



---

Assembler H Version 2

GC26-4037-1

**Language Reference**

Release 1

---

---

## **Second Edition (December 1987)**

This edition replaces and makes obsolete the previous edition, GC26-4037-0, and its technical newsletter, GN26-8318.

This edition applies to Release 1 of Assembler H Version 2, Licensed Program 5668-962, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. This manual merges assembler information contained in *OS/VS-DOS/VSE-VM/370 Assembler Language*, GC33-4010, and *OS Assembler H Language*, GC26-3771.

The changes for this edition are summarized under "Summary of Changes" following the preface. Specific changes are indicated by a vertical bar to the left of the change. These bars will be deleted at any republication of the page affected. Editorial changes that have no technical significance are not noted.

Changes are made periodically to this publication; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, and 4300 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's program may be used. Any functionally equivalent program may be used instead.

Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality. If you request publications from the address given below, your order will be delayed because publications are not stocked there.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, Programming Publishing, P. O. Box 49023, San Jose, California, U.S.A. 95161-9023. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

---

## Preface

This manual contains language reference information for the Assembler H Version 2, Release 1, Licensed Program 5668-962, hereafter referred to as Assembler H or, simply, assembler.

This manual merges assembler information contained in *OS/VS – DOS/VSE – VM/370 Assembler Language*, GC33-4010, and *OS Assembler H Language*, GC26-3771, with the following major differences:

- Only information relevant to Assembler H has been included in this manual. DOS/VSE, OS/MFT, and OS/MVT information has been removed because it is valid only for assemblers other than Assembler H.
- New features provided by Assembler H Version 2, Release 1, have been integrated (see the Summary of Amendments for details).
- Programs may be assembled with Assembler H Version 2, Release 1, under MVS/Extended Architecture (MVS/XA).
- Information available in manuals listed below under “Related Publications” is not included in this publication; references are made to the appropriate manuals.

---

## Whom This Manual Is For

This manual is for application programmers coding in the Assembler H language. It is not intended to be used for tutorial purposes; it is for reference only. If you are interested in learning more about assemblers, most libraries have tutorial books on the subject.

---

## Organization of Manual

### Part 1. Assembler Language

- “Chapter 1. Introduction to Assembler Language” describes what the assembler does, tells about the language and program, gives the relationship of the assembler to the operating system, and supplies some coding aids.
- “Chapter 2. Coding and Structure” describes the coding rules for and the structure of the assembler language. It also discusses terms and expressions.
- “Chapter 3. Addressing, Program Sectioning, and Linking” describes how to handle addressing, control and dummy sections, and symbolic linking.
- “Chapter 4. Machine Instruction Statements” describes the machine instruction types and their formats.
- “Chapter 5. Assembler Instruction Statements” describes the assembler instructions.

## Part 2. Macro Language

- “Chapter 6. Introduction to Macro Language” describes the macro instruction statement, definition, library, and so forth.
- “Chapter 7. How to Prepare Macro Definitions” describes components of a macro definition.
- “Chapter 8. How to Write Macro Instructions” describes the format of operands, sublists, and levels of macro instructions.
- “Chapter 9. How to Write Conditional Assembly Instructions” describes the SET and sequence symbols, and attributes of assembly instructions.

## Part 3. Appendixes

- “Appendix A, Assembler Instructions and Statements” lists the related name, operation, and operand entries.
- “Appendix B, Summary of Constants” lists the constant types and gives related information concerning each.
- “Appendix C, Macro Language Summary” summarizes some of the information contained in Part 2. information contained in Part 2.

---

## Assembler H Version 2 Publications

- *Assembler H Version 2 General Information*, GC26-4035, contains a brief description of Assembler H and compares Version 2, Release 1, features with those of Version 1, Release 5, and also compares Assembler H features with those of the VS Assembler.
- *Assembler H Version 2 Installation*, SC26-4030, which contains information necessary for installation of the assembler program.
- *Assembler H Version 2 Programming Guide*, SC26-4036, tells how to use Assembler H, provides an explanation of each of the diagnostic and abnormal termination messages issued by Assembler H, and suggests how you should respond in each case.
- *Assembler H Version 2 Language Reference*, GC26-4037, describes the basic assembler language functions and specifications that are available with Assembler H.
- *Assembler H Version 2 Logic*, LY26-3908, describes the design logic and functional characteristics of Assembler H.
- *Assembler Coding Form*, GX28-6509, is a form for coding the program in the proper columns.

---

## Related Publications

The following publications provide definitive information about machine instructions:

- *IBM System/370 Principles of Operation*, GA22-7000
- *IBM System/370 Extended Architecture Principles of Operation*, SA22-7085
- *IBM System/370 Vector Operations*, SA22-7125
- *IBM 4300 Processors Principles of Operation*, GA22-7070

For quick reference, see:

- *IBM System/370 Reference Summary*, GX20-1850
- *IBM System/370 Extended Architecture Reference Summary*, GX20-0157



---

# Summary of Changes

---

## Release 1 Update, December 1987

### Changes to the Product

- Support has been added for the IBM DBCS-Host double-byte character set. Double-byte data can be used wherever single-byte data, enclosed by apostrophes, is allowed. Refer to the Glossary of the *Assembler H Version 2 Programming Guide* for the definition of DBCS terms.
- The MHELP facility has been extended. MHELP can now dump SETC symbols and parameters in hexadecimal as well as in EBCDIC format.
- New machine instructions have been added for the IBM 3090 Vector Facility.
- Extended and System/370 instruction sets are now contained in the Universal instruction set.
- The underscore character (`_`) is now allowed in variable symbols and inline macro names, as well as in ordinary symbols.

### Changes to This Manual

Documentation of the above product changes, as well as miscellaneous corrections to existing information, has been added.





# Contents

---

<b>Part 1. Assembler Language</b> .....	1
<b>Chapter 1. Introduction to Assembler Language</b> .....	3
Language Compatibility .....	3
Assembler Language .....	3
Assembler Program .....	4
Relationship of Assembler to Operating System .....	6
Coding Aids .....	7
<b>Chapter 2. Coding and Structure</b> .....	9
Double-Byte Data Notation .....	9
Assembler Language Coding Conventions .....	9
Field Boundaries .....	10
Continuation Lines .....	11
Comments Statement Format .....	14
Instruction Statement Format .....	14
Character Set .....	16
Assembler Language Structure .....	17
Terms and Expressions .....	25
Terms .....	25
Literals .....	36
Expressions .....	39
<b>Chapter 3. Addressing, Program Sectioning, and Linking</b> .....	43
Addressing .....	43
Addressing within Source Modules: Establishing Addressability .....	43
Base Register Instructions .....	43
Relative Addressing .....	48
Program Sectioning and Linking .....	48
Source Module .....	49
Control Sections .....	50
Location Counter Setting .....	51
First Control Section .....	53
Unnamed Control Section .....	55
Literal Pools In Control Sections .....	55
External Symbol Dictionary Entries .....	55
Establishing Residence and Addressing Mode .....	56
Defining a Control Section .....	59
External Dummy Sections .....	66
Symbolic Linkages .....	69
<b>Chapter 4. Machine Instruction Statements</b> .....	73
General Instructions .....	73
Decimal Instructions .....	74
Floating-Point Instructions .....	74
Control Instructions .....	74
Input/Output Operations .....	75
Branching with Extended Mnemonic Codes .....	75
Statement Formats .....	77

Symbolic Operation Codes . . . . .	78
Operand Entries . . . . .	78
Registers . . . . .	79
Addresses . . . . .	80
Lengths . . . . .	83
Immediate Data . . . . .	83
Examples of Coded Machine Instructions . . . . .	84
<b>Chapter 5. Assembler Instruction Statements . . . . .</b>	<b>91</b>
Symbol Definition Instruction . . . . .	92
EQU—Equate Symbol . . . . .	92
Redefining Symbolic Operation Codes . . . . .	94
OPSYN—Equate Operation Code . . . . .	94
Data Definition Instructions . . . . .	96
DC—Define Constant . . . . .	96
DS—Define Storage . . . . .	127
CCW or CCW0—Define Channel Command Word (Format 0) . . . . .	130
CCW1—Define Channel Command Word (Format 1) . . . . .	131
Program Control Instructions . . . . .	132
ICTL—Input Format Control . . . . .	132
ISEQ—Input Sequence Checking . . . . .	133
PUNCH—Punch a Card . . . . .	134
REPRO—Reproduce Following Card . . . . .	136
PUSH Instruction . . . . .	137
POP Instruction . . . . .	137
ORG—Set Location Counter . . . . .	138
LORG—Begin Literal Pool . . . . .	140
CNOP—Conditional No Operation . . . . .	143
COPY—Copy Predefined Source Coding . . . . .	144
END—End Assembly . . . . .	145
Listing Control Instructions . . . . .	146
TITLE—Identify Assembly Output . . . . .	146
EJECT—Start New Page . . . . .	148
SPACE—Space Listing . . . . .	148
PRINT—Print Optional Data . . . . .	149
<b>Part 2. Macro Language . . . . .</b>	<b>151</b>
<b>Chapter 6. Introduction to Macro Language . . . . .</b>	<b>153</b>
Using Macros . . . . .	153
Macro Definition . . . . .	153
Macro Instruction Statement . . . . .	155
Source and Library Macro Definitions . . . . .	156
Macro Library . . . . .	156
System Macro Instructions . . . . .	157
Conditional Assembly Language . . . . .	157
<b>Chapter 7. How to Prepare Macro Definitions . . . . .</b>	<b>159</b>
Where to Define a Macro in a Source Module . . . . .	159
Open Code . . . . .	160
Format of a Macro Definition . . . . .	160
MACRO—Macro Definition Header . . . . .	160
MEND—Macro Definition Trailer . . . . .	160
Macro Instruction Prototype . . . . .	161

Name Field	161
Operation Field	161
Operand Field	162
Body of a Macro Definition	163
Model Statements	163
Variable Symbols as Points of Substitution	164
Listing of Generated Fields	164
Rules for Concatenation	165
Rules for Model Statement Fields	168
Symbolic Parameters	170
Positional Parameters	171
Keyword Parameters	172
Combining Positional and Keyword Parameters	174
Subscripted Symbolic Parameters	175
Processing Statements	176
Conditional Assembly Instructions	176
Inner Macro Instructions	176
COPY Instruction	176
MNOTE Instruction	176
MEXIT Instruction	179
AREAD—Assign Character String Value	181
Comments Statements	182
Ordinary Comments Statements	182
Internal Macro Comments Statements	182
System Variable Symbols	183
<b>Chapter 8. How to Write Macro Instructions</b>	193
Where Macro Instructions Can Appear	193
Macro Instruction Format	193
Alternative Ways of Coding a Macro Instruction	194
Name Entry	194
Operation Entry	195
Operand Entry	195
Sublists in Operands	199
Values in Operands	202
Omitted Operands	202
Special Characters	203
Nesting in Macro Definitions	206
Inner and Outer Macro Instructions	206
Levels of Nesting	206
General Rules and Restrictions	206
Passing Values through Nesting Levels	208
System Variable Symbols in Nested Macros	208
<b>Chapter 9. How to Write Conditional Assembly Instructions</b>	211
Elements and Functions	211
SET Symbols	212
Data Attributes	215
Sequence Symbols	224
Attribute Definition and Lookahead	225
Declaring SET Symbols	226
LCLA, LCLB, LCLC—Define Local Set Symbols	226
GBLA, GBLB, and GBLC Instructions	228
Assigning Values to SET Symbols	230
SETA—Set Arithmetic	230

SETB—Set Binary . . . . .	236
SETC—Set Character . . . . .	241
Extended SET Statements . . . . .	248
Substring Notation . . . . .	248
Branching . . . . .	251
AIF—Conditional Branch . . . . .	251
AGO—Unconditional Branch . . . . .	253
ACTR—Conditional Assembly Loop Counter . . . . .	254
ANOP—Assembly No Operation . . . . .	255
Open Code . . . . .	256
MHELP—Macro Trace Facility . . . . .	257

---

<b>Appendixes</b> . . . . .	261
<b>Appendix A. Assembler Instructions and Statements</b> . . . . .	263
<b>Appendix B. Summary of Constants</b> . . . . .	267
<b>Appendix C. Macro Language Summary</b> . . . . .	269
<b>Index</b> . . . . .	275

# Figures

1.	Double-Byte Data Notation	9
2.	Standard Assembler Coding Form	10
3.	Character Set	16
4.	Double-Byte Character Set (DBCS)	17
5.	Examples Using Character Set	19
6.	Assembler Language Structure	20
7.	Machine Instructions	21
8.	Ordinary Assembler Instruction Statements	22
9.	Conditional Assembly Instructions	23
10.	Macro Instructions	24
11.	Summary of Terms	25
12.	Transition from Assembler Language Statement to Object Code	28
13.	Assignment of Length Attribute Values to Symbols in Name Fields	34
14.	Differences between Literals, Constants, and Self-Defining Terms	36
15.	Differences between Literals, Constants, and Self-Defining Terms	37
16.	Examples of Valid Expressions	39
17.	Definitions of Absolute and Relocatable Expressions	40
18.	Use of Multiple Location Counters	52
19.	Defining CSECTs, DSECTs, and Symbols	56
20.	How the Location Counter Works	61
21.	Extended Mnemonic Codes	76
22.	Object Code Format	82
23.	Types of Data Constants	97
24.	Length Attribute Value of Symbol Naming Constants	99
25.	Alignment of Constants	100
26.	Type Codes for Constants	102
27.	Binary Constants	108
28.	Character Constants	109
29.	Graphic Constants	111
30.	Hexadecimal Constants	112
31.	Fixed-Point Constants	114
32.	Decimal Constants	116
33.	A and Y Address Constants	118
34.	S Address Constants	120
35.	V Address Constants	121
36.	Q Address Constants	122
37.	Floating-Point Constants	124
38.	Floating-Point External Formats	125
39.	Channel Command Word, Format 0	130
40.	Channel Command Word, Format 1	131
41.	Building a Translate Table	139
42.	CNOP Alignment	144
43.	Parts of a Macro Definition	154
44.	Format of a Macro Definition	160
45.	Rules for Concatenation	167
46.	Positional Parameters	172
47.	Keyword Parameters	174
48.	Combining Positional and Keyword Parameters	175
49.	Rules for MNOTE Character Strings	179
50.	MEXIT Operation	180

51.	Relationship between Keyword Operands and Keyword Parameters and Their Assigned Values . . . . .	198
52.	Sublists in Operands . . . . .	200
53.	Relationship between Subscripted Parameters and Sublist Entries . . . . .	201
54.	Values in Nested Macro Calls . . . . .	207
55.	Passing Values through Nesting Levels . . . . .	209
56.	Features of SET Symbols and Other Types of Variable Symbols . . . . .	213
57.	Data Attributes . . . . .	216
58.	Attributes and Related Symbols . . . . .	217
59.	Relationship of Integer to Length and Scaling Attributes . . . . .	222
60.	Use of Arithmetic (SETA) Expressions . . . . .	231
61.	Defining Arithmetic (SETA) Expressions . . . . .	232
62.	Variable Symbols Allowed as Terms in Arithmetic Expressions . . . . .	232
63.	Defining Logical Expressions . . . . .	239
64.	Subscripted SETC Symbols . . . . .	243
65.	Use of Character Expressions . . . . .	244
66.	Substring Notation in Conditional Assembly Instructions . . . . .	249
67.	Summary of Substring Notation . . . . .	250
68.	Restrictions on Coding Expressions . . . . .	257
69.	Assembler Instructions . . . . .	263
70.	Assembler Statements . . . . .	266
71.	Summary of Constants . . . . .	267
72.	Macro Language Elements . . . . .	270
73.	Conditional Assembly Expressions . . . . .	271
74.	Attributes . . . . .	272
75.	Variable Symbols . . . . .	273

---

## Part 1. Assembler Language

- “Chapter 1. Introduction to Assembler Language” describes what the assembler does, tells about the language and program, gives the relationship of the assembler to the operating system, and supplies some coding aids.
- “Chapter 2. Coding and Structure” describes the coding rules for and the structure of the assembler language. It also discusses terms and expressions.
- “Chapter 3. Addressing, Program Sectioning, and Linking” talks about how to handle addressing, control and dummy sections, and symbolic linking.
- “Chapter 4. Machine Instruction Statements” describes the machine instruction types and their formats.
- “Chapter 5. Assembler Instruction Statements” describes the assembler instructions.





---

## Chapter 1. Introduction to Assembler Language

A computer can understand and interpret only machine language. Machine language is in binary form and, thus, very 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 allows you to 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 execute your program, as explained in the following paragraph.

Your program, written in the assembler language, becomes the source module that is input to the assembler. It can be punched into a deck of cards, or entered through a terminal. 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 another processing program, called the linkage editor. The linkage editor produces a load module that can be loaded later into the main storage of the computer. Once your program is loaded, it can then be executed. Your source module and the object code produced are printed, along with other information, on a program listing.

---

### Language Compatibility

The language used by Assembler H Version 2, Release 1, has functional extensions to the language supported by VS Assembler and OS Assembler H Version 1, Release 5. Programs written for VS Assembler and OS Assembler H Version 1, Release 5, that were successfully assembled with no warning or diagnostic messages, will be assembled correctly by Assembler H Version 2, Release 1.

---

### Assembler Language

The assembler language is the symbolic programming language that lies closest to the machine language in form and content. You will, therefore, find the assembler language useful when

- you need to control your program closely, down to the byte and even the bit level, or
- 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 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 IBM System/370 architecture instruction set, or of the IBM System/370 extended architecture instruction set. It is called a machine instruction because the assembler translates it into the machine language code the computer can execute. Machine instructions are described in "Chapter 4. Machine Instruction Statements."

## Assembler Instructions

An assembler instruction is a request to the assembler program to perform certain operations during the assembly of a source module; for example, defining data constants, defining the end of the source module, and reserving storage areas. Except for the instructions that define constants, the assembler does not translate assembler instructions into object code. The assembler instructions are described in "Chapter 3. Addressing, Program Sectioning, and Linking," "Chapter 5. Assembler Instruction Statements," and "Chapter 9. How to Write Conditional Assembly Instructions."

## Macro Instructions

A macro instruction is a request to the assembler program to process a predefined sequence of code 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 appropriate *Macro Instructions* manual.)

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

A complete description of the macro facility, including the macro definition, the macro instruction, and the conditional assembly language, is given in "Part 2. Macro Language."

---

## 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 furnishes a printed listing of the source statements and object program statements and additional information, such as error indications, that are useful in analyzing the program. The object program is in the format required by the linkage editor.

## 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 on two occasions: First at preassembly time and, later, at assembly time. However, it does some processing—for example, macro processing—only at preassembly time.

The assembler also produces information for other processors. The linkage editor uses such information at link-edit time to combine object modules into load modules. The loader loads your program (combined load modules) into virtual storage at program fetch time. Finally, at execution time, the computer executes the object code produced by the assembler at assembly time.

1. The assembler processes all machine instructions, and translates them into object code at assembly time.
2. Assembler instructions are divided into two main types:
  - Ordinary assembler instructions
  - Conditional assembly instructions and the macro processing instructions (MACRO, MEND, MEXIT, MNOTE, and AREAD)

The following discusses these two main types of assembler instructions.

- a. The assembler processes ordinary assembler instructions at assembly time.
  - The assembler evaluates absolute and relocatable expressions at assembly time; they are sometimes called assembly-time expressions.
  - Some instructions produce output for processing after assembly time (DC, DS, CCW, CCW0, CCW1, ENTRY, EXTRN, WXTRN, PUNCH, and REPRO).
- b. The assembler processes conditional assembly instructions and macro processing instructions at preassembly time.
  - The assembler evaluates the conditional assembly expressions—arithmetic, logical, and character—at preassembly time.
  - The assembler processes the machine and assembler instructions generated from preassembly processing at assembly time.

3. The assembler processes macro instructions at preassembly time.

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

The assembler prints in a program listing all the information it produces at the various processing times discussed above.

---

## Relationship of Assembler to Operating System

Assembler H operates under MVS/Extended Architecture (XA), OS/VS2 MVS 3.8, OS/VS1 Release 7, MVS/System Product (SP) V1, VM/XA System Product (VM/XA SP), VM/XA Systems Facility (VM/XA SF), and VM/SP. 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 the desired service functions from the operating system.

OS/VS provides the following services:

1. For assembling the source module:
  - A control program
  - Libraries to contain source code and macro definitions
  - Utilities
2. For preparing for the execution of the assembler program as represented by the object module:
  - A control program
  - Storage allocation
  - Input and output facilities
  - Linkage editor
  - A loader

CMS provides the following services:

1. For assembling the source module:
  - An interactive control program
  - Files to contain source code and macro definitions
  - Utilities
2. 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
  - CMS loader

---

## Coding Aids

It can be very difficult to write an assembler language program using only machine instructions. The assembler provides additional functions that make this task easier. They are summarized below.

### 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.

### 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 appropriate assembler instruction, the assembler will compute the displacement from a base address of any symbolic addresses you specify in a machine instruction. It will insert 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 the base-displacement form. The computer obtains the required address by adding the displacement to the base address contained in the base register.

### Relocatability

The assembler produces an object module that can be relocated from an originally assigned storage area to any other 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. This means that a sectioned program may be loaded and executed even though a continuous block of storage large enough to accommodate the entire program may not be available.

## **Linkage between Source Modules**

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

A discussion of sectioning and linking is contained in "Program Sectioning and Linking" on page 48.

## **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 partly control the form and content of the listing.

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

---

## Chapter 2. Coding and Structure

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

---

### Double-Byte Data Notation

Character(s)	Represents
<	shift-out (SO)
>	shift-in (SI)
D1D2D3...	double-byte characters
DaDbDc...	double-byte characters
.A.B.C.'&.,	double-byte characters: A, B, C, apostrophe, ampersand and comma
eeeeee	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

Figure 1. Double-Byte Data Notation

**Notes:**

1. The DBCS ampersand and apostrophe are not recognized as delimiters.
2. 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.

---

### Assembler Language Coding Conventions

Figure 2 shows the format conventions used to code an assembler program.





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, by changing the standard begin, end, and continue columns can create a field before the begin column; this field can then contain the identification-sequence field. 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-blank character in this column indicates that the current statement is continued on the next line. This column must be blank if a statement is completed on the same line; otherwise, the assembler will treat the statement that follows on the next line as a continuation line of the current statement.

If the assembler is invoked with the DBCS option, 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 performed.
- 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 will verify whether or not the source statements are in the correct sequence.

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

### Continuation Lines

To continue a statement on another line, the following rules apply:

1. Enter a non-blank character in the continuation indicator field (column 72). This non-blank character must not be part of the statement coding. When more than one continuation line is needed, a non-blank character must be entered 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 blank. Comments may be continued after column 16.

Note that, 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 totally new statement after column 16 on the next line, this statement will be taken as a comment belonging to the previous statement.

Unless it is one of the statement types listed below, 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:

- Prototype statement of a macro definition
- Macro instruction statement
- AGO conditional assembly statement
- AIF conditional assembly statement
- GBLA, GBLB, and GBLC conditional assembly statements
- LCLA, LCLB, and LCLC conditional assembly statements
- SETA, SETB, and SETC conditional assembly statements

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 may 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 Assembler H source input or listings is required, special features of the Assembler H language may be used. Assembler H, when invoked with the DBCS option, provides sufficient 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 is performed.
- An extended continuation indicator provides a flexible end column on a line-by-line basis so that any alignment of double-byte data in a source statement can be supported. The end column of continued lines may 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, neither SO nor SI is regarded as a continuation indicator. If either are so used, warning message IEV201 "SO OR SI IN CONTINUATION COLUMN - NO CONTINUATION ASSUMED" is issued.

The use of these features is illustrated by the examples below. Refer to "Double-Byte Data Notation" on page 9 for the notation used in the examples.

#### Source Input Considerations

- Extended continuation indicators may 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.
- On a line with a nonblank continuation indicator, the end column will be 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, care must be taken to 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 error IEV205 "EXTENDED CONTINUATION COLUMN MUST NOT EXTEND INTO CONTINUE COLUMN" will be issued and the extended continuation indicators are treated as data.
- 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	Cont.
DBCS1	DC	C'<D1D2D3D4D5D6D7D8D9>XXXXXXXXXXXXXXXXXXXX <DaDb>'	
DBCS2	DC	C'abcdefghijklmnopqXXXXXXXXXXXXXXXXXXXX <DaDb>'	
DBCS3	DC	C'abc<D1D2D3D4D5D6D7>XXXXXXXXXXXXXXXXXXXX <DaDb>'	

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 assemble value of DBCS1 is:

```
<D1D2D3D4D5D6D7D8D9DaDb>
```

The DBCS2 constant contains an EBCDIC string which is followed by a double-byte string. Since 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:

```
abcdefghijklmnopq<DaDb>
```

The DBCS3 constant contains 3 EBCDIC characters followed by 9 double-byte characters. Alignment of the double-byte data requires that the end column be extended one column to the left. The SI and SO at the continuation point are not assembled into the operand. The assemble value of DBCS3 is:

```
abc<D1D2D3D4D5D6D7DaDb>
```

### *Source Listing Considerations*

- For source that does not contain substituted variable symbols, the listing will exactly reflect the source input.
- Double-byte data input from code-generation programs, and containing no substituted variables, will not be made readable in the listing if the source input was unreadable on a device capable of presenting DBCS characters.
- Refer to "Listing of generated fields containing double-byte data" on page 165 for details of extended continuation and macro-generated statements.

### **Comments Statement Format**

Comments statements are not assembled as part of the object module, but are only printed in the assembly listing. As many comments statements as needed can be written, subject to the following rules:

- Comments statements require an asterisk in the begin column.  
**Note:** Internal macro definition comments statements require a period in the begin column, followed by an asterisk.
- Any characters of the IBM System/370 character set, including blanks, special characters, and double-byte characters, can be used (see "Character Set" on page 16)
- Comments statements must lie in the statement field and not run over into the continuation indicator field; otherwise, the statement following the comments statement will be considered as a continuation line of that comments statement.
- Comments 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 blanks, and must be written in the order stated.

### **Fixed Format**

The standard coding form (Figure 2 on page 10) is divided into fields that provide fixed positions for the first three entries, as follows:

- An 8-character name field starting in column 1
- A 5-character operation field starting in column 10
- An operand field that begins in column 16

**Note:** With this fixed format, one blank separates each field.

## Free Format

It is not necessary to code the name, operation, and operand entries according to the fixed fields on the standard coding form. Instead, these entries can be written in any position, subject to the formatting specifications below.

### Formatting Specifications

Whether using fixed or free format, the following general rules apply to the coding of an instruction statement:

1. The entries must be written in the following order: name, operation, operand, and remarks.
2. 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.
3. The entries must be separated from each other by one or more blanks.
4. If used, a name entry must start in the begin column.
5. The name and operation entries, each followed by at least one blank, must be contained in the first line of an instruction statement.
6. The operation entry must begin at least one column to the right of the begin column.

A description of the name, operation, operand, and remarks entries follows:

**Name Entry:** The name entry is a symbol created by you to identify an instruction statement. A name entry is usually optional. It must be a valid symbol at assembly time (after substitution for variable symbols, if specified); for an exception, see "TITLE—Identify Assembly Output" on page 146.

The symbol must consist of 63 characters or less, and be entered with the first character appearing in the begin column. The first character must be alphabetic. If the begin column is blank, the assembler program assumes no name has been entered. No blanks or double-byte data may appear in the symbol.

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

- An operation entry is mandatory.
- For machine and assembler instructions, it must be a valid symbol at assembly time (after substitution for variable symbols, if specified). The standard symbolic operation codes are five characters or less (see the appropriate principles of operation manual; or, for assembler operations, see Appendix A, "Assembler Instructions and Statements").

The standard set of codes can be changed by OPSYN instructions (see "OPSYN—Equate Operation Code" on page 94).

- For macro instructions, it can be any valid symbol that is not identical to any machine or assembler op-code.

**Operand Entries:** Operand entries contain one 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 usually required, depending on the instruction.
- Operands must be separated by commas. No blanks are allowed between the operands and the commas that separate them.
- Operands must not contain embedded blanks, because a blank normally indicates the end of the operand entry. However, blanks are allowed if they are included in character strings enclosed in single quotation marks, or in logical expressions.

**Remarks Entries:** Remarks are used to describe the current instruction.

- Remarks are optional.
- They can contain any of the 256 valid characters (or punch combinations) of the appropriate character set, including blanks, special characters, and double-byte characters.
- They can follow any operand entry.
- In statements in which an optional operand entry is omitted but a remarks entry is desired, *the absence of the operand entry must be indicated by a comma preceded and followed by one or more blanks*, as illustrated below:

Name	Operation	Operand	Comment
	END	,	REMARKS

**Statement Example:** The following example illustrates the use of name, operation, operand, and remarks entries. A compare instruction has been named by the symbol COMP; 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 "new sum" is being compared to "old" with this instruction.

Name	Operation	Operand	Comment
COMP	CR	5,6	NEW SUM TO OLD

## Character Set

Terms, expressions, and character strings used to build source statements are written with the following characters:

Alphabetic characters	A through Z, and \$, #, @
Digits	0 through 9
Special characters	+ - , = . ^ ( ) ' / & blank
Underscore character	_

Figure 3. Character Set

Examples showing the use of the above characters are given in Figure 5 on page 19.

The term “alphanumeric characters” includes both alphabetic characters and digits, but not special characters, the underscore or double-byte data. Normally, you would use strings of alphanumeric characters to represent data (see “Terms” on page 25), and special characters as:

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

These characters are represented by the card-punch combinations and internal bit configurations listed in the *IBM System/370 Reference Summary*. In addition, any of the 256 punch combinations may be designated anywhere that characters can appear between paired single quotation marks, in comments, and in macro instruction operands.

The **double-byte character set (DBCS)** comprises the following:

Double-byte blank	XX'4040'
Double-byte characters	Each double-byte character contains two bytes, each of which must be in the range XX'41' to XX'FE'.
Shift codes	SO ("shift out," XX'0E') SI ("shift in," XX'0F')

Figure 4. Double-Byte Character Set (DBCS)

**Notes:**

1. SO and SI delimit DBCS data only when the assembler is invoked with the DBCS option.
2. When the assembler is invoked with the DBCS option, double-byte characters may be used anywhere that EBCDIC characters enclosed by single quotation marks can be used.
3. Regardless of the invocation option, double-byte characters may be used in remarks, comments, and the operands of AREAD and REPRO statements.

---

## Assembler Language Structure

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

A source statement is composed of:

- A name entry (usually optional) that is a symbol
- An operation entry (required) that is a symbolic operation code representing a machine, assembler, or macro instruction
- An operand entry (usually required) that is composed of one or more operands
- A remarks entry (optional)

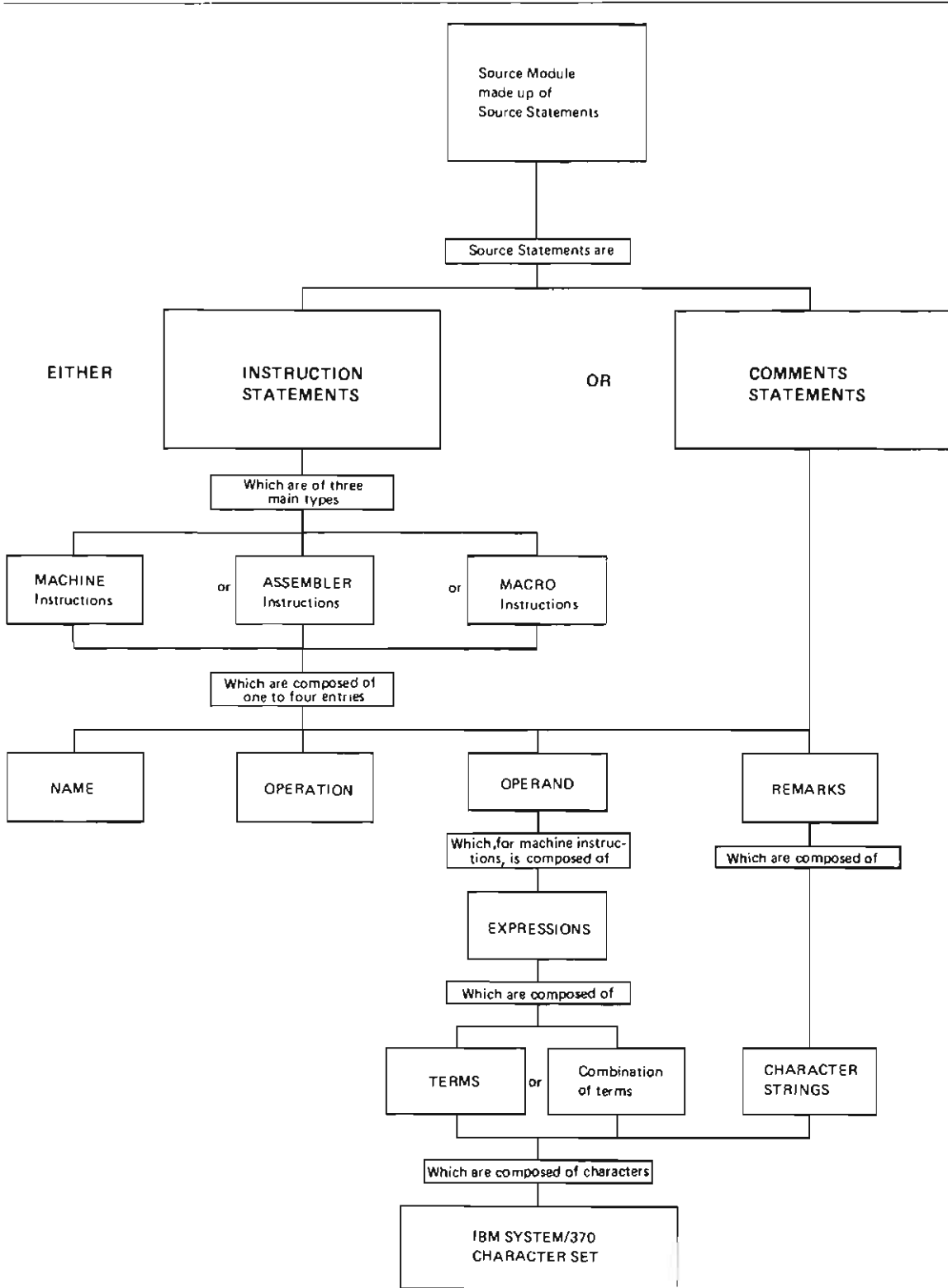
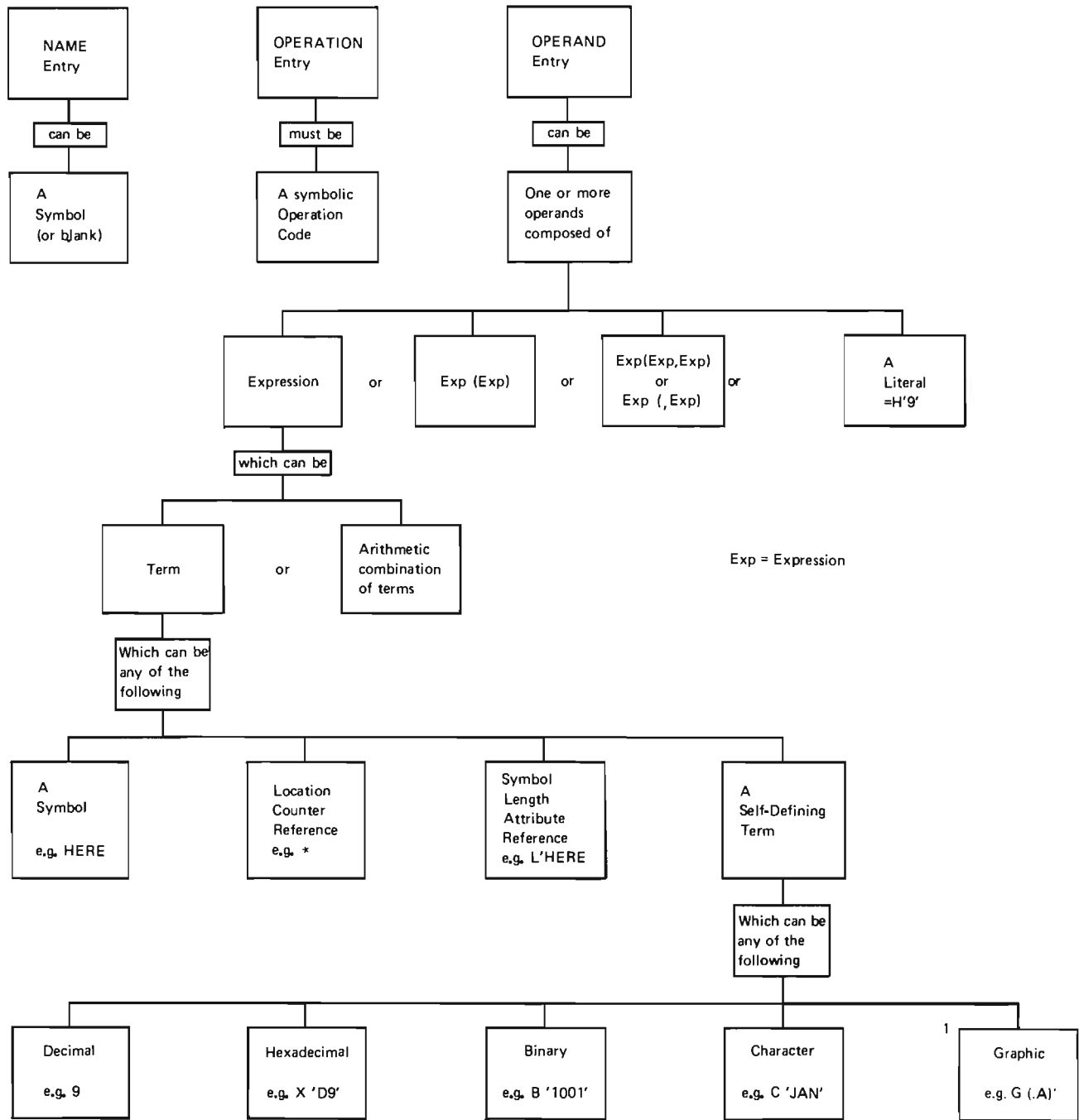


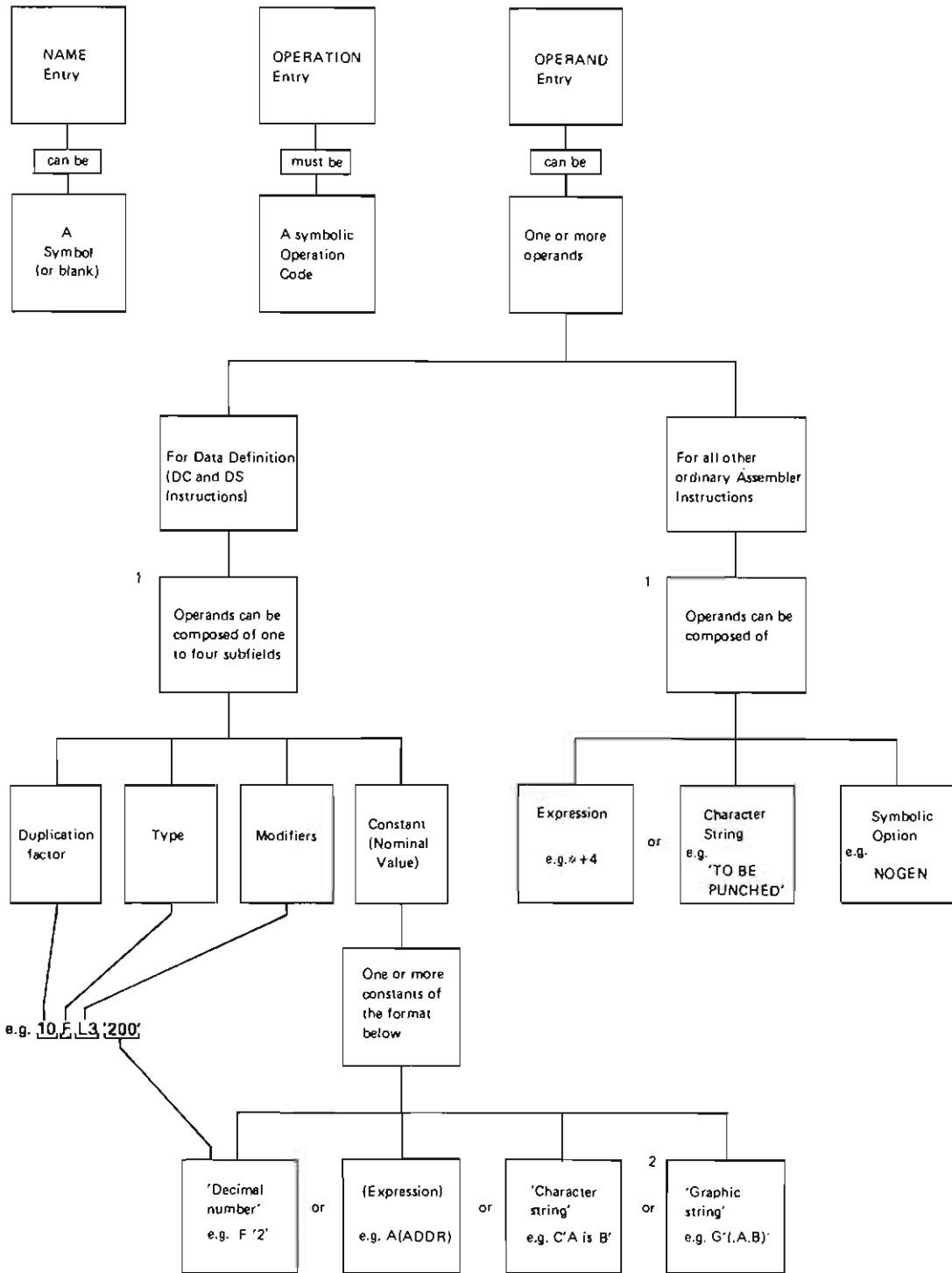
Figure 6. Assembler Language Structure





<sup>1</sup> With DBCS option only

Figure 7. Machine Instructions



<sup>1</sup> Discussed more fully where individual instructions are described

<sup>2</sup> With DBCS option only

Figure 8. Ordinary Assembler Instruction Statements

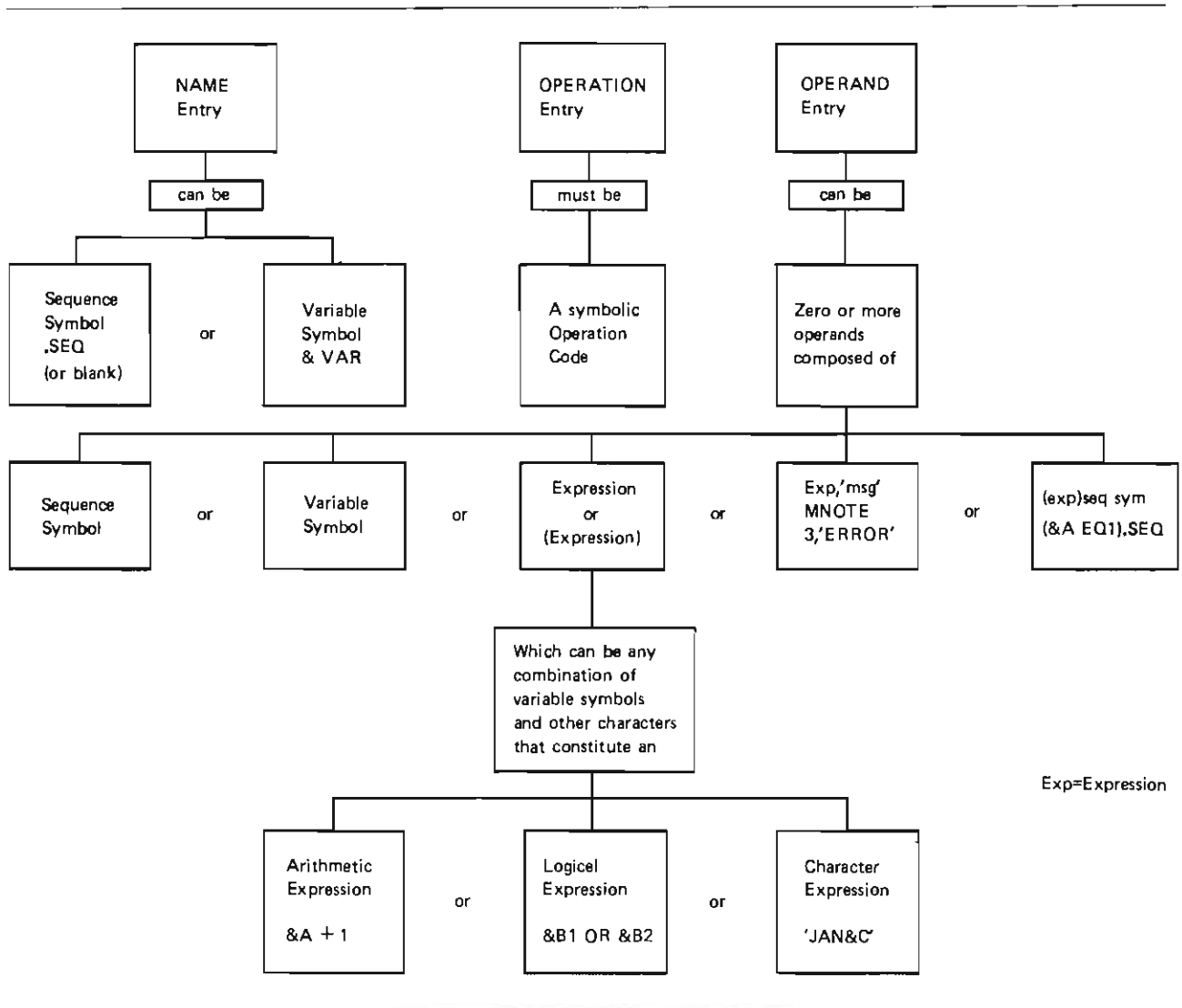


Figure 9. Conditional Assembly Instructions

Macro instruction statements are described in Figure 10 on page 24 and discussed in "Part 2. Macro Language."

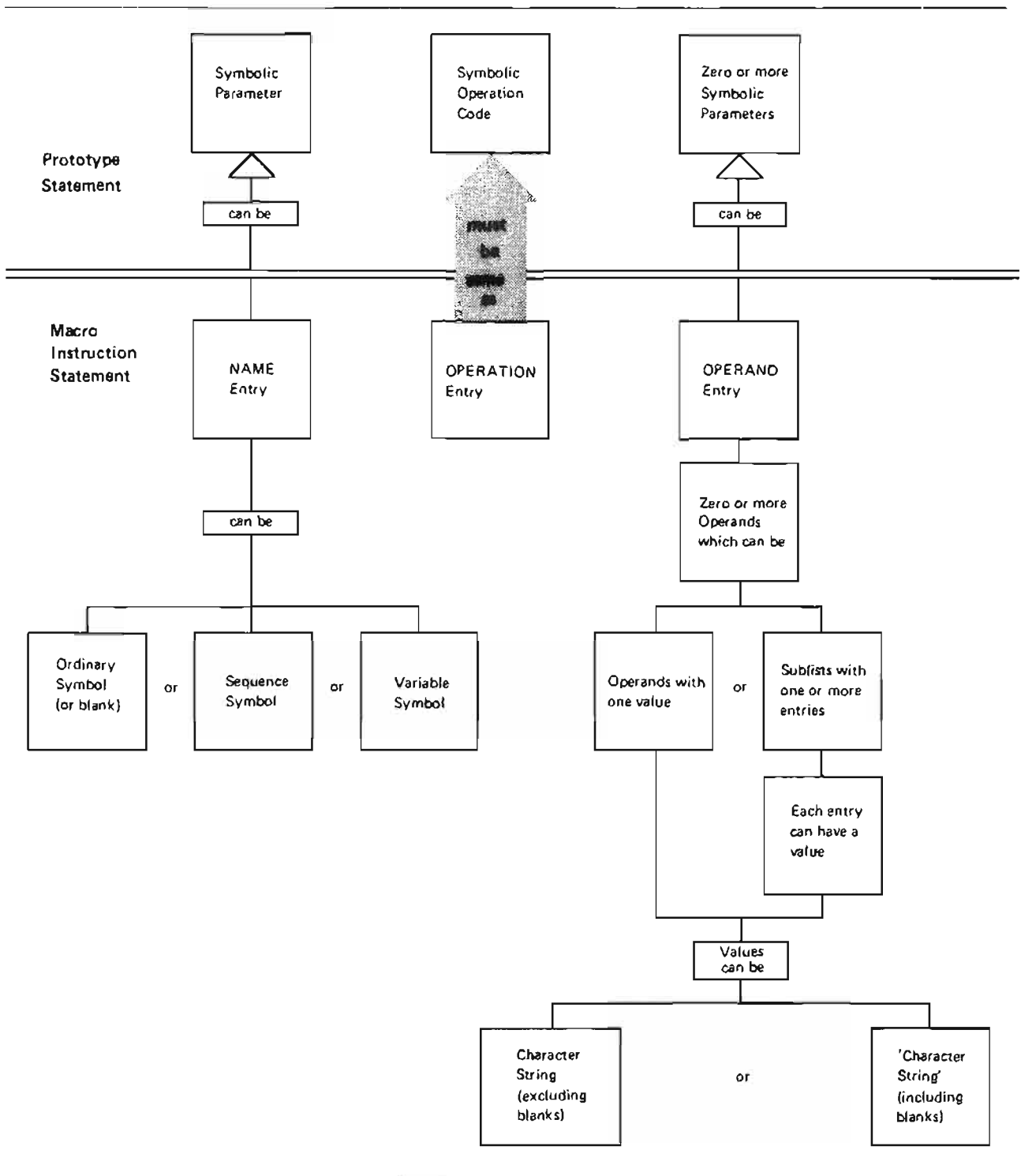


Figure 10. Macro Instructions

---

## Terms and Expressions

### 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, and is relocatable if its value changes upon relocation. Figure 11 summarizes the various types of terms. The following text discusses each term and the rules for its use.

Terms	Term can be absolute	Term can be relocatable	Value is assigned by assembler	Value is inherent in term
Symbols	X	X	X	
Location counter reference		X	X	
Symbol length attribute	X		X	
Other data attributes	X		X	
Self-defining terms	X			X

Figure 11. Summary of Terms

### 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 allows you to use this symbol in instruction operands to represent registers, displacements in explicit addresses, immediate data, lengths, and implicit addresses with absolute values. For details of these program elements, see "Operand Entries" on page 78.

The advantages of symbolic over numeric representation are:

1. Symbols are easier to remember and use than numeric values, thus reducing programming errors and increasing programming efficiency.
2. 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.

3. You can change the value of one symbol (through an EQU instruction) more easily than you can change several numeric values in many instructions.
4. Symbols are entered into a cross-reference table that the assembler prints in the program listing. This table helps you to find a symbol in a program listing, because it lists (a) the number of the statement in which the symbol is defined, that is, used as the name entry, and (b) the numbers of all the statements in which the symbol is used in the operands.

**Symbol Table:** The assembler maintains an internal table called a 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" and "Self-Defining Terms" in this chapter for more details. The three types of symbols recognized by the assembler are:

- Ordinary symbols
- Variable symbols
- Sequence symbols

*Ordinary symbols* can be used in the name and operand fields of machine and assembler instruction statements. They must be coded to conform to these rules:

- The symbol must not consist of more than 63 alphanumeric characters. The first character must be an alphabetic character (A through Z, \$, #, or @). The other characters may be alphabetic characters, digits, or a combination of the two.
- No special characters may be included in an ordinary symbol.
- No blanks are allowed in an ordinary symbol.
- No double-byte data is allowed in an ordinary symbol.
- An underscore character is allowed, with the restrictions listed below.

An underscore character must *not* appear in an external symbol, or in the name field of an OPSYN instruction. The following lists the symbol fields in which the underscore character must *not* appear:

- In the name field of:
  - a CSECT instruction
  - a DXD instruction
  - a COM instruction
  - an OPSYN instruction
- In the operand field of:
  - an EXTRN instruction
  - a WXTRN instruction
  - an ENTRY instruction
- As the nominal value in a V-type or Q-type address constant

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

The following are valid symbols:

ORDSYM#435A	HERE	\$OPEN
K4	#0123	X
B49467LITTLENAIL	@33	SAVE_TOTAL

*Variable symbols* must begin with an & followed by an alphabetic character and, optionally, up to 61 alphanumeric characters (including underscore). Variable symbols can only be used in macro processing and conditional assembly instructions. They allow different values to be assigned to one symbol. A complete discussion of variable symbols appears in "Chapter 7. How to Prepare Macro Definitions."

The following are valid symbols:

&VARYINGSYMBOL	&@ME
&F346944	&A
&EASY_TO_READ	

*Sequence symbols* consist of a period (.) followed by an alphabetic character, and up to 61 additional alphanumeric characters. Sequence symbols can be used only in macro processing and conditional assembly instructions. They are used to indicate the position of statements within the source program or macro definition. Through their use, you can 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.")

The following are valid symbols:

.BLABEL04	.#359
.BRANCHTOMEFIRST	.A

**Symbol Definition:** An ordinary symbol is considered defined when it appears as:

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

**Note:** Ordinary symbols that appear in instructions generated from model statements at preassembly time are also considered defined.

In Figure 12 on page 28, the assembler assigns a value to the ordinary symbol in the name fields as follows:

1. According to the address of the leftmost byte of the storage field that contains one of the following:
  - a. (See (1) in Figure 12) Any machine or assembler instruction (except the EQU or OPSYN instruction)
  - b. (See (2) in Figure 12) A storage area defined by the DS instruction
  - c. (See (3) in Figure 12) Any constant defined by the DC instruction
  - d. A channel command word defined by the CCW, CCW0, or CCW1 instruction

The address value thus assigned is relocatable, because the object code assembled from these items is relocatable; the relocatability of addresses is described "Addresses" on page 80.

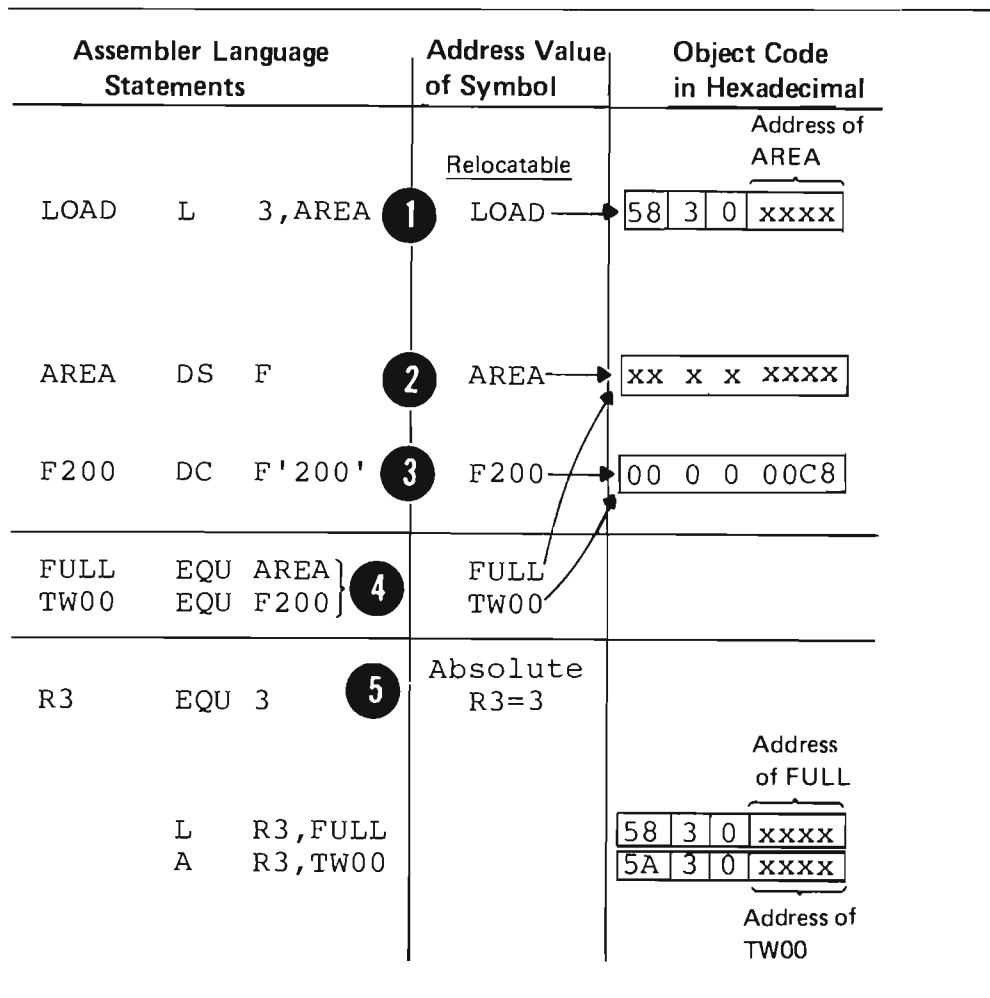


Figure 12. Transition from Assembler Language Statement to Object Code

2. According to 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 12) or *absolute* (see (5) in Figure 12) value, which is then assigned to the ordinary symbol.

The value of an ordinary symbol must lie in the range  $-2^{31}$  through  $+2^{31}-1$ .

**Restrictions on Symbols:** A symbol must be defined only once in a source module with one or more control sections, with the following exception: The symbol in the name field of a LOCTR instruction can be the same as the name of a previous START, CSECT, DSECT, COM, or LOCTR instruction. It identifies the resumption of the location counter specified by the name field.

**Note:** The ordinary symbol that appears in the name field of an OPSYN or a TITLE instruction does not constitute a definition of that symbol. It can, therefore, be used in the name field of any other statement in a source module.

**Previously Defined Symbols:** If ordinary symbols appear in operand expressions of ORG and CNOP instructions, in modifier expressions of DC, DS, and DXD statements, in the first operand of EQU statement, or in Q-type constants, they do not need to be previously defined.



Allowing forward reference in the above statement types creates two new kinds of errors that you should guard against.

- 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
```

Statement A 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 of A.

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

## Self-Defining Terms

A self-defining term allows you to specify a value explicitly. With self-defining terms, you can specify decimal, binary, hexadecimal, or character data. If the DBCS assembler option is invoked, 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. Some examples of self-defining terms and the binary values they represent are given below:

Self-Defining Term	Decimal Value	Binary Value
15	15	1111
241	241	11110001
B'1111'	15	1111
B'11110001'	241	11110001
B'100000001'	257	100000001
X'F'	15	1111
X'F1'	241	11110001
X'101'	257	100000001
C'1'	241	11110001
C'A'	193	11000001
C'AB'	49,602	1100000111000010
G'<.A>'	17,089	100001011000001

The assembler carries the values represented by self-defining terms to 4 bytes or 32 bits; the high-order bit 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 a self-defining term is used in a machine instruction statement, its value 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-justified; truncation or padding with zeros, if necessary, occurs on the left.

*Decimal Self-Defining Term:* A decimal self-defining term is simply an unsigned decimal number written as a sequence of decimal digits. High-order zeros may 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 should have a value between 0 and 15; one that represents an address should not exceed the size of storage. In any case, a decimal term may not consist of more than 10 digits, or exceed  $2^{31}-1$ . 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 1 to 8 hexadecimal digits enclosed in single quotation marks and preceded by the letter X; for example, X'C49'.

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

**Note:** 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 single quotation marks and preceded by the letter B; for example, B'10001101'. This term would appear in storage as shown, occupying 1 byte. A binary term may have up to 32 bits represented. This allows a range of values from -2 147 483 648 through 2 147 483 647.

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

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

The following illustrates 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 single quotation marks, and must be preceded by the letter C. All letters, decimal digits, and special characters may be used in a character term. In addition, any of the remainder of the 256 punch combinations may be designated in a character self-defining term. Examples of character self-defining terms are:

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

Because of the use of single quotation marks 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 term.

For each single quotation mark or ampersand desired in a character self-defining term, two single quotation marks or ampersands must be written. For example, the character value A'# would be written as 'A''#, while a single quotation mark followed by a blank and another single quotation mark would be written as '' ''.

Each character in the character sequence is assembled as its 8-bit code equivalent. The two single quotation marks or ampersands that must be used to represent a single quotation mark or ampersand within the character sequence are assembled as a single quotation mark or ampersand. Double-byte data may appear in a character self-defining term, if the assembler is invoked with the DBCS option. 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.

*Graphic Self-Defining Term:* If the assembler is invoked with the DBCS option, a graphic self-defining term can be specified. A graphic self-defining term consists of 1 to 2 double-byte characters delimited by SO and SI, enclosed in single quotation marks and preceded by the letter G. Any valid double-byte characters may 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 may be present between two double-byte characters. However, if SO and SI without an intervening double-byte character are used, error IEV148 "SELF-DEFINING TERM LACKS ENDING QUOTE OR HAS BAD CHARACTER" will be issued.

## Location Counter Reference

The assembler runs a location counter to assign storage addresses to your program statements. It is the assembler's equivalent of the 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 storage. 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 indicate the proper 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*.

These rules are illustrated below:

Location in Hexadecimal		Source Statements
000004	DONE	DC CL3'ABC'
000007	BEFORE	EQU *
000008	OURING	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 51.

The assembler carries an internal location counter value as a 4-byte (32-bit) value, but it only uses the low-order 3 bytes, which are printed in the program listings. However, if you specify addresses greater than  $2^{24}-1$ , you cause overflow into the high-order byte, and the assembler issues the error message, "LOCATION COUNTER OVERFLOW."

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. Addressing, Program Sectioning, and Linking" and "Chapter 5. Assembler Instruction Statements," respectively. The counter affected by either of these assembler instructions is the counter for the control section in which they appear.

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 can be specified as a relocatable term according to the following rules:

1. The asterisk can be specified only in the operands of:
  - Machine instructions
  - DC and DS instructions
  - EQU, ORG, and USING instructions
2. It can also be specified in *literal constants*. See “Literals” on page 36. For example:

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

The value of the location counter reference (\*) is 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 address value of \* is the address of HERE.

For the value of the asterisk in address constants with duplication factors, see “Address Constants—A and Y” on page 112.

### Symbol Length Attribute Reference

The length attribute of a symbol may 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 will be substituted for the term. When you specify a symbol length attribute reference, you obtain the length of the instruction or data referred to by a symbol. You can use this reference as a term in instruction operands to:

- Specify unknown 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:

1. The format must be L' immediately followed by a valid symbol or the location counter reference (\*).
2. The symbol must be defined in the same source module in which the symbol length attribute reference is specified.
3. 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 Figure 13), it assigns either 2, 4, or 6, depending on the format of the instruction.
- For the DC and DS instructions (see (2) in Figure 13), it assigns either the implicit or explicitly specified length. The length attribute is not affected by a duplication factor.
- For the EQU instruction, it assigns the length attribute value of the leftmost or only term (see (3) in Figure 13) 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 Figure 13)
- Location counter reference (see (5) in Figure 13)
- L'\* (see (6) in Figure 13)

The length attribute of the location counter reference (L'\*—see (7) in Figure 13) is equal to the length attribute of the instruction in which the L'\* appears.

Figure 13 illustrates these rules.

Source Module			Value of Symbol Length Attribute At Assembly Time	
HACHA	HVC	TD, FROM	L'HACHA	6 <sup>1</sup>
HACHB	L	3, ADCON	L'HACHB	4 <sup>1</sup>
HACHC	LR	3, 4	L'HACHC	2 <sup>1</sup>
TD	DS	CL80	L'TD	80 <sup>2</sup>
FROM	DS	CL240	L'FROM	240 <sup>2</sup>
ADCON	DC	A(OTHER)	L'ADCON	4 <sup>2</sup>
CHAR	DC	C'YUKON'	L'CHAR	5 <sup>2</sup>
DUPL	DC	3F'200'	L'DUPL	4 <sup>2</sup>
RELOC1	EQU	TD <sup>3</sup>	L'RELOC1	80
RELOC2	EQU	TD+80 <sup>3</sup>	L'RELOC2	80
ABSOL1	EQU	FROM-TD <sup>3</sup>	L'ABSOL1	240
ABSOL2	EQU	ABSOL1 <sup>3</sup>	L'ABSOL2	240
SDT1	EQU	102 <sup>3</sup>	L'SDT1	1 <sup>4</sup>
SDT2	EQU	X'FF'+A-B <sup>3</sup>	L'SDT2	1 <sup>4</sup>
SDT3	EQU	C'YUK'	L'SDT3	1 <sup>4</sup>
ASTERISK	EQU	*+10 <sup>3</sup>	L'ASTERISK	1 <sup>5</sup>
LOCTREF	EQU	L'* <sup>3</sup>	L'LOCTREF	1 <sup>6</sup>
LENGTH1	DC	A(L'*)	L'*	4 <sup>7</sup>
			L'LENGTH1	4 <sup>7</sup>
LENGTH2	HVC	TO(L'*), FROM	L'*	6 <sup>7</sup>
LENGTH3	HVC	TO(L'TD-20), FROM	L'TD	80

Figure 13. Assignment of Length Attribute Values to Symbols in Name Fields

The following example illustrates use of the L'symbol in moving a character constant into either the high-order or low-order end of a storage field. For ease in following the example, the length attributes of A1 and B2 are mentioned. However, keep in mind that the L'symbol term makes coding such as this possible in situations where lengths are unknown.

A1	DS	CL8
B2	DC	CL2'AB'
HIORD	MVC	A1(L'B2),B2
LOORD	MVC	A1+L'A1-L'82(L'B2),B2

A1 names a storage field 8 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 leftmost 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 results in the addition of the length of A1 to the beginning address of A1, and the subtraction of 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 length specification as in HIORD.

**Note:** The length attribute of the location counter reference (L'\*) is equal to the length attribute of the instruction in which the L'\* appears.

### Other Attribute References

There are other attributes that 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 the type (T'), length (L'), scaling (S'), integer (I'), count (K'), number (N'), and defined (D') attributes.

**Note:** You can refer to these attributes only in conditional assembly instructions and expressions; for full details, see "Data Attributes" on page 215.

### Terms in Parentheses

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

Arithmetically combined terms, enclosed in parentheses, may be used in combination with terms outside the parentheses, as follows:

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

When the assembler program encounters terms in parentheses in combination with other terms, it first reduces the combination of terms inside the parentheses to a single value which may 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.

Terms in parentheses may be included within a set of terms in parentheses:

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

The innermost set of terms in parentheses is evaluated first. Six levels of parentheses are allowed; a level of parentheses is a left parenthesis and its corresponding right parenthesis. Parentheses which occur as part of an operand format do not count in this limit. An arithmetic combination of terms is evaluated as described in the next section.

## Literals

You can use literals as operands in order to introduce data into your program. However, you cannot use a literal as a term in an expression. The literal represents data rather than a reference to data. 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 defining constants elsewhere in your source module and then using their symbolic names in machine instruction operands.

The assembler assembles the data specified in a literal into a “literal pool” (described below). It then assembles the address of this literal data 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 proper boundary alignment and occupies the minimum amount of space.

## Literals, Constants, and Self-Defining Terms

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

1. Where you can specify them in machine instructions, that is, whether they represent data or an address of data
2. Whether they have relocatable or absolute values
3. What is assembled into the object code of the machine instruction in which they appear

- 
1. A literal with a relocatable address:

```
      L   3,=F'33'      See note 1
      L   3,F33         Same effect as L 3,=F'33'.  See also note 2
      .
      .
      .
F33   DC   F'33'
```

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

```
      HVC FLAG,=X'00'   See note 1
      HVI FLAG,X'00'   Same effect as HVC FLAG,=X'00'.  See also note 3
      HVI FLAG,ZERO    Immediate data.  See note 4
      .
      .
      .
FLAG  DS   X
ZERO  EQU  X'00'
```

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

```
      LA  4,LOCORE     Absolute address.  See note 4
      LA  4,1000       Same effect as LA 4,LOCORE.  See also note 3
      .
      .
      .
LOCORE EQU 1000
```

---

Figure 14. Differences between Literals, Constants, and Self-Defining Terms

### Notes to Figure 14:

1. A literal represents data.
2. A constant is represented by its relocatable address.



3. A self-defining term represents data and has an absolute value.
4. A symbol with an absolute value does not represent the address of a constant, but represents either immediate data or an absolute address.

**Compare:**

A literal with a relocatable address

```

      L 3, =F'33' } same effect
      L 3, F33   }
      .
      .
F33   DC F'33'
  
```

A Literal with a self-defining term and a symbol with an absolute value

```

      MVC FLAG, =X'00' } same effect
      MVI FLAG, X'00'  }
      MVI FLAG, ZERO  }
      .
      .
FLAG  DS  X
ZERO  EQU X'00'
  
```

A symbol having an absolute address value with a self-defining term

```

      LA 4, LOCORE } same effect
      LA 4, 1000  }
      .
      .
LOCORE EQU 1000
  
```

Figure 15. Differences between Literals, Constants, and Self-Defining Terms

**Notes to Figure 15:**

1. The address of the literal, rather than the literal data itself, is assembled into the object code.
2. The address of a constant is assembled into the object code.
3. When a symbol with an absolute value represents immediate data, it is the absolute value that is assembled into the object code.
4. The absolute value of a self-defining term is assembled into the object code.

## General Rules for Literal Usage

A literal is not a term and can be specified only as a complete operand in a machine instruction. In instructions with the RX format, they must not be specified in operands in which an index register is also specified.

Because literals provide "read-only" data, they must not be used:

- In operands that represent the receiving field of an instruction that modifies storage
- In any shift or I/O instruction

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.

A literal must be coded as indicated here:

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

where the subfields are:

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

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in the operand of a DC assembler instruction. The major difference is that the literal must start with an equal sign (=), which indicates to the assembler that a literal follows (Refer to the discussion of the DC assembler instruction operand format in "Chapter 5. Assembler Instruction Statements" for the means of specifying a literal.)

The instruction below 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 will be the address at which the value F'274' is stored.

In general, literals can be used wherever a storage address is permitted as an operand. They cannot, however, be used in any assembler instruction that requires the use of a previously defined symbol. Literals are considered relocatable because the address of the literal, rather than the literal itself, will be assembled in the statement that employs a literal. The assembler generates the literals, collects them, and places them in a specific area of storage, as explained under "Literal Pool." A literal is not to be confused with the immediate data in an SI instruction. 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. The location of the literal, rather than the literal itself, is assembled in the statement employing a literal. The positioning of the literal pool can be controlled by you, if desired. 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 sequence in which literals are ordered within the pool is controlled by the assembler. Further information on positioning the literal pool(s) is in “LTOrg—Begin Literal Pool” on page 140.

## Expressions

This section discusses the expressions used in coding operand entries for source statements. You can use an expressions 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. You can use an absolute or relocatable expression in a machine instruction or any assembler instruction other than a conditional assembly instruction. The assembler evaluates relocatable and absolute expressions at assembly time.

**Note:** There are three types of expression that you can use only in conditional assembly instructions: arithmetic, logical, and character expressions. They are evaluated at preassembly time. Figure 17 on page 40 defines both absolute and relocatable expressions.

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. The following are examples of valid expressions:

---

*	BETA*10
AREA1+X'2D'	B'101'
**32	C'ABC'
N-25	29
FIELD+332	L'FIELD
FIELD	LAHBDA+GAHHA
(EXIT-ENTRY+1)+G0	TEN/TWO
ALPHA-BETA/(10+AREA*L'FIELD)-100	=F'1234'

---

Figure 16. Examples of Valid Expressions

## Rules for Coding Expressions

The rules for coding an absolute or relocatable expression are:

1. Both unary (operating on one value) and binary (operating on two values) operators are allowed in expressions.
2. An expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression.
3. An expression must not begin with a binary operator, nor can it contain two binary operators in succession.
4. An expression must not contain two terms in succession.

5. No blanks are allowed between an operator and a term, nor between two successive operators.
6. An expression can contain up to 19 unary and binary operators, and up to 6 levels of parentheses. Note that parentheses that are part of an operand specification do not count toward this limit.
7. A single relocatable term is not allowed in a multiply or divide operation. Note that paired relocatable terms have absolute values and can be multiplied and divided if they are enclosed in parentheses.
8. A literal is not a valid term and is therefore not allowed in an expression.

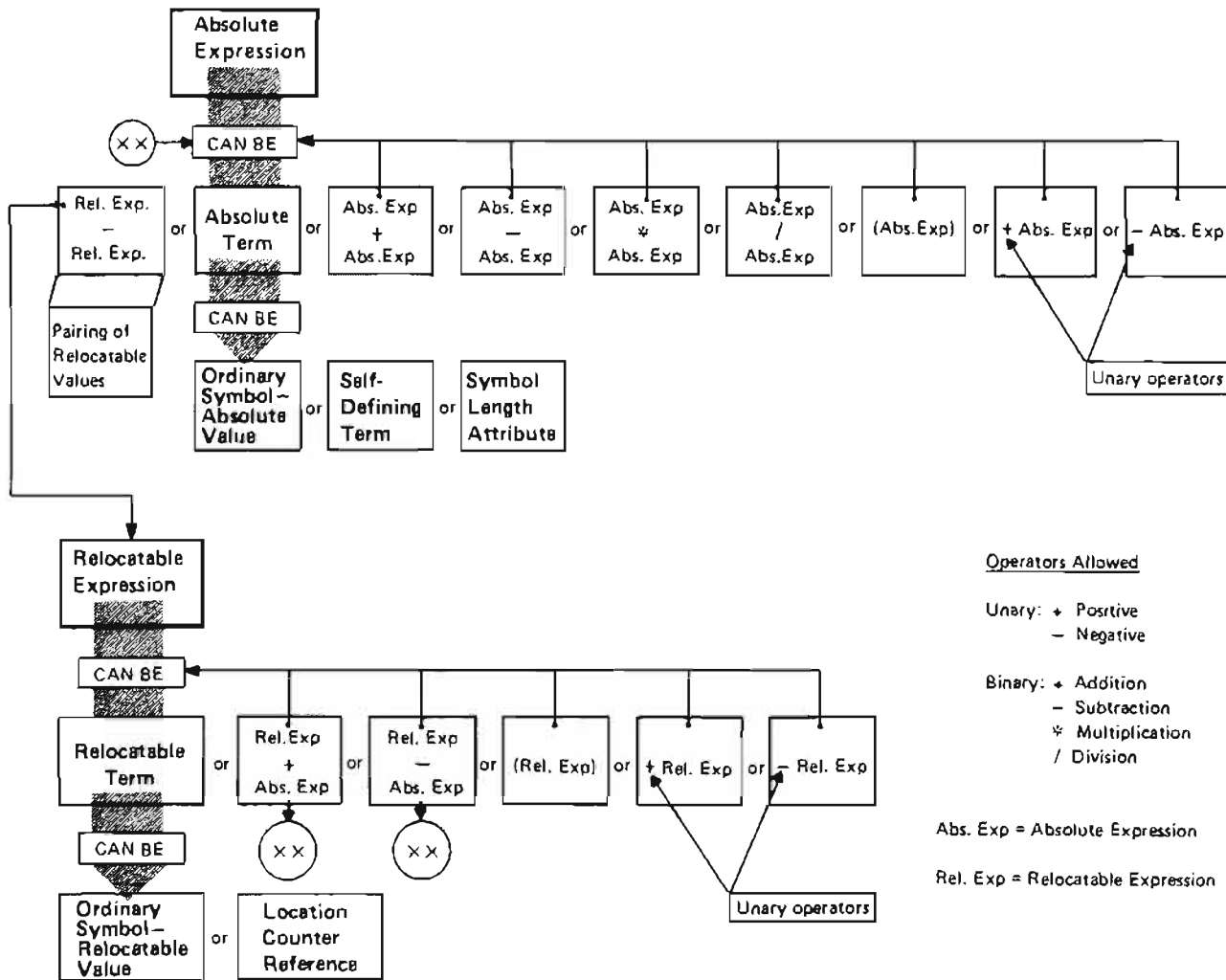


Figure 17. Definitions of Absolute and Relocatable Expressions

## Evaluation of Expressions

A single-term expression, like 29 or BETA, takes on the value of the term involved. A multiterm expression, like  $25*10 + A/B$  or  $BETA + 10$ , is reduced to a single value, as follows:

1. It evaluates each term.
2. It performs arithmetic operations from left to right. However,
  - a. It performs unary operations before binary operations.
  - b. It performs 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  $+2^{31}-1$ .

**Note:** It is assumed that 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:** The assembler reduces an absolute expression 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.
2. Contains relocatable terms alone or in combination with absolute terms, and if all these relocatable terms are paired.

**Paired Relocatable Terms:** An expression can be absolute even though it contains relocatable terms, provided that 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 be defined in the same control section of a source module (that is, 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 value represented by the paired terms is absolute.

The following examples illustrate 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
*-Y1
```

<sup>1</sup> A reference to the location counter must be paired with another relocatable term from the same control section; that is, with the same relocatability.

**Relocatable Expression:** A relocatable expression is one whose value changes by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage.

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. Note that 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 been resolved.

The following examples illustrate 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

Complex relocatable expressions can be used only in A-type and Y-type address constants to generate an address constant value (For more detail, see "A-Type and Y-Type Address Constants" in "Chapter 5. Assembler Instruction Statements.") V-type and S-type constants may not contain complex relocatable expressions.

---

## Chapter 3. Addressing, Program Sectioning, and Linking

---

### Addressing

This part of the chapter describes the techniques and instructions that allow you to use symbolic addresses when referring to data. You can address data that is defined within the same source module, or 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. Also, the assembler can convert the symbolic addresses you specify into their object code form.

#### Addressing within Source Modules: Establishing Addressability

By establishing the addressability of a control section, you can refer to the symbolic addresses defined in it in the operands of machine instructions. This is much easier than coding the addresses in the base-displacement form required by the System/370. The symbolic addresses you code in the instruction operands are called *implicit addresses*, and the addresses in the base-displacement form are called *explicit addresses*.

The assembler will convert these implicit addresses for you into the explicit addresses required for the assembled object code of the machine instruction. However, you must supply the assembler with:

1. A base address from which it can compute displacements to the addresses within a control section
2. A base register to hold this base address

#### How to Establish Addressability

To establish the addressability of a coding section, you must, when coding:

- Specify a base address from which the assembler can compute displacements.
- Assign a base register to contain this base address.
- Write the instruction that loads the base register with the base 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, and should remain there throughout the execution of your program.

#### 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 the operand field of machine instruction statements, you must (1) indicate to the assembler, by means of a USING statement, that one or more general registers are available for use as base registers, (2) specify, by means of the USING statement, what value each base register contains, and (3) load each base register with the value you have specified for it.

Having the assembler determine base registers and displacements relieves you of the need to separate each address into a displacement value and a base address value. This feature of the assembler will eliminate a likely source of programming errors, thus reducing the time required to check out programs. You use the USING and DROP instructions described in this chapter to take advantage of this feature. The principal discussion of this feature follows the description of both instructions.

### USING—Use Base Address Register

The USING instruction allows you to specify a base address and assign one or more base registers. If you also load the base register with the base address, you have established addressability in a control section.

To use the USING instruction correctly, you should know:

1. Which locations in a control section are made addressable by the USING instruction
2. Where in a source module you can use these established addresses as implicit addresses in instruction operands

#### Format of USING:

Name	Operation	Operand
A sequence symbol or blank	USING	BASE,BASEREG1[,BASEREG2]...

The operand, BASE, specifies a base address, which can be a relocatable or an absolute expression. The value of the expression must lie between  $-2^{24}$  and  $2^{24}-1$ .

The remaining operands specify from 1 to 16 base registers. The operands must be absolute expressions whose values lie in the range 0 through 15.

The assembler assumes that the first base register (BASEREG1) contains the base address BASE at execution time. If present, the subsequent operands, BASEREG2, BASEREG3,..., represent registers that the assembler assumes will contain the address values,  $BASE + 4096$ ,  $BASE + 8192$ ,..., respectively.

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 it may assume that the current value of the location counter will be in general register 12 at object time, and that the current value of the location counter, incremented by 4096, will be in general register 13 at object time.

If you change the value in a base register currently being used, and wish the assembler to compute displacement from this value, you must tell the assembler the new value by means of 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
```

If you must refer to the first 4096 bytes of storage, general register 0 can be used as a base register, subject to the following conditions:

- The value of operand BASE must be either absolute or relocatable zero or simply relocatable.
- Register 0 must be specified as BASEREG1.

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, etc.

**Note:** If register 0 is used as a base register, the program is not relocatable, despite the fact that operand BASE may be relocatable. The program 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 a USING Instruction:** The range of a USING instruction (called the USING range) is the 4096 bytes beginning at the base address specified in the USING instruction. Addresses that lie within the USING range can be converted from their implicit to their explicit form; 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.

**Note:** 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 sum of the USING ranges that would apply if the base registers were specified in separate USING instructions.

**Domain of a USING Instruction:** The domain of a USING instruction (called the USING domain) begins where the USING instruction appears in a source module and continues to the end of the source module. (Exceptions are discussed later, under "Notes about the USING Domain ") The assembler converts implicit address references into their explicit form:

- If the address reference appears in the domain of a USING instruction, and
- If the addresses referred to lie within the range of the same USING instruction.

The assembler does not convert 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 performed.

**How to Use the USING Instruction:** You should specify your USING instruction so that:

- All the addresses in each control section lie within a USING range.
- All the references for these addresses lie within the corresponding USING domain.

You should, therefore, place all USING instructions at the beginning of the source module 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, you specify a base address and assign a base register in the USING instruction. At execution time, the base register is loaded with the correct base address.

If a control section is longer than 4096 bytes, you must assign more than one base register. This allows you to establish 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, you should 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 formatted by the dummy section.

**Note:** The assembler assumes that you are referring to the symbolic addresses of the dummy section, and it computes displacements accordingly. However, at execution time, the assembled addresses refer to the location of real data in the storage area.

**Notes about the USING Domain:** The domain of a USING instruction continues until the end of a source module, except when:

- A subsequent DROP instruction specifies the same base register or registers assigned by the preceding USING instruction.
- A subsequent USING instruction specifies the same register or registers assigned by the preceding USING instruction.

**Notes about the USING Range:** 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 first USING domain is terminated after the second USING instruction.

Two USING ranges overlap when the base address of one USING instruction lies within the range of another USING instruction. When two ranges overlap, the assembler computes displacements from the base address that gives the smallest 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.

**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 whole of a source module.

For absolute implicit addresses greater than 4095, 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.

## DROP—Drop Base Register

You can use the DROP instruction to indicate to the assembler that one or more registers are no longer available as base registers. This allows you:

- To free base registers for other programming purposes
- To ensure that the assembler uses the base register you wish in a particular coding situation; for example, when two USING ranges overlap or coincide

### Format of DROP:

---

Name	Operation	Operand
A sequence symbol or blank	DROP	BASEREG1[,BASEREG2]... or blank

---

Up to 16 operands can be specified. They must be absolute expressions whose values represent the general registers 0 through 15. The expressions in the operand indicate general registers previously named in a USING statement that are now unavailable for base addressing. A DROP instruction with a blank operand field causes all currently active base registers assigned by USING instructions to be dropped.

After a DROP instruction, the assembler will not use the registers specified in a 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 following statement, for example, prevents the assembler from using registers 7 and 11:

```
DROP      7,11
```

A DROP instruction is not needed:

- If the base address is being changed by a new USING instruction, and the same base register is assigned; however, the new base address must be loaded into the base register.
- At the end of a source module.

## 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 mnemonics in the example are for 2-byte instructions in the RR format.

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

---

## Program Sectioning and Linking

This part of the chapter explains how you can subdivide a large program into smaller parts that are easier to understand and maintain. It also explains how you can divide these smaller parts into convenient sections; for example, one section to contain your executable instructions, and another section to contain your data constants and areas.

You should consider two different subdivisions when writing an assembler language program:

- The source module
- The control section

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 into load modules to form an executable program.

You can also divide a source module into two or more control sections. Each control section is assembled as part of an object module. By writing the proper link-edit control statements, you can select a complete object module or any individual control section of the object module to be link-edited and later loaded as an executable program.

*Size of Program Parts:* If a source module becomes so large that its logic is not easily understood, divide it into smaller modules.

Unless you have special programming reasons, you should write each control section so that the resulting object code is not larger than 4096 bytes. This is the largest number of bytes that can be covered by one base register.

*Communication between Program Parts:* You must be able to communicate between the parts of your program; that is, be able to refer to data in a different part or be able to branch to another part.

To communicate between two or more source modules, you must symbolically link them together.

To communicate between two or more control sections within a source module, you must establish the addressability of each control properly from one section to another regardless of the relative section.

## 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:

1. You write them on a coding form and then enter them as input through a terminal or, using punched cards, through a card reader.
2. You specify one or more COPY instructions among the source statements being entered. When the 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—Copy Predefined Source Coding" on page 144 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, described in this manual. 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 Control Section" on page 53 for more details.

### End of a Source Module

The END instruction usually marks the end of a source module. However, you can code several END instructions. The assembler stops assembling when it processes the first END instruction. If no END instruction is found, the assembler will generate one. See "END—End Assembly" on page 145 for more details.

**Note:** Conditional assembly processing can determine which of several substituted END instructions is to be processed.

## Control Sections

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.

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

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

*At assembly time:* The assembler translates the source statements in the control section into object code. Each source module is assembled into one object module. The entire object module and each of the control sections it contains are relocatable.

*At link-editing time:* According to linkage editor control statements, the linkage editor combines the object code of one or more control sections into one load module. It also calculates the linkage addresses necessary for communication between two or more control sections from different object modules. In addition, it calculates the space needed to accommodate external dummy sections.

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

*At execution time:* The control program passes control to the load module now in virtual storage, and your program is executed.

**Note:** You can specify the relocatable address of the starting point for program execution in a link-edit control statement or in the operand field of an END statement.

## Executable Control Sections

An executable control section is one you initiate by using the START or CSECT instruction, and is assembled into object code. At execution time, an executable control section contains the binary data assembled from your coded instructions and constants, and is, therefore, executable.

An executable control section can also be initiated as "private code," without using the START or CSECT instruction.

## Reference Control Sections

A reference control section is one you initiate by using the DSECT, COM, or DXD instruction, and is 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 to be empty at assembly time, and the actual binary data to which they refer is not entered until execution time.

## Location Counter Setting

The assembler maintains a separate location counter for each control section. The location counter setting for each control section starts at 0. The location values assigned to the instructions and other data in a control section are, therefore, relative to the location counter setting at the beginning of that control section.

However, for executable control sections, the location values that appear in the listings do not restart at 0 for each subsequent executable control section. They carry on from the end of the previous control section. Your executable control sections are usually loaded into storage in the order in which you write them. You can, therefore, match the source statements and object code produced from them with the contents of a dump of your program.

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

You can continue a control section that has been discontinued by another control section, and, thereby, intersperse code sequences from different control sections. Note that the location values that appear in the listings for a control section, divided into segments, follow from the end of one segment to the beginning of the subsequent segment.

The location values, listed for the next control section defined, begin after the last location value assigned to the preceding control section. The length counter for a CSECT is incremented until it reaches its maximum capacity ( $2^{24} - 1$ , or FFFFFFF hexadecimal bytes). The counter is then locked and remains at that value for the CSECT. No error condition or message is issued by Assembler H Version 2 when the length counter exceeds the hexadecimal value FFFFFFF. However, when the CSECT location counter exceeds hexadecimal FFFFFFF, you will receive messages about other error conditions, as noted below.

**Note:** Message IEV039 is issued for a CSECT location counter that exceeds  $2^{24} - 1$ , or FFFFFFF hexadecimal bytes. Message IEV067 is issued for a repeat factor on a DS or DC statement that exceeds  $2^{24} - 1$ , or FFFFFFF hexadecimal bytes.

## Use of Multiple Location Counters

Assembler H allows you to use multiple location counters for each individual control section. You use the LOCTR instruction (whose format and specifications are described below) 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 will be assembled together, then the coding using the second location counter will be assembled together, and so forth.

A practical use of multiple location counters is illustrated in Figure 18 on page 52. There, executable instructions and data areas have been interspersed throughout the coding in their logical sequence, each group of instructions preceded by a LOCTR instruction identifying the location counter under which it is to be assembled. The assembler will rearrange the control section so that the executable instructions are grouped together and the data areas together.

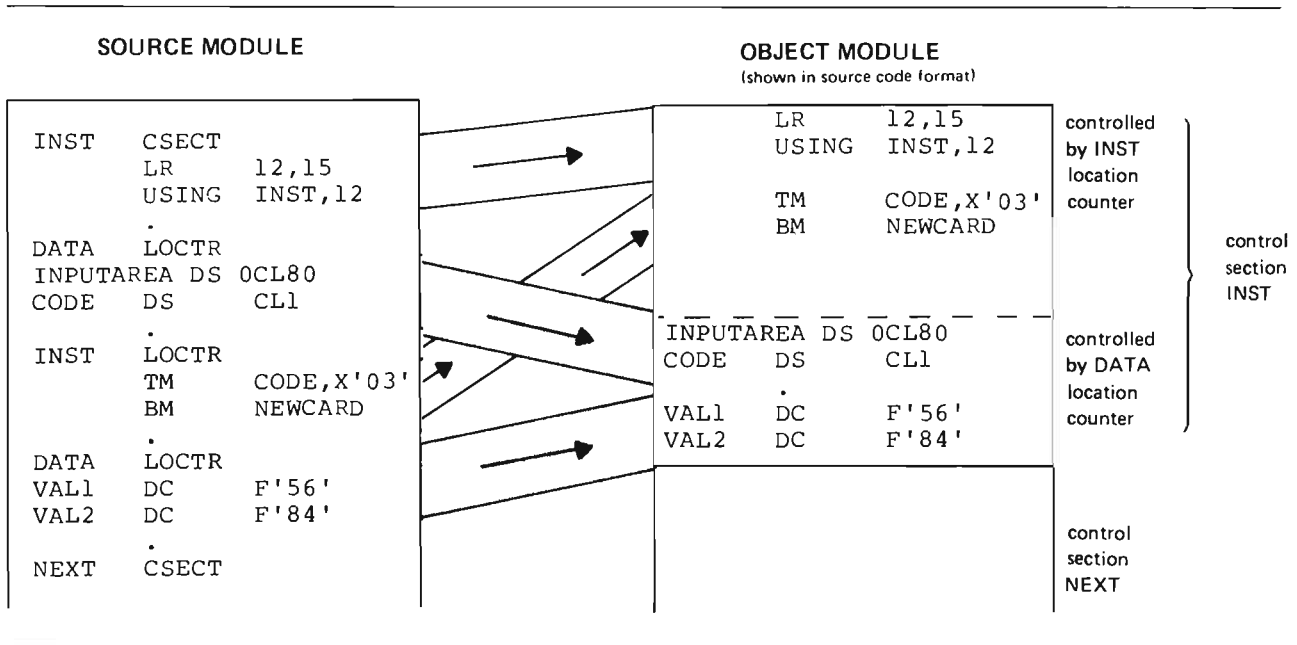


Figure 18. Use of Multiple Location Counters

### LOCTR—Multiple Location Counters

The LOCTR instruction allows you to specify 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.

#### Format of LOCTR:

Name	Operation	Operand
A variable or ordinary symbol	LOCTR	blank

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.



Name	Operation	Operand	Comment
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		

#### Notes:

1. The first location counter of a control section is defined by the name of the START, CSECT, DSECT, or COM instruction defining the section.
2. The LOCTR instruction defines a location counter.
3. The LOCTR resumes a previously defined location counter. A location counter remains in use until it is interrupted by a LOCTR, CSECT, DSECT, or COM instruction.
4. A LOCTR instruction with the same name as a control section resumes the first location counter of that section.
5. A LOCTR instruction with the same name as a LOCTR instruction in a previous control section causes that control section to be resumed using the location counter specified.

A control section cannot have the same name as a previous LOCTR instruction. A LOCTR instruction placed before the first control section definition will initiate 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 running under a location counter other than the first location counter of a control section will be assembled starting at the next available byte after the previous segment.

## First Control Section

The specifications below apply to the first executable control section, and not to a reference control section.

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

CCW, CCW0, and CCW1	DC	LTORG
CNOP	DROP	ORG
(COPY)	DS	START
CSECT	END	USING
CXD	EQU	

**Notes:**

1. These instructions are always considered a part of the control section in which they appear
2. The statements copied into a source module by a COPY instruction determine whether it will initiate the first control section.
3. The DSECT, COM, and DXD instructions initiate reference control sections and do not establish the first executable control section.

*What must come before the first control section:* The following instructions or macro definitions, if specified, belong to a source module, but must appear before the first control section.

- The ICTL instruction, which, if specified, must be the first statement in a source module
- The OPSYN instruction
- Any source macro definitions
- The COPY instruction, if the code to be copied contains only OPSYN instructions or complete macro definitions

*What can optionally come before the first control section:* The instructions or groups of instructions that can optionally be specified before the first control section are listed below:

- The following assembler instructions:

COPY	EXTRN	REPRO
DXD	ISEQ	SPACE
EJECT	PRINT	TITLE
ENTRY	PUNCH	WXTRN

- Comments statements
- Common control sections
- Dummy control sections
- External dummy control sections
- Any conditional assembly instruction
- Macro instructions

**Notes:**

1. The above instructions or groups of instructions belong to a source module, but are not considered as part of an executable control section.
2. Any instructions copied by a COPY instruction, or generated by the processing of a macro instruction before the first control section, must belong exclusively to one of the groups of instructions shown above.

3. The EJECT, ISEQ, OPSYN, PRINT, SPACE, or TITLE instructions and comments statements must follow the ICTL instruction, if specified.
4. All the instructions or groups of instructions listed above can also appear as part of a control section.

## Unnamed Control Section

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

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

The unnamed control section is sometimes referred to as private code.

All control sections ought to be provided with names so that they can be referred to symbolically:

- Within a source module
- In EXTRN and WXTRN instructions and linkage editor control statements for linkage between source modules

### Notes:

1. Unnamed common control sections or dummy control sections can be defined if the name entry is omitted from a COM or DSECT instruction.
2. If you include an AMODE or RMODE instruction in this assembly and leave the name field blank, you must provide an unnamed control section.

## Literal Pools In Control Sections

Literals, collected into pools by the assembler, are assembled as part of the executable control section to which the pools belong. If a LTORG instruction is specified at the end of each control section, the literals specified for that section will be 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 entire source module is assembled at the end of the first control section. This literal pool appears in the listings after the END instruction.

**Note:** If any control section is divided into segments, a LTORG instruction should be specified at the end of each segment to create a separate literal pool for that segment.

## External Symbol Dictionary Entries

The assembler keeps a record of each control section and prints the following information about it in an external symbol dictionary (ESD):

- Symbolic name, if one is specified
- Type code
- Individual identification
- Starting address

Figure 19 lists the assembler instructions that define control sections and dummy control sections, or identify entry and external symbols, and tells their associated type codes. There is no limit to the number of individual control sections and external symbols that can be defined in a source module.

## Establishing Residence and Addressing Mode

You may specify the addressing mode (AMODE) and/or the residence mode (RMODE) to be associated with control sections in the object deck. These modes may be specified for the following types of control sections:

- Control section (ESD type code 00)
- Unnamed control section (ESD type code 04)
- Common control section (ESD type code 05)

The assembler will set the AMODE and/or RMODE indicators in the ESD record for each applicable control section in an assembly, for passage to the linkage editor and loader. The linkage editor and loader will ensure that control is given to programs with the right addressing mode, and that programs are loaded into the correct part of virtual storage.

Name Entry	Instruction	Code Entered into External Symbol Dictionary
Optional	START	SD (if name entry is present)
	CSECT	SD (if name entry is present)
	START	PC (if name entry is omitted)
	CSECT	PC (if name entry is omitted)
	Any instruction that initiates the unnamed control section	PC
Optional	COM	CM
Optional	DSECT	None
Mandatory	DXD	XD
	External DSECT	XD
	ENTRY	LD
	EXTRN	ER
	DC (V-type address constant)	ER
	WXTRN	WX

Figure 19. Defining CSECTs, DSECTs, and Symbols

## AMODE—Addressing Mode

The AMODE instruction allows you to specify the addressing mode to be associated with control sections in the object deck.

### Format of AMODE:

---

Name	Operation	Operand
Any symbol or blank	AMODE	24 31 ANY

---

The name field associates the addressing 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, or COM instruction in this assembly. If the name field is blank, 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.

The operand indicates which addressing mode is to be associated with the control section identified by the name field. The operand must be specified as one of the three values shown. The values cannot be replaced by expressions. The values specify the following:

- 24** specifies that a 24-bit addressing mode is to be associated with a control section.
- 31** specifies that a 31-bit addressing mode is to be associated with a control section.
- ANY** specifies that the control section is not sensitive to addressing mode.

Any field of this instruction may be generated by a macro, or by substitution in open code.

### Notes:

1. AMODE can be specified anywhere in the assembly. It does not initiate an unnamed control section.
2. An assembly can have multiple AMODE instructions; however, two AMODE instructions cannot have the same name field.
3. Specification of AMODE 24 and RMODE ANY for the same name field is invalid. All other combinations are valid.
4. AMODE or RMODE cannot be specified for an unnamed *common* control section.

5. The defaults when AMODE and RMODE are not both specified for a name field are as follows:

Specified	Defaulted
Neither	AMODE 24, RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

## RMODE—Residence Mode

The RMODE instruction allows you to specify the residence mode to be associated with control sections in the object deck.

### Format of RMODE:

Name	Operation	Operand
Any symbol or blank	RMODE	24 ANY

The name field 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, or COM instruction in this assembly. If the name field is blank, 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.

The operand indicates which residence mode is to be associated with the control section identified by the name field. The operand must be specified as one of the two values shown. The values cannot be replaced by expressions. The values specify the following:

- 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 megabytes.
- ANY** 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 megabytes.

Any field of this instruction may 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. Specification of AMODE 24 and RMODE ANY for the same name field is invalid. All other combinations are valid.
4. AMODE or RMODE cannot be specified for an unnamed *common* control section.
5. The defaults when AMODE and RMODE are not both specified for a name field are as follows:

Specified	Defaulted
Neither	AMODE 24, RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

**Defining a Control Section**

You must use the instructions described below to indicate to the assembler:

- Where a control section begins
- Which type of control section is being defined

**START—Start Assembly**

The START instruction can be used only to initiate the first or only control section of a source module. You should use the START instruction for this purpose, because it allows you:

- To determine exactly where the first control section is to begin; you thereby avoid the accidental initiation of the first control section by some other instruction
- To give a symbolic name to the first control section, which can then be distinguished from the other control sections listed in the external symbol dictionary
- To specify the initial setting of the location counter for the first or only control section

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, and thereby causes the first control section to be initiated.

### Format of START:

---

Name	Operation	Operand
Any symbol or blank	START	A self-defining term, an absolute expression, or blank

---

**Note:** If the operand of a START instruction is an absolute expression, any symbols referenced in it must have been previously defined.

The symbol in the name field, if specified, 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. This symbol represents the address of the first byte of the control section, and has a length attribute value of 1.

The assembler uses the value of the self-defining term or absolute 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 a doubleword boundary. Therefore, if the value specified in the operand is not divisible by 8, the assembler sets the initial value of the location counter to the next higher doubleword boundary. If the operand entry 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.

### CSECT—Identify Control Section

The CSECT instruction allows you to initiate an executable control section or indicate the continuation of an executable control section.

The CSECT instruction can be used anywhere in a source module after any source macro definitions that are specified. 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 thereby cause the first control section to be initiated.

### Format of CSECT:

---

Name	Operation	Operand
Any symbol or blank	CSECT	Not required

---



The symbol in the name field, if specified, 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. If the first control section is initiated by a START instruction, the symbol in the name field must be used to indicate any continuation of the first control section.

**Note:** A CSECT instruction with a blank name field either initiates or indicates the continuation of the unnamed control section.

The symbol in the name field represents the address of the first byte of the control section, and has a length attribute value of 1.

The beginning of a control section is aligned on a doubleword boundary. However, when an interrupted control section is resumed using the CSECT instruction, the location counter last specified in that control section will be resumed. Consider the coding in Figure 20.

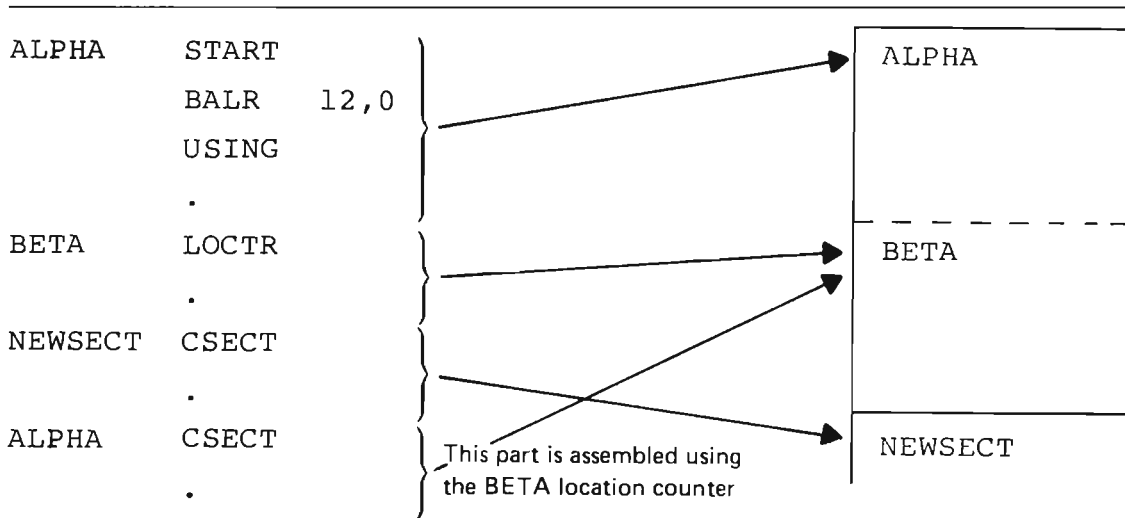


Figure 20. How the Location Counter Works

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.

**Note:** 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.

### DSECT—Identify Dummy Section

You can use the DSECT instruction to initiate a dummy control section or to indicate its continuation.

A dummy control section is a reference control section that allows you to describe the layout of data in a storage area without actually reserving any virtual storage.

You may wish to describe the format of an area whose storage location will not be determined until the program is executed. You can do so by describing the format of the area in a dummy section, and using symbols defined in the dummy section as the operands of machine instructions.

*How to use a dummy control section:* A dummy control section (dummy section) allows you to write a sequence of assembler language statements to describe the layout of unformatted 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 for it. Rather, the dummy section provides a symbolic format 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:

- Reserve a storage area for the unformatted data
- Ensure that this data is loaded into the area at execution time
- Ensure that the locations of the symbols in the dummy section actually 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 unformatted data symbolically by using the symbols defined in the dummy section.

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.

The DSECT instruction can be used anywhere in a source module after the ICTL instruction, or after any source macro definitions that may be specified.

#### **Format of DSECT:**

---

Name	Operation	Operand
Any symbol or blank	DSECT	Not required

---

The symbol in the name field, if specified, 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.

**Note:** A DSECT instruction with a blank name field either initiates or indicates the continuation of the unnamed dummy section.

The symbol in the name field represents the first location in the dummy section, and has a length attribute value of 1.

The location counter for a dummy section is always set to an initial value of 0. However, when an interrupted dummy control section is resumed using the DSECT instruction, the location counter last specified in that control section will be resumed.

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 the opposite sign.

To effect references to the storage area defined by a dummy section, do the following:

- Provide a USING statement specifying both a general register that the assembler can assign to the machine instructions as a base register and a value from the dummy section that the assembler may assume the register contains.
- Ensure that the same register is loaded with the actual address of the storage area.

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 will, at execution time, refer to storage locations relative to the address loaded into the register.

An example is shown in the following coding. Assume that two independent assemblies (assembly 1 and assembly 2) have been loaded and are to be executed as a single overall program. Assembly 1 is an input routine that places a record in a specified area of storage, places the address of the input area containing the record in general register 3, and branches to assembly 2. Assembly 2 processes the record. The coding shown in the example is from assembly 2.

The input area is described in assembly 2 by the DSECT control section named INAREA. Portions of the input area that you want to work with are named in the DSECT control section as shown. The assembler instruction USING INAREA,3 designates general register 3 as the base register to be used in addressing the DSECT control section, and that general register 3 is assumed to contain the address of INAREA.

Assembly 1, during execution, loads the actual beginning address of the input area in general register 3. Because the symbols used in the DSECT section are defined relative to the initial statement in the section, the address values they represent will, at the time of program execution, be the actual storage locations of the input area.

Name	Operation	Operand
ASMBLY2	CSECT	
BEGIN	BALR	2,0
	USING	*,2
	.	
	.	
	USING	INAREA,3
	CLI	INCODE,C'A'
	BE	ATYPE
	.	
	.	
ATYPE	MVC	WORKA,INPUTA
	MVC	WORKB,INPUTB
	.	
	.	
WORKA	DS	CL20
WORKB	DS	CL18
	.	
	.	
INAREA	DSECT	
INCODE	DS	CL1
INPUTA	DS	CL20
INPUTB	DS	CL18
	.	
	END	

## COM—Define Blank Common Control Section

You can use the COM instruction to initiate a common control section, or to indicate its continuation. One or more common sections can be defined in a source module. A common control section is a reference control section that allows you to reserve a storage area that can be used by two or more source modules.

*How to use a common control section:* A common control section (common section) allows you to 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 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 allows you to refer symbolically to the data that will be loaded at execution time. Note that you must establish the addressability of a common control section in every source module in which it is specified. If you code identical common sections in two or more source modules, you can communicate data symbolically between these modules through this common section.

**Note:** You can also code a common control section in a source module written in the FORTRAN language. This allows you to communicate between assembler language modules and FORTRAN modules.

The COM instruction identifies the beginning or continuation of a common control section.

The COM instruction can be used anywhere in a source module after the ICTL instruction, or after any source macro definitions that may be specified.

**Format of COM:**

---

Name	Operation	Operand
Any symbol or blank	COM	Not required

---

The symbol in the name field, if specified, 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.

**Note:** A COM instruction with a blank name field either initiates or indicates the continuation of the unnamed common section.

The symbol in the name field represents the address of the first byte in the common section, and has a length attribute value of 1.

The location counter for a common section is always set to an initial value of 0. However, when an interrupted common control section is resumed using the COM instruction, the location counter last specified in that control section will be resumed.

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.

**Notes:**

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.

In the following example, addressability to the common area of storage is established relative to the named statement XYZ.

Name	Operation	Operand
	.	
	.	
	L	1,=A(XYZ)
	USING	XYZ,1
	MVC	PDQ(16),=4C'ABCD'
	.	
	.	
	COM	
XYZ	DS	16F
PDQ	DS	16C
	.	
	.	

No instructions or constants appearing in a common control section are assembled. Data can only be placed in a common control section through execution of the program. A blank common control section may include any assembler language instructions.

If the assignment of common storage is done in the same manner by each independent assembly, reference to a location in common by any assembly results in the same location being referenced. When the blank common control section is assembled, the initial value of the location counter is set to zero.

## External Dummy Sections

An external dummy section is a reference control section that allows you to 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

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 GETMAIN macro instruction). This is not only convenient, but you save space and prevent fragmentation of virtual storage.

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

*Generating an external dummy section:* An external dummy section is generated when you specify an DXD instruction or a DSECT instruction in combination with a Q-type address constant that contains the name of the DSECT instruction.

You 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 will compute the length and alignment required.
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 linkage editor or loader will insert the total length of all the external dummy sections that are specified in the source modules of your program. The linkage editor computes this length from the lengths of the individual external dummy sections supplied by the assembler.
4. Allocate a storage area using the computed total length.
5. Load the address of the allocated area into a register. Note that this register must contain this address throughout the whole program.
6. Add to the address in the register the offset into the allocated area of the desired external dummy section. The linkage editor inserts this offset into the fullword area reserved by the appropriate 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. Note that the source statements in an external dummy section are not assembled into object code. Thus, at execution time, you must insert the data described into the area reserved for the external dummy sections.

## **DXD—Define External Dummy Section**

The DXD instruction allows you to identify and define an external dummy section. The DXD instruction can be used anywhere in a source module, after the ICTL instruction, or after any source macro definitions that may be specified.

### **Notes:**

1. An external dummy section identified by a DXD instruction will not generate an entry in the external symbol dictionary (ESD) unless it is referenced by a Q-type address constant.
2. The DSECT instruction also defines an external dummy section, but only if the symbol in the name field appears in a Q-type address constant in the same source module. Otherwise, a DSECT instruction defines a dummy section.

### **Format of DXD:**

---

Name	Operation	Operand
A symbol	DXD	Duplication factor, type, modifiers, nominal value

---

The symbol in the name field 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, 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.

The linkage editor or loader uses the information provided by the assembler to compute the total length of storage required for all external dummy sections specified in a program.

**Note:** If two or more external dummy sections for different source modules have the same name, the linkage editor uses the most restrictive alignment, and the largest section to compute the total length.

### CXD—Cumulative Length External Dummy Section

The CXD instruction allows you to reserve a fullword area in storage. The linkage editor or loader will insert into this area the total length of all external dummy sections specified in the source modules that are assembled and linked into one program.

#### Format of CXD:

Name	Operation	Operand
Any symbol or blank	CXD	Not required

The symbol in the name field, if specified, represents the address of a fullword area aligned on a fullword boundary. This symbol has a length attribute value of 4. The linkage editor or loader inserts into this area the total length of storage required for all the external dummy sections specified in a program.

The following example shows how external dummy sections may be used.

#### ROUTINE A

Name	Operation	Operand
ALPHA	DXD	2DL8
BETA	DXD	4FL4
OMEGA	CXD	
	.	
	.	
	DC	Q(ALPHA)
	DC	Q(BETA)
	.	
	.	



## ROUTINE B

Name	Operation	Operand
GAMMA	DXD	5D
DELTA	DXD	10F
	.	
	.	
	DC	Q(GAMMA)
	DC	Q(DELTA)
	.	
	.	

## ROUTINE C

Name	Operation	Operand
EPSILON	DXD	4H
	.	
	.	
	DC	Q(EPSILON)
	.	
	.	

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 and 10 fullwords; Routine C wants 4 halfwords. At the time these routines are brought into storage, the sum of the individual lengths will be 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.

## Symbolic Linkages

Symbols may be defined in one module and referred to in another, thus effecting symbolic linkages between independently assembled program sections. The linkages can be effected only if the assembler is able to provide information about the linkage symbols to the linkage editor, which resolves these linkage references at load time.

*Establishing symbolic linkage:* You must establish symbolic linkage between source modules so that you can refer or branch to symbolic locations defined in the control sections of external source modules. 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 wish 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.
- In the external source modules, you must identify the symbols that are defined in those source modules, and to which you refer from the current source module. These symbols are called entry symbols, because they provide points of entry to a control section in a source module. You identify entry symbols with the ENTRY instruction.
- You must provide the A-type or V-type address constants needed by the assembler to reserve storage for the addresses represented by the external symbols.

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

*Referring to external data:* You should use the EXTRN instruction to identify the external symbol that represents data in an external source module, if you wish to refer to this data symbolically.

For example, you can identify the address of a data area as an external symbol and load the address constant specifying this symbol into a base register. Then, you use this base register when establishing the addressability of a dummy section that formats 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:* You should use the V-type address constant to identify the external symbol that represents the address in an external source module to which you wish to branch.

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 or CSECT 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 symbol identified is not considered as defined in the source module, and can be used as the name entry for any other statement in the same source module.

## **ENTRY—Identify Entry-Point Symbol**

The ENTRY instruction allows you to identify symbols defined in one source module so that they can be referred to in another source module. These symbols are entry symbols.

### **Format of ENTRY:**

---

Name	Operation	Operand
A sequence symbol or blank	ENTRY	One or more relocatable symbols, separated by commas

---

The following applies to the entry symbols identified in the operand field:

- They must be valid symbols.
- They must be defined in an executable control section.
- They must not be defined in a dummy control section, a common control section, or an external control section.
- The length attribute value of entry symbols is the same as the length attribute value of the symbol at its point of definition.

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.

The assembler lists each entry symbol of a source module in an external symbol dictionary, along with entries for external symbols, common control sections, and external control sections.

There is no restriction on the number of control sections, external symbols, and external dummy sections allowed by the assembler. The maximum number depends on the amount of main storage available during link editing.

### EXTRN—Identify External Symbol

The EXTRN instruction allows you to identify symbols referred to in a source module but defined in another source module. These symbols are external symbols.

#### Format of EXTRN:

---

Name	Operation	Operand
A sequence symbol or blank	EXTRN	One or more relocatable symbols, separated by commas

---

**External Symbols:** The following applies to the external symbols identified in the operand field:

- They must be valid symbols.
- They must not be used as the name entry of a source statement in the source module in which they are identified.
- They have a length attribute value of 1.
- They must be used alone and cannot be paired when used in an expression.

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, and external control sections.

There is no restriction on the number of control sections, external symbols, and external dummy sections allowed by the assembler. The maximum number depends on the amount of main storage available during link editing.

## WXTRN—Identify Weak External Symbol

The WXTRN statement allows you to identify symbols referred to in a source module but defined in another source module. The WXTRN instruction differs from the EXTRN instruction as follows:

- The EXTRN instruction causes the linkage editor to make an automatic search of libraries 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 this automatic search of libraries. The linkage editor will only resolve the linkage addresses if the external symbols that you identify in the WXTRN operand field are defined:
  - In a module that is linked and loaded along with the object module assembled from your source module, or
  - In a module brought in from a library because of the presence of an EXTRN instruction in another module linked and loaded with yours.

### Format of WXTRN:

---

Name	Operation	Operand
A sequence symbol or blank	WXTRN	One or more relocatable symbols separated by commas

---

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.

**Note:** If a symbol, specified in a V-type address constant, is also identified by a WXTRN instruction, it is assigned the same type code as the symbol in the WXTRN instruction.

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 warning messages.

---

## Chapter 4. Machine Instruction Statements

This chapter introduces the main functions of the machine instructions 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 appropriate principles of operation manual for your processor.

At assembly time, the assembler converts the symbolic assembler language representation of the machine instructions to the corresponding object code. It is this object code that the computer processes 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 DC and DS assembler instructions (see “Data Definition Instructions” on page 96). At execution time, the machine instructions can refer to these constants and areas, but the constants themselves are not executed.

As defined in the appropriate principles of operation manual, 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

You use general instructions to manipulate data that resides in general registers or in storage, or that is introduced from the instruction stream. These 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 being of 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 appropriate principles of operation manual.

### Notes:

1. Symbols used to represent immediate data (see HEX40 and TEN in the ALPHA2 and BETA1 instructions below) are assumed to be equated to absolute values between 0 and 255.
2. Symbols used to represent implicit addresses (see IMPLICIT, KEY, and NEWSTATE in the BETA and GAMMA2 instructions below) can be either relocatable or absolute.
3. Symbols used to represent displacements (see DISPL40 in the ALPHA2 instruction below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

### Examples:

ALPHA1	CLI	40(9),X'40'
ALPHA2	CLI	DISPL40(NINE),HEX40
BETA1	CLI	IMPLICIT,TEN
BETA2	CLI	KEY,C'E'
GAMMA1	LPSW	0(9)
GAMMA2	LPSW	NEWSTATE

When assembled, the object code for the ALPHA1 instruction, 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

You use the instructions with the SS format mainly to move data between two virtual storage locations. The operand fields and subfields must, therefore, designate virtual storage addresses and the explicit data lengths you wish to include. However, note that, in the Shift and Round Decimal (SRP) instruction, a 4-bit immediate data field (see 3 in SRP instruction below), with a value between 0 and 9, is specified as a third operand.

### Notes:

1. Symbols used to represent base registers (see BASE8 and BASE7 in the ALPHA2 instruction below) in explicit addresses are assumed to be equated to absolute values between 0 and 15.
2. Symbols used to represent explicit lengths (see NINE and SIX in the ALPHA2 instruction below) are assumed to be equated to absolute values between 0 and 256 for SS instructions with one length specification, and between 0 and 16 for SS instructions with two length specifications.
3. Symbols used to represent implicit addresses (see FIELD1, FIELD2, and FIELD1,XX'8' in the ALPHA3 and SRP instructions below) can be either relocatable or absolute.
4. Symbols used to represent displacements (see DISP40 and DISP30 in the ALPHA5 instruction below) in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

**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	T0,FROM
	SRP	FIELD1,X'8',3

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

FA858028701E

where

FA is the operation code.  
 8 is length L1.  
 5 is length L2.  
 8 is base register B1.  
 028 is the displacement from base register B1.  
 7 is base register B2.  
 01E is the displacement from base register B2.

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

D24F80007000

where

D2 is the operation code.  
 4F is length L.  
 8 is base register B1.  
 000 is the displacement from base register B1.  
 7 is base register B2.  
 000 is the displacement from base register B2.

**SSE Format**

You use the instructions with the SSE format mainly for control operations. The operand fields designate virtual storage addresses, encoded as base and displacement.

**Examples:**

ALPHA1	LASP	40(BASE8),30(BASE7)
ALPHA2	LASP	40(8),30(7)
BETA1	TPROT	LOC1,LOC2
BETA2	TPROT	DISP40(8),DISP30(8)

**Notes:**

1. Symbols used to represent base registers in explicit addresses (such as BASE8 and BASE7 in the ALPHA1 instruction) are assumed to be equated to absolute values between 0 and 15.
2. Symbols used to represent implicit addresses (such as LOC1,LOC2 in the BETA1 instruction) can be either relocatable or absolute.
3. Symbols used to represent displacements in explicit addresses (such as DISP40 and DISP30 in the BETA2 instruction) are assumed to be equated to absolute values between 0 and 4095.

When assembled, the object code of the ALPHA2 instruction, in hexadecimal, is:

E5008020701E

where

E500 is the operation code.

8 is base register B1.

02B is the displacement from base register B1.

7 is base register B2.

01E is the displacement from base register B2.



---

## Chapter 5. Assembler Instruction Statements

The following is a list of assembler instructions:

### Symbol Definition Instruction

EQU      Equate symbol

### Operation Code Definition Instruction

OPSYN   Equate operation code

### Data Definition Instructions

CCW      Define channel command word (Format 0: 24-bit data address)  
CCW0     Define channel command word (Format 0: 24-bit data address)  
CCW1     Define channel command word (Format 1: 31-bit data address)  
DC       Define constant  
DS       Define storage

### Program Sectioning and Linking Instructions (discussed in Chapter 3)

AMODE   Specify the addressing mode of a control section  
COM      Identify blank common control section  
CSECT   Identify control section  
CXD      Cumulative length of external dummy section  
DSECT   Identify dummy section  
DXD      Define external dummy section  
ENTRY   Identify entry-point symbol  
EXTRN   Identify external symbol  
LOCTR   Specify multiple location counters within a control section  
RMODE   Specify the residence mode of a control section  
START   Start assembly  
WXTRN   Identify weak external symbol

### Base Register Instructions (discussed in Chapter 3)

DROP     Drop base address register  
USING    Use base address register

### Program Control Instructions

CNOP     Conditional no operation  
COPY     Copy predefined source coding  
END      End assembly  
ICTL     Input format control  
ISEQ     Input sequence checking  
LORG     Begin literal pool  
ORG      Set location counter  
POP      Restore status of current PRINT or USING  
PUNCH    Punch a card  
PUSH     Push-down queue for current PRINT or USING  
REPRO    Reproduce following card

### Listing Control Instructions

EJECT    Start new page  
PRINT    Print optional data  
SPACE    Space listing  
TITLE    Identify assembly output

---

## Symbol Definition Instruction

### EQU—Equate Symbol

The EQU instruction allows you to assign absolute or relocatable values to symbols. You can use it for the following purposes:

1. To assign single *absolute values* to symbols
2. To *assign the values of previously defined symbols* or expressions to new symbols, thus allowing you to use different mnemonics for different purposes.
3. To *compute expressions* whose values are unknown at coding time or difficult to calculate. The value of the expressions is then assigned to a symbol.

The EQU instruction can be used anywhere in a source module after the ICTL instruction, or after any source macro definitions that may be specified. Note, however, that the EQU instruction can initiate an unnamed control section (private code) if it is specified before the first control section (initiated by a START or CSECT instruction).

#### Format of EQU:

---

Name	Operation	Operand
A variable symbol or ordinary symbol	EQU	Expression-1 or Expression-1,Expression-2 or Expression-1,Expression-2,Expression-3, or Expression-1,,Expression-3

---

**Note:** The two commas in the last option above indicate the absence of expression 2.

Expression 1 represents a value. It must always be specified and it may assume 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 four bytes are printed in the program listings opposite the symbol.

Any symbols used in the first operand (expression 1) need not be previously defined. If the expression in the first operand is complexly relocatable, 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), and 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.

Expression 2 represents a length attribute. It is optional, but, if specified, it must have an absolute value in the range of 0 to 65,535. Expression 3 represents a type attribute. It is optional, but, if specified, must be a self-defining term with a value in the range of 0 to 255.

Any symbols appearing in expressions 2 and/or 3 must have been previously defined.

**Expression 1 (Value):** The assembler assigns the relocatable or absolute value of expression 1 to the symbol in the name field at assembly time. If expression 2 is omitted, the assembler also 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. The length attribute value is described in "Chapter 2. Coding and Structure." It is defined as follows:

1. If the leftmost term is a location counter reference (\*), a self-defining term, or a symbol length attribute value reference, the *length attribute is 1*. Note that this also applies if the leftmost term is a symbol that is equated to any of these values.
2. If the leftmost term 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 and DS instructions, have a length attribute value of 1. However, the name of a CCW, CCW0, or CCW1 instruction has a *length attribute value of 8*.
5. The length attribute value assigned in cases 2 to 4 above only applies to the *assembly-time* value of the attribute. Its value at *preassembly* time, during conditional assembly processing, is always 1.
6. Further, if expression 3 is omitted, the assembler assigns a type attribute value of U to the symbol in the name field.

**Expression 2 (Length-Attribute Value):** If expression 2 is specified, the assembler assigns its value as a *length attribute value* to the symbol in the name field. This value overrides the normal length attribute value implicitly assigned from expression 1. If expression 2 is a self-defining term, the assembler also assigns the length attribute value to the symbol at preassembly time (during conditional assembly processing).

**Note:** This expression must have been previously defined.

**Expression 3 (Type-Attribute Value):** If expression 3 is specified, it must be a self-defining term. The assembler assigns its EBCDIC value as a *type attribute value* to the symbol in the name field. This value overrides the normal type attribute value implicitly assigned from expression 1.

*Using Preassembly Values:* You can use the preassembly values assigned by the assembler in conditional assembly processing.

If only expression 1 is specified, the assembler assigns a preassembly value of 1 to the length attribute, and a preassembly value of U to the type attribute of the symbol. These values can be used in conditional assembly (although references to the length attribute of the symbol will be flagged). The absolute or relocatable value of the symbol, however, is not assigned until assembly, and thus may not be used at preassembly.

If you include expressions 2 and 3 and wish to use the explicit attribute values in preassembly processing, then

- The symbol in the name field must be an ordinary symbol.
- Expression 2 and expression 3 must be single self-defining terms.

**Symbol in the Name Field:** The assembler assigns an absolute or relocatable value, a length attribute value, and a type attribute value to the symbol in the name field.

The absolute or relocatable value of the symbol is assigned at assembly time, and is, therefore, not available for conditional assembly processing at preassembly time.

The type and length attribute values of the symbol are available for conditional assembly processing under the following conditions:

- The symbol in the name field must be an ordinary symbol.
- Expression 2 and expression 3 must be single self-defining terms.

---

## Redefining Symbolic Operation Codes

### OPSYN—Equate Operation Code

The OPSYN instruction allows you to define your own set of symbols to represent operation codes for:

- Machine and extended mnemonic branch instructions
- Assembler instructions, including conditional assembly instructions

You can also prevent the assembler from recognizing a symbol that represents a current operation code.

## Two formats of OPSYN:

---

Name	Operation	Operand
Any symbol or operation code	OPSYN	An operation code
	or	
An operation code	OPSYN	Blank

---

The OPSYN instruction can be coded anywhere in the program to redefine an operation code.

The operation code specified in the name field or the operand field must represent either:

1. The operation code of one of the assembler or machine instructions as described in "Chapter 3. Addressing, Program Sectioning, and Linking" on page 43, "Chapter 4. Machine Instruction Statements," "Chapter 5. Assembler Instruction Statements," or "Chapter 9. How to Write Conditional Assembly Instructions" on page 211, respectively, or
2. The operation code defined by a previous OPSYN instruction.

The OPSYN instruction assigns the properties of the operation code specified in the operand field to the symbol in the name field. A blank in the operand field causes the operation code in the name field to lose its properties as an operation code.

### Examples:

1. 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 blank operand field. In the following example, L and LR will both possess the properties of the LR machine instruction operation code:

```
L      OPSYN      LR
```

2. When the same symbol appears in the name field of two OPSYN instructions, the latest definition takes precedence. In the example below, STORE now represents the STH machine operation:

```
STORE  OPSYN      ST  
STORE  OPSYN      STH
```

**Redefining Conditional Assembly Instructions:** A redefinition of a conditional assembly operation code will have an effect only on macro definitions appearing after the OPSYN instruction. Thus, the new definition is not valid during the processing of subsequent macro instructions calling a macro that was defined prior to the OPSYN statement.

Any OPSYN statement redefining the operation code of an instruction generated from a macro instruction will, however, be valid, even if the definition of the macro was made prior to the OPSYN statement. The following example illustrates this difference between conditional assembly instructions and model statements within macro instructions.

	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	...	Evaluated as AIF instruction; generated AIFs not printed]
	MVC	...	Evaluated as MVI instruction
	.		
	.		Open code started at this point
	AIF	...	Evaluated as AGO instruction
	MVC	...	Evaluated as MVI instruction

AIF and MVC instructions are used in a macro definition. OPSYN instructions are used to assign the properties of AGO to AIF and to assign the properties of MVI to MVC, after the macro definition has been edited. In subsequent calls to that macro, AIF is still defined as an AIF operation, while MVC is treated as an MVI operation. In open code following the OPSYN instructions, the operations of both instructions are derived from their new definitions. If the macro is redefined, either by means of a loop to a point before the macro definition or by a subsequent macro definition defining the same macro, the new definitions of AIF and MVC (that is, AGO and MVI) will be fixed for future expansions.

---

## Data Definition Instructions

The data definition instruction statements are: Define Constant (DC), Define Storage (DS), and three types of Channel Command Words (CCW, CCW0, and CCW1).

These statements are used to define constants, reserve storage, and specify the contents of channel command words, respectively. You can also provide a label for these instructions and then refer to the data symbolically in the operands of machine and assembler instructions. This data is generated and storage is reserved at assembly time, and used by the machine instructions at execution time.

### DC—Define Constant

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 can generate the following types of constants:

Type of Constant	Function	Example		
Address	Defines address mainly for the use of fixed-point and other instructions in the universal instruction set	ADCON	L DC	5,ADCON 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	PCON AREA	AP DC DS	AREA,PCON P'100' P
Fixed-point	Used by the fixed-point and other instructions of the universal set	FCON	L DC	3,FCON F'100'
Floating-point	Used by floating-point instructions	ECON	LE DC	2,ECON E'100.50'
Graphic	Defines character strings or messages that contain pure double-byte data	OBCS	DC	G'<.D.B.C.S .S.T.R.I.N.G>'
Hexadecimal	Defines large bit patterns	PATTERN	DC	X'FF00FF00'

Figure 23. Types of Data Constants

#### Format of DC:

Name	Operation	Operand
Any symbol or blank	DC	One or more operands separated by commas

The symbol in the name field represents the address of the first byte of the assembled constant. If several operands are specified, the first constant defined is addressable by the symbol in the name field. The other constants can be reached by relative addressing.

Each operand in a DC instruction consists of four subfields: the first three describe the constant; the fourth provides the nominal value(s) for the constant(s) to be generated. The subfields of each DC operand are written in the following sequence:

1	2	3	4
Duplication Factor	Type	Modifiers	Nominal Value(s)

For example, in

```
10XL2'FA'
```

the four subfields are:

- Duplication factor is 10
- Type is X
- Modifier is L2
- Nominal value is FA

If all subfields are specified, the order given above is required. The first and third subfields can be omitted, but the second and fourth must be specified in that order.

## Rules for DC Operand

1. The type subfield and the nominal value must always be specified.
2. The duplication factor 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 fourth subfield, they must be separated by commas and be of the same type. Multiple nominal values are not allowed for character constants.
5. The descriptive subfields apply to all the nominal values.

**Note:** Separate constants are generated for each separate operand and nominal value specified.

6. No blanks are allowed:
  - Between subfields.
  - Between multiple operands.
  - Within any subfields, unless they occur as part of the nominal value of a character constant, or as part of a character self-defining term in a modifier expression, or in the duplication factor subfield.

## Information about Constants

**Symbolic Addresses of 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 will determine whether the constant is to be aligned on a particular storage boundary or not (see “Alignment of Constants” below). The value of the symbol that names the DC instruction is the address of the leftmost 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:

- The implicit length (see (1) in Figure 24 on page 99) of the constant when no explicit length is specified in the operand of the constant, or
- The explicitly specified length (see (2) in Figure 24) of the constant.

**Note:** 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 implicitly or explicitly specified length.



**Alignment of Constants:** The assembler aligns constants on different boundaries according to the following:

- On boundaries implicit to the type of constant (see (1) in Figure 25 on page 100) when no length specification is supplied.
- On byte boundaries (see (2) in Figure 25) when an explicit length specification is made.

Bytes that are skipped to align a constant at the proper boundary are not considered part of the constant. They are filled with zeros.

**Notes:**

1. The automatic alignment of constants and areas does not occur if the NOALIGN assembler option has been specified when the assembler was invoked.
2. Alignment can be forced to any boundary by a preceding DS (or DC) instruction with a zero duplication factor. This occurs when either the ALIGN or NOALIGN option is set.

Type of constant	Implicit Length <sup>1</sup>	Examples	Value of Length Attribute <sup>2</sup>
B	as needed	DC B'10010000'	1
C	as needed	DC C'WOW' DC CL8'WOW' <sup>2</sup>	3 8
G	as needed	DC G'<DaDb>' DC GL8'<DaDb>'	4 8
X	as needed	DC X'FFEE00' DC XL2'FFEE' <sup>2</sup>	3 2
H	2	DC H'32'	2
F	4	DC FL3'32' <sup>2</sup>	3
P	as needed	DC P'123' DC PL4'123' <sup>2</sup>	2 4
Z	as needed	DC Z'123' DC ZL10'123' <sup>2</sup>	3 10
E	4		
D	8		
L	16		
Y	2	DC Y(HERE)	2
A	4	DC AL1(THERE)	1
S	2		
V	2		
Q	4		

Figure 24. Length Attribute Value of Symbol Naming Constants

**Notes to Figure 24:**

1. Depends on type.
2. Depends on whether or not an explicit length is specified in 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 second operand subfield, 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 267).

Type of constant	Implicit Boundary Alignment <sup>1</sup>	Examples	Boundary Alignment
B	byte		
C	byte		
G	byte		
X	byte		
H	halfword	DC H'25'	halfword
		DC HL3'25' <sup>2</sup>	byte
F	halfword	DC F'225'	fullword
		DC FL7'225' <sup>4</sup>	byte
P	byte	DC P'2934'	byte
Z	byte	DC Z'1235'	byte
		DC ZL2'1235' <sup>2</sup>	byte
E	fullword	DC E'1.25'	fullword
		DC EL5'1.25' <sup>2</sup>	byte
D	doubleword	DC 8D'95'	doubleword
		DC 8DL7'95' <sup>2</sup>	byte
L	doubleword	DC L'2.57E65'	doubleword
Y	halfword	DC Y(HERE)	halfword
A	fullword	DC AL1(THERE) <sup>2</sup>	byte
S	halfword		
V	fullword		
Q	fullword		

Figure 25. Alignment of Constants

**Padding:** If more space is available 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,Q) constants
- With EBCDIC zeros on the left (XX'F0') for the zoned decimal (Z) constants
- With EBCDIC blanks on the right (XX'40') for the character (C) constants
- With double-byte blanks on the right (XX'4040') for the graphic (G) constants

### Notes:

1. Floating-point constants (E,D,L) are also padded on the right with zeros.
2. Padding is on the left for all constants except the character constant and the graphic constant.
3. Padding is on the right for 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), decimal (P and Z), and address (A and Y) constants.
- On the right for the character (C) constant and the graphic (G) constant.

However, the fixed-point constants (H and F) will not be truncated but flagged if significant bits would be lost to truncation.

**Notes:**

1. Floating-point constants (E,D,L) are not truncated; they are rounded.
2. The above rules for padding and truncation also apply when the bit-length specification is used (see “Subfield 3: Modifiers” below).
3. Double-byte data in C-type constants cannot be truncated because truncation creates invalid double-byte data. Error IEV208 “TRUNCATION INTO DOUBLE-BYTE DATA IS NOT PERMITTED” will be issued if such truncation is attempted.
4. Truncation of double-byte data in G-type constants is permitted, because the length modifier restrictions (see “Subfield 3: Modifiers” below) will ensure that invalid double-byte data cannot be created by truncation.

### Subfield 1: Duplication Factor

The duplication factor may be omitted. 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 factor can be specified by an unsigned decimal self-defining term or by an absolute expression enclosed in parentheses.

The expression should have a positive value or be equal to zero.

**Notes:**

1. The value of a location counter reference in a duplication factor is the value before any alignment to boundaries is done, according to the type of constant specified.
2. A duplication factor of zero is permitted 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” above).
  - The length attribute of the symbol naming the constant is established according to the implicitly or explicitly specified length.
3. If duplication is specified for an address constant containing a location counter reference, the value of the location counter reference is incremented by the length of the constant before each duplication is performed (for examples, see “Address Constants—A and Y” on page 112).

## Subfield 2: Type

The type subfield must be specified. From the type specification, the assembler determines how it is to interpret the constant and translate it into the appropriate machine format. The type is specified by a single-letter code as shown in Figure 26.

Further information about these constants is provided in the discussion of the constants themselves under “Subfield 4: Nominal Value” on page 106.

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; each address normally a fullword
Q	Address	Space reserved for external dummy section offset

Figure 26. Type Codes for Constants

The type specification indicates to the assembler:

1. How the nominal value(s) specified in subfield 4 is to be assembled; 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 specification is present.
3. How much storage the constant is to occupy, according to the implicit length of the constant, if no explicit length specification is present (for details, see “Padding and Truncation of Values” on page 100).

## Subfield 3: Modifiers

Modifiers describe the length in bytes desired for a constant (in contrast to an implied length), and the scaling and exponent for the constant.

The three modifiers are:

1. The length modifier (L), which explicitly defines the length in bytes desired for a constant. For example:

```
LENGTH    DC          XL10'FF'
```

2. The scale modifier (S), which is only used with the fixed-point or floating-point constants (for details, see below under "Scale Modifier"). For example:

```
SCALE     DC          FS8'35.92'
```

3. The exponent modifier (E), that is only used with fixed-point or floating-point constants, and which 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, exponent. For example:

```
ALL3      DC          DL7S3E50'2.7182'
```

Symbols used in subfield 3 need not be previously defined. This does not apply to literals.

**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, and any symbols it contains must be previously defined.

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 100).
- No boundary alignment, according to constant type, is provided (see "Alignment of Constants" above).
- 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.
- The length modifier must be a multiple of 2 in a G-type constant.

**Note:** When no length is specified, for character and graphic constants, the whole constant is assembled into its implicit length.

**Bit-Length Specification:** The length modifier can be specified to indicate the number of bits into which a constant is to be assembled. The bit-length specification 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. Symbols that it contains need not be previously defined.

The value of  $n$  must lie between 1 and 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 specification cannot be used with the G-, S-, V-, and Q-type constants.

When only one operand and one nominal value are specified in a DC instruction, the following rules apply:

1. The bit-length specification 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 specified is greater than 8.  
  
If the field does not end at a byte boundary and if the bit length specified is not a multiple of 8, the remainder of the last byte is filled with zeros.
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 allow bit-length specification
3. The nominal value is padded or truncated to fit the field (see “Padding and Truncation of Values” on page 100).  
  
Padding of character constants is done with hexadecimal blanks, X'40'; other constant types are padded with zeros.

**Note:** 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. L'TRUNCF is equal to 2. Thus, a reference to TRUNCF would address the entire two 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 specifications also apply, except:

1. The first field allocated starts at a byte boundary, but the succeeding fields start at the next available bit.
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.  
  
**Note:** If duplication is specified, filling with zeros occurs once at the end of all the fields occupied by the duplicated constants.
3. The length attribute value of the symbol naming the DC instruction is equal to the number of integral bytes that would be needed to contain the bit length specified for the first constant to be assembled. ~

**Note: For double-byte data in C-type constants:** If bit-length specifications are used, with a duplication factor greater than 1, and a bit-length which is not a multiple of 8, then the double-byte data will no longer be valid for devices capable of presenting DBCS characters. No error message will be 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 length (implicit or explicit)
  - The number of nominal values
  - The duplication factor, if specified

2. The number of bytes skipped for the boundary alignment between different operands.

**Scale Modifier:** The scale modifier specifies the amount of internal scaling that is desired:

- 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 constant.

The allowable range for each type of constant is as follows:

Fixed-point constant H	-187 to 15
Fixed-point constant F	-187 to 30
Floating-point constant E	0 to 5
Floating-point constant D	0 to 13
Floating-point constant L	0 to 27

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 types of specification 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 to be 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 to be deleted from the integer portion of the binary number.
3. When 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 Floating-Point Constants:* The scale modifier for 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 to be 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 is when 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 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 En, 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.

**Notes:**

1. The exponent modifier is not to be confused 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 specification are present in a DC operand, their values are algebraically added together before the nominal value is converted to binary form. However, this sum must lie within the permissible range of -85 to +75.

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 4: Nominal Value**

The nominal value subfield must always be 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. The formats for specifying constants are described as follows:

Constant Type	Single Nominal Value	Multiple Nominal Value
C	'value'	not allowed
G	'<.v.a.l.u.e>'	not allowed
B		
X		
H		
F		
P	'value'	'value,value,...value'
Z		
E		
D		
L		
A		
Y		
S	(value)	(value,value,...value)
Q		
V		

As the above list shows, a data constant value (any type except A, Y, S, Q, and V) is enclosed by apostrophes. An address constant value (type A, Y, S, Q, or V) is enclosed by parentheses. To specify two or more values in the subfield,



the values must be separated by commas, and the entire sequence of values must be enclosed by the appropriate delimiters; that is, apostrophes or parentheses. Multiple values are not permitted for character constants.

How nominal values are specified and interpreted by the assembler is explained in each of the following subsections, starting with “Binary Constant—B” below.

**Literal Constants:** Literal constants allow you to 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 difference between a literal, a data constant, and a self-defining term is described in “Literals” on page 36.

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.

The following text describes each of the constant types and provides examples. The constant types are:

Binary  
Character  
Graphic  
Hexadecimal  
Fixed-Point  
Decimal  
Packed Decimal  
Zoned Decimal  
Address  
Floating-Point

**Binary Constant—B:** The binary constant allows you to specify the precise bit pattern you want assembled into storage. Each binary constant is assembled into the integral number of bytes (see (1) in Figure 27 on page 108) required to contain the bits specified.

The following example shows the coding used to designate a binary constant. BCON would have a length attribute of 1.

BCON	DC	B'11011101'
BTRUNC	DC	BL1'100100011'
BPAD	DC	BL1'101'

BTRUNC would assemble with the leftmost bit truncated, as follows:

```
00100011
```

BPAD would assemble with five zeros as padding, as follows:

```
00000101
```

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u>	As needed	B DC B'10101111'	L'B = 1 <sup>1</sup>
Implicit length: (length modifier not present)		C DC B'101'	B'C = 1 <sup>1</sup>
Alignment: (Length modifier not present)	Byte		
Range for length:	1 to 256 (byte length)		
	.1 to .2048 (bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u>			
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		

Figure 27. Binary Constants

**Character Constant—C:** The character constant allows you to specify character strings, such as error messages, identifiers, or other text, that the assembler will convert into their binary (EBCDIC) representation.

Any of the valid 256 punch combinations can be designated in a character constant. Each character specified in the nominal value subfield is assembled into one byte (see (1) in Figure 28 on page 109).

Multiple nominal values are not allowed, because if a comma is specified in the nominal value subfield, the assembler considers the comma a valid character (see (2) in Figure 28) and, therefore, assembles it into its binary (EBCDIC) representation. For example

```
DC          C'A,B'
```

is assembled as A,B with object code C16BC2.

Special consideration must be given to representing apostrophes and ampersands as characters. Each single apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. They are assembled as single apostrophes and ampersands (see (3) in Figure 28).

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 blanks 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 count as only one byte.

```
FIELD    DC          C'TOTAL IS &&10'
```

Note that, 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 would be:

```
ABCDABCDABCD
```

On the other hand, if the length had been specified as 6 instead of 4, the generated constant would have been:

```
ABCDE ABCDE ABCDE
```

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u> Implicit length: (length modifier not present)	As needed	C DC C'LENGTH'	L'C = 6
Alignment: (Length modifier not present)	Byte		
Range for length:	1 to 256 (byte length)		
	.1 to .2048 (bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u>			Object code
Represented by:	Characters (all 256 8-bit combinations)	DC C'A'B' DC C'A&&B'	X'C17DC2' <sup>3</sup> X'C150C2' <sup>3</sup>
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per operand:	One	DC C'A,B'	Object code X'C16BC2' <sup>2</sup>
Padding:	With blanks at right (X'40')		
Truncation of assembled value:	At right		

Figure 28. Character Constants

Note that the same constant could be specified as a literal.

```
MVC          AREA(12),=3CL4'ABCDE'
```

*Double-byte data in character constants:* When the assembler is invoked with the DBCS option, double-byte data may 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 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>'
```

will result 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.

Examples of character constants containing double-byte data are:

```
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 DBCS2 is 11. No truncation of double-byte character strings within C-type constants is allowed, since invalid double-byte data would be created.

**Graphic Constant—G:** When the assembler is invoked with the DBCS option, 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>'
```

will result 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 blanks 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.

**Note:** The type attribute of a Graphic constant is not G; it is @.

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed	3G'<.A>'	Object code X'42C142C142C1'
2. <u>Modifiers</u>	As needed		
Implicit length: (length modifier not present)	Twice the number of DBCS charac- ters	GC DC G'<.A.B>'	L'GC = 4
Alignment:	Byte		
Range for length:	2 to 256 (byte length). Must be a multiple of 2. Bit length not allowed.		
3. <u>Nominal value</u>			Object code
Represented by:	DBCS characters delimited by SO and SI	DC G'<.&.'> DC G'<.A><.B>'	X'4250427D' X'42C142C2'
Enclosed by:	Apostrophes		
Number of values per operand:	One	DC G'<.A.,.B>'	Object code X'42C1426B42C2'
Padding:	With DBCS blanks at right	DC GL6'<.A>'	Object code X'42C140404040'
Truncation of assembled value:	At right	DC GL2'<.A.B>'	Object code X'42C1'

Figure 29. Graphic Constants

**Hexadecimal Constant—X:** You can use hexadecimal constants to generate large bit patterns more conveniently than with binary constants. Also, the hexadecimal values you specify in a source module allow you to 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 Figure 30 on page 112) specified in the nominal value subfield is assembled into four bits (their binary patterns can be found in "Self-Defining Terms" on page 29). See (2) in Figure 30. The implicit length in bytes of a hexadecimal constant is then half the number of hexadecimal digits specified (assuming that a hexadecimal zero is added to an odd number of digits). See (3) in Figure 30.

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 would set the first and third bytes of a word to 1s:

```

          DS          0F
TEST     DC          X'FF00FF00'
```

The DS instruction sets the location counter to a fullword boundary. (See "DS—Define Storage" on page 127.)

The next example uses a hexadecimal constant as a literal and inserts 1s into bits 24 to 31 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'
```

The resulting constant is 6F4E, which occupies the specified 2 bytes. It is duplicated three times, as requested by the duplication factor. If it had merely been specified as 3X'A6F4E', the resulting constant would have a hexadecimal zero in the leftmost position.

```
0A6F4E0A6F4E0A6F4E
```

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u>	As needed	X DC X'FF00A2'	L'X = 3 <sup>2</sup>
Implicit length: (length modifier not present)		Y DC X'F00A2'	X'Y = 3 <sup>2</sup>
Alignment: (Length modifier not present)	Byte		
Range for length:	1 to 256 (byte length)		
	.1 to .2048 (bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u>			Object code
Represented by:	Hexadecimal digits (0 to 9 and A to F)	DC X'1F' DC X'91F'	X'1F' <sup>1</sup> X'091F' <sup>3</sup>
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	At left		

Figure 30. Hexadecimal Constants

**Fixed-Point Constant—F and H:** Fixed-point constants allow you to introduce data that is in a form suitable for the operations of the fixed-point machine instructions of the universal instruction set. The constants you define can also be automatically aligned to the proper fullword or halfword boundary for the instructions that refer to addresses on these boundaries (unless the NOALIGN option has been specified; see “Information about Constants” on page 98). You can perform algebraic functions 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 if desired. The format of the constant is as follows:

1. The nominal value can be a signed (see (1) in Figure 31 on page 114)—plus is assumed if the number is unsigned—integer, fraction, or mixed number (see (2) Figure 31) followed by an exponent (see (3) in Figure 31): positive or negative.
2. The exponent must lie within the permissible range (see (4) in Figure 31). If an exponent modifier is also specified, the algebraic sum (see (5) in Figure 31) of the exponent and the exponent modifier must lie within the permissible range.

Some examples of the range of values that can be assembled into fixed-point constants are given below:

<b>Length</b>	<b>Range of values that can be assembled</b>
8	$-2^{63}$ to $2^{63}-1$
4	$-2^{31}$ to $2^{31}-1$
2	$-2^{15}$ to $2^{15}-1$
1	$-2^7$ to $2^7-1$

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 performed, 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 will 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 twos complement form.
5. Duplication is applied after the constant has been assembled.

A field of three fullwords is generated from the statement below. The location attribute of CONWRD is the address of the leftmost byte of the first word, and the length attribute is 4, the implied length for a fullword fixed-point constant. The expression CONWRD + 4 could be used to address the second constant (second word) in the field.

```
CONWRD    DC          3F'658474'
```

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u>	Fullword: 4 bytes Implicit length: (length modifier not present)		
Alignment: (Length modifier not present)	Fullword or halfword		
Range for length:	1 to 8 (byte length)  .1 to .64 (bit length)		
Range for scale	F: -187 to +30 H: -187 to +15		
Range for exponent	-85 to +75 <sup>a</sup>	DC HE +90'2E-88' <sup>s</sup>	value = 2x10 <sup>2</sup>
3. <u>Nominal value</u>			
Represented by:	Decimal digits (0 to 9)	Fullword: DC F'-200' <sup>1</sup> DC FS4'2.25' <sup>2</sup>  Halfword: DC H' +200' DC HS4'.25'	
Enclosed by:	Apostrophes		
Exponent allowed:	Yes	Fullword: DC F'2E6' <sup>1</sup>  Halfword: DC H'2E-6'	
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	Not allowed (error message issued)		

Figure 31. Fixed-Point Constants

The next statement causes the generation of a 2-byte field containing a negative constant. Notice that scaling has been specified in order to reserve 6 bits for the fractional portion of the constant.

```
HALFCON DC HS6'-.25.46'
```

The next 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 could be specified as a literal:

```
AH 7,=HS12'3.50E-2'
```



The final example specifies three constants. Notice that the scale modifier requests 4 bits for the fractional portion of each constant. The 4 bits are provided whether or not the fraction exists.

```
THRECON DC FS4'10,25.3,100'
```

**Decimal Constants—P and Z:** The decimal constants allow you to introduce data in a form suitable for the operations of the decimal feature machine instructions. 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 digits in the rightmost byte.

The nominal value can be a signed (plus is assumed if the number is unsigned) decimal number. A decimal point may be written anywhere in the number, or it may 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.

The specified digits are assumed to constitute an integer (see (1) in Figure 32 on page 116). You may determine proper decimal point alignment either by defining data so that the point is aligned or by selecting machine instructions that will operate on the data properly (that is, shift it for purposes of alignment).

Decimal constants are assembled as follows:

*Packed Decimal Constants:* Each digit is converted into its 4-bit binary equivalent (see (2) in Figure 32). The sign indicator (see (3) in Figure 32) is assembled into the rightmost four bits of the constant.

*Zoned Decimal Constants:* Each digit is converted into its 8-bit EBCDIC representation (see (4) in Figure 32). The sign indicator (see (5) in Figure 32) replaces the first four bits of the low-order byte of the constant.

The range of values that can be assembled into a decimal constant is shown below:

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 an even number of packed decimal digits is specified, one digit will be left unpaired because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits will be set to zeros and the rightmost four bits will contain the odd (first) digit.

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u>	As needed	Packed: P DC P'+593	L'P = 2
Implicit length: (length modifier not present)		Zoned: Z DC Z'-593	L'Z = 3
Alignment: (Length modifier not present)	Byte		
Range for length:	1 to 16 (byte length)		
	.1 to .128 (bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u>			
Represented by:	Decimal digits (0 to 9)	Packed: DC P'5.5' <sup>1</sup> DC P'55' <sup>1</sup> DC P'+555' <sup>2</sup>	Object code X'055C' X'055C' X'555C' <sup>3</sup>
		Zoned: DC Z'-555' <sup>4</sup>	Object code X'F5F5D5' <sup>5</sup>
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		

Figure 32. Decimal Constants

**Examples:**

```

DC          P'+1.25'
DC          Z'-543'
DC          Z'79.68'
DC          PL3'79.68'

```

The following 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). Note that a literal could not specify both operands.

DECIMALS	DC	PL8'+25.8,-3874, +2.3',Z'+80,-3.72'	X
----------	----	--	---

The last example illustrates the use of a packed decimal literal.

UNPK	OUTAREA,=PL2'+25'
------	-------------------

**Address Constants:** An address constant is 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 are also dependent on the use of the USING assembler instruction and the loading of registers. Coding examples illustrating these considerations are provided in “How to Use the USING Instruction” in “USING—Use Base Address Register” on page 44.

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 entire sequence is enclosed by parentheses. There are five types of address constants: A, Y, S, Q, and V. A relocatable address constant may not be specified with bit lengths.

*Complex Relocatable Expressions:* A complex relocatable expression can only be used to specify an A- or Y-type address constant. These expressions contain two or more unpaired relocatable terms and/or negative relocatable terms in addition to any absolute or paired relocatable terms that may be present. 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.

*Address Constants—A and Y:* The following sections describe how the different types of address constants are assembled from expressions that usually represent storage addresses, and how the constants are used for addressing within and between source modules.

In the A-type and Y-type address constant, you can specify any of the three types of assembly-time expressions whose values the assembler then computes and assembles into object code. You 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

**Notes:**

1. No bit-length specification (see (1) in Figure 33 on page 118) is allowed when a relocatable or complex relocatable expression (see (2) in Figure 33) is specified. The only explicit lengths that can be specified with these addresses are:
  - 3 or 4 bytes for A-type constants
  - 2 bytes 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 Figure 33)
- A duplication factor (see (4) in Figure 33)

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.

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed	A DC 5AL1(*-A) <sup>4</sup>	Object code X'0001020304'
2. <u>Modifiers</u> Implicit length: (length modifier not present)	A-type: 4 bytes Y-type: 2 bytes		
Alignment: (Length modifier not present)	A-type: fullword Y-type: halfword		
Range for length:	A-type: 1 to 4 <sup>1</sup> (byte length) .1 to .32 (bit length)  Y-type: 1 to 2 (byte length) .1 to .16 (bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u> Represented by:	Absolute, relocatable, or complex relocatable expressions <sup>2</sup>	A-type: DC A(ABSOL + 10)  Y-type: DC Y(RELOC + 32) A DC Y(*-A,* + 4) <sup>3</sup>	values = 0,A + 6 <sup>3</sup>
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	At left		

Figure 33. A and Y Address Constants

**Caution:** Specification of Y-type address constants with relocatable expressions should be avoided in programs that are to be executed on machines having more than 32,767 bytes of storage capacity. In any case, Y-type relocatable address constants should not be used in programs to be executed under IBM System/370 control.

The A-type and Y-type address constants are processed as follows: If the nominal value is an absolute expression, it is computed to its 32-bit value and then truncated 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 when the object modules are transformed into load modules. The 24-bit (or smaller) 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 four bytes. Note that there is a location counter reference in one. The value of the location counter will be the address of the first byte allocated to the fourth constant. The second statement 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)
```

**Note:** When the location counter reference occurs in a literal, as in the LM instruction above, the value of the location counter is the address of the first byte of the instruction.

*Address Constant—S:* You can use the S-type address constant to assemble an explicit address; that is, an address in base-displacement form. You can specify the explicit address yourself or allow the assembler to 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 Figure 34 on page 120) representing an implicit address
2. As two absolute expressions (see (2) in Figure 34) the first of which represents the displacement and the second, the base register.

The address value represented by the expression in 1 in Figure 34, will be converted by the assembler into the proper base register and displacement value. An S-type constant is assembled as a halfword and aligned on a halfword boundary. The leftmost four bits of the assembled constant represent the base register designation; the remaining 12 bits, the displacement value.

If length specification is used, only 2 bytes may be specified. S-type address constants may not be specified as literals.

**Address Constant—V:** The V-type constant allows you to reserve storage for the address of a location in a control section that lies in another source module. You should use the V-type address constant only to branch to the external address specified. This use is contrasted with another method; that is, of specifying an external symbol, identified by an EXTRN instruction, in an A-type address constant.

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 Figure 35) 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.

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u> Implicit length: (length modifier not present)	2 bytes		
Alignment: (Length modifier not present)	Halfword		
Range for length:	2 only (no bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
4. <u>Nominal value</u> Represented by:	Absolute or relocatable, expression <sup>1</sup>	DC S(RELOC) DC S(1024)	Base Disp C XXX 0 400
	Two absolute expressions <sup>2</sup>	DC S(512(12))	C 200
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	Not applicable		
Truncation of assembled value:	Not applicable		

Figure 34. S Address Constants

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u>	4 bytes		
Implicit length: (length modifier not present)			
Alignment: (Length modifier not present)	Fullword		
Range for length:	4 or 3 only (no bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u>			
Represented by:	A single relocatable symbol	DC V(MODA) <sup>1</sup> DC V(EXTADR) <sup>1</sup>	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	Not applicable		

Figure 35. V Address Constants

In the following example, 12 bytes will be reserved, because there are three symbols. The value of each assembled constant will be zero until the program is loaded. It must be emphasized that a V-type address constant of length less than 4 can and will be processed by the assembler, but cannot be handled by the linkage editor.

```
VCONST    DC          V(SORT,MERGE,CALC)
```

*Address Constant—Q:* You use this constant to reserve storage for the offset into a storage area of an external dummy section. The offset is entered into this space by the linkage editor. When the offset is added to the address of an overall block of storage set aside for external dummy sections, it allows you to address the desired section.

For a description of the use of the Q-type address constant in combination with an external dummy section, see “External Dummy Sections” on page 66. See also Figure 36 for details.

In the following example, to access VALUE, the value of A is added to the base address of the block of storage allocated for external dummy sections. Q-type address constants may not be specified in literals.

```
A          DC          Q(VALUE)
```

**Note:** The DXD or DSECT names referenced in the Q-type address constant need not be previously defined.

Subfield	Value	Example	Result
1. <u>Duplication factor</u>	Allowed		
2. <u>Modifiers</u> Implicit length: (length modifier not present)	4 bytes		
Alignment: (Length modifier not present)	Fullword		
Range for length:	1 to 4 bytes (no bit length)		
Range for scale	Not allowed		
Range for exponent	Not allowed		
3. <u>Nominal value</u> Represented by:	A single relocatable symbol	DC Q(DUMMYEXT) DC V(DXDEXT)	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	At left		

Figure 36. Q Address Constants

**Floating-Point Constants—E, D, and L:** Floating-point constants allow you to 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 performed.
- 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 Figure 37 on page 124)—plus is assumed if the number is unsigned—integer, fraction, or mixed number (see (2) in Figure 37) followed by an exponent (positive or negative). The exponent (see (3) in Figure 37) 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.

The format of the constant is shown in Figure 38 on page 125.



The value of the constant is represented by two parts:

- An exponent portion (see (1) in Figure 38 on page 125), followed by
- A fractional portion (see (2) in Figure 38).

A sign bit (see (3) in Figure 38) 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 proper internal format. The quantity expressed is the product of the fraction (see (4) in Figure 38) and the number 16 raised to a power (see (5) in Figure 38). Figure 38 shows the external format of the three types of floating-point constants.

The range of values that can be assembled into floating-point constants is given below:

Type of Constant	Range of Magnitude (M) of Values (Positive and Negative)
E	$16^{-65} \leq M \leq (1-16^{-6}) \times 16^{63}$
D	$16^{-65} \leq M \leq (1-16^{-14}) \times 16^{63}$
L	$16^{-65} \leq M \leq (1-16^{-28}) \times 16^{63}$
E,D,L	$5.4 \times 10^{-79} \leq M \leq 7.2 \times 10^{75}$ (approximate)

If the value specified for a particular constant does not lie within these ranges, the constant is not assembled, but is flagged as an error.

Subfield	Value	Example
1. <u>Duplication factor</u>	Allowed	
2. <u>Modifiers</u>	E-type: 4 bytes D-type: 8 bytes L-type: 16 bytes	
Implicit length: (length modifier not present)		
Alignment: (Length modifier not present)	E-type: Fullword D-type: Doubleword L-type: Doubleword	
Range for length:	E-type: 1 to 8 (byte length) .1 to .64 (bit length)	
	D-type: 1 to 8 (byte length) .1 to .64 (bit length)	
	L-type: 1 to 16 (byte length) .1 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	
3. <u>Nominal value</u>		
Represented by:	Decimal digits (0 to 9)	E-type: DC E' + 525' <sup>1</sup> DC E'5.25' <sup>2</sup>
		D-type: DC D'-525' <sup>1</sup> DC D' + .001' <sup>2</sup>
		L-type: DC L'525' DC L'3.414' <sup>2</sup>
Enclosed by:	Apostrophes	
Exponent allowed:	Yes	E-type: DC E'1E+60' <sup>3</sup>
		D-type: DC D'-2.5E10' <sup>3</sup>
		L-type: DC L'3.712E-3' <sup>3</sup>
Number of values per operand:	Multiple	
Padding:	With hexadecimal zeros at left	
Truncation of assembled value:	Not applicable (values are rounded)	

Figure 37. Floating-Point Constants

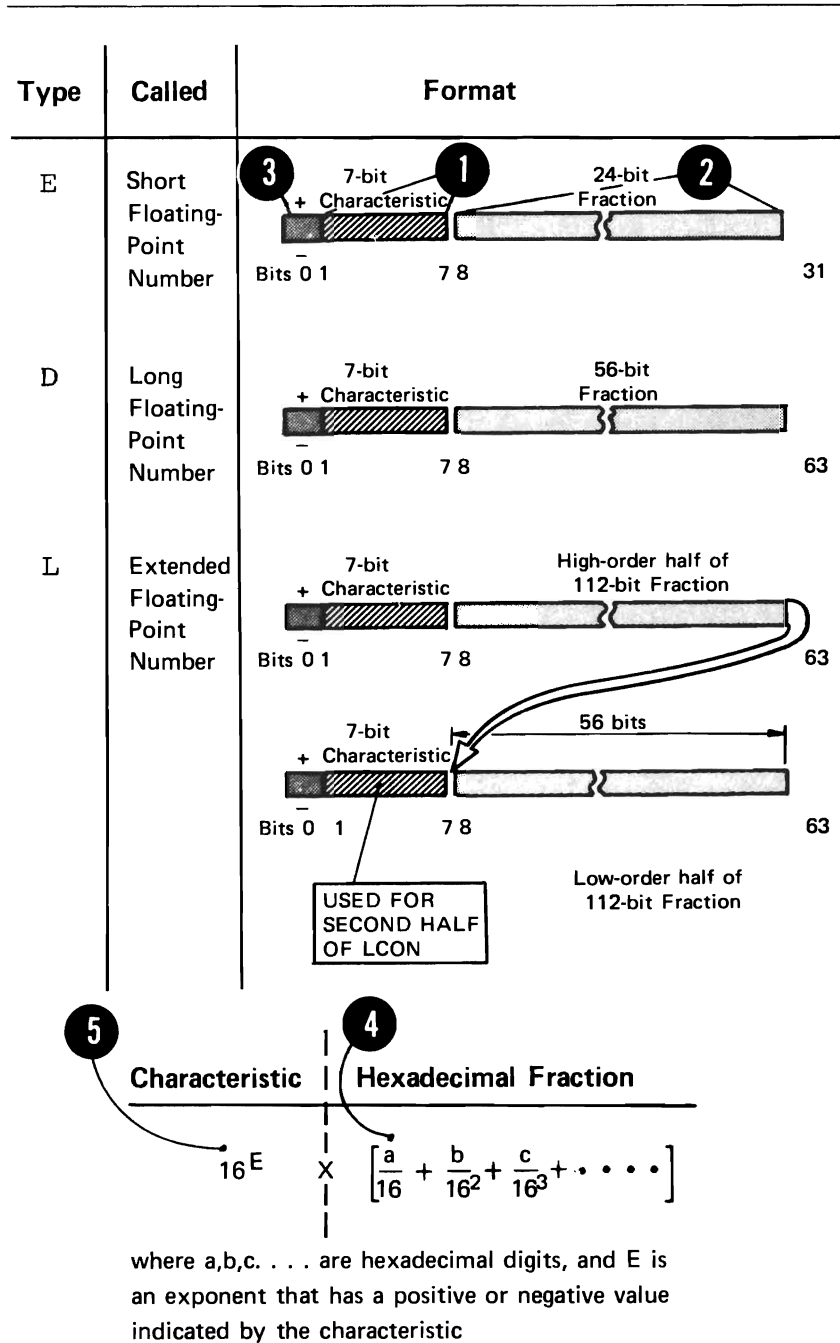


Figure 38. Floating-Point External Formats

**Binary Representation:** 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 when the value of the characteristic between +127 and +64 represents the exponents of 16 between +63 and 0 (by subtracting 64), and the value of the characteristic between +63 and 0 represents the exponents of 16 between -1 and -64.

**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).

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 3: Modifiers" on page 102.)
3. Rounding of the fraction is performed according to the implied or explicit length of the constant. The resulting number will not differ from the exact value specified by more than one in the last place.
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 8 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 proper word (E) or doubleword (D and L) boundary if a length is not specified. However, any length up to and including 8 bytes (E and D) or 16 bytes (L) can be specified by a length modifier. In this case, no boundary alignment occurs.

Any of the following statements could be used to specify 46.415 as a positive, fullword, floating-point constant; the last is a machine instruction statement with a literal operand. Note that 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 would each be generated as doubleword floating-point constants.

FLOAT	DC	DE+4'+46,-3.729,+473'
-------	----	-----------------------

## DS—Define Storage

The DS instruction allows you to:

- Reserve areas of storage
- Provide labels for these areas
- Use these areas by referring to the symbols defined as labels

The DS instruction causes no data to be assembled. Unlike the DC instruction, you do not have to specify the nominal value (fourth subfield) of a DS instruction operand. Therefore, the DS instruction is the best way of symbolically defining storage for work areas, input/output buffers, etc.

### Format of DS:

---

Name	Operation	Operand
Any symbol or blank	DS	One or more operands, separated by commas, written in the format described in the following text

---

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. Although the formats are identical, there are two differences in the specification of subfields. They are:

1. 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.
2. 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.

The label used in the name entry of a DS instruction, as with the label for a DC instruction:

- Has an address value of the leftmost byte of the area reserved, after any boundary alignment is performed
- 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.

**Note:** 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 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, which is smaller.

## How to Use the DS Instruction

**To Reserve Storage:** If you want to take advantage of automatic boundary alignment (if the ALIGN option is specified) and implicit length calculation, you should not supply a length modifier in your operand specifications. You should specify a type subfield that corresponds to the type of area you need for your instructions.

**Note:** Duplication has no effect on implicit length.

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 will not be aligned automatically according to their type. If you omit the nominal value in the operand, you should use a length modifier for the binary (B), character (C), graphic (G), hexadecimal (X), and decimal (P and Z) type areas; otherwise, their labels will be given a length attribute value of 1 (2 for G-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 label. You can also use the character (C) and hexadecimal (X) field types to specify large areas using the length modifier.

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 will be placed in the area at execution time.

**To Force Alignment:** You can use the DS instruction to force alignment to a boundary that otherwise would not be provided. You can force the location counter to a doubleword, fullword, or halfword boundary by using the appropriate field type (for example, D, F, or H) with a duplication factor of zero. No space is reserved for such an instruction, yet the data that follows is aligned on the desired boundary. For example, the following statements would set the location counter to the next doubleword boundary and reserve storage space for a 128-byte field (whose leftmost byte would be on a doubleword boundary).

```
                DS          0D
AREA            DS          CL128
```

**Note:** Alignment is forced when either the ALIGN or the NOALIGN assembler option is set.

**To Name Fields of an Area:** Using a duplication factor of zero in a DS instruction also allows you to provide a label for an area of storage without actually reserving the area. You can 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—Identify Dummy Section" on page 61.) The whole area is addressable by its label. In addition, the symbolic label will have 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 illustrates 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. Note that the first statement names the entire 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 actually 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

```

Additional examples of DS statements are shown below:

ONE	DS	CL80	One 80-byte field, length attribute of 80
TWO	DS	80C	80 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	80 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 would be pertinent when using FIELD as an SS machine instruction operand.

## CCW or CCW0—Define Channel Command Word (Format 0)

You can use the CCW or CCW0 instruction to define and generate an 8-byte channel command word aligned at a doubleword boundary for input/output operations. The CCW and CCW0 instructions have identical functions; however, the CCW0 instruction is not included in the S/370 instruction set. A CCW or CCW0 will cause any bytes skipped to be zeroed. A CCW or CCW0 instruction will result in a Format 0 channel command word which allows 24-bit data addresses. The internal machine format of a channel command word is shown in Figure 39.

Byte	Bits	Usage
0	0-7	Command code
1-3	8-31	Address of data to operate upon
4	32-37	Flags
	38-39	Must be specified as zeros
5	40-47	Set to zeros by assembler
6-7	48-63	Byte count or length of data

Figure 39. Channel Command Word, Format 0

### Format of CCW and CCW0:

Name	Operation	Operand
Any symbol or blank	CCW or CCW0	Command code, data address, flags, data count

All four operands must appear. They are written, from left to right, as follows:

1. An absolute expression that specifies the command code. This expression's value is right-justified in byte 0.
2. A relocatable or absolute expression specifying 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-justified in bytes 1 to 3.
3. An absolute expression that specifies the flags for bits 32 to 37, and zeros for bits 38 and 39. The value of this expression is right-justified in byte 4. (Byte 5 is set to zero by the assembler.)
4. An absolute expression that specifies the byte count or length of data. The value of this expression is right-justified in bytes 6 and 7.

The generated channel command word is aligned on a doubleword boundary. Any bytes skipped are set to zero.

The symbol in the name field, if present, is assigned the value of the address of the leftmost byte of the channel command word generated. The length attribute value of the symbol is 8.

The following are examples of CCW and CCW0 statements:

```
WRITE1  CCW      1,DATADR,X'48',X'50'  
WRITE2  CCW0    1,DATADR,X'48',X'50'
```



The object code generated (in hexadecimal) for either of the above examples is:

```
01 xxxxxx 48 00 0050
```

where xxxxxx contains the address of DATADR, and DATADR must reside below 16 megabytes.

**Notes:**

1. If you use the EXCP access method, you must use CCW or CCW0, because EXCP does not support 31-bit data addresses in channel command words.
2. You should use RMODE 24 with CCW or CCW0 to ensure that valid data addresses are generated in the channel command words at execution time.

### CCW1—Define Channel Command Word (Format 1)

You can use the CCW1 instruction to specify the object code format to be used for an 8-byte channel command word aligned at a doubleword boundary for input/output operations. The object code for a Format 1 channel command word allows a 31-bit data address, whereas the object code generated by a CCW or CCW0 instruction allows only a 24-bit data address. A CCW1 will cause any bytes skipped to be zeroed. The internal machine format of a channel command word is shown in Figure 40.

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

Figure 40. Channel Command Word, Format 1

**Format of CCW1:**

Name	Operation	Operand
Any symbol or blank	CCW1	Command code, data address, flags, data count

All four operands must appear. They are written, from left to right, as follows:

1. An absolute expression that specifies the command code. This expression's value is right-justified in byte 0.
2. An expression specifying the data address. This value is treated as a 4-byte, A-type address constant. The value of this expression is in bytes 4 to 7, the first bit of which is set to 0.
3. An absolute expression that specifies the flags for bits 8 to 15. The value of this expression is right-justified in byte 1.
4. An absolute expression that specifies the count. The value of this expression is right-justified in bytes 2 and 3.

**Note:** The expression for the data address should be such that the address is within the range 0 to  $2^{31}-1$ , inclusive, after possible relocation. This will be the case if the expression refers to a location within one of the control sections which will be link-edited together. An expression such as \*-1000000000 will yield an acceptable value only when the command control word is placed in storage location 1000000000 or higher.

The generated channel command word is aligned on a doubleword boundary. Any bytes skipped are set to zero.

The symbol in the name field, if present, is assigned the value of the address of the leftmost byte of the channel command word generated. The length attribute value of the symbol is 8.

The following is an example of a CCW1 statement:

```
A          CCW1          X'0C',BUF1,X'00',L'BUF1
```

The object code (in hexadecimal) generated in the above example is:

```
0C 00 yy xxxxxxxx
```

where yy is length of BUF1, and xxxxxxxx is BUF1 address.

**Note:** BUF1 can reside anywhere in virtual storage.

---

## Program Control Instructions

You use the program control instructions to:

- Specify the end of an assembly
- Set the location counter to a value or word boundary
- Insert previously written coding in the program
- Specify the placement of literals in storage
- Check the sequence of input cards
- Indicate statement format
- Punch a card

Except for the CNOP and COPY instructions, none of these assembler instructions generate instructions or constants in the object program.

### ICTL—Input Format Control

The ICTL instruction allows you to change the begin, end, and continue columns that establish the coding format of the assembler language source statements.

For example, with the ICTL instruction, you can 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.

You can use the ICTL instruction only once, at the very beginning of a source program. If you do not use it, the assembler recognizes the standard values for the begin, end, and continue columns.

## Format of ICTL:

---

Name	Operation	Operand
Blank	ICTL	1-3 decimal self-defining terms of the form b or b,e or b,e,c

---

The operand entry must be one to three decimal self-defining terms. There are only three possible ways of specifying the operand entry:

1. The operand b specifies the begin column of the source statement. It must always be specified, and must be within the range of 1 to 40, inclusive.
2. The operand e specifies the end column of the source statement. The end column, when specified, must be within the range of 41 to 80, inclusive; when not specified, it is assumed to be 71.
3. The operand c specifies the continue column of the source statement. The continue column, when specified, must be within the range of 2 to 40. If the continue column is not specified, or if column 80 is specified as the end column, the assembler assumes that continuation lines are not allowed.

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, respectively.

The values specified for the three operands depend on each other. Two rules governing the interaction of b, e, and c are:

1. The position of the end column must not be less than the position of the begin column + 5, but must be greater than the position of the continue column.
2. The position of the continue column must be greater than that of the begin column.

The next example designates the begin column as 25. Since the end column is not specified, it is assumed to be column 71. No continuation cards are recognized because the continue column is not specified.

```
ICTL          25
```

**Note:** 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 10.

## ISEQ—Input Sequence Checking

You can use the ISEQ instruction to cause 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.

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 terminated.

**Note:** If the sequence field in the preceding line is blank, the assembler uses the last preceding line with a nonblank sequence field to make its comparison.

#### Format of ISEQ:

---

Name	Operation	Operand
Blank	ISEQ	Two decimal self-defining terms of the form 1,r or blank

---

If the operand is two decimal self-defining terms, then ISEQ initiates sequence checking, beginning at the statement or line following the ISEQ instruction. Checking begins at the column represented by 1 and ends at the column represented by r. The second option of the ISEQ format terminates the sequence checking operation.

If the operand is a blank, ISEQ terminates the operation. (Note that this ISEQ statement is also sequence checked.) Checking may be resumed with another ISEQ statement.

The rules for interaction are:

1. 1 specifies the leftmost column of the field to be checked, and r specifies the rightmost column of the field to be checked. r must be greater than or equal to 1.
2. 1 and r can be anywhere on the cards in the input. Thus, they can also be between the begin and end columns.

**Note:** 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.

However, the assembler does not check:

1. Statements inserted by a COPY instruction
2. 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 Prepare Macro Definitions" on page 159)
3. Statements in library macro definitions

## PUNCH—Punch a Card

The PUNCH instruction allows you to punch source or other statements into a single card.

Code PUNCH statements in:

- a source module to produce control statements for the linkage editor. The linkage editor uses these control statements to process the object module.
- macro definitions to produce, for instance, source statements in other computer languages or for other processing phases.

The card that is punched has a physical position immediately after the PUNCH instruction and before any other TXT cards of the object decks that are to follow.

The PUNCH instruction causes the data in its operand to be punched into a card. One PUNCH instruction produces one punched card, but as many PUNCH instructions as necessary can be used.

The PUNCH instruction statement can appear anywhere in a source module except before and between source macro definitions. If a PUNCH instruction occurs before the first control section, the resultant card punched will precede all other cards in the object deck.

The cards punched as a result of a PUNCH instruction are not a logical part of the object deck, even though they can be physically interspersed in the object deck.

**Format of PUNCH:**

---

Name	Operation	Operand
A sequence symbol or blank	PUNCH	A character string of up to 80 characters, enclosed in apostrophes

---

All 256 punch combinations of the IBM System/370 character set are allowed in the character string of the operand field. Variable symbols are also allowed.

The position of each character specified in the PUNCH statement corresponds to a column in the card to be punched. However, the following rules apply to ampersands and apostrophes:

1. A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.
2. Double ampersands or apostrophes are punched as single ampersands or apostrophes.
3. A single apostrophe followed by one or more blanks simply terminates the string of characters punched. If a nonblank character follows a single apostrophe, an error message is issued and nothing is punched.

Only the characters punched, including blanks, count toward the maximum of 80 allowed.

**Notes:**

1. No sequence number or identification is punched into the card produced.
2. If the NODECK option is specified when the assembler is invoked, no cards are punched, neither for the PUNCH or REPRO instructions, nor for the object deck of the assembly.
3. Double-byte data is permissible in the operand field when the DBCS assembler option is specified. The double-byte data must be valid.
4. The DBCS ampersand and apostrophe are not recognized as delimiters.
5. A double-byte character which contains the value of an EBCDIC ampersand or apostrophe in either byte is not recognized as a delimiter when enclosed by SO and SI.

**REPRO—Reproduce Following Card**

The REPRO instruction causes the data specified in the statement that follows to be punched into a card. Unlike the PUNCH instruction, the REPRO instruction does not allow values to be substituted into variable symbols before the card is punched. One REPRO instruction produces one punched card.

The REPRO instruction can appear anywhere in a source module except before and between source macro definitions. The punched cards are not part of the object deck, even though they can be physically interspersed in the object deck.

**Format of REPRO:**

---

Name	Operation	Operand
A sequence symbol or blank	REPRO	Not required

---

The line to be reproduced can contain any of the 256 punch characters, including blanks, ampersands, and apostrophes. No substitution is performed for variable symbols.

**Notes:**

1. Sequence numbers and identification are not punched in the card.
2. If the NODECK option is specified in the job control language for the assembler program, no cards are punched: neither for the PUNCH or REPRO instructions, nor for the object deck of the assembly.
3. 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 will not be validated.

## PUSH Instruction

The PUSH instruction allows you to save the current PRINT or USING status in “push-down” storage on a last-in, first-out basis. You can restore this PRINT and USING status later, also on a last-in, first-out basis, by using a corresponding POP instruction.

### Format of PUSH:

---

Name	Operation	Operand
A sequence symbol or blank	PUSH	PRINT or USING or PRINT,USING or USING,PRINT

---

One of the four options for the operand entry must be specified. The PUSH instruction does not change the status of the current PRINT or USING instructions; the status is only saved.

**Note:** When the PUSH instruction is used in combination with the POP instruction, a maximum of four nests of PUSH PRINT - POP PRINT or PUSH USING - POP USING are allowed.

## POP Instruction

The POP instruction allows you to restore the PRINT or USING status saved by the most recent PUSH instruction.

### Format of POP:

---

Name	Operation	Operand
A sequence symbol or blank	POP	PRINT or USING or PRINT,USING or USING,PRINT

---

One of the four options for the operand entry must be specified. The POP instruction causes the status of the current PRINT or USING instruction to be overridden by the PRINT or USING status saved by the last PUSH instruction.

**Note:** When the POP instruction is used in combination with the PUSH instruction, a maximum of four nests of PUSH PRINT - POP PRINT or PUSH USING - POP USING are allowed.

## ORG—Set Location Counter

You use the ORG instruction to alter the setting of the location counter and thus control the structure of the current control section. This allows you to redefine portions of a control section.

Using Figure 41 as an example, if you wish to build a translate table (for example, to convert EBCDIC character code into some other internal code):

- You define the table (see (1) in Figure 41) as being filled with zeros.
- You use the ORG instruction to alter the location counter so that its counter value indicates a desired location (see (2) in Figure 41) within the table.
- You redefine the data (see (3) in Figure 41) to be assembled into that location.
- After repeating the first three steps (see (4) in Figure 41) until your translate table is complete, you use an ORG instruction with a blank operand field to alter the location counter. The counter value then indicates the next available location (see (5) in Figure 41) 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.

In other words, the ORG instruction can cause the location to point to any part of a control section, even the middle of an instruction, into which you can assemble desired data. It can also cause the location counter to point to the next available location so that your program can continue to be assembled in a sequential fashion.



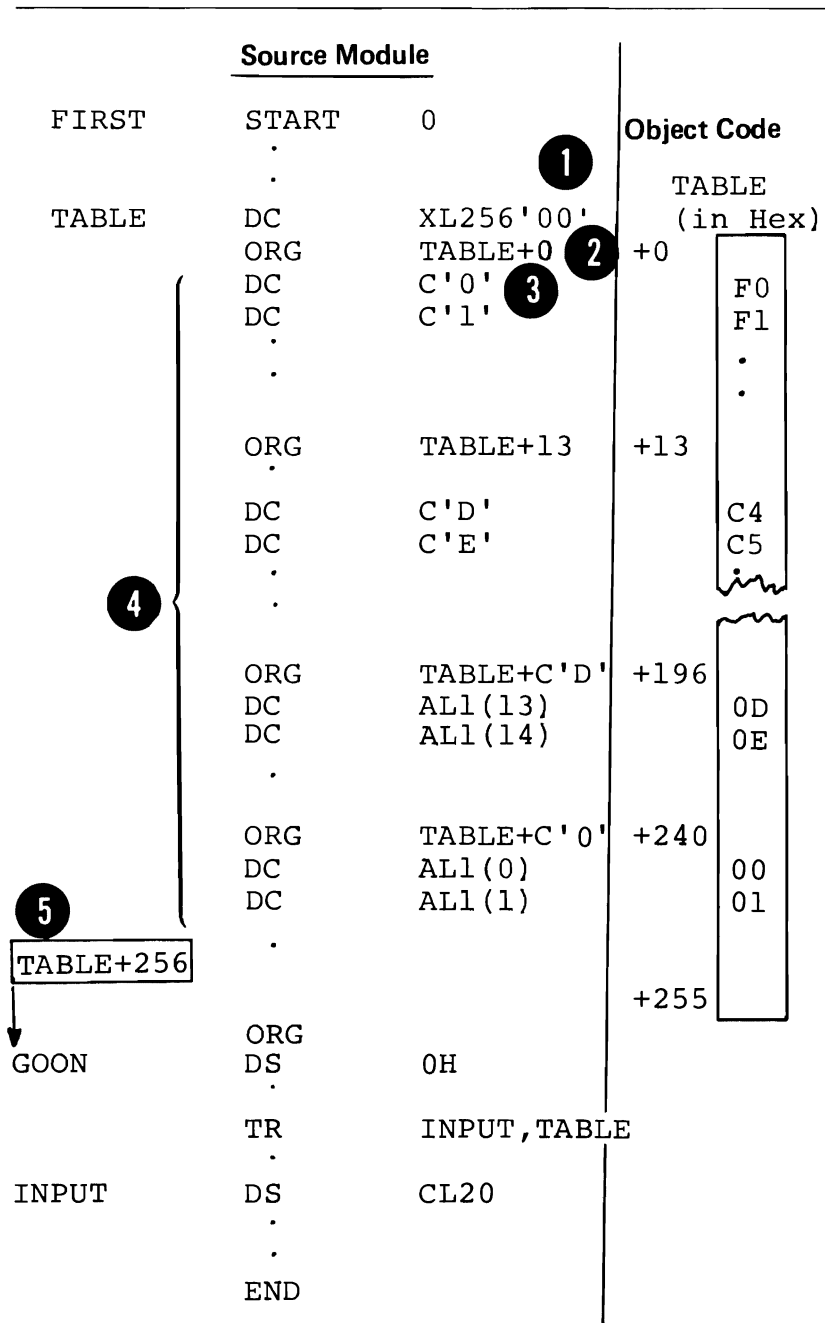


Figure 41. Building a Translate Table

### Format of ORG:

---

Name	Operation	Operand
A sequence symbol or blank	ORG	A relocatable expression or blank

---

In general, symbols used in the operand field need not have been previously defined. However, the relocatable component of the 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.

The location counter is set to the value of the expression in the operand. If the operand is omitted, the location counter is set to the next available location for the current control section.

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 is invalid if it appears less than 500 bytes from the beginning of the current control section.

```
ORG          *-500
```

This is because the expression specified is then negative, and will set the location counter to a value larger than the assembler can process. The location counter will “wrap around” (the location counter is discussed in detail in “Location Counter Reference” on page 32).

**Note:** With the ORG statement, you can give two instructions the same location counter values. In such a case, the second instruction will not always eliminate the effects of the first instruction. Consider the following example:

```
ADDR    DC      A(LOC)
         ORG     *-4
B        DC      C'BETA'
```

In this example, the value of B (BETA) will be destroyed by the relocation of ADDR during linkage editing.

**Restriction on ORG when the LOCTR Instruction is Used:** 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.

### LTORG—Begin Literal Pool

You 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.

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 142).

The creation of a literal pool gives the following advantages:

- Automatic organization of the literal data into sections that are properly aligned and arranged so that no space is wasted.
- 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.

#### Format of LTORG:

---

Name	Operation	Operand
Any symbol or blank	LTORG	Not used

---

If an ordinary symbol is specified in the name field, it represents the first byte of the literal pool; this symbol is aligned on a doubleword boundary and has a length attribute value 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.

## Literal Pool

A literal pool is created immediately after a LTORG instruction or, if no LTORG instruction is specified, at the end of the first control section.

Each literal pool has four 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), as described below.

- The *first segment* contains all literal constants whose assembled lengths are a multiple of 8.
- The *second segment* contains those whose assembled lengths are a multiple of 4, but not of 8.
- The *third segment* contains those whose assembled lengths are even, but not a multiple of 4.
- The *fourth segment* contains all the remaining literal constants whose assembled lengths are odd.

Since each literal pool is aligned on a doubleword boundary, this guarantees that all literals in the first segment are doubleword aligned; in the second segment, fullword aligned; and, in the third, halfword aligned. No space is wasted except, possibly, at the origin of the pool.

Literals from the following statement are in the pool, in the segments indicated by the parenthesized numbers:

FIRST	START	θ	
	.		
	MVC	T0,=3F'9'	(2)
	AD	2,=D'7'	(1)
	IC	2,=XL1'8'	(4)
		,=CL3'JAN'	(4)
		,=2F'1,2'	(1)
		,=H'33'	(3)
		,=A(ADDR)	(2)
		,=XL8'05'	(1)

## Addressing Considerations

If you specify literals in source modules with multiple control sections, you should:

- Write a 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, you should write a LTORG instruction at the end of each segment of the interspersed control section.
- When establishing the addressability of each control section, make sure (a) that the entire literal pool for that section is also addressable, by including it within a USING range, and (b) that the literal specifications are within the corresponding USING domain. The USING range and domain are described in "USING—Use Base Address Register" on page 44.

**Note:** 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 a 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 (or Y-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 is different in the two literals.

The following examples illustrate 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'	the first is stored

## CNOP—Conditional No Operation

You can use the CNOP instruction to align any instruction or other data on a specific halfword boundary. The CNOP instruction ensures an unbroken flow of executable instructions by generating no-operation instructions to fill the bytes skipped to perform the alignment that you specified.

For example, when you code the linkage to a subroutine, you may wish 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. This is illustrated below:

	CNOP	6,8
LINK	BALR	2,10
	CCW	1,DATADR,X'48',X'50'

Assume that the location counter is currently aligned at a doubleword boundary. Then the CNOP instruction causes three branch-on-conditions (no-operations) to be generated, thus aligning the BALR instruction at the last halfword in a doubleword as follows:

	BCR	0,0
	BCR	0,0
	BCR	0,0
	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, thereby 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, or doubleword boundary. It does not affect the location counter if the counter is already properly aligned. If the specified alignment requires the location counter to be incremented, one to three no-operation instructions (BCR 0,0 occupying two bytes each) are generated to fill the skipped bytes. Any single byte skipped to achieve alignment to the first no-operation instruction is filled with zeros.

### Format of CNOP:

Name	Operation	Operand
Any symbol or blank	CNOP	Two absolute expressions of the form b,w

The operands must be absolute expressions, and the symbols in them need not be previously defined. The first operand, b, specifies at which even-numbered byte in a fullword or doubleword the location counter is set. The second operand, w, specifies whether the byte is in a fullword (w = 4) or a doubleword (w = 8).

Valid pairs of b and w are indicated below:

b,w	Specifies
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

Figure 42 shows the position in a doubleword that each of these pairs specifies. Note that both 0,4 and 2,4 specify two locations in a doubleword.

Doubleword							
Fullword				Fullword			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0,4		2,4		0,4		2,4	
0,8		2,8		4,8		6,8	

Figure 42. CNOP Alignment

### COPY—Copy Predefined Source Coding

You use the COPY instruction to obtain source language coding from a library and include it in the programs currently being assembled. You thereby avoid writing the same, often-used sequence of code over and over.

### Format of COPY:

---

Name	Operation	Operand
Blank	COPY	One ordinary symbol

---

The operand is a symbol that identifies a partitioned data set member to be copied from either the system macro library or a user library concatenated to it.

The source coding that is copied into a source module:

- Is inserted immediately after the COPY instruction
- Is 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. Thus, if the statement 'COPY A' is coded, and A contains a statement 'COPY B', B must not contain a statement 'COPY A'.)
- Can contain macro definitions

If a source macro definition is copied into the beginning of a source module, both the MACRO and MEND statements that delimit the definition must be contained in the same level of copied code.

#### 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.

## END—End Assembly

You use the END instruction to terminate the assembly of a program. You can also supply an address in the operand field to which control may be transferred after the program is loaded. The END instruction must always be the last statement in the source program.

### Format of END:

---

Name	Operation	Operand
A sequence symbol or blank	END	A relocatable expression or blank expression or blank

---

The operand specifies the point to which control may be transferred when loading is complete. This point is usually 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, the operand entry can be generated by substitution into variable symbols. However, after substitution, that is, at assembly time:

- It must be a relocatable expression representing an address in the source module delimited by the END instruction, or
- 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.
- It must not be a literal.

---

## Listing Control Instructions

The instructions described in this section request the assembler to produce listings and identify output cards in the object deck according to your special needs. They allow you to determine printing and page formatting options other than the ones the assembler program assumes by default. Among other things, you can introduce your own page headings, control line spacing, and suppress unwanted detail.

### TITLE—Identify Assembly Output

The TITLE instruction allows you to:

- Provide headings for each page of the assembly listing of your source modules.
- Identify the assembly output cards of your object modules. You can specify up to 8 identification characters that the assembler will punch into all the output cards, beginning at column 73. The assembler punches sequence numbers into the columns that are left, up to column 80.

#### Format of TITLE:

---

Name	Operation	Operand
A string of alphanumeric characters, a variable symbol, a combination of above, a sequence symbol, or a blank	TITLE	A character string up to 100 characters, enclosed in apostrophes

---



The first three options for the name field have a special significance only for the first TITLE instruction in which they are specified. For subsequent TITLE instructions, the first three options do not apply.

For the first TITLE instruction of a source module that has a nonblank name entry that is not a sequence symbol, up to 8 alphanumeric characters can be specified in any combination in the name field.

These characters are punched as identification, beginning at column 73, into all the output cards from the assembly, except those produced by the PUNCH and REPRO instructions. The assembler substitutes the current value into a variable symbol and uses the generated result as identification characters.

If a valid ordinary symbol is specified, its appearance in the name field does not constitute a definition of that symbol for the source module. It can, therefore, be used in the name field of any other statement in the same source module.

The character string in the operand field is printed as a heading at the top of each page of the assembly listing. The heading is printed beginning on the page in the listing following the page on which the TITLE instruction is specified. A new heading is printed when a subsequent TITLE instruction appears in the source module.

For example, if the following statement is the first TITLE statement to appear in a program:

```
PGM1      TITLE      'FIRST HEADING'
```

then PGM1 is punched into all of the output cards (columns 73 to 76) and this heading appears at the top of each subsequent page: PGM1 FIRST HEADING.

If the following statement occurs later in the program:

```
          TITLE      'A NEW HEADING'
```

then PGM1 is still punched into the output cards, but each following page begins with the heading: PGM1 A NEW HEADING.

Each inline TITLE statement causes the listing to be advanced to a new page before the heading is printed. If the TITLE statement appears in a macro *and* PRINT NOGEN is specified, the listing will not be advanced to a new page.

Any printable character specified will appear in the heading, including blanks. 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.
- Double ampersands or apostrophes specified, print as single ampersands or apostrophes in the heading.

- A single apostrophe followed by one or more blanks simply terminates the heading prematurely. If a nonblank character follows a single 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.

**Note:** The TITLE statement itself is not printed in an assembly listing.

## EJECT—Start New Page

The EJECT instruction allows you to stop the printing of the assembler listing on the current page, and continue the printing on the next page.

### Format of EJECT:

---

Name	Operation	Operand
A sequence symbol or blank	EJECT	Not required

---

The EJECT instruction causes the next line of the assembly 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 causes a blank page in the listing.

**Note:** The EJECT instruction statement itself is not printed in the listing.

## SPACE—Space Listing

You can use the SPACE instruction to insert one or more blank lines in the listing of a source module. This allows you to separate sections of code on the listing page.

### Format of SPACE:

---

Name	Operation	Operand
A sequence symbol or blank	SPACE	A decimal self-defining term or blank

---

The operand entry specifies the number of lines to be left blank. A blank operand entry causes one blank line to be inserted. A blank operand causes one blank line to be inserted. If the operand specified has a value greater than the number of lines remaining on the listing page, the instruction will have the same effect as an EJECT statement.

**Note:** The SPACE instruction itself is not printed in the listing.

## PRINT—Print Optional Data

The PRINT instruction allows you to control the amount of detail you wish printed in the listing of your programs. The three options that you can set are given in the table below:

Hierarchy	Description	Options
1	A listing is printed.	ON
	No listing is printed.	OFF
2	All statements generated by the processing of a macro instruction are printed.	GEN
	Statements generated by the processing of a macro instruction are not printed. (The MNOTE instruction always causes a message to be printed.)	NOGEN
3	Constants are printed in full in the listing.	DATA
	Only the leftmost 8 bytes of constants are printed in the listing.	NODATA

The options are listed in hierarchic order; if OFF is specified, GEN and DATA will not apply. If NOGEN is specified, DATA will not apply to constants that are generated. The standard options inherent in the assembler program are ON, GEN, and NODATA.

### Format of PRINT:

Name	Operation	Operand
A sequence symbol or blank	PRINT	[ON OFF] [,GEN NOGEN] [,NODATA DATA]

**Note:** Any sequence of specification is allowed.

At least one of the operands must be specified, and at most one of the options from each group. The PRINT instruction can be specified any number of times in a source module, but only those print options actually specified in the instruction change the current print status.

PRINT options can be generated by macro processing, at preassembly time. 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 137 and “POP Instruction” on page 137, also influence the PRINT options by saving and restoring the PRINT status.

**Note:** The option specified in a PRINT instruction takes effect after the PRINT instruction. If PRINT OFF is specified, the PRINT instruction itself is printed, but not the statements that follow it. If the NOLIST assembler option is specified when the assembler is invoked, the entire listing for the assembly is suppressed.



---

## Part 2. Macro Language

- “Chapter 6. Introduction to Macro Language” describes the macro instruction statement, definition, library, and so on.
- “Chapter 7. How to Prepare Macro Definitions” and “Chapter 8. How to Write Macro Instructions” describe the basic rules for preparing macro definitions and for writing macro instructions.
- “Chapter 9. How to Write Conditional Assembly Instructions” describes the rules for writing conditional assembly instructions.



---

## 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 desired 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 desired 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 desired functions.

In addition, conditional assembly allows you to code statements that may or may not be assembled, depending upon conditions evaluated at assembly time. These conditions are usually tests of values which may be defined, set, changed, and tested during assembly. Conditional assembly can be used without using macro instruction statements.

---

### 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. The assembler expands a macro at preassembly time.

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 usually 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 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 perform only internal processing.

A macro definition provides the assembler with (1) the name of the macro, (2) the parameters used in the macro, and (3) 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 43.

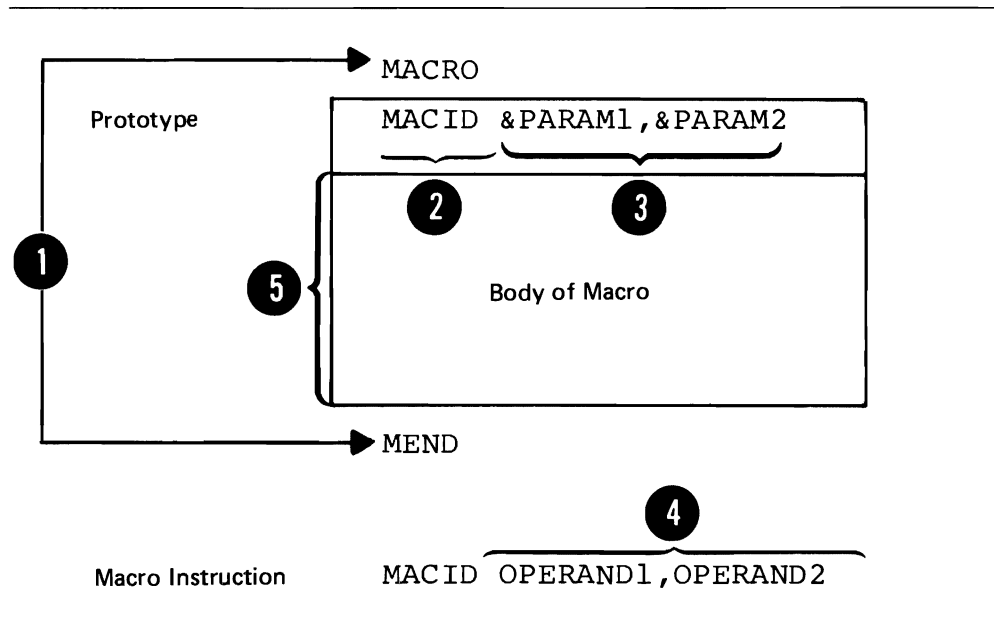


Figure 43. Parts of a Macro Definition

- 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 43).
- The macro instruction prototype statement is used to name the macro (see (2) in Figure 43), and to declare its parameters (see (3) in Figure 43). In the operand field of the macro instruction, you can assign values (see (4) in Figure 43) to the parameters declared for the called macro definition.
- The body of a macro definition (see (5) in Figure 43) contains the statements that will be generated when you call the macro. These statements are called model statements; they are usually interspersed with conditional assembly statements or other processing statements.



## Model Statements

You can also write assembler language statements as model statements. When it expands the macro, 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 will enter 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 perform functions at preassembly time when macros are expanded, but they are not themselves generated for further processing at assembly time. The processing statements are:

- Conditional assembly instructions
- Inner macro calls
- MNOTE instructions
- MEXIT instructions
- AREAD instructions

The MNOTE instruction allows you to generate an error message with an error condition code attached, or to generate comments in which you can display the results of preassembly computation.

The MEXIT instruction tells the assembler to stop processing a macro definition. The MEXIT instruction, therefore, provides an exit from the middle of a macro definition.

The MEND instruction not only delimits the contents of a macro definition, but also provides an exit from the definition.

The AREAD instruction allows you to assign to a SETC symbol the character string value of a statement that is placed immediately after a macro instruction.

## Comments Statements

One type of comments statement describes preassembly operations and is not generated. The other type describes assembly-time operations and is, therefore, generated.

---

## Macro Instruction Statement

A macro instruction statement (hereafter called 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*.

You can also insert a macro definition into a system or user library (located, for example, on disk) by using the appropriate 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.

Source and library macros are expanded in the same way, but syntax errors are handled differently. In source macros, error messages are attached to the statements in error. In library macros, however, error messages cannot be associated with the statement in error, because these macros are located and edited after the entire source module has been read. Therefore, the error messages are associated with the END statement.

Because of the difficulty of finding syntax errors in library macros, a macro definition should be run and "debugged" as a source macro before it is placed in a macro library.

### Macro Library

The same macro definition may 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. Once a macro definition has been placed in the macro library, it may be used by writing its corresponding macro instruction in a source program. Macro definitions must be in the system macro library under the same name as the prototype. The procedure for placing macro definitions in the macro library is described in the appropriate utilities manual.

## 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 appropriate supervisor services and macro instructions and data management macro instructions manuals.

---

## 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

You can 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.

You can use the functions of the conditional assembly language to select statements for generation, to determine their order of generation, and to perform computations that affect the content of the generated statements.

The conditional assembly language is described in "Chapter 9. How to Write Conditional Assembly Instructions."



---

## Chapter 7. How to Prepare Macro Definitions

Defining a macro means preparing the statements that constitute a macro definition. 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 and a MEND instruction

Except for conditional assembly instructions, this chapter describes all the statements that can be used to prepare macro definitions. Conditional assembly instructions are described in “Chapter 9. How to Write Conditional Assembly Instructions” on page 211.

### 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. Thus, you can redefine a macro at any point in your program. The new definition will be used for all subsequent calls to the macro in the program.

This type of macro definition is called a source 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 appropriate 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.

#### Example:

```
MACRO                                Macro header for outer macro
OUTER                                Macro prototype
    AIF                                (&A,&C=
    MACRO                                ('&C' EQ '').A
    INNER                                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 44. The four parts are described in detail below:

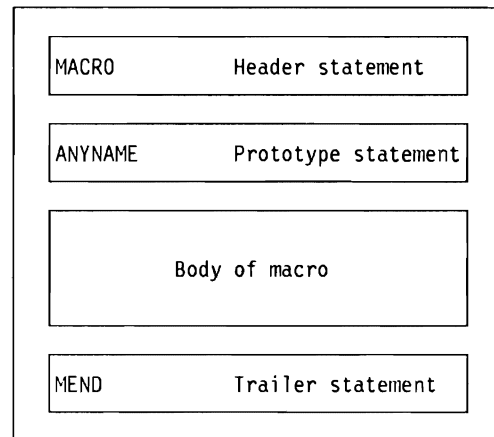


Figure 44. Format of a Macro Definition

---

## MACRO—Macro Definition Header

MACRO indicates the beginning of a macro definition. It must be the first statement in every macro definition.

### Format of MACRO:

---

Name	Operation	Operand
	MACRO	

---

---

## MEND—Macro Definition Trailer

MEND indicates 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.

### Format of MEND:

---

Name	Operation	Operand
A sequence symbol or blank	MEND	

---

---

## Macro Instruction Prototype

The macro instruction prototype statement (hereafter 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 noncomment statement in every macro definition. Only internal comments statements are allowed between the macro header and the macro prototype. Internal comments statements are listed only with the macro definition.

### Format of the prototype statement:

---

Name	Operation	Operand
A name field parameter or blank	A symbol (mandatory)	Zero or more symbolic parameters separated by commas

---

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 170.

## Name Field

You can write a name field parameter, similar to 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 used, the name entry must be a variable symbol. If this parameter also appears in the body of a macro, it will be given the value assigned to the parameter in the name field of the corresponding macro instruction. Note that the value assigned to the name field parameter has special restrictions that are listed in "Formatting Specifications" on page 15.

## Operation Field

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, or is the operation code of a library macro.

Macros that are defined inline may use any ordinary symbol for the operation field. However, operating system rules may prevent some of these macros from being stored as member names.

**Note:** The assembler requires that the member name and macro name be the same; otherwise, an error occurs.

## Operand Field

The operand field in a prototype statement allows you to 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 240 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 blanks, remarks are allowed.

The following is an example of a prototype statement:

```
&NAME      MOVE          &TO,&FROM
```

## Alternative Ways of Coding 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 following examples illustrate (1) the normal statement format, (2) the alternative statement format, and (3) a combination of both statement formats.

Name	Operation	Operand	Comment	Cont.
NAME1	OP1	&OPERAND1,&OPERAND2,&OPERAND3	This is the normal statement format	X
NAME2	OP2	&OPERAND1, &OPERAND2	This is the alternative statement format	X X
NAME3	OP3	&OPERAND1, &OPERAND2,&OPERAND3,&OPERAND4	This is a combination of both	X X



**Notes:**

1. Any number of continuation lines are allowed. However, each continuation line must be indicated by a nonblank character in the column after the end column on the preceding card.
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 will be considered to contain remarks.
3. The standard value for the continue column is 16, and, for the column after the end column, is 72.
4. A comma is required after each parameter except the last.
5. One or more blanks is required between the operand and the remarks.
6. If the assembler is invoked with the DBCS option, the continuation features outlined in “Continuation of double-byte data” on page 12 apply to continuation in the macro language. Extended continuation may be 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:

1. Model statements to be generated
2. Processing statements that, for example, can alter the content and sequence of the statements generated or issue error messages
3. Comments statements, some of which are generated and others which are not
4. 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. Numbers 1 through 3 in the list above are the three main types of statements allowed in the body of a macro. The body of a macro definition can be empty, that is, contain no statements.

**Note:** You can include macro definitions in the body of a macro definition. This is explained under “Using a Macro Definition” in this chapter.

---

## Model Statements

Model statements are statements from which assembler language statements are generated at preassembly time. They allow you to 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 use model statements into which you substitute values in open code.

A model statement consists of one or more fields, separated by one or more blanks, 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 alphameric and special characters
- A variable symbol as a point of substitution
- Any combination of ordinary character strings and variable symbols to form a concatenated string.

The statements generated at preassembly time from model statements must be valid machine or assembler instructions, but must not be conditional assembly instructions. They must obey the coding rules described in “Rules for Model Statement Fields” on page 168 or they will be flagged as errors at assembly time.

**Examples:**

LABEL	L	3,AREA
LABEL	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 (&SYSLIST, &SYSNDX, &SYSECT, &SYSPARM, &SYSDATE, &SYSLOC, and &SYSTIME)
- SET symbols (global or local SETA, SETB, or SETC symbols)

**Examples:**

```
&PARAM(3)
&SYSLIST(1,3)
&SYSLIST(2)
&SETA(10)
&SETC(15)
```

**Note:** Symbolic parameters, SET symbols, and the system variable symbol, &SYSLIST, can all be subscripted. The remaining system variable symbols (&SYSNDX, &SYSECT, &SYSPARM, &SYSDATE, &SYSLOC, and &SYSTIME) cannot be subscripted.

## 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 will be moved to the right with one blank 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 value substituted in the operation field of a macro-generated statement contains leading blanks, the blanks are ignored.
- If the value substituted in the operation field of a model statement in open code contains leading blanks, the blanks will be used to move the field to the right.
- If the value substituted in the operand field contains leading blanks, the blanks will be used to move the field to the right.
- If the value substituted contains trailing blanks, the blanks are ignored.

**Listing of generated fields containing double-byte data:** If the assembler is invoked with the DBCS option, then the following differences apply:

- Any continuation indicators present in the model statement will be discarded.
- Double-byte data that must be split at a continuation point will always be made readable on a device capable of presenting DBCS characters—SI and SO will be inserted at the break point, and the break-point will always occur between double-byte characters.
- The continuation indicator will be 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 so used is 3.
- If continuation is required and the character to the left of the continuation indicator is X, then + will be used as the continuation indicator so as to clearly distinguish the position of the end column. This will apply to any generated field, regardless of its contents, in order to prevent ambiguity.
- Redundant SI/SO pairs may be present in a field after substitution. If they occur at a continuation point, the assembler will not distinguish them from SI and SO inserted by the assembler to preserve readability. You must refer to the object code to resolve this ambiguity.

## 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 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 alphameric character is to follow a variable symbol.
- (2) A left parenthesis that does not enclose a subscript is to follow 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) The concatenation character must not be used between a variable symbol and its subscript; otherwise, the characters will be considered a concatenated string and not a subscripted variable symbol.

Figure 45 on page 167, in which the circled numbers correspond to the numbers in the above list, gives the rules for concatenating variable symbols to ordinary character strings.

**Concatenation of fields containing double-byte data:** If the assembler is invoked with the DBCS option, 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.

The following example illustrates 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 .&, will be 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>'
```

Concatenated String	Values to be Substituted		Generated Result
	Variable symbol	Value	
&FIELD.A <b>1</b> &FIELD.A &FIELD.A	&FIELD &FIELD.A	AREA SUM	AREA AREA AREA
&DISP.( <b>2</b> &BASE)	&DISP &BASE	100 10	100(10)
Concatenation character is not generated			
DC D'&INT. <b>4</b> &FRACT' DC D'&INT&FRACT' DC D'&INT. <b>7</b> &FRACT' optional	&INT &FRACT	99 88	DC D'99.88' <b>3</b> DC D'9988' DC D'9988'
Concatenation character is not generated			
<b>5</b> FIELD&A &A+&B*3-D <b>6</b> &A&B <b>7</b>	&A &A &B	A A B	FIELD A A+B*3-D AB
&SYM( <b>8</b> &SUBSCR)	&SUBSCR &SYM(10)	10 ENTRY	ENTRY

Figure 45. Rules for Concatenation

## 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. It is also possible to specify a continuation-indicator field, an identification-sequence field, and a field before the begin column, if the appropriate 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 given below.

- Blank
- Ordinary symbol
- Sequence symbol
- Variable symbol
- Any combination of variable symbols and other character strings concatenated together

The generated result must either be a blank or a valid ordinary symbol.

Variable symbols must not be used to generate comments statement indicators (\* or .\*).

**Note:** Restrictions on the name entry 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
  - The following assembler instructions:

AMODE	DSECT	PRINT
CCW	DXD	PUNCH
CCW0	EJECT	PUSH
CCW1	END	RMODE
CNOP	ENTRY	REPRO
COM	EQU	SPACE
COPY	EXTRN	START
CSECT	ISEQ	TITLE
CXD	LTORG	USING
DC	OPSYN	WXTRN
DROP	ORG	MEXIT
DS	POP	MNOTE

**Note:** MNOTE and MEXIT are not model statements; they are described in “Chapter 7. How to Prepare Macro Definitions.”

- A variable symbol
- A combination of variable strings concatenated together

Operation code ICTL is not allowed inside a macro definition. The MACRO and MEND operation codes 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 performed for the variable symbols in the statement line 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 listed above, except COPY, ISEQ, REPRO, and MEXIT

The generated operation code must not be an operation code for the following (or their OPSYN equivalents):

- A macro instruction
- A conditional assembly instruction
- The following assembler instructions: COPY, ICTL, ISEQ, MACRO, MEND, MEXIT, and REPRO

**Operand Field:** The entries allowed in the operand field of a model statement, before generation, are given below:

- Blank (if valid)
- An ordinary symbol
- A character string, combining alphameric and special characters (but not variable symbols)
- A variable symbol
- A combination of variable symbols and other character strings concatenated together
- If the assembler is invoked with the DBCS option, character strings may contain double-byte data, provided the character strings are enclosed by apostrophes.

The allowable results of generation are a blank (if valid) and a character string that represents a valid assembler or machine instruction operand field.

**Note:** Variable symbols must not be used in the operand field of a COPY, ICTL, or ISEQ instruction.

**Remarks Field:** The remarks field of a model statement can contain any combination of characters. No substitution is performed for variable symbols appearing in the remarks field. Only generated statements will be printed in the listing.

**Note:** One or more blanks must be used in a model statement to separate the name, operation, operand, and remarks fields from each other. Blanks cannot be generated between fields in order to create a complete assembler language statement. The exception to this rule is that a combined operand-remarks field can be generated with one or more blanks to separate the two fields.

---

## Symbolic Parameters

Symbolic parameters allow you to pass 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 alphameric characters.

The following are valid symbolic parameters:

&READER	&LOOP2
&A23456	&N
&X4F2	&\$4

The following are invalid symbolic parameters:

CARDAREA	first character is not an ampersand
&256B	first character after ampersand is not a letter
&BCD%34	contains a special character other than initial ampersand
&IN AREA	contains a special character [the blank] other than initial ampersand

Symbolic parameters have a local scope; that is, the 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 for every call on that definition.

**Note:** 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" in this chapter, and to local and global SET symbols described in "SET Symbols" on page 212.



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.

The following 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

In the following macro instruction that calls the above macro, the characters HERE, FIELD A, and FIELD B of the MOVE macro instruction correspond to the symbolic parameters &NAME, &TO, and &FROM, respectively, of the MOVE prototype statement.

```
HERE      MOVE          FIELD A, FIELD B
```

If the preceding macro instruction were used in a source program, the following assembler language statements would be generated:

```
HERE      ST          2,SAVE
           L           2,FIELD B
           ST          2,FIELD A
           L           2,SAVE
```

## Positional Parameters

You should 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 parameter to have in the proper position in the operand of the calling macro instruction.

For keyword parameters (described below), you must write the entire keyword and the equal sign that precedes the value to be passed. However, if you need a large number of parameters, you should 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.

The general specifications for symbolic parameters, described in "Symbols" on page 25, also apply to positional parameters. Note that the specification for each positional parameter declared in the prototype statement definition must be a valid variable symbol. Values are assigned to the positional parameters by the corresponding positional operands specified in the macro instruction that calls the definition.

## **&SYSECT—Current Control Section**

You can 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.

The local 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 does 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 START, CSECT, DSECT, or COM instruction.
2. The value of the type attribute of &SYSECT (T'&SYSECT) is always U, and the value of the count attribute (K'&SYSECT) is equal to the number of characters assigned as a value to &SYSECT.
3. Throughout the use of a macro definition, the value of &SYSECT may be considered a constant, independent of any CSECT or DSECT statements or inner macro instructions in that definition.

The next example illustrates these rules. In it, statement 8 is the last CSECT, DSECT, or START 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 last CSECT, DSECT, or START 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 CSECT, DSECT, or START 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.

	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
-----			
MAINPROG	CSECT		
	DS	200C	
CSOUT1	CSECT		
	DS	100C	
INA	CSECT		
	DC	A(CSOUT1)	
INB	CSECT		
	DC	A(INA)	
	DC	A(MAINPROG)	
	DC	A(INB)	

Statement 1 is used to generate a CSECT statement for statement 5. This is the last CSECT, DSECT, or START 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.

### &SYSLIST—Macro Instruction Operand

You can 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, 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 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 alone. One of the two following forms must be used as a point of substitution:

1. &SYSLIST(n) may be used to refer to the nth positional operand
2. If the nth operand is a sublist, then &SYSLIST(n,m) may be used to refer to the mth operand in the sublist.

The subscripts *n* and *m* can be any arithmetic expression allowed in the operand of a SETA instruction. The subscript *n* must be greater than or equal to 0. The subscript *m* must be greater than or equal to 1.

When referring to multilevel (nested) sublists in operands of macro instructions, reference to elements of inner sublists can be made using the appropriate number of subscripts for &SYSLIST.

The examples below show the values assigned to &SYSLIST according to the value of its subscripts *n* and *m*.

Macro instruction:		
-----		
NAME	MACALL	ONE,TWO,(3,4,,6),,EIGHT
Point of Substitution in Macro Definition:	Value Substituted:	See note:
-----	-----	-----
&SYSLIST(2)	TWO	
&SYSLIST(3,2)	4	
&SYSLIST(4)	Null	1
&SYSLIST(9)	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,,6)	

**Notes:**

1. If the position indicated by *n* refers to an omitted operand, or refers 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 but &SYSLIST(*n,m*), where *m* is greater than 1, will cause 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 will be 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." One of two forms (N'&SYSLIST or N'&SYSLIST(*n*)) can be used for the number attribute:

- To indicate the number of positional operands specified in a call, you use the form N'&SYSLIST.
- To indicate the number of sublist entries that have been specified in a positional operand, you use the form N'&SYSLIST(n).

**Notes:**

1. For N'&SYSLIST, positional operands are counted if specifically omitted by specifying the comma that would normally have followed the operand.
2. For N'&SYSLIST(n), sublist entries are counted if specifically omitted by specifying the comma that would normally have followed 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.

**Examples:**

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,KEY=(A,B,C)	0
MACSUB	0

**&SYSNDX—Macro Instruction Index**

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 the same symbol is generated by two or more calls to the same definition, the suffix provided by &SYSNDX produces two or more unique symbols. Thus you avoid an error being flagged for multiply defined symbols.

The local 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).



---

## Chapter 8. How to Write Macro Instructions

This chapter describes macro instructions: where they can be used and how they are specified, including details on the name, operation, and operand entries, and what will be generated as a result of that macro call.

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

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 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 (see “SET Symbols” on page 212).

### Where Macro Instructions Can Appear

A macro instruction can be written anywhere in your program, if the assembler finds its definition either in a macro library or in the source module before it finds the macro instruction. However, the statements generated from the called macro definition must be valid assembler language instructions and allowed where the calling macro instruction appears. A macro instruction can be nested inside a macro definition (see “Nesting in Macro Definitions” on page 206).

---

### Macro Instruction Format

**Format of macro instruction:**

---

Name	Operation	Operand
Any symbol or blank	Symbolic operation code	0 through 240 operands separated by commas

---

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 blanks, 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 “ENTRY—Identify Entry-Point Symbol” on page 70).

## Notes:

1. An omitted operand has null character value.
2. Each positional operand can be up to 255 characters long.
3. If the assembler is invoked with the DBCS option, the positional operand can be a quoted string containing double-byte data.

The following 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 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. The default value should be 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.

A keyword operand must be coded in the format shown below:

```
KEYWORD=VALUE
```

where: KEYWORD has up to 62 characters without ampersand.  
= is an equal sign.  
VALUE can be up to 255 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 assembler is invoked with the DBCS option, the keyword operand can be a quoted string containing double-byte data.

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>'
```

To summarize the relationship of keyword operands to keyword parameters:

- The keyword of the operand corresponds (see (1) in Figure 51 on page 198) 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 51), the default value of the parameter is used.
- If the keyword of the operand does not correspond (see (3) in Figure 51) to any keyword parameter, the assembler issues an error message, but the macro is generated using the default values of the other parameters.

### 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 operands can be combined in the macro instruction operand field. However, the positional operands must be in the same order as the corresponding positional parameter in the macro prototype statement.

**Note:** The system variable symbol &SYSLIST(n) refers only to the positional operands in a macro instruction.

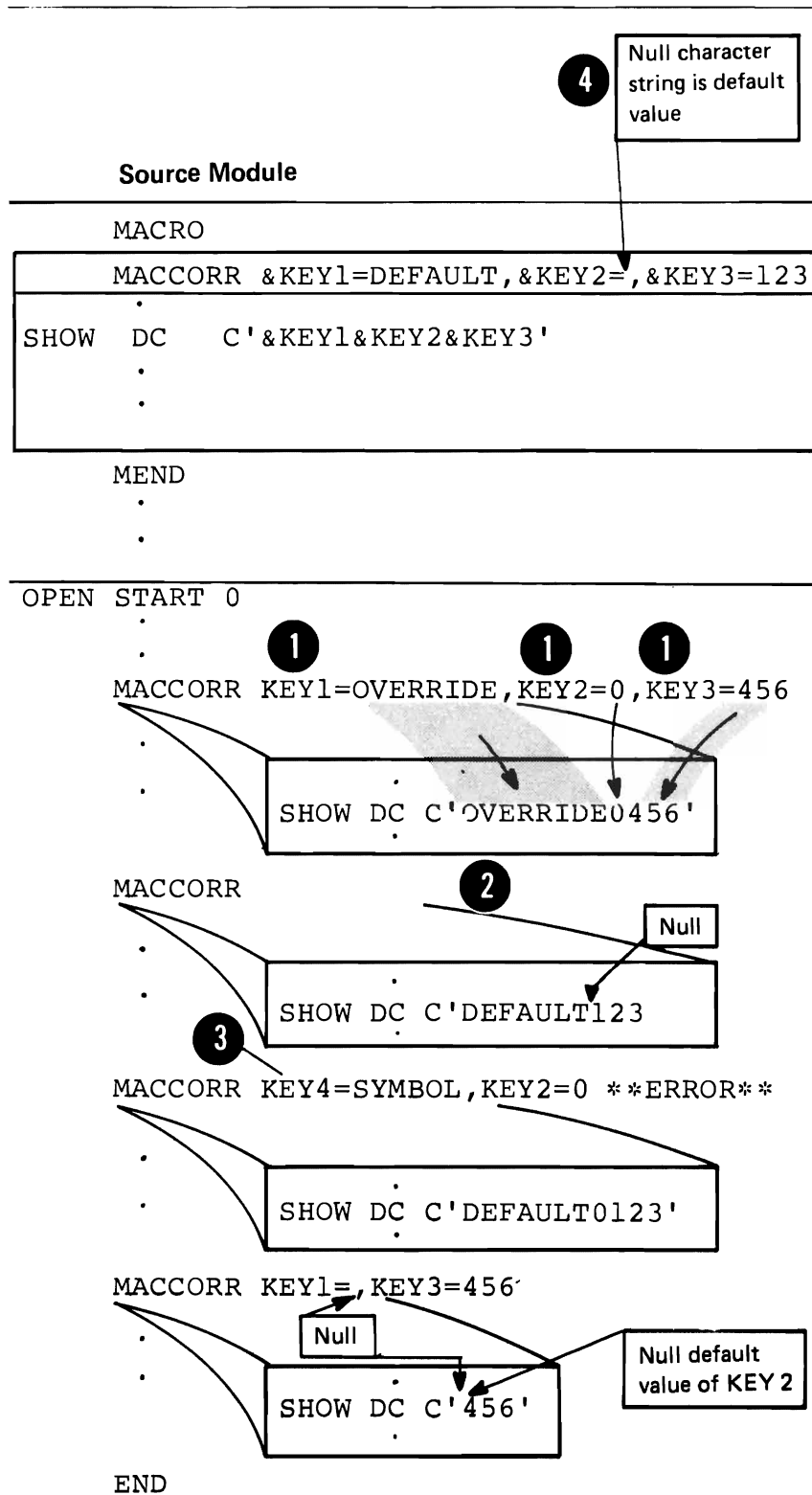


Figure 51. Relationship between Keyword Operands and Keyword Parameters and Their Assigned Values

**Note:** The default value specified for a keyword parameter can be the null character string (see (4) in Figure 51). The null character string is a character string with a length of zero; it is not a blank, because a blank occupies one character position.

---

## Sublists in Operands

You can use a sublist in a positional or keyword operand to specify several values. A sublist is one or more entries separated by commas and enclosed in parentheses. Each entry is a value to which you can refer in a macro definition by coding:

- The corresponding symbolic parameter with an appropriate subscript, or
- The system variable symbol &SYSLIST with appropriate 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 52 on page 200 illustrates that the value specified in a positional or keyword operand can be a sublist.

A symbolic parameter can refer to the entire sublist (see (1) in Figure 52), or to an individual entry of the sublist. To refer to an individual entry, the symbolic parameter (see (2) in Figure 52) must have a subscript whose value indicates the position (see (3) in Figure 52) 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 255 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.

The following list shows the relationship between subscripted parameters and sublist entries if:

1. A sublist entry is omitted: &PAR(3) (1,2,,4)
2. The subscript refers to an entry that is not in the sublist: &PAR(5) (1,2,3,4)
3. The value of the operand is not a sublist:  
    &PAR     A  
    &PAR(1)  A  
    &PAR(2)  A
4. The parameter is not subscripted: &PAR ()

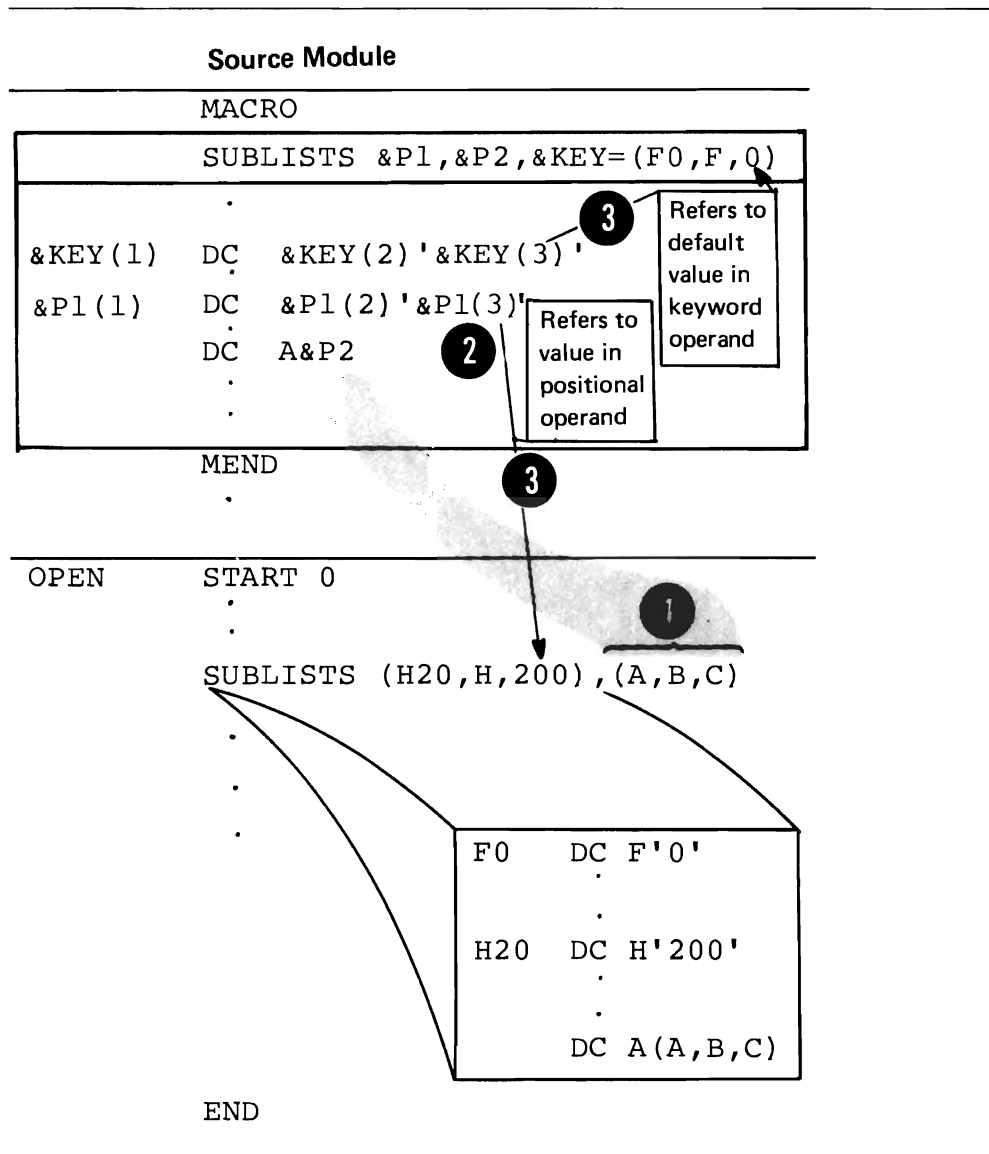


Figure 52. Sublists in Operands

Figure 53 shows the relationship between subscripted parameters and sublist entries if:

- A sublist entry is omitted (see (1) in Figure 53).
- The subscript refers past the end of the sublist (see (2) in Figure 53).
- The value of the operand is not a sublist (see (3) in Figure 53).
- The parameter is not subscripted (see (4) in Figure 53).

**Note:** The system variable symbol, `&SYSLIST(n,m)`, can also refer to sublist entries, but only if the sublist is specified in a positional operand.

Parameter	Sublist specified in corresponding operand (or as default value of keyword parameter)	Value generated (or used in computation)
&PAR(3)	<b>1</b> (1, 2, , 4)	Null character string
&PAR(5)	<b>2</b> (1, 2, 3, 4)	Null character string
&PAR &PAR(1) &PAR(2)	<b>3</b> { A A A	A A Null character string
&PAR &PAR(1) &PAR(2) &PAR &PAR(1) &PAR(3)	<b>4</b> (A) (A) <b>2</b> (A) } ( ) } ( ) } ( ) } } Considered as Sublists	(A) A Null character string ( ) Null character string Null character string
&PAR(2)  &PAR(1)	(A, , C, D)  ( ) } } This blank indicates end of operand field	Nothing  *ERROR* Unmatched left parentheses Nothing
&POSPAR(3) &SYSLIST(2,3)	Positional Operands A, (1, 2, 3, 4) A, (1, 2, 3, 4)	3 3

Figure 53. Relationship between Subscripted Parameters and Sublist Entries

## 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 255 characters. Inner elements of the sublists are referenced using additional subscripts on symbolic parameters or on &SYSLIST.

N'&SYSLIST with an n-element subscript array 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 level sublist.

For example, if &P is the first positional parameter and the value assigned in a macro instruction is (A,(B,(C)),D) then:

&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
&P(3)	=&SYSLIST(1,3)	= D
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.

---

## 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 one or more variable symbols.

The character string must not be greater than 255 characters after substitution of values for any variable symbols. This includes a character string that constitutes a sublist.

The character string values, including sublist entries, in the operands 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. Blanks appearing between commas do not signify an omitted positional operand or an omitted sublist entry; they indicate the end of the operand field.
2. Commas indicate omission of positional operands; no comma is needed to indicate omission of the last 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
EXAMPLE      17,*+4,,AREA,FIELD(6)
```

## Special Characters

Any of the 256 characters of the System/370 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 they are to be passed to the macro definition.

#### Examples:

```
&VAR
&A+&B+3+&C*10
'&MESSAGE'
&&REGISTER
```

### Single Quotation Marks

A single quotation mark is used:

- To indicate the beginning and end of a quoted string
- In a length attribute notation that is not within a quoted string

#### Examples:

```
'QUOTED STRING'
L'SYMBOL
```

### Shift-out (SO) and Shift-in (SI)

If the assembler is invoked with the DBCS option, then SO (X'0E') and SI (X'0F') are recognized as shift codes within quoted strings. SO and SI delimit the start and end of double-byte data respectively. Double-byte data will only be recognized within a quoted string.

### Quoted Strings

A quoted string is any sequence of characters that begins and ends with a single quotation mark (compare with conditional assembly character expressions described in "Character (SETC) Expressions").

*Two single quotation marks* must be specified inside each quoted string. This includes substituted single quotation marks.

Quoted strings can contain double-byte data, if the assembler is invoked with the DBCS option. 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 may be in any byte position after the SO. If the end of the operand is reached before SI is found, then error IEV203 "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 single quotation marks.

**Examples:**

```
' '
'L 'SYMBOL '
'QUOTE1 'AND'QUOTE2'
```

## Length Attribute Notation

In macro instruction operand values, the length attribute notation with ordinary symbols can be used outside of quoted strings, if the length attribute notation is preceded by any special character except the ampersand.

**Example:**

```
L'SYMBOL,10+L'AREA*L'FIELD
```

## Parentheses

In macro instruction operand values, there must be an equal number of left and right parentheses. They must be paired, that is, to each left parenthesis belongs 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)
```

## Blanks

One or more blanks outside a quoted string indicates the end of the entire operand field of a macro instruction. Thus blanks should only be used inside quoted strings.

**Example:**

```
'BLANKS 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

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 the following is the prototype of a macro definition:

```
MAC1 &F
```

the following macro instruction would generate a warning message:

```
MAC1 K=L (K is a valid keyword)
```

while the following macro instruction would not:

```
MAC1 2+2=4 (2+2 is not a valid keyword)
```

### Examples:

```
=H'201'  
A='B  
C(A=B)  
2X=B  
KEY=A=B
```

## Periods

A period (.) can be used in the value of an operand or sublist entry. It will be passed as a period. However, if it is used immediately after a variable symbol, it becomes a concatenation character. Then, two periods are required if one is to be passed as a character.

### Examples:

```
3.4  
&A.1  
&A..1
```

---

## Nesting in Macro Definitions

A nested macro instruction is a macro instruction you specify as one of the statements in the body of a macro definition. This allows you to call for the expansion of a macro definition from within another macro definition.

### 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 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 zero level includes outer macro calls, calls that appear in open code; the first level of nesting includes inner macro calls that appear inside macro definitions called from the zero level; the second level of nesting includes inner macro calls inside macro definitions that are called from the first level, etc.

### 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 allows you to 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 202, nested macro calls can specify values that include (see Figure 54 on page 207):

- Any of the symbolic parameters (see (1) in Figure 54) specified in the prototype statement of the containing macro definition
- Any SET symbols (see (2) in Figure 54) declared in the containing macro definition
- Any of the system variable symbols such as &SYSDATE, &SYSTIME, etc. (see (3) in Figure 54).

## Macro Definitions

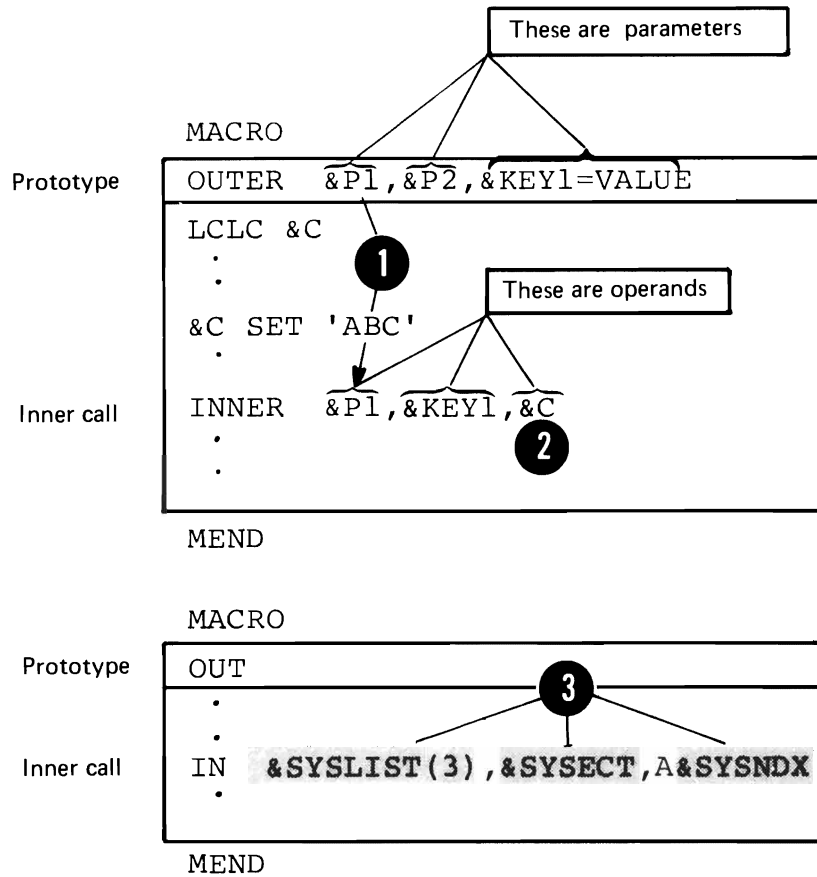


Figure 54. Values in Nested Macro Calls

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 and global SET symbols declared, and the number of sequence symbols used.

Exits taken from the different levels of nesting when a MEXIT or MEND instruction is encountered are as follows:

1. From the expansion of a macro definition called by an inner macro call, an exit is taken to the next sequential instruction that appears after the inner macro call in the containing macro definition.
2. From the expansion of a macro definition called by an outer macro, an exit is taken to the next sequential instruction that appears after the outer macro call in the open code of a source module.

## 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 55 on page 209). However, the value specified (see (1) in Figure 55) in the inner macro instruction operand must be identical to the corresponding symbolic parameter (see (2) in Figure 55) declared in the prototype of the containing macro definition.

Thus, a sublist can be passed (see (3) in Figure 55) and referred to (see (4) in Figure 55) as a sublist in the macro definition called by the inner macro call. Also, any symbol (see (5) in Figure 55) that is passed will carry its inherent 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.

**Note:** If 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 and attributes of symbols are not available for reference in the inner macro.

## System Variable Symbols in Nested Macros

The global read-only system variable symbols (&SYSPARM, &SYSDATE, and &SYSTIME) are not affected by the nesting of macros. The remaining system variable symbols are given local read-only values that depend on the position of a macro instruction in code and the operand value specified in the macro instruction.

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.

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.

The assembler gives &SYSECT the character string value of the name of the control section in force at the point at which a macro call is made. For a macro definition called by an inner macro call, the assembler will assign to &SYSECT the name of the control section generated in the macro definition that contains the inner macro call. The control section must be generated before the inner macro call is processed.

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.

The assembler gives &SYSLOC the character string value of the name of the location counter in use at the point at which a macro call is made. For a macro definition called by an inner macro call, the assembler will assign to &SYSLOC the name of the location counter in effect in the macro definition that contains the inner macro call.

&SYSECT and &SYSLOC have local scope; their read-only values remain constant throughout the expansion of the called macro definition.

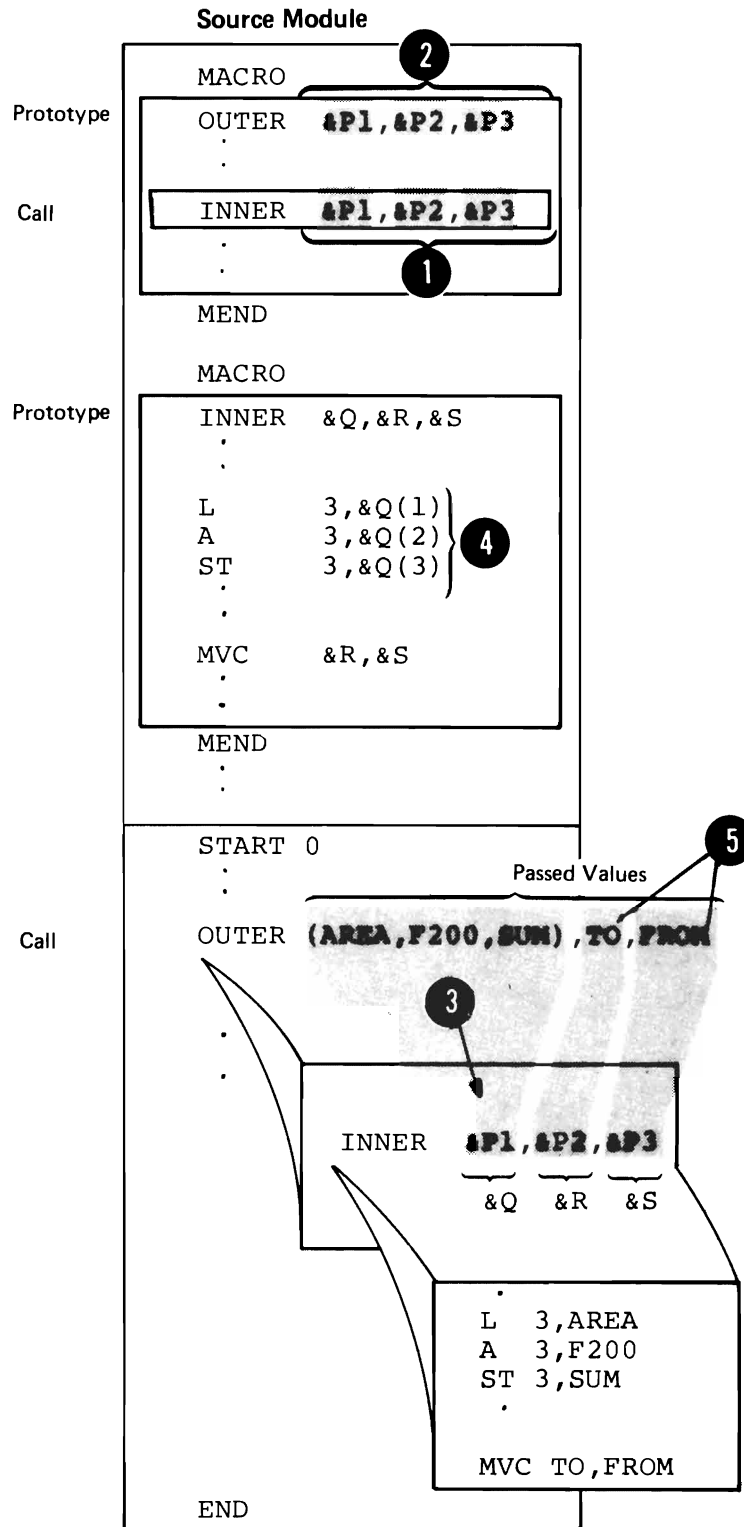


Figure 55. Passing Values through Nesting Levels



---

## Chapter 9. How to Write Conditional Assembly Instructions

This chapter describes the conditional assembly language. With the conditional assembly language, you can perform general arithmetic and logical computations, as well as many of the other functions you can perform with any other programming language. In addition, 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 at preassembly time. Then, at assembly time, it processes the generated instructions. Conditional assembly instructions, however, are not processed after preassembly time.

The conditional assembly language is more versatile when used 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 in an assembler language source program.

---

### Elements and Functions

The elements of the conditional assembly language are:

- SET symbols that represent data
- Attributes that represent different characteristics of data
- Sequence symbols that act as labels for branching to statements at preassembly time

The functions of the conditional assembly language are:

- Declaring SET symbols as variables for use by the conditional assembly language in its computations
- Assigning values to the declared SET symbols
- Evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, or as condition tests for branch instructions
- Selecting characters from strings for substitution in, and concatenation to, other strings; or for inspection in condition tests
- Branching and exiting from conditional assembly loops

## 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 column must contain an ampersand (&)
- The second column must contain an alphabetic character
- The remaining columns must contain 0 to 61 alphameric characters, including underscore (\_)

### 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. 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 Figure 56.

Features	SETA, SETB, SETC symbols	Symbolic Parameters	System Variable Symbols
Can be used in:			
Open code	Yes	No	&SYSPARM &SYSDATE &SYSTIME
Macro definitions	Yes	Yes	All
Scope:			
Local	Yes	Yes	&SYSLIST &SYSECT &SYSLOC &SYSNDX
Global	Yes	No	&SYSPARM &SYSDATE &SYSTIME
Values can be changed within scope of symbol	Yes <sup>1</sup>	No, read only value <sup>2</sup>	No, read only value <sup>2</sup>

Figure 56. Features of SET Symbols and Other Types of Variable Symbols

### Notes to Figure 56:

1. The value assigned to a SET symbol can be changed by using the SETA, SETB, or SETC instruction within the declared scope of the SET symbol.



2. A symbolic parameter and the system variable symbols are assigned values that remain fixed throughout their scope. Wherever a SET symbol appears in a statement, the assembler replaces the symbol with the last value assigned to the symbol.

SET symbols can be used in the name and operand fields of macro instructions. However, the value thus passed through a symbolic parameter into a macro definition is considered as a character string and is generated as such.

## Subscripted SET Symbols Specifications

### Format of a subscripted SET symbol:

---

&SETSYM(subscript)

where:

- &SETSYM is a variable symbol.
- 'subscript' is an arithmetic expression, whose value must not be 0 or negative.

---

**Example:** LCLA &ARRAY(20)

The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (see "Arithmetic (SETA) Expressions" below).

A subscripted SET symbol can be used anywhere an unscripted SET symbol is allowed. However, subscripted SET symbols must be declared as subscripted by a previous local or global declaration instruction.

The subscript refers to one of the many positions in an array of values identified by the SET symbol.

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 subsequent SETx instructions.

**Note:** The subscript can be a subscripted SET symbol. Five levels of subscript nesting are allowed.

## Created SET Symbols

Assembler H 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 the following:

- 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 will consider the preceding ampersand and this string as the name of a SET variable.

You can use created SET symbols wherever ordinary SET symbols are permitted, including declarations. You can also nest them 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: first character after '&' not alphabetic
3	\$5	&\$5QUA3	Valid
4		&QUA4	Valid

The created SET symbol can be thought of as a form of indirect addressing. With nested created SET symbols, you can get 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 "Ith" attribute of the symbol name in &SYM.

Created SET symbols also enable you to get some of the effect of multiple-dimensional 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) could be addressed as &(X&I.\$&J.\$&K). Thus, &X(2,3,4) would be represented by &X2\$3\$4. The \$s 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, which you define in a source module, can be described in terms of:

- Type, which distinguishes one form of data from another; 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 data
- Scaling, which indicates the number of positions occupied by the fractional portion of fixed-point and decimal constants in their object code form
- Integer, which indicates the number of positions occupied by the integer portion of fixed-point and decimal constants in their object code form
- Count, which gives the number of characters that would be required to represent the 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

These characteristics are called the attributes of 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. The attributes and their main uses are shown below:

Attribute	Purpose	Main Uses
Type	Gives a letter that identifies type of data represented	<ul style="list-style-type: none"> <li>• In tests to distinguish between different data types</li> <li>• For value substitution</li> <li>• In macros to discover missing operands</li> </ul>
Length	Gives number of bytes that data occupies in storage	<ul style="list-style-type: none"> <li>• For substitution into length fields</li> <li>• For computation of storage requirements</li> </ul>
Scaling	Refers to the position of the decimal point in decimal, fixed-point, and floating-point constants	<ul style="list-style-type: none"> <li>• For testing and regulating the position of decimal points</li> <li>• For substitution into a scale modifier</li> </ul>
Integer	Is a function of the length and scaling attributes of decimal, fixed-point, and floating-point constants	<ul style="list-style-type: none"> <li>• To keep track of significant digits (integers)</li> </ul>
Count	Gives the number of characters required to represent data	<ul style="list-style-type: none"> <li>• For scanning and decomposing of character strings</li> <li>• As indexes in substring notation</li> </ul>
Number <sup>1</sup>	Gives the number of sublist entries in a macro instruction operand sublist	<ul style="list-style-type: none"> <li>• For scanning sublists</li> <li>• As counter to test for end of sublist</li> </ul>
Defined	Indicates whether the symbol referenced has been defined prior to the attribute reference	<ul style="list-style-type: none"> <li>• To avoid assembling a statement again if the symbol referenced has been previously defined</li> </ul>

Figure 57. Data Attributes

**Notes to Figure 57:**

1. The number attribute of &SYSLIST(m) and &SYSLIST(m,n) is described in “&SYSLIST—Macro Instruction Operand” on page 185.

**Format of an attribute reference:**

---

Attribute ' Ordinary or  
Notation Variable Symbol

---

**Examples:**

```
T 'SYMBOL
L '&VAR
K '&PARAM
```

The attribute notation indicates the attribute whose value is desired. The ordinary or variable symbol represents the data that possesses the attribute. The assembler substitutes the value of the attribute for the attribute reference.

An attribute reference to the type, scaling, integer, count, and number attributes can be used only in a conditional assembly instruction. The length attribute reference can be used both in a conditional assembly instruction and in a machine or assembler instruction.

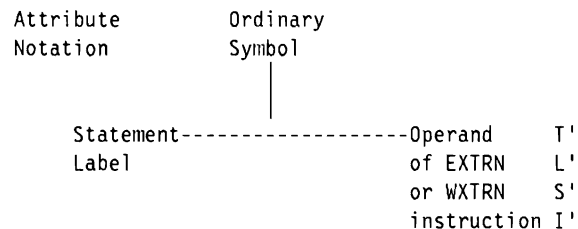
## Combination with Symbols

Figure 58 shows the seven kinds of attributes, identifying the types of symbols they can be combined with.

Symbols Specified	Type L'	Length S'	Scaling I'	Integer K'	Count N'	Number D'	Defined
<b>In open code:</b>							
Ordinary symbols	Yes	Yes	Yes	Yes	No	No	Yes
SET symbols	Yes	SETC only	SETC only	SETC only	Yes	Yes subscripted	SETC only
System variable symbols							
&SYSPARM &SYSDATE &SYSTIME	Yes	No	No	No	Yes	No	No
<b>In macro definitions:</b>							
Ordinary symbols	Yes	Yes	Yes	Yes	No	No	Yes
SET symbols	Yes	SETC only	SETC only	SETC only	Yes	Yes subscripted	SETC only
Symbolic parameters	Yes	Yes	Yes	Yes	Yes	Yes	Yes
System variable symbols							
&SYSLIST	Yes	Yes	Yes	Yes	Yes	Yes	Yes
&SYSECT,&SYSLOC, &SYSNDX,&SYSPARM, &SYSDATE,&SYSTIME	Yes	No	No	No	Yes	Yes	No

Figure 58. Attributes and Related Symbols

The value of an attribute for an ordinary symbol specified in an attribute reference comes from the data represented by the symbol, as shown below:



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 instruction in which the symbol is specified must appear in open code and must not contain any variable symbols.

**Note:** You can refer to instructions generated by conditional assembly substitution or macro expansion with attributes. However, no such reference can be made until the instruction is generated.

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:

1. For SET symbols and the system variable symbols: &SYSECT, &SYSLOC, &SYSNDX, &SYSPARM, &SYSDATE, and &SYSTIME, the attribute values come from the current data value of these symbols.
2. 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 values of the type, length, scaling, and integer attributes, however, come from the values represented by the macro instruction operands, as follows:

- a. If the operand is a sublist, the entire sublist and each entry of the sublist can possess attributes; all the individual entries and the whole sublist have the same attributes as those of the first suboperand in the sublist (except for "count," which can be different, and "number," which is relevant only for the whole sublist).
- b. 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 blank, then the value of the attributes for the operand are the same as for the ordinary symbol.
- c. If the operand (or sublist entry) is a character string other than a sublist or the character string described in b above, the type attribute is undefined (U) and the length, scaling, and integer attributes are invalid.

Because attribute references are allowed only in conditional assembly instructions, their values are available only at preassembly time, except for the length attribute which can be referred to outside conditional assembly instructions, and is, therefore, also available at assembly time.

**Note:** The system variable symbol, &SYSLIST, 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 listed below can also be substituted for an attribute reference containing &SYSLIST.

## Type Attribute (T')

The type attribute has a value of a single alphabetic character that indicates the type of data represented by:

- An ordinary symbol
- A macro instruction operand
- A SET symbol

The type attribute reference can be used only in the operand field of the SETC instruction or as one of the values used for comparison in the operand field of a SETB or AIF instruction.

**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.

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-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-, V-, or Y-type address constant, explicit length
- S S-type address constant, implicit length, aligned
- V 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

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 Identified as a control section name
- M Macro instruction
- 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 letters are used for the type attribute of data represented by inner and outer macro instruction operands only.

- N Self-defining term or the value of a SETA or SETB variable
- O Omitted operand (has a value of a null character string)

The following letter is used for symbols or macro instruction operands that cannot be assigned any of the above letters.

- U Undefined

The type attribute value U is assigned to the following:

- Ordinary symbols used as labels:
  - For the LORG 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 label will be generated due to conditional assembly statements
- SETC variable symbol
- System variable symbols: &SYSPARM, &SYSDATE, and &SYSTIME
- Macro instruction operands that specify literals
- Inner macro instruction operands that are ordinary symbols

**Note:** Because Assembler H allows attribute references to statements generated through substitution, certain cases in which a type attribute of U (undefined) or M (macro) is given under the OS/VS Assembler, may give a valid type attribute under Assembler H. If the value of the SETC symbol is equal to the name of an instruction that can be referred to by the type attribute, Assembler H allows you to use the type attribute with a SETC symbol.

## Length Attribute (L')

The length attribute has a numeric value equal to the number of bytes occupied by the data that is represented by the symbol specified in the attribute reference.

If the length attribute value is desired for preassembly processing, the symbol specified in the attribute reference must ultimately represent the name entry of a statement in open code. In such a statement, the length modifier (for DC and DS instructions) or the length field (for a machine instruction), if specified, must be a self-defining term. The length modifier or length field must not be coded as a multiterm expression, because the assembler does not evaluate this expression until assembly time.

Assembler H allows you to use the length attribute with a SETC symbol, if the value of the SETC symbol is equal to the name of an instruction that can be referenced by the length attribute.

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.

At preassembly time, an ordinary symbol used in the name field of an EQU instruction has a length attribute value of 1. At assembly time, the symbol has the same length attribute value as the first symbol 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.

**Notes:**

1. The length attribute reference, when used in conditional assembly processing, can be specified only in arithmetic expressions.
2. A length attribute reference to a symbol with the type attribute value of M, N, O, T, U, or \$ will be flagged. The length attribute for the symbol will be given the default value of 1.

**Scaling Attribute (S')**

The scaling attribute can be used only when referring to fixed-point, floating-point, or decimal constants. It has a numeric value that is assigned as shown below:

Constant Types Allowed	Type Attributes Allowed	Value of Scaling Attribute Assigned
Fixed-Point	H, F, and G	Equal to the value of the scale modifier (-187 through + 346)
Floating Point	D, E, L, and K	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)

**Notes:**

1. The scaling attribute reference can be used only in arithmetic expressions.
2. When no scaling attribute value can be determined, the reference is flagged and the scaling attribute is given the value of 1.
3. If the value of the SETC symbol is equal to the name of an instruction that can be referenced by the scaling attribute, Assembler H allows you to use the scaling attribute with a SETC symbol.

**Integer Attribute (I')**

The integer attribute has a numeric value that is a function of (depends on) the length and scaling attribute values of the data being referred to by the attribute reference. The formulas relating the integer attribute to the length and scaling attributes are given in Figure 59 on page 222.



Constant Type Allowed (attribute value)	Formula Relating the Integer to the Length and Scaling Attributes <b>1</b>	Examples	Values Of the Integer Attribute
Fixed-point (H,F, and G)	$I' = 8 * L' - S' - 1$	HALFCON DC HS6'-25.93' } 8*2-6-1 ONECON DC FS8'100.3E-2' } 8*4-8-1	9 23
Floating-point (D,E,L, and K)	when $L' \leq 8$ $I' = 2 * (L' - 1) - S'$	SHORT DC ES2'46.415' } 2*(4-1)-2 LONG DC DS5'-3.729' } 2*(8-1)-5	4 9
Only for L-Type	when $L' > 8$ $I' = 2 * (L' - 1) - S' - 2$	EXTEND DC LS10'5.312' } 2*(16-1)-10 -2	18
Decimal equal to the number of decimal digits to the left of the assumed decimal point after the number is assembled			
Packed (P)	$I' = 2 * L' - S' - 1$	PACK DC P'+3.513' } 2*3-3-1 03 513C	2
Zoned (Z)	$I' = L' - S'$	ZONE DC Z'3.513' } 4-3	1

Figure 59. Relationship of Integer to Length and Scaling Attributes

**Notes to Figure 59:**

1. The integer attribute reference can be used only in arithmetic expressions.
2. If the value of the SETC symbol is equal to the name of an instruction that can be referenced by the integer attribute, Assembler H allows you to use the integer attribute with a SETC symbol.

**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 would be 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 default value of 0.

**Number Attribute (N')**

The number attribute applies only to the operands of macro instructions. It has a numeric value equal to the number of sublist entries in the operand.

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. For example, if the only references to &A have been

	LCLA	&A(100)	
&A(5)	SETA	20,,,70	see description of
	AIF	(&A(20) GT 50).M	extended SET statements

then N'&A is equal to 8, because &A(8) is assigned the value 70.

**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(m) refers to the number of sublist entries in the m-th operand.

**Defined Attribute (D')**

The defined attribute indicates whether or not the symbol referenced has been defined prior to the attribute reference. A symbol is considered as 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. The value of the defined attribute is 1, if the symbol has been defined, or 0, if the symbol has not been defined.

The defined attribute can reference all symbols that can be referenced by the scaling (S') attribute.

The following is an example of how you can use the defined attribute:

	AIF	(D'A).AROUND
A	LA	1,4
.AROUND	ANOP	

In this example, the statement at A would be assembled, since the branch around it would not be taken. However, if by a branch the same statement were processed again, the statement at A would not be assembled:

.UP	AIF	(D'A).AROUND
A	LA	1,4
.AROUND	ANOP	
	.	
	AGO	.UP

You can save assembly time using the defined attribute. Each time the assembler finds a reference (attribute or branch) to an undefined symbol, it initiates a forward scan until it finds that symbol or reaches the END statement. You can use the defined attribute in your program to prevent the assembler from making this time-consuming forward scan.

## Sequence Symbols

You can use a sequence symbol in the name field of a statement to branch to that statement at preassembly time, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can thus select the model statements from which the assembler generates assembler language statements for processing at assembly time.

### Format of sequence symbol:

- 
- The first column must contain a period (.)
  - The second column must contain an alphabetic character
  - The remaining columns must contain 0 to 61 alphameric characters
- 

### Examples:

```
.BRANCHINGLABEL1  
.A
```

Sequence symbols can be specified in the name field of assembler language statements and model statements; however, the following lists assembler instructions in which sequence symbols must not be used as name entries:

COPY	GBLC	LCLB
EQU	ICTL	LCLC
GBLA	ISEQ	MACRO
GBLB	LCLA	OPSYN

In addition, 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.

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.

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.

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.

### 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,FIELDDB   Statement 3
-----
ST        2,SAVEAREA           Statement 4
L        2,FIELDDB
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.

## Attribute Definition and Lookahead

Symbol attributes are established in either definition mode or lookahead mode. Lookahead mode is entered when Assembler H encounters an attribute reference to a symbol that is not yet defined.

**Definition Mode:** Definition 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 generated.

**Lookahead Mode:** Lookahead is a sequential, statement-by-statement, forward scan over the source text. It is initiated when reference is made to an attribute (other than D') of a symbol not yet encountered, either by macro or open-code attribute reference, or by a forward AGO or AIF branch in open code.

If reference is made in a macro, forward scan begins with the first source statement following the outermost macro instruction. Programmer macros are bypassed. The text is not assembled. Lookahead attributes are tentatively established for all intervening undefined symbols. Tentative attributes are replaced and fixed when the symbol is subsequently encountered in definition mode. No macro expansion or open-code substitution is performed; no conditional or unconditional (AIF or AGO) branches are taken. COPY instructions are executed during lookahead, and the copied statements are scanned.

Lookahead ends when the desired symbol or sequence symbol is found, or when the END card or end of file is reached. All statements passed over by lookahead are saved on an internal file, and processed when the lookahead ends.

For purposes of attribute definition, a symbol is considered undefined 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, or if a DC, DS, or DXD modifier expression contains symbols not yet defined, that symbol is assigned a type attribute of U.

**Note:** Because no variable symbol substitution is performed by a lookahead, you should be careful when using a macro or open code substitution to generate END statements that separate source modules assembled in one job step (option BATCH). If a symbol is undefined within a module, lookahead will read in records past the point where the END statement is to be generated. All statements between the generated statement and the point at which lookahead stops (either because it finds a matching symbol, or because it finds an END statement) are ignored by the assembler. The next module will start at the point where lookahead stops.

**Lookahead Restrictions:** Assembler statements are analyzed only to the extent necessary 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 may eliminate one (or both) of the definitions.

Lookahead does not check undefined operation codes against library (system) 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 may be wrong if the undefined operation code 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.

---

## 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 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, or SETC 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.

### LCLA, LCLB, LCLC—Define Local Set Symbols

You 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, respectively.

### Format of LCLA, LCLB, LCLC:

---

Name	Operation	Operand
	LCLA, LCLB, or LCLC	One or more variable symbols, separated by commas

---

These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

A local SET symbol should not begin with &SYS because these characters are reserved for system variable symbols.

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 if 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:

1. Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.
2. It must not be the same as any global variable symbol declared within the same local scope.
3. 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.

**Subscripted Local SET Symbols:** A local subscripted SET symbol is declared by the LCLA, LCLB, or LCLC instruction.

### Format of subscripted local SET symbol:

---

Name	Operation	Operand
	LCLA LCLB, or LCLC	&SETSYM(dimension)

where:

- &SETSYM is a variable symbol.
- dimension must be an unsigned, decimal, self-defining term, but not 0.

---

**Example:** LCLB &B(10)

There is no limit to SET symbol dimensioning. The limit specified in the explicit (LCLx) or implicit (SETx) declaration can also be exceeded by subsequent SETx statements. The dimension indicates 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.

**Note:** A subscripted local SET symbol can be used only if the declaration has a subscript, which represents a dimension; a nonsubscripted local SET symbol can be used only if the declaration had no subscript.

**Alternative Format for LCLX Statements:** Assembler H 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	

## GBLA, GBLB, and GBLC Instructions

You 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, respectively.

### Format of GBLA, GBLB, and GBLC:

---

Name	Operation	Operand
GBLA, GBLB, or GBLC		One or more variable symbols, separated by commas

---

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. Subsequent GBLA, GBLB, or GBLC instructions do not reassign an initial value to the SET symbol.

Multiple GLBx statements can declare the same variable symbol if 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:

1. Within a macro definition, it must not be the same as any symbolic parameter declared in the prototype statement.

2. It must not be the same as any local variable symbol declared within the same local scope.
3. 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.

**Note:** A global SET symbol should not begin with the four characters &SYS, which are reserved for system variable symbols.

**Subscripted Global SET Symbols:** A global subscripted SET symbol is declared by the GBLA, GBLB, or GBLC instruction.

**Format of subscripted global SET symbol:**

---

Name	Operation	Operand
GBLA GBLB, or GBLC		&SETSYM(dimension)

where: &SETSYM is a variable symbol.  
dimension must be an unsigned, decimal, self-defining term,  
but not 0.

---

**Example:** GBLA &GA

There is no limit on the maximum subscript allowed. Also, the limit specified in the global declaration (GBLx) can be exceeded. The dimension indicates 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; a nonsubscripted global SET symbol can be used only if the declaration had no subscript.
3. Wherever a particular global SET symbol is declared with a dimension as a subscript, the dimension must be the same in each declaration.

**Alternative Format for GBLX Statements:** Assembler H permits the alternate statement format for GBLx instructions:

Cont.

   	GBLA	&GLOBAL_SYMBOL_FOR_DC_GEN, &LOOP_CONTRL_A, &VALUE_PASSED_TO_FIDO, &VALUE_RETURNED_FROM_FIDO	X X X
-----------	------	--	-------------



---

## Assigning Values to SET Symbols

### SETA—Set Arithmetic

The SETA instruction allows you to assign an arithmetic value to a SETA symbol. You can specify a single value or an arithmetic expression from which the assembler will compute the value to assign.

You can change the values assigned to an arithmetic or SETA symbol. This allows you to use SETA symbols as counters, indexes, or for other repeated computations that require varying values.

#### Format of SETA:

---

Name	Operation	Operand
A variable symbol	SETA	An arithmetic expression

---

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 a 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.

The expression in the operand field is 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  $+2^{31}-1$ , respectively.

### Subscripted SETA Symbols

The SETA symbol in the name field can be subscripted, but only if the same SETA symbol has been previously declared in a GBLA or LCLA instruction with an allowable dimension.

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, or exceed the dimension actually specified in the declaration.

## Arithmetic (SETA) Expressions

Arithmetic expressions can be used as shown in Figure 60.

Used in	Used as	Example
SETA instruction	Operand	&A1 SETA &A1 + 2
AIF or SETB instruction	Comparand in arithmetic relation	AIF (&A*10 GT 30).A
Subscripted SET symbols	Subscript	&SETSYM(&A + 10 - &C)
Substring notation (see L6)	Subscript	'STRING'(&A*2,&A-1)
Sublist notation	Subscript	Given sublist (A,B,C,D) 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' if &A=10 then &C=5-10*10 <sup>2</sup>

Figure 60. Use of Arithmetic (SETA) Expressions

**Note:** When an arithmetic expression is used in the operand field of a SETC instruction (see (1) in Figure 60), the assembler assigns the character value representing the arithmetic expression to the SETC symbol, after substituting values (see (2) in Figure 60) into any variable symbols. It does not evaluate the arithmetic expression.

Figure 61 defines an arithmetic expression.

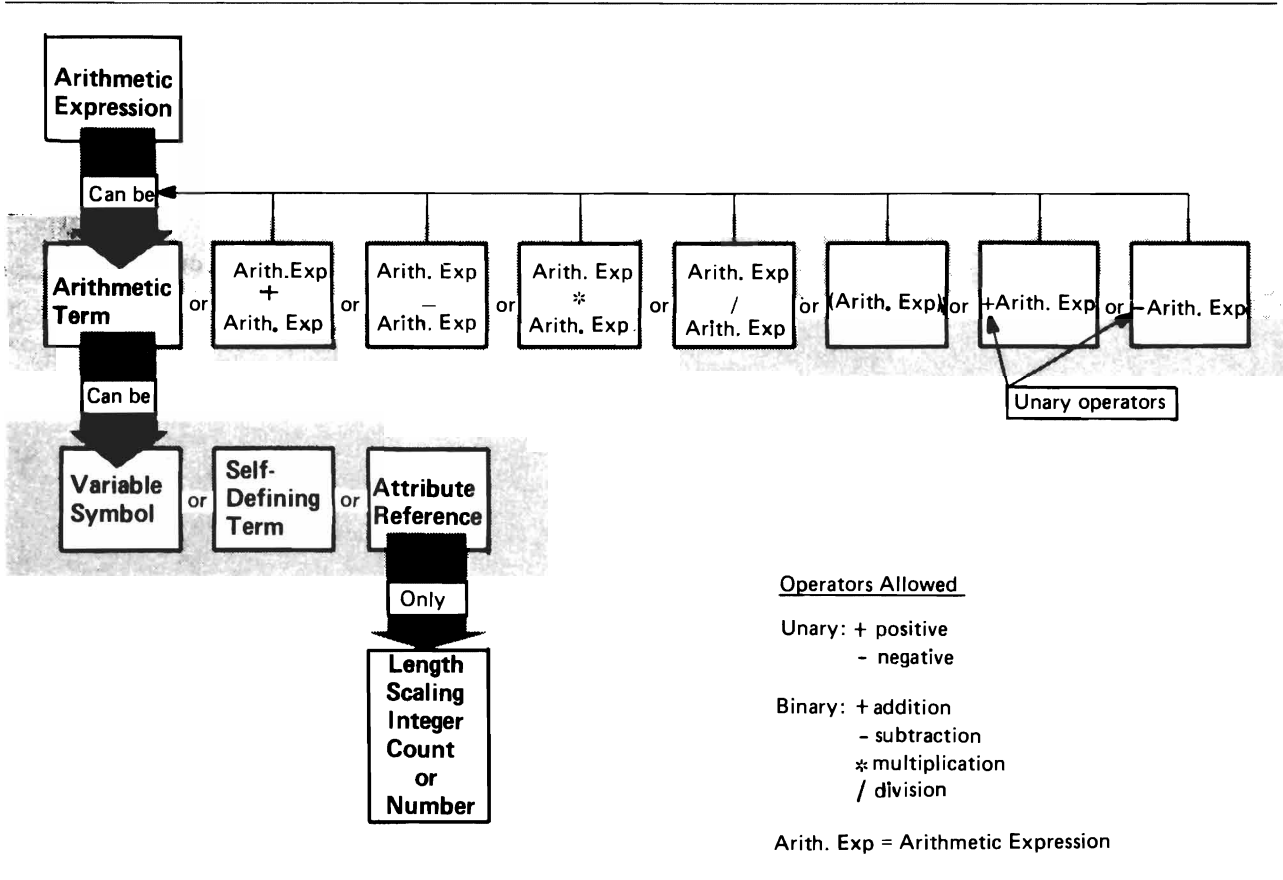


Figure 61. Defining Arithmetic (SETA) Expressions

Figure 62 shows the variable symbols that are allowed as terms in an arithmetic expression.

Variable symbol	Restrictions	Example	Value
SETA	None	---	---
SETB	None	---	---
SETC	Value must be an unsigned decimal self-defining term in the range 0 to 2,147,483,647	&C	123
&SYSPARM		&SYSPARM	2000
Symbolic parameters	Value must be a self-defining term	&PARAM	X'A1'
		&SUBLIST(3)	C'Z'
&SYSLIST(n)	Corresponding operand or sublist entry must be a self-defining term	&SYSLIST(3)	24
&SYSLIST(n,m)		&SYSLIST(3,2)	B'101'
&SYSNDX	None	---	---

Figure 62. Variable Symbols Allowed as Terms in Arithmetic Expressions

**Rules for Coding Arithmetic Expressions:** The following is a summary of coding rules for arithmetic expressions:

1. Both unary (operating on one value) 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 expression must not contain two terms in succession.
6. An arithmetic expression must not contain blanks between an operator and a term, nor between two successive operators.
7. An arithmetic expression can contain up to 24 unary and binary operators, and up to 255 levels of parentheses.

**Note:** The parentheses required for sublist notation, substring notation, and subscript notation count toward this limit.

**Evaluation of Arithmetic Expressions:** The assembler evaluates arithmetic expressions at preassembly time as follows:

1. It evaluates each arithmetic term.
2. It performs arithmetic operations from left to right. However,
  - a. It performs unary operations before binary operations, and
  - b. It performs the 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 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  $+2^{31}-1$ .

**SETC Variables in Arithmetic Expressions:** Assembler H 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. (The OS/VS Assembler allows SETC variables as arithmetic terms only if the value of the variable is a decimal self-defining term, not longer than 10 characters.)

**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)1
&AA          SETA      &C(3)2

```

**Notes:**

1. Allowed only by Assembler H
2. Allowed by the OS/VS Assembler and Assembler H

In evaluating the arithmetic expression in the fifth statement, the first term (&C(1)) is assigned the binary value 101 (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 allows you to 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 will convert a hexadecimal value in &H into a decimal value in &VAL:

```

&X           SETC      'X''&H''
&VAL         SETA      &X

```

The following code will convert 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

```

**Note:** The G-type self-defining term is valid only if the assembler is invoked with the DBCS option.

An arithmetic expression must not contain two terms in succession; however, any term may be preceded by any number of unary operators. +&A\*-&B is a value operand for a SETA instruction. The expression &FIELD +- is invalid because it has no final term.

## 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 an unsigned integer, 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,FIELDDB
-----
HERE   ST       2,SAVEAREA
        L        2,FIELDDB2
        ST       2,FIELDDB2
        L        2,SAVEAREA

```

Statements 1 and 2 assign the arithmetic values +10 and +12, respectively, 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 unsigned integer 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 unsigned integer 8.

The following example shows how the value assigned to a SETA symbol may 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,FIELDDB
-----
HERE   ST       2,SAVEAREA
        L        2,FIELDDB5
        ST       2,FIELDDB5
        L        2,SAVEAREA

```

Statement 1 assigns the arithmetic value +5 to SETA symbol &A. In statement 2, &A is converted to the unsigned integer 5. Statement 3 assigns the arithmetic value +8 to &A. In statement 4, therefore, &A is converted to the unsigned integer 8, instead of 5.

A SETA symbol may 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 may be used in the operand field of a SETA instruction may be used to refer to an operand in an operand sublist. Sublists are described in "Sublists in Operands" on page 199.

The following macro definition may be used to add the last operand in an operand sublist to the first operand in an operand sublist and store 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—Set Binary

You use the SETB instruction to assign a binary 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, respectively, to the SETB symbol. You can use this computed value in condition tests or for substitution.

## Format of SETB:

---

Name	Operation	Operand
A variable symbol	SETB	One of three options described below

---

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 a 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.

The three options that can be specified in the operand field are:

- A binary value (0 or 1)
- A binary value enclosed in parentheses

**Note:** An arithmetic value enclosed in parentheses is allowed. This value can be represented by an unsigned, decimal, self-defining term; a SETA symbol; or 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 or false, respectively. 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.

## Subscripted SETB Symbols

The SETB symbol in the name field can be subscripted, but only if the same SETB symbol has been previously declared in a GBLB or LCLB instruction with an allowable dimension.

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, or exceed the dimension actually specified in the declaration.

## 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 allows you to code a logical expression whose value (0 or 1) will vary according to the values substituted into the expression and thereby determine whether or not a branch is to be taken.

Figure 63 on page 239 defines a logical expression.



**Note:** An arithmetic relation is two arithmetic expressions separated by a relational operator. 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

**Rules for Coding Logical Expressions:** The following is a summary of coding rules for logical expressions:

1. A logical expression must not contain two logical terms in succession.
2. A logical expression can begin with the logical operator NOT.
3. A logical expression can contain two logical operators in succession; however, the only combinations allowed are: OR NOT or AND NOT. The two operators must be separated from each other by one or more blanks.
4. Any logical term, relation, or inner logical expression can be optionally enclosed in parentheses.
5. The relational and logical operators must be immediately preceded and followed by at least one blank or other special character.
6. A logical expression can contain up to 18 logical operators. Note that the relational and other operators used by the arithmetic and character expressions in relations do not count toward this total. There is no limit on the number of parentheses.

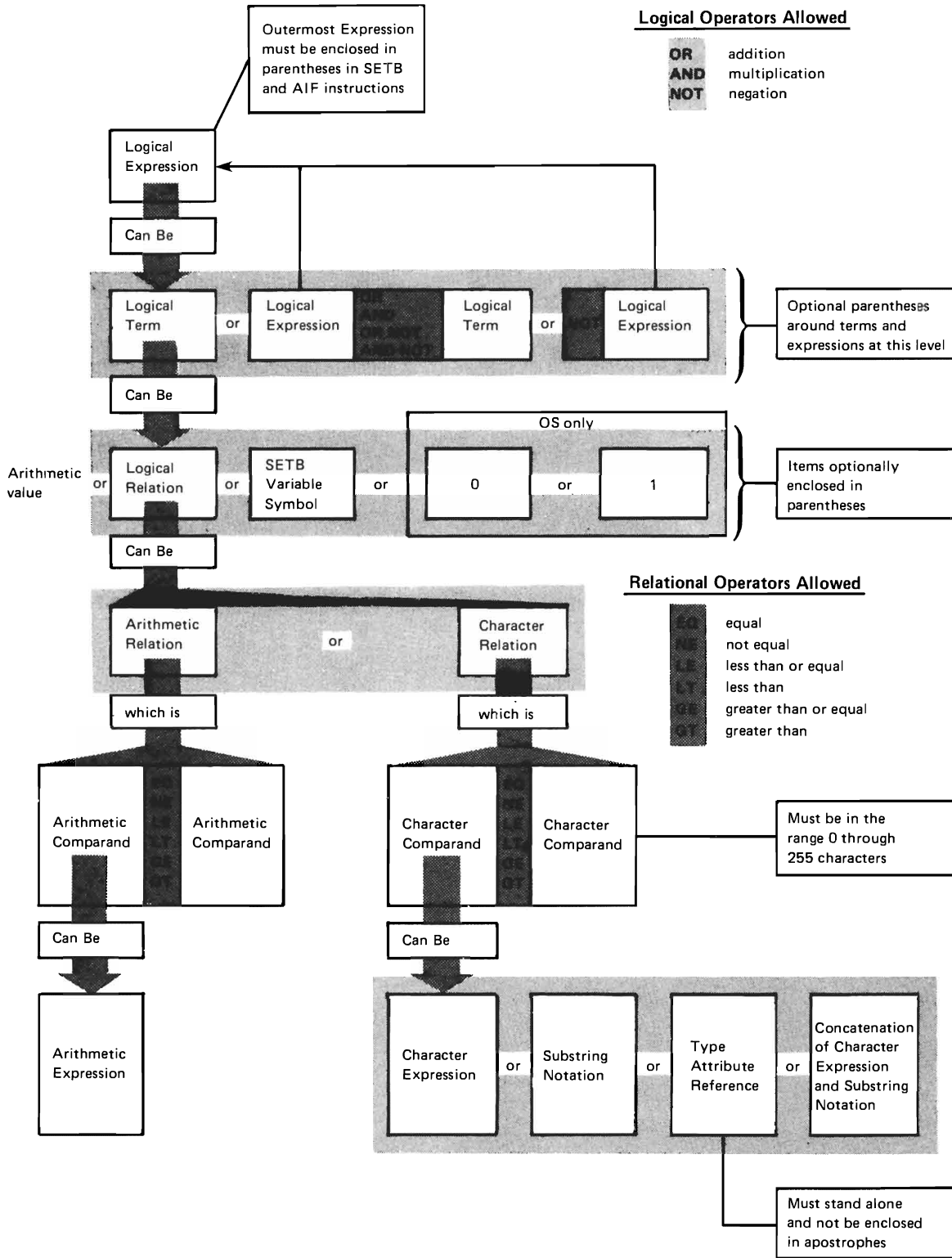


Figure 63. Defining Logical Expressions

**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, and then
  - b. The arithmetic or character relation, and finally
  - 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.

**Note:** The two comparands in a character relation are compared, character by character, according to binary (EBCDIC) representation of the character. If two comparands in a character relation have character values of unequal length, the assembler always takes the shorter character value to be less than the longer one.

3. The assembler performs logical operations from left to right. However,
  - a. It performs logical NOTs before logical ANDs and ORs
  - b. It performs logical ANDs before logical ORs
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 the outermost expression is evaluated.

**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, respectively.

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, respectively.

The following example illustrates these rules. It is assumed 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,FLDDB
-----
HERE   ST      2,SAVEAREA
        L      2,FLDDB1
        ST      2,FLDDB0
        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—Set Character

The SETC instruction allows you to assign a character value to a SETC symbol. You can assign whole character strings, or concatenate several smaller strings together. The assembler will assign 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.

You can change the character value assigned to a SETC symbol. This allows you to use the same SETC symbol with different values for character comparisons in several places, or for substituting different values into the same model statement.

### Format of SETC:

Name	Operation	Operand
A variable symbol	SETC	One of four options described below

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 a 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 four options that can be specified in the operand field are:

- A type attribute reference
- A character expression
- A substring notation
- A concatenation of substring notations, or character expressions, or both

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 255 characters.

**Note:** 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 any of the first three options, or any of the parts (character expression or substring notation) that make up the fourth option of the SETC instruction operand. The duplication factor can be any arithmetic expression allowed in the operand of a SETA instruction. For example:

```
&C1      SETC      (3)'ABC'
```

assigns the value 'ABCABCABC' to &C1.

**Notes:**

1. The assembler evaluates the represented character string (in particular, the substring) 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.

**Subscripted SETC Symbols:** The SETC symbol (see (1) in Figure 64 on page 243) in the name field can be subscripted, but only if the same SETC symbol has been previously declared (see (2) in Figure 64) in a GBLC or an LCLC instruction with an allowable dimension.

The assembler assigns the character value represented in the operand field to the position in the declared array (see (3) in Figure 64) given by the value of the subscript. The subscript expression must not be 0, or have a negative value, or exceed the dimension (see (4) in Figure 64) actually specified in the declaration.



Used in	Used as	Example
SETC instruction	Operand	&C SETC 'STRING0'
AIF or SETB instruction	Character string in character relation	AIF ('&' EQ 'STRING1').B
Substring notation	First part of notation	'SELECT'(2,5)=ELECT

Figure 65. Use of Character Expressions

**Evaluation of Character Expressions:** The value of a character expression is the character string within the enclosing single quotation marks, after the assembler performs 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 used in conditional assembly operations; for example, the value assigned to a SETC symbol.

**Notes:**

1. Two single quotation marks must be used to generate a single quotation mark 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 will generate 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, ('&&(1,1)).

The following statement assigns the character value HALF&& to the SETC symbol &AND.

```
&AND SETC 'HALF&&'
```

3. To generate a period, two periods must be specified after a variable symbol, or the variable symbol must have a period as part of its value.

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. 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.
5. The DBCS ampersand and apostrophe are not recognized as delimiters.
6. 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.

**Concatenation of Character String Values:** Character expressions can be concatenated to each other or to substring notations in any order. 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. The resulting value is a character string composed of the concatenated parts.

**Note:** The concatenation character (a period) is needed to separate the single quotation mark that ends one character expression from the single quotation mark that begins the next.

For example, either of the following statements may 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 are 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, it is possible to leave a redundant SI/SO pair at a concatenation point.

**Examples:**

```
&DBDA SETC '<Da>'
&SO SETC '&DBDA'(1,1)
&SI SETC '&DBDA'(4,1)
&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'
```

The substring notation is used to create variables &SO and &SI which have the values of SO and SI, respectively. The variable &DBCS1A is assigned the value <DaDb> with the SI/SO pair at the join removed. The assignment to variable &DBCS1E will fail with error IEV035, since the symbol &DBDA is terminated by SO and not by a period. The variable &DBCS2 will be assigned the value <DaDb>. The variable &DBCS2A will be assigned the value <DaDbDa>. In both cases, redundant SI/SO pairs are removed at the joins. The variable &DBCS3 will be assigned the value <DaDb>. Although SI and SO have been added at the join, the concatenation operation will remove two SI and two SO characters, since redundant SI/SO pairs will be 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.



**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,FIELD B
ST 2,FIELD A
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 may 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,FIELD B
ST 2,AREA A
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 illustrates the use of a substring notation as 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      FIELD,A,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.

**Note:** It is not possible, 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 call, to pass a string of values as parameters in a macro instruction. If you attempt to do this, the operand of the SETC instruction will be passed to the macro instruction as one parameter, not as a list of parameters. If the SETC operand is a sublist, it will also be passed to the macro instruction as one parameter.

*Concatenating Substring Notations and Character Expressions:* Substring notations 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 single quotation mark of the character expression and the opening single quotation mark 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 single quotation mark 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 single quotation mark 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 8 decimal digits.

If a SETA symbol is used in the operand field of a SETC statement, 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

In addition to 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.

### Format of extended SETx:

---

Name	Operation	Operand
A subscripted variable symbol	SETA, SETB, or SETC	operand1,operand2,...operandn

---

The name field specifies the name of the SET symbol and the position in the array to which the first value in the operand field is to be assigned. The successive operand values are then assigned to the successive positions in the array. If an operand is omitted, the corresponding element of the array is unchanged. Consider the following example:

```

                LCLA      &LIST(50)
&LIST(11) 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(11) SETA  5
&LIST(12) SETA 10
&LIST(14) SETA 20
&LIST(15) SETA 25
&LIST(16) SETA 30

```

**Alternative Statement Format:** You can use the alternative statement format for extended SETx statements. The above coding could then be written as follows:

Name	Operation	Operand	Comment	Cont.
&LIST(11)	SETA	5,	THIS IS	X
		10,,	AN ARRAY	X
		20,25,30	SPECIFICATION	

---

## Substring Notation

The substring notation allows you to 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, as shown in Figure 66 on page 249.

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 indicates the first character that is to be extracted from the character string. The second subscript indicates 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.

Used in	Used as	Example	Value assigned to SETC Symbol
SETC instruction operand	Operand	&C1 SETC 'ABC'(1,3)	ABC
	Part of operand	&C2 SETC '&C1'(1,2).'DEF'	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')	---

Figure 66. Substring Notation in Conditional Assembly Instructions

**Examples:**

Examples	Value of Variable Symbol	Character Value of Substring
'ABCDE'(1,5)		ABCDE
'ABCDE'(2,3)		BCD
'&C'(3,3)	ABCDE	CDE
'&PARAM'(3,3)	((A+3)*10)	A+3

The character string must be a valid character expression with a length, N, in the range 1 through 255 characters. The length of the resulting substring must be within the range 0 through 255.

The subscripts, e1 and e2, must be arithmetic expressions. The substring notation is replaced by a value that depends on the three elements: N, e1, and e2, as summarized in Figure 67 on page 250.

The numbers in the following list relate to the numbers in Figure 67:

- (1) In the usual case, the assembler generates a correct substring of the specified length.
- (2) When e1 has a value of 0 or a negative value, the assembler issues an error message.
- (3) When the value of e1 exceeds N, the assembler issues a warning message, and a null string is generated.
- (4) When e2 has a value of 0, the assembler generates the null character string. Note that, if e2 is negative, the assembler issues an error message.

- (5) When e2 indexes past the end of the character expression (that is, e1 + e2 is greater than N + 1), the assembler issues a warning message and generates a substring that includes only the characters up to the end of the character expression specified.



Examples:	Assume $0 < N \leq 255$	Character Value of Substring
1	$0 < e1 \leq N$ , $0 < e2 \leq N$ , and $e1 + e2 \leq N + 1$ 'ABCDEF' (2, 5)      N=6	BCDEF
2	$e1 \leq 0$ 'ABCDEF' (0, 5)      **ERROR** Value of e2 disregarded	null
3	$e1 > N$ 'ABCDEF' (7, 3)      N=6 *WARNING*	null
4	$e2 = 0$ 'ABCDEF' (3, 0) Value of e1 disregarded	null
5	$0 < e1 \leq N$ , $0 < e2 \leq N$ , but $e1 + e2 > N + 1$ 'ABCDEF' (3, 5)      N=6 *WARNING* 'ABCDEF' (3, 4)	CDEF CDEF

Figure 67. Summary of Substring Notation

---

## Branching

### AIF—Conditional Branch

You 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 allows you to control the sequence of statements to be generated.

It also allows you to check for error conditions and thereby to branch to the appropriate MNOTE instruction to issue an error message.

#### Format of AIF:

---

Name	Operation	Operand
A sequence symbol or blank	AIF	A logical expression enclosed in parentheses, immediately followed by a sequence symbol

---

The logical expression in the operand field is evaluated at preassembly 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, a branch is taken 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 an address at preassembly time. It is the address of the statement to which a branch is taken 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 does, or
- In the same macro definition in which the corresponding AIF instruction appears.

No branch can be taken from open code into a macro definition or between macro definitions, regardless of nested calls to other macro definitions.

The following macro definition may be used to generate the statements needed to move a fullword fixed-point number from one storage area to another. The statements will be 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 allows you to combine several successive AIF statements into one statement.

### Format of extended AIF:

---

Name	Operation	Operand
A sequence symbol or blank	AIF	(logical expression).S1, (logical expression).S2,... (logical expression).Sn

---

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, respectively, if the string being examined contains any of these characters.

**Alternative Statement Format:** The alternative statement format is allowed for extended AIF instructions. The format is illustrated in the above example.

## AGO—Unconditional Branch

The AGO instruction allows you to branch 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.

### Format of AGO:

---

Name	Operation	Operand
A sequence symbol or blank	AGO	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 does, or
- 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 is used to determine 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 is used to indicate 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 allows you to make branches according to the value of an arithmetic expression specified in the operand.

### Format of computed AGO:

---

Name	Operation	Operand
A sequence symbol or blank	AGO	(arithmetic expression) .S1,.S2,...,.Sn

---

If the arithmetic expression evaluates to k, where k lies between 1 and n (inclusive), then the branch is taken 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.
                                X
AGO          (&I).FIRST,.SECOND,
                                .THIRD,.FOURTH

```

**Alternative Statement Format:** The alternative statement format is allowed for computed AGO instructions. The above example could be coded as follows:

```

                                Cont.
                                X
AGO          (&I).FIRST,
                                .SECOND,
                                .THIRD,
                                .FOURTH
                                X

```

## ACTR—Conditional Assembly Loop Counter

The ACTR instruction allows you to set a conditional assembly loop 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 an AIF or AGO branching instruction in a macro definition or in open code, the loop counter for that part of the program is decremented by one. When the number of conditional assembly branches taken reaches the value assigned by the ACTR instruction to the loop counter, 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 at preassembly time.

## Format of ACTR:

---

Name	Operation	Operand
A sequence symbol or blank	ACTR	Any valid arithmetic (SETA) expression

---

A conditional assembly loop counter is set (or reset) to the value of the arithmetic expression in the operand field. The loop counter has a local scope; its value is decremented only by AGO and 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 individual loop counters.

The assembler sets its own internal loop counter both for open code and for each macro definition, if neither contains an ACTR instruction. The assembler assigns a standard value of 4096 to each of these internal loop counters.

**Loop Counter Operations:** Within the local scope of a particular loop counter (including the internal counters run by the assembler), the following occurs:

1. Each time an AGO or AIF branch is executed, the assembler checks the loop 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, before decrementing, the assembler will take one of two actions:
  - a. If it is processing instructions in open code, the assembler will process 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 inner macro instruction, the assembler processes the next sequential instruction after this inner call; that is, it continues processing at the next outer level of nested macros.

**Note:** The assembler halves the ACTR counter value when it encounters serious syntax errors in conditional assembly instructions.

## ANOP—Assembly No Operation

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 performs no operation itself, but you can use it to branch to instructions that already have symbols in their name fields. 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 could insert a labeled ANOP instruction immediately before the assignment instruction. By branching to the ANOP instruction with an AIF or AGO instruction, you would, in effect, be branching to the assignment instruction.

## Format of ANOP:

---

Name	Operation	Operand
A sequence symbol or blank	ANOP	

---

No operation is performed 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
&NAME  ST&TYPE  2,SAVEAREA                    Statement 4
        L&TYPE  2,&F
        ST&TYPE 2,&T
        L&TYPE  2,SAVEAREA
MEND
```

Statement 1 is used to determine 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, since 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. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, the statement following the ANOP instruction.

---

## Open Code

Conditional assembly instructions in open code allow you:

- To select, at preassembly time, 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.
- To pass local variable information from open code through parameters into macro definitions.

- To control the computation in and generation of macro definitions using global SET symbols.
- To 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:

1. To attributes in open code: For ordinary symbols, only references to the type, length, scaling, and integer attributes are allowed.

**Note:** 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.

2. To conditional assembly expressions in open code (see Figure 68).

Expression	Must not contain
Arithmetic (SETA)	<ul style="list-style-type: none"> <li>• &amp;SYSLIST</li> <li>• Symbolic parameters</li> <li>• Any attribute references to symbolic parameters, or &amp;SYSLIST, &amp;SYSECT, &amp;SYSNDX</li> </ul>
Character (SETC)	<ul style="list-style-type: none"> <li>• &amp;SYSLIST, &amp;SYSECT, &amp;SYSNDX</li> <li>• Attribute references to &amp;SYSLIST, &amp;SYSECT, &amp;SYSNDX, or to symbolic parameters</li> <li>• Symbolic parameters</li> </ul>
Logical (SETB)	<ul style="list-style-type: none"> <li>• Arithmetic expressions with the items listed above</li> <li>• Character expressions with the items listed above</li> </ul>

Figure 68. Restrictions on Coding Expressions

## MHELP—Macro Trace Facility

The MHELP instruction controls a set of trace and dump facilities. Options are selected by an absolute expression in the MHELP operand field. MHELP statements can occur anywhere in open code or in macro definitions. MHELP options remain in effect until superseded by another MHELP statement.

### Format of MHELP:

Name	Operation	Operand
	MHELP	Absolute expression, binary or decimal options (see below)

**MHELP B'1' or MHELP1, 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 MHELP2, 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 in conjunction with the Macro AIF dump, the Macro Exit dump, or the Macro Entry dump, will dump the parameter and SETC symbol values in EBCDIC and hexadecimal formats. Only positional and keyword parameters will be dumped in hexadecimal; system parameters will be 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 MHELP operand field is actually mapped into a fullword. Previously defined MHELP codes correspond to the fourth byte of this fullword.

&SYSNDX control is turned on by any bit in the third byte (operand values 256 through 65535, inclusive). Then, when &SYSNDX (total number of macro calls) exceeds the value of the fullword which contains the MHELP operand value, control is forced to stay at the open code level by, in effect, making every statement in a macro behave like a MEXIT. Open code macro calls are honored, but with an immediate exit back to open code. When the value of &SYSNDX reaches its limit, the message 'ACTR EXCEEDED—&SYSNDX' is issued.

**Examples:**

MHELP 256	Limit &SYSNDX to 256.
MHELP 1	Trace macro calls.
MHELP 256+1	Trace calls and limit &SYSNDX to 257.
MHELP 65536	No effect. No bits in bytes 3,4.
MHELP 65792	Limit &SYSNDX to 65792.

**Combining Options**

As shown in the example above, multiple options can be obtained 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, or MHELP 3. Substitution by means of variable symbols may also be used.



---

## Appendixes

- “Appendix A, Assembler Instructions and Statements” lists the related name, operation, and operand entries.
- “Appendix B, Summary of Constants” lists the constant types and gives related information concerning each.
- “Appendix C, Macro Language Summary” summarizes some of the information contained in Part 2.





## Appendix A. Assembler Instructions and Statements

Figure 69 summarizes assembler instructions, and Figure 70 on page 266 summarizes assembler statements.

Operation Entry	Name Entry	Operand Entry
ACTR	A sequence symbol or not present	An arithmetic SETA expression
AGO	A sequence symbol or not present	A sequence symbol
AIF	A sequence symbol or not present	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
AMODE	A sequence symbol or blank	24, 31, or ANY
ANOP	A sequence symbol or not present	Will be taken as a remark
AREAD	Any SETC symbol	One ordinary symbol
CCW	Any symbol or not present	Four operands, separated by commas
CCW0	Any symbol or not present	Four operands, separated by commas
CCW1	Any symbol or not present	Four operands, separated by commas
CNOP	Any symbol or not present	Two absolute expressions, separated by a comma
COM	A sequence symbol or not present	Will be taken as a remark
COPY	Must not be present	A symbol
CSECT	Any symbol or not present	Will be taken as a remark
CXD	Any symbol or not present	Will be taken as a remark
DC	Any symbol or not present	One or more operands, separated by commas
DROP	A sequence symbol or not present	One to 16 absolute expressions, separated by commas
DS	Any symbol or not present	One or more operands, separated by commas
DSECT	A variable symbol or an ordinary symbol	Will be taken as a remark
DXD	A symbol	One or more operands, separated by commas
EJECT	A sequence symbol or not present	Will be taken as a remark
END	A sequence symbol or not present	A relocatable expression or not present

Figure 69 (Part 1 of 3). Assembler Instructions

<b>Operation Entry</b>	<b>Name Entry</b>	<b>Operand Entry</b>
ENTRY	A sequence symbol or not present	One or more relocatable symbols, separated by commas
EQU	A variable symbol or an ordinary symbol	An absolute or relocatable expression
EXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas
GBLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas <sup>1</sup>
GBLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas <sup>1</sup>
GBLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas <sup>1</sup>
ICTL	Must not be present	One to three decimal values, separated by commas
ISEQ	Must not be present	Two decimal values, separated by a comma
LCLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas <sup>1</sup>
LCLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas <sup>1</sup>
LCLC	Must not be present	One or more variable symbols separated by commas <sup>1</sup>
LOCTR	A variable or ordinary symbol	Blank
LTORG	Any symbol or not present	Will be taken as a remark
MACRO <sup>2</sup>	Must not be present	Will be taken as a remark
MEND <sup>2</sup>	A sequence symbol or not present	Will be taken as a remark
MEXIT <sup>2</sup>	A sequence symbol or not present	Will be taken as a remark
MHELP	Any symbol or not present	Absolute expression, binary or decimal options
MNOTE <sup>2</sup>	A sequence symbol, a variable symbol, or not present	A severity code, followed by a comma, followed by any combination of characters (including double-byte characters, if the DBCS assembler option is specified) enclosed in single quotation marks
OPSYN	An ordinary symbol	A machine instruction mnemonic or an operation code defined by a previous OPSYN instruction
	A machine code mnemonic	Blank

Figure 69 (Part 2 of 3). Assembler Instructions

<b>Operation Entry</b>	<b>Name Entry</b>	<b>Operand Entry</b>
ORG	A sequence symbol or not present	A relocatable expression or not present
POP	Any symbol or not present	One or more operands, separated by commas
PRINT	A sequence symbol or not present	One to three operands
PUNCH	A sequence symbol or not present	One to 80 characters (including double-byte characters, if the DBCS assembler option is specified) enclosed in single quotation marks
PUSH	Any symbol or not present	One or more operands, separated by commas
REPRO	A sequence symbol or not present	Will be taken as a remark
RMODE	Any symbol or blank	24 or ANY
SETA	A SETA symbol	An arithmetic expression
SETB	A SETB symbol	A 0 or a 1, or 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.
SPACE	A sequence symbol or not present	A decimal self-defining term or not present
START	Any symbol or not present	A self-defining term or not present
TITLE <sup>1</sup>	A special symbol (0 to 4 characters), a sequence symbol, a variable symbol, or not present	One to 100 EBCDIC characters (or one to 49 DBCS characters, if the DBCS assembler option is specified) enclosed in single quotation marks
USING	A sequence symbol or not present	An absolute or relocatable expression followed by 1 to 16 absolute expressions, separated by commas
WXTRN	A sequence symbol or not present	One or more relocatable symbols, separated by commas

Figure 69 (Part 3 of 3). Assembler Instructions

**Notes to Figure 69:**

1. SET symbols may be defined as subscripted SET symbols.
2. May only be used as part of a macro definition.
3. See "Chapter 5. Assembler Instruction Statements" on page 91 for a description of the name entry.

<b>Instruction Entry</b>	<b>Name Entry</b>	<b>Operand Entry</b>
Model Statements <sup>1 2</sup>	An ordinary symbol, variable symbol, sequence variable symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present	Any combination of characters (including variable symbols)
Prototype Statement <sup>3</sup>	A symbolic parameter or not present	Zero or more operands that are symbolic parameters (separated by commas) followed by zero or more operands (separated by commas) of the form symbolic parameter, equal sign, optional standard value
Macro Instruction Statement <sup>3</sup>	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, <sup>4</sup> or not present	Zero or more positional operands (separated by commas) followed by zero or more keyword operands (separated by commas) of the form keyword, equal sign, value <sup>5</sup>
Assembler Language Statement <sup>1 2</sup>	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or not present	Any combination of characters (including variable symbols)

Figure 70. Assembler Statements

**Notes to Figure 70:**

1. Variable symbols may be used to generate assembler language mnemonic operation codes (listed in "Chapter 5. Assembler Instruction Statements" on page 91), except ACTR, COPY, END, ICTL, CSECT, DSECT, ISEQ, PRINT, REPRO, and START. Variable symbols may not be used in the name and operand entries of: COPY, END, ICTL, or ISEQ.
2. No substitution is performed for variables in the line following a REPRO statement.
3. May 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

Type	Implicit Length (Bytes)	Alignment	Length Modifier Range	Specified by	No. of constants per operand	Range for exponents	Range for scale	Truncation/padding side
A	4	Fullword	.1 to 4 (2)	Any expression	Multiple			Left
B	As needed	Byte	.1 to 256	Binary digits	Multiple			Left
C	As needed	Byte	.1 to 256 (1)	Characters	One			Right
D	8	Doubleword	.1 to 8	Decimal digits	Multiple	-85 to +75	0 to 14	Right (3)
E	4	Fullword	.1 to 8	Decimal digits	Multiple	-85 to +75	0 to 14	Right (3)
F	4	Fullword	.1 to 8	Decimal digits	Multiple	-85 to +75	-187 to +346	Left (3)
G	As needed	Byte	2 to 256 (4)	DBCS characters	One			Right
H	2	Halfword	.1 to 8	Decimal digits	Multiple	-85 to +75	-187 to +346	Left (3)
L	16	Doubleword	.1 to 16	Decimal digits	Multiple	-85 to +75	0 to 28	Right (3)
P	As needed	Byte	.1 to 16	Decimal digits	Multiple			Left
Q	4	Fullword	1 to 4	Symbol naming a DXD or DSECT	Multiple			Left
S	2	Halfword	2 only	One absolute or relocatable expression or two relocatable expressions: exp (exp)	Multiple			
V	4	Fullword	3,4	Relocatable symbol	Multiple			Left
X	As needed	Byte	.1 to 256 (1)	Hex digits	Multiple			Left
Y	2	Halfword	.1 to 2 (2)	Any expression	Multiple			Left
Z	As needed	Byte	.1 to 16	Decimal digits	Multiple			Left

Figure 71. Summary of Constants

### Notes to Figure 71:

1. In a DS assembler instruction, C and X type constants can have length specification to 65535.
2. Bit length specification permitted with absolute expressions only; relocatable A-type constants, 3 or 4 bytes only; relocatable Y-type constants, 2 bytes only.
3. Errors will be flagged if significant bits are truncated or if the value specified cannot be contained in the implicit length of the constant.
4. The length modifier must be a multiple of 2, and may be up to 65534 in a DS assembler instruction.



---

## Appendix C. Macro Language Summary

This appendix summarizes the macro language described in Part 2 of this publication. Figure 72 on page 270 indicates which macro language elements may be used in the name and operand entries of each statement. Figure 73 on page 270 is a summary of the expressions that may be used in macro instruction statements. Figure 74 on page 272 is a summary of the attributes that may be used in each expression. Figure 75 on page 273 is a summary of the variable symbols that may be used in each expression.



Figure 72. Macro Language Elements

Statement	Variable Symbols													Attributes						Sequence Symbol	
	Symbolic Parameter	Global SET Symbols			Local SET Symbols			System Variable Symbols						Type	Length	Scaling	Integer	Count	Number		
		SETA	SETB	SETC	SETA	SETB	SETC	&SYSNDX	&SYSECT	&SYSLIST	&SYSPARM	&SYSDATE	&SYSTIME								
MACRO																					
Prototype Statement	Name Operand																				
GBLA		Operand																			
GBLB			Operand																		
GBLC				Operand																	
LCLA					Operand																
LCLB						Operand															
LCLC							Operand														
Model Statement	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Operand	Operand							Name
SETA	Operand <sup>2</sup>	Name Operand	Operand <sup>3</sup>	Operand <sup>9</sup>	Name Operand	Operand <sup>3</sup>	Operand <sup>9</sup>	Operand		Operand <sup>2</sup>	Operand <sup>9</sup>				Operand	Operand	Operand	Operand	Operand	Operand	
SETB	Operand <sup>6</sup>	Operand <sup>6</sup>	Name Operand	Operand <sup>6</sup>	Operand <sup>6</sup>	Name Operand	Operand <sup>6</sup>	Operand <sup>6</sup>	Operand <sup>6</sup>	Operand <sup>4</sup>	Operand <sup>6</sup>	Operand <sup>6</sup>			Operand <sup>4</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>
SETC	Operand	Operand <sup>7</sup>	Operand <sup>8</sup>	Name Operand	Operand <sup>7</sup>	Operand <sup>8</sup>	Name Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand						
AIF	Operand <sup>6</sup>	Operand <sup>6</sup>	Operand	Operand <sup>6</sup>	Operand <sup>6</sup>	Operand	Operand <sup>6</sup>	Operand <sup>6</sup>	Operand <sup>4</sup>	Operand <sup>6</sup>	Operand <sup>6</sup>				Operand <sup>4</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Operand <sup>5</sup>	Name Operand
AGO																					Name Operand
ACTR	Operand <sup>2</sup>	Operand	Operand <sup>3</sup>	Operand <sup>2</sup>	Operand	Operand <sup>3</sup>	Operand <sup>2</sup>	Operand		Operand <sup>2</sup>	Operand <sup>2</sup>				Operand	Operand	Operand	Operand	Operand	Operand	
ANOP																					Name
				Name			AREAD														
MEXIT																					Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	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 Operand			Name Operand	Operand	Operand								Name
Inner Macro	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Operand	Operand								Name
Assembler Language Statement		Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand	Name Operation Operand													Name

1. Variable symbols in macro-instructions are replaced by their values before processing.  
 2. Only if value is self-defining term.  
 3. Converted to arithmetic +1 or +0.  
 4. Only in character relations.  
 5. Only in arithmetic relations.  
 6. Only in arithmetic or character relations.  
 7. Converted to unsigned number.  
 8. Converted to character 1 or 0.  
 9. Only if one to ten decimal digits

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions	
Can contain	Self-defining terms	Any combination of characters (including double-byte characters, if the DBCS assembler option is specified) enclosed in apostrophes	A 0 or a 1	
	Length, scaling, integer, count, and number attributes		SETB symbols	
	SETA and SETB symbols <sup>1</sup>	Any variable symbol enclosed in apostrophes	Arithmetic relations <sup>1</sup>	
	SETC symbols whose values are a decimal self-defining term <sup>1</sup>		Character relations <sup>2</sup>	
	&SYSPARM if its value is a decimal self-defining term		Arithmetic value	
	Symbolic parameters if the corresponding operand is a decimal self-defining term	A concatenation of variable symbols and other characters enclosed in apostrophes		
	&SYSLIST (n) if the corresponding operand is a decimal self-defining term	A type attribute reference		
&SYSLIST (n,m) if the corresponding operand is a decimal self-defining term				
	&SYSNDX			
Operations	+, - (unary and binary), *, and /; parentheses permitted	Concatenation, with a period (.)	AND, OR, and NOT parentheses permitted	
Range of values	-2 <sup>31</sup> to +2 <sup>31</sup> -1	0 through 255 characters	0 (false) or 1 (true)	
Used in	SETA operands	SETC operands	SETB operands	
	Arithmetic relations	Character relations <sup>2</sup>	AIF operands	
	Subscripted SET symbols			
	&SYSLIST subscript(s)			
	Substring notation			
	Sublist notation			

Figure 73. Conditional Assembly Expressions

**Notes to Figure 73:**

1. Values must be from 0 through 2 147 483 647.
2. A character relation consists of two character expressions related by the operator GT, LT, EQ, NE, GE, or LE. Type attribute notation and substring notation may also be used in character relations. The maximum size of the character expressions that can be compared is 255 characters. If the two character expressions are of unequal size, the smaller one will always compare less than the larger.

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 (m), &SYSLIST (m,n), SET symbols: &SYSTIME, &SYSPARM, &SYSDATE, &SYSECT, &SYSNDX, &SYSLOC	(May always be used)	1. SETC operand fields 2. Character relations
Length	L'	Ordinary Symbols defined in open code; symbolic parameters inside macro definitions: &SYSLIST (m), and &SYSLIST (m,n) inside macro definitions	Any letter except M,N,O,T and U	Arithmetic expressions
Scaling	S'	Ordinary Symbols defined in open code; symbolic parameters inside macro definitions: &SYSLIST (m), and &SYSLIST (m,n) inside macro definitions	H,F,G,D,E,L,K,P, and Z	Arithmetic expressions
Integer	I'	Ordinary Symbols defined in open code; symbolic parameters inside macro definitions: &SYSLIST (m), and &SYSLIST (m,n) inside macro definitions	H,F,G,D,E,L,K,P, and Z	Arithmetic expressions
Count	K'	Symbolic parameters, &SYSLIST (m) and &SYSLIST (m,n) inside macro definitions SET symbols; all system variable symbols	Any letter	Arithmetic expressions
Number	N'	Symbolic parameters, &SYSLIST and &SYSLIST (m) inside macro definitions	Any letter	Arithmetic expressions
Defined	D'	Ordinary Symbols defined in open code; symbolic parameters inside macro definitions: &SYSLIST (m), and &SYSLIST (m,n) inside macro definitions	H,F,G,D,E,L,K,P, and Z	Arithmetic expressions

Figure 74. Attributes

Refer to "Chapter 9. How to Write Conditional Assembly Instructions" on page 211 for usage restrictions of the attributes in Figure 74.

Variable Symbol	Declared by:	Initialized, or set to:	Value changed by:	May be used in:
Symbolic <sup>1</sup> parameter	Prototype statement	Corresponding macro instruction operand	Constant throughout definition	Arithmetic expressions if operand is decimal self-defining term Character expressions
SETA	LCLA or GBLA instruction	0	SETA instruction	Arithmetic expressions Character expressions
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 decimal self-defining term Character expressions
&SYSNDX <sup>1</sup>	The assembler	Macro instruction index	Constant throughout definition; unique for each macro instruction	Arithmetic expressions Character expressions
&SYSECT <sup>1</sup>	The assembler	Control section in which macro instruction appears	Constant throughout definition; set by CSECT, DSECT, START, and COM	Character expressions
&SYSLIST <sup>1</sup>	The assembler	Not applicable	Not applicable	N'&SYSLIST in arithmetic expressions
&SYSLIST(n) <sup>1</sup> &SYSLIST(n,m) <sup>1</sup>	The assembler	Corresponding macro instruction operand	Constant throughout definition	Arithmetic expressions if operand is decimal self-defining term Character expressions
&SYSPARM	PARM field	User defined or null	Constant throughout assembly	Arithmetic expression if value is decimal self-defining term Character expression
&SYSTIME	The assembler	System time	Constant throughout assembly	Character expression
&SYSDATE	The assembler	System date	Constant throughout assembly	Character expression
&SYSLOC <sup>1</sup>	The assembler	Location counter in effect where macro instruction appears	Constant throughout definition; set by CSECT, DSECT, START, COM, and LOCTR	Character expression

Figure 75. Variable Symbols

**Note to Figure 75:**

1. Can be used only in macro definitions.



# Index

## Special Characters

&SYSDATE system variable symbol 183  
&SYSECT system variable symbol 184  
&SYSLIST system variable symbol 185  
&SYSLOC system variable symbol 191  
&SYSNDX system variable symbol 187  
&SYSPARM system variable symbol 189  
&SYSTIME system variable symbol 190

## A

A-type constant 117  
absolute addresses, base registers for 47  
ACTR instruction 254  
address constants  
  A-type 117  
  complex relocatable 117  
  Q-type 121  
  S-type 119  
  V-type 120  
  Y-type 117  
addressability  
  by means of the DROP instruction 47  
  by means of the USING instruction 44  
  establishing 43  
  relative 48  
  using base register instructions 43  
addresses, relocatable or absolute 80  
addressing mode (AMODE) 56  
AGO instruction 253  
AIF instruction 251  
AMODE  
  indicators in ESD 56  
  instruction to specify addressing mode 57  
ANOP instruction 255  
AREAD instruction 181  
arithmetic (SETA) expressions  
  evaluation of 233  
  rules for coding 233  
  SETC variables in 234  
  using 230  
assembler instruction statements  
  base register instructions 43  
    See also base register instructions  
  data definition instructions 96  
    See also data definition instructions  
  listing control instructions 146  
    See also listing control instructions  
  operation code definition instruction 94  
  OPSYN instruction 94  
  program control instructions 132  
    See also program control instructions  
  program sectioning and linking instructions 48  
    See also program sectioning and linking instructions

assembler instruction statements (*continued*)  
  symbol definition instruction 92  
assembler language  
  assembler instruction statements 3  
  coding aids overview 7  
  coding conventions of 10  
  coding form for 10  
  compatibility of 3  
  conditional assembly instructions 211  
  introduction to 3  
  machine instruction statements 3, 73  
  macro instruction statements 3  
  statements, summary of 266  
  structure of 17  
  summary of instructions 263  
assembler program  
  basic functions 4  
  processing sequence 5  
  relationship to operating system 6  
attributes  
  count (K') 222  
  defined (D') 223  
  definition and lookahead 225  
  integer (I') 221  
  length (L') 220  
  number (N') 223  
  scaling (S') 221  
  summary of 269, 272  
  type (T') 218  
attributes in combination with symbols 217  
attributes, data 215

## B

base register instructions  
  DROP instruction 47  
  USING instruction 44  
base registers for absolute addresses 47  
binary constants 107  
binary self-defining term 30  
branching 251  
branching with extended mnemonic codes 75

## C

CCW instruction 130  
CCW0 instruction 130  
CCW1 instruction 131  
character (SETC) expressions, using 241  
character constants 108  
character relations in logical expressions 240  
character self-defining term 31  
character set 16  
character string values, concatenation of 245

- characters, special 203
- CNOP instruction 143
- coding aids overview 7
- coding conventions, assembler language
  - character set 16
  - comments statement 14
  - continuation lines 12
  - field boundaries
    - continuation indicator field 10
    - identification-sequence field 10
    - statement field 10
  - fixed format instruction statements 14
  - formatting specifications 14
  - free format instruction statements 14
  - standard coding form 10
- COM instruction 64
- combining keyword and positional parameters 174, 197
- comments statement format 14
- comments statements
  - function of 155
  - internal macro 182
  - ordinary 182
- compatibility, language 3
- computed AGO instruction 254
- concatenation of character string values 245
- concatenation of characters in model statements 165
- conditional assembly instructions
  - ACTR instruction 254
  - AGO instruction 253
  - AIF instruction 251
  - ANOP instruction 255
  - computed AGO instruction 254
  - extended AIF instruction 252
  - function of 176
  - GBLA instruction 228
  - GBLB instruction 228
  - GBLC instruction 228
  - how to write 211
  - LCLA instruction 226
  - LCLB instruction 226
  - LCLC instruction 226
  - MHELP instruction 257
  - SETA instruction 230
  - SETB instruction 236
  - SETC instruction 241
  - substring notations in 248
- conditional assembly language
  - overview 157
  - summary of expressions 271
- constants
  - address 117
  - alignment of 99
  - binary 107
  - character 108
  - decimal 115
  - duplication factor 101
  - fixed-point 112
- constants (*continued*)
  - floating-point 122
  - graphic 110
  - hexadecimal 111
  - information about 98
  - length attribute value of symbols naming 98
  - modifiers of 102
  - nominal values of 106
  - padding of values 100
  - subfield 1 101
  - subfield 2 102
  - subfield 3 102
  - subfield 4 106
  - summary of 267
  - symbolic addresses of 98
  - truncation of values 100
  - types of 97, 102
- continuation indicator field 10
- continuation lines 12
- control instructions 74
- control sections
  - concept of 50
  - defining a 59
  - defining blank common 64
  - executable 50
  - first 53
  - identifying a 60
  - reference 50
  - unnamed 55
- COPY instruction 144, 176
- CSECT instruction 60
- CXD instruction 68

## D

- D-type floating-point constant 122
- D' defined attribute 223
- data attributes 215
- data definition instructions
  - CCW instruction 130
  - CCW0 instruction 130
  - CCW1 instruction 131
  - DC instruction 96
  - DS instruction 127
- data, immediate, in machine instructions 83
- DBCS
  - See double-byte data
- DC instruction 96
- decimal constants
  - p and z 115
  - packed 115
  - zoned 115
- decimal instructions 74
- decimal self-defining term 29
- double-byte character set
  - See double-byte data
- double-byte data
  - code conversion in the macro language 234
  - concatenation in SETC expressions 245

- double-byte data (*continued*)
  - concatenation of fields 166
  - continuation of 11, 12
  - definition of 17
  - duplication of 242
  - graphic constants 97, 110
  - graphic self-defining term 31
  - in C-type constants 110
  - in character self-defining terms 31
  - in comments 14
  - in keyword operands 196
  - in macro comments 182
  - in macro operands 169
  - in MNOTE operands 178
  - in positional operands 196
  - in PUNCH operands 136
  - in quoted strings 203
  - in remarks 16
  - in REPRO operands 136
  - in TITLE operands 147
  - listing of macro-generated fields 165
  - notation 9
- DROP instruction 47
- DS instruction 127
- DSECT instruction 61
- dummy section, identifying a 61
- dummy sections, external 66
  - See also external dummy sections
- duplication factor in constants 101
- DXD instruction 67

## E

- E-type floating-point constant 122
- EJECT instruction 148
- elements and functions
  - data attributes 215
  - sequence symbols 224
  - SET symbols 212
- END instruction 145
- ENTRY instruction 70
- EQU instruction 92
- ESD entries 55
- expressions
  - absolute 41
  - arithmetic 230
  - character 241
  - complex relocatable 42
  - conditional assembly, summary of 271
  - discussion of 39
  - evaluation of 41, 240
  - evaluation of character 244
  - logical 236
  - paired relocatable terms 41
  - relocatable 41
  - rules for coding 39, 238
- extended AIF instruction 252
- extended continuation indicator 12, 165

- extended mnemonic codes, branching with 75
- extended SET statement 248
- external dummy sections
  - CXD instruction to define an 68
  - discussion of 66
  - DXD instruction to define an 67
- external symbol dictionary entries 55
- EXTRN instruction 71

## F

- field boundaries
  - continuation indicator field 10
  - identification-sequence field 10
  - statement field 10
- first control section 53
- fixed format for instruction statements 14
- fixed-point constants 112
- floating-point constants
  - D-type 122
  - E-type 122
  - L-type 122
- floating-point instructions 74
- formatting specifications
  - name entry 15
  - operand entries 16
  - operation entry 15
  - remarks entries 16
- free format for instruction statements 14

## G

- GBLA instruction 228
- GBLB instruction 228
- GBLC instruction 228
- general instructions 73
- generated fields, listing of 164
- graphic constants 110
- graphic self-defining term 31

## H

- header, macro definition 160
- hexadecimal constants 111
- hexadecimal self-defining term 30

## I

- I' integer attribute 221
- ICTL instruction 132
- identification-sequence field 10
- immediate data in machine instructions 83
- inner and outer macro instructions 206
- inner macro instructions 176
- inner macro instructions, passing sublists to 202
- input/output operations 75
- instruction statement format 14
- internal macro comments statements 182



ISEQ instruction 133

## K

K' count attribute 222

keyword parameters 172, 196

## L

L-type floating-point constant 122

L' length attribute 220

LCLA instruction 226

LCLB instruction 226

LCLC instruction 226

length attribute 33

length fields in machine instructions 83

library macro definitions 156

linkages

by means of the ENTRY instruction 70

by means of the EXTRN instruction 71

by means of the WXTRN instruction 72

symbolic 69

linking 48

listing control instructions

EJECT instruction 148

PRINT instruction 149

SPACE instruction 148

TITLE instruction 146

listing of generated fields 164

literal pool 38, 141

literals

differences between constants, self-defining terms,  
and 36

duplicate 142

explanation of 36

general rules for usage 38

location counter reference 32

location counter setting 51

LOCTR instruction 52

logical (SETB) expressions 236

lookahead mode 225

LTOrg instruction 140

## M

machine instruction formats

RR format 84

RRE format 84

RS format 85

RX format 86

S format 87

SI format 87

SS format 88

SSE format 89

machine instruction statements 77

addresses 80

control 74

decimal 74

examples of 84

machine instruction statements (*continued*)

floating-point 74

general 73

immediate data 83

input/output 75

length field in 83

operand entries 78

registers, use of 79

symbolic operations codes in 78

macro definitions

body of a 163

combining positional and keyword  
parameters 174

comments statements 182

COPY instruction 176

format of 160

header 160

how to prepare 159

inner macro instructions 176

internal macro comments statements 182

keyword parameters 172

MEXIT instruction 179

MNOTE instruction 176

nesting in 206

positional parameters 171

subscripted symbolic parameters 175

symbolic parameters 170

trailer 160

where to define in a source module 159

where to define in open code 159

macro instruction

alternative ways of coding 193

description of 193

format of 193

general rules and restrictions 206

inner and outer 206

multilevel sublists 201

name entry 194

operand entry 195

operation entry 194

passing sublists to inner 202

passing values through nesting levels 208

prototype 160

(see also prototype, macro definition)

sublists in operands 199

summary of 266

values in operands 202

macro language

comments statements 155

conditional assembly language 157

defining 153

library macro definition 156

macro instruction statement 155

model statements 155

processing statements 155

source macro definition 156

summary of 269

using 153

- macro library 156
- MEXIT instruction 179
- MHELP instruction
  - combining options 259
  - format 257
  - global suppression—operand=32 258
  - macro AIF dump—operand=4 258
  - macro branch trace—operand=2 258
  - macro call trace—operand=1 258
  - macro entry dump—operand=16 258
  - macro exit dump—operand=8 258
  - macro hex dump—operand=64 258
  - MHELP control on &SYSNDX 258
  - MHELP suppression—operand=128 258
- mnemonic codes, extended, branching with 75
- MNOTE instruction 176
- model statements
  - explanation of 163
  - function of 155
  - rules for concatenation of characters in 165
  - rules for specifying fields in 168
  - summary of 266
  - variable symbols as points of substitution in 163
- modifiers of constants
  - exponent 106
  - length 103
  - scale 105
- multilevel sublists 201

## N

- N' number attribute 223
- name entry 15
- nested macros, system variable symbols in 208
- nesting
  - levels of 206
  - recursion 206
- nesting in macro definitions 206
- nesting levels, passing values through 208
- nominal values of constants (literal)
  - address 117
  - binary 107
  - character 108
  - decimal 115
  - fixed-point 112
  - floating-point 122
  - graphic 110
  - hexadecimal 111

## O

- omitted operands 202
- open code 159, 160, 256
- operand entries
  - coding rules for 16
  - combining positional and keyword 197
  - in machine instructions 78
  - keyword 196
  - multilevel sublists in 201

- operand entries (*continued*)
  - omitted 202
  - positional 195
  - special characters in 203
  - sublists in 199
- operands
  - omitted 202
  - sublists in 199
  - values in 202
- operating system, relationship to assembler program 6
- operation codes, symbolic 78
- operation entry 15
- OPSYN instruction 94
- ordinary comments statements 182
- ordinary symbols 26
- ORG instruction 138

## P

- parameters
  - combining positional and keyword 174
  - keyword 172
  - positional 171
  - subscripted symbolic 175
  - symbolic 170
- parentheses, terms in 35
- pool, literal
  - See literal pool
- POP instruction 137
- positional parameters 171, 195
- PRINT instruction 149
- processing statements
  - conditional assembly instructions 176
  - COPY instruction 176
  - function of 155
  - inner macro instructions 176
  - MEXIT instruction 179
  - MNOTE instruction 176
- program control instructions
  - AREAD instruction 181
  - CNOP instruction 143
  - COPY instruction 144
  - END instruction 145
  - ICTL instruction 132
  - ISEQ instruction 133
  - LTORG instruction 140
  - ORG instruction 138
  - POP instruction 137
  - PUNCH instruction 134
  - PUSH instruction 137
  - REPRO instruction 136
- program sectioning 48
  - See also sectioning, program
- program sectioning and linking instructions
  - AMODE instruction 57
  - COM instruction 64
  - CSECT instruction 60
  - CXD instruction 68

program sectioning and linking instructions (*continued*)

- DSECT instruction
- DXD instruction 67
- ENTRY instruction 70
- EXTRN instruction 71
- LOCTR instruction 52
- RMODE instruction 58
- START instruction 59
- WXTRN instruction 72

prototype, macro instruction

- alternative ways of coding 162
- format of 161
- function of 160
- name field 161
- operand field 161
- operation field 161
- summary of 266

PUNCH instruction 134

PUSH instruction 137

## Q

Q-type constant 121

## R

registers, use of, by machine instructions 79

relative addressing 48

remarks entries 16

REPRO instruction 136

residence mode (RMODE) 56

RMODE

- indicators in ESD 56
- instruction to specify residence mode 58

RR format 84

RRE format 84

RS format 85

RX format 86

## S

S format 87

S-type constant 119

S' scaling attribute 221

sectioning, program

- accumulating the cumulative length of external dummy sections with the CSD instruction 68
- control sections 50
- defining an external dummy section with a DXD instruction 67
- ESD entries 55
- first control section 53
- identifying a blank common control section with a COM instruction 64
- identifying a control section with a CSECT instruction 60
- identifying a dummy section with a DSECT instruction 61

sectioning, program (*continued*)

- identifying external symbols with the EXTRN instruction 71
- identifying the entry-point symbol with the ENTRY instruction 70
- identifying weak external symbols with the WXTRN instruction 72
- location counter setting 51
- source module 49
- specifying multiple location counters within a control section with a LOCTR instruction 52
- specifying the addressing mode of a control section with an AMODE instruction 57
- specifying the residence mode of a control section with an RMODE instruction 58
- starting assembly with a START instruction 59
- unnamed control section 55

self-defining terms

- binary 30
- character 31
- decimal 29
- graphic 31
- hexadecimal 30
- using 29

sequence symbols 27, 224

SET symbols

- assigning values to 230
- created 214
- declaring 226
- define global 228
- define local 226
- description of 212
- extended 248
- scope of 212
- SETA (set arithmetic) 230
- SETB (set binary) 236
- SETC (set character) 241
- specifications 212
- specifications for subscripted 214
- subscripted 212

SETA

- arithmetic expression 230
- instruction format 230
- symbols, subscripted 230
- symbols, using 235

SETB

- character relations in logical expressions 240
- instruction format 236
- logical expression 236
- symbols, subscripted 236
- symbols, using 240

SETC

- character expression 241
- character expressions 243
- instruction format 241
- symbols, subscripted 241

SI format 87

source macro definitions 156

- SPACE instruction 148
- special characters 203
- SS format 88
- SSE format 89
- START instruction 59
- statement field 10
- structure, assembler language
  - symbols 25
  - terms 25
- subfield 1 of constant 101
- subfield 2 of constant 102
- subfield 3 of constant 102
- subfield 4 of constant 106
- sublists in operands 199
- sublists, multilevel 201
- sublists, passing, to inner macro instructions 202
- subscripted symbolic parameters 175
- substring notation 248
- symbol definition instruction
  - EQU instruction 92
- symbol table 26
- symbolic operation codes 78
- symbolic parameters 170
- symbols
  - attributes in combination with 217
  - defining 27
  - explanation of 25
  - extended SET 248
  - length attribute 33
  - ordinary 26
  - previously defined 28
  - restrictions on 28
  - sequence 27, 224
  - system variable 183
  - variable 27
  - variable, as points of substitution in model statements 163
- system macro instructions 156
- system variable symbols
  - &SYSDATE 183
  - &SYSECT 184
  - &SYSLIST 185
  - &SYSLOC 191
  - &SYSNDX 187
  - &SYSPARM 189
  - &SYSTIME 190
  - in nested macros 208
  - summary of 273

## T

- T' type attribute 218
- terms 25
  - See also self-defining terms
- terms in parentheses 35
- TITLE instruction 146
- trailer, macro definition 160
- types of constants 102

## U

- underscore character 26
- unnamed control section 55
- USING instruction
  - base registers for absolute addresses 47
  - discussion of 44
  - domain of a 46
  - how to use the 46
    - for executable control sections 46
    - for reference control sections 46
  - notes about the domain of a 46
  - notes about the range of a 47
  - range of a 45

## V

- V-type constant 120
- values in operands 202
- variable symbols 27
- variable symbols as points of substitution 163
- variable symbols, system
  - &SYSDATE 183
  - &SYSECT 184
  - &SYSLIST 185
  - &SYSLOC 191
  - &SYSNDX 187
  - &SYSPARM 189
  - &SYSTIME 190
  - summary of 273

## W

- WXTRN instruction 72

## Y

- Y-type constant 117

Assembler H Version 2  
Language Reference  
GC26-4037-1

This manual is part of a library that serves as a reference source for system analysts, programmers, and operators of IBM systems. You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, are deemed appropriate.

**Note: Do not use this form to request IBM publications. If you do, your order will be delayed because publications are not stocked at the address printed on the reverse side. Instead, you should direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.**

If you have applied any technical newsletters (TNLs) to this book, please list them here: \_\_\_\_\_

Chapter/Section \_\_\_\_\_

\_\_\_\_\_ Page No. \_\_\_\_\_

Comments:

Note: Staples can cause problems with automated mail sorting equipment.  
Please use pressure sensitive or other gummed tape to seal this form.

If you want a reply, please complete the following information.

Name \_\_\_\_\_ Phone No. (\_\_\_\_) \_\_\_\_\_

Company \_\_\_\_\_

Address \_\_\_\_\_

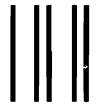
Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Reader's Comment Form

Fold and tape

Please do not staple

Fold and tape



NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES

**BUSINESS REPLY MAIL**  
FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE

IBM Corporation  
P.O. Box 49023  
Programming Publishing  
San Jose, California 95161-9023



Fold and tape

Please do not staple

Fold and tape

