It names the location of the extended attribute information to be associated with *symbol*.

Instructs the assembler to place the ESDID and offset of the label in the GOFF External Symbol Dictionary record.

LINKAGE

LINKAGE(OS), abbreviation LINK(OS)
Instructs the assembler to set the "Linkage Type" attribute to standard OS linkage.

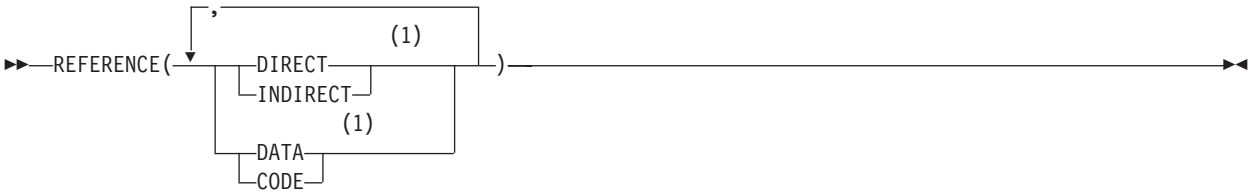
LINKAGE(XPLINK), abbreviation LINK(XPLINK)
Instructs the assembler to set the "Linkage Type" attribute to indicate "Extra Performance Linkage".

PSECT

PSECT (*name*)
Identifies the private read-write "section" or PSECT associated with *name* by its being an internal or external symbol belonging to an element in the class to which the PSECT belongs. The *name* is one of:

- An ENTRY name, where the entry is in the same section (CSECT or RSECT) as *name*, but in a different class. For reentrant code, the PSECT is normally a non-shared class, so a separate CATTR statement is needed to declare that class and its attributes.
- An internal label within the PSECT.

REFERENCE



Notes:

1 Select no more than one option from each group.

REFERENCE(DIRECT), abbreviation REF(DIRECT)
Instructs the assembler to reset (clear) the "Indirect Reference" attribute.

REFERENCE(INDIRECT), abbreviation REF(INDIRECT)
Instructs the assembler to assign the "Indirect Reference" attribute.

REFERENCE(CODE), abbreviation REF(CODE)
Instructs the assembler to set the Executable attribute.

REFERENCE(DATA), abbreviation REF(DATA)
Instructs the assembler to set the Not Executable attribute.

SCOPE

SCOPE(SECTION), abbreviation SCOPE(S)
Instructs the assembler to set the binding scope to "Section".

SCOPE(MODULE), abbreviation SCOPE(M)
Instructs the assembler to set the binding scope to "Module".

SCOPE(LIBRARY), abbreviation SCOPE(L)
Instructs the assembler to set the binding scope to "Library".

SCOPE(IMPORT), abbreviation SCOPE(X)
Instructs the assembler to set the binding scope to "Export-Import" (see note following this list).

SCOPE(EXPORT), abbreviation SCOPE(X)
Instructs the assembler to set the binding scope to "Export-Import".

This statement indicates only that the name field symbol has the specified scope. A symbol having SCOPE(X) has IMPORT status only if declared in an EXTRN statement, and has EXPORT status only if declared explicitly in an ENTRY statement, or declared implicitly as an entry on a CSECT or RSECT statement.

The SCOPE(IMPORT) or SCOPE(EXPORT) attribute is required for using Dynamic Link Libraries under the Language Environment®. For details, refer to *z/OS Language Environment Programming Guide* (SA22-7561).

Association of code and data areas (z/OS and CMS)

To provide support for application program reentrancy and dynamic binding, the assembler provides a way to associate read-only code and read-write data areas. This is done by defining and accessing "associated data areas" called PSECTs. A PSECT (Private or Prototype Control Section) when instantiated becomes the non-shared working storage for an invocation of a shared reentrant program.

In the Program Object model, a PSECT is an element within the same section as the element containing the shared code to which it belongs. The two classes defining these elements have attributes appropriate to their respective uses.

Typically, V-type and R-type address constants are used to provide code and data-area addressability for a reentrant program using PSECTs.

Figure 28 on page 206 shows an example of two sections A and B, each with a PSECT. When the program object AB containing A and B is instantiated, a single copy of the reentrant CODE class is loaded into read-only storage, and a copy of the PSECT class belonging to AB is loaded into read-write storage. The invoker of A provides the address for the PSECT of A, so that A can address its own read-write data. A later instantiation of AB loads only a new copy of the PSECT class.

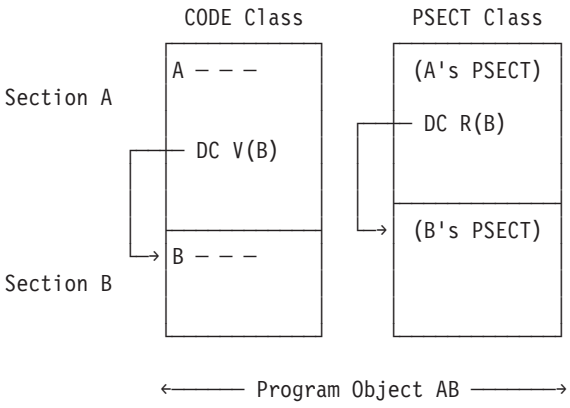


Figure 28. Program object with PSECTs, example 1

When a program in the CODE class of section A calls a program in the CODE class of section B, a linkage convention might require loading the entry address of B into general register 15 and the address of B's PSECT into general register 0. For example:

```
L     15,=V(B)      B's entry point address
L     0,=R(B)       B's PSECT address (from A's PSECT)
BASR  14,15         Linkage to B
```

Further information about linkage conventions for referencing Dynamic Link Libraries (DLLs) under the Language Environment can be found in *z/OS Language Environment Programming Guide* (SA22-7561).

Chapter 6. Introduction to macro language

This chapter introduces the basic macro concept: what you can use the macro facility for, how you can prepare your own macro definitions, and how you call these macro definitions for processing by the assembler.

Macro language is an extension of assembler language. It provides a convenient way to generate a sequence of assembler language statements many times in one or more programs. A macro definition is written only once; thereafter, a single statement, a macro instruction statement, is written each time you want to generate the sequence of statements. This simplifies the coding of programs, reduces the chance of programming errors, and ensures that standard sequences of statements are used to accomplish the functions you want.

In addition, conditional assembly lets you code statements that are assembled or not, depending upon conditions evaluated at conditional assembly time. These conditions are generally tests of values which can be defined, set, changed, and tested during assembly. Conditional assembly statements can be used within macro definitions or in open code.

Using macros

The main use of macros is to insert assembler language statements into a source program.

You call a named sequence of statements (the *macro definition*) by using a macro instruction, or *macro call*. The assembler replaces the macro call by the statements from the macro definition and inserts them into the source module at the point of call. The process of inserting the text of the macro definition is called *macro generation* or macro expansion. Macro generation occurs during conditional assembly.

The expanded stream of code then becomes the input for processing at assembly time; that is, the time at which the assembler translates the machine instructions into object code.

Macro definition

A macro definition is a named sequence of statements you can call with a macro instruction. When it is called, the assembler processes and normally generates assembler language statements from the definition into the source module. The statements generated can be:

- Copied directly from the definition
- Modified by parameter values and other values in variable symbols before generation
- Manipulated by internal macro processing to change the sequence in which they are generated

You can define your own macro definitions in which any combination of these three processes can occur. Some macro definitions, like some of those used for system generation, do not generate assembler language statements, but do only internal processing.

A macro definition provides the assembler with:

- The name of the macro
- The parameters used in the macro
- The sequence of statements the assembler generates when the macro instruction appears in the source program.

Every macro definition consists of a macro definition header statement (MACRO), a macro instruction prototype statement, one or more assembler language statements, and a macro definition trailer statement

(MEND), as shown in Figure 29.

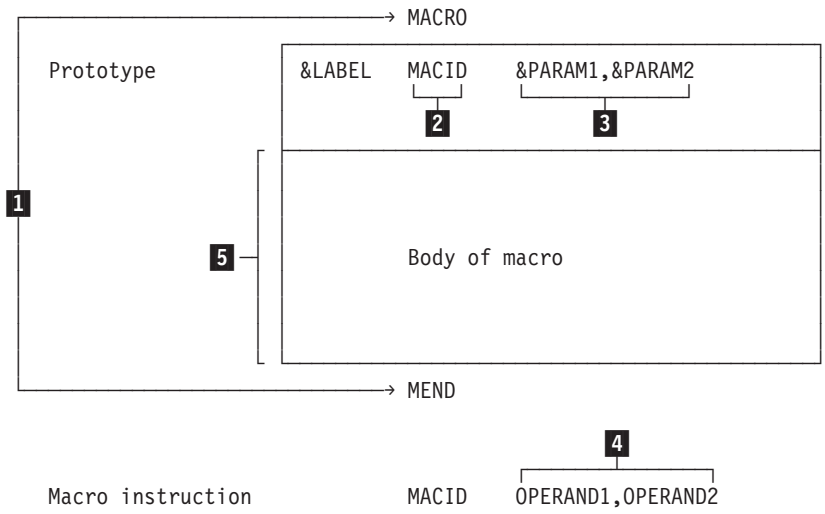


Figure 29. Parts of a macro definition

- The macro definition header and trailer statements (MACRO and MEND) indicate to the assembler the beginning and end of a macro definition (see 1 in Figure 29).
- The macro instruction prototype statement names the macro (see 2 in Figure 29), and declares its parameters (see 3 in Figure 29). In the operand field of the macro instruction, you can assign values (see 4 in Figure 29) to the parameters declared for the called macro definition.
- The body of a macro definition (see 5 in Figure 29) contains the statements that are generated when you call the macro. These statements are called *model statements*; they are normally interspersed with conditional assembly statements or other processing statements.

Model statements

You can write machine instruction statements and assembler instruction statements as model statements. During macro generation, the assembler copies them exactly as they are written. You can also use variable symbols as points of substitution in a model statement. The assembler enters values in place of these points of substitution each time the macro is called.

The three types of variable symbols in the assembler language are:
- Symbolic parameters, declared in the prototype statement
- System variable symbols
- SET symbols, which are part of the conditional assembly language

The assembler processes the generated statements, with or without value substitution, at assembly time.

Processing statements

Processing statements are processed during conditional assembly, when macros are expanded, but they are not themselves generated for further processing at assembly time. The processing statements are:
- AEJECT instructions
- AREAD instructions
- ASPACE instructions
- Conditional assembly instructions
- Inner macro calls
- MEXIT instructions
- MNOTE instructions

The AEJECT and ASPACE instructions let you control the listing of your macro definition. Use the AEJECT instruction to stop printing the listing on the current page and continue printing on the next. Use the ASPACE instruction to insert blank lines in the listing. The AEJECT instruction is described in "AEJECT instruction" on page 225. The ASPACE instruction is described in "ASPACE instruction" on page 227.

The AREAD instruction assigns a character string value, of a statement that is placed immediately after a macro instruction, to a SETC symbol. The AREAD instruction is described in "AREAD instruction" on page 225.

Conditional assembly instructions, inner macro calls, and macro processing instructions are described in detail in the following chapters.

The MNOTE instruction generates an error message with an error condition code attached, or generates comments in which you can display the results of a conditional assembly computation. The MNOTE instruction is described in "MNOTE instruction" on page 173.

The MEND statement delimits the contents of a macro definition, and also provides an exit from the definition. The MEND instruction is described in "MEND statement" on page 214.

The MEXIT instruction tells the assembler to stop processing a macro definition, and provides an exit from the macro definition at a point before the MEND statement. The MEXIT instruction is described in "MEXIT instruction" on page 228.

Comment statements

One type of comment statement describes conditional assembly operations and is not generated. The other type describes assembly-time operations and is, therefore, generated. For a description of the two types of comment statements, see "Comment statements" on page 229.

Macro instruction

A macro instruction is a source program statement that you code to tell the assembler to process a particular macro definition. The assembler generates a sequence of assembler language statements for each occurrence of the same macro instruction. The generated statements are then processed as any other assembler language statement.

The macro instruction provides the assembler with:

- The name of the macro definition to be processed.
- The information or values to be passed to the macro definition. The assembler uses the information either in processing the macro definition or for substituting values into a model statement in the definition.

The output from a macro definition, called by a macro instruction, can be:

- A sequence of statements generated from the model statements of the macro for further processing at assembly time.
- Values assigned to global SET symbols. These values can be used in other macro definitions and in open code.

You can call a macro definition by specifying a macro instruction anywhere in a source module. You can also call a macro definition from within another macro definition. This type of call is an inner macro call; it is said to be nested in the macro definition.

Source and library macro definitions

You can include a macro definition in a source module. This type of definition is called a *source macro definition*, or, sometimes, an *in-line macro definition*.

You can also insert a macro definition into a system or user library by using the applicable utility program. This type of definition is called a *library macro definition*. The IBM-supplied macro definitions are examples of library macro definitions.

You can call a source macro definition only from the source module in which it is included. You can call a library macro definition from any source module if the library containing the macro definition is available to the assembler.

Syntax errors in processing statements are handled differently for source macro definitions and library macro definitions. In source macro definitions, error messages are listed following the statements in error. In library macros, however, error messages cannot be associated with the statement in error, because the statements in library macro definitions are not included in the assembly listing. Therefore, the error messages are listed directly following the first call of that macro.

Because of the difficulty of finding syntax errors in library macros, run and "debug" a macro definition as a source macro before placing it in a macro library. Alternatively, use the LIBMAC assembler option to have the assembler automatically include the source statements of the library macro in your source module. For more information about the LIBMAC option, see the section "LIBMAC" in the *HLASM Programmer's Guide*.

Macro library

The same macro definition can be made available to more than one source program by placing the macro definition in the macro library. The macro library is a collection of macro definitions that can be used by all the assembler language programs in an installation. When a macro definition has been placed in the macro library, it can be called by coding its corresponding macro instruction in a source program. Macro definitions must be in a macro library with a member name that is the same as the macro name. The procedure for placing macro definitions in the macro library is described in the applicable utilities manual.

The DOS/VSE assembler requires library macro definitions to be placed in the macro library in a special edited format. High Level Assembler does not require this. Library macro definitions must be placed in the macro library in source statement format. If you wish to use edited macros in z/VSE you can provide a LIBRARY exit to read the edited macros and convert them into source statement format. A library exit is supplied with z/VSE and is described in *z/VSE: Guide to System Functions*.

System macro instructions

The macro instructions that correspond to macro definitions prepared by IBM are called *system macro instructions*. System macro instructions are described in the applicable operating system manuals that describe macro instructions for supervisor services and data management.

Conditional assembly language

The conditional assembly language is a programming language with most of the features that characterize a programming language. For example, it provides:

- Variables
- Data attributes
- Expression computation
- Assignment instructions
- Labels for branching
- Branching instructions

- Substring operators that select characters from a string

Use the conditional assembly language in a macro definition to receive input from a calling macro instruction. You can produce output from the conditional assembly language by using the MNOTE instruction.

Use the functions of the conditional assembly language to select statements for generation, to determine their order of generation, and to do computations that affect the content of the generated statements.

The conditional assembly language is described in Chapter 9, "How to write conditional assembly instructions," on page 279.

Chapter 7. How to specify macro definitions

A *macro definition* is a set of statements that defines the name, the format, and the conditions for generating a sequence of assembler language statements. The macro definition can then be called by a macro instruction to process the statements. See page "Macro instruction" on page 209 for a description of the macro instruction. To define a macro you must:

- Give it a name
- Declare any parameters to be used
- Write the statements it contains
- Establish its boundaries with a macro definition header statement (MACRO) and a macro definition trailer statement (MEND)

Except for conditional assembly instructions, this chapter describes all the statements that can be used to specify macro definitions. Conditional assembly instructions are described in Chapter 9, "How to write conditional assembly instructions," on page 279.

Where to define a macro in a source module

Macro definitions can appear anywhere in a source module. They remain in effect for the rest of your source module, or until another macro definition defining a macro with the same operation code is encountered, or until an OPSYN statement deletes its definition. Thus, you can redefine a macro at any point in your program. The new definition is used for all subsequent calls to the macro in the program.

This type of macro definition is called a *source macro definition*, or, sometimes, an *in-line macro definition*. A macro definition can also reside in a system library; this type of macro is called a *library macro definition*. Either type can be called from the source module by the applicable macro instruction.

Macro definitions can also appear inside other macro definitions. There is no limit to the levels of macro definitions permitted.

The assembler does not process inner macro definitions until it finds the definition during the processing of a macro instruction calling the outer macro. The following example shows an inner macro definition:

Example:

```
        MACRO                           Macro header for outer macro
        OUTER           &A,&C=          Macro prototype
        AIF             ('&C' EQ '').A
        MACRO                           Macro header for inner macro
        INNER                           Macro prototype
        .
        .
        MEND                            Macro trailer for inner macro
.A      ANOP
        .
        MEND                            Macro trailer for outer macro
```

The assembler does not process the macro definition for `INNER` until `OUTER` is called with a value for `&C` other than a null string.

Open Code: Open code is that part of a source module that lies outside of any source macro definition. At coding time, it is important to distinguish between source statements that lie in open code, and those that lie inside macro definitions.

Format of a macro definition

The general format of a macro definition is shown in Figure 30. The four parts are described in detail in the following sections.

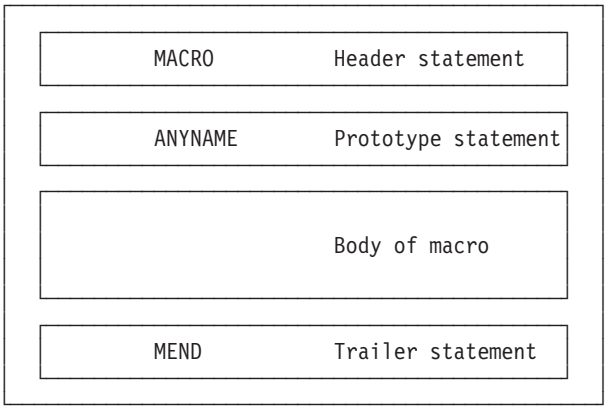


Figure 30. Format of a macro definition

Macro definition header and trailer

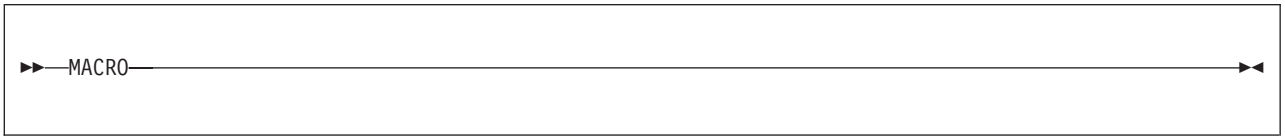
You must establish the boundaries of a macro definition by coding:

- A macro definition header statement as the first statement of the macro definition (a MACRO statement)
- A macro definition trailer statement as the last statement of the macro definition (a MEND statement)

The instructions used to define the boundaries of a macro instruction are described in the following sections.

MACRO statement

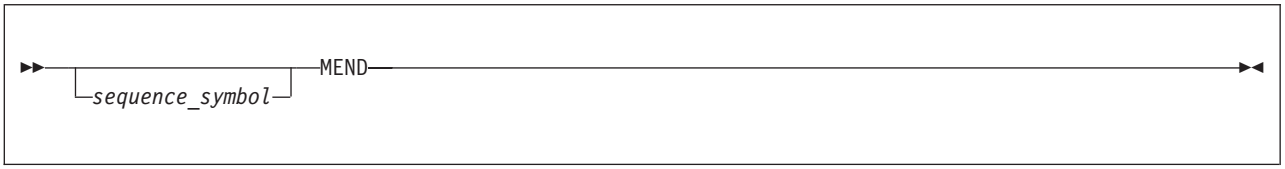
Use the MACRO statement to indicate the beginning of a macro definition. It must be the first non-comment statement in every macro definition. Library macro definitions can have ordinary or internal macro comments before the MACRO statement.



The MACRO statement must not have a name entry or an operand entry.

MEND statement

Use the MEND statement to indicate the end of a macro definition. It also provides an exit when it is processed during macro expansion. It can appear only once within a macro definition and must be the last statement in every macro definition.



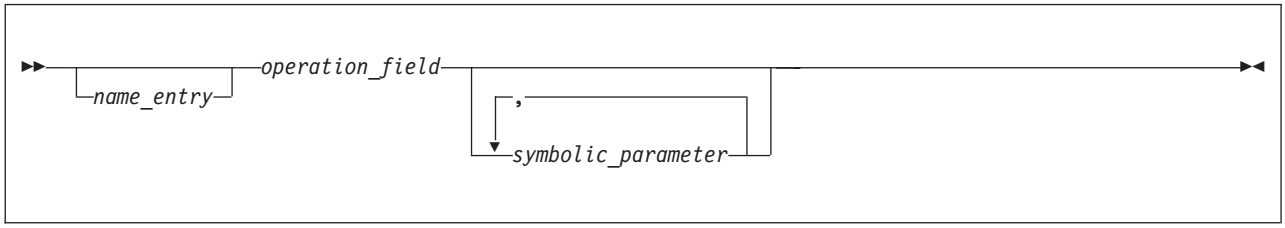
sequence_symbol
Is a sequence symbol.

See "MEXIT instruction" on page 228 for details on exiting from a macro before the MEND statement.

Macro instruction prototype

The macro instruction prototype statement (from here on called the "prototype statement") specifies the mnemonic operation code and the format of all macro instructions that you use to call the macro definition.

The prototype statement must be the second non-comment statement in every macro definition. Both ordinary comment statements and internal comment statements are allowed between the macro definition header and the macro prototype. Such comment statements are listed only with the macro definition.



name_entry
Is a variable symbol.

You can write this parameter, like the symbolic parameter, as the name entry of a macro prototype statement. You can then assign a value to this parameter from the name entry in the calling macro instruction.

If this parameter also appears in the body of a macro, it is given the value assigned to the parameter in the name field of the corresponding macro instruction.

operation_field
Is an ordinary symbol.

The symbol in the operation field of the prototype statement establishes the name by which a macro definition must be called. This name becomes the operation code required in any macro instruction that calls the macro.

Any operation code can be specified in the prototype operation field. If the entry is the same as an assembler or a machine operation code, the new definition overrides the previous use of the symbol. The same is true if the specified operation code has been defined earlier in the program as a macro, in the operation code of a library macro, or defined in an OPSYN instruction as equivalent to another operation code.

Macros that are defined inline can use any ordinary symbol, up to 63 characters in length, for the operation field. However, operating system rules might prevent some of these macros from being stored as member names in a library.

The assembler requires that the library member name and macro name are the same; otherwise error diagnostic message `ASMA126S Library macro name incorrect` is issued.

symbolic_parameter
The symbolic parameters are used in the macro definition to represent the operands of the corresponding macro instruction. A description of symbolic parameters appears under "Symbolic parameters" on page 222.

The operand field in a prototype statement lets you specify positional or keyword parameters. These parameters represent the values you can pass from the calling macro instruction to the statements within the body of a macro definition.

The operand field of the macro prototype statement must contain 0 to 32000 symbolic parameters separated by commas. They can be positional parameters or keyword parameters, or both.

If no parameters are specified in the operand field and if the absence of the operand entry is indicated by a comma preceded and followed by one or more spaces, remarks are allowed.

Here is an example of a prototype statement:

```
&NAME    MOVE           &TO,&FROM
```

Alternative formats for the prototype statement

The prototype statement can be specified in one of the following three ways:

- The normal way, with all the symbolic parameters preceding any remarks
- An alternative way, allowing remarks for each parameter
- A combination of the first two ways

The continuation rules for macro instructions are different from those for machine or assembler instruction statements. This difference is important for those who write macros that override a machine/assembler mnemonic.

The following examples show the normal statement format (&NAME1), the alternative statement format (&NAME2), and a combination of both statement formats (&NAME3):

```
       Opera-
Name   tion   Operand                        Comment                Cont.

&NAME1 OP1    &OPERAND1,&OPERAND2,&OPERAND3 This is the normal        X
                                             statement format

&NAME2 OP2    &OPERAND1,                     This is the alter-       X
              &OPERAND2                      native statement format

&NAME3 OP3    &OPERAND1,                     This is a combination    X
              &OPERAND2,&OPERAND3,           of both                  X
              &OPERAND4
```

Notes:

1. Any number of continuation lines is allowed. However, each continuation line must be indicated by a non-space character in the column after the end column on the preceding line.
2. For each continuation line, the operand field entries (symbolic parameters) must begin in the continue column; otherwise, the whole line and any lines that follow are considered to contain remarks.

   No error diagnostic message is issued to indicate that operands are treated as remarks in this situation. However, the FLAG(CONT) assembler option can be specified so that the assembler issues warning messages if it suspects an error in a continuation line.
3. The standard value for the continue column is 16 and the standard value for the end column is 71.
4. A comma is required after each parameter except the last. If you code excess commas between parameters, they are considered null positional parameters. No error diagnostic message is issued.
5. One or more spaces is required between the operand and the remarks.
6. If the DBCS assembler option is specified, the continuation features outlined in “Continuation of double-byte data” on page 13 apply to continuation in the macro language. Extended continuation is useful if a macro keyword parameter contains double-byte data.

Body of a macro definition

The body of a macro definition contains the sequence of statements that constitutes the working part of a macro. You can specify:

- Model statements to be generated
- Processing statements that, for example, can alter the content and sequence of the statements generated or issue error messages
- Comment statements, some that are generated and others that are not
- Conditional assembly instructions to compute results to be displayed in the message created by the MNOTE instruction, without causing any assembler language statements to be generated

The statements in the body of a macro definition must appear between the macro prototype statement and the MEND statement of the definition. The body of a macro definition can be empty, that is, contain no statements.

Nesting Macros: You can include macro definitions in the body of a macro definition.

Model statements

Model statements are statements from which assembler language statements are generated during conditional assembly. They let you determine the form of the statements to be generated. By specifying variable symbols as points of substitution in a model statement, you can vary the contents of the statements generated from that model statement. You can also substitute values into model statements in open code.

A model statement consists of one or more fields, separated by one or more spaces, in columns 1 to 71. The fields are called the name, operation, operand, and remarks fields.

Each field or subfield can consist of:

- An ordinary character string composed of alphanumeric and special characters
- A variable symbol as a point of substitution, except in remarks fields and comment statements
- Any combination of ordinary character strings and variable symbols to form a concatenated string

The statements generated from model statements during conditional assembly must be valid machine or assembler instructions, but must not be conditional assembly instructions. They must follow the coding rules described in "Rules for model statement fields" on page 220 or they are flagged as errors at assembly time.

Examples:

```
LABEL    L              3,AREA
LABEL2   L              3,20(4,5)
&LABEL   L              3,&AREA
FIELD&A  L              3,AREA&C
```

Variable symbols as points of substitution

Values can be substituted for variable symbols that appear in the name, operation, and operand fields of model statements; thus, variable symbols represent points of substitution. The three main types of variable symbol are:

- Symbolic parameters (positional or keyword)
- System variable symbols (see "System variable symbols" on page 229)
- SET symbols (global-scope or local-scope SETA, SETB, or SETC symbols)

Examples:

```
&PARAM(3)
&SYSLIST(1,3)
&SYSLIST(2)
&SETA(10)
&SETC(15)
```

Symbols That Can Be Subscripted: Symbolic parameters, SET symbols, and the system variable symbols &SYSLIST and &SYSMAC, can all be subscripted. All remaining system variable symbols contain only one value.

Listing of generated fields

The different fields in a macro-generated statement or a statement generated in open code appear in the listing in the same column as they are coded in the model statement, with the following exceptions:

- If the substituted value in the name or operation field is too large for the space available, the next field is moved to the right with one space separating the fields.
- If the substituted value in the operand field causes the remarks field to be displaced, the remarks field is written on the next line, starting in the column where it is coded in the model statement.
- If the substituted value in the operation field of a macro-generated statement contains leading spaces, the spaces are ignored.
- If the substituted value in the operation field of a model statement in open code contains leading spaces, the spaces are used to move the field to the right.
- If the substituted value in the operand field contains leading spaces, the spaces are used to move the field to the right.
- If the substituted value contains trailing spaces, the spaces are ignored.

Listing of generated fields containing double-byte data

If the DBCS assembler option is specified, then the following differences apply:

- Any continuation indicators present in the model statement are discarded.
- Double-byte data that must be split at a continuation point is always readable on a device capable of presenting DBCS characters—SI and SO are inserted at the break point, and the break-point always occurs between double-byte characters.
- The continuation indicator is extended to the left, if necessary, to fill space that cannot be filled with double-byte data because of alignment and delimiter considerations. The maximum number of columns filled is 3.
- If continuation is required and the character to the left of the continuation indicator is `X`, then `+` is used as the continuation indicator so as to clearly distinguish the position of the end column. This applies to any generated field, regardless of its contents, to prevent ambiguity.
- Redundant SI/SO pairs can be present in a field after substitution. If they occur at a continuation point, the assembler does not distinguish them from SI and SO inserted in the listing by the assembler to preserve readability. Refer to the generated object code to resolve this ambiguity. For more information, see Table 37 on page 219.

Rules for concatenation

If a symbolic parameter in a model statement is immediately preceded or followed by other characters or another symbolic parameter, the characters that correspond to the symbolic parameter are combined in the generated statement with the other characters, or with the characters that correspond to the other symbolic parameter. This process is called *concatenation*.

When variable symbols are concatenated to ordinary character strings, the following rules apply to the use of the concatenation character (a period). The concatenation character is mandatory when:

1 An alphanumeric character follows a variable symbol.

2 A left parenthesis that does not enclose a subscript follows a variable symbol.

3 — **4**
A period (.) is to be generated. Two periods must be specified in the concatenated string following a variable symbol.

The concatenation character is not required when:

5 An ordinary character string precedes a variable symbol.

6 A special character, except a left parenthesis or a period, is to follow a variable symbol.

7 A variable symbol follows another variable symbol.

8 A variable symbol is used with a subscript. The concatenation character must not be used between a variable symbol and its subscript; otherwise, the characters are considered a concatenated string and not a subscripted variable symbol.

Table 37 gives the rules for concatenating variable symbols to ordinary character strings. The highlighted numbers correspond to the numbers in the mandatory and not required lists.

Table 37. Rules for concatenation

Concatenated String	Values to be Substituted		Generated Result
	Variable Symbol	Value	
`&FIELD.A` **1** `&FIELDA`	`&FIELD&FIELDA`	`AREASUM`	`AREAASUM`
`&DISP.(&BASE)`[1] ↑ ↑ **2** **6**	`&DISP&BASE`	`10010`	`100(10)`
`DC D'&INT..&FRACT'`[1] ↑ **3**	`&INT&FRACT`	`9988`	`DC D'99.88'` ↑ **4**
`DC D'&INT&FRACT'` ↑ **7**			`DC D'9988'`
`FIELD&A` **5**	`&A`	`A`	`FIELDA`
`&A+&B*3-D` ↑ ↑ └─┘ **6**	`&A&B`	`AB`	`A+B*3-D`
`&SYM(&SUBSCR)` ↑ **8**	`&SUBSCR&SYM(10)`	`10ENTRY`	`ENTRY`

Notes:

1. The concatenation character is not generated.

Concatenation of fields containing double-byte data

If the DBCS assembler option is specified, then the following additional rules apply:

- Because ampersand is not recognized in double-byte data, variable symbols must not be present in double-byte data.
- The concatenation character is mandatory when double-byte data is to follow a variable symbol.
- The assembler checks for redundant SI and SO at concatenation points. If the byte to the left of the join is SI and the byte to the right of the join is SO, then the SI/SO pair is considered redundant and is removed.

Note: The rules for redundant SI and SO are different for variable substitution and listing display, which are described at "Listing of generated fields containing double-byte data" on page 218.

The following example shows these rules:

```
&SYMBOL  SETC            '<DcDd>'
DBCS     DC              C'<DaDb>&SYMBOL.<.&.S.Y.M.B.O.L>'
```

The SI/SO pairs between double-byte characters Db and Dc, and Dd and .&, are removed. The variable symbol &SYMBOL is recognized *between* the double-byte strings but not *in* the double-byte strings. The result after concatenation is:

```
DBCS     DC              C'<DaDbDcDd.&.S.Y.M.B.O.L>'
```

Rules for model statement fields

The fields that can be specified in model statements are the same fields that can be specified in an ordinary assembler language statement. They are the name, operation, operand, and remarks fields. You can also specify a continuation-indicator field, an identification-sequence field, and, in source macro definitions, a field before the begin column if the correct ICTL instruction has been specified. Character strings in the last three fields (in the standard format only, columns 72 through 80) are generated exactly as they appear in the model statement, and no values are substituted for variable symbols.

Model statements must have an entry in the operation field, and, in most cases, an entry in the operand field in order to generate valid assembler language instructions.

Name field

The entries allowed in the name field of a model statement, before generation, are:

- Space
- An ordinary symbol
- A sequence symbol
- A variable symbol
- Any combination of variable symbols, or system variable symbols such as &SYSNDX, and other character strings concatenated together

The generated result must be spaces (if valid) or a character string that represents a valid assembler or machine instruction name field. Double-byte data is not valid in an assembler or machine instruction name field and must not be generated.

Variable symbols must not be used to generate comment statement indicators (* or .*).

Notes:

1. You cannot reference an ordinary symbol defined in the name field of a model statement until the macro definition containing the model statement has been called, and the model statement has been generated.
2. Restrictions on the name entry of assembler language instructions are further specified where each individual assembler language instruction is described in this manual.

Operation field

The entries allowed in the operation field of a model statement, before generation, are given in the following list:

- An ordinary symbol that represents the operation code for:
  - Any machine instruction
  - A macro instruction
  - MNOTE instruction
  - A variable symbol
  - A combination of variable strings concatenated together
  - All assembler instructions, except ICTL and conditional assembly instructions

The following rules apply to the operation field of a model statement:

- Operation code ICTL is not allowed inside a macro definition.
- The MACRO and MEND statements are not allowed in model statements; they are used only for delimiting macro definitions.
- If the REPRO operation code is specified in a model statement, no substitution is done for the variable symbols in the record following the REPRO statement.
- Variable symbols can be used alone or as part of a concatenated string to generate operation codes for:
  - Any machine instruction
  - Any assembler instruction, except COPY, ICTL, ISEQ, REPRO, and MEXIT

  The generated operation code must not be an operation code for these instructions, either directly, or from copies created by use of OPSYN:
  - A conditional assembly instruction, such as LCLx, GBLx, SETx, AIF, and AGO
  - The following assembler instructions: COPY, ICTL, ISEQ, MACRO, MEND, MEXIT, and REPRO
- Double-byte data is not valid in the operation field.

Operand field

The entries allowed in the operand field of a model statement, before generation, are:

- Spaces (if valid)
- An ordinary symbol
- A character string, combining alphanumeric and special characters (but not variable symbols)
- A variable symbol
- A combination of variable symbols and other character strings concatenated together
- If the DBCS assembler option is specified, character strings that are enclosed in apostrophes can contain double-byte data.

The allowable results of generation are spaces (if valid) and a character string that represents a valid assembler, machine instruction, or macro instruction operand field.

Variable symbols: Variable symbols must not be used in the operand field of an ICTL or ISEQ instruction. A variable symbol must not be used in the operand field of a COPY instruction that is inside a macro definition.

Remarks field

The remarks field of a model statement can contain any combination of characters. No substitution is done for variable symbols appearing in the remarks field.

Using spaces

One or more spaces are used to separate the fields in a model statement from each other. Spaces cannot be generated inside a field in order to delimit another field. However, spaces in a combined operand-remarks field can be generated to separate these two fields. Note that if the generated operand field is part of a macro instruction, the entire string (including spaces) is passed as an operand.

```
MACRO
&PARMTAG PARMCMD
&PARMOPC SETC 'LA 1,=C''PARAMETER HEADER>>>'' PARAMETER HEADER' .* GENERATE LA INSTRUCTION USING R1
&PARMTAG &PARMOPC MEND
```

Executing this macro would generate the following:

```
PARMLIST PARMCMD
+PARMLIST LA 1,=C'PARAMETER HEADER>>>' PARAMETER HEADER
```

Also notice how this correct example contains a remark field encoded within the operand as permitted by the rules governing the use of spaces.

Both examples also supply a number of spaces in the operand field in a character literal string. This is valid since a literal string does not cross field boundaries.

Symbolic parameters

Symbolic parameters let you receive values into the body of a macro definition from the calling macro instruction. You declare these parameters in the macro prototype statement. They can serve as points of substitution in the body of the macro definition and are replaced by the values assigned to them by the calling macro instruction.

By using symbolic parameters with meaningful names, you can indicate the purpose for which the parameters (or substituted values) are used.

Symbolic parameters must be valid variable symbols. A symbolic parameter consists of an ampersand followed by an alphabetic character and from 0 to 61 alphanumeric characters.

Here are valid symbolic parameters:

```
&READER    &LOOP2
&A23456    &N
&X4F2      &$4
```

Here are invalid symbolic parameters:

```
CARDAREA   first character is not an ampersand
&256B      first character after ampersand is not alphabetic
&BCD%34    contains a special character other than initial ampersand
&IN AREA   contains a special character [space] other than initial ampersand
```

Symbolic parameters have a local scope; that is, the name and value they are assigned only applies to the macro definition in which they have been declared.

The value of the parameter remains constant throughout the processing of the containing macro definition during each call of that definition.

Notes:

1. Symbolic parameters must not have multiple definitions or be identical to any other variable symbols within the given local scope. This applies to the system variable symbols described in "System variable symbols" on page 229, and to local-scope and global-scope SET symbols described in "SET symbols" on page 279.
2. Do not begin symbolic parameters with `&SYS`, these characters are used for system variable symbols provided with High Level Assembler.

The two kinds of symbolic parameters are:
- Positional parameters
- Keyword parameters

Each positional or keyword parameter used in the body of a macro definition must be declared in the prototype statement.

Here is an example of a macro definition with symbolic parameters.

```
         MACRO                              Header
&NAME    MOVE           &TO,&FROM           Prototype
&NAME    ST             2,SAVE              Model
         L              2,&FROM             Model
         ST             2,&TO               Model
         L              2,SAVE              Model
         MEND                               Trailer
```

Here is a macro instruction that calls this macro. The characters HERE, FIELDA, and FIELDB of the MOVE macro instruction correspond to the symbolic parameters &NAME, &TO, and &FROM, of the MOVE prototype statement.

```
HERE     MOVE            FIELDA,FIELDB
```

If the macro instruction is used in a source program, these assembler language statements are generated:

```
HERE     ST              2,SAVE
         L               2,FIELDB
         ST              2,FIELDA
         L               2,SAVE
```

Positional parameters

Use a positional parameter in a macro definition if you want to change the value of the parameter each time you call the macro definition. This is because it is easier to supply the value for a positional parameter than for a keyword parameter. You only have to write the value you want the corresponding argument to have in the correct position in the operand of the calling macro instruction. However, if you need a many parameters, use keyword parameters. The keywords make it easier to keep track of the individual values you must specify at each call by reminding you which parameters are being given values.

See “Positional operands” on page 262 for details of how to write macro definitions with positional parameters.

Keyword parameters

Use a keyword parameter in a macro definition for a value that changes infrequently, or if you have many parameters. The keyword, repeated in the operand, reminds you which parameter is being given a value and for which purpose the parameter is being used. By specifying a standard default value to be assigned to the keyword parameter, you can omit the corresponding keyword argument operand in the calling macro instruction. You can specify the corresponding keyword operands in any order in the calling macro instruction.

See “Keyword operands” on page 263 for details of how to write macro definitions with keyword parameters.

Combining positional and keyword parameters

By using positional and keyword parameters in a prototype statement, you combine the benefits of both. You can use positional parameters in a macro definition for passing values that change frequently, and keyword parameters for passing values that do not change often.

Positional and keyword parameters can be mixed freely in the macro prototype statement.

See “Combining positional and keyword operands” on page 265 for details of how to write macro definitions using combined positional and keyword parameters.

Subscripted symbolic parameters

Subscripted symbolic parameters must be coded in the format:

```
&PARAM(subscript)
```

where &PARAM is a variable symbol and *subscript* is an arithmetic expression. The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (arithmetic expressions are discussed in “SETA instruction” on page 308). The arithmetic expression can contain subscripted variable symbols. Subscripts can be nested to any level if the total length of an individual operand does not exceed 1024 characters.

The value of the subscript must be greater than or equal to one. The subscript indicates the position of the entry in the sublist that is specified as the value of the subscripted parameter (sublists as values in macro instruction operands are fully described in "Sublists in operands" on page 266).

Processing statements

This section provides information about these processing statements:

- "Conditional assembly instructions"
- "Inner macro instructions"
- "Other conditional assembly instructions"
- "AEJECT instruction" on page 225
- "AINSERT instruction" on page 225
- "AREAD instruction" on page 225
- "ASPACE instruction" on page 227
- "COPY instruction" on page 228
- "MEXIT instruction" on page 228

Conditional assembly instructions

Conditional assembly instructions let you determine at conditional assembly time the content of the generated statements and the sequence in which they are generated. Here are the instructions and their functions:

Conditional Assembly	Operation Done
GBLA, GBLB, GBLCLCLA, LCLB, LCLC	Declaration of variable symbols (global-scope and local-scope SET symbols) and setting of default initial values
SETA, SETB, SETC	Assignment of values to variable symbols (SET symbols)
SETAF, SETCF	External function assignment of values to variable symbols (SET symbols)
ACTR	Setting loop counter
AGO	Unconditional branch
AIF	Conditional branch (based on logical test)
ANOP	Pass control to next sequential instruction (no operation)

Conditional assembly instructions can be used both inside macro definitions and in open code. They are described in Chapter 9, "How to write conditional assembly instructions," on page 279.

Inner macro instructions

Macro instructions can be nested inside macro definitions, allowing you to call other macros from within your own definition.

Other conditional assembly instructions

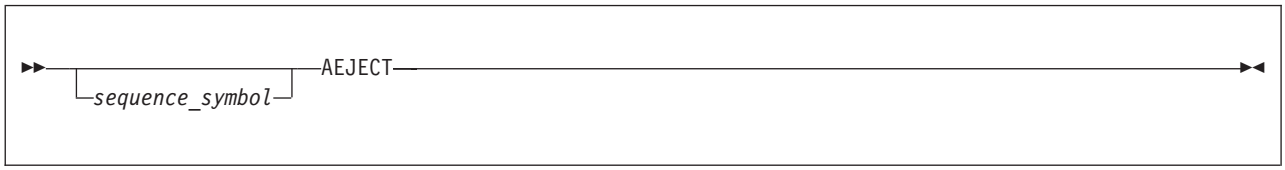
Several additional instructions can help you write your macro definitions. Here are the instructions and their functions:

Inner Macro Instruction	Operation Done
AEJECT	Skip to next page
AINSERT	Insert statement into input stream

Inner Macro Instruction	Operation Done
AREAD	Assign an arbitrary character string to a variable symbol (SETC symbol)
ASPACE	Insert one or more blank lines in listing
COPY	Copy the source statements from a source language library member.
MEXIT	Exit from the macro definition

AEJECT instruction

Use the AEJECT instruction to stop the printing of the assembler listing of your macro definition on the current page, and continue the printing on the next page.



sequence_symbol
Is a sequence symbol.

The AEJECT instruction causes the next line of the assembly listing of your macro definition to be printed at the top of a new page. If the line before the AEJECT statement appears at the bottom of a page, the AEJECT statement has no effect. An AEJECT instruction immediately following another AEJECT instruction causes a blank page in the listing of the macro definition.

Notes:

1. The AEJECT instruction can only be used inside a macro definition.
2. The AEJECT instruction itself is not printed in the listing.
3. The AEJECT instruction does not affect the listing of statements generated when the macro is called.

AINSERT instruction

The AINSERT instruction, inside macro definitions, harnesses the power of macros to generate source statements, for instance, using variable substitution. Generated statements are queued in a special buffer and read after the macro generator finishes.

The specifications for the AINSERT instruction, which can also be used in open code, are described in "AINSERT instruction" on page 92.

AREAD instruction

The AREAD instruction assigns an arbitrary character string value to a SETC symbol.

The AREAD instruction has two formats. The first format lets you assign to a SETC symbol the character string value of a statement that is placed immediately after a macro instruction.

The AREAD instruction can only be used inside macro definitions.

Assign character string value

The second format of the AREAD instruction assigns to a SETC symbol a character string containing the local time.

Assign local time

SETC_symbol
Is a SETC symbol. See "SETC instruction" on page 326.

NOSTMT
Specifies that the statement to be read by the AREAD instruction is printed in the assembly listing, but not given any statement number.

NOPRINT
specifies that the statement does not appear in the listing, and no statement number is assigned to it.

CLOCKB
Assigns an 8-character string to *SETC_symbol* containing the local time in hundredths of a second since midnight.

CLOCKD
Assigns an 8-character string to *SETC_symbol* containing the local time in the format *HHMMSSTH*, where *HH* is a value 00 - 23, *MM* and *SS* each have a value 00 - 59, and *TH* has a value 00 - 99.

Assign character string value

The first format of AREAD functions in much the same way as symbolic parameters, but instead of providing your input to macro processing as part of the macro instruction, you can supply full input records from either the AINSERT buffer (if any are present), or from the records in the primary input stream that follow immediately after the macro instruction. Any number of successive records can be read into the macro for processing. If no records remain, a null value is assigned, but no diagnostic message is issued.

SETC_symbol can be subscripted. When the assembler encounters a Format-1 AREAD statement during the processing of a macro instruction, it reads the source record following the macro instruction and assigns an 80-character string to the SETC symbol in the name field. For nested macros, it reads the record following the outermost macro instruction.

If no operand is specified, the record to be read by AREAD is printed in the listing and assigned a statement number. The AREAD action is indicated in the listing by a minus sign between the statement number and the first character of the record.

Repeated AREAD instruction statements read successive records. In the following example, the input record starting with INRECORD1 is read by the first AREAD statement, and assigned to the SETC symbol &VAL. The input record starting with INRECORD2 is read by the second AREAD statement, and assigned to the SETC symbol &VAL1.

Example:

```
         MACRO
         MAC1
         .
&VAL     AREAD
         .
&VAL1    AREAD
         .
         MEND
         CSECT
         .
         MAC1
INRECORD1 THIS IS THE STATEMENT TO BE PROCESSED FIRST
INRECORD2 THIS IS THE NEXT STATEMENT
         .
         END
```

The records read by the AREAD instruction can be in code brought in with the COPY instruction, if the macro instruction employing the AREAD appears in the COPY member. Otherwise the COPY instruction may be read as an ordinary text line.

If no more records exist in the code brought in by the COPY instruction, subsequent records are read from the AINSERT buffer or the primary input stream.

Assign local time of day

The second format of AREAD functions in much the same way as a SETC instruction, but instead of supplying the value you want assigned to the SETC symbol as a character string in the operand of the AREAD instruction, the value is provided by the operating system in the form of an 8-character string containing the local time. A Format-2 AREAD instruction does not cause the assembler to read the statement following the macro instruction.

Example:

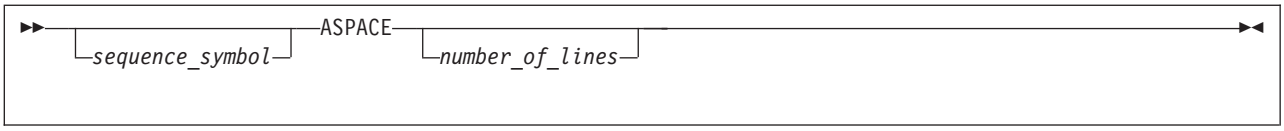
```
         MACRO
         MAC2
         .
&VAL     AREAD CLOCKB
         DC    C'&VAL'
&VAL1    AREAD CLOCKD
         DC    C'&VAL1'
         .
         MEND
```

When this macro definition is called, these statements are generated:

```
          MAC2
+         DC    C'03251400'
+         DC    C'09015400'
```

ASPACE instruction

Use the ASPACE instruction to insert one or more blank lines in the listing of a macro definition in your source module, thus separating sections of macro definition code on the listing page.



sequence_symbol
Is a sequence symbol.

number_of_lines
Is a non-negative decimal integer that specifies the number of lines to be left blank. If *number_of_lines* is omitted, one line is left blank. If *number_of_lines* has a value greater than the number of lines remaining on the listing page, the instruction has the same effect as an AEJECT statement.

Notes:

1. The ASPACE instruction can only be used inside a macro definition.
2. The ASPACE instruction itself is not printed in the listing.
3. The ASPACE instruction does not affect the listing of statements generated when the macro is called.

COPY instruction

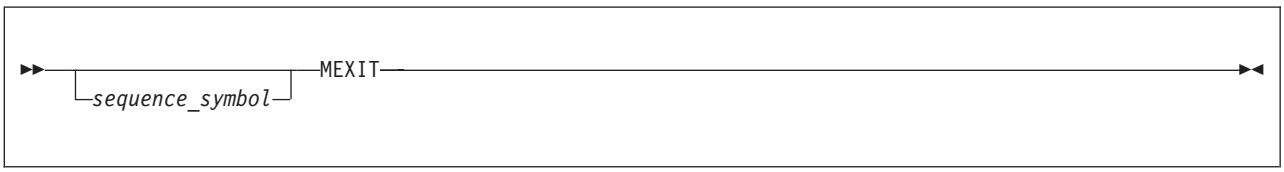
The COPY instruction, inside macro definitions, lets you copy into the macro definition any sequence of statements allowed in the body of a macro definition. These statements become part of the body of the macro before macro processing takes place. You can also use the COPY instruction to copy complete macro definitions into a source module.

The specifications for the COPY instruction, which can also be used in open code, are described in "COPY instruction" on page 105.

MEXIT instruction

The MEXIT instruction provides an exit for the assembler from any point in the body of a macro definition. The MEND instruction provides an exit only from the end of a macro definition (see "MEND statement" on page 214 for details).

The MEXIT instruction statement can be used only inside macro definitions.



sequence_symbol
Is a sequence symbol.

The MEXIT instruction causes the assembler to exit from a macro definition to the next sequential instruction after the macro instruction that calls the definition. (This also applies to nested macro instructions, which are described in "Nesting macro instruction definitions" on page 272.)

For example, the following macro definition contains an MEXIT statement:

```
MACRO
EXITS
DC    C'A'
DC    C'B'
DC    C'C'
MEXIT
```

```
         DC    C'D'
         DC    C'E'
         DC    C'F'
         MEND
```

When this macro definition is called, these statements are generated:

```
          EXITS
+         DC    C'A'
+         DC    C'B'
+         DC    C'C'
```

Comment statements

Two types of comment statements can be used within a macro definition:
- Ordinary comment statements
- Internal macro comment statements

Ordinary comment statements

Ordinary comment statements let you make descriptive remarks about the generated output from a macro definition. Ordinary comment statements can be used in macro definitions and in open code.

An ordinary comment statement consists of an asterisk in the begin column, followed by any character string. The comment statement is used by the assembler to generate an assembler language comment statement, just as other model statements are used by the assembler to generate assembler statements. No variable symbol substitution is done.

Internal macro comment statements

You can also write internal macro comments in the body of a macro definition to describe the operations done during conditional assembly when the macro is processed.

An internal macro comment statement consists of a period in the begin column, followed by an asterisk, followed by any character string. No values are substituted for any variable symbols that are specified in internal macro comment statements.

Internal macro comment statements can appear anywhere in a macro definition.

Notes:
1. Internal macro comments are not generated.
2. The comment character string can contain double-byte data.
3. Internal macro comment statements can be used in open code, however, they are processed as *ordinary* comment statements.

System variable symbols

System variable symbols are a special class of variable symbols, starting with the characters `&SYS`. Their values are set by the assembler according to specific rules. You cannot declare them in local-scope SET symbols or global-scope SET symbols, nor use them as symbolic parameters in macro prototype statements. You can use these symbols as points of substitution in model statements and conditional assembly instructions.

All system variable symbols are subject to the same rules of concatenation and substitution as other variable symbols.

A description of each system variable symbols begins with "&SYSADATA_DSN System Variable Symbol" on page 231.

Do not prefix your SET symbols with the character sequence &SYS. The assembler uses this sequence as a prefix to all system variable symbol names, and using them for other SET symbol names might cause future conflicts.

Scope and variability of system variable symbols

Global Scope

Some system variable symbols have values that are established at the beginning of an assembly and are available both in open code and from within macros. These symbols have global scope. Most system variable symbols with global scope have fixed values, although there are some whose value can change within a single macro expansion. The global-scope system variables symbols with variable values are &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV.

Local Scope

Some system variable symbols have values that are available only from within a macro expansion. These system variables have local scope. Since the value of system variable symbols with local scope is established at the beginning of a macro expansion and remains unchanged throughout the expansion, they are designated as having constant values, even though they might have different values in a later expansion of the same macro, or within inner macros.

Over half of the system variable symbols have local scope and therefore are not available in open code.

```
                                    1          macro
                                    2          getlocalsys
                                    3 .*       Define globals for values of interest
                                    4          Gblc  &clock,&location,&dsname,&nest
                                    5          Gbla  &nesta
                                    6 .*       now update the globals from within the macro
                                    7 &clock    setc '&sysclock'
                                    8 &location setc '&sysloc'
                                    9 &dsname  setc  '&sysin_dsn'
                                   10 &nest    setc  '&sysnest'
                                   11 &nesta   seta  &sysnest
                                   12          mend
000000                00000 00020  14 r        csect
                                   15 *
                                   16 *        define globals in opencode
                                   17 *
                                   18          Gblc  &clock,&location,&dsname,&nest
                                   19          Gbla  &nesta
                                   20 *
                                   21 *        invoke macro to update the global values
                                   22 *
                                   23          getlocalsys
                                   24 *
                                   25 *        now use the updated values
                                   26 *
                                   27          dc    c'&clock'
000000 F2F0F0F460F0F660              +         dc    c'2008-07-11 17:48:42.914829'
                                   28          dc    c'&nest'
00001A F1                            +         dc    c'1'
                                   29          dc    f'&nesta'
00001B 00
00001C 00000001                      +         dc    f'1'
000000                             31          end r
```

Figure 31. Exposing the value of a local scope variable to open code

Uses, values, and properties

System variable symbols have many uses, including:

- Helping to control conditional assemblies
- Capturing environmental data for inclusion in the generated object code
- Providing program debugging data

Refer to Appendix C, "Macro and conditional assembly language summary," on page 363 for a summary of the values and properties that can be assigned to system variable symbols.

&SYSADATA_DSN System Variable Symbol

Use &SYSADATA_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the associated data.

The local-scope system variable symbol &SYSADATA_DSN is assigned a read-only value each time a macro definition is called.

z/OS

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSADATA_DSN is always the value stored in the JFCB for SYSADATA. If SYSADATA is allocated to DUMMY, or a NULLFILE, the value in &SYSADATA_DSN is `NULLFILE`.

For example, &SYSADATA_DSN might be assigned a value such as:

```
IBMAPC.SYSADATA
```

z/VM

When the assembler runs on the CMS component of the z/VM operating systems, the value of the character string assigned to &SYSADATA_DSN is determined as follows:

Table 38. Contents of &SYSADATA_DSN on CMS

SYSADATA Allocated To:	Contents of &SYSADATA_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Dummy file (no physical I/O)	`DUMMY`
Labeled tape file	The data set name of the tape file
Unlabeled tape file	`TAP`*n*, where *n* is a value from 0 to 9, or A to F.

For example, &SYSADATA_DSN might be assigned a value such as:

```
SAMPLE   SYSADATA A1
```

z/VSE

The value of the character string assigned to &SYSADATA_DSN is the file ID from the SYSADAT dlbl.

For example, &SYSADATA_DSN might be assigned a value such as:

```
MYDATA
```

Notes:

1. The value of the type attribute of &SYSADATA_DSN (T'&SYSADATA_DSN) is always U.
2. The value of the count attribute of &SYSADATA_DSN (K'&SYSADATA_DSN) is equal to the number of characters assigned as a value to &SYSADATA_DSN. In the previous CMS example, the count attribute of &SYSADATA_DSN is 20.

&SYSADATA_MEMBER System Variable Symbol

z/VSE The value of &SYSADATA_MEMBER is always null. The value of the type attribute is O, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSADATA_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the associated data.

The local-scope system variable symbol &SYSADATA_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the associated data is not a z/OS partitioned data set, &SYSADATA_MEMBER is assigned a null character string.

Notes:

1. The value of the type attribute of &SYSADATA_MEMBER (T'&SYSADATA_MEMBER) is U, unless &SYSADATA_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
2. The value of the count attribute of &SYSADATA_MEMBER (K'&SYSADATA_MEMBER) is equal to the number of characters assigned as a value to &SYSADATA_MEMBER. If &SYSADATA_MEMBER is assigned a null character string, the value of the count attribute is 0.

&SYSADATA_VOLUME System Variable Symbol

Use &SYSADATA_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler is writing the associated data.

The local-scope system variable symbol &SYSADATA_VOLUME is assigned a read-only value each time a macro definition is called.

z/VM If the assembler runs on the CMS component of the z/VM operating system, and the associated data is being written to a Shared File System CMS file, &SYSADATA_VOLUME is assigned the value "** SFS".

If the volume on which the data set resides is not labeled, &SYSADATA_VOLUME is assigned a null character string.

Notes:

1. The value of the type attribute of &SYSADATA_VOLUME (T'&SYSADATA_VOLUME) is U, unless &SYSADATA_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
2. The value of the count attribute of &SYSADATA_VOLUME (K'&SYSADATA_VOLUME) is equal to the number of characters assigned as a value to &SYSADATA_VOLUME. If &SYSADATA_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.

&SYSASM System Variable Symbol

Use &SYSASM to obtain the name of the assembler being used to assemble your source module. &SYSASM has a global scope. For example, when IBM High Level Assembler for z/OS & z/VM & z/VSE is used, &SYSASM has the value:

```
HIGH LEVEL ASSEMBLER
```

Notes:

1. The value of the type attribute of &SYSASM (T'&SYSASM) is always U.
2. The value of the count attribute (K'&SYSASM) is the number of characters assigned. In the example, the count attribute of &SYSASM is 20.

&SYSCLOCK System Variable Symbol

Use &SYSCLOCK to obtain the TOD clock date and time at which the macro was generated, based on Universal Time (GMT).

The local-scope system variable symbol &SYSCLOCK is assigned a read-only value each time a macro definition is called.

The value of &SYSCLOCK is a 26-character string in the format:

YYYY-MM-DD HH:MM:SS.mmmmmm

where:

YYYY Is a four-digit field that gives the year, including the century. It has a value 0000 - 9999.

MM Is a two-digit field that gives the month of the year. It has a value 01 - 12.

DD Is a two-digit field that gives the day of the month. It has a value 01 - 31.

HH Is a two-digit field that gives the hour of the day. It has a value 00 - 23.

MM Is a two-digit field that gives the minute of the hour. It has a value 00 - 59.

SS Is a two-digit field that gives the second of the minute. It has a value 00 - 59.

mmmmmm
Is a six-digit field that gives the microseconds within the seconds. It has a value 000000 - 999999.

Example:

```
2001-06-08 17:36:03 043284
```

Notes:

1. The value of the type attribute of &SYSCLOCK (T'&SYSCLOCK) is always U.
2. The value of the count attribute (K'&SYSCLOCK) is always 26.

&SYSDATC System Variable Symbol

Use &SYSDATC to obtain the date, including the century, on which your source module is assembled. &SYSDATC has a global scope.

The value of &SYSDATC is an 8-character string in the format:

YYYYMMDD

where:

YYYY Is four-digit field that gives the year, including the century. It has a value 0000 - 9999.

MM Is two-digit field that gives the month of the year. It has a value 01 - 12.

DD Is two-digit field that gives the day of the month. It has a value 01 - 31.

Example:

```
20000328
```

Notes:

1. The date corresponds to the date printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of &SYSDATC (T'&SYSDATC) is always N.
3. The value of the count attribute (K'&SYSDATC) is always 8.

&SYSDATE System Variable Symbol

Use &SYSDATE to obtain the date, in standard format, on which your source module is assembled. &SYSDATE has a global scope.

The value of &SYSDATE is an 8-character string in the format:

MM/DD/YY

where:

MM Is a two-digit field that gives the month of the year. It has a value 01 - 12.

DD Is a two-digit field that gives the day of the month. It has a value 01 - 31. It is separated from MM by a slash.

YY Is a two-digit field that gives the year of the century. It has a value 00 - 99. It is separated from DD by a slash.

Example:

```
07/11/08
```

Notes:

1. The date corresponds to the date printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of &SYSDATE (T'&SYSDATE) is always U.
3. The value of the count attribute (K'&SYSDATE) is always 8.

&SYSECT System Variable Symbol

Use &SYSECT in a macro definition to generate the name of the current control section. The current control section is the control section in which the macro instruction that calls the definition appears. You cannot use &SYSECT in open code.

The local-scope system variable symbol &SYSECT is assigned a read-only value each time a macro definition is called.

The value assigned is the symbol that represents the name of the current control section from which the macro definition is called. Note that it is the control section in effect when the macro is called. A control section that has been initiated or continued by substitution does not affect the value of &SYSECT for the expansion of the current macro. However, it might affect &SYSECT for a subsequent macro call. Nested macros cause the assembler to assign a value to &SYSECT that depends on the control section in force inside the outer macro when the inner macro is called.

Notes:

1. The control section whose name is assigned to &SYSECT can be defined by a program sectioning statement. This can be a START, CSECT, RSECT, DSECT, or COM statement.
2. The value of the type attribute of &SYSECT (T'&SYSECT) is always U.
3. The value of the count attribute (K'&SYSECT) is equal to the number of characters assigned as a value to &SYSECT.
4. Throughout the use of a macro definition, the value of &SYSECT is considered a constant, independent of any program sectioning statements or inner macro instructions in that definition.

The next example shows these rules:

```
         MACRO
         INNER           &INCSECT
&INCSECT CSECT                           Statement 1
         DC              A(&SYSECT)      Statement 2
```

```
         MEND

         MACRO
         OUTER1
CSOUT1   CSECT                           Statement 3
         DS              100C
         INNER           INA             Statement 4
         INNER           INB             Statement 5
         DC              A(&SYSECT)      Statement 6
         MEND

         MACRO
         OUTER2
         DC              A(&SYSECT)      Statement 7
         MEND
---------------------------------------------------------------------
MAINPROG CSECT                           Statement 8
         DS              200C
         OUTER1                          Statement 9
         OUTER2                          Statement 10
---------------------------------------------------------------------
Generated Program
---------------------------------------------------------------------
MAINPROG CSECT
         DS              200C
CSOUT1   CSECT
         DS              100C
INA      CSECT
         DC              A(CSOUT1)
INB      CSECT
         DC              A(INA)
         DC              A(MAINPROG)
         DC              A(INB)
```

In this example:

- Statement 8 is the last program sectioning statement processed before statement 9 is processed. Therefore, &SYSECT is assigned the value `MAINPROG` for macro instruction `OUTER1` in statement 9. `MAINPROG` is substituted for &SYSECT when it appears in statement 6.
- Statement 3 is the program sectioning statement processed before statement 4 is processed. Therefore, &SYSECT is assigned the value `CSOUT1` for macro instruction `INNER` in statement 4. `CSOUT1` is substituted for &SYSECT when it appears in statement 2.
- Statement 1 is used to generate a CSECT statement for statement 4. This is the last program sectioning statement that appears before statement 5. Therefore, &SYSECT is assigned the value `INA` for macro instruction `INNER` in statement 5. `INA` is substituted for &SYSECT when it appears in statement 2.
- Statement 1 is used to generate a CSECT statement for statement 5. This is the last program sectioning statement that appears before statement 10. Therefore, &SYSECT is assigned the value `INB` for macro instruction `OUTER2` in statement 10. `INB` is substituted for &SYSECT when it appears in statement 7.

&SYSIN_DSN System Variable Symbol

Use &SYSIN_DSN in a macro definition to obtain the name of the data set from which the assembler is reading the source module.

The local system variable symbol &SYSIN_DSN is assigned a read-only value each time a macro definition is called.

z/OS If concatenated data sets are used to provide the source module, &SYSIN_DSN has a value equal to the data set name of the data set that contains the open code source line of the macro call statement, irrespective of the nesting depth of the macro line containing the &SYSIN_DSN reference.

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSIN_DSN is always the value stored in the JFCB for SYSIN.

z/VM When the assembler runs on the CMS component of the z/VM operating systems, the value of the character string assigned to &SYSIN_DSN is determined as follows:

Table 39. Contents of &SYSIN_DSN on CMS

SYSIN Allocated To:	Contents of &SYSIN_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Reader	`READER`
Terminal	`TERMINAL`
Labeled tape file	The data set name of the tape file
Unlabeled tape file	`TAP`*n*, where *n* is a value from 0 to 9, or A to F.

z/VSE When the assembler runs on the z/VSE operating system, the value of the character string assigned to &SYSIN_DSN is determined as follows:

Table 40. Contents of &SYSIN_DSN on z/VSE

SYSIPT Assigned To:	Contents of &SYSIN_DSN:
Job stream (SYSIPT)	`SYSIPT`
Disk	The file-id
Labeled tape file	The file ID of the tape file
Unlabeled tape file	`SYSIPT`

Examples:

On z/OS, &SYSIN_DSN might be assigned a value such as:

```
IBMAPC.ASSEMBLE.SOURCE
```

On CMS, &SYSIN_DSN might be assigned a value such as:

```
SAMPLE   ASSEMBLE A1
```

Notes:

1. If the SOURCE user exit provides the data set information then the value in &SYSIN_DSN is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSIN_DSN (T'&SYSIN_DSN) is always U.
3. The value of the count attribute of &SYSIN_DSN (K'&SYSIN_DSN) is equal to the number of characters assigned as a value to &SYSIN_DSN. In the previous CMS example, the count attribute of &SYSIN_DSN is 20.
4. Throughout the use of a macro definition, the value of &SYSIN_DSN is considered a constant.

&SYSIN_MEMBER System Variable Symbol

z/VSE The value of &SYSIN_MEMBER is always null.

The value of the type attribute is O, and the value of the count attribute is 0.

zVM z/OS

You can use &SYSIN_MEMBER in a macro definition to obtain the name of the data set member from which the assembler is reading the source module. If concatenated data sets are used to provide the source module, &SYSIN_MEMBER has a value equal to the name of the data set member that contains the macro instruction that calls the definition.

The local-scope system variable symbol &SYSIN_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set from which the assembler is reading the source module is not a z/OS partitioned data set or a CMS MACLIB, &SYSIN_MEMBER is assigned a null character string.

Notes:

1. If the SOURCE user exit provides the data set information then the value in &SYSIN_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSIN_MEMBER (T'&SYSIN_MEMBER) is U, unless &SYSIN_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSIN_MEMBER (K'&SYSIN_MEMBER) is equal to the number of characters assigned as a value to &SYSIN_MEMBER. If &SYSIN_MEMBER is assigned a null character string, the value of the count attribute is 0.
4. Throughout the use of a macro definition, the value of &SYSIN_MEMBER is considered a constant.

&SYSIN_VOLUME System Variable Symbol

Use &SYSIN_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set from which the assembler is reading the source module.

z/VM and z/OS
If concatenated data sets are used to provide the source module, &SYSIN_VOLUME has a value equal to the volume identifier of the first volume containing the data set that contains the macro call instruction.

The local-scope system variable symbol &SYSIN_VOLUME is assigned a read-only value each time a macro definition is called.

z/VM If the assembler runs on the CMS component of the z/VM operating system, and the source module is being read from a Shared File System CMS file, &SYSIN_VOLUME is assigned the value "** SFS".

If the volume on which the input data set resides is not labeled, &SYSIN_VOLUME is assigned a null character string.

Notes:

1. If the SOURCE user exit provides the data set information then the value in &SYSIN_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSIN_VOLUME (T'&SYSIN_VOLUME) is U, unless &SYSIN_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSIN_VOLUME (K'&SYSIN_VOLUME) is equal to the number of characters assigned as a value to &SYSIN_VOLUME. If &SYSIN_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.
4. Throughout the use of a macro definition, the value of &SYSIN_VOLUME is considered a constant.

&SYSJOB System Variable Symbol

Use &SYSJOB to obtain the job name of the assembly job used to assemble your source module. &SYSJOB has a global scope.

When the assembler runs on the CMS component of the VM operating systems, &SYSJOB is assigned a value of `(NOJOB)`.

Notes:

1. The value of the type attribute of &SYSJOB (T'&SYSJOB) is always U.
2. The value of the count attribute (K'&SYSJOB) is the number of characters assigned.

&SYSLIB_DSN System Variable Symbol

Use &SYSLIB_DSN in a macro definition to obtain name of the data set from which the assembler read the macro definition statements. If the macro definition is a source macro definition, &SYSLIB_DSN is assigned the same value as &SYSIN_DSN.

The local-scope system variable symbol &SYSLIB_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSLIB_DSN is always the value stored in the JFCB for SYSLIB.

When the assembler runs on the CMS component of the VM operating systems, and the macro definition is a library macro definition, &SYSLIB_DSN is assigned the file name, file type, and file mode of the data set.

z/VSE When the macro definition is a library macro definition, &SYSLIB_DSN is assigned the library name and sublibrary name of the z/VSE Librarian file.

Examples

Under z/OS, &SYSLIB_DSN might be assigned a value such as:

```
SYS1.MACLIB
```

Under CMS, &SYSLIB_DSN might be assigned a value such as:

```
DMSGPI MACLIB S2
```

Under z/VSE, &SYSLIB_DSN might be assigned a value such as:

```
IJSYSRS.SYSLIB
```

Notes:

1. If the LIBRARY user exit provides the data set information then the value in &SYSLIB_DSN is the value extracted from the Exit-Specific Information block described in the section “Exit-Specific Information Block” in the *HLASM Programmer’s Guide*.
2. The value of the type attribute of &SYSLIB_DSN (T'&SYSLIB_DSN) is always U.
3. The value of the count attribute of &SYSLIB_DSN (K'&SYSLIB_DSN) is equal to the number of characters assigned as a value to &SYSLIB_DSN.
4. Throughout the use of a macro definition, the value of &SYSLIB_DSN is considered a constant.

&SYSLIB_MEMBER System Variable Symbol

Use &SYSLIB_MEMBER in a macro definition to obtain the name of the data set member from which the assembler read the macro definition statements. If the macro definition is a source macro definition, &SYSLIB_MEMBER is assigned the same value as &SYSIN_MEMBER.

The local-scope system variable symbol &SYSLIB_MEMBER is assigned a read-only value each time a macro definition is called.

Notes:

1. If the LIBRARY user exit provides the data set information then the value in &SYSLIB_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSLIB_MEMBER (T'&SYSLIB_MEMBER) is U, unless &SYSLIB_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSLIB_MEMBER (K'&SYSLIB_MEMBER) is equal to the number of characters assigned as a value to &SYSLIB_MEMBER. If &SYSLIB_MEMBER is assigned a null character string, the value of the count attribute is 0.
4. Throughout the use of a macro definition, the value of &SYSLIB_MEMBER is considered a constant.

&SYSLIB_VOLUME System Variable Symbol

Use &SYSLIB_VOLUME in a macro definition to obtain the volume identifier of the volume containing the data set from which the assembler read the macro definition statements. If the macro definition is a source macro definition, &SYSLIB_VOLUME is assigned the same value as &SYSIN_VOLUME.

The local-scope system variable symbol &SYSLIB_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the source module is being read from a Shared File System CMS file, &SYSLIB_VOLUME is assigned the value `** SFS`.

Notes:

1. If the LIBRARY user exit provides the data set information then the value in &SYSLIB_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSLIB_VOLUME (T'&SYSLIB_VOLUME) is U, unless &SYSLIB_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSLIB_VOLUME (K'&SYSLIB_VOLUME) is equal to the number of characters assigned as a value to &SYSLIB_VOLUME. If &SYSLIB_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.
4. Throughout the use of a macro definition, the value of &SYSLIB_VOLUME is considered a constant.

&SYSLIN_DSN System Variable Symbol

Use &SYSLIN_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the object records when assembler option OBJECT, GOFF, or XOBJECT is specified.

The local-scope system variable symbol &SYSLIN_DSN is assigned a read-only value each time a macro definition is called.

z/OS The value of the character string assigned to &SYSLIN_DSN is always the value stored in the JFCB for SYSLIN. If SYSLIN is allocated to DUMMY, or a NULLFILE, the value in &SYSLIN_DSN is "`NULLFILE`".

z/VM The value of the character string assigned to &SYSLIN_DSN is determined as follows:

Table 41. Contents of &SYSLIN_DSN on CMS

SYSLIN Allocated To:	Contents of &SYSLIN_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Dummy file (no physical I/O)	`DUMMY`
Punch	`PUNCH`
Labeled tape file	The data set name of the tape file
Unlabeled tape file	`TAP`*n*, where *n* is a value from 0 to 9, or A to F.

z/VSE

The value of the character string assigned to &SYSLIN_DSN is determined as follows:

Table 42. Contents of &SYSLIN_DSN on z/VSE

SYSLNK Assigned To:	Contents of &SYSLIN_DSN:
Disk file	The file-id
Labeled tape file	The file ID of the tape file
Unlabeled tape file	`SYSLNK`

Examples:

On z/OS, &SYSLIN_DSN might be assigned a value such as:

```
IBMAPC.OBJ
```

On CMS, &SYSLIN_DSN might be assigned a value such as:

```
SAMPLE   TEXT     A1
```

Notes:

1. If the OBJECT user exit provides the data set information then the value in &SYSLIN_DSN is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSLIN_DSN (T'&SYSLIN_DSN) is always U.
3. The value of the count attribute of &SYSLIN_DSN (K'&SYSLIN_DSN) is equal to the number of characters assigned as a value to &SYSLIN_DSN.

&SYSLIN_MEMBER System Variable Symbol

z/VSE The value of &SYSLIN_MEMBER is always null.

The value of the type attribute is O, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSLIN_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the object module when the assembler option OBJECT, GOFF, or XOBJECT is specified.

The local-scope system variable symbol &SYSLIN_MEMBER is assigned a read-only value each time a macro definition is called.

If the library to which the assembler is writing the object records is not a z/OS partitioned data set, &SYSLIN_MEMBER is assigned a null character string.

Notes:

1. If the OBJECT user exit provides the data set information then the value in &SYSLIN_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSLIN_MEMBER (T'&SYSLIN_MEMBER) is U, unless &SYSLIN_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSLIN_MEMBER (K'&SYSLIN_MEMBER) is equal to the number of characters assigned as a value to &SYSLIN_MEMBER. If &SYSLIN_MEMBER is assigned a null character string, the value of the count attribute is 0.

&SYSLIN_VOLUME System Variable Symbol

Use &SYSLIN_VOLUME in a macro definition to obtain the volume identifier of the object data set. The volume identifier is of the first volume containing the data set. &SYSLIN_VOLUME is only assigned a value when you specify the OBJECT, GOFF, or XOBJECT assembler option.

The local-scope system variable symbol &SYSLIN_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the assembler listing is being written to a Shared File System CMS file, &SYSLIN_VOLUME is assigned the value `** SFS`.

If the volume on which the data set resides is not labeled, &SYSLIN_VOLUME is assigned a null character string.

Notes:

1. If the OBJECT user exit provides the data set information then the value in &SYSLIN_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSLIN_VOLUME (T'&SYSLIN_VOLUME) is U, unless &SYSLIN_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSLIN_VOLUME (K'&SYSLIN_VOLUME) is equal to the number of characters assigned as a value to &SYSLIN_VOLUME. If &SYSLIN_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.

&SYSLIST System Variable Symbol

Use &SYSLIST instead of a positional parameter inside a macro definition; for example, as a point of substitution. By varying the subscripts attached to &SYSLIST, you can refer to any sublist entry in a macro call *operand*, or any positional operands in a macro call. You can also refer to positional operands for which no corresponding positional parameter is specified in the macro prototype statement.

The local-scope system variable symbol &SYSLIST is assigned a read-only value each time a macro definition is called.

&SYSLIST refers to the complete list of positional operands specified in a macro instruction. &SYSLIST does not refer to keyword operands. However, &SYSLIST cannot be specified as &SYSLIST without a subscript. One of the two following forms must be used for references or as a point of substitution:

1. &SYSLIST(*n*) can be used to refer to the *n*-th positional operand
2. If the *n*-th operand is a sublist, then &SYSLIST(*n*,*m*) can be used to refer to the *m*-th operand in the sublist.

3. When referring to multilevel (nested) sublists in operands of macro instructions, refer to elements of inner sublists by using the applicable number of subscripts for &SYSLIST.

The subscripts *n* and *m* can be any arithmetic expression allowed in the operand of a SETA instruction (See "SETA instruction" on page 308). The subscript *n* must be greater than or equal to 0. The subscript *m* and any additional subscripts after *m* must be greater than or equal to 1.

The examples show the values assigned to &SYSLIST according to the value of its subscripts *n* and *m*.

```
Macro instruction:
------------------
NAME     MACALL          ONE,TWO,(3,(4,5,6),,8),,TEN,()

Use Within a
Macro Definition:        Value              See note:
---------------------    ------------       ---------
&SYSLIST(2)              TWO
&SYSLIST(3,1)            3
&SYSLIST(3,2,2)          5

&SYSLIST(4)              Null                  1

&SYSLIST(12)             Null                  1
&SYSLIST(3,3)            Null                  2

&SYSLIST(3,5)            Null                  2

&SYSLIST(2,1)            TWO                   3
&SYSLIST(2,2)            Null

&SYSLIST(0)              NAME                  4
&SYSLIST(3)              (3,(4,5,6),,8)
&SYSLIST(11)             ()
&SYSLIST(11,1)           Null                  2
```

Note:

1. If the position indicated by *n* refers to an omitted operand, or refers to an entry past the end of the list of positional operands specified, the null character string is substituted for &SYSLIST(*n*).
2. If the position (in a sublist) indicated by the second subscript, *m*, refers to an omitted entry, or refers past the end of the list of entries specified in the sublist referred to by the first subscript *n*, the null character string is substituted for &SYSLIST(*n*,*m*).
3. If the *n*-th positional operand is not a sublist, &SYSLIST(*n*,1) refers to the operand. However, &SYSLIST(*n*,*m*), where *m* is greater than 1, causes the null character string to be substituted.
4. If the value of subscript *n* is 0, then &SYSLIST(*n*) is assigned the value specified in the name field of the macro instruction, except when it is a sequence symbol.

Attribute references can be made to the previously described forms of &SYSLIST. The attributes are the attributes inherent in the positional operands or sublist entries to which you refer. However, the number attribute of &SYSLIST (N'&SYSLIST) is different from the number attribute described in "Data attributes" on page 284. One of two forms can be used for the number attribute:

- To indicate the number of positional operands specified in a call, use the form N'&SYSLIST.
- To indicate the number of sublist entries that have been specified in a positional operand, use the form N'&SYSLIST(*n*).
- To indicate the number of entries in nested sublists, specify the appropriate set of subscripts need to reference the selected sublist.

Notes:

1. N'&SYSLIST includes any positional operands that are omitted. Positional operands are omitted by coding a comma where an operand is expected.

2. N'&SYSLIST(*n*) includes those sublist entries specifically omitted by specifying the comma that normally follows the entry.
3. If the operand indicated by *n* is not a sublist, N'&SYSLIST(*n*) is 1. If it is omitted, N'&SYSLIST(*n*) is 0.

The COMPAT(SYSLIST) assembler option instructs the assembler to treat sublists in macro instruction operands as character strings, not sublists. See the section "COMPAT" in the *HLASM Programmer's Guide* for a description of the COMPAT(SYSLIST) assembler option.

Examples of sublists:

```
Macro Instruction               N'&SYSLIST

MACLST   1,2,3,4                    4
MACLST   A,B,,D,E                   5
MACLST   ,A,B,C,D                   5
MACLST   (A,B,C),(D,E,F)            2
MACLST                              0
MACLST   KEY1=A,KEY2=B              0
MACLST   A,B,KEY1=C                 2

                                 N'&SYSLIST(2)

MACSUB   A,(1,2,3,4,5),B            5
MACSUB   A,(1,,3,,5),B              5
MACSUB   A,(,2,3,4,5),B             5
MACSUB   A,B,C                      1
MACSUB   A,,C                       0
MACSUB   A,(),C                     1
MACSUB   A,KEY=(A,B,C)              0
MACSUB                              0
```

&SYSLOC System Variable Symbol

Use &SYSLOC in a macro definition to generate the name of the location counter in effect. If you have not coded a LOCTR instruction between the macro instruction and the preceding START, CSECT, RSECT, DSECT, or COM instruction, the value of &SYSLOC is the same as the value of &SYSECT.

The assembler assigns to the system variable symbol &SYSLOC a local read-only value each time a macro definition containing it is called. The value assigned is the symbol representing the name of the location counter in use at the point where the macro is called.

&SYSLOC can only be used in macro definitions; it has local scope.

Notes:

1. The value of the type attribute of &SYSLOC (T'&SYSLOC) is always U.
2. The value of the count attribute (K'&SYSLOC) is equal to the number of characters assigned as a value to &SYSLOC.
3. Throughout the use of a macro definition, the value of &SYSLOC is considered a constant.

&SYSMAC System Variable Symbol

By varying the subscripts attached to the &SYSMAC you can refer to the name of any of the macros called between open code and the current nesting level, that is, &SYSMAC(&SYSNEST) returns 'OPEN CODE'. Valid subscripts are 0 to &SYSNEST. If &SYSMAC is used with a subscript greater than &SYSNEST, a null character string is returned.

&SYSMAC with no subscript is treated as &SYSMAC(0) and so provides the name of the macro being expanded. This is not considered to be an error and so no message is issued.

The local-scope system variable symbol &SYSMAC is assigned a read-only value each time a macro definition is called.

Notes:

1. The value of the type attribute of &SYSMAC (T'&SYSMAC(n)) is U, unless &SYSMAC(n) is assigned a null character string, in which case the value of the type attribute is O.
2. The value of the count attribute (K'&SYSMAC(n)) is equal to the number of characters assigned as a value to &SYSMAC(n).

&SYSM_HSEV System Variable Symbol

Use &SYSM_HSEV to get the highest MNOTE severity so far for the assembly.

The global-scope system variable symbol &SYSM_HSEV is assigned a read-only value. The assembler compares this value with the severity of MNOTE assembler instructions as they are encountered and, if lower, updates it with the higher value.

Notes:

1. The value of the variable symbol is supplied as three numeric characters, not as an arithmetic (binary) value.
2. The value of the type attribute of &SYSM_SEV (T'&SYSM_SEV) is always N.
3. The value of the count attribute (K'&SYSM_SEV) is always 3.
4. The value of &SYSM_HSEV is unreliable if any MNOTE is incorrectly coded such that a diagnostic message is generated for the MNOTE statement. The cause of the diagnostic message must be corrected.

In Figure 32 on page 245 the &SYSM_HSEV variable is updated immediately when an MNOTE is issued with a higher severity.

&SYSM_SEV System Variable Symbol

Use &SYSM_SEV to get the highest MNOTE severity code for the macro most recently called directly from this level.

The global-scope system variable symbol &SYSM_SEV is assigned a read-only value. The assembler assigns a value of zero when a macro is called and when a macro returns (MEND or MEXIT), the highest severity of all MNOTE assembler instructions executed in the called macro is used to update the variable.

Notes:

1. The value of the variable symbol is supplied as three numeric characters, not as an arithmetic (binary) value.
2. The value of the type attribute of &SYSM_SEV (T'&SYSM_SEV) is always N.
3. The value of the count attribute (K'&SYSM_SEV) is always 3.
4. The value of &SYSM_SEV is unreliable if any MNOTE is incorrectly coded such that a diagnostic message is generated for the MNOTE statement. The cause of the diagnostic message must be corrected.

In Figure 32 on page 245 the &SYSM_SEV variable has a value of 0 until INNER returns. The OUTER macro uses &SYSM_SEV to determine which statements to generate, and in this case issues an MNOTE to pass the severity back to the open code.

```
                                    1     MACRO
                                    2     OUTER &SEV
                                    3     DC     A(&SYSM_HSEV,&SYSM_SEV) outer 1
                                    4     MNOTE  &SEV,'OUTER - parm severity=&SEV'
                                    5     DC     A(&SYSM_HSEV,&SYSM_SEV) outer 2
                                    6     INNER
                                    7     DC     A(&SYSM_HSEV,&SYSM_SEV) outer 3
                                    8     AIF ('&SEV' GT '&SYSM_SEV').MN
                                    9     MNOTE  &SYSM_SEV,'OUTER - returned severity=&SYSM_SEV'
                                   10 .MN ANOP
                                   11     DC     A(&SYSM_HSEV,&SYSM_SEV) outer 4
                                   12     MEND
                                   13     MACRO
                                   14     INNER
                                   15     DC     A(&SYSM_HSEV,&SYSM_SEV) inner 1
                                   16     MNOTE  8,'INNER'
                                   17     DC     A(&SYSM_HSEV,&SYSM_SEV) inner 2
                                   18     MEND
000000                 00000 00040 19 E_G CSECT
                                   20 *,OPEN CODE    an mnote comment - sev=0
                                   21     DC     A(&SYSM_HSEV,&SYSM_SEV) open_code
000000 0000000000000000              +    DC     A(000,000)              open_code
                                   22     OUTER  4
000008 0000000000000000            23+    DC     A(000,000)              outer 1
** ASMA254I *** MNOTE ***          24+     4,OUTER - parm severity=4
000010 0000000400000000            25+    DC     A(004,000)              outer 2
000018 0000000400000000            26+    DC     A(004,000)              inner 1
** ASMA254I *** MNOTE ***          27+     8,INNER
000020 0000000800000000            28+    DC     A(008,000)              inner 2
000028 0000000800000008            29+    DC     A(008,008)              outer 3
** ASMA254I *** MNOTE ***          30+     008,OUTER - returned severity=008
000030 0000000800000008            31+    DC     A(008,008)              outer 4
                                   32 *,OPEN CODE    an mnote comment - sev=0
                                   33     DC     A(&SYSM_HSEV,&SYSM_SEV) open_code
000038 0000000800000008              +    DC     A(008,008)              open_code
                                   34     END
```

Figure 32. Example of the behavior of the &SYSM_HSEV and &SYSM_SEV variables

&SYSNDX System Variable Symbol

For each macro invocation, a new value of &SYSNDX is assigned. The previous value is incremented by 1. Thus, you can attach &SYSNDX to the end of a symbol inside a macro definition to generate a unique suffix for that symbol each time you call the definition. Although an apparently identical symbol is to be generated by two or more calls to the same definition, the suffix provided by &SYSNDX produces two or more unique symbols. For example, the symbol ABC&SYSNDX might generate ABC0001 on one invocation of a macro, and ABC0002 on the next invocation. Thus you avoid an error being flagged for multiply defined symbols.

The local-scope system variable symbol &SYSNDX is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to &SYSNDX is a number from 1 to 9999999. For the numbers 0001 through 9999, four digits are generated. For the numbers 10000 through 9999999, the value is generated with no zeros to the left. The value 0001 is assigned to the first macro called by a program, and is incremented by one for each subsequent macro call (including nested macro calls).

The maximum value for &SYSNDX can be controlled by the MHELP instruction described under 351.

Notes:

1. &SYSNDX does not generate a valid symbol, and it must:
   - Follow the alphabetic character to which it is concatenated
   - Be concatenated to a symbol containing 59 characters or fewer
2. The value of the type attribute of &SYSNDX (T'&SYSNDX) is always N.
3. The value of the count attribute (K'&SYSNDX) is equal to the number of digits generated. If a symbol generated by one macro is to be referenced by code generated by another macro, the two macros must provide means for communicating the necessary information. Their respective values of &SYSNDX cannot be guaranteed to differ by any fixed amount.

The example that follows shows the use of &SYSNDX, and a way to communicate local &SYSNDX values among macro instructions. It is assumed that the first macro instruction processed, OUTER1, is the 106th macro instruction processed by the assembler.

```
         MACRO
         INNER1
         GBLC              &NDXNUM
A&SYSNDX SR                2,5                 Statement 1
         CR                2,5
         BE                B&NDXNUM            Statement 2
         B                 A&SYSNDX            Statement 3
         MEND

         MACRO
&NAME    OUTER1
         GBLC              &NDXNUM
&NDXNUM  SETC              '&SYSNDX'           Statement 4
&NAME    SR                2,4
         AR                2,6
         INNER1                                Statement 5
B&SYSNDX S                 2,=F'1000'          Statement 6
         MEND
---------------------------------------------------------------------
ALPHA    OUTER1                                Statement 7
BETA     OUTER1                                Statement 8
---------------------------------------------------------------------
ALPHA    SR                2,4
         AR                2,6
A0107    SR                2,5
         CR                2,5
         BE                B0106
         B                 A0107
B0106    S                 2,=F'1000'
BETA     SR                2,4
         AR                2,6
A0109    SR                2,5
         CR                2,5
         BE                B0108
         B                 A0109
B0108    S                 2,=F'1000'
```

Statement 7 is the 106th macro instruction processed. Therefore, &SYSNDX is assigned the number 0106 for that macro instruction. The number 0106 is substituted for &SYSNDX when it is used in statements 4 and 6. Statement 4 is used to assign the character value 0106 to the SETC symbol &NDXNUM Statement 6 is used to create the unique name B0106.

Statement 5 is the 107th macro instruction processed. Therefore, &SYSNDX is assigned the number 0107 for that macro instruction. The number 0107 is substituted for &SYSNDX when it is used in statements 1 and 3. The number 0106 is substituted for the global-scope SETC symbol &NDXNUM in statement 2.

Statement 8 is the 108th macro instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0108. For example, statement 6 is used to create the unique name B0108.

When statement 5 is used to process the 108th macro instruction, statement 5 becomes the 109th macro instruction processed. Therefore, each occurrence of &SYSNDX is replaced by the number 0109. For example, statement 1 is used to create the unique name A0109.

&SYSNEST System Variable Symbol

Use &SYSNEST to obtain the current macro instruction nesting level.

The local-scope system variable symbol &SYSNEST is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to &SYSNEST is a number from 1 to 99999999. No leading zeros are generated as part of the number. When a macro is called from open code, the value assigned to &SYSNEST is the number 1. Each time a macro definition is called by an inner macro instruction, the value assigned to &SYSNEXT is incremented by 1. Each time an inner macro exits, the value is decremented by 1.

Notes:

1. The value of the type attribute of &SYSNEST (T'&SYSNEST) is always N.
2. The value of the count attribute (K'&SYSNEST) is equal to the number of digits assigned.

The following example shows the values assigned to &SYSNEST:

```
          MACRO
          OUTER
          DC              A(&SYSNEST)         Statement 1
          INNER1                              Statement 2
          INNER2                              Statement 3
          MEND

          MACRO
          INNER1
          DC              A(&SYSNEST)         Statement 4
          INNER2                              Statement 5
          MEND

          MACRO
          INNER2
          DC              A(&SYSNEST)         Statement 6
          MEND
-----------------------------------------------------------------
          OUTER                               Statement 7
+         DC              A(1)
+         DC              A(2)
+         DC              A(3)
+         DC              A(2)
```

Statement 7 is in open code. It calls the macro OUTER. &SYSNEST is assigned a value of 1 which is substituted in statement 1.

Statement 2, within the macro definition of OUTER, calls macro INNER1. The value assigned to &SYSNEST is incremented by 1. The value 2 is substituted for &SYSNEST in statement 4.

Statement 5, within the macro definition of INNER1, calls macro INNER2. The value assigned to &SYSNEST is incremented by 1. The value 3 is substituted for &SYSNEST in statement 6.

When the macro INNER2 exits, the value assigned to &SYSNEST is decremented by 1. The value of &SYNEST is 2.

When the macro INNER1 exits, the value assigned to &SYSNEST is decremented by 1. The value of &SYSNEST is 1.

Statement 3, within the macro definition of OUTER, calls macro INNER2. The value assigned to &SYSNEST is incremented by 1. The value 2 is substituted for &SYSNEST in statement 6.

&SYSOPT_DBCS System Variable Symbol

You can use &SYSOPT_DBCS to determine if the DBCS assembler option was supplied for the assembly of your source module. &SYSOPT_DBCS is a Boolean system variable symbol, and has a global scope.

If the DBCS assembler option was specified, &SYSOPT_DBCS is assigned a value of 1. If the DBCS assembler option was not specified, &SYSOPT_DBCS is assigned a value of 0.

For more information about the DBCS assembler option, see the section "DBCS" in the *HLASM Programmer's Guide*.

Notes:

1. The value of the type attribute of &SYSOPT_DBCS (T'&SYSOPT_DBCS) is always N.
2. The value of the count attribute (K'&SYSOPT_DBCS) is always 1.

&SYSOPT_OPTABLE System Variable Symbol

Use &SYSOPT_OPTABLE to determine the value that was specified for the OPTABLE assembler option. &SYSOPT_OPTABLE has a global scope.

The value that was specified for the OPTABLE assembler option indicates which operation code table was specified by the OPTABLE or MACHINE option.

For more information about the OPTABLE assembler option, see the section "OPTABLE" in the *HLASM Programmer's Guide*).

Notes:

1. The value of the type attribute of &SYSOPT_OPTABLE (T'&SYSOPT_OPTABLE) is always U.
2. The value of the count attribute (K'&SYSOPT_OPTABLE) is the number of characters assigned.

&SYSOPT_RENT System Variable Symbol

Use &SYSOPT_RENT to determine if the RENT assembler option was specified for the assembly of your source module. The RENT option instructs the assembler to check for possible coding violations of program reenterability. &SYSOPT_RENT is a Boolean system variable symbol, and has a global scope.

If the RENT assembler option was specified, &SYSOPT_RENT is assigned a value of 1. If the RENT assembler option was not specified, &SYSOPT_RENT is assigned a value of 0.

For more information about the RENT assembler option, see the section "RENT" in the *HLASM Programmer's Guide*.

Notes:

1. The value of the type attribute of &SYSOPT_RENT (T'&SYSOPT_RENT) is always N.
2. The value of the count attribute (K'&SYSOPT_RENT) is always 1.

&SYSOPT_XOBJECT System Variable Symbol

The &SYSOPT_XOBJECT system variable is set to 1 if GOFF or XOBJECT is specified, otherwise it is set to 0.

&SYSOPT_XOBJECT is a Boolean system variable symbol with global scope.

Notes:

1. The value of the type attribute of &SYSOPT_XOBJECT (T'&SYSOPT_XOBJECT) is always N.
2. The value of the count attribute (K'&SYSOPT_XOBJECT) is always 1.

&SYSPARM System Variable Symbol

The &SYSPARM system variable is assigned a read-only value from the assembler option SYSPARM. It is treated as a global-scope SETC symbol in a source module except that its value cannot be changed. (See the chapter "Controlling Your Assembly with Options" in the *HLASM Programmer's Guide* for information about assembler options.)

Notes:

1. The largest value that &SYSPARM can hold is 1024 characters. However, if the PARM field of the EXEC statement is used to specify its value, the PARM field restrictions reduce its maximum possible length.
2. No values are substituted for variable symbols in the specified value, however, on z/OS and z/VSE, you must use double ampersands to represent a single ampersand.
3. On z/OS and z/VSE, you must use two apostrophes to represent an apostrophes, because the entire EXEC PARM field is enclosed in apostrophes.
4. If the SYSPARM assembler option is not specified, &SYSPARM is assigned the default value that was specified when the assembler was installed on your system.

   If a default value for SYSPARM was not specified when the assembler was installed on your system, &SYSPARM is assigned a value of the null character string.
5. The value of the type attribute of &SYSPARM (T'&SYSPARM) is U, unless &SYSPARM is assigned a null value, in which case the value of the type attribute is O.
6. The value of the count attribute (K'&SYSPARM) is the number of characters assigned as a value to &SYSPARM. If &SYSPARM is assigned a null character string, the value of the count attribute is 0.
7. If the SYSPARM option is passed to the assembler through the ASMAOPT file (CMS and z/OS) or Librarian member (z/VSE) and the option contains embedded spaces, it must be enclosed in quotes.

&SYSPRINT_DSN System Variable Symbol

Use &SYSPRINT_DSN in a macro definition to obtain the name of the data set to which the assembler writes the assembler listing.

The local-scope system variable symbol &SYSPRINT_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSPRINT_DSN is always the value stored in the JFCB for SYSPRINT. If SYSPRINT is allocated to DUMMY, or a NULLFILE, the value in &SYSPRINT_DSN is `NULLFILE`.

When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to &SYSPRINT_DSN is determined as follows:

Table 43. Contents of &SYSPRINT_DSN on CMS

SYSPRINT Allocated To:	Contents of &SYSPRINT_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Dummy file (no physical I/O)	`DUMMY`
Printer	`PRINTER`
Labeled tape file	The data set name of the tape file

Table 43. Contents of &SYSPRINT_DSN on CMS (continued)

SYSPRINT Allocated To:	Contents of &SYSPRINT_DSN:
Unlabeled tape file	`TAPn`, where *n* is a value from 0 to 9, or A to F.
Terminal	`TERMINAL`

When the assembler runs on z/VSE, the value of the character string assigned to &SYSPRINT_DSN is determined as follows:

Table 44. Contents of &SYSPRINT_DSN on z/VSE

SYSLST Assigned To:	Contents of &SYSPRINT_DSN:
Disk file (not for dynamic partitions)	The file-id
Printer	`SYSLST`
Labeled tape file	The file ID of the tape file
Unlabeled tape file	`SYSLST`

Examples:

On z/OS, &SYSPRINT_DSN might be assigned a value such as:

```
IBMAPC.IBMAPCA.JOB06734.D0000102.?
```

On CMS, &SYSPRINT_DSN might be assigned a value such as:

```
SAMPLE   LISTING  A1
```

Notes:

1. If the LISTING user exit provides the listing data set information then the value in &SYSPRINT_DSN is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSPRINT_DSN (T'&SYSPRINT_DSN) is always U.
3. The value of the count attribute of &SYSPRINT_DSN (K'&SYSPRINT_DSN) is equal to the number of characters assigned as a value to &SYSPRINT_DSN.

&SYSPRINT_MEMBER System Variable Symbol

z/VSE The value of &SYSPRINT_MEMBER is always null.

The value of the type attribute is O, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSPRINT_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the assembler listing.

The local-scope system variable symbol &SYSPRINT_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the assembler listing is not a z/OS partitioned data set, &SYSPRINT_MEMBER is assigned a null character string.

Notes:

1. If the LISTING user exit provides the listing data set information then the value in &SYSPRINT_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.

2. The value of the type attribute of &SYSPRINT_MEMBER (T'&SYSPRINT_MEMBER) is U, unless &SYSPRINT_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSPRINT_MEMBER (K'&SYSPRINT_MEMBER) is equal to the number of characters assigned as a value to &SYSPRINT_MEMBER. If &SYSPRINT_MEMBER is assigned a null character string, the value of the count attribute is 0.

&SYSPRINT_VOLUME System Variable Symbol

Use &SYSPRINT_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler writes the assembler listing.

The local-scope system variable symbol &SYSPRINT_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the assembler listing writes to a Shared File System CMS file, &SYSPRINT_VOLUME is assigned the value `** SFS`.

If the volume on which the data set resides is not labeled, &SYSPRINT_VOLUME is assigned a null character string.

Notes:

1. If the LISTING user exit provides the listing data set information then the value in &SYSPRINT_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSPRINT_VOLUME (T'&SYSPRINT_VOLUME) is U, unless &SYSPRINT_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSPRINT_VOLUME (K'&SYSPRINT_VOLUME) is equal to the number of characters assigned as a value to &SYSPRINT_VOLUME. If &SYSPRINT_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.

&SYSPUNCH_DSN System Variable Symbol

Use &SYSPUNCH_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the object records when assembler option DECK is specified.

The local-scope system variable symbol &SYSPUNCH_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSPUNCH_DSN is always the value stored in the JFCB for SYSPUNCH. If SYSPUNCH is allocated to DUMMY, or a NULLFILE, the value in &SYSPUNCH_DSN is `NULLFILE`.

When the assembler runs on the CMS component of the VM operating systems, the value of the character string assigned to &SYSPUNCH_DSN is determined as follows:

Table 45. Contents of &SYSPUNCH_DSN on CMS

SYSPUNCH Allocated To:	Contents of &SYSPUNCH_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Dummy file (no physical I/O)	`DUMMY`
Punch	`PUNCH`

Table 45. Contents of &SYSPUNCH_DSN on CMS (continued)

SYSPUNCH Allocated To:	Contents of &SYSPUNCH_DSN:
Labeled tape file	The data set name of the tape file
Unlabeled tape file	TAP*n*, where *n* is a value from 0 to 9, or A to F.

On z/VSE, the value of the character string assigned to &SYSPUNCH_DSN is determined as follows:

Table 46. Contents of &SYSPUNCH_DSN on z/VSE

SYSPCH Assigned To:	Contents of &SYSPUNCH_DSN:
Disk file	The file-id
Punch	SYSPCH
Labeled tape file	The file ID of the tape file
Unlabeled tape file	SYSPCH

Examples:

On z/OS, &SYSPUNCH_DSN might be assigned a value such as:

```
IBMAPC.IBMAPCA.JOB06734.D0000103.?
```

On CMS, &SYSPUNCH_DSN might be assigned a value such as:

```
PUNCH
```

Notes:

1. If the PUNCH user exit provides the punch data set information then the value in &SYSPUNCH_DSN is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSPUNCH_DSN (T'&SYSPUNCH_DSN) is always U.
3. The value of the count attribute of &SYSPUNCH_DSN (K'&SYSPUNCH_DSN) is equal to the number of characters assigned as a value to &SYSPUNCH_DSN.

&SYSPUNCH_MEMBER System Variable Symbol

z/VSE The value of &SYSPUNCH_MEMBER is always null.

The value of the type attribute is O, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSPUNCH_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the object records when the assembler option DECK is specified.

The local system variable symbol &SYSPUNCH_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the object records is not a z/OS partitioned data set, &SYSPUNCH_MEMBER is assigned a null character string.

Notes:

1. If the PUNCH user exit provides the punch data set information then the value in &SYSPUNCH_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.

2. The value of the type attribute of &SYSPUNCH_MEMBER (T'&SYSPUNCH_MEMBER) is U, unless &SYSPUNCH_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSPUNCH_MEMBER (K'&SYSPUNCH_MEMBER) is equal to the number of characters assigned as a value to &SYSPUNCH_MEMBER. If &SYSPUNCH_MEMBER is assigned a null character string, the value of the count attribute is 0.

&SYSPUNCH_VOLUME System Variable Symbol

Use &SYSPUNCH_VOLUME in a macro definition to obtain the volume identifier of the object data set. The volume identifier is of the first volume containing the data set. &SYSPUNCH_VOLUME is only assigned a value when you specify the DECK assembler option.

The local-scope system variable symbol &SYSPUNCH_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the VM operating system, and the object records are being written to a Shared File System CMS file, &SYSPUNCH_VOLUME is assigned the value `** SFS`.

If the volume on which the data set resides is not labeled, &SYSPUNCH_VOLUME is assigned a null character string.

Notes:

1. If the PUNCH user exit provides the punch data set information then the value in &SYSPUNCH_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSPUNCH_VOLUME (T'&SYSPUNCH_VOLUME) is U, unless &SYSPUNCH_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSPUNCH_VOLUME (K'&SYSPUNCH_VOLUME) is equal to the number of characters assigned as a value to &SYSPUNCH_VOLUME. If &SYSPUNCH_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.

&SYSSEQF System Variable Symbol

Use &SYSSEQF in a macro definition to obtain the value of the identification-sequence field of the macro instruction in open code that caused, directly or indirectly, the macro to be called.

The local-scope system variable symbol &SYSSEQF is assigned a read-only value each time a macro definition is called from a source module.

The value assigned to &SYSSEQF is determined as follows:

1. If no ICTL instruction has been specified and sequence checking is not active, the contents of columns 73 to 80 inclusive of the source statement are assigned to &SYSSEQF.
2. If an ICTL instruction has been specified, but sequence checking is not active, the contents of the columns of the source statement to the right of the continuation-indicator column are assigned to &SYSSEQF. If the end column or the continuation-indicator column is 80, &SYSSEQF is assigned a null character string.
3. If an ISEQ instruction with operands has been specified to start sequence checking, the contents of columns specified in the ISEQ instruction operand are assigned to &SYSSEQF.
4. If an ISEQ instruction without an operand has been specified to end sequence checking, steps (1) and (2) are used to determine the value assigned to &SYSSEQF.

Notes:

1. The value of the type attribute of &SYSSEQF (T'&SYSSEQF) is U, unless &SYSSEQF is assigned a null character string, in which case the value of the type attribute is O.
2. The value of the count attribute of &SYSSEQF (K'&SYSSEQF) is equal to the number of characters assigned as a value to &SYSSEQF. If &SYSSEQF is assigned a null character string, the value of the count attribute is 0.
3. Throughout the use of a macro definition, the value of &SYSSEQF is considered a constant.

&SYSSTEP System Variable Symbol

Use &SYSSTEP to obtain the step name of the job step used to assemble your source module. &SYSSTEP has a global scope.

On CMS and z/VSE the value of &SYSSTEP is always (NOSTEP).

Notes:

1. The value of the type attribute of &SYSSTEP (T'&SYSSTEP) is always U.
2. The value of the count attribute (K'&SYSSTEP) is the number of characters assigned.

&SYSSTMT System Variable Symbol

Use &SYSSTMT to obtain the next statement number that is assigned to a statement by the assembler. &SYSSTMT has a global scope.

The value assigned to &SYSSTMT is an 8-character string, padded on the left with leading zero (X'F0') characters. The following example shows the value assigned to &SYSSTMT. It assumes that the DC statement is in open code, and is the 23rd statement in the source module.

```
23          DC    C'&SYSSTMT'
  +         DC    C'00000024'
```

Notes:

1. The value of the type attribute of &SYSSTMT (T'&SYSSTMT) is always N.
2. The value of the count attribute of &SYSSTMT (K'&SYSSTMT) is always 8.

&SYSSTYP System Variable Symbol

Use &SYSSTYP in a macro definition to generate the type of the current control section. The current control section is the control section in which the macro instruction that calls the definition appears.

The local-scope system variable symbol &SYSSTYP is assigned a read-only value each time a macro definition is called.

The value assigned is the symbol that represents the type of the current control section in effect when the macro is called. A control section that has been initiated or continued by substitution does not affect the value of &SYSSTYP for the expansion of the current macro. However, it does affect &SYSSTYP for a subsequent macro call. Nested macros cause the assembler to assign a value to &SYSSTYP that depends on the control section in force inside the calling macro when the inner macro is called.

The control section whose type is assigned to &SYSSTYP can be defined by a program sectioning statement. This can be a START, CSECT, RSECT, DSECT, or COM statement, or, for the first control section, any instruction described in “First section” on page 46. Depending upon the instruction used to initiate the current control section, the value assigned to &SYSSTYP is either CSECT, RSECT, DSECT, or COM. If the current control section is unnamed, or is an executable control section initiated by other than a START, CSECT, or RSECT instruction, then the value assigned to &SYSSTYP is CSECT.

If a control section has not been initiated, &SYSSTYP is assigned a null character string.

Notes:

1. The value of the type attribute of &SYSSTYP (T'&SYSSTYP) is U, unless &SYSSTYP is assigned a null character string, in which case the value of the type attribute is O.
2. The value of the count attribute of &SYSSTYP (K'&SYSSTYP) is equal to the number of characters assigned as a value to &SYSSTYP. If &SYSSTYP is assigned a null character string, the value of the count attribute is 0.
3. Throughout the use of a macro definition, the value of &SYSSTYP is considered a constant.

&SYSTEM_ID System Variable Symbol

Use &SYSTEM_ID to obtain the name and release of the operating system under which your source module is being assembled. &SYSTEM_ID has a global scope.

For example, on z/OS, &SYSTEM_ID might contain one of the following:

```
z/OS 01.04.00
z/OS 01.05.00
... and so on
```

on CMS, &SYSTEM_ID might contain one of the following:

```
CMS 18
CMS 19
... and so on
```

on z/VSE, &SYSTEM_ID might contain one of the following:

```
VSE/AF 6.6.0
... and so on
```

Notes:

1. The value of the type attribute of &SYSTEM_ID (T'&SYSTEM_ID) is always U.
2. The value of the count attribute (K'&SYSTEM_ID) is the number of characters assigned.

&SYSTERM_DSN System Variable Symbol

Use &SYSTERM_DSN in a macro definition to obtain the name of the data set to which the assembler is writing the terminal records.

The local-scope system variable symbol &SYSTERM_DSN is assigned a read-only value each time a macro definition is called.

When the assembler runs on the z/OS operating systems, the value of the character string assigned to &SYSTERM_DSN is always the value stored in the JFCB for SYSTERM. If SYSTERM is allocated to DUMMY, or a NULLFILE, the value in &SYSTERM_DSN is `NULLFILE`.

When the assembler runs on the CMS component of the z/VM operating systems, the value of the character string assigned to &SYSTERM_DSN is determined as follows:

Table 47. Contents of &SYSTERM_DSN on CMS

SYSTERM Allocated To:	Contents of &SYSTERM_DSN:
CMS file	The 8-character file name, the 8-character file type, and the 2-character file mode of the file, each separated by a space
Dummy file (no physical I/O)	`DUMMY`
Printer	`PRINTER`
Labeled tape file	The data set name of the tape file

Table 47. Contents of &SYSTERM_DSN on CMS (continued)

SYSTERM Allocated To:	Contents of &SYSTERM_DSN:
Unlabeled tape file	TAP*n*, where *n* is a value from 0 to 9, or A to F.
Terminal	TERMINAL

On z/VSE, the value of the character string assigned to &SYSTERM_DSN is always SYSLOG.

Examples:

On z/OS, &SYSTERM_DSN might be assigned a value such as:

```
IBMAPC.IBMAPCA.JOB06734.D0000104.?
```

On CMS, &SYSTERM_DSN might be assigned a value such as:

```
TERMINAL
```

Notes:

1. If the TERM user exit provides the terminal data set information then the value in &SYSTERM_DSN is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSTERM_DSN (T'&SYSTERM_DSN) is always U.
3. The value of the count attribute of &SYSTERM_DSN (K'&SYSTERM_DSN) is equal to the number of characters assigned as a value to &SYSTERM_DSN.

&SYSTERM_MEMBER System Variable Symbol

z/VSE The value of &SYSTERM_MEMBER is always null.

The value of the type attribute is O, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSTERM_MEMBER in a macro definition to obtain the name of the data set member to which the assembler is writing the terminal records.

The local-scope system variable symbol &SYSTERM_MEMBER is assigned a read-only value each time a macro definition is called.

If the data set to which the assembler is writing the terminal records is not a z/OS partitioned data set, &SYSTERM_MEMBER is assigned a null character string.

Notes:

1. If the TERM user exit provides the terminal data set information then the value in &SYSTERM_MEMBER is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSTERM_MEMBER (T'&SYSTERM_MEMBER) is U, unless &SYSTERM_MEMBER is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSTERM_MEMBER (K'&SYSTERM_MEMBER) is equal to the number of characters assigned as a value to &SYSTERM_MEMBER. If &SYSTERM_MEMBER is assigned a null character string, the value of the count attribute is 0.

&SYSTERM_VOLUME System Variable Symbol

z/VSE The value of &SYSTERM_VOLUME is always null.

The value of the type attribute is U, and the value of the count attribute is 0.

z/VM and z/OS

You can use &SYSTERM_VOLUME in a macro definition to obtain the volume identifier of the first volume containing the data set to which the assembler is writing the terminal records.

The local-scope system variable symbol &SYSTERM_VOLUME is assigned a read-only value each time a macro definition is called.

If the assembler runs on the CMS component of the z/VM operating system, and the terminal records are being written to a Shared File System CMS file, &SYSTERM_VOLUME is assigned the value "** SFS".

If the volume on which the data set resides is not labeled, &SYSTERM_VOLUME is assigned a null character string.

Notes:

1. If the TERM user exit provides the terminal data set information then the value in &SYSTERM_VOLUME is the value extracted from the Exit-Specific Information block described in the section "Exit-Specific Information Block" in the *HLASM Programmer's Guide*.
2. The value of the type attribute of &SYSTERM_VOLUME (T'&SYSTERM_VOLUME) is U, unless &SYSTERM_VOLUME is assigned a null character string, in which case the value of the type attribute is O.
3. The value of the count attribute of &SYSTERM_VOLUME (K'&SYSTERM_VOLUME) is equal to the number of characters assigned as a value to &SYSTERM_VOLUME. If &SYSTERM_VOLUME is assigned a null character string, the value of the count attribute is 0. The maximum length of this system variable symbol is 6.

&SYSTIME System Variable Symbol

Use &SYSTIME to obtain the time at which your source module is assembled. It has local scope, but can be used in open code. It is assigned a read-only value.

The value of &SYSTIME is a 5-character string in the format:

```
HH.MM
```

where:

HH Is two-digit field that gives the hour of the day. It has a value 00 - 23.

MM Is two-digit field that gives the minute of the hour. It has a value 00 - 59. It is separated from HH by a period.

Example:

```
09.45
```

Notes:

1. The time corresponds to the time printed in the page heading of listings and remains constant for each assembly.
2. The value of the type attribute of &SYSTIME (T'&SYSTIME) is always U.
3. The value of the count attribute (K'&SYSTIME) is always 5.

&SYSVER System Variable Symbol

Use &SYSVER to obtain the version, release, and modification level of the assembler being used to assemble your source module. &SYSVER has a global scope. For example, when High Level Assembler 6 is used, &SYSVER has the value "1.6.0".

Notes:

1. The value of the type attribute of &SYSVER (T'&SYSVER) is always U.
2. The value of the count attribute (K'&SYSVER) is the number of characters assigned. In the example where the value of &SYSVER is “1.6.0”, the count attribute of &SYSVER is 5.

Chapter 8. How to write macro instructions

This chapter describes macro instructions; where you can use them and how you specify them.

The first section, "Macro instruction format" describes the macro instruction format, including details on the name, operation, and operand entries, and what is generated as a result of a macro instruction.

"Sublists in operands" on page 266 describes how you can use sublists to specify several values in an operand entry.

"Values in operands" on page 269 describes the values you can specify in an operand entry when you call a macro definition.

"Nesting macro instruction definitions" on page 272 describes how you can use nested macro call instructions to call macros from within a macro.

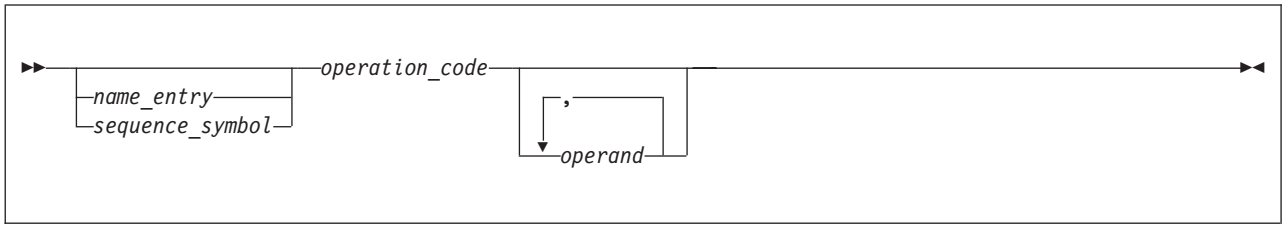
What is a Macro Instruction: The macro instruction provides the assembler with:
- The name of the macro definition to process
- The information or values to pass to the macro definition

This information is the input to a macro definition. The assembler uses the information either in processing the macro definition, or for substituting values into model statements in the definition.

The output from a macro definition can be:
- A sequence of statements generated from the model statements of the macro for further processing at assembly time.
- Values assigned to global-scope SET symbols. These values can be used in other macro definitions and in open code (see "SET symbols" on page 279).

Where Macro Instructions Can Appear: A macro instruction can be written anywhere in your program, provided the assembler can find the macro definition. The macro definition can be found either in a macro library, or in the source program before the macro instruction, or be provided by a LIBRARY user exit. However, the statements generated from the called macro definition must be valid assembler language instructions and allowed where the calling macro instruction appears.

Macro instruction format



name_entry
Is a special positional operand that can be used to pass a value into the called macro definition. For a detailed description of what form *name_entry* can take, see "Name entry" on page 261.

sequence_symbol
Is a sequence symbol. If a sequence symbol is coded in the name entry of a macro instruction, the value of the symbol is not passed to the called macro definition and therefore cannot be used as a value for substitution in the macro definition.

operation_code
Is the symbolic operation code which identifies the macro definition that you want the assembler to process. For more information, see "Operation entry" on page 261.

operand
The positional operands or keyword operands that you use to pass values into the called macro definition. For more information, see "Operand entry" on page 261.

If no operands are specified in the operand field, and if the absence of the operand entry is indicated by a comma preceded and followed by one or more spaces, remarks are allowed.

The entries in the name, operation, and operand fields correspond to entries in the prototype statement of the called macro definition (see "Macro instruction prototype" on page 215).

Alternative formats for a macro instruction

A macro instruction can be specified in one of the three following ways:
- The normal way, with the operands preceding any remarks
- The alternative way, allowing remarks for each operand
- A combination of the first two ways

The alternative statement format is not available for machine instructions.

The following example shows the normal statement format (*NAME1*), the alternative statement format (*NAME2*), and a combination of both statement formats (*NAME3*).

```
       Opera-
Name   tion   Operand                        Comment                Cont.

NAME1  OP1    OPERAND1,OPERAND2,OPERAND3     This is the normal       X
                                             statement format

NAME2  OP2    OPERAND1,                      This is the alter-       X
              OPERAND2                       native statement format

NAME3  OP3    OPERAND1,                      This is a combination    X
              OPERAND2,OPERAND3              of both
```

Notes:

1. Any number of continuation lines are allowed. However, each continuation line must be indicated by a non-space character in the column after the end column of the previous statement line (see "Continuation lines" on page 13).
2. If the DBCS assembler option is specified, the continuation features outlined in "Continuation of double-byte data" on page 13 apply to continuation in the macro language. Extended continuation might be useful if a macro operand contains double-byte data.
3. Operands on continuation lines must begin in the continue column (column 16), or the assembler assumes that the current line and any lines that follow contain remarks.

   If any entries are made in the columns before the continue column in continuation lines, the assembler issues an error message and the whole statement is not processed.
4. One or more spaces must separate the operand from the remarks.
5. A comma after an operand indicates more operands follow.
6. The last operand requires no comma following it, but using a comma does not cause an error.

7. You do not need to use the same format when you code a macro instruction as you use when you code the corresponding macro prototype statement.
8. Continued comments for a macro with an operand list that terminates in a null operand are recognized provided each continued comment begins in the same or later column as the preceding line's comment.

Name entry

Use the name entry of a macro instruction to:

- Pass a value into a macro definition through the name entry declared in the macro definition
- Provide a conditional assembly label (see "Sequence symbols" on page 298) so that you can branch to the macro instruction during conditional assembly if you want the called macro definition expanded.

The name entry of a macro instruction can be:

- Space
- An ordinary symbol, such as `HERE`
- A variable symbol, such as `&A.`
- Any combination of variable symbols and other character strings concatenated together, such as `HERE.&A`
- Any character string allowed in a macro instruction operand, such as `'Now is the hour'` or `STRING00`, excluding sublist entries and certain attribute references (see "Values in operands" on page 269)
- A sequence symbol, which is not passed to the macro definition, such as `.SEQ`

Operation entry

The operation entry is the symbolic name of the operation code that identifies a macro definition to process.

The operation entry must be a valid symbol, and must be identical to the operation field in the prototype statement of the macro definition.

The assembler searches for source macro definitions before library macro definitions. If you have a source macro definition that has the same name as a library macro definition, the assembler only processes the source macro definition.

You can use a variable symbol as a macro instruction. For example if `MAC1` has been defined as a macro, you can use the following statements to call it:

```
&CALL    SETC            'MAC1'
         &CALL
```

You cannot use a variable symbol as a macro instruction that passes operands to the macro. The second statement in the following example generates an error:

```
&CALL    SETC            'MAC1 OPERAND1=VALUE'
         &CALL
```

You must specify operand entries after the variable symbol, as shown in the following example:

```
&CALL    SETC            'MAC1'
         &CALL OPERAND1=VALUE
```

Operand entry

Use the operand entry of a macro instruction to pass values into the called macro definition. These values can be passed through:

- The symbolic parameters you have specified in the macro prototype.

- The system variable symbol &SYSLIST if it is specified in the body of the macro definition (see "&SYSLIST System Variable Symbol" on page 241).

The two types of operands allowed in a macro instruction are positional and keyword operands. You can specify a sublist with multiple values in both types of operands. Special rules for the various values you can specify in operands are given in the following subsections.

Positional operands

You can use a positional operand to pass a value into a macro definition through the corresponding positional parameter declared for the definition. Declare a positional parameter in a macro definition when you want to change the value passed at every call to that macro definition.

You can also use a positional operand to pass a value to the system variable symbol &SYSLIST. If &SYSLIST, with the applicable subscripts, is specified in a macro definition, you do not need to declare positional parameters in the prototype statement of the macro definition. You can thus use &SYSLIST to refer to any positional operand. This allows you to vary the number of operands you specify each time you call the same macro definition.

The positional operands of a macro instruction must be specified in the same order as the positional parameters declared in the called macro definition.

Each positional operand constitutes a character string. This character string is the value passed through a positional parameter into a macro definition.

The specification for each positional parameter in the prototype statement definition must be a valid variable symbol. Values are assigned (see **1**) to the positional operands by the corresponding positional arguments see **2** below) specified in the macro instruction that calls the macro definition.

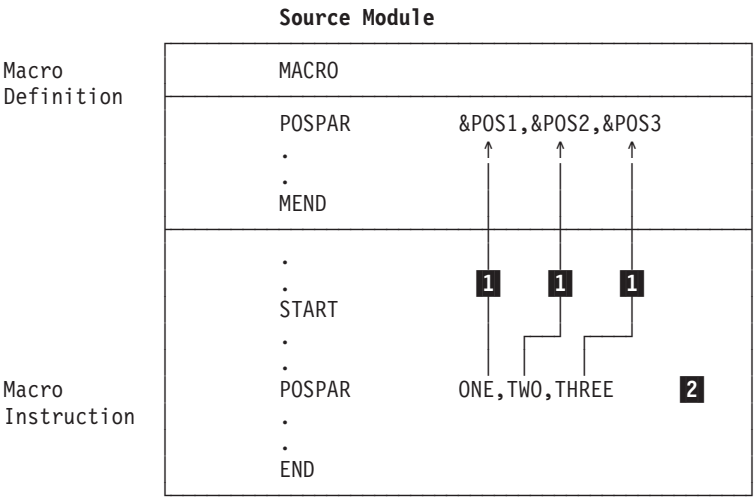


Figure 33. Positional operands

Notes:

1. An omitted operand has a null character value.
2. Each positional operand can be up to1024 characters long.
3. If the DBCS assembler option is specified, the positional operand can be a string containing double-byte data. The string does not need to be quoted.

Here are examples of macro instructions with positional operands:

```
MACCALL         VALUE,9,8
MACCALL         &A,'QUOTED STRING'
MACCALL         EXPR+2,,SYMBOL
MACCALL         (A,B,C,D,E),(1,2,3,4)
MACCALL         &A,'<.S.T.R.I.N.G>'
```

The following list shows what happens when the number of positional operands in the macro instruction is equal to or differs from the number of positional parameters declared in the prototype statement of the called macro definition:

Equal Valid, if operands are correctly specified.

Greater than
Meaningless, unless &SYSLIST is specified in definition to refer to excess operands.

Less than
Omitted operands give null character values to corresponding parameters (or &SYSLIST specification).

Keyword operands

You can use a keyword operand to pass a value through a keyword parameter into a macro definition. The values you specify in keyword operands override the default values assigned to the keyword parameters. Set the default value to a value you use frequently. Thus, you avoid having to write this value every time you code the calling macro instruction.

When you need to change the default value, you must use the corresponding keyword operand in the macro instruction. The keyword can indicate the purpose for which the passed value is used.

Any keyword operand specified in a macro instruction must correspond to a keyword parameter in the macro definition called. However, keyword operands do not have to be specified in any particular order.

The general specifications for symbolic parameters also apply to keyword operands. The actual operand keyword must be a valid variable symbol. A null character string can be specified as the standard value of a keyword operand, and is generated if the corresponding keyword operand is omitted.

A keyword operand must be coded in this format:

```
KEYWORD=VALUE
```

where:

KEYWORD
Has up to 62 characters without an ampersand.

VALUE
Can be up to 1024 characters.

The corresponding keyword parameter in the called macro definition is specified as:

```
&KEYWORD=DEFAULT
```

If a keyword operand is specified, its value overrides the default value specified for the corresponding keyword parameter.

If the DBCS assembler option is specified, the keyword operand can be a string containing double-byte data. The string does not need to be quoted.

If the value of a keyword operand is a literal, two equal signs must be specified.

The following examples of macro instructions have keyword operands:

```
MACKEY          KEYWORD=(A,B,C,D,E)
MACKEY          KEY1=1,KEY2=2,KEY3=3
MACKEY          KEY3=2000,KEY1=0,KEYWORD=HALLO
MACKEY          KEYWORD='<.S.T.R.I.N.G>'
MACKEY          KEYWORD==C'STRING'
```

To summarize the relationship of keyword operands to keyword parameters:

- The keyword of the operand corresponds (see **1** in Figure 34 on page 265) to a keyword parameter. The value in the operand overrides the default value of the parameter.
- If the keyword operand is not specified (see **2** in Figure 34 on page 265), the default value of the parameter is used.
- If the keyword of the operand does not correspond (see **3** in Figure 34 on page 265) to any keyword parameter, the assembler issues an error message, but the macro is generated using the default values of the other parameters.
- The default value specified for a keyword parameter can be the null character string (see **4** in Figure 34 on page 265). The null character string is a character string with a length of zero; it is not a space, because a space occupies one character position.

```
                                    ┌─ 4 Null character string
                                    │     is default value
          MACRO                     ↓
          MACCORR &KEY1=DEFAULT,&KEY2=,&KEY3=123
          .
          .
          DC    C'&KEY1&KEY2&KEY3'
          .
          .
          MEND
_____________________________________________________
______
 OPEN     START 0
          .
          .       1           1     1
          MACCOOR KEY1=OVERRIDE,KEY2=0,KEY3=456
          .        │           │     │
          .        ┌──┘        ┌─────┘     │
          .        ↓           ↓           │
+         DC    C'OVERRIDE0456'            │
          .                   ↑           │
          .                   └───────────┘
          .
          MACCOOR   2
          .                ┌─── KEY2 has null character
          .                │    string as default
          .                ↓
+         DC    C'DEFAULT123'
          .
          .
          .        3
          MACCOOR   KEY4=SYMBOL,KEY2=0
ASMA017W ** WARNING ** Undefined keyword parameter . . .
          .
          .
          .
+         DC    C'DEFAULT0123'
          .
          .
          .
          MACCOOR   KEY1=,KEY3=456
          .        ┌──────────────────── • KEY1 parameter has null
          .        │                       character string value
          .        ↓                   • KEY2 has null character
+         DC    C'456'                    string as default
          END
```

Figure 34. Relationship between keyword operands and keyword parameters and their assigned values

Combining positional and keyword operands

You can use positional and keyword operands in the same macro instruction. Use a positional operand for a value that you change often, and a keyword operand for a value that you change infrequently.

Positional and keyword parameters can be mixed freely in the macro prototype statement (see 1 in Figure 35 on page 266). The same applies to the positional and keyword operands of the macro instruction (see 2 in Figure 35 on page 266). Note, however, that the order in which the positional parameters appear (see 3 in Figure 35 on page 266) determines the order in which the positional operands must appear. Interspersed keyword parameters and operands (see 4 in Figure 35 on page 266) do not affect this order.

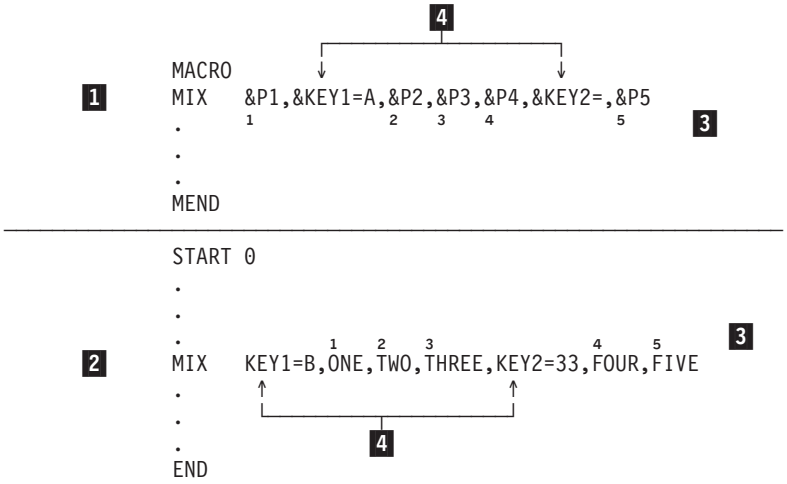


Figure 35. Combining positional and keyword parameters

&SYSLIST(*n*): The system variable symbol &SYSLIST(*n*) refers only to the positional operands in a macro instruction.

Sublists in operands

You can use a sublist in a positional or keyword operand to specify several values. A *sublist* is a character string that consists of one or more entries separated by commas and enclosed in parentheses.

If the COMPAT(SYSLIST) assembler option is not specified, a variable symbol that has been assigned a character string that consists of one or more entries separated by commas and enclosed in parentheses is also treated as a sublist. However, if the COMPAT(SYSLIST) assembler option is specified, a sublist assigned to a variable symbol is treated as a character string, not as a sublist.

A variable symbol is not treated as a sublist if the parentheses are not present. The following example shows two calls to macro MAC1. In the first call, the value of the operand in variable &VAR1 is treated as a sublist. In the second call, the value of the operand is treated as a character string, not a sublist, because the variable &VAR2 does not include parentheses.

```
&VAR1     SETC     '(1,2)'
          MAC1     KEY=&VAR1
&VAR2     SETC     '1,2'
          MAC1     KEY=(&VAR2)
```

To refer to an entry of a sublist code, use:

- The corresponding symbolic parameter with an applicable subscript.
- The system variable symbol &SYSLIST with applicable subscripts, the first of which refers to the positional operand, and the second to the sublist entry in the operand. &SYSLIST can refer only to sublists in positional operands.

Figure 36 on page 267 shows that the value specified in a positional or keyword operand can be a sublist.

A symbolic parameter can refer to the whole sublist (see 1 in Figure 36 on page 267), or to an individual entry of the sublist. To refer to an individual entry, the symbolic parameter (see 2 in Figure 36 on page 267) must have a subscript whose value indicates the position (see 3 in Figure 36 on page 267) of the entry in the sublist. The subscript must have a value greater than or equal to 1.

A sublist, including the enclosing parentheses, must not contain more than 1024 characters. It consists of one or more entries separated by commas and enclosed in parentheses; for example, `(A,B,C,D,E)`. `()` is a valid sublist with the null character string as the only entry.

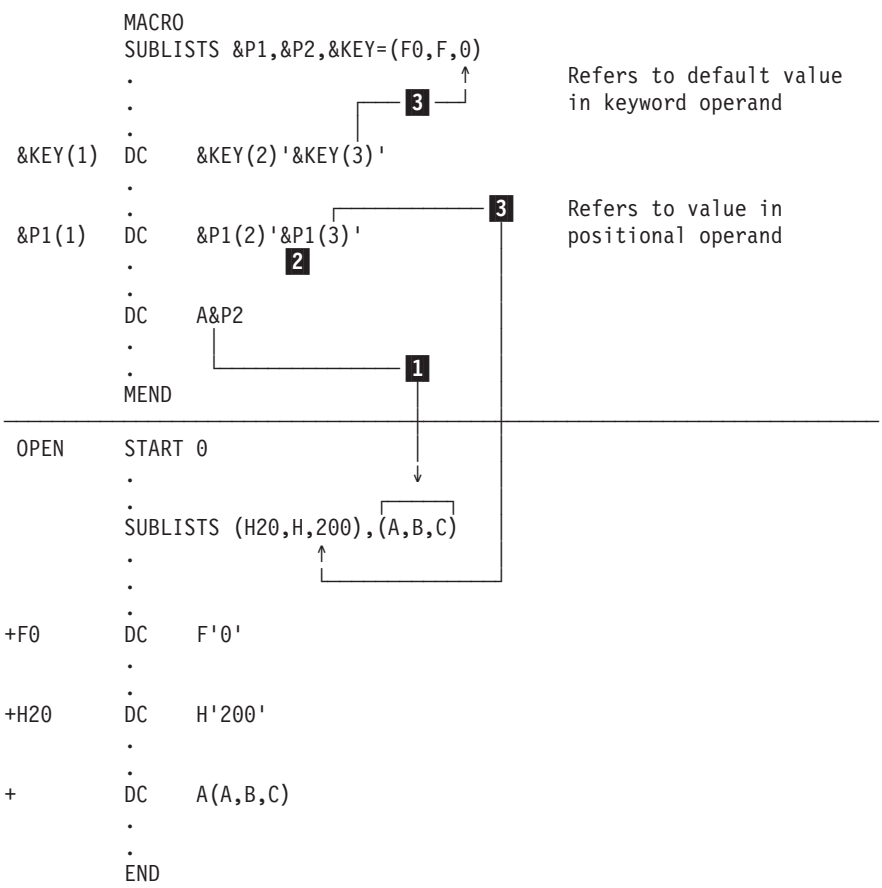


Figure 36. Sublists in operands

Table 48 shows the relationship between subscripted parameters and sublist entries if:
- A sublist entry is omitted (see **1** in Table 48).
- The subscript refers past the end of the sublist (see **2** in Table 48).
- The value of the operand is not a sublist (see **3** in Table 48).
- The parameter is not subscripted (see **4** in Table 48).

&SYSLIST(n,m): The system variable symbol, &SYSLIST(*n,m*), can also refer to sublist entries, but only if the sublist is specified in a positional operand.

Table 48. Relationship between subscripted parameters and sublist entries

Parameter	Sublist specified in corresponding operand or as default value of a keyword parameter	Value generated or used in computation
1 &PARM1(3)	(1,2,,4)	Null character string
2 &PARM1(5)	(1,2,3,4)	Null character string

Table 48. Relationship between subscripted parameters and sublist entries (continued)

Parameter	Sublist specified in corresponding operand or as default value of a keyword parameter	Value generated or used in computation
&PARM1 **3**	A	A
&PARM1(1)	A	A
&PARM1(2)	A	Null character string
4 &PARM1	(A)[1]	(A)
&PARM1(1) **2**	(A)[1]	A
&PARM1(2)	(A)[1]	Null character string
&PARM1	()[1]	()
&PARM1(1)	()[1]	Null character string
&PARM1(2)	()[1]	Null character string
&PARM1(2)	(A, ,C,D)[2]	Nothing[3]
&PARM1(1)	( )[2]	Nothing[3]
&PARM1	A,(1,2,3,4)[4]	A
&PARM2(3)	A,(1,2,3,4)[4]	3
&SYSLIST(2,3)	A,(1,2,3,4)[4]	3

Notes:

1. Considered a sublist.
2. The space indicates the end of the operand field.
3. Produces error diagnostic message `ASMA088E Unbalanced parentheses in macro call operand`.
4. Positional operands.

Multilevel sublists

You can specify multilevel sublists (sublists within sublists) in macro operands. The depth of this nesting is limited only by the constraint that the total operand length must not exceed 1024 characters. Inner elements of the sublists are referenced using additional subscripts on symbolic parameters or on &SYSLIST.

N'&SYSLIST(*n*) gives the number of operands in the indicated *n*-th level sublist. The number attribute (N') and a parameter name with an *n*-element subscript array gives the number of operands in the indicated (*n*+1)-th operand sublist. Table 49 shows the value of selected elements if &P is the first positional parameter, and the value assigned to it in a macro instruction is (A,(B,(C)),D).

Table 49. Multilevel sublists

Selected Elementsfrom &P	Selected Elementsfrom &SYSLIST	Value ofSelected Element
&P	&SYSLIST(1)	(A,(B,(C)),D)
&P(1)	&SYSLIST(1,1)	A
&P(2)	&SYSLIST(1,2)	(B,(C))
&P(2,1)	&SYSLIST(1,2,1)	B
&P(2,2)	&SYSLIST(1,2,2)	(C)
&P(2,2,1)	&SYSLIST(1,2,2,1)	C
&P(2,2,2)	&SYSLIST(1,2,2,2)	null
N'&P(2,2)	N'&SYSLIST(1,2,2)	1
N'&P(2)	N'&SYSLIST(1,2)	2
N'&P(3)	N'&SYSLIST(1,3)	1
N'&P	N'&SYSLIST(1)	3

Passing sublists to inner macro instructions

You can pass a suboperand of an outer macro instruction sublist as a sublist to an inner macro instruction. However, if you specify the COMPAT(SYSLIST) assembler option, a sublist assigned to a variable symbol is treated as a character string, not as a sublist.

Values in operands

You can use a macro instruction operand to pass a value into the called macro definition. The two types of value you can pass are:

- Explicit values or the actual character strings you specify in the operand
- Implicit values, or the attributes inherent in the data represented by the explicit values

The explicit value specified in a macro instruction operand is a character string that can contain zero or more variable symbols.

The character string must not be greater than 1024 characters after substitution of values for any variable symbols. This includes a character string that constitutes a sublist.

The character string values in the operands, including sublist entries, are assigned to the corresponding parameters declared in the prototype statement of the called macro definition. A sublist entry is assigned to the corresponding subscripted parameter.

Omitted operands

When a keyword operand is omitted, the default value specified for the corresponding keyword parameter is the value assigned to the parameter. When a positional operand or sublist entry is omitted, the null character string is assigned to the parameter.

Notes:

1. Spaces appearing between commas (without surrounding apostrophes) do not signify an omitted positional operand or an omitted sublist entry; they indicate the end of the operand field.
2. Adjacent commas indicate omission of positional operands; no comma is needed to indicate omission of the last or only positional operand.

The following example shows a macro instruction preceded by its corresponding prototype statement. The macro instruction operands that correspond to the third and sixth operands of the prototype statement are omitted in this example.

```
EXAMPLE         &A,&B,&C,&D,&E,&F           macro prototype
EXAMPLE         17,*+4,,AREA,FIELD(6)       macro instruction
```

Unquoted operands

The assembler normally retains the case of unquoted macro operands. However, to maintain uppercase alphabetic character set compatibility with earlier assemblers, High Level Assembler provides the COMPAT(MACROCASE) assembler option. When you specify this option, the assembler internally converts lowercase alphabetic characters (`a` through `z`) in unquoted macro instruction operands to uppercase alphabetic characters (`A` though `Z`), before macro expansion begins.

Special characters

Any of the 256 characters of the EBCDIC character set can appear in the value of a macro instruction operand (or sublist entry). However, the following characters require special consideration:

Ampersands

A single ampersand indicates the presence of a variable symbol. The assembler substitutes the value of the variable symbol into the character string specified in a macro instruction operand. The resultant string is then the value passed into the macro definition. If the variable symbol is undefined, an error message is issued.

Double ampersands must be specified if a single ampersand is to be passed to the macro definition.

Examples:

```
&VAR
&A+&B+3+&C*10
'&MESSAGE'
&&REGISTER
```

Apostrophes

An apostrophe is used:

- To indicate the beginning and end of a quoted string
- In a length, type, integer, opcode, or scale attribute reference notation that is not within a quoted string

Examples:

```
'QUOTED STRING'
L'SYMBOL
T'SYMBOL
```

Shift-out (SO) and shift-in (SI)

If the DBCS assembler option is specified, then SO (X'0E') and SI (X'0F') are recognized as shift codes. SO and SI delimit the start and end of double-byte data.

Quoted strings and character strings

A “quoted string” is any sequence of characters that begins and ends with an apostrophe (compare with conditional assembly character expressions described in “Character (SETC) expressions” on page 328).

To include one or more apostrophes or substituted apostrophes within the string (inside the delimiting apostrophes) two apostrophes must be specified for each apostrophe.

A “character string” is a sequence of characters that is not delimited with apostrophes.

Quoted strings can contain double-byte data, if the DBCS assembler option is specified. The double-byte data must be bracketed by the SO and SI delimiters. Only valid double-byte data is recognized between the SO and SI. The SI must be in any odd-numbered byte position after the SO. If the end of the operand is reached before SI is found, then error `ASMA203E Unbalanced double-byte delimiters` is issued.

Macro instruction operands can have values that include one or more quoted strings. Each quoted string can be separated from the following quoted string by one or more characters, and each must contain an even number of apostrophes.

Examples:

```
''
'L''SYMBOL'
'QUOTE1'AND'QUOTE2'
A'B'C
```

Attribute reference notation

You can specify an attribute reference notation as a macro instruction operand value. The attribute reference notation must be preceded by a space or any other special character except the ampersand and the apostrophe. See “Data attributes” on page 284 for details about data attributes, and the format of attribute references.

Examples:

```
MAC1          L'SYMBOL,10+L'AREA*L'FIELD
MAC1          I'PACKED-S'PACKED
```

Parentheses

In macro instruction operand values, there must be an equal number of left and right parentheses. They must be paired, that is, each left parenthesis needs a following right parenthesis at the same level of nesting. An unpaired (single) left or right parenthesis can appear only in a quoted string.

Examples:

```
(PAIRED-PARENTHESES)
()
(A(B)C)D(E)
(IN'('STRING)
```

Spaces

One or more spaces outside a quoted string indicates the end of the operands of a macro instruction. If this is not your intention, place the spaces inside quoted strings.

Example:

```
'SPACES ALLOWED'
```

Commas

A comma outside a quoted string indicates the end of an operand value or sublist entry. Commas that do not delimit values can appear inside quoted strings or paired parentheses that do not enclose sublists.

Examples:

```
A,B,C,D
(1,2)3'5,6'
```

Equal signs

An equal sign can appear in the value of a macro instruction operand or sublist entry:

- As the first character
- Inside quoted strings
- Between paired parentheses
- In a keyword operand
- In a positional operand, provided the parameter does not resemble a keyword operand

Examples:

```
=H'201'
A'='B
C(A=B)
2X=B
KEY=A=B
```

The assembler issues a warning message for a positional operand containing an equal sign, if the operand resembles a keyword operand. Thus, if we assume that this is the prototype of a macro definition:

```
MAC1          &F
```

then this macro instruction generates a warning message:

```
MAC1          K=L  (K appears to be a valid keyword)
```

while this macro instruction does not:

```
MAC1          2+2=4  (2+2 is not a valid keyword)
```

Periods

A period (.) can be used in the value of an operand or sublist entry. It is passed as a period. However, if it is used immediately after a variable symbol, it becomes a concatenation character. Two periods are required if one is to be passed as a character.

Examples:

```
3.4
&A.1
&A..1
```

Nesting macro instruction definitions

A nested macro instruction definition is a macro instruction definition you can specify as a set of model statements in the body of an enclosing macro definition. This lets you create a macro definition by expanding the outer macro that contains the nested definition.

All nested inner macro definitions are effectively "black boxes": there is no visibility to the outermost macro definition of any variable symbol or sequence symbol within any of the nested macro definitions. This means that you cannot use an enclosing macro definition to tailor or parameterize the contents of a nested inner macro definition.

High Level Assembler allows both inner macro instructions and inner macro definitions. The inner macro definition is not edited until the outer macro is generated as the result of a macro instruction calling it, and then only if the inner macro definition is encountered during the generation of the outer macro. If the outer macro is not called, or if the inner macro is not encountered in the generation of the outer macro, the inner macro definition is never edited. Figure 37 shows the editing of inner macro definitions.

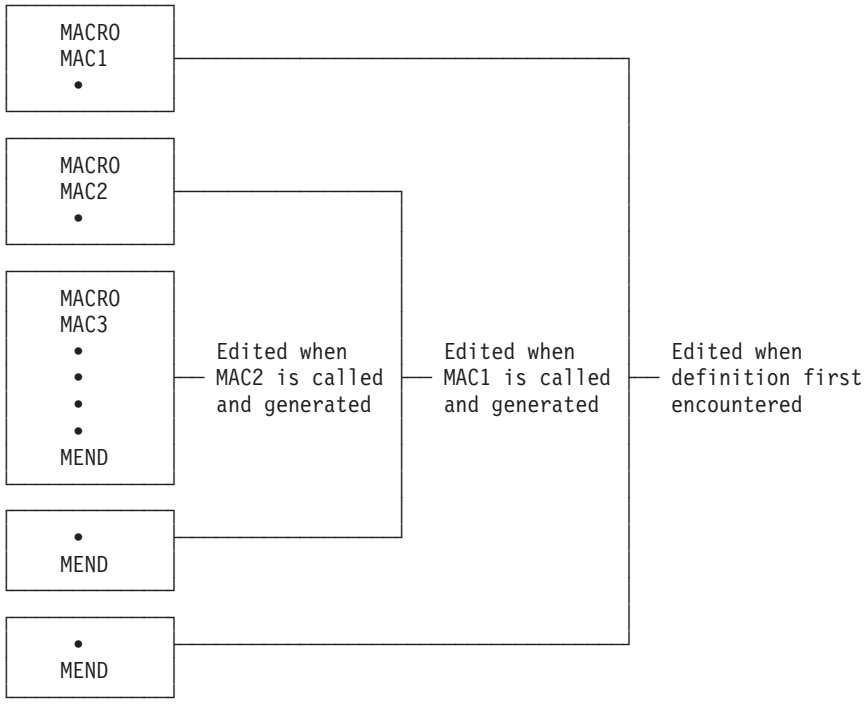


Figure 37. Editing inner macro definitions

First MAC1 is edited, and MAC2 and MAC3 are not. When MAC1 is called, MAC2 is edited (unless its definition is bypassed by an AIF or AGO branch); when MAC2 is called, MAC3 is edited. No macro can be called until it has been edited.

There is no limit to the number of nestings allowed for inner macro definitions.

The lack of parameterization can be overcome in some cases by using the AINSERT statement. This lets you generate a macro definition from within another macro generation. A simple example is shown at "Where to define a macro in a source module" on page 213. In Figure 38, macro ainsert_test_macro generates the macro mac1 using a combination of AINSERT and AREAD instructions. The mac1 macro is then called with a list of seven parameters.

```
 1          macro
 2 &name    ainsert_test_macro
 3          ainsert   '         Macro',back
 4          ainsert   '         mac1',back
 5          ainsert   'Blah blah blah',front
 6 &aread   aread
 7 &aread   setc   '&aread'(1,10)
 8          ainsert   '&&n      seta n''&&syslist ',back
 9          ainsert   '         dc a(&&n)',back
10          ainsert   '         dc c''&aread'' ',back
11          ainsert   '         mend',back
12          mend
13 *
14 testains csect 0
15 *
16          ainsert_test_macro
17+         ainsert   '         Macro',back
18+         ainsert   '         mac1',back
19+         ainsert   'Blah blah blah',front
20-Blah blah blah
21+         ainsert   '&&n       seta n''&&syslist ',back
22+         ainsert   '         dc a(&&n)',back
23+         ainsert   '         dc c''Blah blah '' ',back
24+         ainsert   '         mend',back
25>         Macro
26>         mac1
27>&n       seta n'&syslist
28>         dc a(&n)
29>         dc c'Blah blah '
30>         mend
31 *
32          mac1  a,b,c,d,e,f,g
33+         dc a(7)
34+         dc c'Blah blah '
35 *
36          end
```

Figure 38. Expanding nested macro definitions

Inner and outer macro instructions

Any macro instruction you write in the open code of a source module is an *outer macro instruction* or call. Any macro instruction that appears within a macro definition is an *inner macro instruction* or call.

Levels of macro call nesting

The code generated by a macro definition called by an inner macro call is nested inside the code generated by the macro definition that contains the inner macro call. In the macro definition called by an inner macro call, you can include a macro call to another macro definition. Thus, you can nest macro calls at different levels.

The &SYSNEST system variable indicates how many levels you called. It has the value 1 in an outer macro, and is incremented by one at a macro call.

Recursion

You can also call a macro definition recursively; that is, you can write macro instructions inside macro definitions that are calls to the containing definition. This is how you define macros to process recursive functions.

General rules and restrictions

Macro instruction statements can be written inside macro definitions. Values are substituted in the same way as they are for the model statements of the containing macro definition. The assembler processes the called macro definition, passing to it the operand values (after substitution) from the inner macro instruction. In addition to the operand values described in "Values in operands" on page 269, nested macro calls can specify values that include:

- Any of the symbolic parameters (see **1** in Figure 39) specified in the prototype statement of the containing macro definition
- Any SET symbols (see **2** in Figure 39) declared in the containing macro definition
- Any of the system variable symbols, such as &SYSDATE or &SYSTIME, (see **3** in Figure 39).

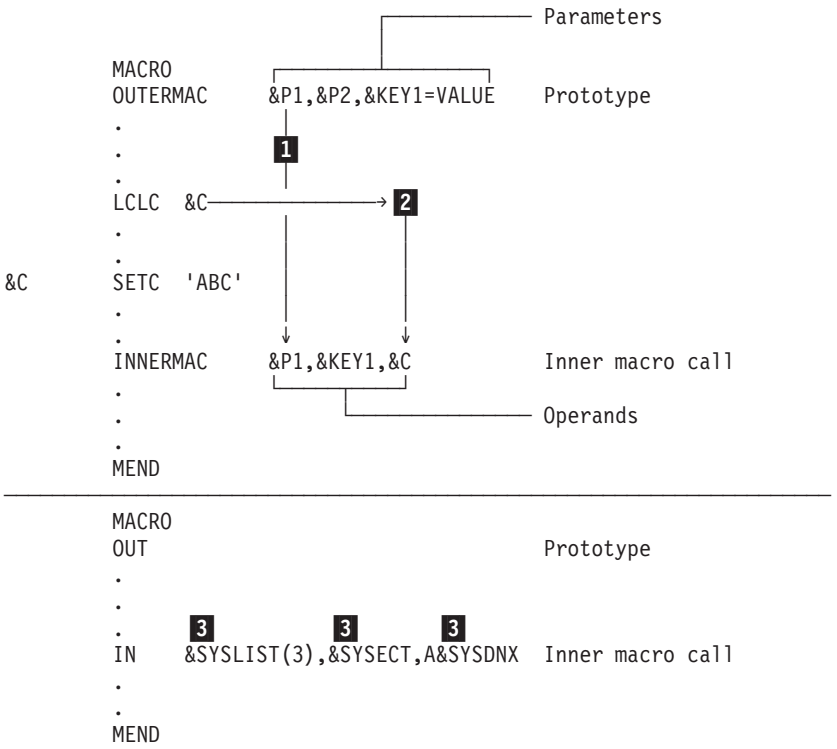


Figure 39. Values in nested macro calls

The number of nesting levels permitted depends on the complexity and size of the macros at the different levels; that is, the number of operands specified, the number of local-scope and global-scope SET symbols declared, and the number of sequence symbols used.

When the assembler processes a macro exit instruction, either MEXIT or MEND, it selects the next statement to process depending on the level of nesting. If the macro exit instruction is from an inner macro, the assembler processes the next statement after the statement that called the outer macro. The next statement in open code might come from the AINSERT buffer. If the macro exit instruction is from an outer macro, the assembler processes the next statement in open code, after the statement that called the outer macro.

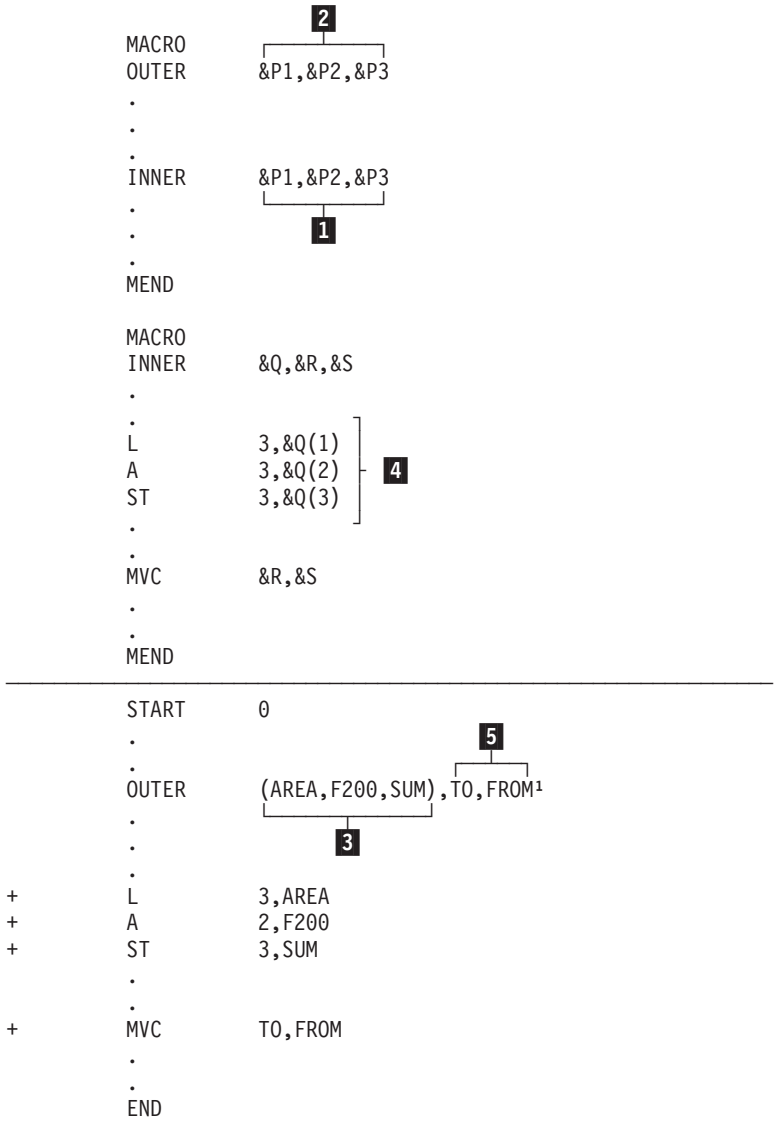
Passing values through nesting levels

The value contained in an outer macro instruction operand can be passed through one or more levels of nesting (see Figure 40 on page 276). However, the value specified (see 1 in Figure 40 on page 276) in the inner macro instruction operand must be identical to the corresponding symbolic parameter (see 2 in Figure 40 on page 276) declared in the prototype of the containing macro definition.

Thus, a sublist can be passed (see 3 in Figure 40 on page 276) and referred to (see 4 in Figure 40 on page 276) as a sublist in the macro definition called by the inner macro call. Also, any symbol (see 5 in Figure 40 on page 276) that is passed carries its attribute values through the nesting levels.

If inner macro calls at each level are specified with symbolic parameters as operand values, values can be passed from open code through several levels of macro nesting.

COMPAT(SYSLIST) Assembler Option: If the COMPAT(SYSLIST) assembler option is specified, and a symbolic parameter is only a part of the value specified in an inner macro instruction operand, only the character string value given to the parameter by an outer call is passed through the nesting level. Inner sublist entries are, therefore, not available for reference in the inner macro.



Note:

1. The following inner macro call statement is generated, but not listed unless the PCONTROL(MCALL) option is specified, or the assembler instruction ACONTROL MCALL is active:

```
INNER (AREA,F200,SUM),TO,FROM
```

Figure 40. Passing values through nesting levels

System variable symbols in nested macros

The fixed global-scope system variable symbols (see "System variable symbols" on page 229) are not affected by the nesting of macros. The variable global-scope system variable symbols have values which might change during the expansion of a macro definition. The following system variable is influenced by nested macros:

&SYSM_SEV
Provides the highest MNOTE severity code from the nested macro most recently called.

The local system variable symbols are given read-only values each time a macro definition is called.

The following system variable symbols can be affected by the position of a macro instruction in code or the operand value specified in the macro instruction:

&SYSCLOCK
The assembler assigns &SYSCLOCK the constant string value representing the TOD clock value at the time at which a macro call is made. The time portion of this value is precise to the microsecond. For any inner macro call, the value assigned to &SYSCLOCK differs from that of its parent.

&SYSECT
The assembler gives &SYSECT the character string value of the name of the control section in use at the point at which a macro call is made. For a macro definition called by an inner macro call, the assembler assigns to &SYSECT the name of the control section in effect in the macro definition that contains the inner macro call, at the time the inner macro is called.

If no control section is generated within a macro definition, the value assigned to &SYSECT does not change. It is the same for the next level of macro definition called by an inner macro instruction.

&SYSLIB_DSN, &SYSLIB_MEMBER, &SYSLIB_VOLUME
The assembler assigns the character string value of the &SYSLIB system variable symbols at the point at which a macro is called. For an inner macro call whose definition is from a library member, these values might differ, if this is the first time this macro is invoked.

&SYSLIST
If &SYSLIST is specified in a macro definition called by an inner macro instruction, &SYSLIST refers to the positional operands of the inner macro instruction.

&SYSLOC
The assembler gives &SYSLOC the character string value of the name of the location counter in use at the point at which a macro is called. For a macro definition called by an inner macro call, the assembler assigns to &SYSLOC the name of the location counter in effect in the macro definition that contains the inner macro call. If no LOCTR or control section is generated within a macro definition, the value assigned to &SYSLOC does not change. It is the same for the next level of macro definition called by an inner macro instruction.

&SYSNDX
The assembler increments &SYSNDX by one each time it encounters a macro call. It retains the incremented value throughout the expansion of the macro definition called, that is, within the local scope of the nesting level.

&SYSNEST
The assembler increments &SYSNEST by one each time it encounters a nested macro instruction. It retains the incremented value within the local scope of the macro definition called by the inner macro instruction. Subsequent nested macro instructions cause &SYSNEST to be incremented by 1. When the assembler exits from a nested macro it decreases the value in &SYSNEST by 1.

&SYSSEQF
The assembler assigns &SYSSEQF the character string value of the identification-field of the outer-most macro instruction statement. The value of &SYSSEQF remains constant throughout the expansion of the called macro definition and all macro definitions called from within the outer macro.

&SYSSTYP
The assembler gives &SYSSTYP the character string value of the type of the control section in use at the point at which a macro is called. For a macro definition called by an inner macro call, the assembler assigns to &SYSSTYP the type of the control section in effect in the macro definition that contains the inner macro call, at the time the inner macro is called.

If no control section is generated within a macro definition, the value assigned to &SYSSTYP does not change. It is the same for the next level of macro definition called by an inner macro instruction.

Chapter 9. How to write conditional assembly instructions

This chapter describes the conditional assembly language. With the conditional assembly language, you can carry out general arithmetic and logical computations, and many of the other functions you can carry out with any other programming language. Also, by writing conditional assembly instructions in combination with other assembler language statements, you can:

- Select sequences of these source statements, called *model statements*, from which machine and assembler instructions are generated
- Vary the contents of these model statements during generation

The assembler processes the instructions and expressions of the conditional assembly language during conditional assembly processing. Then, at assembly time, it processes the generated instructions. Conditional assembly instructions, however, are not processed after conditional assembly processing is completed.

The conditional assembly language is more versatile when you use it to interact with symbolic parameters and the system variable symbols inside a macro definition. However, you can also use the conditional assembly language in open code; that is, code that is not within a macro definition.

Elements and functions

The elements of the conditional assembly language are:

- SET symbols that represent data. See "SET symbols."
- Attributes that represent different characteristics of symbols. See "Data attributes" on page 284.
- Sequence symbols that act as labels for branching to statements during conditional assembly processing. See "Sequence symbols" on page 298.

The functions of the conditional assembly language are:

- Declaring SET symbols as variables for use locally and globally in macro definitions and open code. See "Declaring SET symbols" on page 302.
- Assigning values to the declared SET symbols. See "Assigning values to SET symbols" on page 305.
- Selecting characters from strings for substitution in, and concatenation to, other strings; or for inspection in condition tests. See "Substring notation" on page 328.
- Branching and exiting from conditional assembly loops. See "Branching" on page 344.

The conditional assembly language can also be used in open code with few restrictions. See "Open code" on page 301.

The conditional assembly language provides instructions for evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, and as condition tests for branching. See "Conditional assembly instructions" on page 302 for details about the syntax and usage rules of each instruction.

SET symbols

SET symbols are variable symbols that provide you with arithmetic, binary, or character data, and whose values you can vary during conditional assembly processing.

Use SET symbols as:

- Terms in conditional assembly expressions

- Counters, switches, and character strings
- Subscripts for variable symbols
- Values for substitution

Thus, SET symbols let you control your conditional assembly logic, and to generate many different statements from the same model statement.

Subscripted SET symbols

You can use a SET symbol to represent a one-dimensional array of many values. You can then refer to any one of the values of this array by subscripting the SET symbol. For more information, see "Subscripted SET symbol specification" on page 283.

Scope of SET symbols

The scope of a SET symbol is that part of a program for which the SET symbol has been declared. Local SET symbols need not be declared by explicit declarations. The assembler considers any undeclared variable symbol found in the name field of a SETx instruction as a local SET symbol.

If you declare a SET symbol to have a local scope, you can use it only in the statements that are part of either:
- The same macro definition, or
- Open code

If you declare a SET symbol to have a global scope, you can use it in the statements that are part of any one of:
- The same macro definition
- A different macro definition
- Open code

You must, however, declare the SET symbol as global for each part of the program (a macro definition or open code) in which you use it.

You can change the value assigned to a SET symbol without affecting the scope of this symbol.

Scope of symbolic parameters

A symbolic parameter has a local scope. You can use it only in the statements that are part of the macro definition for which the parameter is declared. You declare a symbolic parameter in the prototype statement of a macro definition.

The scope of system variable symbols is described in Table 50 on page 281.

SET symbol specifications

SET symbols can be used in model statements, from which assembler language statements are generated, and in conditional assembly instructions. The three types of SET symbols are: SETA, SETB, and SETC. A SET symbol must be a valid variable symbol.

The rules for creating a SET symbol are:
- The first character must be an ampersand (&)
- The second character must be an alphabetic character
- The remaining characters must be 0 to 61 alphanumeric
- Do not set the first four characters to &SYS, which is used for system variable symbols

Examples:

```
&ARITHMETICVALUE439
&BOOLEAN
&C
&EASY_TO_READ
```

Local SET symbols need not be declared by explicit declarations. The assembler considers any undeclared variable symbol found in the name field of a SETx instruction as a local SET symbol, and implicitly declares it to have the type specified by the SETx instruction. The instruction that declares a SET symbol determines its scope and type.

The features of SET symbols and other types of variable symbols are compared in Table 50.

Table 50. Features of SET symbols and other types of variable symbols

Features	SETA, SETB, SETC symbols	Symbolic Parameters	System Variable Symbols
Can be used in:			
Open code	Yes	No	&SYSASM &SYSDATC &SYSDATE &SYSJOB &SYSM_HSEV &SYSM_SEV &SYSOPT_DBCS &SYSOPT_OPTABLE &SYSOPT_RENT &SYSOPT_XOBJECT &SYSPARM &SYSSTEP &SYSSTMT &SYSTEM_ID &SYSTIME &SYSVER
Macro definitions	Yes	Yes	All

Table 50. Features of SET symbols and other types of variable symbols (continued)

Features	SETA, SETB, SETC symbols	Symbolic Parameters	System Variable Symbols
Scope:			
Local	Yes	Yes	&SYSADATA_DSN &SYSADATA_MEMBER &SYSADATA_VOLUME &SYSCLOCK &SYSECT &SYSIN_DSN &SYSIN_MEMBER &SYSIN_VOLUME &SYSLIB_DSN &SYSLIB_MEMBER &SYSLIB_VOLUME &SYSLIN_DSN &SYSLIN_MEMBER &SYSLIN_VOLUME &SYSLIST &SYSLOC &SYSMAC &SYSNDX &SYSNEST &SYSPRINT_DSN &SYSPRINT_MEMBER &SYSPRINT_VOLUME &SYSPUNCH_DSN &SYSPUNCH_MEMBER &SYSPUNCH_VOLUME &SYSSEQF &SYSTERM_DSN &SYSTERM_MEMBER &SYSTERM_VOLUME
Global	Yes	No	&SYSASM &SYSDATC &SYSDATE &SYSJOB &SYSM_HSEV &SYSM_SEV &SYSOPT_DBCS &SYSOPT_OPTABLE &SYSOPT_RENT &SYSOPT_XOBJECT &SYSPARM &SYSSTEP &SYSSTMT &SYSTEM_ID &SYSTIME &SYSVER
Values can be changed within scope of symbol	Yes[1]	No, read only value[2]	No, read only value[2]

Table 50. Features of SET symbols and other types of variable symbols (continued)

Features	SETA, SETB, SETC symbols	Symbolic Parameters	System Variable Symbols

Notes:

1. The value assigned to a SET symbol can be changed by using the SETA, SETAF, SETB, SETC, or SETCF instruction within the declared or implied scope of the SET symbol.
2. A symbolic parameter and the system variable symbols (except for &SYSSTMT, &SYSM_HSEV, and &SYSM_SEV) are assigned values that remain fixed throughout their scope. Wherever a SET symbol appears in a statement, the assembler replaces the symbol's current value with the value assigned to it.

SET symbols can be used in the name, operation, and operand fields of macro instructions. The value thus passed through the name field symbolic parameter into a macro definition is considered as a character string and is generated as such. If the COMPAT(SYSLIST) assembler option is specified, the value passed through an operand field symbolic into a macro definition is also considered a character string and is generated as such. However, if the COMPAT(SYSLIST) assembler option is not specified, SET symbols can be used to pass sublists into a macro definition.

Subscripted SET symbol specification

Here is the format of a subscripted SET symbol:

&symbol
Is a variable symbol.

subscript
Is an arithmetic expression with a value greater than or equal to 1.

Example:

```
&ARRAY(20)
```

The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (see "Arithmetic (SETA) expressions" on page 311).

The subscript refers to one of the many positions in an array of values identified by the SET symbol.

A subscripted SET symbol can be used anywhere an unsubscripted SET symbol is allowed. However, subscripted SET symbols must be declared as subscripted by a previous local or global declaration instruction, or implicitly as a local subscripted SET symbol in a SETx instruction of the desired type.

The dimension (the maximum value of the subscript) of a subscripted SET symbol is not determined by the explicit or implicit declaration of the symbol. The dimension specified can be exceeded in later SETx instructions. Note, however, that increasing the dimension of a subscripted SET symbol also increases the storage required. For example, referencing only &ARRAY(1000000) still causes the preceding 999999 elements to be allocated. You can determine the maximum subscript using the N' attribute (see "Number attribute (N')" on page 295).

The subscript can be a subscripted SET symbol.

Created SET symbols

The assembler can create SET symbols during conditional assembly processing from other variable symbols and character strings. A SET symbol thus created has the form &(*e*), where *e* represents one or more of these:

- Variable symbols, optionally subscripted
- Strings of alphanumeric characters
- Other created SET symbols

After substitution and concatenation, *e* must consist of a string of up to 62 alphanumeric characters, the first of which is alphabetic. The assembler considers the preceding ampersand and this string as the name of a SET variable. If this created SET symbol has the same name as an existing SET symbol, they are treated as identical. If this created SET symbol does not have the name of any existing SET symbol, the usual rules for assigning type and scope apply.

You can use created SET symbols wherever ordinary SET symbols are permitted, including declarations. A created SET symbol must not match the name of a system variable symbol, nor the name of a symbolic parameter in a macro prototype statement. You can also nest created SET symbols in other created SET symbols.

Consider the following example:

```
&ABC(1)  SETC            'MKT','27','$5'
```

Let &(*e*) equal `&(&ABC(&I)QUA&I)`.

```
&I  &ABC(&I)  Created SET Symbol  Comment

1   MKT       &MKTQUA1            Valid
2   27        &27QUA2             Invalid: character after '&' not alphabetic
3   $5        &$5QUA3             Valid
4             &QUA4               Valid
```

The name of a created SET symbol cannot match the name of a system variable symbol or of a symbolic parameter in a macro definition.

The created SET symbol can be thought of as a form of indirect addressing. With nested created SET symbols, you can perform this kind of indirect addressing to any level.

In another sense, created SET symbols offer an associative storage facility. For example, a symbol table of numeric attributes can be referred to by an expression of the form `&(&SYM)(&I)` to yield the `I`th attribute of the symbol name in `&SYM`. As this example indicates, created SET symbols can be declared and used as arrays of dimensioned variables.

Created SET symbols also enable you to achieve some of the effect of multiple-dimensioned arrays by creating a separate name for each element of the array. For example, a 3-dimensional array of the form `&X(&I,&J,&K)` might be addressed as `&(X&I.$&J.$&K)`, where &I, &J, and &K typically have numeric values. Thus, `&X(2,3,4)` is represented by `&X2$3$4`. The `$` separators guarantee that `&X(2,33,55)` and `&X(23,35,5)` are unique:

```
&X(2,33,55) becomes &X2$33$55
&X(23,35,5) becomes &X23$35$5
```

Data attributes

The data, such as instructions, constants, and areas, that you define in a source module, can be described by its:

- Type, which distinguishes a property of a named object or macro argument, for example, fixed-point constants from floating-point constants, or machine instructions from macro instructions

- Length, which gives the number of bytes occupied by the object code of the named data
- Scaling, which shows the number of positions occupied by the fractional portion of named fixed-point, floating-point, and decimal constants in their object code form
- Integer, which shows the number of positions occupied by the integer portion of named fixed-point and decimal constants in their object code form
- Count, which gives the number of characters that are required to represent the named data, such as a macro instruction operand, as a character string
- Number, which gives the number of sublist entries in a macro instruction operand
- Defined, which determines whether a symbol has been defined prior to the point where the attribute reference is coded
- Operation Code, which shows if an operation code, such as a macro definition or machine instruction, is defined prior to the point where the attribute reference is coded

These characteristics are called the attributes of the symbols naming the data. The assembler assigns attribute values to the ordinary symbols and variable symbols that represent the data.

Specifying attributes in conditional assembly instructions allows you to control conditional assembly logic, which, in turn, can control the sequence and contents of the statements generated from model statements. The specific purpose for which you use an attribute depends on the kind of attribute being considered. Here are the attributes and their main uses:

Table 51. Data attributes

Attribute	Purpose	Main Uses
Type	Gives a letter that identifies type of data represented	• In tests to distinguish between different data types • For value substitution • In macros to discover missing operands
Length	Gives number of bytes that data occupies in storage	• For substitution into length fields • For computation of storage requirements
Scaling	Refers to the position of the decimal point in fixed-point, floating-point, and decimal constants	• For testing and regulating the position of decimal points • For substitution into a scale modifier
Integer	Is a function of the length and scale attributes of decimal, fixed-point, and floating-point constants	• To keep track of significant digits (integers)
Count	Gives the number of characters required to represent data	• For scanning and decomposing character strings • As indexes in substring notation
Number[1]	Gives the number of sublist entries in a macro instruction operand sublist, or the maximum subscript of a dimensioned SET symbol to which a value has been assigned.	• For scanning sublists • As a counter to test for end of sublist • For testing array limits
Defined	Shows whether the symbol referenced has been defined prior to the attribute reference	• To avoid defining a symbol again if the symbol referenced has been previously defined
Operation Code	Shows whether a given operation code has been defined prior to the attribute reference	• To avoid assembling a macro or instruction if it does not exist.

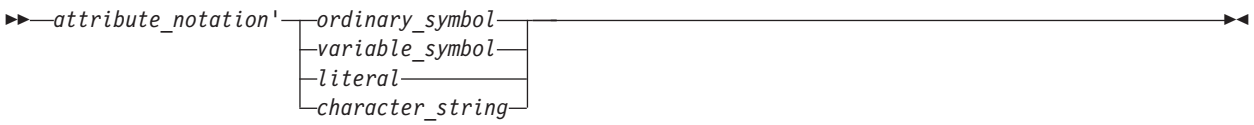
Table 51. Data attributes (continued)

Attribute	Purpose	Main Uses

Notes:

1. The number attribute of &SYSLIST(*n*) and &SYSLIST(*n,m*) is described in "&SYSLIST System Variable Symbol" on page 241.

Attribute reference



attribute_notation'
Is the attribute whose value you want, followed by a apostrophe. Valid attribute letters are "D", "O", "N", "S", "K", "I", "L", and "T".

ordinary_symbol
Is an ordinary symbol that represents the data that possesses the attribute. An ordinary symbol cannot be specified with the operation code attribute.

variable_symbol
Is a variable symbol that represents the data that possesses the attribute.

literal
Is a literal that represents the data that possesses the attribute. A literal cannot be specified with the operation code attribute or count attribute.

character_string
Is a character string that represents the operation code in the operation code attribute.

Examples:

```
T'SYMBOL
L'&VAR
K'&PARAM
O'MVC
S'=P'975.32'
```

The assembler substitutes the value of the attribute for the attribute reference.

Reference to the count (K'), defined (D'), number (N'), operation code (O'), and type (T') attributes can be used only in conditional assembly instructions or within macro definitions. The length (L'), integer (I'), and scale (S') attribute references can be in conditional assembly instructions, machine instructions, assembler instructions, and the operands of macro instructions.

Attributes of symbols and expressions

Table 52 on page 287 shows attribute references (in the columns) and types of symbols (in the rows). Each intersection shows whether ("Yes") or not ("No") you can validly apply the attribute reference to that symbol type, or (for SET symbols) to the value of the symbol.

Table 52. Attributes and related symbols

Symbols Specified	Type T'	Length L'	Scale S'	Integer I'	Count K'	Number N'	Defined D'	Operation Code O'
In open code:								
Ordinary symbols	Yes	Yes	Yes	Yes	No	No	Yes	No
System variable symbols with global scope	Yes	No	No	No	Yes	Yes	No	No
Literals in macro instruction operands	Yes	Yes	Yes	Yes	No	No	Yes	No
In macro definitions:								
Ordinary symbols	Yes	Yes	Yes	Yes	No	No	Yes	No
Symbolic parameters	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
System variable symbols:								
&SYSLIST	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
All others	Yes	No	No	No	Yes	Yes	No	No
Literals in macro instruction operands	Yes	Yes	Yes	Yes	No	No	Yes	No

The values of attribute references can be used in ordinary and conditional assembly expressions, as shown in Table 53.

Table 53. Using attribute values

Symbols Specified	Type T'	Length L'	Scale S'	Integer I'	Count K'	Number N'	Defined D'	Operation Code O'
In open code: SET symbols	SETB[1], SETC	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETB[1], SETC
In ordinary assembly:	No	Yes	Yes	Yes	No	No	No	No
In macro definitions: SET symbols	SETB[1], SETC	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETA, SETB[2]	SETB[1], SETC

Table 53. Using attribute values (continued)

Symbols Specified	Type T'	Length L'	Scale S'	Integer I'	Count K'	Number N'	Defined D'	Operation Code O'

Notes:

1. Only in character relations.
2. Only in arithmetic relations.

The value of an attribute for an ordinary symbol specified in an attribute reference comes from the item named by the symbol. The symbol must appear in the name field of an assembler or machine instruction, or in the operand field of an EXTRN or WXTRN instruction.

The value of an attribute reference to an expression is the value of that attribute reference to its leftmost term.

Notes:

1. You cannot refer to the names of instructions generated by conditional assembly substitution or macro generation until the instruction is generated.
2. If you use a symbol qualifier to qualify an ordinary symbol in an attribute reference, the qualifier is ignored.

The value of an attribute for a variable symbol specified in an attribute reference comes from the value substituted for the variable symbol as follows:

SET Symbols and System Variable Symbols
For SET symbols and all system variable symbols other than &SYSLIST, the attribute values come from the current value of these symbols.

Symbolic Parameters and &SYSLIST
For symbolic parameters and the system variable symbol, &SYSLIST, the values of the count and number attributes come from the operands of macro instructions. The name field entry of the call is an "operand", and is referenced as &SYSLIST(0). The values of the type, length, scale, and integer attributes, however, come from the values represented by the macro instruction operands, as follows:

1. If the operand is a sublist, the entire sublist and each entry of the sublist can possess attributes. The whole sublist has the same attributes as those of the first suboperand in the sublist (except for the count attribute, which can be different, and the number attribute which is relevant only for the whole sublist).
2. If the first character or characters of the operand (or sublist entry) constitute an ordinary symbol, and this symbol is followed by either an arithmetic operator (+, -, *, or /), a left parenthesis, a comma, or a space, then the value of the attributes for the operand are the same as for the ordinary symbol.
3. If the operand (or sublist entry) is a character string other than a sublist or the character string described in the previous point, the type attribute is undefined (U) and the length, scale, and integer attributes are invalid.

Because the count (K'), number (N'), and defined (D') attribute references are allowed only in conditional assembly instructions, their values are available only during conditional assembly processing. They are not available at ordinary assembly time.

The system variable symbol &SYSLIST, with a valid subscript, can be used in an attribute reference to refer to a macro instruction operand, and, in turn, to an ordinary symbol. Thus, any of the attribute values for macro instruction operands and ordinary symbols in the following subsections can also be substituted for an attribute reference containing &SYSLIST (see "&SYSLIST System Variable Symbol" on page 241).

Type attribute (T')

The type attribute has a value of a single alphabetic character that shows the type of data represented by:
- An ordinary symbol
- A macro instruction operand
- A SET symbol
- A literal
- A system variable symbol

The type attribute can change during an assembly. The lookahead search might assign one attribute, whereas the symbol table at the end of the assembly might display another.

The type attribute reference can be used in the operand field of a SETC instruction or as one of the values used for comparison in the operand field of a SETB or AIF instruction.

The type attribute can also be specified outside conditional assembly instructions. Then, the type attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

The following letters are used for the type attribute of data represented by ordinary symbols and outer macro instruction operands that are symbols that name DC or DS statements.

A A-, J-type address constant, implied length, aligned (also CXD instruction label)
B Binary constant
C Character constant
D Long floating-point constant, implicit length, aligned
E Short floating-point constant, implicit length, aligned
F Fullword fixed-point constant, implicit length, aligned
G Fixed-point constant, explicit length
H Halfword fixed-point constant, implicit length, aligned
K Floating-point constant, explicit length
L Extended floating-point constant, implicit length, aligned
P Packed decimal constant
Q Q-type address constant, implicit length, aligned
R A-, S-, Q-, J-, R-, V-, or Y-type address constant, explicit length
S S-type address constant, implicit length, aligned
V R-, V-type address constant, implicit length, aligned
X Hexadecimal constant
Y Y-type address constant, implicit length, aligned
Z Zoned decimal constant
@ Graphic (G) constant

When a literal is specified as the name field on a macro call instruction, and if the literal has previously been used in a machine instruction, the type attribute of the literal is the same as for data represented by ordinary symbols or outer macro instructions operands.

The following letters are used for the type attribute of data represented by ordinary symbols (and outer macro instruction operands that are symbols) that name statements other than DC or DS statements, or that appear in the operand field of an EXTRN or WXTRN statement:

I Machine instruction
J Control section name
M The name field on a macro instruction, when the name field is:
- A valid symbol not previously defined
- A valid literal not previously defined

T Identified as an external symbol by EXTRN instruction
W CCW, CCW0, or CCW1 instruction
$ Identified as an external symbol by WXTRN instruction

The following letter is used for the type attribute of data represented by inner and outer macro instruction operands only:

O Omitted operand (has a value of a null character string). Such an operand need not be a null string: a macro operand such as `(,,)` has a null first suboperand.

The following attribute is used for the type attribute of the value of variable symbols:

N The value is numeric

The following letter is used for symbols or macro instruction operands that cannot be assigned any of the above letters:

U Undefined, unknown, or unassigned

The common use of the U type attribute is to describe a valid symbol that has not been assigned any of the type attribute values described above. If the assembler is not able to determine what the named symbol represents, it also assigns the U type attribute. Thus, the U type attribute can mean *undefined*, or *unknown*, or *unassigned* at the time of the reference. Consider the following macro definition:

```
Name      Operation      Operand
          macro
          MAC1 &op1,&op2
&A        setc T'&op1
&B        setc T'&op2
          DC C'&A'          DC containing type attribute for op1
          DC C'&B'          DC containing type attribute for op2
          mend
```

When the macro `MAC1` is called in Figure 41, neither of the operands has previously been defined, however `GOOD_SYMBOL` is a valid symbol name, whereas `?BAD_SYMBOL?` is not a valid symbol name. The type attribute for both operands is `U`, meaning `GOOD_SYMBOL` is undefined, and `?BAD_SYMBOL?` is unknown.

When the macro `MAC1` is called in Figure 42, `GOOD_SYMBOL` is a valid symbol name, and has been

```
000000               00000 00004     8 a   csect
                                     9     mac1 GOOD_SYMBOL,?BAD_SYMBOL?
000000 E4                           10+          DC C'U'           DC containing type attribute for op1
000001 E4                           11+          DC C'U'           DC containing type attribute for op2
                                    12    end
```

Figure 41. Undefined and unknown type attributes

defined in the DC instruction at statement 12. `?BAD_SYMBOL?` is a not valid symbol name, and the assembler issues an error message at statement 13. The type attribute for `GOOD_SYMBOL` is `C`, meaning that the symbol represents a character constant. The type attribute for `?BAD_SYMBOL?` is `U`, meaning that the type is unknown.

```
000000               00000 00006     8 a   csect
                                     9     mac1 GOOD_SYMBOL,?BAD_SYMBOL?
000000 C3                           10+          DC C'C'           DC containing type attribute for op1
000001 E4                           11+          DC C'U'           DC containing type attribute for op2
000002 A9                           12 GOOD_SYMBOL  dc cl1'z'
000003 A9                           13 ?BAD_SYMBOL? dc cl1'z'
** ASMA147E Symbol too long, or first character not a letter - ?BAD_SYMBOL?
                                    14    end
```

Figure 42. Unknown type attribute for invalid symbol

The type attribute value U, meaning *undefined*, *unknown*, or *unassigned*, is assigned to the following:

- Ordinary symbols used as labels:
  - For the LTORG instruction
  - For the EQU instruction without a third operand
  - For DC and DS statements that contain variable symbols, for example, `U1 DC &X'1'`
  - That are defined more than once, even though only one instance of the label is generated due to conditional assembly statements. A lookahead scan for attributes of a symbol might encounter more than one occurrence of a symbol, in which case the assembler cannot yet tell which statements will be generated. In such cases, type attribute U is assigned. At a later time, when the symbol has been generated, its type attribute is changed to the correct value for the type of statement it names.
- SETC variable symbols that have a value other than a null character string or the name of an instruction that can be referred to be a type attribute reference
- System variable symbols except:
  - &SYSDATC, &SYSM_HSEV, &SYSM_SEV, &SYSNDX, &SYSNEST, &SYSOPT_DBCS, &SYSOPT_RENT, &SYSOPT_XOBJECT, and &SYSSTMT, which always have a type attribute value of N
  - Some other character type system variable symbols can be assigned the value of a null string, in which case they have a type attribute value of O
- Macro instruction operands that specify a literal that is not a duplicate of a literal used in a machine instruction
- Inner macro instruction operands that are ordinary symbols

Notes:

1. Ordinary symbols used in the name field of an EQU instruction have the type attribute value U. However, the third operand of an EQU instruction can be used explicitly to assign a type attribute value to the symbol in the name field.
2. The type attribute of a sublist is set to the same value as the type attribute of the first element of the sublist.
3. High Level Assembler and earlier assemblers treat the type attribute differently:
   - Because High Level Assembler allows attribute references to statements generated through substitution, certain cases in which a type attribute of U (undefined, unknown, or unassigned) or M (macro name field) is given under the DOS/VSE Assembler, might give a valid type attribute under High Level Assembler. If the value of the SETC symbol is equal to the name of an instruction that can be referred to by the type attribute, High Level Assembler lets you use the type attribute with a SETC symbol.
   - Because High Level Assembler allows attribute references to literals, certain cases in which a type attribute of U (undefined, unknown, or unassigned) is given by Assembler F and Assembler H for a macro operand that specifies a literal, might give a valid type attribute under High Level Assembler. If the literal specified in the macro instruction operand is a duplicate of a literal specified in open code, or previously generated by conditional assembly processing or macro generation, High Level Assembler gives a type attribute that shows the type of data specified in the literal. The COMPAT(LITTYPE) option causes High Level Assembler to behave like Assembler H, always giving a type attribute of U for the T' literal.
   - When a type attribute reference is made outside conditional assembly instructions, its value is treated as a character self-defining term. For example, if the symbol A is defined in this statement:

     ```
     A       DC    A(*)
     ```

     then the symbol A has type attribute 'A' in conditional assembly instructions. However, if this statement is followed by

```
DC    A(T'A)          Generates X'000000C1'
```

the generated data is the same as if you had written

```
DC    A(C'A')         Generates X'000000C1'
```

Length attribute (L')

The length attribute has a numeric value equal to the number of bytes occupied by the data that is named by the symbol specified in the attribute reference.

Evaluation of length attribute references for conditional assembly statements is handled differently from references in ordinary assembly.

In conditional assembly statements, the operand of a length attribute reference must be either an ordinary symbol whose length attribute is either known, or can be determined in lookahead mode (Figure 43 on page 293); or it must be a variable symbol whose value is that of an ordinary symbol satisfying the same rules.

In ordinary assembly statements, the operand of a length attribute reference can be a character-valued conditional assembly expression whose value is that of an ordinary symbol.

Here is an example to clarify this distinction:

```
&B     SETC   'B'
AB     DC     C'A&B'           Valid in ordinary assembly
LAB    DC     AL1(L'A&B)       Valid in ordinary assembly
&N     SETA   L'A&B            Invalid in conditional assembly
&T1    SETB   (L'A&B EQ 2)     Invalid in conditional assembly
&T2    SETB   (2 EQ L'A&B)     Invalid in conditional assembly
```

The two SETB statements receive different diagnostic messages, because the errors are detected during different parts of the assembler's analysis of the SETB expressions.

In conditional assembly statements, the operand of a length attribute reference must be an ordinary or variable symbol, and not a character expression.

The length attribute can also be specified outside conditional assembly instructions. Then, the length attribute value is not available for conditional assembly processing, but is used as a value at assembly time.

Figure 43 on page 293 is an example showing the evaluation of the length attribute for an assembler instruction in statement 1 and for a conditional assembly instruction in statement 8.

```
000000 E740                          1 CSYM DC    CL(L'ZLOOKAHEAD)'X'  Length resolved later
                                     2 &LEN SETA  L'CSYM
** ASMA042E Length attribute of symbol is unavailable; default=1
                                     3      DC    C'&LEN '  REAL LENGTH NOT AVAILABLE
000002 F140                           +     DC    C'1 '     REAL LENGTH NOT AVAILABLE
                                     4 &TYP SETC  T'CSYM
                                     5      DC    C'&TYP '  TYPE IS KNOWN
000004 C340                           +     DC    C'C '     TYPE IS KNOWN
                                     6 &DEF SETA  D'CSYM
                                     7      DC    C'&DEF '  SYMBOL IS DEFINED
000006 F140                           +     DC    C'1 '     SYMBOL IS DEFINED
                                     8 &LEN SETA  L'zlookahead  Length resolved immediately
                                     9 CSYM2 DC    CL(&len)'X'
000008 E740                           +CSYM2 DC    CL(2)'X'
                                    10 &LEN SETA  L'CSYM2
                                    11      DC    C'&LEN '  REAL LENGTH NOW AVAILABLE
00000A F240                           +     DC    C'2 '     REAL LENGTH NOW AVAILABLE
00000C 0001                         12 ZLOOKAHEAD   DC    H'1'
                                    13      END
```

Figure 43. Evaluation of length attribute references

In statement 2 the length of CSYM has not been established because the definition of CSYM in statement 1 is not complete. The reference to the length attribute results in a length of 1 and error message ASMA042E. However, statement 5 shows that the type attribute is assigned, and statement 7 shows that the defined attribute is assigned. In comparison, the length attribute for symbol CSYM2 is available immediately, as it was retrieved indirectly using the conditional assembly instruction in statement 8.

During conditional assembly, an ordinary symbol used in the name field of an EQU instruction has a length attribute value that depends on the order of the symbol's definition and the reference to its length attribute.

- If the first operand of the EQU instruction is a self-defining term, the length attribute value is 1.
- If the first operand of the EQU instruction is a symbol whose value and length attribute are defined, the length attribute value is that of the symbol in the first operand.
- If the first operand of the EQU instruction is a defined symbol and the EQU instruction specifies a length value in the second operand, the length attribute value is that of the second operand.

At assembly time, the symbol has the same length attribute value as the first term of the expression in the first operand of the EQU instruction. However, the second operand of an EQU instruction can be used to assign a length attribute value to the symbol in the name field. This second operand cannot be a forward reference to another EQU instruction.

Notes:

1. The length attribute reference, when used in conditional assembly processing, can be specified only in arithmetic expressions.
2. When used in conditional assembly processing, a length attribute reference to a symbol with the type attribute value of M, N, O, T, U, or $ is flagged. The length attribute for the symbol has the default value of 1.

Scale attribute (S')

The scale attribute can be used only when referring to fixed-point, floating-point, or decimal constants. The following table shows the numeric value assigned to the scale attribute:

Constant Types Allowed	Type of DC or DS Allowed	Value of Scale Attribute Assigned
Fixed-Point	H and F	Equal to the value of the scale modifier (-187 through +346)
Floating Point	D, E, and L	Equal to the value of the scale modifier(0 through 14 — D, E)(0 through 28 — L)
Decimal	P and Z	Equal to the number of decimal digits specified to the right of the decimal point(0 through 31 — P)(0 through 16 — Z)

The scale attribute can also be specified outside conditional assembly instructions. Then, the scale attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

Notes:

1. The scale attribute reference can be used only in arithmetic expressions.
2. When no scale attribute value can be determined, the reference is flagged and the scale attribute is 1.
3. If the value of the SETC symbol is equal to the name of an instruction that can validly define the scale attribute, the assembler lets you use the scale attribute with a SETC symbol.
4. Binary floating-point constants return an attribute of 0.
5. Decimal floating-point constants return an attribute of 0.
6. The scale attribute reference can only be used in arithmetic expressions in conditional assembly instructions, and in absolute and relocatable expressions in assembler and machine instructions.

Integer attribute (I')

The integer attribute has a numeric value that depends on the length and scale attribute values of the data being referred to by the attribute reference. The formulas relating the integer attribute to the length and scale attributes are given in Table 54.

The integer attribute can also be specified outside conditional assembly instructions. Then, the integer attribute value is not used for conditional assembly processing, but is used as a value at assembly time.

Notes:

1. The integer attribute reference can be used only in arithmetic expressions.
2. When no integer attribute value can be determined, the reference is flagged and the integer attribute is 1.
3. If the value of the SETC symbol is equal to the name of an instruction that can validly define the integer attribute, the assembler lets you use the integer attribute with a SETC symbol.
4. Binary floating-point constants return an attribute of 0.
5. Decimal floating-point constants return an attribute of 0.
6. The integer attribute reference can only be used in arithmetic expressions in conditional assembly instructions, and in absolute and relocatable expressions in assembler and machine instructions.

Table 54. Relationship of integer to length and scale attributes

Constant Type	Formula Relating Integer to Length and Scale Attributes	Examples	Values of the Integer Attribute
Fixed-point(H and F)	`I' = 8*L'-S'-1`	`HALFCON DC HS6'-25.93'`	`I' = 8*2-6-1` `= 9`
		`ONECON DC FS8'100.3E-2'`	`I' = 8*4-8-1` `= 23`

Table 54. Relationship of integer to length and scale attributes (continued)

Constant Type	Formula Relating Integer to Length and Scale Attributes	Examples		Values of the Integer Attribute
Floating-point (D, E, and L)	when L' ≤ 8 I' = 2*(L'-1)-S'	SHORT	DC ES2'46.415'	I' = 2*(4-1)-2 = 4
		LONG	DC DS5'-3.729'	I' = 2*(8-1)-5 = 9
L-type only	when L' > 8 I' = 2*(L'-1)-S'-2	EXTEND	DC LS10'5.312'	I' = 2*(16-1)-10-2 = 18
Decimal[1] Packed (P)	I' = 2*L'-S'-1	PACK	DC P'+3.513'	I' = 2*3-3-1 = 2
Zoned (Z)	I' = L'-S'	ZONE	DC Z'3.513'	I' = 4-3 = 1

Note:

1. The value of the integer attribute is equal to the number of digits to the left of the assumed decimal point after the constant is assembled, and the value of the scale attribute is equal to the number of digits to the right of the assumed decimal point.

Count attribute (K')

The count attribute applies only to macro instruction operands, to SET symbols, and to the system variable symbols. It has a numeric value equal to the number of characters:

- That constitute the macro instruction operand, or
- That are required to represent as a character string the current value of the SET symbol or the system variable symbol.

Notes:

1. The count attribute reference can be used only in arithmetic expressions.
2. The count attribute of an omitted macro instruction operand has a value of 0.
3. Doubled quotes ('') in quoted character strings count as one character. Doubled ampersands (&&) in quoted character strings count as two characters. For more information about character pairs see "Evaluation of character expressions" on page 338.
4. These pairing rules mean that the length attribute of a character variable substituted into a character constant might be different from the count attribute of the substituted variable.
5. The count attribute differs from the Number (N') attribute, described below.

Number attribute (N')

The number attribute applies to the operands of macro instructions and subscripted SET symbols.

When applied to a macro operand, the number attribute is a numeric value equal to the number of sublist entries.

When applied to a subscripted SET symbol, the number attribute is equal to the highest element to which a value has been assigned in a SETx instruction. Consider the example in Figure 44 on page 296.

```
                                    1            macro
                                    2            MAC1 &op1
                                    3            lcla &SETSUB(100)
                                    4 &SETSUB(5) seta 20,,,70
                                    5 &B         seta N'&SETSUB
                                    6 &C         seta N'&op1
                                    7            DC C'Highest referenced element of SETSUB = &B'
                                    8            DC C'Number of sublist entries in OP1 = &C'
                                    9            mend
000000                00000 0004C  10 a          csect
                                   11            MAC1 (1,(3),(4))
000000 C889878885A2A340            12+           DC C'Highest referenced element of SETSUB = 8'
000028 D5A4948285994096            13+           DC C'Number of sublist entries in OP1 = 3'
                                   14            end
```

Figure 44. Number attribute reference

N'&op1 is equal to 3 because there are three subscripts in the macro operand in statement 11: 1, (3), and (4).

N'&SETSUB is equal to 8 because &SETSUB(8), assigned the value 70 in statement 4, is the highest referenced element of the &SETSUB array entries.

Notes:

1. The number attribute reference can be used only in arithmetic expressions.
2. N'&SYSLIST refers to the number of positional operands in a macro instruction, and N'&SYSLIST(*n*) refers to the number of sublist entries in the *n*-th operand.
3. For positional macro parameters, either explicitly named or implicitly named as &SYSLIST(*n*):
   a. If the first character of an operand is a left parenthesis, count the number of unquoted and unnested commas between it and the next matching right parenthesis. That number plus one is the number attribute of the operand.
   b. If there is no initial left parenthesis, the number attribute is one.
4. For all other system variable symbols, the number attribute value is always one. This is also true for &SYSMAC. The range of the subscript for &SYSMAC is 0 - &SYSNEST.
5. N' is always zero for unsubscripted set symbols. The number attribute (N'), when used with a macro instruction operand, examines its list structure, not the number of characters in the operand. (The number of characters is determined by the count (K') attribute.)

Defined attribute (D')

The defined attribute shows whether the ordinary symbol or literal referenced has been defined prior to the attribute reference. A symbol is defined if it has been encountered in the operand field of an EXTRN or WXTRN statement, or in the name field of any other statement except a TITLE statement or a macro instruction. A literal is defined if it has been encountered in the operand field of a machine instruction. The value of the defined attribute is an arithmetic value that can be assigned to a SETA symbol, and is equal to 1 if the symbol has been defined, or 0 if the symbol has not been defined.

The defined attribute can reference:
- Ordinary symbols not constructed by substitution
- Macro instruction operands
- SETC symbols whose value is an ordinary symbol
- System variable symbols whose value is an ordinary symbol
- Literals

Here is an example of how you can use the defined attribute:

```
Name      Operation      Operand

          AIF            (D'A).AROUND
A         LA             1,4
.AROUND   ANOP
```

In this example, assuming there has been no previous definition of the symbol A, the statement labeled A is assembled, since the conditional-assembly branch around it is not taken. However, if by an AGO or AIF conditional-assembly branch the same statement is processed again, the statement at A is not assembled:

```
Name      Operation      Operand

.UP       AIF            (D'A).AROUND
A         LA             1,4
.AROUND   ANOP
          .
          .
          AGO            .UP
```

You can save assembly time using the defined attribute which avoids lookahead mode (see "Lookahead" on page 299 for more information). You can use the defined attribute in your program to prevent the assembler from making this time-consuming forward scan. This attribute reference can be used in the operand field of a SETA instruction or as one of the values in the operand field of a SETB or AIF instruction.

Notes:

1. D' applied to a SETA or SETB symbol, or to a system variable symbol, is an error.
2. D' applied to a SETC symbol or symbolic parameter is valid only if the value of the SETC symbol or symbolic parameter is a valid ordinary symbol or literal.

Operation code attribute (O')

The operation code attribute shows whether a given operation code has been defined prior to the attribute reference. The operation code can be represented by a character string or by a variable symbol containing a character string. The variable must be set using a SETC assembler instruction prior to being referenced by the operation code (O') attribute.

The operation code attribute has a value of a single alphabetic character that shows the type of operation represented.

This attribute reference can be used in the operand field of the SETC instruction or as one of the values used in the operand field of a SETB or AIF instruction.

The following letters are used for the value of the operation code attribute:

A Assembler operation code
E Extended mnemonic operation code
M Macro definition
O Machine operation code
S Macro definition found in library
U Undefined, unknown, unassigned, or deleted operation code

Notes:

1. The operation code (O') attribute can only be used in a conditional assembly statement.
2. The assembler does not enter lookahead mode to resolve the operation code type, therefore only operation codes defined at the time the attribute is referenced return an operation code type value other than U.

3. When the operation code is not an assembler instruction or a machine instruction, and the operation code is not a previously defined macro, then all libraries in the library data set definition list are searched. This might have an adverse impact on the performance of the assembly, depending on the number of libraries assigned in the assembly job and the number of times the operation code attribute is used.

Examples:

```
Name      Operation      Operand

&A        SETC           O'MVC
```

&A contains the letter O, because MVC is a machine operation code:

```
Name      Operation      Operand

&A        SETC           'DROP'
&B        SETC           O'&A
```

&B contains the letter A, because DROP is an assembler operation code.

The following example checks to see if the macro MAC1 is defined. If not, the MAC1 macro instruction is bypassed. This prevents the assembly from failing when the macro is not available.

```
Name      Operation      Operand

&CHECKIT  SETC           O'MAC1
          AIF            ('&CHECKIT' EQ 'U').NOMAC
          MAC1
.NOMAC    ANOP
          .
```

Redefined Operation Codes: If an operation code is redefined using the OPSYN instruction then the value returned by a subsequent operation code attribute reference represents the new operation code. If the operation code is deleted using the OPSYN instruction then the value returned is U.

Sequence symbols

You can use a sequence symbol in the name field of a statement to branch to that statement during conditional assembly processing, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can select the model statements from which the assembler generates assembler language statements for processing at assembly time.

A sequence symbol consists of a period (.) followed by an alphabetic character, followed by 0 to 61 alphanumeric characters.

Examples:

```
.BRANCHING_LABEL#1
.A
```

Sequence symbols can be specified in the name field of assembler language statements and model statements; however, sequence symbols must not be used as name entries in the following assembler instructions:

```
ALIAS             EQU               OPSYN             SETC
AREAD             ICTL              SETA              SETAF
CATTR             LOCTR             SETB              SETCF
DXD
```

Also, sequence symbols cannot be used as name entries in macro prototype instructions, or in any instruction that already contains an ordinary or a variable symbol in the name field.

Sequence symbols can be specified in the operand field of an AIF or AGO instruction to branch to a statement with the same sequence symbol as a label.

Scope: A sequence symbol has a local scope. Thus, if a sequence symbol is used in an AIF or an AGO instruction, the sequence symbol must be defined as a label in the same part of the program in which the AIF or AGO instruction appears; that is, in the same macro definition or in open code.

Symbolic Parameters:

If a sequence symbol appears in the name field of a macro instruction, and the corresponding prototype statement contains a symbolic parameter in the name field, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro definition. The value of the symbolic parameter is a null character string.

Example:

```
          MACRO
 &NAME    MOVE            &TO,&FROM           Statement 1
 &NAME    ST              2,SAVEAREA          Statement 2
          L               2,&FROM
          ST              2,&TO
          L               2,SAVEAREA
          MEND
 ------------------------------------------------------------------
 .SYM     MOVE            FIELDA,FIELDB       Statement 3
 ------------------------------------------------------------------
+         ST              2,SAVEAREA          Statement 4
+         L               2,FIELDB
+         ST              2,FIELDA
+         L               2,SAVEAREA
```

The symbolic parameter `&NAME` is used in the name field of the prototype statement (Statement 1) and the first model statement (Statement 2). In the macro instruction (Statement 3), a sequence symbol (`.SYM`) corresponds to the symbolic parameter `&NAME`. `&NAME` is not replaced by `.SYM` and, therefore, the generated statement (Statement 4) does not contain an entry in the name field.

Lookahead

Symbol attributes are established in either definition mode or lookahead mode.

Definition mode occurs whenever a previously undefined symbol is encountered in the name field of a statement, or in the operand field of an EXTRN or WXTRN statement during open code processing. Symbols within a macro definition are defined when the macro is expanded.

Lookahead mode is entered:

- When the assembler processes a conditional assembly instruction and encounters an attribute reference (other than D' and O') to an ordinary symbol that is not yet defined.
- When the assembler encounters a forward AGO or AIF branch in open code to a sequence symbol that is not yet defined.

Lookahead is a sequential, statement-by-statement, forward scan over the source text.

If the attribute reference is made in a macro, forward scan begins with the first source statement following the outermost macro instruction. During lookahead the assembler:

- Bypasses macro definition and generation
- Does not generate object text
- Does not perform open-code variable substitution
- Ignores AIF and AGO branch instructions
- Ignores the record following a REPRO statement
- Establishes interim data attributes for undefined symbols it encounters in operand fields of instructions. The data attributes are replaced when a symbol is encountered in definition mode.

Lookahead mode ends when the desired symbol or sequence symbol is found, or when the END statement or end of file is reached. All statements read by lookahead are saved on an internal file, and are fully processed when the lookahead scan ends.

If a COPY instruction is encountered during lookahead, it is fully processed at that time, the assembler copies the statements from the library, scans them, and saves them on the lookahead file. When lookahead mode has ended any COPY instructions saved to the lookahead file are ignored, as the statements from the copy member have already been read and saved to the lookahead file.

If a variable symbol is used for the member name of a COPY that is expanded during lookahead, the value of the variable symbol at the time the COPY is expanded is used.

For purposes of attribute definition, a symbol is considered partially defined if it depends in any way upon a symbol not yet defined. For example, if the symbol is defined by a forward EQU that is not yet resolved, that symbol is assigned a type attribute of U.

In this case it is possible that, by the end of the assembly, the type attribute has changed to some other value.

Generating END statements

Because no variable symbol substitution is carried out during lookahead, consider the following effects of using macro, AINSERT or open code substitution to generate END statements that separate source modules assembled in one job step (BATCH assembler option). If a symbol is undefined within a module, lookahead might read statements past the point where the END statement is to be generated. Lookahead stops when:

1. It finds the symbol
2. It finds an END statement
3. It reaches the end of the source input data set

In the first two cases, the assembler begins the next module at the statement after lookahead stopped, which might be after the point where you wanted to generate the END statement.

Lookahead restrictions

The assembler analyzes the statements it processes during lookahead, only to establish attributes of symbols in their name fields.

Variable symbols are not replaced. Modifier expressions are evaluated only if all symbols involved were defined prior to lookahead. Possible multiple or inconsistent definition of the same symbol is not diagnosed during lookahead because conditional assembly might eliminate one (or more) of the definitions.

Lookahead does not check undefined operation codes against library macro names. If the name field contains an ordinary symbol and the operation code cannot be matched with one in the current operation code table, then the ordinary symbol is assigned the type attribute of M. If the operation code contains special characters or is a variable symbol, a type attribute of U is assumed. This can be wrong if the undefined operation code is later substituted with a known operation code or is later defined by OPSYN. OPSYN statements are not processed; thus, labels are treated in accordance with the operation code definitions in effect at the time of entry to lookahead.

Sequence symbols

The conditional assembly instructions AGO and AIF in open code control the sequence in which source statements are processed. Using these instructions it is possible to branch back to a sequence symbol label and reuse previously processed statements. Due to operating system restrictions, the primary input source can only be read sequentially, and cannot be reread. Whenever a sequence symbol in the name field is encountered in open code, the assembler must assume that all subsequent statements might need

to be processed more than once. The assembler uses the lookahead file to save the statement containing the sequence symbol label and all subsequent statements as they are read and processed. Any subsequent AGO or AIF to a previously encountered sequence symbol is resolved to an offset into the lookahead file and input continues from that point.

Open code

Conditional assembly instructions in open code let you:

- Select, during conditional assembly, statements or groups of statements from the open code portion of a source module according to a predetermined set of conditions. The assembler further processes the selected statements at assembly time.
- Pass local variable information from open code through parameters into macro definitions.
- Control the computation in and generation of macro definitions using global SET symbols.
- Substitute values into the model statements in the open code of a source module and control the sequence of their generation.

All the conditional assembly elements and instructions can be specified in open code.

The specifications for the conditional assembly language described in this chapter also apply in open code. However, the following restrictions apply:

To Attributes In Open Code:

For ordinary symbols, only references to the type, length, scale, integer, defined, and operation code attributes are allowed.

References to the number attribute have no meaning in open code, because &SYSLIST is not allowed in open code, and symbolic parameters have no meaning in open code.

To Conditional Assembly Expressions:

Table 55 shows the restrictions for different expression types.

Table 55. Restrictions on coding expressions in open code

Expression	Must not contain
Arithmetic (SETA)	• &SYSLIST • Symbolic parameters • Any attribute references to symbolic parameters, or system variable symbols with local scope
Character (SETC)	• System variables with local scope • Attribute references to system variables with local scope • Symbolic parameters
Logical (SETB)	• Arithmetic expressions with the items listed above • Character expressions with the items listed above

Conditional assembly instructions

The remainder of this chapter describes, in detail, the syntax and rules for use of each conditional assembler instruction. The following table lists the conditional assembler instructions by type, and provides the page number where the instruction is described in detail.

Table 56. Assembler instructions

Type of Instruction	Instruction	Page No.
Establishing SET symbols	GBLA	302
	GBLB	302
	GBLC	302
	LCLA	304
	LCLB	304
	LCLC	304
	SETA	308
	SETB	321
	SETC	326
Branching	ACTR	348
	AGO	347
	AIF	344
	ANOP	349
External Function Calling	SETAF	343
	SETCF	344

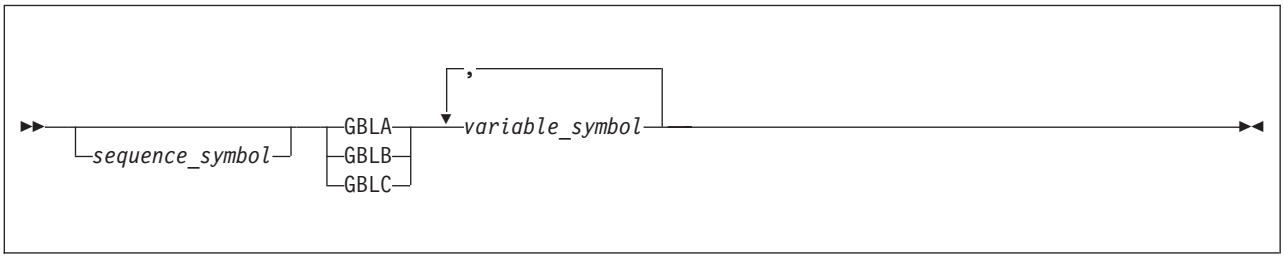
Declaring SET symbols

You must declare a global SET symbol before you can use it. The assembler assigns an initial value to a global SET symbol at its first point of declaration.

Local SET symbols need not be declared explicitly with LCLA, LCLB, or LCLC statements. The assembler considers any undeclared variable symbol found in the name field of a SETA, SETB, SETC, SETAF, or SETCF statement to be a local SET symbol. It is given the initial value specified in the operand field. If the symbol in the name field is subscripted, it is declared as a subscripted SET symbol.

GBLA, GBLB, and GBLC instructions

Use the GBLA, GBLB, and GBLC instructions to declare the global SETA, SETB, and SETC symbols you need. The SETA, SETB, and SETC symbols are assigned the initial values of 0, 0, and null character string.



sequence_symbol
Is a sequence symbol.

variable_symbol
Is a variable symbol, with or without the leading ampersand (&).

These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

Any variable symbols declared in the operand field have a global scope. They can be used as SET symbols anywhere after the pertinent GBLA, GBLB, or GBLC instructions. However, they can be used only within those parts of a program in which they have been declared as global SET symbols; that is, in any macro definition and in open code.

The assembler assigns an initial value to the SET symbol only when it processes the first GBLA, GBLB, or GBLC instruction in which the symbol appears. Later GBLA, GBLB, or GBLC instructions do not reassign an initial value to the SET symbol.

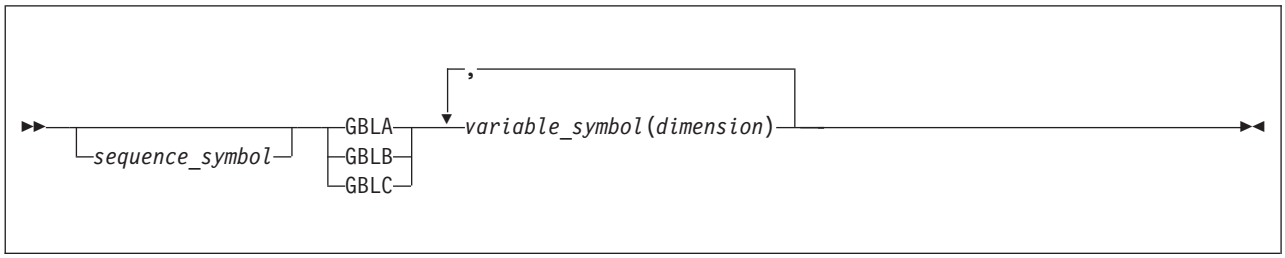
Multiple GBLx statements can declare the same variable symbol so long as only one declaration for a given symbol is encountered during the expansion of a macro.

The following rules apply to the global SET variable symbol:

- Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any local variable symbol declared within the same local scope.
- The same variable symbol must not be declared or used as two different types of global SET symbol; for example, as a SETA or SETB symbol.
- Do not begin a global SET symbol with `&SYS`, because these characters are used for system variable symbols.
- If the variable symbol is the same as the character value, the assembler considers the variable symbol to be an implicitly defined local SETC symbol which is given a null character string value. For example: `&C6 SETC '&C6'` , assigns the value `''` to `&C6`.

Subscripted global SET symbols

A global subscripted SET symbol is declared by the GBLA, GBLB, or GBLC instruction.



sequence_symbol
Is a sequence symbol.

variable_symbol
Is a variable symbol, with or without the leading ampersand (&).

dimension
Is the dimension of the array. It must be an unsigned, decimal, self-defining term greater than zero.

Example:

```
GBLA            &GA(25),&GA1(15)
```

There is no limit on the maximum subscript allowed, except that each subscripted variable is allocated storage, so the maximum subscript can be limited by the amount of storage available. Also, the limit

specified in the global declaration (GBLx) can be exceeded. The dimension shows the number of SET variables associated with the subscripted SET symbol. The assembler assigns an initial value to every variable in the array thus declared.

Notes:

1. Global arrays are assigned initial values only by the first global declaration processed, in which a global subscripted SET symbol appears.
2. A subscripted global SET symbol can be used only if the declaration has a subscript, which represents a dimension; an unsubscripted global SET symbol can be used only if the declaration had no subscript, except for a number attribute reference to the name of a dimensioned SET symbol.

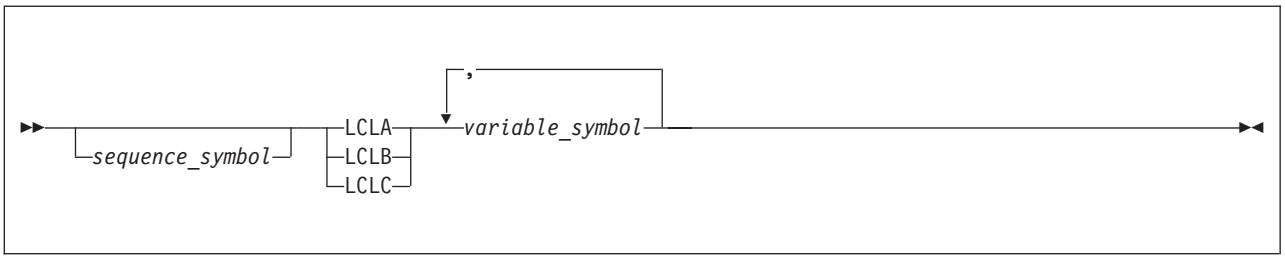
Alternative format for GBLx statements

The assembler permits the alternative statement format for GBLx instructions:

```
                                                Cont.

      GBLA          &GLOBAL_SYMBOL_FOR_DC_GEN,    X
                    &LOOP_CONTRL_A,               X
                    &VALUE_PASSED_TO_FIDO,        X
                    &VALUE_RETURNED_FROM_FIDO
```

LCLA, LCLB, and LCLC instructions

Use the LCLA, LCLB, and LCLC instructions to declare the local SETA, SETB, and SETC symbols you need. The SETA, SETB, and SETC symbols are assigned the initial values of 0, 0, and null character string.



sequence_symbol
Is a sequence symbol.

variable_symbol
Is a variable symbol, with or without the leading ampersand (&).

These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

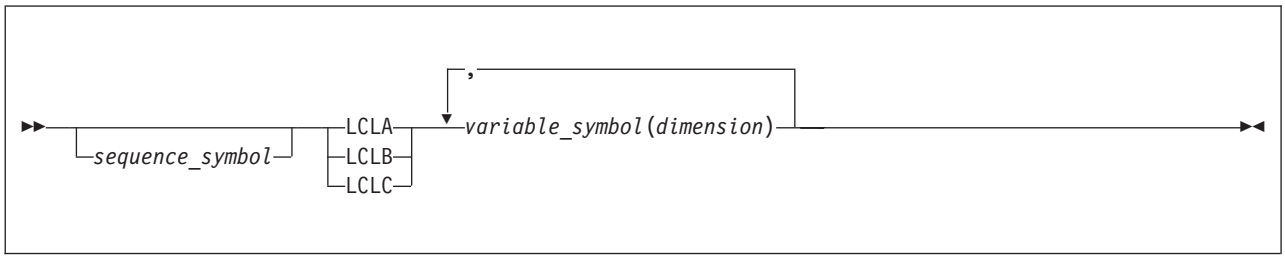
Any variable symbols declared in the operand field have a local scope. They can be used as SET symbols anywhere after the pertinent LCLA, LCLB, or LCLC instructions, but only within the declared local scope. Multiple LCLx statements can declare the same variable symbol so long as only one declaration for a given symbol is encountered during the expansion of a macro.

The following rules apply to a local SET variable symbol:

- Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.
- It must not be the same as any global variable symbol declared within the same local scope.
- The same variable symbol must not be declared or used as two different types of SET symbols; for example, as a SETA and a SETB symbol, within the same local scope.
- Do not begin a local SET symbol with &SYS, because these characters are used for system variable symbols.

Subscripted local SET symbols

A local subscripted SET symbol is declared by the LCLA, LCLB, or LCLC instruction.



sequence_symbol
Is a sequence symbol.

variable_symbol
Is a variable symbol, with or without the leading ampersand (&).

dimension
Is the dimension of the array. It must be an unsigned, decimal, self-defining term greater than zero.

Example:

```
        LCLB            &B(10)
```

There is no limit to SET symbol dimensioning other than storage availability. The limit specified in the explicit (LCLx) or implicit (SETx) declaration can also be exceeded by later SETx statements. The dimension shows the number of SET variables associated with the subscripted SET symbol. The assembler assigns an initial value to every variable in the array thus declared.

Subscripted local SET symbol: A subscripted local SET symbol can be used only if the declaration has a subscript, which represents a dimension; an unsubscripted local SET symbol can be used only if the declaration had no subscript, except for a number attribute reference to the dimensioned SET symbol.

Alternative format for LCLx statements

The assembler permits an alternative statement format for LCLx instructions:

```
                                                  Cont.

        LCLA            &LOCAL_SYMBOL_FOR_DC_GEN,   X
                        &COUNTER_FOR_INNER_LOOP,    X
                        &COUNTER_FOR_OUTER_LOOP,    X
                        &COUNTER_FOR_TRAILING_LOOP
```

Assigning values to SET symbols

You can assign values to SET symbols by using the SETA, SETB, SETC, SETAF, and SETCF instructions (SETx). You can also use these instructions to implicitly define local SET symbols. Local SET symbols need not be declared explicitly with LCLA, LCLB, or LCLC statements. The assembler considers any undeclared variable symbol found in the name field of a SETx statement to be a local SET symbol. It is given the initial value specified in the operand field of SETA, SETB, and SETC instructions, and the value returned from the external function specified in the operand of SETAF and SETCF instructions. If the symbol in the name field is subscripted, it is declared as a subscripted SET symbol.

Spaces do not terminate the operand field when used in logical expressions and in built-in functions. For more information, see "Logical (SETB) expressions" on page 323.

Introducing Built-In Functions

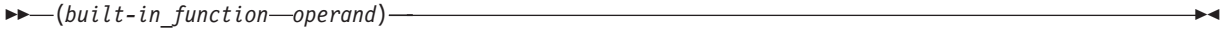
The assembler provides built-in functions for the SETA, SETB, and SETC expressions.

Each function returns one value - an arithmetic value for SETA, a binary bit for SETB, and a character string for SETC.

There are two different forms of invocation for the built-in functions:

- The **logical-expression** format encloses the function and operands in parentheses. In the unary format, the function is followed by the one operand. In the binary format, the function is placed between the two operands. For both unary and binary formats, the function is separated from the operand or operands by spaces.

  Logical-expression unary format

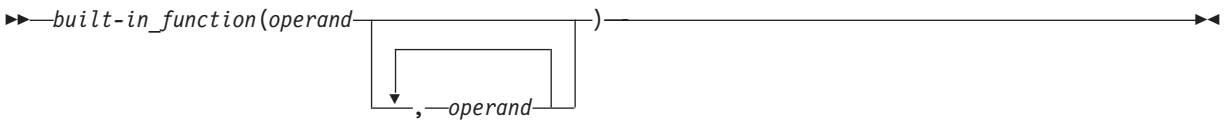
  Logical-expression binary format

  `(A OR B)` and `(&J SLL 2)` are examples of binary logical-expression format functions, and `(NOT C)` and `(SIGNED &J)` are examples of unary logical-expression format functions.

- The **function-invocation** format has the function first, followed by one or more operands in parentheses.

  Function-invocation format

  `FIND('abcde','d')` is an example of a function-invocation format. (The equivalent logical-expression format is `('abcde' FIND 'd')`.)

  Spaces are not allowed between the arguments of functions in function-invocation format.

In either format, the *operand* is an expression of the type expected by the built-in function. (The particular details of the number of operands and the operand type are provided with the information for each built-in function.)

Conditional-assembly functions do not always behave like functions in traditional high-level languages. The results of a function might not be automatically converted to the type expected in the invoking expression, and nested invocations might not produce expected results. In general, it is safest to invoke only one conditional assembly function in a SET expression or AIF statement.

Some functions are available in one format, some are available in both. Table 57 on page 307, which provides a summary of all the built-in functions, shows the forms in which a function is available.

Because some function names such as AND and OR are used both as arithmetic operators and as logical connectives, their use might appear to be ambiguous. For example, the function `(1 AND 2)` in an arithmetic expression is interpreted as the logical AND of the two 32 bit SETA expressions 1 and 2, resulting in zero. In a logical expression, the two nonzero operands are converted to 1 (meaning "true") and the result is 1. Similarly, the function `(1 XOR 2)` in an arithmetic expression has value 3, while in a logical expression it has value 0.

To avoid ambiguities, such function names are interpreted as arithmetic operators in SETA statements, and as logical operators in SETB and AIF statements.

Table 57. Summary of Built-In Functions and Operators

Function	Type	L-E[1]	F-I[2]	Result[3]	Operands[3]	Page
A2B	Representation conversion		✓	C	A	331
A2C	Representation conversion		✓	C	A	331
A2D	Representation conversion		✓	C	A	331
A2X	Representation conversion		✓	C	A	331
AND	Logical	✓		A	A	311
AND	Logical	✓		B	B	323
AND NOT	Logical	✓		B	B	323
B2A	Representation conversion		✓	A	C	311
B2C[4]	Representation conversion		✓	C	C	331
B2D	Representation conversion		✓	C	C	331
B2X[4]	Representation conversion		✓	C	C	331
BYTE	Representation conversion	✓	✓	C	A	331
C2A	Representation conversion		✓	A	C	311
C2B[4]	Representation conversion		✓	C	C	331
C2D	Representation conversion		✓	C	C	331
C2X[4]	Representation conversion		✓	C	C	331
D2A	Representation conversion		✓	A	C	311
D2B	Representation conversion		✓	C	C	331
D2C	Representation conversion		✓	C	C	331
D2X	Representation conversion		✓	C	C	331
DCLEN	String manipulation		✓	A	C	311
DCVAL	String manipulation		✓	C	C	331
DEQUOTE	String manipulation		✓	C	C	331
DOUBLE	String manipulation	✓	✓	C	C	331
FIND	String scanning	✓	✓	A	C	311
INDEX	String scanning	✓	✓	A	C	311
ISBIN	Validity checking		✓	B	C	311
ISDEC	Validity checking		✓	B	C	311
ISHEX	Validity checking		✓	B	C	311
ISSYM	Validity checking		✓	B	C	311
LOWER	String manipulation	✓	✓	C	C	331
NOT	Logical	✓	✓	A	A	311
NOT	Logical	✓		B	B	323

Table 57. Summary of Built-In Functions and Operators (continued)

Function	Type	L-E[1]	F-I[2]	Result[3]	Operands[3]	Page
OR	Logical	✓		A	A	311
OR	Logical	✓		B	B	323
OR NOT	Logical	✓		B	B	323
SIGNED	Representation conversion	✓	✓	C	A	331
SLA	Shift	✓		A	A	311
SLL	Shift	✓		A	A	311
SRA	Shift	✓		A	A	311
SRL	Shift	✓		A	A	311
SYSATTRA	Information retrieval		✓	C	C	331
SYSATTRP	Information retrieval		✓	C	C	331
UPPER	String manipulation	✓	✓	C	C	331
X2A	Representation conversion		✓	A	C	311
X2B[4]	Representation conversion		✓	C	C	331
X2C[4]	Representation conversion		✓	C	C	331
X2D	Representation conversion		✓	C	C	331
XOR	Logical	✓		A	A	311
XOR	Logical	✓		B	B	323
XOR NOT	Logical	✓		B	B	323

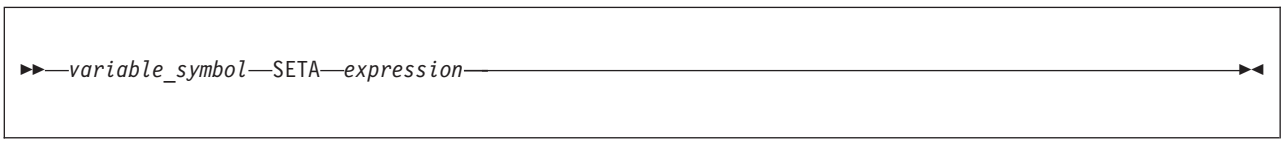
Notes:

1. If a ✓ is in this column, the function is available in the "logical-expression" format.
2. If a ✓ is in this column, the function is available in the "function-invocation" format.
3. Possible values in these columns are:
   A Arithmetic
   B Binary
   C Character
4. For these functions, the maximum length of the operand (and output) is the maximum string length that the assembler supports, currently 1024.

SETA instruction

The SETA instruction assigns an arithmetic value to a SETA symbol. You can specify a single value or an arithmetic expression from which the assembler computes the value to assign.

You can change the values assigned to an arithmetic or SETA symbol. This lets you use SETA symbols as counters, indexes, or for other repeated computations that require varying values.



variable_symbol
Is a variable symbol.

A global variable symbol in the name field must have been previously declared as a SETA symbol in a GBLA instruction. Local SETA symbols need not be declared in an LCLA instruction. The assembler

considers any undeclared variable symbol found in the name field of a SETA instruction as a local SET symbol. The variable symbol is assigned a type attribute value of N.

expression
Is an arithmetic expression evaluated as a signed 32 bit arithmetic value that is assigned to the SETA symbol in the name field. The minimum and maximum allowable values of the expression are -2^{31} and $+2^{31}-1$.

Figure 45 defines an arithmetic expression.

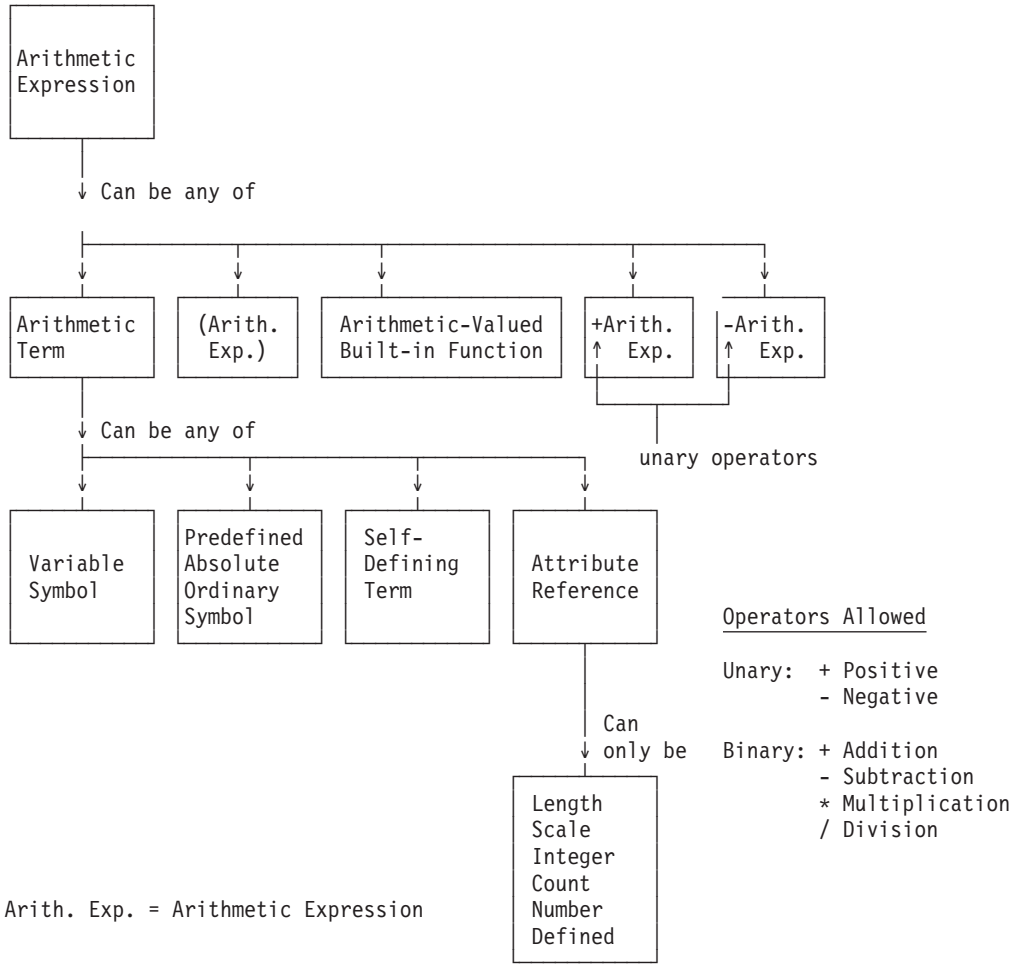


Figure 45. Defining arithmetic (SETA) expressions

Table 58 shows the variable symbols that are allowed as terms in an arithmetic expression.

Table 58. Variable symbols allowed as terms in arithmetic expressions

Variable symbol	Restrictions	Example	Valid value
SETA	None	---	---
SETB	None	---	---
SETC	Value must evaluate to an unsigned binary, hexadecimal, or decimal self-defining term	123	123

Table 58. Variable symbols allowed as terms in arithmetic expressions (continued)

Variable symbol	Restrictions	Example	Valid value
Symbolic parameters	Value must be a self-defining term	`&PARAM`	`X'A1'`
		`&SUBLIST(3)`	`C'Z'`
&SYSLIST(*n*) &SYSLIST(*n*,*m*)	Corresponding operand or sublist entry must be a self-defining term	`&SYSLIST(3)`	`24`
		`&SYSLIST(3,2)`	`B'101'`
&SYSDATC &SYSM_HSEV &SYSM_SEV &SYSNDX &SYSNEST &SYSOPT_DBCS &SYSOPT_RENT &SYSOPT_XOBJECT &SYSSTMT	None	---	---

The following example shows a SETA statement with a valid self-defining term in its operand field:

```
&ASYM1   SETA  C'D'           &ASYM1 has value 196 (C'D')
```

The second statement in the following example is valid because in the two positions in the SETA operand where a term is required (either side of the + sign), the assembler finds a valid self-defining term:

```
&CSYM2   SETC  'C''A'''       &CSYM2 has value C'A'
&ASYM3   SETA  &CSYM2+&CSYM2  &ASYM3 has value 386 (C'A' + C'A')
```

If the variable symbol is the same as the character value, the assembler considers the variable symbol to be an implicitly defined local SETA symbol, which is given a value of zero. For example:

```
&ASYM2   SETA  &ASYM2
```

`&ASYM2` has a value 0.

A SET statement is not rescanned by the assembler to see if substitutions might affect the originally determined syntax. The original syntax of the self-defining term must be correct. Therefore the assembler does not construct a self-defining term in a SETA statement. The third statement of the next example shows this:

```
&CSYM3   SETC  '3'            &CSYM has value 3 (C'3')
&ASYM3   SETA  &CSYM3         &ASYM has value 3
&ASYM4   SETA  C'&ASYM3'      Invalid self-defining term
```

In this example C'&ASYM3' is not a valid term.

Subscripted SETA symbols

The SETA symbol in the name field can be subscripted. If the same SETA symbol has not been previously declared in a GBLA or LCLA instruction with an allowable dimension, or has not been implicitly declared in a SETA instruction as a scalar (unsubscripted) variable symbol, then the symbol is implicitly declared as a local SETA array variable.

The assembler assigns the value of the expression in the operand field to the position in the declared array given by the value of the subscript. The subscript expression must not be 0 or have a negative value.

Arithmetic (SETA) expressions

Table 59 shows how arithmetic expressions can be used.

Table 59. Use of arithmetic expressions

Used in	Used as	Example
SETA instruction	Operand	`&A1 SETA &A1+2`
AIF or SETB instruction	Term in arithmetic relation	`AIF (&A*10 GT 30).A`
Subscripted SET symbols	Subscript	`&ASYM(&A+10-&C)`
Substring notation	Subscript	`'STRING'(&A*2,&A-1)`
Sublist notation	Subscript	Given sublist `(A,B,C,D)` named &PARAM, if `&A=1` then `&PARAM(&A+1)=B`
&SYSLIST	Subscript	`&SYSLIST(&M+1,&N-2)` `&SYSLIST(N'&SYSLIST)`
SETC instruction	Character string in operand	Given `&C SETC '5-10*&A'` 1 if `&A=10` then `&C=5-10*10` 2 Given `&D SETC '5-10*&A'` 1 if `&A=-10` then `&D=5-10*10` 3
Built-in functions	Operand	`&VAR SETA (NOT &OP1) &VAR SETA BYTE(64)`

When an arithmetic expression is used in the operand field of a SETC instruction (see 1 in Table 59), the assembler assigns the character value representing the arithmetic expression to the SETC symbol, after substituting values (see 2 in Table 59) into any variable symbols. It does not evaluate the arithmetic expression. The mathematical sign (+ or -) is not included in the substituted value of a variable symbol (see 3 in Table 59), and any insignificant leading zeros are removed.

Here are the built-in functions for arithmetic (SETA) expressions:

AND

Format: Logical-expression
Operands: Arithmetic

Output: `(aexpr1 AND aexpr2)` provides an arithmetic result where each bit position in the result is set to 1 if the corresponding bit positions in both operands contain 1, otherwise, the result bit is set to 0.

Example

After the following statements `&VAR` contains the arithmetic value `+2`.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 AND &OP2)
```

B2A

Format: Function-invocation
Operands: Character

Output: B2A('bitstring') converts a character string argument containing '0' and '1' characters to an arithmetic value.
- Fewer than 32 characters are padded internally on the left with '0' characters to a length of 32 characters.
- Error conditions are detected if the argument contains invalid characters, or if the argument length exceeds 32 characters, generating the message ASMA214E.
- Null argument strings return zero.

The result of the B2A function is the same as

```
&value  SETA  B'bitstring'
```

except that null strings are allowed by B2A but not by SETA.

Examples

```
B2A('')              has value 0
B2A('0000000101')    has value 5
B2A('11111111111111111111111111111110')    has value -2
```

C2A

Format: Function-invocation
Operands: Character

Output: C2A('charstring') converts a character string of zero to four characters to a binary arithmetic value having the same bit pattern.
- Fewer than four characters are padded internally on the left with EBCDIC null characters to a length of four characters.
- An error condition is detected if the argument length exceeds 4 characters, generating the message ASMA214E.
- Null argument strings return zero.

The result of C2A is the same as is obtained from

```
&value  SETA C'charstring'
```

except that C2A gives a zero result for null strings, and does not pair apostrophes or ampersands before conversion.

Example

```
C2A('')              has value 0
C2A('+')             has value 78
C2A('1')             has value 241
C2A('0000')          has value -252645136
```

D2A

Format: Function-invocation
Operands: Character

Output: D2A('decstring') converts a character string argument containing an optional leading plus or minus sign followed by decimal digits to an arithmetic value. Error conditions are detected if:
- The argument contains invalid characters.
- No digits are present following a sign.
- The argument length exceeds 11 characters.
- The resulting value is too large.

Null argument strings return zero.

The result of the D2A function is the same as

```
&value  SETA  decstring
```

except that SETA does not allow leading plus or minus signs.

Examples

```
D2A('')             indicates an error condition
D2A('000')          has value 0
D2A('10')           has value 10
D2A('+100')         has value 100
D2A('-5')           has value -5
```

DCLEN

Format: Function-invocation
Operands: Character

Output: `DCLEN('cexpr')` returns the length of its argument string after pairs of apostrophes and ampersands have been internally replaced by single occurrences. No change is made to the argument. Such pairing occurs only once; that is, three successive occurrences of an apostrophe or ampersand result in two occurrences, not one.

Examples

```
DCLEN('')          has value 0 (null string)
DCLEN('''')        has value 1 (argument is a single apostrophe)
DCLEN('''''')      has value 1 (argument is two apostrophes)
DCLEN('&&')        has value 1 (argument is two ampersands)
DCLEN('a''''b')    has value 3 (DCVAL string is "a'b")
DCLEN('a''''b&&c') has value 5 (DCVAL string is "a'b&c")
DCLEN('&&&&.''''''''')  has value 4 (DCVAL string is "&&''")
```

Note: DCLEN is like DCVAL, except that DCLEN returns only the length of the result, not the paired string.

FIND

Format: Logical-expression, function-invocation
Operands: Character

Output: `('string1' FIND 'string2')` or `FIND('string1','string2')` finds the first match of any character from *operand2* within *operand1*. The value returned by FIND indicates the position where the match occurs. FIND returns 0 if no match occurs or if either operand is a null string.

Examples

After the following statements `&VAR` contains the arithmetic value `3`.

```
Name      Operation      Operand

&OP1      SETC           'abcdef'
&OP2      SETC           'cde'
&VAR      SETA           ('&OP1' FIND '&OP2')
```

In the above example the character `c` in `&OP2` is the first character found in `&OP1`. Consider the following example where the character `c`, in `&OP1`, has been replaced with the character `g`.

```
Name      Operation      Operand

&OP1      SETC           'abcdef'
&OP2      SETC           'gde'
&VAR      SETA           ('&OP1' FIND '&OP2')
```

`&VAR` contains the arithmetic value `4`. The character `d` in `&OP2` is the first character found in `&OP1`.

In the following example, the ordering of the characters in the second operand is changed to `egd`.

```
Name      Operation      Operand

&OP1      SETC           'abcdef'
&OP2      SETC           'egd'
&VAR      SETA           FIND('&OP1','&OP2')
```

&VAR still contains the arithmetic value 4. Because FIND is looking for a single character from the character string, the order of the characters in the second operand string is irrelevant.

INDEX

Format: Logical-expression, function-invocation
Operands: Character

Output: INDEX('cexpr1','cexpr2') or ('cexpr1' INDEX 'cexpr2') locates the first occurrence of the second argument within the first argument, and returns the position of the match. A zero value is returned if:

- Either argument is null
- No match is found
- The second argument is longer than the first

Examples

```
INDEX('ABC','B')     has value 2
INDEX('ABC','D')     has value 0
```

ISBIN

Format: Function-invocation
Operands: Character

Output: ISBIN('cexpr') determines the validity of cexpr, a string of 1 to 32 characters, as the nominal value of a binary self-defining term usable in a SETA expression. If valid, ISBIN returns 1; otherwise, it returns zero. The argument string must not be null.

Example

```
ISBIN('10101')          returns 1
ISBIN('10101010101010101010101010101010101') returns 0 (excess digits)
ISBIN('12121')          returns 0 (non-binary digits)
ISBIN('')               indicates an error condition
```

ISDEC

Format: Function-invocation
Operands: Character

Output: ISDEC('cexpr') determines the validity of cexpr, a string of 1 to 10 characters, as the nominal value of a decimal self-defining term usable in a SETA expression. If valid, ISDEC returns 1; otherwise, it returns zero. The argument string must be null.

Example

```
ISDEC('12345678')       returns 1
ISDEC('+25')            returns 0 (non-decimal character)
ISDEC('2147483648')     returns 0 (value too large)
ISDEC('00000000005')    returns 0 (too many characters)
ISDEC('')               indicates an error condition
```

ISHEX

Format: Function-invocation
Operands: Character

Output: ISHEX('cexpr') determines the validity of cexpr, a string of 1-8 characters, as the nominal value of a hexadecimal self-defining term usable in a SETA expression. If valid, ISHEX returns 1; otherwise, it returns zero. The argument string mmust not be null.

Example

```
ISHEX('ab34CD9F')       returns 1
ISHEX('abcdEFGH')       returns 0 (non-hexadecimal digits)
ISHEX('123456789')      returns 0 (too many characters)
ISHEX('')               indicates an error condition
```

ISSYM

Format: Function-invocation
Operands: Character

Output: ISSYM('cexpr') determines the validity of cexpr, a string of 1 to 63 characters, for use as an ordinary symbol. If valid, ISSYM returns 1; otherwise, it returns zero. The argument string must not be null.

Examples

```
ISSYM('Abcd_1234')      returns 1
ISSYM('_Abcd1234')      returns 1
ISSYM('##@$_')          returns 1
ISSYM('1234_Abcd')      returns 0 (invalid initial character)
ISSYM('')               indicates an error condition
```

NOT

Format: Logical-expression
Operands: Arithmetic

Output: (NOT aexp) provides the ones complement of the value contained or evaluated in the operand.

Example

After the following statements &VAR contains the arithmetic value -11.

```
Name      Operation      Operand

&OP1      SETA           10
&VAR      SETA           (NOT &OP1)
```

OR

Format: Logical-expression
Operands: Arithmetic

Output: Each bit position in the result is set to 1 if the corresponding bit positions in one or both operands contains a 1, otherwise the result bit is set to 0.

Example

After the following statements &VAR contains the arithmetic value +10.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 OR &OP2)
```

SLA

Format: Logical-expression
Operands: Arithmetic

Output: The 31 bit numeric part of the signed first operand is shifted left the number of bits specified in the rightmost six bits of the second operand. The sign of the first operand remains unchanged. Zeros are used to fill the vacated bit positions on the right.

Example

After the following statements &VAR contains the arithmetic value +8.

```
Name      Operation      Operand

&OP1      SETA           2
&OP2      SETA           2
&VAR      SETA           (&OP1 SLA &OP2)
```

SLL

Format: Logical-expression
Operands: Arithmetic

Output: (aexp1 SLL aexp2) shifts the 32 bit first operand left the number of bits specified in the rightmost six bits of the second operand. Bits shifted out of bit position 0 are lost. Zeros are used to fill the vacated bit positions on the right.

Example

After the following statements &VAR contains the arithmetic value +40.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 SLL &OP2)
```

SRA

Format: Logical-expression
Operands: Arithmetic

Output: The 31 bit numeric part of the signed first operand is shifted right the number of bits specified in the rightmost six bits of the second operand. The sign of the first operand remains unchanged. Bits shifted out of bit position 31 are lost. Bits equal to the sign are used to fill the vacated bit positions on the left.

Examples

After the following statements &VAR contains the arithmetic value +2.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 SRA &OP2)
```

After the following statements &VAR contains the arithmetic value -1.

```
Name      Operation      Operand

&OP1      SETA           -344
&OP2      SETA           40
&VAR      SETA           (&OP1 SRA &OP2)
```

Compare this result with the result in the second example under SRL below.

SRL

Format: Logical-expression
Operands: Arithmetic

Output: The 32 bit first operand is shifted right the number of bits specified in the rightmost six bits of the second operand. Bits shifted out of bit position 31 are lost. Zeros are used to fill the vacated bit positions on the left.

Examples

After the following statements `&VAR` contains the arithmetic value `+2`.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 SRL &OP2)
```

After the following statements `&VAR` contains the arithmetic value `0`.

```
Name      Operation      Operand

&OP1      SETA           -344
&OP2      SETA           40
&VAR      SETA           (&OP1 SRL &OP2)
```

X2A

Format: Function-invocation
Operands: Character

Output: `X2A('hexstring')` converts a character string argument containing hexadecimal digits to an arithmetic value.

- If the character string contains fewer than eight characters, it is padded internally on the left with `'0'` characters.
- Error conditions are detected if the argument contains invalid characters, or if the argument length exceeds eight characters, generating the message ASMA214E.
- Null argument strings return zero.

The result of the X2A function is the same as

```
&value  SETA  X'hexstring'
```

except that null strings are allowed by X2A but not by SETA.

Examples

```
X2A('00000101')      has value 257
X2A('C1')            has value 193
X2A('')              has value 0
X2A('FFFFFFF0')      has value -16
```

XOR

Format: Logical-expression
Operands: Arithmetic

Output: Each bit position in the result is set to 1 if the corresponding bit positions in the two operands are unlike, otherwise the result bit is set to 0.

Example After the following statements `&VAR` contains the arithmetic value `+8`.

```
Name      Operation      Operand

&OP1      SETA           10
&OP2      SETA           2
&VAR      SETA           (&OP1 XOR &OP2)
```

Rules for coding arithmetic expressions: Here is a summary of coding rules for arithmetic expressions:

1. Unary (operating on one value) operators and binary (operating on two values) operators are allowed in arithmetic expressions.
2. An arithmetic expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression. The unary operators are + (positive) and - (negative).
3. The binary operators that can be used to combine the terms of an expression are + (addition), - (subtraction), * (multiplication), and / (division).
4. An arithmetic expression must not begin with a binary operator, and it must not contain two binary operators in succession.
5. An arithmetic-valued function is a term.
6. An arithmetic expression must not contain two terms in succession.
7. An arithmetic expression must not contain a decimal point. For example, `123.456` is not a valid arithmetic term, but `123` is.
8. An arithmetic expression must not contain spaces between an operator and a term, nor between two successive operators except for built-in functions using the "logical-expression format" described at "Logical (SETB) expressions" on page 323.
9. Ordinary symbols specified in arithmetic expressions must be defined before the arithmetic expression is encountered, and must have an absolute value.
10. An arithmetic expression can contain up to 24 unary and binary operators, and is limited to 255 levels of parentheses. The parentheses required for sublist notation, substring notation, and subscript notation count toward this limit.

    An arithmetic expression must not contain two terms in succession; however, any term can be preceded by up to 24 unary operators. `+&A*-&B` is a valid operand for a SETA instruction. The expression `&FIELD+-` is invalid because it has no final term.

Evaluation of arithmetic expressions: The assembler evaluates arithmetic expressions during conditional assembly processing as follows:

1. It evaluates each arithmetic term.
2. It carries out arithmetic operations from left to right. However,
   a. It carries out unary operations before binary operations.
   b. It carries out the binary operations of multiplication and division before the binary operations of addition and subtraction.
   c. It carries out the binary operations of addition and subtraction before the bitwise logical operations.
   d. It carries out the bitwise logical operations before shift operations.
3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives a 0 result.
4. In parenthesized arithmetic expressions, the assembler evaluates the innermost expressions first, and then considers them as arithmetic terms in the next outer level of expressions. It continues this process until the outermost expression is evaluated.
5. The computed result, including intermediate values, must lie in the range -2^{31} through $+2^{31}-1$. (If the value -2^{31} is substituted in a SETC expression, its magnitude, 2147483648, is invalid if substituted in a SETA expression.)

SETC variables in arithmetic expressions: The assembler permits a SETC variable to be used as a term in an arithmetic expression if the character string value of the variable is a self-defining term. The value represented by the string is assigned to the arithmetic term. A null string is treated as zero.

Examples:

```
         LCLC            &C(5)
&C(1)    SETC            'B''101'''
&C(2)    SETC            'C''A'''
&C(3)    SETC            '23'
&A       SETA            &C(1)+&C(2)-&C(3)
```

In evaluating the arithmetic expression in the fifth statement, the first term, `&C(1)`, is assigned the binary value `101` (decimal `5`). To that is added the value represented by the EBCDIC character `A` (hexadecimal `C1`, which corresponds to decimal `193`). Then the value represented by the third term `&C(3)` is subtracted, and the value of `&A` becomes `5+193-23=175`.

This feature lets you associate numeric values with EBCDIC or hexadecimal characters to be used in such applications as indexing, code conversion, translation, and sorting.

Assume that `&X` is a character string with the value `ABC`.

```
&I       SETC            'C'''.'&X'(1,1).''''
&VAL     SETA            &TRANS(&I)
```

The first statement sets `&I` to `C'A'`. The second statement extracts the 193rd element of `&TRANS (C'A' = X'C1' = 193)`.

The following code converts a hexadecimal value in `&H` into a decimal value in `&VAL`:

```
&X       SETC            'X''&H'''
&VAL     SETA            &X
```

The following code converts the double-byte character `Da` into a decimal value in `&VAL`. `&VAL` can then be used to find an alternative code in a subscripted SETC variable:

```
&DA      SETC            'G''<Da>'''
&VAL     SETA            &DA
```

Although you can use a predefined absolute symbol as an operand in a SETA expression, you cannot substitute a SETC variable whose value is the same as the symbol. For example:

```
ABS      EQU    5
&ABS     SETA   ABS           &ABS has value 5
&CABS    SETC   'ABS'         &CABS has value 'ABS'
&ABS     SETA   &CABS         invalid usage
```

DBCS assembler option: The G-type self-defining term is valid only if the DBCS assembler option is specified.

Using SETA symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in an arithmetic expression. If the SETA symbol is not used in an arithmetic expression, the arithmetic value is converted to a character string containing its absolute value, with leading zeros removed. If the value is 0, it is converted to a single 0.

Example:

```
          MACRO
 &NAME    MOVE            &TO,&FROM
          LCLA            &A,&B,&C,&D
 &A       SETA            10                  Statement 1
 &B       SETA            12                  Statement 2
 &C       SETA            &A-&B               Statement 3
```

```
 &D       SETA            &A+&C               Statement 4
 &NAME    ST              2,SAVEAREA
          L               2,&FROM&C           Statement 5
          ST              2,&TO&D             Statement 6
          L               2,SAVEAREA
          MEND
 ---------------------------------------------------------------------
 HERE     MOVE            FIELDA,FIELDB
 ---------------------------------------------------------------------
+HERE ST             2,SAVEAREA
+    L               2,FIELDB2
+    ST              2,FIELDA8
+    L               2,SAVEAREA
```

Statements 1 and 2 assign the arithmetic values +10 and +12 to the SETA symbols &A and &B. Therefore, statement 3 assigns the SETA symbol &C the arithmetic value -2. When &C is used in statement 5, the arithmetic value -2 is converted to the character 2. When &C is used in statement 4, however, the arithmetic value -2 is used. Therefore, &D is assigned the arithmetic value +8. When &D is used in statement 6, the arithmetic value +8 is converted to the character 8.

The following example shows how the value assigned to a SETA symbol can be changed in a macro definition.

```
          MACRO
 &NAME    MOVE            &TO,&FROM
          LCLA            &A
 &A       SETA            5                   Statement 1
 &NAME    ST              2,SAVEAREA
          L               2,&FROM&A           Statement 2
 &A       SETA            8                   Statement 3
          ST              2,&TO&A             Statement 4
          L               2,SAVEAREA
          MEND
 ---------------------------------------------------------------------
 HERE     MOVE            FIELDA,FIELDB
 ---------------------------------------------------------------------
+HERE ST             2,SAVEAREA
+    L               2,FIELDB5
+    ST              2,FIELDA8
+    L               2,SAVEAREA
```

Statement 1 assigns the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the character 5. Statement 3 assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the character 8, instead of 5.

A SETA symbol can be used with a symbolic parameter to refer to an operand in an operand sublist. If a SETA symbol is used for this purpose, it must have been assigned a positive value.

Any expression that can be used in the operand field of a SETA instruction can be used to refer to an operand in an operand sublist. Sublists are described in "Sublists in operands" on page 266.

The following macro definition adds the last operand in an operand sublist to the first operand in an operand sublist and stores the result at the first operand. A sample macro instruction and generated statements follow the macro definition.

```
          MACRO
          ADDX            &NUMBER,&REG        Statement 1
          LCLA            &LAST
 &LAST    SETA            N'&NUMBER           Statement 2
          L               &REG,&NUMBER(1)
          A               &REG,&NUMBER(&LAST) Statement 3
          ST              &REG,&NUMBER(1)
          MEND
 ---------------------------------------------------------------------
```

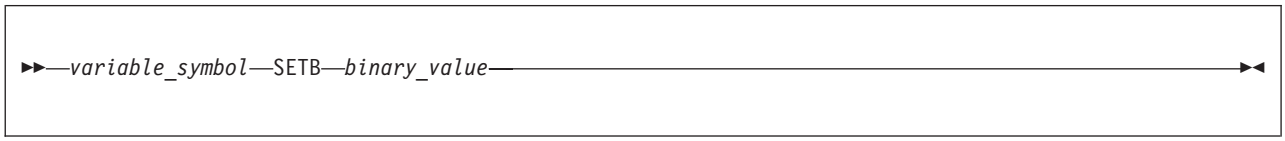
```
          ADDX            (A,B,C,D,E),3       Statement 4
 ------------------------------------------------------------------
+    L               3,A
+    A               3,E
+    ST              3,A
```

&NUMBER is the first symbolic parameter in the operand field of the prototype statement (statement 1). The corresponding characters (A,B,C,D,E) of the macro instruction (statement 4) are a sublist. Statement 2 assigns to &LAST the arithmetic value +5, which is equal to the number of operands in the sublist. Therefore, in statement 3, &NUMBER(&LAST) is replaced by the fifth operand of the sublist.

SETB instruction

Use the SETB instruction to assign a bit value to a SETB symbol. You can assign the bit values, 0 or 1, to a SETB symbol directly and use it as a switch.

If you specify a logical (Boolean) expression in the operand field, the assembler evaluates this expression to determine whether it is true or false, and then assigns the value 1 or 0 to the SETB symbol. You can use this computed value in condition tests or for substitution.



variable_symbol
Is a variable symbol.

A global variable symbol in the name field must have been previously declared as a SETB symbol in a GBLB instruction. Local SETB symbols need not be declared in an LCLB instruction. The assembler considers any undeclared variable symbol found in the name field of a SETB instruction as a local SET symbol. The variable symbol is assigned a type attribute value of N.

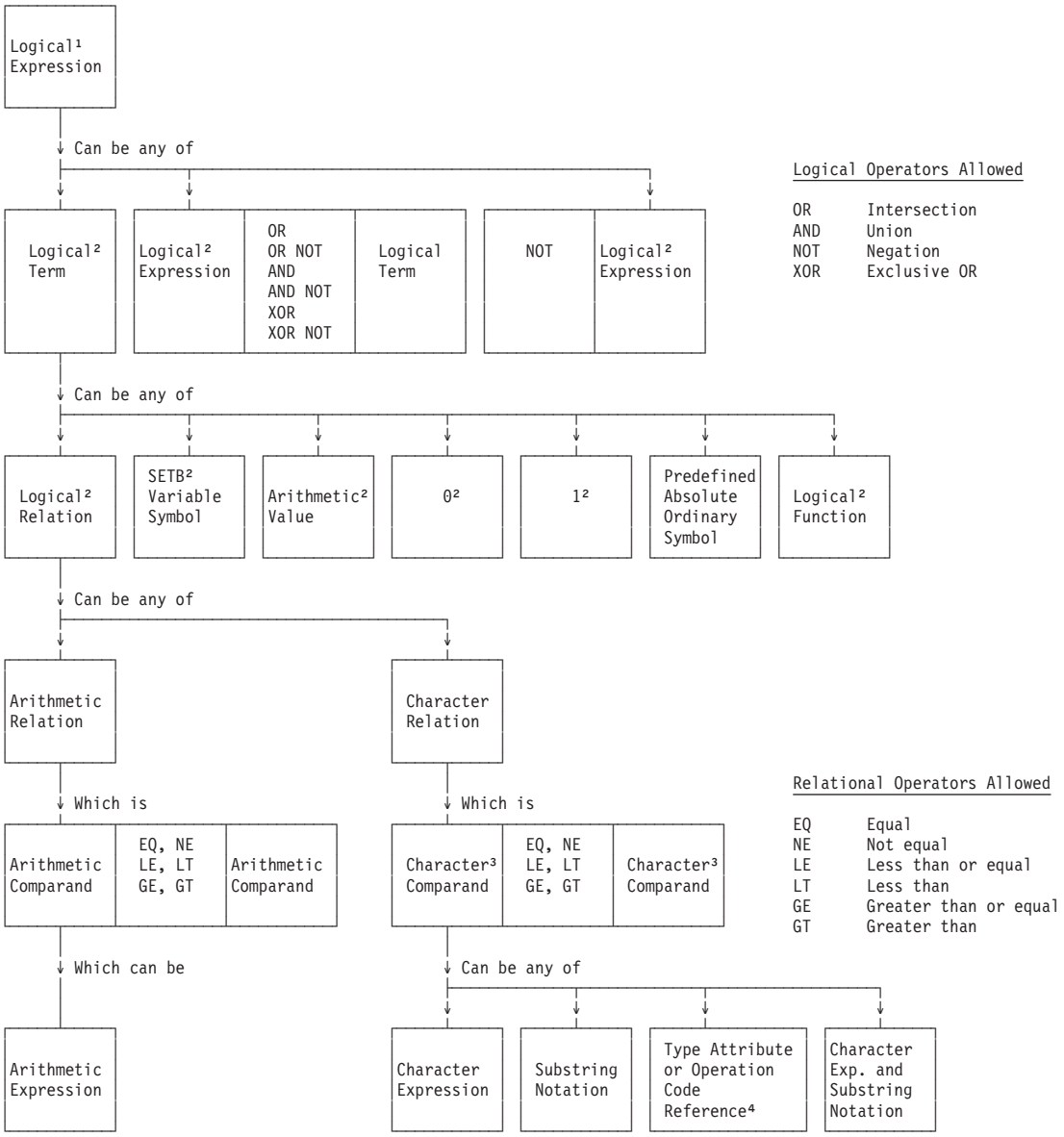
binary_value
Is a binary bit value specified as:

- A binary digit (0 or 1)
- A binary value enclosed in parentheses

  An arithmetic value enclosed in parentheses is allowed. This value can be represented by:
  - An unsigned self-defining term
  - A SETA symbol
  - A previously defined ordinary symbol with an absolute value
  - An attribute reference other than the type attribute reference.

  If the value is 0, the assembler assigns a value of 0 to the symbol in the name field. If the value is not 0, the assembler assigns a value of 1.
- A logical expression enclosed in parentheses

  A logical expression is evaluated to determine if it is true or false; the SETB symbol in the name field is then assigned the binary value 1 or 0, corresponding to true (1) or false (0). The assembler assigns the explicitly specified binary value (0 or 1) or the computed logical value (0 or 1) to the SETB symbol in the name field.



Notes:

1. Outermost expression must be enclosed in parentheses in SETB and AIF instructions.
2. Optional parentheses around terms and expressions at this level.
3. Must be in the range 0 through 1024 characters.
4. Must stand alone and not be enclosed in apostrophes.

Figure 46. Defining logical expressions

Rules for Coding Logical Expressions: The following is a summary of coding rules for logical expressions:

- A logical expression must not contain two logical terms in succession.
- A logical expression can contain two logical operators in succession; however, the only allowed combinations are OR NOT, XOR NOT and AND NOT. The two operators must be separated from each other by one or more spaces.
- Any logical term, relation, or inner logical expression can be optionally enclosed in parentheses.
- The relational and logical operators must be immediately preceded and followed by at least one space, except when written `(NOT bexpr)`.

- A logical expression can begin with the logical unary operator NOT.
- A logical expression can contain up to 18 logical operators. The relational and other operators used by the arithmetic and character expressions in relations do not count toward this total.
- Up to 255 levels of nested parentheses are allowed.
- Absolute ordinary symbols specified in logical expressions must be defined before the logical expression is encountered.
- The assembler determines the type of a logical relation by the first comparand. If the first comparand is a character expression that begins with an apostrophe, then the logical relation is a character relation, otherwise the assembler treats it as an arithmetic relation.

Subscripted SETB symbols

The SETB symbol in the name field can be subscripted. If the same SETB symbol has not been previously declared in a GBLB or LCLB instruction with an allowable dimension, and has not been implicitly declared in a SETB instruction as a scalar (unscripted)variable symbol, then the symbol is implicitly declared as a local SETB array variable.

The assembler assigns the binary value explicitly specified, or implicit in the logical expression present in the operand field, to the position in the declared array given by the value of the subscript. The subscript expression must not be 0 or have a negative value.

Logical (SETB) expressions

You can use a logical expression to assign a binary value to a SETB symbol. You can also use a logical expression to represent the condition test in an AIF instruction. This use lets you code a logical expression whose value (0 or 1) varies according to the values substituted into the expression and thus determine whether or not a branch is to be taken.

Figure 46 on page 322 defines a logical expression.

Logical expressions contain unquoted spaces that do *not* terminate the operand field. This is called "logical-expression format", and such expressions are always enclosed in parentheses.

A logical expression can consist of a logical expression and a logical term separated by a logical operator delimited by spaces. The logical operators are:

AND

Format: Logical-expression
Operands: Binary

Output: (`bexpr1 AND bexpr2`) has value 1, if each logical expression evaluates to 1, otherwise the value is 0.

Example

After the following statements `&VAR` contains the arithmetic value `0`.

```
Name      Operation      Operand

&OP1      SETB           1
&OP2      SETB           0
&VAR      SETB           (&OP1 AND &OP2)
```

AND NOT

Format: Logical-expression
Operands: Binary

Output: The value of the second logical term is inverted, and the expression is evaluated as though the AND operator was specified.

Example

`(1 AND NOT 0)` is equivalent to `(1 AND 1)`.

NOT

Format:
Logical-expression
Operands: Binary

Output: `NOT(bexp)` inverts the value of the logical expression.

OR

Format: Logical-expression
Operands: Binary

Output: `(bexp1 OR bexp2)` returns a value of 1, if either of the logical expressions contain or evaluate to 1. If they both contain or evaluate to 0 then the value is 0.

OR NOT

Format: Logical-expression
Operands: Binary

Output: `(bexp1 OR NOT bexp2)` inverts the value of the second logical term, and the expression is evaluated as though the OR operator was specified. For example, `(1 OR NOT 1)` is equivalent to `(1 OR 0)`.

XOR

Format: Logical-expression
Operands: Binary

Output: `(bexp1 XOR bexp2)` evaluates to 1 if the logical expressions contain or evaluate to opposite bit values. If they both contain or evaluate to the same bit value, the result is 0.

XOR NOT

Format: Logical-expression
Operands: Binary

Output: `(bexp1 XOR NOT bexp2)` inverts the second logical term, and the expression is evaluated as though the XOR operator was specified.

Example `(1 XOR NOT 1)` is equivalent to `(1 XOR 0)`.

Relational operators: Relational operators provide the means for comparing two items. A relational operator plus the items form a relation. An arithmetic relation is two arithmetic expressions separated by a relational operator, and a character relation is two character strings (for example, a character expression and a type attribute reference) separated by a relational operator.

The relational operators are:
EQ Equal
NE Not equal
LE Less than or equal

LT Less than
GE Greater than or equal
GT Greater than

Evaluation of logical expressions: The assembler evaluates logical expressions as follows:

1. It evaluates each logical term, which is given a binary value of 0 or 1.
2. If the logical term is an arithmetic or character relation, the assembler evaluates:
   a. The arithmetic or character expressions specified as values for comparison in these relations
   b. The arithmetic or character relation
   c. The logical term, which is the result of the relation. If the relation is true, the logical term it represents is given a value of 1; if the relation is false, the term is given a value of 0.

   The two comparands in a character relation are compared, character by character, according to binary (EBCDIC) representation of the characters. If two comparands in a relation have character values of unequal length, the assembler always takes the shorter character value to be less.

   Character comparisons are recognized by the presence of an opening apostrophe in the first operand. For example, if a character comparison involves a character function and a character constant, the constant must be written first, as in

   ```
   AIF  ('A' eq UPPER('a')).Okay
   ```
3. The assembler carries out logical operations from left to right. However,
   a. It carries out logical NOTs before logical ANDs, ORs, and XORs
   b. It carries out logical ANDs before logical ORs and XORs
   c. It carries out logical ORs before logical XORs
4. In parenthesized logical expressions, the assembler evaluates the innermost expressions first, and then considers them as logical terms in the next outer level of expressions. It continues this process until it evaluates the outermost expression.

Using SETB symbols: The logical value assigned to a SETB symbol is used for the SETB symbol appearing in the operand field of an AIF instruction or another SETB instruction.

If a SETB symbol is used in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of AIF and SETB instructions, the binary values 1 (true) and 0 (false) are converted to the arithmetic values 1 and 0.

If a SETB symbol is used in the operand field of a SETC instruction, in character relations in the operand fields of AIF and SETB instructions, or in any other statement, the binary values 1 (true) and 0 (false), are converted to the character values `'1'` and `'0'`.

The following example illustrates these rules. It assumes that `(L'&TO EQ 4)` is true, and `(S'&TO EQ 0)` is false.

```
         MACRO
 &NAME   MOVE            &TO,&FROM
         LCLA            &A1
         LCLB            &B1,&B2
         LCLC            &C1
 &B1     SETB            (L'&TO EQ 4)        Statement 1
 &B2     SETB            (S'&TO EQ 0)        Statement 2
 &A1     SETA            &B1                 Statement 3
 &C1     SETC            '&B2'               Statement 4
         ST              2,SAVEAREA
         L               2,&FROM&A1
         ST              2,&TO&C1
         L               2,SAVEAREA
         MEND
 ---------------------------------------------------------------------
 HERE    MOVE            FIELDA,FIELDB
 ---------------------------------------------------------------------
```

```
+HERE ST             2,SAVEAREA
+    L               2,FIELDB1
+    ST              2,FIELDA0
+    L               2,SAVEAREA
```

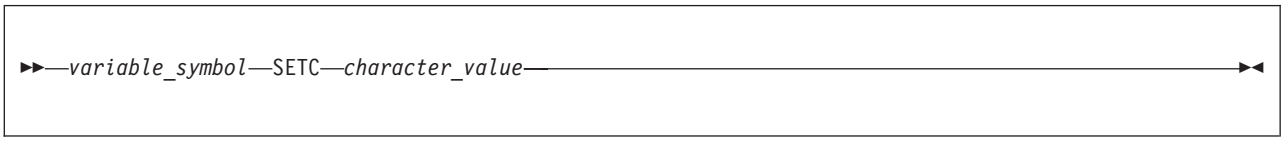
Because the operand field of statement 1 is true, &B1 is assigned the binary value 1. Therefore, the arithmetic value +1 is substituted for &B1 in statement 3. Because the operand field of statement 2 is false, &B2 is assigned the binary value 0. Therefore, the character value 0 is substituted for &B2 in statement 4.

SETC instruction

The SETC instruction assigns a character value to a SETC symbol. You can assign whole character strings, or concatenate several smaller strings together. The assembler assigns the composite string to your SETC symbol. You can also assign parts of a character string to a SETC symbol by using the substring notation; see "Substring notation" on page 328.

A character string consists of any combination of characters enclosed in apostrophes. Variable symbols are allowed. The assembler substitutes the representation of their values as character strings into the character expression before evaluating the expression. Up to 1024 characters are allowed in a character expression.

You can change the character value assigned to a SETC symbol. This lets you use the same SETC symbol with different values for character comparisons in several places, or for substituting different values into the same model statement.



variable symbol
Is a variable symbol.

A global variable symbol in the name field must have been previously declared as a SETC symbol in a GBLC instruction. Local SETC symbols need not be declared in an LCLC instruction. The assembler considers any undeclared variable symbol found in the name field of a SETC instruction as a local SET symbol. The variable symbol is assigned a type attribute value of U.

character_value
Is a character value specified by:
- An operation code attribute reference
- A type attribute reference
- A character expression

The assembler assigns the character string value represented in the operand field to the SETC symbol in the name field. The string length must be in the range 0 (null character string) through 1024 characters.

When a SETA or SETB symbol is specified in a character expression, the unsigned decimal value of the symbol (with leading zeros removed) is the character value given to the symbol.

A duplication factor can precede a character expression or substring notation. The duplication factor can be any non-negative arithmetic expression allowed in the operand of a SETA instruction. For example:

```
&C1      SETC           (3)'ABC'
```

assigns the value 'ABCABCABC' to &C1.

A zero duplication factor results in a null (zero-length) string.

Notes:

1. The assembler evaluates the represented character string (in particular, the substring; see "Substring notation" on page 328) before applying the duplication factor. The resulting character string is then assigned to the SETC symbol in the name field. For example:

   ```
   &C2      SETC            'ABC'.(3)'ABCDEF'(4,3)
   ```

   assigns the value `'ABCDEFDEFDEF'` to `&C2`.
2. If the character string contains double-byte data, then redundant SI/SO pairs are not removed on duplication. For example:

   ```
   &C3      SETC            (3)'<.A.B>'
   ```

   assigns the value `'<.A.B><.A.B><.A.B>'` to `&C3`.
3. To duplicate double-byte data, without including redundant SI/SO pairs, use the substring notation. For example:

   ```
   &C4      SETC            (3)'<.A.B>'(2,4)
   ```

   assigns the value `'.A.B.A.B.A.B'` to `&C4`.
4. To duplicate the arithmetic value of a previously defined ordinary symbol with an absolute value, first assign the arithmetic value to a SETA symbol. For example:

   ```
   A        EQU             123
   &A1      SETA            A
   &C5      SETC            (3)'&A1'
   ```

   assigns the value `'123123123'` to `&C5`.

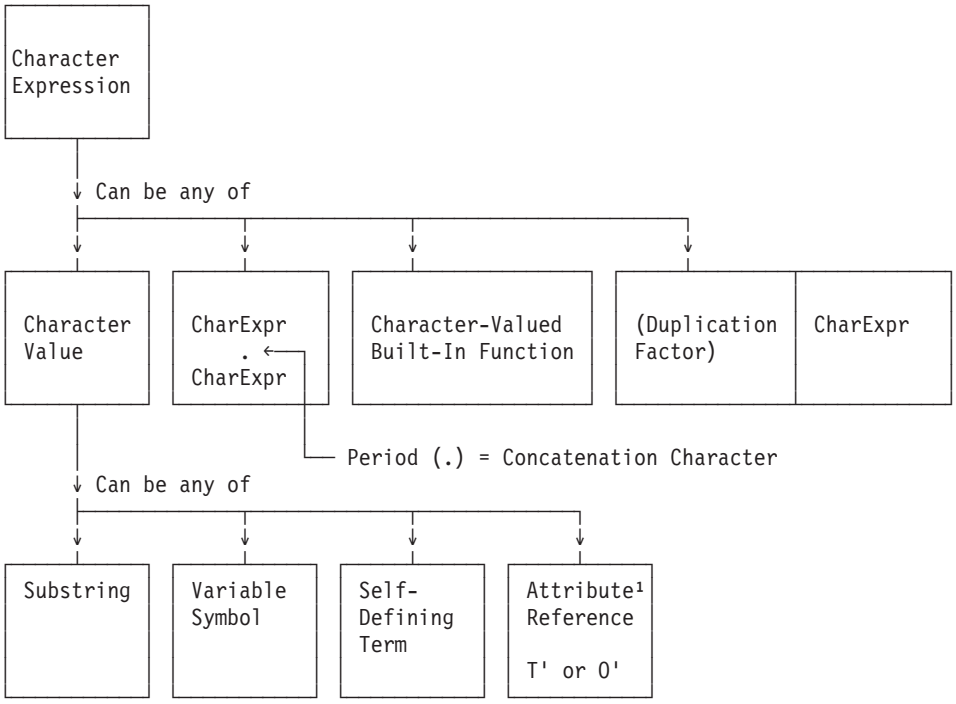


Figure 47. Defining character (SETC) expressions

Note:

1. The attribute reference term must not be preceded by a duplication factor.

Subscripted SETC symbols

The SETC symbol (see 1 in Figure 48) in the name field can be subscripted. If the same SETC symbol has not been previously declared in a GBLC or LCLC instruction with an allowable dimension (see 2 in Figure 48), or has been implicitly declared in a SETC instruction as a scalar (unscripted) variable symbol, then the symbol is implicitly declared as a local SETC array variable.

The assembler assigns the character value represented in the operand field to the position in the declared array (see 3 in Figure 48) given by the value of the subscript. The subscript expression must not be 0 or have a negative value.

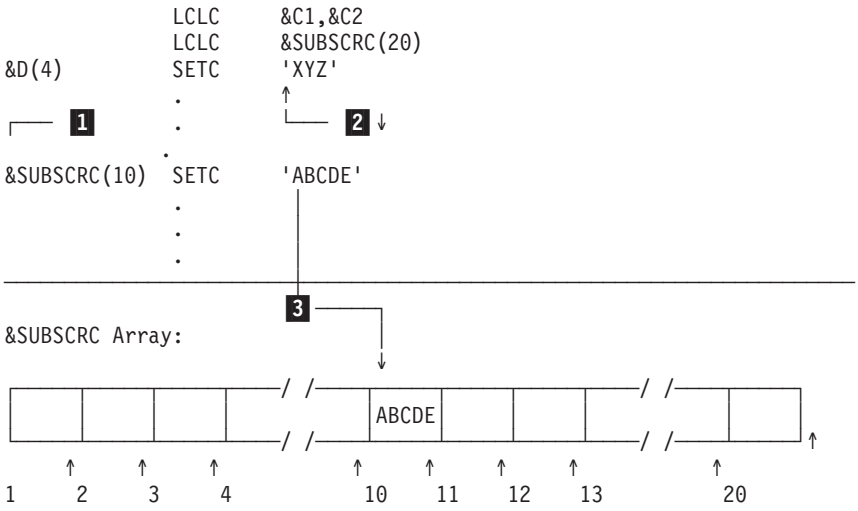


Figure 48. Subscripted SETC symbols

Character (SETC) expressions

The main purpose of a character expression is to assign a character value to a SETC symbol. You can then use the SETC symbol to substitute the character string into a model statement.

You can also use a character expression as a value for comparison in condition tests and logical expressions. Also, a character expression provides the string from which characters can be selected by the substring notation.

Substitution of one or more character values into a character expression lets you use the character expression wherever you need to vary values for substitution or to control loops.

An attribute reference must be the only term in a character expression.

Substring notation: The substring notation lets you refer to one or more characters within a character string. You can, therefore, either select characters from the string and use them for substitution or testing, or scan through a complete string, inspecting each character. By concatenating substrings with other substrings or character strings, you can rearrange and build your own strings.

The substring notation can be used only in conditional assembly instructions. Table 60 on page 329 shows how to use the substring notation.

Table 60. Substring notation in conditional assembly instructions

Used in	Used as	Example	Value assigned to SETC Symbol
SETC instruction operand	Operand	`&C1 SETC 'ABC'(1,3)`	ABC
		`&C2 SETC '&C1'(1,2).'DEF'`	
	Part of operand		ABDEF
AIF or SETB instruction operand (logical expression)	Character value in comparand of character relation	`AIF ('&STRING'(1,4) EQ 'AREA').SEQ` `&B SETB ('&STRING'(1,4).'9' EQ 'FULL9')`	---

The substring notation must be specified as follows:

`'CHARACTER STRING'(`*e1*`,`*e2*`)`

where the `CHARACTER STRING` is a character expression from which the substring is to be extracted. The first subscript (*e1*) shows the position of the first character that is to be extracted from the character string. The second subscript (*e2*) shows the number of characters to be extracted from the character string, starting with the character indicated by the first subscript. Thus, the second subscript specifies the length of the resulting substring.

The second subscript value of the substring notation can be specified as an asterisk (*), to indicate that all the characters beginning at the position of the first expression are used. The extracted string is equal to the length of the character expression, less the number of characters before the starting character.

The character string must be a valid character expression with a length, *n*, in the range 1 through 1024 characters. The length of the resulting substring must be in the range 0 through 1024.

The subscripts, *e1* and *e2*, must be arithmetic expressions.

When you use subscripted variable symbols in combination with substring notation, take care to distinguish variable subscripts from substring-operation subscripts.

```
       LCLC   &DVAR(10),&SVAR,&C(10)
&C(1)  SETC   '&DVAR(5)'            Select 5th element of &DVAR
&C(2)  SETC   '&SVAR'(1,3)          Select substring of &SVAR
&C(3)  SETC   '&DVAR(5)'(1,3)       Select substring of &DVAR(5)
&C(4)  SETC   '&SYSLIST(1,3)'(1,3)  Select substring of &SYSLIST(1,3)
```

Evaluation of substrings: The following examples show how the assembler processes substrings depending on the value of the elements *n*, *e1*, and *e2*.

- In the usual case, the assembler generates a correct substring of the specified length:

```
                Value of Variable   Character Value
Notation        Symbol              of Substring

'ABCDE'(1,5)                        ABCDE
'ABCDE'(2,3)                        BCD
'ABCDE'(2,*)                        BCDE
'ABCDE'(4,*)                        DE
'&C'(3,3)       ABCDE               CDE
'&PARAM'(3,3)   ((A+3)*10)          A+3
```

- When *e1* has a zero or negative value, the assembler generates a null string and issues error message `ASMA093E`.

```
                  Character Value
Notation          of Substring

'ABCDE'(0,5)      null character string
'ABCDE'(0,*)      null character string
```

- When the value of *e1* exceeds *n*, the assembler generates a null string and issues error message ASMA092E.

```
                 Value of Variable   Character Value
Notation         Symbol              of Substring

'ABCDE'(7,3)                         null character string
'ABCDE'(6,*)                         null character string
```

- When *e2* has a value less than one, the assembler generates the null character string. If *e2* is negative, the assembler also issues error message ASMA095W.

```
                 Value of Variable   Character Value
Notation         Symbol              of Substring

'ABCDE'(4,0)                         null character string
'ABCDE'(3,-2)                        null character string
```

- When *e2* indexes past the end of the character expression (that is, *e1*+*e2* is greater than *n*+1), the assembler issues warning message ASMA094I, and generates a substring that includes only the characters up to the end of the character expression specified.

```
                 Value of Variable   Character Value
Notation         Symbol              of Substring

'ABCDE'(3,5)                         CDE
```

Figure 49 shows the results of an assembly of SETC instructions with different substring notations.

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0       2008/07/11 17.48
                                      8 &STRING  SETC  'STRING'                                            00008000
                                      9 &SUBSTR1 SETC  '&STRING'(0,4)                                      00009000
** ASMA093E Substring expression 1 less than 1; default=null - OPENC
                                     10 &SUBSTR2 SETC  '&STRING'(7,4)                                      00010000
** ASMA092E Substring expression 1 points past string end; default=null - OPENC
                                     11 &SUBSTR3 SETC  '&STRING'(3,0)                                      00011000
                                     12 &SUBSTR4 SETC  '&STRING'(3,-2)                                     00012000
** ASMA095W Substring expression 2 less than 0; default=null - OPENC
                                     13 &SUBSTR5 SETC  '&STRING'(3,4)                                      00013000
                                     14 &SUBSTR6 SETC  '&STRING'(3,5)                                      00014000
** ASMA094I Substring goes past string end; default=remainder
                                     15          END                                                       00015000
```

Figure 49. Sample assembly using substring notation

You can suppress the ASMA094I message by specifying the FLAG(NOSUBSTR) option or by setting the ACONTROL FLAG(NOSUBSTR) value. When this is done, the listing changes (Figure 50 on page 331).

```
  Loc  Object Code    Addr1 Addr2  Stmt   Source Statement                                HLASM R6.0       2008/07/11 17.48
                                      7          ACONTROL FLAG(NOSUBSTR)
                                      8 &STRING  SETC  'STRING'                                            00008000
                                      9 &SUBSTR1 SETC  '&STRING'(0,4)                                      00009000
** ASMA093E Substring expression 1 less than 1; default=null - OPENC
                                     10 &SUBSTR2 SETC  '&STRING'(7,4)                                      00010000
** ASMA092E Substring expression 1 points past string end; default=null - OPENC
                                     11 &SUBSTR3 SETC  '&STRING'(3,0)                                      00011000
                                     12 &SUBSTR4 SETC  '&STRING'(3,-2)                                     00012000
** ASMA095W Substring expression 2 less than 0; default=null - OPENC
                                     13 &SUBSTR5 SETC  '&STRING'(3,4)                                      00013000
                                     14 &SUBSTR6 SETC  '&STRING'(3,5)                                      00014000
                                     15          END                                                       00015000
```

Figure 50. Sample assembly using substring notation with messages suppressed

Character (SETC) expressions can be used only in conditional assembly instructions. Table 61 shows examples of using character expressions.

Table 61. Use of character expressions

Used in	Used as	Example
SETC instruction	Operand	`&C  SETC  'STRING0'`
AIF or SETB instruction	Character string in character relation	`AIF  ('&C' EQ 'STRING1').B`
Substring notation	First part of notation	`'SELECT'(2,5) returns 'ELECT'`
Built-in functions	Operand	`&VAR  SETC  (LOWER '&twenty.&six')` `&AB   SETA  A2B('10')`

Character-valued built-in functions: Character-valued built-in functions have arithmetic-only operands, character-only operands, or both arithmetic and character operands. Each type is described in a separate section. The maximum string length of any SETC variable is 1024 bytes. If this length is exceeded, the string value is truncated, and message ASMA091E is generated.

The following discussion uses these special notations:

n The EBCDIC character containing all 0 bits.

f The EBCDIC character containing all 1 bits.

Here are the SETC built-in functions:

A2B

Format: Function-invocation
Operands: Arithmetic

Output: `A2B(aexpr)` converts the value of its arithmetic argument to a string of 32 zero (`'0'`) and one (`'1'`) characters. The value of `aexpr` must be representable as a 32 bit binary integer. If the `aexpr` argument is negative, the result contains 32 characters, the first of which is `'1'`.

Examples

```
A2B(0)           has value '00000000000000000000000000000000'
A2B(5)           has value '00000000000000000000000000000101'
A2B(1022)        has value '00000000000000000000001111111110'
A2B(-7)          has value '11111111111111111111111111111001'
A2B(2345678901)  indicates an error (value too large)
```

A2C

Format: Function-invocation
Operands: Arithmetic

Output: A2C(aexpr) converts the value of its arithmetic argument to a string of four characters whose bit pattern is the same as the argument's.

Examples

```
A2C(0)            has value 'nnnn' (4 EBCDIC nulls)
A2C(241)          has value 'nnn1'
A2C(20046)        has value 'nn++'
A2C(-252645136)   has value '0000'
```

A2D

Format: Function-invocation
Operands: Arithmetic

Output: A2D(aexpr) converts the value of its arithmetic argument to a string of decimal digits preceded by a plus or minus sign.

Note: The A2D function is like the SIGNED function, except that A2D always provides an initial sign character.

Examples

```
A2D(0)            has value '+0'
A2D(241)          has value '+241'
A2D(16448)        has value '+16448'
A2D(-3)           has value '-3'
```

A2X

Format: Function-invocation
Operands: Arithmetic

Output: A2X(aexpr) converts the value of its arithmetic argument to a string of eight hexadecimal characters.

Examples

```
A2X(0)            has value '00000000'
A2X(10)           has value '0000000A'
A2X(257)          has value '00000101'
A2X(1022)         has value '000003FE'
A2X(-7)           has value 'FFFFFFF9'
```

B2C

Format: Function-invocation
Operands: Character

Output: B2C('bitstring') converts the bit-string character argument to characters representing the same bit pattern. Null arguments return a null string.

If needed, the argument string is padded internally on the left with zeros so that its length is a multiple of eight.

The operand must contain only *ones* and *zeros*. Any other value causes the message ASMA214E to be generated.

Examples

```
B2C('11110011')        has value '3'
B2C('101110011110001') has value '*1'
B2C('0')               has value '0' (EBCDIC null character)
B2C('00010010001')     has value '0j'
B2C('000000000')       has value '00' (two EBCDIC nulls)
B2C('')                has value ''  (null string)
```

B2D

Format: Function-invocation
Operands: Character

Output: `B2D('bitstring')` converts a bit-string argument of at most 32 `'0'` and `'1'` characters to one to ten decimal characters preceded by a plus or minus sign, representing the value of the argument. Null arguments return `'+0'`.

Examples

```
B2D('')                has value '+0'
B2D('00010010001')     has value '+145'
B2D('11110001')        has value '+241'
B2D('01111111111111111111111111111111') has value '+2147483647'
B2D('11111111111111111111111111110001') has value '-15'
```

B2X

Format: Function-invocation
Operands: Character

Output: `B2X('bitstring')` converts the bit-string argument to hexadecimal characters representing the same bit pattern. Null arguments return a null string.

If needed, the argument string is padded internally on the left with zeros so that its length is a multiple of four.

The operand must contain only *ones* and *zeros*. Any other value causes the message ASMA214E to be generated.

Examples

```
B2X('')                  has value ''  (null string)
B2X('00000')             has value '00'
B2X('0000010010001')     has value '0091'
B2X('11110001')          has value 'F1'
B2X('1111110001')        has value '3F1'
```

BYTE

Format: Logical-expression, function-invocation
Operands: Arithmetic

Output: `BYTE(aexpr)` or `(BYTE aexpr)` returns a one-character EBCDIC character expression in which the binary value of the character is specified by the arithmetic argument. The argument must have a value 0 - 255.

This function might be used to introduce characters which are not on the keyboard.

Examples

```
BYTE(0)          has value '0' (EBCDIC null character)
BYTE(97)         has value '/'
BYTE(129)        has value 'a'
```

C2B

Format: Function-invocation
Operands: Character

Output: `C2B('charstring')` converts the character argument to a string of `'0'` and `'1'` characters representing the same bit pattern. Null arguments return a null string.

If the result is not too long, the length of the result is eight times the length of the `'charstring'` argument.

Examples

```
C2B('')           has value ''
C2B('n')          has value '00000000'
C2B(' ')          has value '01000000'
C2B('1')          has value '11110001'
C2B('1234')       has value '11110001111100101111001111110100'
```

C2D

Format: Function-invocation
Operands: Character

Output: `C2D('charstring')` converts a character-string argument of at most four characters to one to ten decimal characters preceded by a plus or minus sign, representing the numeric value of the argument. Null arguments return `'+0'`.

Examples

```
C2D('')            has value '+0'
C2D('nj')          has value '+145'
C2D('1')           has value '+241'
C2D('0000')        has value '-252645136
```

C2X

Format: Function-invocation
Operands: Character

Output: `C2X('charstring')` converts the character-string argument to hexadecimal characters representing the same bit pattern. Null arguments return a null string.

If the result is not too long, the length of the result is two times the length of the `'charstring'` argument.

Examples

```
C2X('')           has value ''
C2X('n')          has value '00'
C2X('1')          has value 'F1'
C2X('a')          has value '81'
C2X('1234567R')   has value 'F1F2F3F4F5F6F7D9'
```

D2B

Format: Function-invocation
Operands: Character

Output: `D2B('decstring')` converts an argument string of optionally signed decimal characters to a string of 32 `'0'` and `'1'` characters representing a bit string with the same binary value. The value of `decstring` must be representable as a 32 bit binary integer. A null argument string returns a null string.

Examples

```
D2B('')          has value ''
D2B('0')         has value '00000000000000000000000000000000'
D2B('+5')        has value '00000000000000000000000000000101'
D2B('1022')      has value '00000000000000000000001111111110'
D2B('-7')        has value '11111111111111111111111111111001'
```

D2C

Format: Function-invocation
Operands: Character

Output: `D2C('decstring')` converts an argument string of optionally signed decimal characters to a string of four characters whose byte values represent the same binary value. The value of `decstring` must be representable as a 32 bit binary integer. The argument string must not be null.

Examples

```
D2C('')          indicates an error
D2C('0')         has value 'nnnn' (4 EBCDIC null bytes)
D2C('126')       has value 'nnn='
D2C('247')       has value 'nnn7'
D2C('23793')     has value 'nn*1'
D2C('-7')        has value 'fff9' (f=byte of all 1 bits)
```

D2X

Format: Function-invocation
Operands: Character

Output: `D2X('decstring')` converts an argument string of optionally signed decimal characters to a string of eight hexadecimal characters whose digits represent the same hexadecimal value. The value of `decstring` must be representable as a 32 bit binary integer. The argument string must not be null.

Examples

```
D2X('')          indicates an error
D2X('0')         has value '00000000'
D2X('+5')        has value '00000005'
D2X('255')       has value '000000FF'
D2X('01022')     has value '000003FE'
D2X('-7')        has value 'FFFFFFF9'
DSX('2345678901') causes an error condition (value too large)
```

DCVAL

Format: Function-invocation
Operands: Character

Output: `DCVAL('cexpr')` performs a single scan of the argument string to find successive pairs of apostrophes and ampersands, and returns a string value in which each such pair has been replaced by a single occurrence. This pairing action occurs only once; that is, three successive occurrences of an apostrophe or ampersand result in two occurrences, not one. A null argument is returned unchanged.

DCVAL is like DCLEN, except that DCLEN returns only the length of the result, not the paired string.

Examples

```
DCVAL('')        has value "" (null string)
DCVAL('''')      has value "'" (single apostrophe)
DCVAL('&&')      has value "&" (single ampersand)
DCVAL('a''''b')  has value "a'b"
```

```
        DCVAL('a''''b&&c')   has value "a'b&c"
|       .* Suppose &C has value "&&&&''''" (4 ampersands, 4 apostrophes)
|       &X SETC DCVAL('&C') &X has value "&&''" (2 of each)
```

DEQUOTE

Format: Function-invocation
Operands: Character

Output: `DEQUOTE('cexpr')` removes a single occurrence of an apostrophe from each end of the argument string, if any are present. A null argument is returned unchanged.

Examples

```
&C   SETC  DEQUOTE('charstring')    &C has value "charstring"
&C   SETC  DEQUOTE('')              &C is a null string
&C   SETC  DEQUOTE('a')             &C has value "a"
&ARG SETC  '''a'''                  &ARG has value "'a'"
&C   SETC  DEQUOTE('&ARG')          &C has value "a"
&C   SETC  DEQUOTE('a''b')          &C has value "a'b"
&ARG SETC  ''''''                   &ARG has value "''"
&C   SETC  DEQUOTE('&ARG')          &C has value "" (null string)
```

DOUBLE

Format: Logical-expression, function-invocation
Operands: Character

Output: `DOUBLE('cexpr')` or `(DOUBLE 'cexpr')` converts each occurrence of an apostrophe or ampersand character in the argument string to a pair of apostrophes and ampersands. In this form, the string is suitable for substitution into statements such as DC and MNOTE. Null arguments return a null string. An error condition is detected if the resulting string is too long.

Examples

Suppose the SETC variable &C contains the characters `"&&''&"` (two apostrophes, three ampersands):

```
DOUBLE('&C')          has value "&&&&''''&&"
```

LOWER

Format: Logical-expression, function-invocation
Operands: Character

Output: `LOWER('cexpr')` or `(LOWER 'cexpr')` converts the alphabetic characters A-Z in the argument to lowercase, a-z. Null arguments return a null string.

Examples

```
LOWER('aBcDefG')      has value 'abcdefg'
```

SIGNED

Format: Logical-expression, function-invocation
Operands: Arithmetic

Output: `SIGNED(aexpr)` or `(SIGNED aexpr)` converts its arithmetic argument to a decimal character string representation of its value, with a leading minus sign if the argument is negative.

Examples

```
SIGNED(10)        has value '10'
SIGNED(-10)       has value '-10'
```

Note: The SIGNED function creates properly signed values for display, whereas assigning a SETA value to a SETC variable produces only the magnitude of the SETA value. For example:

```
&A   SETA  10     &A has value 10
&C   SETC  '&A'   &C has value '10'
&A   SETA  -10    &A has value -10
&C   SETC  '&A'   &C has value '10' (unsigned)
```

SYSATTRA

Format: Function-invocation
Operands: Character

Output: `SYSATTRA('symbol')` returns the assembler-type value for the specified symbol.

- The 1 to 4 character assembler type is returned, with trailing spaces removed. For symbols defined in DC and DS statements, the assembler type includes the type extensions, if any.
- Symbols without an assigned assembler type, undefined symbols, and null arguments return null.

Examples

Given that symbol `Sym1` has previously been assigned an assembler type of `GR`, and variable symbol &SName has a value of `SYM1`, then:

```
SYSATTRA('Sym1')       has value 'GR'
SYSATTRA('&SName')     has value 'GR'
```

SYSATTRP

Format: Function-invocation
Operands: Character

Output: `SYSATTRP('symbol')` returns the program-type value for the specified symbol.

- The 4 byte program type is returned.
- Symbols without an assigned program type, undefined symbols, and null arguments return null.

Examples

Given that symbol `Sym1` has previously been assigned a program type of “Box7”, and variable symbol &SName has a value of `SYM1`, then:

```
SYSATTRP('Sym1')       has value 'Box7'
SYSATTRP('&SName')     has value 'Box7'
```

UPPER

Format: Logical-expression, function-invocation
Operands: Character

Output: `UPPER('cexpr')` or `(UPPER 'cexpr')` converts the alphabetic characters a-z in the argument to uppercase, A-Z. Null arguments return a null string.

Examples

```
UPPER('aBcDefG')       has value 'ABCDEFG'
```

X2B

Format: Function-invocation
Operands: Character

Output: X2B('hexstring') converts the value of its argument string of hexadecimal characters to a character string containing only zero ('0') and one ('1') characters representing the same bit pattern. Null arguments return a null string.

If the result is not too long, the length of the result is four times the length of the 'hexstring' argument.

The operand must contain only *hexadecimal digits*. Any other value causes the message ASMA214E to be generated.

Examples

```
X2B('')           has value '' (null string)
X2B('00')         has value '00000000'
X2B('1')          has value '0001'
X2B('F3')         has value '11110011'
X2B('00F3')       has value '0000000011110011'
```

X2C

Format: Function-invocation
Operands: Character

Output: X2C('hexstring') converts the hexstring argument to characters representing the same bit pattern. Null arguments return a null string.

If needed, the argument string is padded internally on the left with a zero character so that its length is a multiple of two.

The operand must contain only *hexadecimal digits*. Any other value causes the message ASMA214E to be generated.

Examples

```
X2C('')           has value '' (null string)
X2C('F3')         has value '3'
X2C('0')          has value 'n' (EBCDIC null character)
X2C('F1F2F3F4F5') has value '12345'
X2C('000F1')      has value 'nn1'
```

X2D

Format: Function-invocation
Operands: Character

Output: X2D('hexstring') converts its argument string of at most eight hexadecimal characters to one to ten decimal characters preceded by a plus or minus sign, representing the value of the argument. Null arguments return '+0'. For example:

```
X2D('')           has value '+0'
X2D('91')         has value '+145'
X2D('000F1')      has value '+241'
X2D('7FFFFFFF')   has value '+2147483647'
X2D('FFFFFFF1')   has value '-15'
```

Evaluation of character expressions: The value of a character expression is the character string within the enclosing apostrophes, after the assembler carries out any substitution for variable symbols.

Character strings, including variable symbols, can be concatenated to each other within a character expression. The resultant string is the value of the expression.

Notes:

1. Use two apostrophes to generate a single apostrophe as part of the value of a character expression.

The following statement assigns the character value L'SYMBOL to the SETC symbol &LENGTH.

```
&LENGTH  SETC            'L''SYMBOL'
```

2. A double ampersand generates a double ampersand as part of the value of a character expression. To generate a single ampersand in a character expression, use the substring notation; for example:

   ```
   &AMP     SETC            '&&'(1,1)
   ```

   Note: A quoted single ampersand '&' is not a valid character string.

   The following statement assigns the character value HALF&& to the SETC symbol &AND.

   ```
   &AND     SETC            'HALF&&'
   ```

   This is the only instance when the assembler does not pair ampersands to produce a single ampersand. However, if you substitute a SETC symbol with such a value into the nominal value in a DC instruction operand, or the operand of an MNOTE instruction, when the assembler processes the DC or MNOTE instruction, it pairs the ampersands and produces a single ampersand.

3. To generate a period, two periods must be specified after a variable symbol.

   For example, if &ALPHA has been assigned the character value AB%4, the following statement can be used to assign the character value AB%4.RST to the variable symbol &GAMMA.

   ```
   &GAMMA   SETC            '&ALPHA..RST'
   ```

4. To generate a period, the variable symbol must have a period as part of its value. For example:

   ```
   &DOT    SETC  '.'
   &DELTA  SETC  'A&DOT.&DOT'    &DELTA has value 'A..'
   ```

5. Double-byte data can appear in the character string if the assembler is invoked with the DBCS option. The double-byte data must be bracketed by the SO and SI delimiters, and the double-byte data must be valid.
6. The DBCS ampersand and apostrophe are not recognized as delimiters.
7. A double-byte character that contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.
8. Duplication (replication) factors are permitted before character built-in functions.
9. Releases of HLASM prior to Version 1 Release 4 permitted predefined absolute symbols in character expressions. To remove inconsistencies when handling character and arithmetic expressions such usage is no longer permitted and results in message ASMA137S if attempted. The built-in function BYTE can be used to convert a numeric value in a character expression as shown.

   ```
   RPTDS      EQU     X'01'
   &RPTC1     SETC    'SEND  '.(BYTE RPTDS)
   ```

Concatenation of character string values: Character expressions can be concatenated to each other or to substring notations in any order. The resulting value is a character string composed of the concatenated parts. This concatenated string can then be used in the operand field of a SETC instruction, or as a value for comparison in a logical expression.

You need the concatenation character (a period) to separate the apostrophe that ends one character expression from the apostrophe that begins the next.

For example, either of the following statements can be used to assign the character value ABCDEF to the SETC symbol &BETA.

```
&BETA    SETC            'ABCDEF'
&BETA    SETC            'ABC'.'DEF'
```

Concatenation of strings containing double-byte data: If the assembler is invoked with the DBCS option, then the following additional considerations apply:

- When a variable symbol adjoins double-byte data, the SO delimiting the double-byte data is not a valid delimiter of the variable symbol. The variable symbol must be terminated by a period.

- The assembler checks for SI and SO at concatenation points. If the byte to the left of the join is SI and the byte to the right of the join is SO, then the SI/SO pair is considered redundant and are removed.
- To create redundant SI/SO pairs at concatenation points, use the substring notation and SETC expressions to create additional SI and SO characters. By controlling the order of concatenation, you can leave a redundant SI/SO pair at a concatenation point.

  Instead of substring notation, you can use the BYTE function to create additional SI and SO characters:

```
&SO   SETC (BYTE 14)
&SI   SETC (BYTE 15)
```

Examples:

```
&DBDA    SETC            '<Da>'
&SO      SETC            BYTE(X'0E')
&SI      SETC            BYTE(X'0F')
&DBCS1A  SETC            '&DBDA.<Db>'
&DBCS1E  SETC            '&DBDA<Db>'
&DBCS2   SETC            '&DBDA'.'<Db>'
&DBCS2A  SETC            '&DBDA'.'<Db>'.'&DBDA'
&DBCS3   SETC            '&DBDA'.'&SI'.'&SO'.'<Db>'
&DBCS3P  SETC            '&DBDA'.'&SI'
&DBCS3Q  SETC            '&SO'.'<Db>'
&DBCS3R  SETC            '&DBCS3P'.'&DBCS3Q'
```

These examples use the BYTE function to create variables &SO and &SI, which have the values of SO and SI. The variable &DBCS1A is assigned the value **<**DaDb**>** with the SI/SO pair at the join removed. The assignment to variable &DBCS1E fails with error ASMA035E Invalid delimiter, because the symbol &DBDA is terminated by SO and not by a period. The variable &DBCS2 is assigned the value **<**DaDb**>**. The variable &DBCS2A is assigned the value **<**DaDbDa**>**. As with &DBCS1A, redundant SI/SO pairs are removed at the joins. The variable &DBCS3 is assigned the value **<**DaDb**>**. Although SI and SO have been added at the join, the concatenation operation removes two SI and two SO characters, since redundant SI/SO pairs are found at the second and third concatenations. However, by using intermediate variables &DBCS3P and &DBCS3Q to change the order of concatenation, the string **<**Da**><**Db**>** can be assigned to variable &DBCS3R. Substituting the variable symbol &DBCS3R in the nominal value of a G-type constant results in removal of the SI/SO pair at the join.

Using SETC symbols

The character value assigned to a SETC symbol is substituted for the SETC symbol when it is used in the name, operation, or operand field of a statement.

For example, consider the following macro definition, macro instruction, and generated statements:

```
          MACRO
 &NAME    MOVE            &TO,&FROM
          LCLC            &PREFIX
 &PREFIX  SETC            'FIELD'             Statement 1
 &NAME    ST              2,SAVEAREA
          L               2,&PREFIX&FROM      Statement 2
          ST              2,&PREFIX&TO        Statement 3
          L               2,SAVEAREA
          MEND
 ---------------------------------------------------------------------
 HERE     MOVE            A,B
 ---------------------------------------------------------------------
+HERE ST             2,SAVEAREA
+     L              2,FIELDB
+     ST             2,FIELDA
+     L              2,SAVEAREA
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX. In statements 2 and 3, &PREFIX is replaced by FIELD.

The following example shows how the value assigned to a SETC symbol can be changed in a macro definition.

```
         MACRO
 &NAME   MOVE            &TO,&FROM
         LCLC            &PREFIX
 &PREFIX SETC            'FIELD'             Statement 1
 &NAME   ST              2,SAVEAREA
         L               2,&PREFIX&FROM      Statement 2
 &PREFIX SETC            'AREA'              Statement 3
         ST              2,&PREFIX&TO        Statement 4
         L               2,SAVEAREA
         MEND
 ---------------------------------------------------------------------
 HERE    MOVE            A,B
 ---------------------------------------------------------------------
+HERE ST             2,SAVEAREA
+     L              2,FIELDB
+     ST             2,AREAA
+     L              2,SAVEAREA
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX. Therefore, &PREFIX is replaced by FIELD in statement 2. Statement 3 assigns the character value AREA to &PREFIX. Therefore, &PREFIX is replaced by AREA, instead of FIELD, in statement 4.

The following example uses the substring notation in the operand field of a SETC instruction.

```
         MACRO
 &NAME   MOVE            &TO,&FROM
         LCLC            &PREFIX
 &PREFIX SETC            '&TO'(1,5)          Statement 1
 &NAME   ST              2,SAVEAREA
         L               2,&PREFIX&FROM      Statement 2
         ST              2,&TO
         L               2,SAVEAREA
         MEND
 ---------------------------------------------------------------------
 HERE    MOVE            FIELDA,B
 ---------------------------------------------------------------------
+HERE ST             2,SAVEAREA
+     L              2,FIELDB
+     ST             2,FIELDA
+     L              2,SAVEAREA
```

Statement 1 assigns the substring character value FIELD (the first five characters corresponding to symbolic parameter &TO to the SETC symbol &PREFIX. Therefore, FIELD replaces &PREFIX in statement 2.

Notes:

1. If the COMPAT(SYSLIST) assembler option is not specified, you can pass a sublist into a macro definition by assigning the sublist to a SETC symbol, and then specifying the SETC symbol as an operand in a macro instruction. However, if the COMPAT(SYSLIST) assembler option is specified, sublists assigned to SETC symbols are treated as a character string, not as a sublist.
2. Regardless of the setting of the COMPAT(SYSLIST) assembler option, you cannot pass separate (as opposed to a sublist of) parameters into a macro definition, by specifying a string of values separated by commas as the operand of a SETC instruction and then using the SETC symbol as an operand in the macro instruction. If you attempt to do this, the operand of the SETC instruction is passed to the macro instruction as one parameter, not as a list of parameters.

Concatenating substring notations and character expressions: Substring notations (see "Substring notation" on page 328) can be concatenated with character expressions in the operand field of a SETC

instruction. If a substring notation follows a character expression, the two can be concatenated by placing a period between the terminating apostrophe of the character expression and the opening apostrophe of the substring notation.

For example, if &ALPHA has been assigned the character value AB%4, and &BETA has been assigned the character value ABCDEF, the following statement assigns &GAMMA the character value AB%4BCD:

```
&GAMMA   SETC            '&ALPHA'.'&BETA'(2,3)
```

If a substring notation precedes a character expression or another substring notation, the two can be concatenated by writing the opening apostrophe of the second item immediately after the closing parenthesis of the substring notation.

Optionally, you can place a period between the closing parenthesis of a substring notation and the opening apostrophe of the next item in the operand field.

If &ALPHA has been assigned the character value AB%4, and &ABC has been assigned the character value 5RS, either of the following statements can be used to assign &WORD the character value AB%45RS.

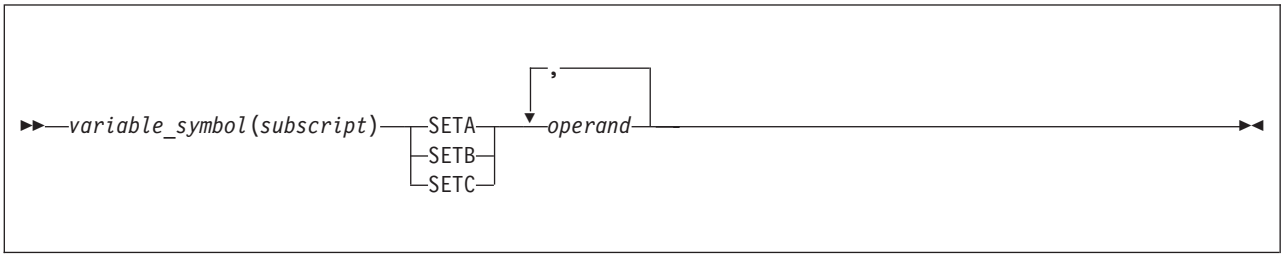
```
&WORD    SETC            '&ALPHA'(1,4).'&ABC'
&WORD    SETC            '&ALPHA'(1,4)'&ABC'(1,3)
```

If a SETC symbol is used in the operand field of a SETA instruction, the character value assigned to the SETC symbol must be 1-to-10 decimal digits (not greater than 2147483647), or a valid self-defining term.

If a SETA symbol is used in the operand field of a SETC statement, the magnitude of the arithmetic value is converted to an unsigned integer with leading zeros removed. If the value is 0, it is converted to a single 0.

Extended SET statements

As well as assigning single values to SET symbols, you can assign values to multiple elements in an array of a subscripted SET symbol with one single SETx instruction. Such an instruction is called an extended SET statement.



variable_symbol(*subscript*)
: Is a variable symbol and a subscript that shows the position in the SET symbol array to which the first *operand* is to be assigned.

operand
: Is the arithmetic value, binary value, or character value to be assigned to the corresponding SET symbol array element.

The first *operand* is assigned to the SET symbol denoted by *variable_symbol*(*subscript*). Successive *operand*s are then assigned to successive positions in the SET symbol array. If an *operand* is omitted, the corresponding element of the array is unchanged. Consider the following example:

```
         LCLA            &LIST(50)
&LIST(3) SETA            5,10,,20,25,30
```

The first instruction declares &LIST as a subscripted local SETA symbol. The second instruction assigns values to certain elements of the array &LIST. Thus, the instruction does the same as the following sequence:

```
&LIST(3) SETA           5
&LIST(4) SETA           10
&LIST(6) SETA           20
&LIST(7) SETA           25
&LIST(8) SETA           30
```

Alternative statement format

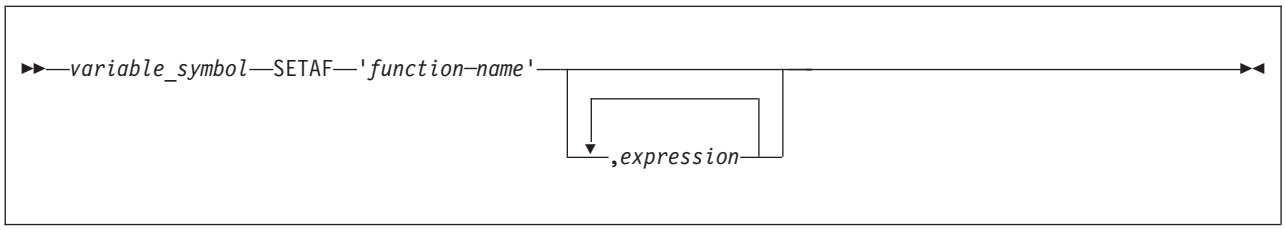
You can use the alternative statement format for extended SETx statements. This coding can be written as:

```
&LIST(3) SETA  5,               THIS IS                   X
               10,,             AN ARRAY                  X
               20,25,30         SPECIFICATION
```

SETAF instruction

Use the SETAF instruction to call an external function to assign any number of arithmetic values to a SETA symbol. You can assign many parameters—the exact number depending on factors such as the size of the program and of virtual storage—to pass to the external function routine.

The SETAF instruction can be used anywhere that a SETA instruction can be used.



variable symbol
Is a variable symbol.

A global variable symbol in the name field must have been previously declared as a SETA symbol in a GBLA instruction. Local SETA symbols need not be declared in an LCLA instruction. The assembler considers any undeclared variable symbol found in the name field of a SETA instruction as a local SET symbol.

The variable symbol is assigned a type attribute value of N.

function_name
The name of an external function load module. The name must be specified as a character expression, and must evaluate to a valid module name no longer than eight bytes.

See the chapter "Providing External Functions" in the *HLASM Programmer's Guide* for information about external function load modules.

expression
Is an arithmetic expression evaluated as a signed 32 bit arithmetic value. The minimum and maximum allowable values of the expression are -2^{31} and $+2^{31}-1$.

See "SETA instruction" on page 308 for further information about setting SETA symbols, and ways to specify arithmetic expressions.

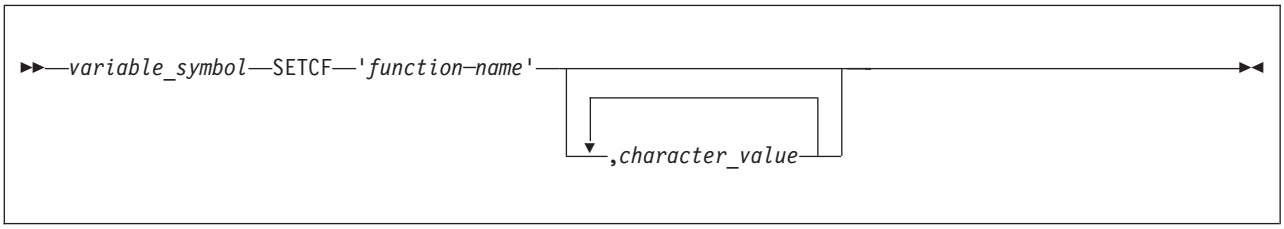
The function name must be enclosed in single quotes. For example:

```
&MAX_VAL   SETAF  'MAX',7,4        Calls the external function  X
                                   MAX, passing values 7 and    X
                                   4 as operands.
```

SETCF instruction

Use the SETCF instruction to call an external function to assign a character value to a SETC symbol. You can specify a many parameters—the exact number depending on factors such as the size of the program and of virtual storage—to pass to the external function routine.

The SETCF instruction can be used anywhere that a SETC instruction can be used.



variable symbol
Is a variable symbol.

A global variable symbol in the name field must have been previously declared as a SETC symbol in a GBLC instruction. Local SETC symbols need not be declared in an LCLC instruction. The assembler considers any undeclared variable symbol found in the name field of a SETC instruction as a local SET symbol. The variable symbol is assigned a type attribute value of U.

The character value assigned to the variable symbol can have a string length in the range 0 (for a null character string) through 1024.

function_name
The name of an external function load module. The name must be specified as a character expression, and must evaluate to a valid module name no longer than eight bytes.

See the chapter "Providing External Functions" in the *HLASM Programmer's Guide* for information about external function load modules.

character_value
Is a character value specified by:
- A type attribute reference
- An operation code attribute reference
- A character expression
- A substring notation
- A concatenation of one or more of the above

The character value can have a string length in the range 0 (for a null character string) through 1024.

When a SETA or SETB symbol is specified in a character expression, the unsigned decimal value of the symbol (with leading zeros removed) is the character value given to the symbol.

See "SETC instruction" on page 326 for further information about setting SETC symbols, and ways to specify character expressions.

Branching

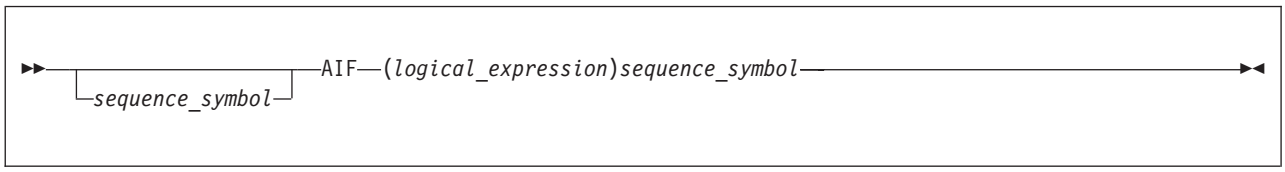
You can control the sequence in which source program statements are processed by the assembler by using the conditional assembly branch instructions described in this section.

AIF instruction

Use the AIF instruction to branch according to the results of a condition test. You can thus alter the sequence in which source program statements or macro definition statements are processed by the assembler.

The AIF instruction also provides loop control for conditional assembly processing, which lets you control the sequence of statements to be generated.

It also lets you check for error conditions and thus branch to the appropriate MNOTE instruction to issue an error message.



sequence_symbol
Is a sequence symbol

logical_expression
Is a logical expression (see "Logical (SETB) expressions" on page 323) the assembler evaluates during conditional assembly time to determine if it is true or false. If the expression is true (logical value=1), the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is false (logical value=0), the next sequential statement is processed by the assembler.

In the following example, the assembler branches to the label `.OUT` if `&C = YES`:

```
         AIF             ('&C' EQ 'YES').OUT
.ERROR   ANOP
         .
         .
         .
.OUT     ANOP
```

The sequence symbol in the operand field is a conditional assembly label that represents a statement number during conditional assembly processing. It is the number of the statement that is branched to if the logical expression preceding the sequence symbol is true.

The statement identified by the sequence symbol referred to in the AIF instruction can appear before or after the AIF instruction. However, the statement must appear within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear:

- In open code, if the corresponding AIF instruction appears in open code
- In the same macro definition in which the corresponding AIF instruction appears.

You cannot branch from open code into a macro definition or between macro definitions, regardless of nested calls to other macro definitions.

The following macro definition generates the statements needed to move a fullword fixed-point number from one storage area to another. The statements are generated only if the type attribute of both storage areas is the letter `F`.

```
         MACRO
&N       MOVE            &T,&F
         AIF             (T'&T NE T'&F).END  Statement 1
         AIF             (T'&T NE 'F').END   Statement 2
&N       ST              2,SAVEAREA          Statement 3
         L               2,&F
         ST              2,&T
         L               2,SAVEAREA
.END     MEND                                Statement 4
```

The logical expression in the operand field of Statement 1 has the value true if the type attributes of the two macro instruction operands are not equal. If the type attributes are equal, the expression has the logical value false.

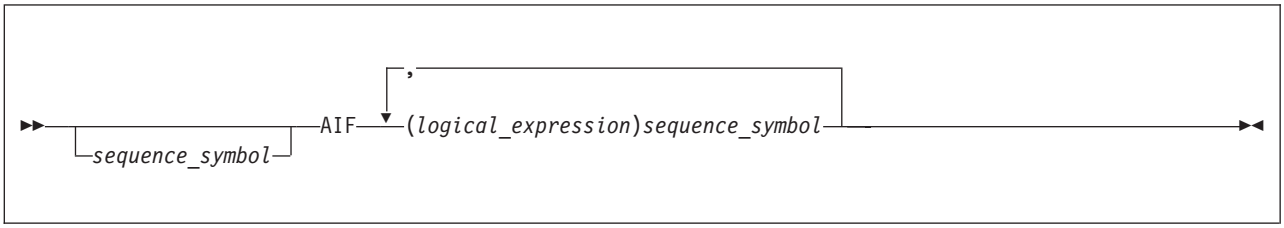
Therefore, if the type attributes are not equal, Statement 4 (the statement named by the sequence symbol `.END`) is the next statement processed by the assembler. If the type attributes are equal, Statement 2 (the next sequential statement) is processed.

The logical expression in the operand field of Statement 2 has the value true if the type attribute of the first macro instruction operand is not the letter `F`. If the type attribute is the letter `F`, the expression has the logical value false.

Therefore, if the type attribute is not the letter `F`, Statement 4 (the statement named by the sequence symbol `.END`) is the next statement processed by the assembler. If the type attribute is the letter `F`, Statement 3 (the next sequential statement) is processed.

Extended AIF instruction

The extended AIF instruction combines several successive AIF statements into one statement.



sequence_symbol
Is a sequence symbol

logical_expression
Is a logical expression the assembler evaluates during conditional assembly time to determine if it is true or false. If the expression is true (logical value=1), the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If the expression is false (logical value=0), the next logical expression is evaluated.

The extended AIF instruction is exactly equivalent to `n` successive AIF statements. The branch is taken to the first sequence symbol (scanning left to right) whose corresponding logical expression is true. If none of the logical expressions is true, no branch is taken.

Example:

```
                                                    Cont.
        AIF             ('&L'(&C,1) EQ '$').DOLR,     X
                        ('&L'(&C,1) EQ '#').POUND,    X
                        ('&L'(&C,1) EQ '@').AT,       X
                        ('&L'(&C,1) EQ '=').EQUAL,    X
                        ('&L'(&C,1) EQ '(').LEFTPAR,  X
                        ('&L'(&C,1) EQ '+').PLUS,     X
                        ('&L'(&C,1) EQ '-').MINUS
```

This statement looks for the occurrence of a `$`, `#`, `@`, `=`, `(`, `+`, and `-`, in that order; and causes control to branch to `.DOLR`, `.POUND`, `.AT`, `.EQUAL`, `.LEFTPAR`, `.PLUS`, and `.MINUS`, if the string being examined contains any of these characters at the position designated by `&C`.

Alternative format for AIF instruction

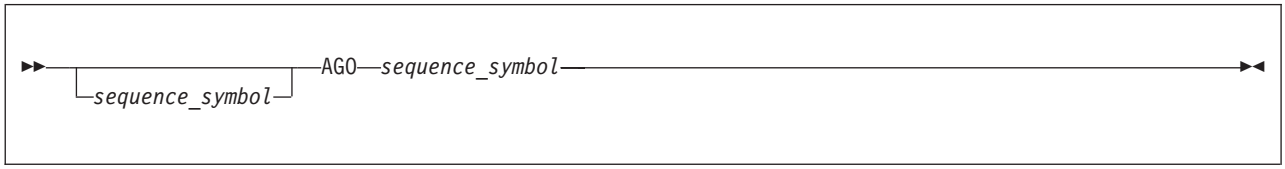
The alternative statement format is allowed for extended AIF instructions. This format is illustrated in the above example.

AIFB—synonym of the AIF instruction

For compatibility with some earlier assemblers, High Level Assembler supports the AIFB symbolic operation code as a synonym of the AIF instruction. However, do not use the AIFB instruction in new applications as support for it might be removed in the future.

AGO instruction

The AGO instruction branches unconditionally. You can thus alter the sequence in which your assembler language statements are processed. This provides you with final exits from conditional assembly loops.



sequence_symbol
Is a sequence symbol.

The statement named by the sequence symbol in the operand field is the next statement processed by the assembler.

The statement identified by a sequence symbol referred to in the AGO instruction can appear before or after the AGO instruction. However, the statement must appear within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear:

- In open code, if the corresponding AGO instruction appears in open code
- In the same macro definition in which the corresponding AGO instruction appears.

Example:

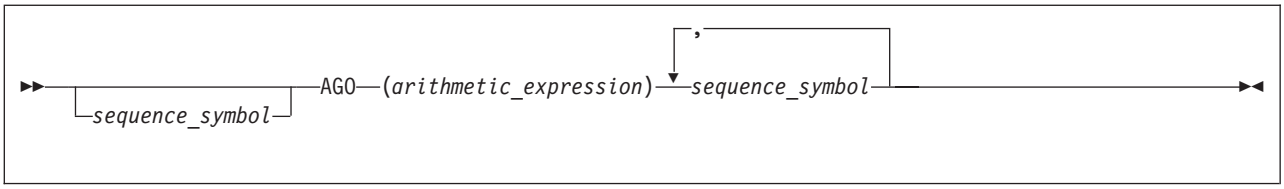
```
         MACRO
&NAME    MOVE            &T,&F
         AIF             (T'&T EQ 'F').FIRST     Statement 1
         AGO             .END                    Statement 2
.FIRST   AIF             (T'&T NE T'&F).END      Statement 3
&NAME    ST              2,SAVEAREA
         L               2,&F
         ST              2,&T
         L               2,SAVEAREA
.END     MEND                                    Statement 4
```

Statement 1 determines if the type attribute of the first macro instruction operand is the letter `F`. If the type attribute is the letter `F`, Statement 3 is the next statement processed by the assembler. If the type attribute is not the letter `F`, Statement 2 is the next statement processed by the assembler.

Statement 2 indicates to the assembler that the next statement to be processed is Statement 4 (the statement named by sequence symbol `.END`).

Computed AGO instruction

The computed AGO instruction makes branches according to the value of an arithmetic expression specified in the operand.



sequence_symbol
Is a sequence symbol.

arithmetic_expression
Is an arithmetic expression the assembler evaluates to k, where *k* is 1 - *n* (the number of occurrences of *sequence_symbol* in the operand field). The assembler branches to the *k*-th sequence symbol in the list. If *k* is outside that range, no branch is taken.

In the following example, control passes to the statement at `.THIRD` if `&I= 3`. Control passes through to the statement following the AGO if `&I` is less than 1 or greater than 4.

```
                                                    Cont.
        AGO             (&I).FIRST,.SECOND,           X
                        .THIRD,.FOURTH
```

Alternative format for AGO instruction

The alternative statement format is allowed for computed AGO instructions. The example can be coded:

```
                                                    Cont.
        AGO             (&I).FIRST,                   X
                        .SECOND,                      X
                        .THIRD,                       X
                        .FOURTH
```

AGOB - synonym of the AGO instruction

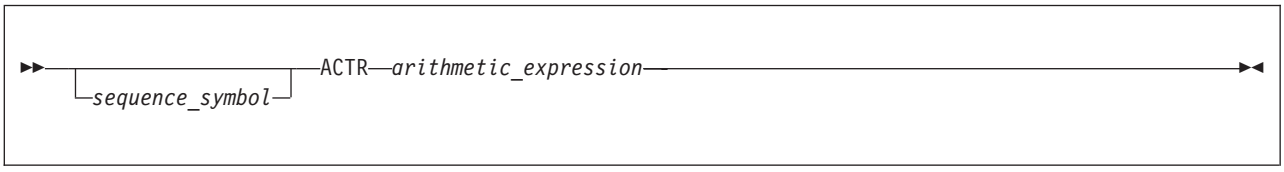
For compatibility with some earlier assemblers, High Level Assembler supports the AGOB symbolic operation code as a synonym of the AGO instruction. However, do not use the AGOB instruction in new applications as support for it might be removed in the future.

ACTR instruction

The ACTR instruction sets a conditional assembly branch counter either within a macro definition or in open code. The ACTR instruction can appear anywhere in open code or within a macro definition.

Each time the assembler processes a successful AIF or AGO branching instruction in a macro definition or in open code, the branch counter for that part of the program is decremented by one. When the number of conditional assembly branches reaches the value assigned to the branch counter by the ACTR instruction, the assembler exits from the macro definition or stops processing statements in open code.

By using the ACTR instruction, you avoid excessive looping during conditional assembly processing.



sequence_symbol
Is a sequence symbol.

arithmetic_expression
Is an arithmetic expression used to set or reset a conditional assembly branch counter.

A conditional assembly branch counter has a local scope; its value is decremented by AGO and successful AIF instructions, and reassigned only by ACTR instructions that appear within the same scope. Thus, the nesting of macros has no effect on the setting of branch counters in other scopes. The assembler assigns a branch counter for open code and for each macro definition. In the absence of an ACTR instruction, a default value of 4096 is assigned.

Branch counter operations

Within the scope of a branch counter, the following occurs:

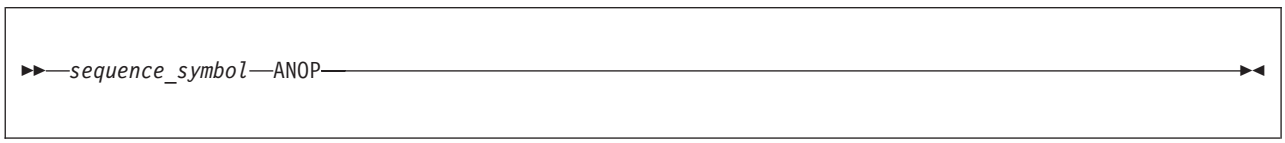
1. Each time an AGO or AIF branch is executed, the assembler checks the branch counter for zero or a negative value.
2. If the count is not zero or negative, it is decremented by one.
3. If the count is zero or negative, the assembler takes one of two actions:
   a. If it is processing instructions in open code, the assembler processes the remainder of the instructions in the source module as comments. Errors discovered in these instructions during previous passes are flagged.
   b. If it is processing instructions inside a macro definition, the assembler terminates the expansion of that macro definition and processes the next sequential instruction after the calling macro instruction. If the macro definition is called by an outer macro instruction, the assembler processes the next sequential prototype instruction after the call; that is, it continues processing at the next outer level of nested macros.

The assembler halves the ACTR counter value when it encounters serious syntax errors in conditional assembly instructions.

ANOP instruction

You can specify a sequence symbol in the name field of an ANOP instruction, and use the symbol as a label for branching purposes.

The ANOP instruction carries out no operation itself, but you can use it to allow conditional assembly to resume assembly or conditional generation at an instruction that does not have a sequence symbol in its name field. For example, if you wanted to branch to a SETA, SETB, or SETC assignment instruction, which requires a variable symbol in the name field, you can insert a labeled ANOP instruction immediately before the assignment instruction. By branching to the ANOP instruction with an AIF or AGO instruction, you are, in effect, branching to the assignment instruction.



sequence_symbol
Is a sequence symbol.

No operation is carried out by an ANOP instruction. Instead, if a branch is taken to the ANOP instruction, the assembler processes the next sequential instruction.

Example:

```
         MACRO
&NAME    MOVE           &T,&F
         LCLC           &TYPE
         AIF            (T'&T EQ 'F').FTYPE      Statement 1
&TYPE    SETC           'E'                      Statement 2
.FTYPE   ANOP                                    Statement 3
&NAME    ST&TYPE        2,SAVEAREA               Statement 4
```

```
L&TYPE          2,&F
ST&TYPE         2,&T
L&TYPE          2,SAVEAREA
MEND
```

Statement 1 determines if the type attribute of the first macro instruction operand is the letter F. If the type attribute is not the letter F, Statement 2 is the next statement processed by the assembler. If the type attribute is the letter F, Statement 4 should be processed next. However, because there is a variable symbol (&NAME) in the name field of Statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP instruction (Statement 3) must be placed before Statement 4.

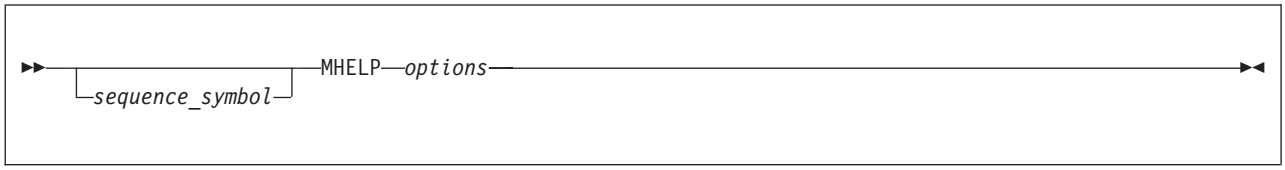
Then, if the type attribute of the first operand is the letter F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE retains its initial null character value because the SETC instruction is not processed. Because .FTYPE names an ANOP instruction, the next statement processed by the assembler is Statement 4, the statement following the ANOP instruction.

Chapter 10. MHELP instruction

The MHELP instruction controls a set of trace and dump facilities. MHELP statements can occur anywhere in open code or in macro definitions. MHELP options remain in effect until superseded by another MHELP statement.

MHELP options

Options are selected by an absolute expression in the MHELP operand field.



sequence_symbol
Is a sequence symbol.

options
Is the sum of the binary or decimal options. Here is a description of these options:

MHELP B'1' or MHELP 1, Macro Call Trace:
This option provides a one-line trace listing for each macro call, giving the name of the called macro, its nested depth, and its &SYSNDX value. The trace is provided only upon entry into the macro. No trace is provided if error conditions prevent entry into the macro.

MHELP B'10' or MHELP 2, Macro Branch Trace:
This option provides a one-line trace-listing for each AGO and AIF conditional assembly branch within a macro. It gives the model statement numbers of the "branched from" and the "branched to" statements, and the name of the macro in which the branch occurs. This trace option is suppressed for library macros.

MHELP B'100' or MHELP 4, Macro AIF Dump:
This option dumps undimensioned SET symbol values from the macro dictionary immediately before each AIF statement that is encountered.

MHELP B'1000' or MHELP 8, Macro Exit Dump:
This option dumps undimensioned SET symbols from the macro dictionary whenever an MEND or MEXIT statement is encountered.

MHELP B'10000' or MHELP 16, Macro Entry Dump:
This option dumps parameter values from the macro dictionary immediately after a macro call is processed.

MHELP B'100000' or MHELP 32, Global Suppression:
This option suppresses global SET symbols in two preceding options, MHELP 4 and MHELP 8.

MHELP B'1000000' or MHELP 64, Macro Hex Dump:
This option, when used with the Macro AIF dump, the Macro Exit dump, or the Macro Entry dump, dumps the parameter and SETC symbol values in EBCDIC and hexadecimal formats. Only positional and keyword parameters are dumped in hexadecimal; system parameters are dumped in EBCDIC. The full value of SETC variables or parameters is dumped in hexadecimal.

MHELP B'10000000' or MHELP 128, MHELP Suppression:
This option suppresses all currently active MHELP options.

MHELP Control on &SYSNDX:

The maximum value of the &SYSNDX system variable can be controlled by the MHELP instruction. The limit is set by specifying a number in the operand of the MHELP instruction that is not one of the MHELP codes defined above, and is in the following number ranges:

- 256 to 65535
- Most numbers in the range 65792 to 9999999. Refer to "MHELP operand mapping" for details.

When the &SYSNDX limit is reached, message `ASMA013S ACTR counter exceeded` is issued, and the assembler in effect ignores all further macro calls.

MHELP operand mapping

The MHELP operand field is mapped into a fullword. The predefined MHELP codes correspond to the fourth byte of this fullword, while the &SYSNDX limit is controlled by setting any bit in the third byte to 1. If all bits in the third byte are 0, then the &SYSNDX limit is not set.

The bit settings for bytes 3 and 4 are shown in Table 62.

Table 62. &SYSNDX Control Bits

Byte	Description
Byte 3 - &SYSNDX control	**1... ....** Bit 0 = 1. Value=32768. Limit &SYSNDX to 32768. **.1.. ....** Bit 1 = 1. Value=16384. Limit &SYSNDX to 16384. **..1. ....** Bit 2 = 1. Value=8192. Limit &SYSNDX to 8192. **...1 ....** Bit 3 = 1. Value=4096. Limit &SYSNDX to 4096. **.... 1...** Bit 4 = 1. Value=2048. Limit &SYSNDX to 2048. **.... .1..** Bit 5 = 1. Value=1024. Limit &SYSNDX to 1024. **.... ..1.** Bit 6 = 1. Value=512. Limit &SYSNDX to 512. **.... ...1** Bit 7 = 1. Value=256. Limit &SYSNDX to 256.
Byte 4	**1... ....** Bit 0 = 1. Value=128. MHELP Suppression. **.1.. ....** Bit 1 = 1. Value=64. Macro Hex Dump. **..1. ....** Bit 2 = 1. Value=32. Global Suppression. **...1 ....** Bit 3 = 1. Value=16. Macro Entry Dump. **.... 1...** Bit 4 = 1. Value=8. Macro Exit Dump. **.... .1..** Bit 5 = 1. Value=4. Macro AIF Dump. **.... ..1.** Bit 6 = 1. Value=2. Macro Branch Trace. **.... ...1** Bit 7 = 1. Value=1. Macro Call Trace.

Note: You can use any combination of bit settings in any byte of the MHELP fullword to set the limit, provided at least one bit in byte 3 is set. This explains why not all values 65792 - 9999999 can be used to set the limit. For example, the number 131123 does not set the &SYSNDX limit because none of the bits in byte 3 are set to 1.

Examples:

```
MHELP 256      Limit &SYSNDX to 256
MHELP 1        Trace macro calls
MHELP 65536    No effect. No bits in bytes 3,4
MHELP 65792    Limit &SYSNDX to 65792
```

See Figure 51 for more examples.

Combining options

More than one MHELP option, including the limit for &SYSNDX, can be specified at the same time by combining the option codes in one MHELP operand. For example, call and branch traces can be invoked by:

```
MHELP B'11'
MHELP 2+1
MHELP 3
```

Substitution by variable symbols can also be used.

MHELP Instruction	MHELP Operand				MHELP Effect
	Decimal	Hexadecimal			
			&SYSNDX	MHELP	
MHELP 4869	4869	0000	13	05	Macro call trace and AIF dump; &SYSNDX limited to 4869
MHELP 65536	65536	0001	00	00	No effect
MHELP 16777232	16777232	0010	00	10	Macro entry dump
MHELP 28678	28678	0000	70	06	Macro branch trace and AIF dump; &SYSNDX limited to 28678
MHELP 256+1	257	0000	01	01	Macro call trace; &SYSNDX limited to 257
MHELP B'11'	3	0000	00	03	Macro call trace, and macro branch trace

Figure 51. MHELP control on &SYSNDX

Appendix A. Assembler instructions

Table 63 summarizes the basic formats of assembler instructions, and Table 64 on page 357 summarizes assembler statements.

Table 63. Assembler instructions

Operation Entry	Name Entry	Operand Entry
ACONTROL[5]	A sequence symbol or space	One or more operands, separated by commas
ACTR	A sequence symbol or space	An arithmetic SETA expression
ADATA[5]	A sequence symbol or space	One-to-four decimal, self-defining terms, and one character string, separated by commas.
AEJECT[2]	A sequence symbol or space	Taken as a remark
AGO	A sequence symbol or space	A sequence symbol
AIF	A sequence symbol or space	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
AINSERT[5]	A sequence symbol or space	A character string, followed by FRONT or BACK
AMODE	Any symbol or space	24, 31, 64, ANY, or ANY31
ALIAS[5]	A symbol	A character string or a hexadecimal string
ANOP	A sequence symbol or space	Taken as a remark
AREAD[2]	Any SETC symbol	NOPRINT, NOSTMT, CLOCKB, CLOCKD, or spaces
ASPACE	A sequence symbol or space	An absolute expression
CATTR (z/OS and CMS)	A valid program object external class name	One or more attributes
CCW[4]	Any symbol or space	Four operands, separated by commas
CCW0[4]	Any symbol or space	Four operands, separated by commas
CCW1[4]	Any symbol or space	Four operands, separated by commas
CEJECT[5]	A sequence symbol or space	An absolute expression or space
CNOP[4]	Any symbol or space	Two absolute expressions, separated by a comma
COM	Any symbol or space	Taken as a remark
COPY[5]	A sequence symbol or space	An ordinary symbol, or, for open code statements, a variable symbol
CSECT	Any symbol or space	Taken as a remark
CXD[4]	Any symbol or space	Taken as a remark
DC[4]	Any symbol or space	One or more operands, separated by commas
DROP	A sequence symbol or space	One or more absolute expressions and symbols, separated by commas, or space
DS[4]	Any symbol or space	One or more operands, separated by commas
DSECT	A symbol or space	Taken as a remark
DXD[5]	A symbol	One or more operands, separated by commas
EJECT[5]	A sequence symbol or space	Taken as a remark
END	A sequence symbol or space	A relocatable expression or space

Table 63. Assembler instructions (continued)

Operation Entry	Name Entry	Operand Entry
ENTRY[5]	A sequence symbol or space	One or more relocatable symbols, separated by commas
EQU[4]	A variable symbol or an ordinary symbol	One to five operands, separated by commas
EXITCTL[5]	A sequence symbol or space	A character-string operand followed by one to four decimal self-defining terms, separated by commas
EXTRN[5]	A sequence symbol or space	One or more relocatable symbols, separated by commas
GBLA	A sequence symbol or space	One or more variable symbols that are to be used as SET symbols, separated by commas[1]
GBLB	A sequence symbol or space	One or more variable symbols that are to be used as SET symbols, separated by commas[1]
GBLC	A sequence symbol or space	One or more variable symbols that are to be used as SET symbols, separated by commas[1]
ICTL	Space	One to three decimal self-defining terms, separated by commas
ISEQ[5]	A sequence symbol or space	Two decimal self-defining terms, separated by a comma, or space
LCLA	A sequence symbol or space	One or more variable symbols that are to be used as SET symbols, separated by commas[1]
LCLB	A sequence symbol or space	One or more variable symbols that are to be used as SET symbols, separated by commas[1]
LCLC	A sequence symbol or space	One or more variable symbols separated by commas[1]
LOCTR	A variable or ordinary symbol	Space
LTORG	Any symbol or space	Taken as a remark
MACRO[2,5]	Space	Taken as a remark
MEND[2,5]	A sequence symbol or space	Taken as a remark
MEXIT[2,5]	A sequence symbol or space	Taken as a remark
MHELP	A sequence symbol or space	Absolute expression, binary, or decimal options
MNOTE	A sequence symbol or space	A severity code, followed by a comma, followed by a character string enclosed in apostrophes. Double-byte characters are permitted if the DBCS assembler option is specified.
OPSYN		
	An ordinary symbol	A machine instruction mnemonic or an operation code defined by a previous macro definition or OPSYN instruction
	An operation code mnemonic	Space
ORG	A sequence symbol or space	A relocatable expression or space
POP[5]	A sequence symbol or space	One or more operands, separated by commas
PRINT[5]	A sequence symbol or space	One or more operands, separated by commas
PUNCH[5]	A sequence symbol or space	A 1-to-80-character string enclosed in apostrophes. Double-byte characters are permitted if the DBCS assembler option is specified.
PUSH[5]	A sequence symbol or space	One or more operands, separated by commas

Table 63. Assembler instructions (continued)

Operation Entry	Name Entry	Operand Entry
REPRO[5]	A sequence symbol or space	Taken as a remark
RMODE	Any symbol or space	24, 31, 64, or ANY
RSECT	Any symbol or space	Taken as a remark
SETA	A SETA symbol	An arithmetic expression
SETAF	A SETA symbol	An external function module, and the arithmetic expressions it requires, separated by commas
SETB	A SETB symbol	A 0 or a 1, or a logical expression enclosed in parentheses
SETC	A SETC symbol	A type attribute, a character expression, a substring notation, or a concatenation of character expressions and substring notations. Double-byte characters are permitted if the DBCS assembler option is specified.
SETCF	A SETC symbol	An external function module, and the character expressions it requires, separated by commas
SPACE[5]	A sequence symbol or space	An absolute expression
START	Any symbol or space	An absolute expression or space
TITLE[3,5]	A 1-to-8-character string, a variable symbol, a combination of character string or variable symbol, a sequence symbol, or space	A 1-to-100-character string enclosed in apostrophes. Double-byte characters are permitted if the DBCS assembler option is specified.
USING	A symbol or space	Either a single absolute or relocatable expression or a pair of absolute or relocatable expressions enclosed in parentheses and followed by 1 to 16 absolute expressions, separated by commas, or followed by a relocatable expression
WXTRN[5]	A sequence symbol or space	One or more relocatable symbols, separated by commas
XATTR[5] (z/OS and CMS)	An external symbol	One or more operands, separated by commas

Notes:
1. SET symbols can be defined as subscripted SET symbols.
2. Can only be used as part of a macro definition.
3. See "TITLE instruction" on page 190 for a description of the name entry.
4. These instructions start a private section.
5. These instructions can be specified before the first executable control section.

Table 64. Assembler statements

Instruction Entry	Name Entry	Operand Entry
ModelStatements[1 and 2]	An ordinary symbol, variable symbol, sequence symbol, or a combination of variable symbols and other characters that is equivalent to a symbol, or space	Any combination of characters (including variable symbols)
Prototype Statement[3]	A symbolic parameter or space	Zero or more operands that are symbolic parameters (separated by commas), and zero or more operands (separated by commas) of the form symbolic parameter, equal sign, optional standard value

Table 64. Assembler statements (continued)

Instruction Entry	Name Entry	Operand Entry
Macro Instruction Statement[3]	An ordinary symbol, a variable symbol, or a combination of variable symbols and other characters that is equivalent to a symbol, any character string, a sequence symbol[4] or space	Zero or more positional operands (separated by commas), and zero or more keyword operands (separated by commas) of the form keyword, equal sign, value[5]
Assembler Language Statement[12]	An ordinary symbol, a variable symbol, a sequence symbol, or a combination of variable symbols and other characters that is equivalent to a symbol, or space	Any combination of characters (including variable symbols)

Notes:

1. Variable symbols can be used to generate assembler language mnemonic operation codes (listed in Chapter 5, "Assembler instruction statements," on page 83), except COPY, ICTL, ISEQ, and REPRO. Variable symbols cannot be used in the name and operand entries of COPY, ICTL, and ISEQ instructions, except for the COPY instruction in open code, where a variable symbol is allowed for the operand entry.
2. No substitution is done for variables in the line following a REPRO statement.
3. Can only be used as part of a macro definition.
4. When the name field of a macro instruction contains a sequence symbol, the sequence symbol is not passed as a name field parameter. It only has meaning as a possible branch target for conditional assembly.
5. Variable symbols appearing in a macro instruction are replaced by their values before the macro instruction is processed.

Appendix B. Summary of constants

Table 65 and Table 66 on page 360 summarize the types of assembler constants.

Table 65. Summary of constants (part 1 of 2)

Constant	Type	Implicit Length (Bytes)	Alignment	Length Modifier Range	Specified By
Address	A	4	Fullword	.1 to 4[1]	Any expression
Doubleword Address	AD	8	Doubleword	.1 to 8[1]	Any expression
Binary	B	As needed	Byte	.1 to 256	Binary digits
Character	C	As needed	Byte	.1 to 256[2]	Characters
ASCII Character	CA	As needed	Byte	.1 to 256[2]	Characters
Unicode Character	CU	As needed	Byte	2 to 256[3]	Characters
Floating Point Hex	D	8	Doubleword	.1 to 8	Decimal digits
Floating Point Hex	DH	8	Doubleword	.12 to 8	Decimal digits
Floating Point Binary	DB	8	Doubleword	.12 to 8	Decimal digits
Floating Point Decimal	DD	8	Doubleword	8	Decimal digits
Floating Point Hex	E	4	Fullword	.1 to 8	Decimal digits
Floating Point Hex	EH	4	Fullword	.12 to 8	Decimal digits
Floating Point Binary	EB	4	Fullword	.9 to 8	Decimal digits
Floating Point Decimal	ED	4	Fullword	4	Decimal digits
Fixed Point	F	4	Fullword	.1 to 8	Decimal digits
Doubleword Fixed Point	FD	8	Doubleword	.1 to 8	Decimal digits
Graphic (DBCS)	G	As needed	Byte	2 to 256[3]	DBCS characters
Fixed Point	H	2	Halfword	.1 to 8	Decimal digits
Length	J	4	Fullword	1 to 4	Class name or external DSECT name[4]
Floating Point Hex	L	16	Doubleword	.1 to 16	Decimal digits
Floating Point Hex	LH	16	Doubleword	.12 to 16	Decimal digits
Floating Point Binary	LB	16	Doubleword	.16 to 16	Decimal digits
Floating Point Decimal	LD	16	Doubleword	16	Decimal digits
Floating Point Hex	LQ	16	Quadword	.1 to 16	Decimal digits
Decimal	P	As needed	Byte	.1 to 16	Decimal digits
Offset	Q QY[4]	4 3	Fullword Halfword	1 to 4 3 only	Symbol naming a DXD, DSECT, or part
Address	R[4]	4	Fullword	3, 4	Symbol
Address	S SY	2 3	Halfword Halfword	2 only 3 only	One absolute or relocatable expression, or two absolute expressions: `exp(exp)`
Address	V	4	Fullword	3, 4	Relocatable symbol
Hexadecimal	X	As needed	Byte	.1 to 256[2]	Hex digits

Table 65. Summary of constants (part 1 of 2) (continued)

Constant	Type	Implicit Length (Bytes)	Alignment	Length Modifier Range	Specified By
Address	Y	2	Halfword	.1 to 2[1]	Any expression
Decimal	Z	As needed	Byte	.1 to 16	Decimal digits

Notes:

1. Bit length specification permitted with absolute expressions only; relocatable A-type constants, 2, 3, or 4 bytes only; relocatable Y-type constants, 2 bytes only.
2. In a DS assembler instruction, C-and-X type constants can have length specification to 65535.
3. The length modifier must be a multiple of 2, and can be up to 65534 in a DS assembler instruction.
4. GOFF only.

Table 66. Summary of constants (part 2 of 2)

Constant	Type	No. of Constants per Operand	Range for Exponents	Range for Scale	Truncation or Padding Side
Address	A	Multiple			Left
Binary	B	Multiple			Left
Character	C	One			Right
ASCII Character	CA	One			Right
Unicode Character	CU	One			Right
Floating Point Hex	D	Multiple	-85 to +75	0 to 13	Right[1]
Floating Point Hex	DH	Multiple	-2^{31} to $2^{31}-1$	0 to 13	Right[1]
Floating Point Binary	DB	Multiple	-2^{31} to $2^{31}-1$	N/A	Right[1]
Floating Point Decimal	DD	Multiple	-2^{31} to $2^{31}-1$	N/A	
Floating Point Hex	E	Multiple	-85 to +75	0 to 5	Right[1]
Floating Point Hex	EH	Multiple	-2^{31} to $2^{31}-1$	0 to 5	Right[1]
Floating Point Binary	EB	Multiple	-2^{31} to $2^{31}-1$	N/A	Right[1]
Floating Point Decimal	ED	Multiple	-2^{31} to $2^{31}-1$	N/A	
Fixed Point	F	Multiple	-85 to +75	-187 to +346	Left[1]
Graphic (DBCS)	G	One			Right
Fixed Point	H	Multiple	-85 to +15	-187 to +346	Left[1]
Length	J	Multiple			Left[1]
Floating Point Hex	L	Multiple	-85 to +75	0 to 27	Right[1]
Floating Point Hex	LH	Multiple	-2^{31} to $2^{31}-1$	0 to 27	Right[1]
Floating Point Binary	LB	Multiple	-2^{31} to $2^{31}-1$	N/A	Right[1]
Floating Point Decimal	LD	Multiple	-2^{31} to $2^{31}-1$	N/A	
Floating Point Hex	LQ	Multiple	-2^{31} to $2^{31}-1$	0 to 28	Right[1]
Decimal	P	Multiple			Left
Offset	Q	Multiple			Left
Address	R	Multiple			Left
Address	S	Multiple			
Address	V	Multiple			Left

Table 66. Summary of constants (part 2 of 2) (continued)

Constant	Type	No. of Constants per Operand	Range for Exponents	Range for Scale	Truncation or Padding Side
Hexadecimal	X	Multiple			Left
Address	Y	Multiple			Left
Decimal	Z	Multiple			Left

Notes:

1. Errors are flagged if significant bits are truncated or if the value specified cannot be contained in the implicit length of the constant.

Appendix C. Macro and conditional assembly language summary

This appendix summarizes the macro and conditional assembly language described in Part 3 of this publication. Table 67 on page 364 indicates which macro and conditional assembly language elements can be used in the name and operand entries of each statement. Table 69 on page 368 summarizes the expressions that can be used in macro instruction statements. Table 71 on page 368 summarizes the attributes that can be used in each expression. Table 72 on page 369 summarizes the variable symbols that can be used in each expression. Table 73 on page 370 summarizes the system variable symbols that can be used in each expression.

Table 67. Macro language elements (part 1)

	Variable Symbols							
		Global-scope SET Symbols[2]			Local SET Symbols[2]			
Statement	**Symbolic Parameter**	**SETA**	**SETB**	**SETC**	**SETA**	**SETB**	**SETC**	**Sequence Symbol**
MACRO								
Prototype Statement	Name Operand							
GBLA	Operand[11]	Operand	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Name
GBLB	Operand[11]	Operand[11]	Operand	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Name
GBLC	Operand[11]	Operand[11]	Operand[11]	Operand	Operand[11]	Operand[11]	Operand[11]	Name
LCLA	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand	Operand[11]	Operand[11]	Name
LCLB	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand	Operand[11]	Name
LCLC	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand[11]	Operand	Name
Model Statement	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name
SETA	Name[12] Operand[3]	Name Operand	Name[12] Operand[4]	Name[12] Operand[10]	Name Operand	Name[12] Operand[4]	Name[12] Operand[10]	
SETAF	Name[12] Operand[3,13]	Name Operand[13]	Name[12] Operand[4,13]	Name[10,12] Operand[13]	Name Operand[13]	Name[12] Operand[4,13]	Name[10,12] Operand[13]	
SETB	Name[12] Operand[7]	Name[12] Operand[7]	Name Operand	Name[12] Operand[7]	Name[12] Operand[7]	Name Operand	Name[12] Operand[7]	
SETC	Name[12] Operand	Name[12] Operand[8]	Name[12] Operand[9]	Name Operand	Name[12] Operand[8]	Name[12] Operand[9]	Name Operand	Operand
SETCF	Name[12] Operand[13]	Name[12] Operand[8,13]	Name[12] Operand[9,13]	Name Operand[13]	Name[12] Operand[8,13]	Name[12] Operand[9,13]	Name Operand[13]	
ACTR	Operand[3]	Operand	Operand[4]	Operand[3]	Operand	Operand[4]	Operand[3]	Name
AEJECT								Name
AGO								Name Operand
AIF	Operand[7]	Operand[7]	Operand	Operand[7]	Operand[7]	Operand	Operand[7]	Name Operand
ANOP								Name
AREAD	Name[12]	Name[12]	Name[12]	Name	Name[12]	Name[12]	Name	

Table 67. Macro language elements (part 1) (continued)

	Variable Symbols							
		Global-scope SET Symbols[2]			Local SET Symbols[2]			
Statement	**Symbolic Parameter**	**SETA**	**SETB**	**SETC**	**SETA**	**SETB**	**SETC**	**Sequence Symbol**
ASPACE	Operand[3]	Operand	Operand[4]	Operand[3]	Operand	Operand[4]	Operand[3]	Name
MEXIT								Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Name
MEND								Name
Outer Macro		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name

Notes:

1. Variable symbols in macro instructions are replaced by their values before processing.
2. Depending upon their values, system variable symbols with global scope can be used in the same way as global SET symbols, and system variable symbols with local scope can be used in the same way as local SET symbols.
3. Only if value is self-defining term.
4. Converted to arithmetic 0 or 1.
5. Only in character relations.
6. Only in arithmetic relations.
7. Only in arithmetic or character relations.
8. Converted to an unsigned number.
9. Converted to character 0 or 1.
10. Only if one to ten decimal digits, not greater than 2147483647.
11. Only in created SET symbols if value of parenthesized expression is an alphabetic character followed by 0 to 61 alphanumeric characters.
12. Only in created SET symbols (as described above) and in subscripts (see SETA statement).
13. The first operand of a SETAF or SETCF instruction must be a character (SETC) expression containing or evaluating to an eight byte module name.

Table 68. Macro language elements (part 2)

	Attributes							Type
Statement	**Length**	**Scale**	**Integer**	**Count**	**Number**	**Defined**	**Operation Code**	
MACRO								
Prototype Statement								
GBLA								
GBLB								
GBLC								
LCLA								
LCLB								
LCLC								
Model Statement								
SETA		Operand	Operand	Operand	Operand	Operand	Operand	
SETAF		Operand[13]	Operand[13]	Operand[13]	Operand[13]	Operand[13]		
SETB	Operand[5]	Operand[6]	Operand[6]	Operand[6]	Operand[6]	Operand[6]	Operand[6]	Operand[5]
SETC	Operand							Operand
SETCF	Operand[13]							
ACTR		Operand	Operand	Operand	Operand	Operand		
AEJECT								
AGO								
AIF	Operand[5]	Operand[6]	Operand[6]	Operand[6]	Operand[6]	Operand[6]	Operand	Operand
ANOP								
AREAD								
ASPACE		Operand	Operand	Operand	Operand	Operand		
MEXIT								
MNOTE								
MEND								
Outer Macro								

Table 68. Macro language elements (part 2) (continued)

	Attributes							Type
Statement	**Length**	**Scale**	**Integer**	**Count**	**Number**	**Defined**	**Operation Code**	

Notes:

1. Variable symbols in macro instructions are replaced by their values before processing.
2. Depending upon their values, system variable symbols with global scope can be used in the same way as global SET symbols, and system variable symbols with local scope can be used in the same way as local SET symbols.
3. Only if value is self-defining term.
4. Converted to arithmetic 0 or 1.
5. Only in character relations.
6. Only in arithmetic relations.
7. Only in arithmetic or character relations.
8. Converted to an unsigned number.
9. Converted to character 0 or 1.
10. Only if one to ten decimal digits, not greater than 2147483647.
11. Only in created SET symbols if value of parenthesized expression is an alphabetic character followed by 0 to 61 alphanumeric characters.
12. Only in created SET symbols (as described above) and in subscripts (see SETA statement).
13. The first operand of a SETAF or SETCF instruction must be a character (SETC) expression containing or evaluating to an eight byte module name.

Table 69. Conditional assembly expressions

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions
Can contain	Self-defining terms Absolute, predefined ordinary symbols Length, scale, integer, count, defined, and number attributes SETA and SETB symbols SETC symbols whose values are a self-defining term Symbolic parameters if the corresponding operand is a self-defining term Built-in Functions &SYSDATC &SYSLIST(*n*) if the corresponding operand is a self-defining term &SYSLIST (*n*,*m*) if the corresponding operand is a self-defining term &SYSOPT_DBCS, &SYSOPT_RENT, and &SYSOPT_XOBJECT &SYSM_HSEV and &SYSM_SEV &SYSNDX, &SYSNEST, and &SYSSTMT	Any combination of characters (including double-byte characters, if the DBCS assembler option is specified) enclosed in apostrophes Any variable symbol enclosed in apostrophes A concatenation of variable symbols and other characters enclosed in apostrophes Built-in Functions A type or operation code attribute reference Substrings	A 0 or a 1 Absolute, predefined ordinary symbols SETB symbols Arithmetic relations Character relations Arithmetic value
Operations	+, - (unary and binary), *, and /; Parentheses permitted	Concatenation, with a period (.), or by juxtaposition; substrings	AND, OR, NOT, XOR Parentheses permitted
Range of values	-2^{31} to $+2^{31}-1$	0 through 1024 characters	0 (false) or 1 (true)
Used in	SETA operands Arithmetic relations Created SET symbols Subscripted SET symbols &SYSLIST subscripts Substring notation Sublist notation	SETC operands Character relations Created SET symbols	SETB operands AIF operands Created SET symbols

Built-in functions fall into the following categories:

Table 70. Built-in functions

Value Type	Functions
Arithmetic	AND, B2A, C2A, D2A, DCLEN, FIND, INDEX, NOT, OR, SLA, SLL, SRA, SRL, X2A, XOR
Logical	AND, AND NOT, ISBIN, ISDEC, ISHEX, ISSYM, NOT, OR, OR NOT, XOR, XOR NOT
Character	A2B, A2C, A2D, A2X, B2C, B2D, B2X, BYTE, C2B, C2D, C2X, D2B, D2C, D2X, DCVAL, DEQUOTE, DOUBLE, LOWER, SIGNED, UPPER, X2B, X2C, X2D

Table 71. Attributes

Attribute	Notation	Can be used with:	Can be used only if Type Attribute is:	Can be used in:
Type	T'	Ordinary symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST(*n*), &SYSLIST(*n*,*m*) inside macro definitions; SET symbols; all system variable symbols	Any value	SETC expressions Character relations

Table 71. Attributes (continued)

Attribute	Notation	Can be used with:	Can be used only if Type Attribute is:	Can be used in:
Length	L'	Ordinary symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST(*n*), and &SYSLIST(*n*,*m*) inside macro definitions	Any value except M, N, O, T, U	SETA and ordinary arithmetic expressions
Scale	S'	Ordinary symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST(*n*), and &SYSLIST(*n*,*m*) inside macro definitions	H,F,G,D,E,L,K,P, and Z	SETA and ordinary arithmetic expressions
Integer	I'	Ordinary symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST(*n*), and &SYSLIST(*n*,*m*) inside macro definitions	H,F,G,D,E,L,K,P, and Z	SETA and ordinary arithmetic expressions
Count	K'	Symbolic parameters inside macro definitions; &SYSLIST(*n*), and &SYSLIST(*n*,*m*) inside macro definitions; SET symbols; all system variable symbols	Any letter or @ or $	SETA and ordinary arithmetic expressions
Number	N'	Symbolic parameters, &SYSLIST and &SYSLIST(*n*) inside macro definitions, with dimensioned SET symbols	Any value	SETA and ordinary arithmetic expressions
Defined	D'	Ordinary symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST and &SYSLIST(*n*) inside macro definitions; SETC symbols whose value is an ordinary symbol	Any value except M, N, O, T, U	SETA arithmetic expressions
Operation Code	O'	A character string, or variable symbol containing a character string.	@, $, and any letter except N, O and (only sometimes) U	SETC expressions Character relations

Refer to Chapter 9, "How to write conditional assembly instructions," on page 279 for usage restrictions of the attributes in Table 71 on page 368.

Table 72. Variable symbols

Variable Symbol	Declared by:	Initialized or set to:	Value changed by:	Can be used in:
Symbolic[1] parameter	Prototype statement	Corresponding macro instruction operand	Constant throughout definition	Arithmetic expressions if operand is self-defining term Character expressions
SETA	LCLA or GBLA instruction	0	SETA instruction	Arithmetic expressions Character expressions Logical expressions

Table 72. Variable symbols (continued)

Variable Symbol	Declared by:	Initialized or set to:	Value changed by:	Can be used in:
SETB	LCLB or GBLB instruction	0	SETB instruction	Arithmetic expressions Character expressions Logical expressions
SETC	LCLC or GBLC instruction	String of length 0 (null)	SETC instruction	Arithmetic expressions if value is self-defining term Character expressions Logical expressions if value is self-defining term

Notes:

1. Can be used only in macro definitions.

Table 73. System variable symbols

System Variable Symbol	Avail-ability[1]	Type[2]	Type Attr.[3]	Scope	Initialized or set to	Value changed by	Can be used in
&SYSADATA_DSN	HLA2	C	U,O	L	Current associated data file	Constant throughout assembly	Character expressions
&SYSADATA_MEMBER	HLA2	C	U,O	L	Current associated data file member name	Constant throughout assembly	Character expressions
&SYSADATA_VOLUME	HLA2	C	U,O	L	Current associated data file volume identifier	Constant throughout assembly	Character expressions
&SYSASM	HLA1	C	U	G	Assembler name	Constant throughout assembly	Character expression
&SYSCLOCK	HLA3	C	U	L	Current date and time	Constant throughout macro expansion	Character expressions
&SYSDATC	HLA1	C,A	N	G	Assembly date (with century)	Constant throughout assembly	Arithmetic expressions Character expressions
&SYSDATE	AsmH	C	U	G	Assembly date	Constant throughout assembly	Character expressions
&SYSECT	All	C	U	L	Name of control section in effect where macro instruction appears	Constant throughout definition; set by START, CSECT, RSECT, DSECT, or COM	Character expressions
&SYSIN_DSN	HLA1	C	U	L	Current primary input data set name	Constant throughout definition	Character expressions

Table 73. System variable symbols (continued)

System Variable Symbol	Availability[1]	Type[2]	Type Attr.[3]	Scope	Initialized or set to	Value changed by	Can be used in
&SYSIN_MEMBER	HLA1	C	U,O	L	Current primary input member name	Constant throughout definition	Character expressions
&SYSIN_VOLUME	HLA1	C	U,O	L	Current primary input volume identifier	Constant throughout definition	Character expressions
&SYSJOB	HLA1	C	U	G	Source module assembly job name	Constant throughout assembly	Character expressions
&SYSLIB_DSN	HLA1	C	U	L	Current macro library filename	Constant throughout definition	Character expressions
&SYSLIB_MEMBER	HLA1	C	U,O	L	Current macro library member name	Constant throughout definition	Character expressions
&SYSLIB_VOLUME	HLA1	C	U,O	L	Current macro library volume identifier	Constant throughout definition	Character expressions
&SYSLIN_DSN	HLA2	C	U	L	Current object data set name	Constant throughout assembly	Character expressions
&SYSLIN_MEMBER	HLA2	C	U,O	L	Current object data set member name	Constant throughout assembly	Character expressions
&SYSLIN_VOLUME	HLA2	C	U,O	L	Current object data set volume identifier	Constant throughout assembly	Character expressions
&SYSLIST	All	C	any	L	Not applicable	Not applicable	N'&SYSLIST in arithmetic expressions
&SYSLIST(*n*)&SYSLIST(*n*,*m*)	All	C	any	L	Corresponding macro instruction operand	Constant throughout definition	Arithmetic expressions if operand is self-defining term Character expressions
&SYSLOC	AsmH	C	U	L	Location counter in effect where macro instruction appears	Constant throughout definition; set by START, CSECT, RSECT, DSECT, COM, and LOCTR	Character expressions
&SYSMAC	HLA3	C	U,O	L	Macro name	Constant throughout definition	Arithmetic expressions
&SYSMAC(n)$_1$	HLA3	C	U,O	L	Ancestor macro name	Constant throughout definition	Arithmetic expressions
&SYSM_HSEV	HLA3	A	N	G	0	Mnote	Arithmetic expressions

Table 73. System variable symbols (continued)

System Variable Symbol	Avail-ability[1]	Type[2]	Type Attr.[3]	Scope	Initialized or set to	Value changed by	Can be used in
&SYSM_SEV	HLA3	A	N	G	0	At nesting and unnesting of macros, from MNOTE	Arithmetic expressions
&SYSNDX	All	C	N	L	Macro instruction index	Constant throughout definition; unique for each macro instruction	Arithmetic expressions Character expressions
&SYSNEST	HLA1	A	N	L	Macro instruction nesting level	Constant throughout definition; unique for each macro nesting level	Arithmetic expressions Character expressions
&SYSOPT_DBCS	HLA1	B	N	G	DBCS assembler option indicator	Constant throughout assembly	Arithmetic expressions Character expressions Logical expressions
&SYSOPT_OPTABLE	HLA3	C	U	G	OPTABLE assembler option value	Constant throughout assembly	Character expressions
&SYSOPT_RENT	HLA1	B	N	G	RENT assembler option indicator	Constant throughout assembly	Arithmetic expressions Character expressions Logical expressions
&SYSOPT_XOBJECT	HLA3	B	N	G	XOBJECT assembler option indicator	Constant throughout assembly	Arithmetic expressions Character expressions Logical expressions
&SYSPARM	All	C	U,O	G	User defined or null	Constant throughout assembly	Arithmetic expressions if value is self-defining term Character expressions
&SYSPRINT_DSN	HLA2	C	U	L	Current assembler listing data set name	Constant throughout assembly	Character expressions
&SYSPRINT_MEMBER	HLA2	C	U,O	L	Current assembler listing data set member name	Constant throughout assembly	Character expressions
&SYSPRINT_VOLUME	HLA2	C	U,O	L	Current assembler listing data set volume identifier	Constant throughout assembly	Character expressions

Table 73. System variable symbols (continued)

System Variable Symbol	Avail-ability[1]	Type[2]	Type Attr.[3]	Scope	Initialized or set to	Value changed by	Can be used in
&SYSPUNCH_DSN	HLA2	C	U	L	Current object data set name	Constant throughout assembly	Character expressions
&SYSPUNCH_MEMBER	HLA2	C	U,O	L	Current object data set member name	Constant throughout assembly	Character expressions
&SYSPUNCH_VOLUME	HLA2	C	U,O	L	Current object data set volume identifier	Constant throughout assembly	Character expressions
&SYSSEQF	HLA1	C	U,O	L	Outer-most macro instruction identification-sequence field	Constant throughout definition	Character expressions
&SYSSTEP	HLA1	C	U	G	Source module assembly job name	Constant throughout assembly	Character expressions
&SYSSTMT	HLA1	C,A	N	G	Next statement number	Assembler increments each time a statement is processed	Arithmetic expressions Character expressions
&SYSSTYP	HLA1	C	U,O	L	Type of control section in effect where macro instruction appears	Constant throughout definition; set by START, CSECT, RSECT, DSECT, or COM	Character expressions
&SYSTEM_ID	HLA1	C	U	G	Assembly operating system environment identifier	Constant throughout assembly	Character expressions
&SYSTERM_DSN	HLA2	C	U	L	Current terminal data set name	Constant throughout assembly	Character expressions
&SYSTERM_MEMBER	HLA2	C	U,O	L	Current terminal data set member name	Constant throughout assembly	Character expressions
&SYSTERM_VOLUME	HLA2	C	U,O	L	Current terminal data set volume identifier	Constant throughout assembly	Character expressions
&SYSTIME	AsmH	C	U	G	Source module assembly time	Constant throughout assembly	Character expressions
&SYSVER	HLA1	C	U	G	Assembler release level	Constant throughout assembly	Character expressions

Table 73. System variable symbols (continued)

System Variable Symbol	Avail-ability[1]	Type[2]	Type Attr.[3]	Scope	Initialized or set to	Value changed by	Can be used in

Notes:

1. **Availability:**
   - **All** All assemblers, including the DOS/VSE Assembler
   - **AsmH** Assembler H Version 2 and High Level Assembler
   - **HLA1** High Level Assembler Release 1
   - **HLA2** High Level Assembler Release 2
   - **HLA3** High Level Assembler Release 3
   - **HLA4** High Level Assembler Release 4
   - **HLA5** High Level Assembler Release 5
2. **Type:**
   - **A** Arithmetic
   - **B** Boolean
   - **C** Character
3. **Type Attr:**
   - **N** Numeric (self-defining term)
   - **O** Omitted
   - **U** Undefined, unknown, deleted, or unassigned
4. **Scope:**
   - **L** Local - only in macro
   - **G** Global - in entire program

Appendix D. Standard character set code table

Table 74. Standard character set code table - from code page 00037

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
00	0		0000 0000	20	32		0010 0000
01	1		0000 0001	21	33		0010 0001
02	2		0000 0010	22	34		0010 0010
03	3		0000 0011	23	35		0010 0011
04	4		0000 0100	24	36		0010 0100
05	5		0000 0101	25	37		0010 0101
06	6		0000 0110	26	38		0010 0110
07	7		0000 0111	27	39		0010 0111
08	8		0000 1000	28	40		0010 1000
09	9		0000 1001	29	41		0010 1001
0A	10		0000 1010	2A	42		0010 1010
0B	11		0000 1011	2B	43		0010 1011
0C	12		0000 1100	2C	44		0010 1100
0D	13		0000 1101	2D	45		0010 1101
0E	14		0000 1110	2E	46		0010 1110
0F	15		0000 1111	2F	47		0010 1111
10	16		0001 0000	30	48		0011 0000
11	17		0001 0001	31	49		0011 0001
12	18		0001 0010	32	50		0011 0010
13	19		0001 0011	33	51		0011 0011
14	20		0001 0100	34	52		0011 0100
15	21		0001 0101	35	53		0011 0101
16	22		0001 0110	36	54		0011 0110
17	23		0001 0111	37	55		0011 0111
18	24		0001 1000	38	56		0011 1000
19	25		0001 1001	39	57		0011 1001
1A	26		0001 1010	3A	58		0011 1010
1B	27		0001 1011	3B	59		0011 1011
1C	28		0001 1100	3C	60		0011 1100
1D	29		0001 1101	3D	61		0011 1101
1E	30		0001 1110	3E	62		0011 1110
1F	31		0001 1111	3F	63		0011 1111

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
40	64	SPACE	0100 0000	60	96	-	0110 0000
41	65		0100 0001	61	97	/	0110 0001

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
42	66		0100 0010	62	98		0110 0010
43	67		0100 0011	63	99		0110 0011
44	68		0100 0100	64	100		0110 0100
45	69		0100 0101	65	101		0110 0101
46	70		0100 0110	66	102		0110 0110
47	71		0100 0111	67	103		0110 0111
48	72		0100 1000	68	104		0110 1000
49	73		0100 1001	69	105		0110 1001
4A	74		0100 1010	6A	106		0110 1010
4B	75	.	0100 1011	6B	107	,	0110 1011
4C	76		0100 1100	6C	108		0110 1100
4D	77	(	0100 1101	6D	109	_	0110 1101
4E	78	+	0100 1110	6E	110		0110 1110
4F	79		0100 1111	6F	111		0110 1111
50	80	&	0101 0000	70	112		0111 0000
51	81		0101 0001	71	113		0111 0001
52	82		0101 0010	72	114		0111 0010
53	83		0101 0011	73	115		0111 0011
54	84		0101 0100	74	116		0111 0100
55	85		0101 0101	75	117		0111 0101
56	86		0101 0110	76	118		0111 0110
57	87		0101 0111	77	119		0111 0111
58	88		0101 1000	78	120		0111 1000
59	89		0101 1001	79	121		0111 1001
5A	90		0101 1010	7A	122		0111 1010
5B	91	$	0101 1011	7B	123	#	0111 1011
5C	92	*	0101 1100	7C	124	@	0111 1100
5D	93	)	0101 1101	7D	125	'	0111 1101
5E	94		0101 1110	7E	126	=	0111 1110
5F	95		0101 1111	7F	127		0111 1111

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
80	128		1000 0000	A0	160		1010 0000
81	129	a	1000 0001	A1	161		1010 0001
82	130	b	1000 0010	A2	162	s	1010 0010
83	131	c	1000 0011	A3	163	t	1010 0011
84	132	d	1000 0100	A4	164	u	1010 0100
85	133	e	1000 0101	A5	165	v	1010 0101
86	134	f	1000 0110	A6	166	w	1010 0110
87	135	g	1000 0111	A7	167	x	1010 0111
88	136	h	1000 1000	A8	168	y	1010 1000

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
89	137	i	1000 1001	A9	169	z	1010 1001
8A	138		1000 1010	AA	170		1010 1010
8B	139		1000 1011	AB	171		1010 1011
8C	140		1000 1100	AC	172		1010 1100
8D	141		1000 1101	AD	173		1010 1101
8E	142		1000 1110	AE	174		1010 1110
8F	143		1000 1111	AF	175		1010 1111
90	144		1001 0000	B0	176		1011 0000
91	145	j	1001 0001	B1	177		1011 0001
92	146	k	1001 0010	B2	178		1011 0010
93	147	l	1001 0011	B3	179		1011 0011
94	148	m	1001 0100	B4	180		1011 0100
95	149	n	1001 0101	B5	181		1011 0101
96	150	o	1001 0110	B6	182		1011 0110
97	151	p	1001 0111	B7	183		1011 0111
98	152	q	1001 1000	B8	184		1011 1000
99	153	r	1001 1001	B9	185		1011 1001
9A	154		1001 1010	BA	186		1011 1010
9B	155		1001 1011	BB	187		1011 1011
9C	156		1001 1100	BC	188		1011 1100
9D	157		1001 1101	BD	189		1011 1101
9E	158		1001 1110	BE	190		1011 1110
9F	159		1001 1111	BF	191		1011 1111

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
C0	192		1100 0000	E0	224		1110 0000
C1	193	A	1100 0001	E1	225		1110 0001
C2	194	B	1100 0010	E2	226	S	1110 0010
C3	195	C	1100 0011	E3	227	T	1110 0011
C4	196	D	1100 0100	E4	228	U	1110 0100
C5	197	E	1100 0101	E5	229	V	1110 0101
C6	198	F	1100 0110	E6	230	W	1110 0110
C7	199	G	1100 0111	E7	231	X	1110 0111
C8	200	H	1100 1000	E8	232	Y	1110 1000
C9	201	I	1100 1001	E9	233	Z	1110 1001
CA	202		1100 1010	EA	234		1110 1010
CB	203		1100 1011	EB	235		1110 1011
CC	204		1100 1100	EC	236		1110 1100
CD	205		1100 1101	ED	237		1110 1101
CE	206		1100 1110	EE	238		1110 1110
CF	207		1100 1111	EF	239		1110 1111

Hex.	Dec.	EBCDIC	Binary	Hex.	Dec.	EBCDIC	Binary
D0	208		1101 0000	F0	240	0	1111 0000
D1	209	J	1101 0001	F1	241	1	1111 0001
D2	210	K	1101 0010	F2	242	2	1111 0010
D3	211	L	1101 0011	F3	243	3	1111 0011
D4	212	M	1101 0100	F4	244	4	1111 0100
D5	213	N	1101 0101	F5	245	5	1111 0101
D6	214	O	1101 0110	F6	246	6	1111 0110
D7	215	P	1101 0111	F7	247	7	1111 0111
D8	216	Q	1101 1000	F8	248	8	1111 1000
D9	217	R	1101 1001	F9	249	9	1111 1001
DA	218		1101 1010	FA	250		1111 1010
DB	219		1101 1011	FB	251		1111 1011
DC	220		1101 1100	FC	252		1111 1100
DD	221		1101 1101	FD	253		1111 1101
DE	222		1101 1110	FE	254		1111 1110
DF	223		1101 1111	FF	255		1111 1111

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie New York 12601-5400
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at Copyright and trademark information at http://www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Bibliography

High Level Assembler Documents

HLASM General Information, GC26-4943
HLASM Installation and Customization Guide, SC26-3494
HLASM Language Reference, SC26-4940
HLASM Programmer's Guide, SC26-4941

Toolkit Feature document

HLASM Toolkit Feature User's Guide, GC26-8710
HLASM Toolkit Feature Debug Reference Summary, GC26-8712
HLASM Toolkit Feature Interactive Debug Facility User's Guide, GC26-8709
HLASM Toolkit Feature Installation and Customization Guide, GC26-8711

Related documents (Architecture)

z/Architecture Principles of Operation, SA22-7832

Related documents for z/OS

z/OS:
z/OS MVS JCL Reference, SA23-1385
z/OS MVS JCL User's Guide, SA23-1386
z/OS MVS Programming: Assembler Services Guide, SA23-1368
z/OS MVS Programming: Assembler Services Reference, Volume 1 (ABE-HSP), SA23-1369
z/OS MVS Programming: Assembler Services Reference, Volume 2 (IAR-XCT), SA23-1370
z/OS MVS Programming: Authorized Assembler Services Guide, SA23-1371
z/OS MVS Programming: Authorized Assembler Services Reference, Volumes 1 - 4, SA23-1372 - SA23-1375
z/OS MVS Program Management: User's Guide and Reference, SA23-1393
z/OS MVS System Codes, SA38-0665
z/OS MVS System Commands, SA38-0666
z/OS MVS System Messages, Volumes 1 - 10, SA38-0668 - SA38-0677
z/OS Communications Server: SNA Programming, SC27-3674
UNIX System Services:
z/OS UNIX System Services User's Guide, SA23-2279
DFSMS/MVS:
z/OS DFSMS Program Management, SC27-1130
z/OS DFSMSdfp Utilities, SC23-6864
TSO/E (z/OS):
z/OS TSO/E Command Reference, SA32-0975
SMP/E (z/OS):
SMP/E for z/OS Messages, Codes, and Diagnosis, GA32-0883
SMP/E for z/OS Reference, SA23-2276
SMP/E for z/OS User's Guide, SA23-2277

Related documents for z/VM

z/VM: VMSES/E Introduction and Reference, GC24-6243
z/VM: Service Guide, GC24-6247
z/VM: CMS Commands and Utilities Reference, SC24-6166
z/VM: CMS File Pool Planning, Administration, and Operation, SC24-6167
z/VM: CP Planning and Administration, SC24-6178
z/VM: Saved Segments Planning and Administration, SC24-6229
z/VM: Other Components Messages and Codes, GC24-6207
z/VM: CMS and REXX/VM Messages and Codes, GC24-6161
z/VM: CP System Messages and Codes, GC24-6177
z/VM: CMS Application Development Guide, SC24-6162
z/VM: CMS Application Development Guide for Assembler, SC24-6163
z/VM: CMS User's Guide, SC24-6173
z/VM: XEDIT User's Guide, SC24-6245
z/VM: XEDIT Commands and Macros Reference, SC24-6244
z/VM: CP Commands and Utilities Reference, SC24-6175

Related documents for z/VSE

z/VSE: Guide to System Functions, SC33-8312
z/VSE: Administration, SC34-2627
z/VSE: Installation, SC34-2631
z/VSE: Planning, SC34-2635
z/VSE: System Control Statements, SC34-2637
z/VSE: Messages and Codes, Vol.1 , SC34-2632
z/VSE: Messages and Codes, Vol.2, SC34-2633
z/VSE: Messages and Codes, Vol.3, SC34-2634
REXX/VSE Reference, SC33-6642
REXX/VSE User's Guide, SC33-6641

Index

Special characters

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

SC26-4940-06