GC33-4010-5
File No. S370-21

Systems

OS/VS – DOS/VSE – VM/370 Assembler Language

IBM

Sixth Edition (March 1979)

This edition, as amended by technical newsletter GN20-9372, applies to Release 4 of OS/VS1, Release 3 of OS/VS2, Release 2 of VM/370, DOS/VSE, and to any subsequent releases until otherwise indicated in new editions or technical newsletters.

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

Publications are not stocked at the address given below; requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California, U.S.A. 95150. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Read This First

This manual describes the OS/VS - DOS/VSE - VM/370 assembler language.

The OS/VS - VM/370 assembler language offers the following improvements over the OS/360 assembler language as processed by the F assembler:

1. New instructions and functions

2. Relaxation of language restrictions on character string lengths, attribute usage, SET symbol dimensions, and on the number of entries allowed in the External Symbol Dictionary

3. New system variable symbols

4. New options: for example, for the printing of statements in the program listings or for the alignment of constants and areas.

The figure on the following pages lists in detail these assembler language improvements and indicates the sections in the manual where the instructions and functions incorporating these improvements are described. If you are already familiar with the OS/360 assembler language as processed by the F assembler, you need only read those sections. Also included in the figure on the following pages are the improvements of the DOS/VS assembler language over the DOS/360 assembler language as processed by the D assembler.

NOTE: Sections I through L, describing the macro facility and the conditional assembly language, have been expanded to include more examples and detailed descriptions.

Note for VM/370 Users

The services provided by the OS Linkage Editor and Loader programs are paralleled in VM/370 by those provided by the CMS Loader. Therefore, for any reference in this publication to those OS programs, you may assume that the CMS Loader performs the same function.

Certain shaded notes in this publication refer to "OS only" information. Where you see these notes you may assume the information also applies for VM/370 users.

Note for DOS/VSE Users

All references to DOS and DOS/VS are also applicable to DOS/VSE.

COMPARISON OF ASSEMBLERS						
	Language Feature	Assemblers				Described in
		DOS/360 (D)	DOS/VSE	OS/360 (F)	OS/VS - VM/370	
1.	No of continuation lines allowed in one statement	1	2	2	2	B1B
2.	Location Counter value printed for EQU, USING, ORG (in ADDR2 field)	3 bytes	3 bytes	3 bytes	4 bytes (up to 3 leading zeros suppressed)	C4B
3.	Self-Defining Terms maximum value:	2^{24}-1	2^{24}-1	2^{24}-1	2^{31}-1	C4E
	number of digits binary: decimal: hexadecimal: character:	 24 8 6 3	 24 8 6 3	 24 8 6 3	 32 10 8 4	
4.	Relocatable and Absolute Expressions unary operators allowed: value carried: number of operators: levels of parentheses:	 no truncated to 24 bits 15 5	 yes truncated to 24 bits 15 5	 no truncated to 24 bits 15 5	 yes 31 bits 19 6	C6B
5.	Alignment of Constants (with no length modifier) when NOALIGN option specified:	ALIGN/ NOALIGN option not allowed	constants not aligned	constants aligned	constants not aligned	D2
6.	Extended Branching Mnemonics for RR format instructions:	no	yes	no	yes	D1H
7.	COPY Instruction nesting depth allowed: macro definitions copied:	 none no	 3 yes	 none no	 5 yes	E1A
8.	END Instruction generated or copied END instructions:	no	no	no	yes	E1
9.	All control sections initiated by a CSECT start at location 0 in listing and object deck	no	yes	no	no	E2C
10.	External Symbol Dictionary Entries maximum allowed:	255	511	255	399 (including entry symbols identified by ENTRY)	E2G
11.	DSECT Instruction blank name entry:	no	yes	no	yes	E3C
12.	DROP Instruction blank operand entry:	not allowed	signifies all current base registers dropped	not allowed	signifies all current base registers dropped	F1B
13.	EQU Instruction second operand as length attribute: third operand as type attribute:	 no no	 no no	 no no	 yes yes	G2A
14.	DC/DS Instruction; number of operands:	one	multiple	multiple	multiple	G3B

COMPARISON OF ASSEMBLERS					
Language Feature	Assemblers				Described in
	DOS/360 (D)	DOS/VSE	OS/360 (F)	OS/VS - VM/370	
15. Bit-length specification allowed:	no	yes	yes	yes	G3B
16. Literal Constants					G3C
multiterm expression for duplication factor:	no	yes	no	yes	
length, scale, and exponent modifier:	no	yes	no	yes	
Q- or S-type address constant:	no	no	no	yes	
17. Binary and Hexadecimal Constants number of nominal values:	one	one	one	multiple	G3D G3F
18. Q-type address constant allowed:	no	no	yes	yes	G3M
19. ORG Instruction name entry allowed:	sequence symbol or blank	sequence symbol or blank	sequence symbol or blank	any symbol or blank	H1A
20. Literal cross-reference:	no	yes	no	yes	H1B
21. CNOP Instruction symbol as name entry:	sequence symbol or blank	sequence symbol or blank	only sequence symbol or blank	any symbol or blank	H1C
22. PRINT Instruction inside macro definition:	no	yes	no	yes	H3A
23. TITLE Instruction number of characters in name (if not a sequence symbol):	4	4	4	8	H3B
24. OPSYN Instruction:	no	no	yes	yes	H5A
25. PUSH and POP Instructions for saving PRINT and USING status:	no	no	no	yes	H6
26. Symbolic Parameters and Macro Instruction Operands maximum number:	100	200	200	no fixed maximum	J2C K1B
mixing positional and keyword:	all positional parameters or operands must come first	all positional parameters or operands must come first	all positional parameters or operands must come first	keyword parameters or operands can be interspersed among positional parameters or operands	J3C K3C
27. Generated op-codes START, CSECT, DSECT, COM allowed	no	yes	no	yes	J4B
28. Generated Remarks due to generated blanks in operand field:	no	no	no	yes	J4B
29. MNOTE Instruction in open code:	no	no	no	yes	J5D
30. System Variable Symbols					J7
&SYSPARM:	yes	yes	no	yes	
&SYSDATE:	no	no	no	yes	
&SYSTIME:	no	no	no	yes	
31. Maximum number of characters in macro instruction operand:	127	255	255	255	K5
32. Type and Count Attribute of SET symbols:	no	no	no	yes	L1B
&SYSPARM, &SYSNDX, &SYSECT. &SYSDATE, &SYSTIME:	no	no	no	yes	

COMPARISON OF ASSEMBLERS

Language Feature	Assemblers				Described in
	DOS/360 (D)	DOS/VSE	OS/360 (F)	OS/VS - VM/370	
33. SET Symbol Declaration					L2
global and local mixed:	no, global must precede local	no, global must precede local	no, global must precede local	yes	
global and local must immediately follow prototype statement, if in macro definition:	yes	yes	yes	no	
must immediately follow any source macro definitions, if in open code:	yes	yes	yes	no	
34. Subscripted SET Symbols					L2
maximum dimension:	255	4095	2500	32,767	
35. SETC Instruction					L3B
duplication factor in operand:	no	no	no	yes	
maximum number of characters assigned	8	255	8	255	
36. Arithmetic Expressions in conditional assembly					L4A
unary operators allowed:	no	yes	no	yes	
number of terms:	16	16	16	up to 25	
levels of parentheses:	5	5	5	up to 11	
37. ACTR Instruction allowed anywhere in open code and inside macro definitions:	no, only immediately after global and local SET symbol declarations	yes	no, only immediately after global and local SET symbol declarations	yes	L6C
38. Options for Assembler Program					
ALIGN	no	yes	yes	yes	D2
ALOGIC	no	no	no	yes	L8
MCALL	no	no	no	yes	J8B
EDECK	no	yes	no	no	Order No. GC33-4024
MLOGIC	no	no	no	yes	L8
LIBMAC	no	no	no	yes	J8A

Preface

This is a reference manual for the OS/VS - DOS/VSE - VM/370 assembler language. It will enable you to answer specific questions about language functions and specifications. In many cases it also provides information about the purpose of the instruction you refer to, as well as examples of its use.

The manual is not intended as a text for learning the assembler language.

Who This Manual Is For

This manual is for programmers coding in the OS/VS - VM/370 or DOS/VSE assembler language.

Major Topics

This manual is divided into four main parts (aside from the "Introduction" and the Appendixes):

PART I (Sections B and C) describes the coding rules for, and the structure of, the assembler language.
PART II (Section D) describes the machine instruction types and their formats.
PART III (Sections E through H) describes the assembler instructions.
PART IV (Sections I through L) describes the macro facility and the conditional assembly language.

How To Use This Manual

Since this is a reference manual, you should use the Index or the Table of Contents to find the subject you are interested in.

Complete specifications are given for each instruction or feature of the assembler language (except for the machine instructions, which are documented in Principles of Operation -- see "References You May Need"). In many cases, a "Purpose" section suggests why you might use the feature; a "Specifications" section explains use of a complex feature; and one or more figures give examples of coding an instruction.

If you are a present user of the OS Assembler F or the DOS Assembler D, you need only read those sections listed in the table preceding this "Preface" which indicates those language features that are different from the DOS or OS System/360 languages.

TABS: Tabs mark the beginning of the specifications portion of the language descriptions. Use the tabs for quick referencing.

Tab - **USING**

OS-DOS DIFFERENCES: Wherever the OS/VS and DOS/VS assembler languages differ, the specifications that apply only to one assembler or the other are so marked. The 'OS only' markings also apply for the VM/370 assembler.

OS only

KEYS: The majority of figures are placed to the right of the text that describes them. Numbered keys within a figure are duplicated to the left of the text describing the figure. Use the numbered keys to tie the underlined passages in the text to specific parts of the figure.

Key - **3**

GLOSSARY: The glossary at the back of the manual contains terms that apply to assembler programming specifically and to allied terms in data processing in general. You can use the Glossary for terms that are unfamiliar to you.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard Vocabulary for Information Processing, which was prepared by Subcommittee X3.5 on Terminology and Glossary of American National Standards Committee X3.

References You May Need

You may want to refer to

IBM System/370 Principles of Operation, GA22-7000, or IBM 4300 Processors Principles of Operation for ECPS:VSE Mode, GA22-7070, the definitive publications for machine instructions, and to

OS/VS - VM/370 Assembler Programmer's Guide, GC33-4021

for detailed information about the OS/VS - VM/370 Assembler.

Guide to the DOS/VSE Assembler, GC33-4024

for detailed information about the DOS/VSE Assembler.

Contents

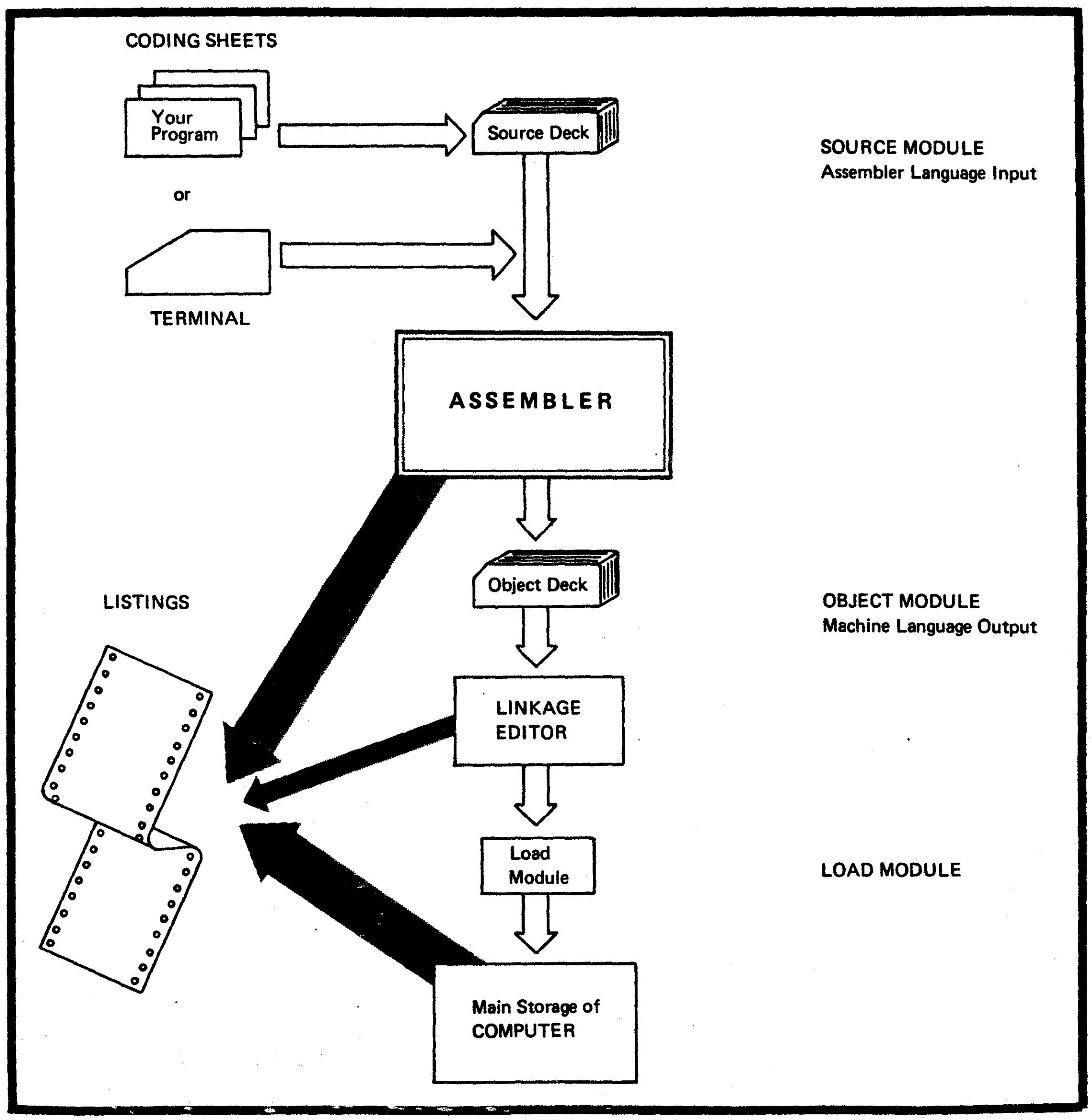
Section A: Introduction

What the Assembler Does

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 the 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 shown in the figure below.



Assume that your program, written in the assembler language, has been punched into a deck of cards called the source deck. This deck, also known as a source module, is the input to the assembler. (You can also enter a source module as input to the assembler through a terminal.)

The assembler processes your source module and produces an object module in machine language (called object code). Assume that the assembler punches this object module into a deck of cards called the object deck.

The object deck or 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, which then executes the program. Your source module and the object code produced is printed, along with other information on a program listing.

A1 – The 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 bit level or

- You must write subroutines for functions that are not provided by other symbolic programming languages such as: ALGOL, 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:

1. Machine instructions

2. Assembler instructions

3. Macro instructions.

Machine Instructions

A machine instruction is the symbolic representation of a machine language instruction of the IBM System/370 instruction set. It is called a machine instruction because the assembler translates it into the machine language code which the computer can execute. Machine instructions are described in PART II; SECTION C of this manual.

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 PART III; SECTIONS E, F, G, and H and PART IV; SECTIONS J, K, and L of this manual.

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 not described in this manual.)

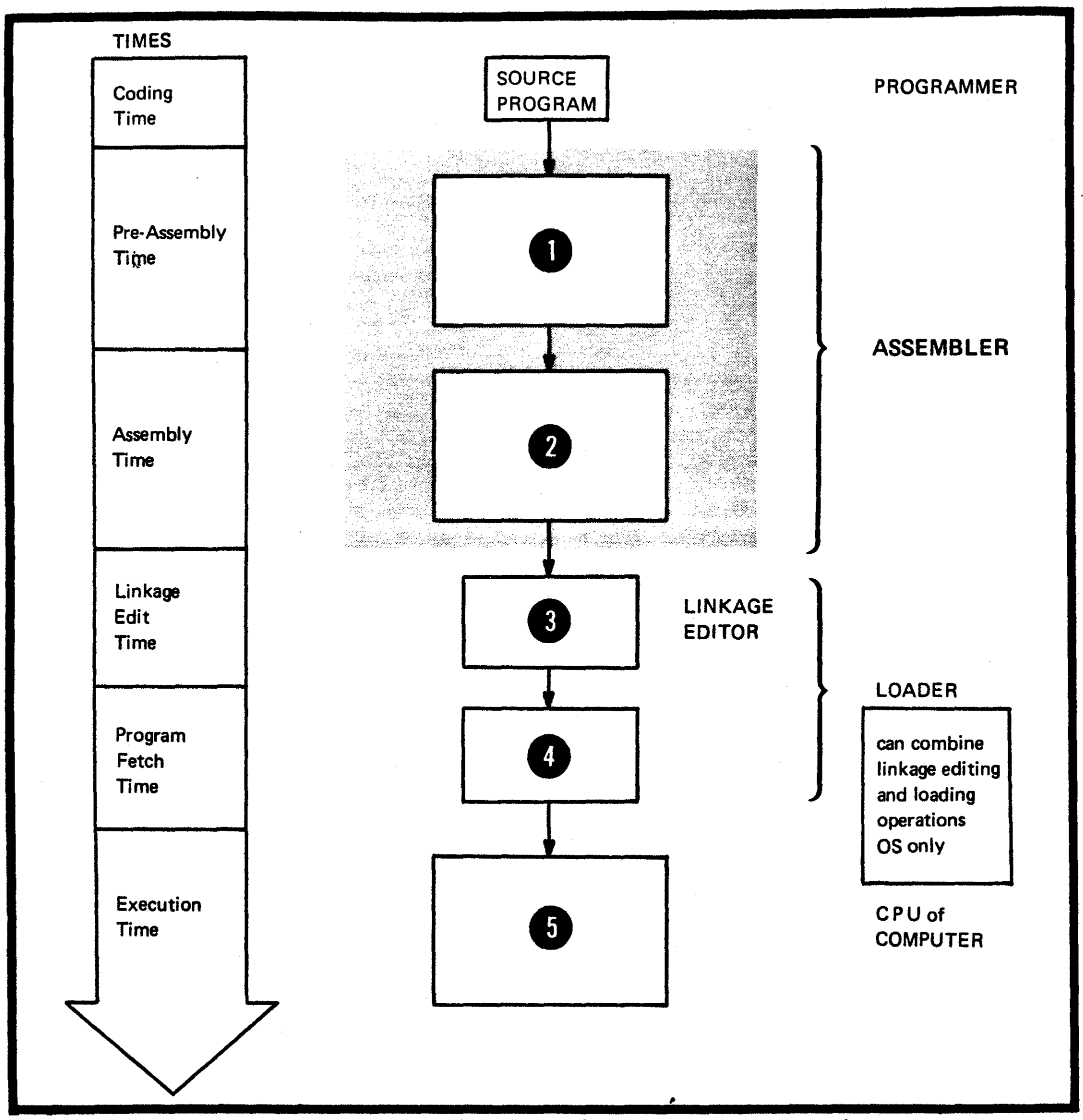
You can also prepare your own macro definitions and call them by coding the corresponding macro instructions. This macro facility is introduced in PART IV; SECTION I. A complete description of the macro facility, including the macro definition, the macro instruction and the conditional assembly language, is given in PART IV; SECTIONS J, K, and L.

A2 – The Assembler Program

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

A2A - ASSEMBLER PROCESSING SEQUENCE

The assembler processes the three types of 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 figure below relates the assembler processing sequence to the other times at which your program is processed and executed.

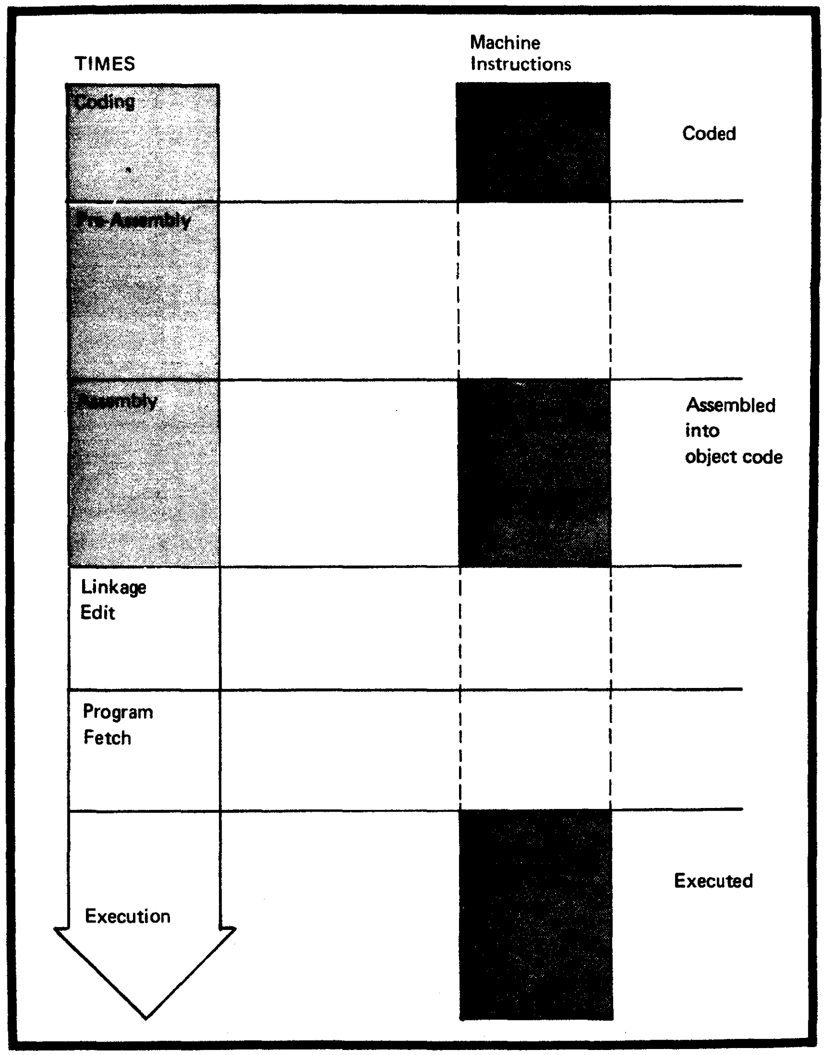


❶ The assembler processes most instructions on two occasions; first at pre-assembly time and later at assembly time. ❷ However, it does some processing, for example, macro processing, only at pre-assembly time.

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

Machine Instruction Processing

The assembler processes all machine instructions and translates them into object code at assembly time, as shown in the figure below.



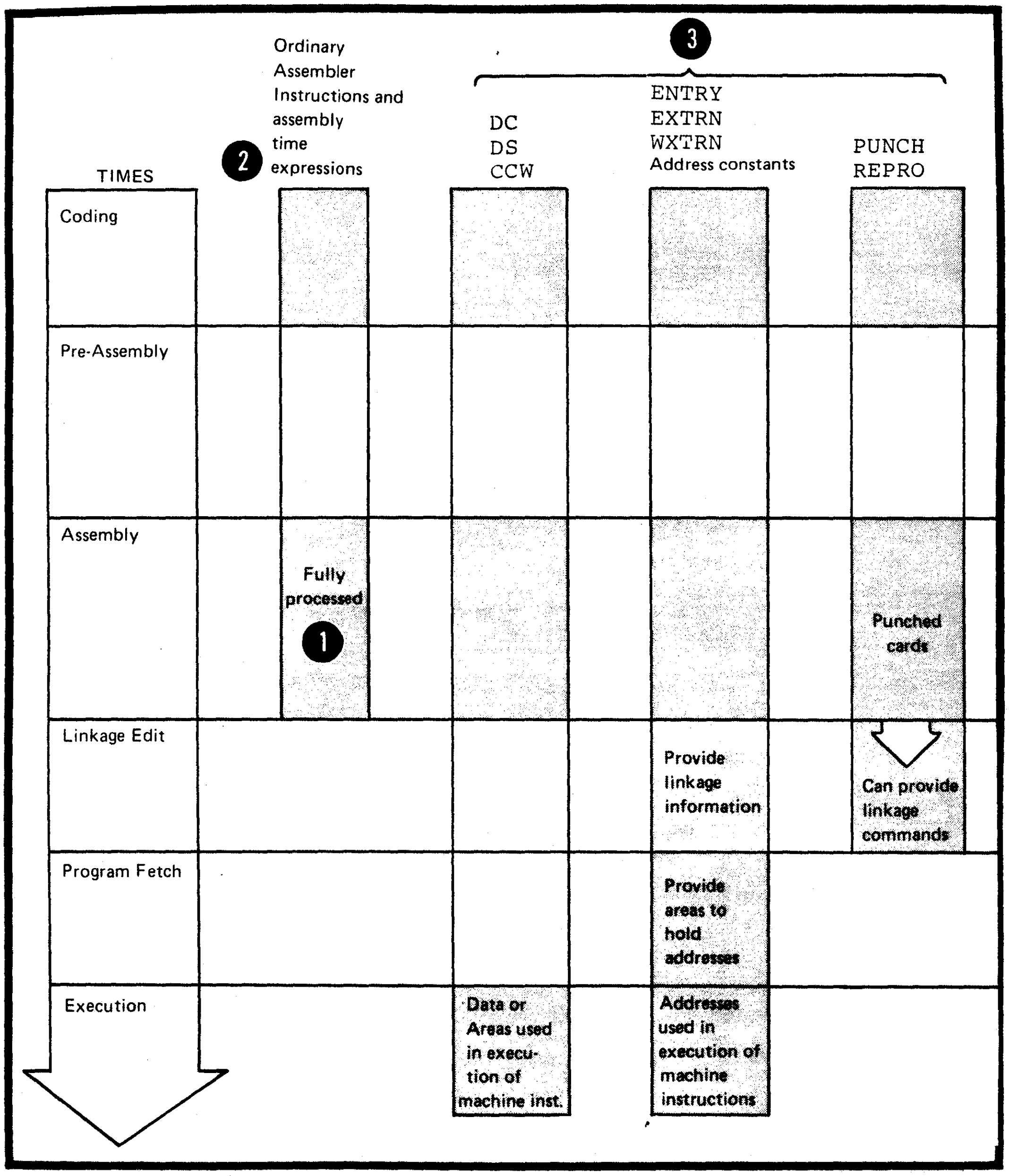
Assembler Instruction Processing

Assembler instructions are divided into two main types:

1. Ordinary assembler instructions

2. Conditional assembly instructions and the macro processing instructions (MACRO, MEND, MEXIT and MNOTE) .

❶ The assembler processes ordinary assembler instructions at assembly time, as shown in the figure below.

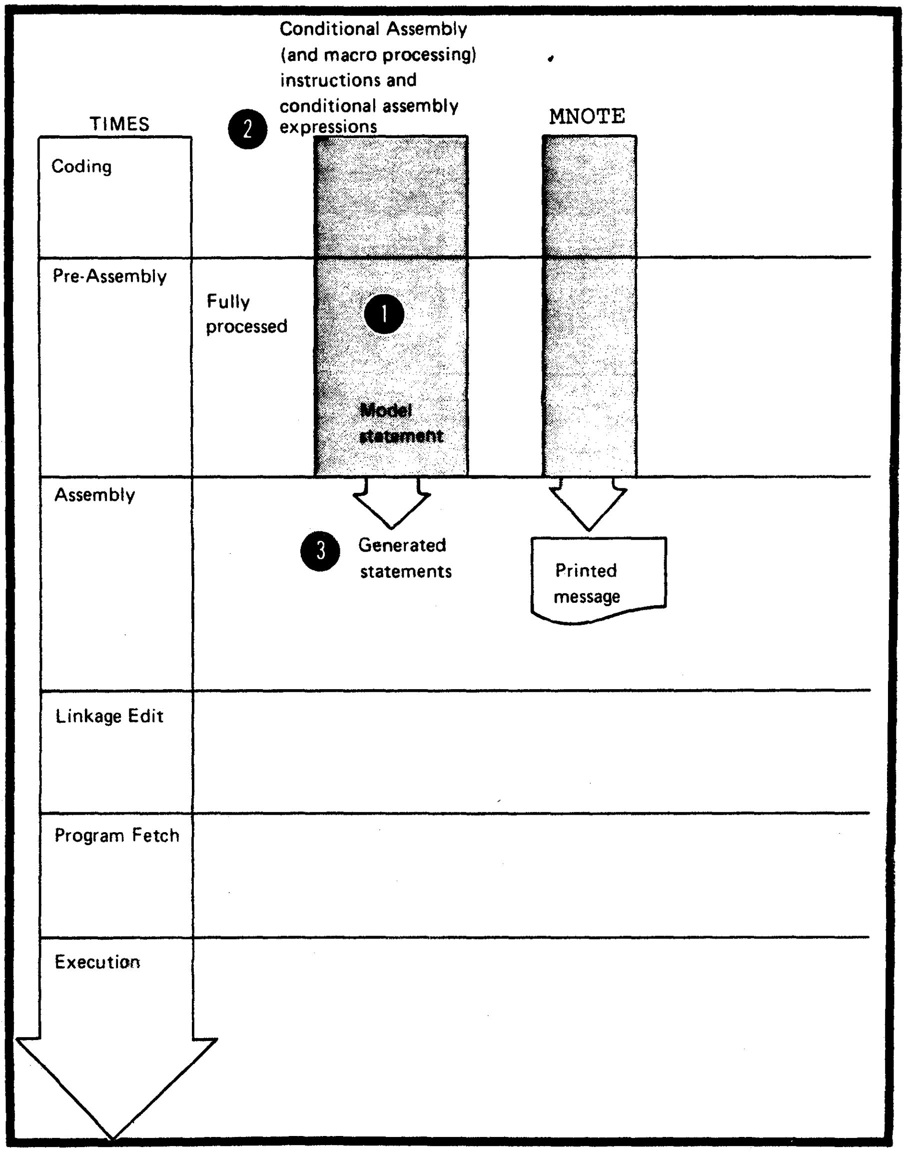


NOTES:

❷ 1. The assembler evaluates absolute and relocatable expressions at assembly time; they are sometimes called assembly time expressions.

❸ 2. Some instructions produce output for processing after assembly time.

1 The assembler processes conditional assembly instructions and macro processing instructions at pre-assembly time, as shown in the figure below.



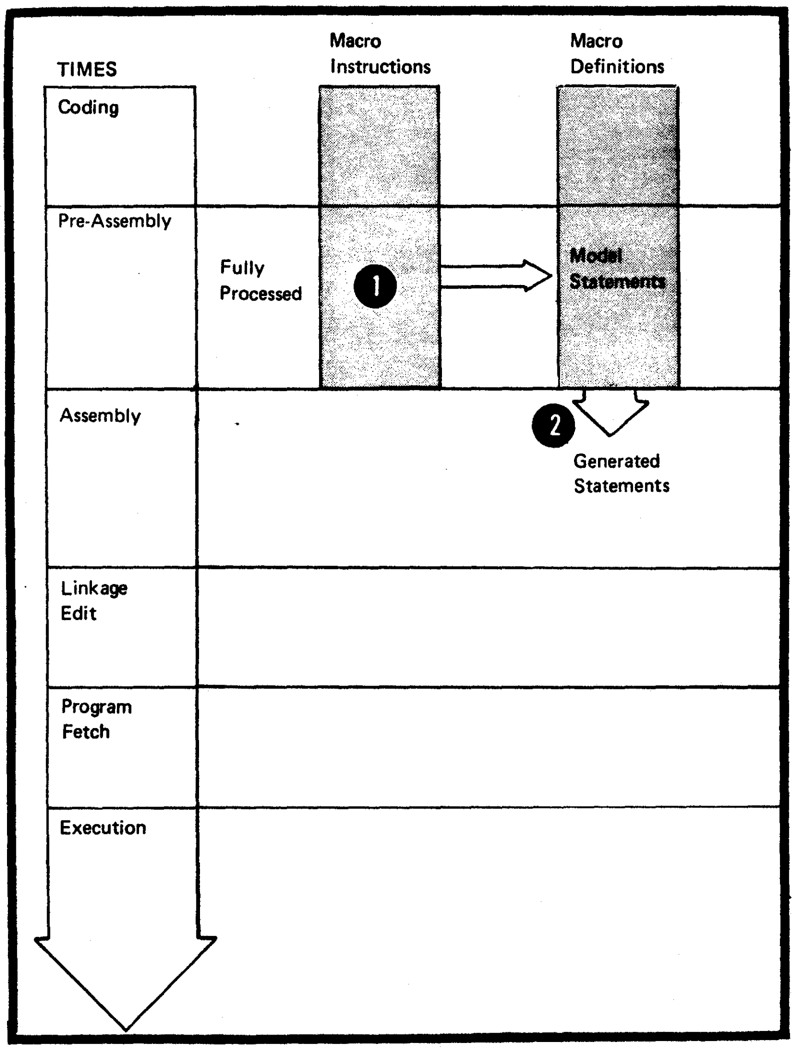
NOTES:

2 1. The assembler evaluates the conditional assembly expressions (arithmetic, logical, and character) at pre-assembly time.

3 2. The assembler processes the machine and assembler instructions generated from pre-assembly processing at assembly time.

Macro Instruction Processing

1 The assembler processes macro instructions at pre-assembly time, as shown in the figure below.



2 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 described in the above figures.

A3 - Relationship of Assembler to Operating System

The assembler is a programming component of the OS/VS, VM/370, or DOS/VS. These system control programs provide the assembler with the services:

- For assembling a source module and
- For 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.

<u>Services Provided by the Operating System</u>

OS/VS and DOS/VS provide the following services:

1. For assembling the source module:

   a. A control program

   b. Libraries to contain source code and macro definitions

   c. Utilities

2. For preparing for the execution of the assembler program as represented by the object module:

   a. A control program

   b. Storage allocation

   c. Input and output facilities

   d. A linkage editor

   e. A loader.

VM/370 provides the following services:

1. For assembling the source module:

   a. An interactive control program

   b. Files to contain source code and macro definitions

   c. Utilities.

2. For preparing for the execution of the assembler programs as represented by the object modules:

   a. An interactive control program

   b. Storage allocation

   c. Input and output facilities

   d. The CMS Loader.

A4 -- 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 of Data Representation

You can use decimal, binary, hexadecimal or character representation which the assembler will convert for you 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.

Segmenting 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.

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.

Program Listings

The assembler produces a listing of your source module, including any generated statements, and the object code assembled from the source module. You can control the form and content of the listing to a certain extent. The assembler also prints messages about actual errors and warnings about potential errors in your source module.

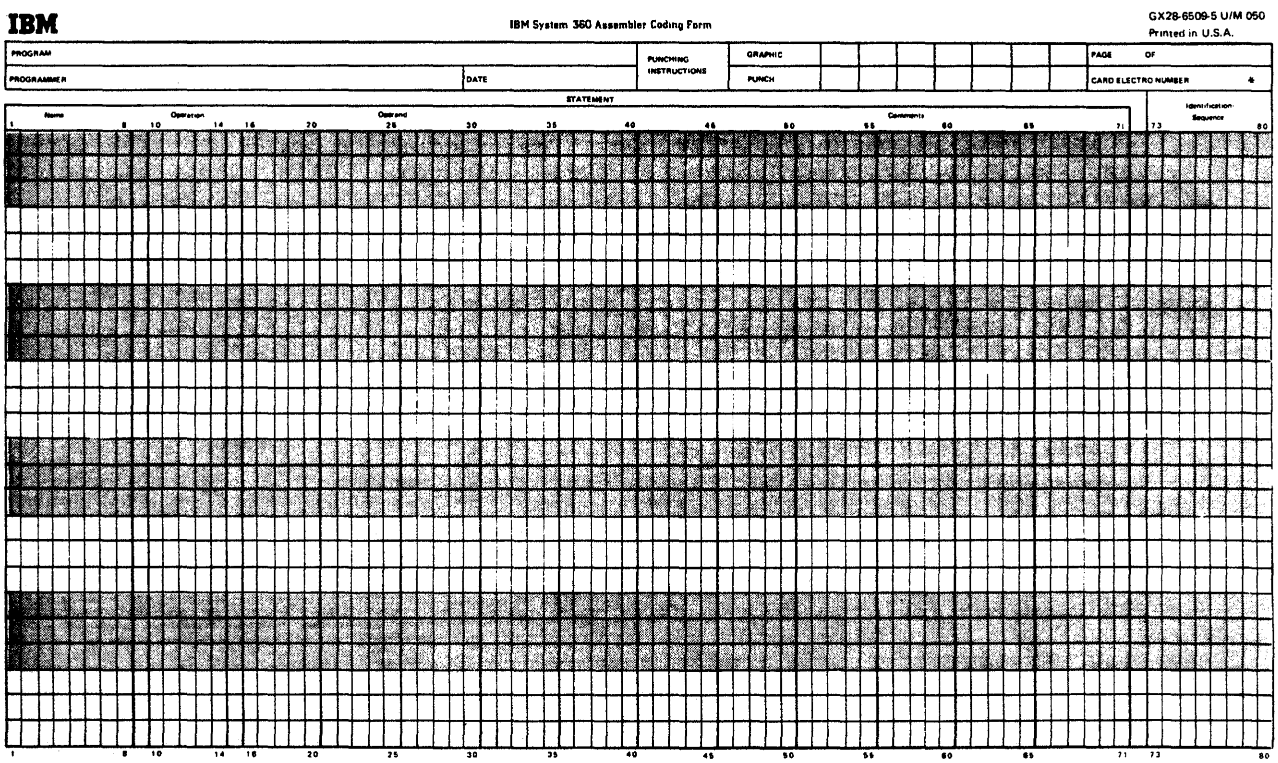
Part I: Coding and Structure

Section B: Coding Conventions

This section describes the coding conventions that you must follow in writing assembler language programs. Assembler language statements are usually written on a coding form before they are punched onto cards, or entered as source statements through other forms of input (for example, through terminals or directly onto tape).

Standard Assembler Coding Form

You can write assembler language statements on the standard coding form (Order No. GX28-6509) shown below. The columns on this form correspond to the columns on a punched card or positions on a source statement entered through a terminal. The form has space for program identification and instructions to keypunch operators.



* A standard card form, IBM electro 6509, is available for punching source statements from this form. Instructions for using this form are in any IBM System/360 Assembler Reference Manual. Address comments concerning this form to IBM Nordic Laboratory, Publications Development, Box 962 S - 181 09 Lidingo 9, Sweden.

B1A - FIELD BOUNDARIES

Assembler language statement usually occupy one 80-column line on the standard form (for statements occupying more than 80 columns, see B1B below). Note that any printable character punched into any column of a card, or otherwise entered as a position in a source statement, is reproduced in the listing printed by the assembler. All characters are placed in the line by the assembler. Whether they are printed or not depends on the printer. Each line of the coding form is divided into three main fields:

Stmnt Field

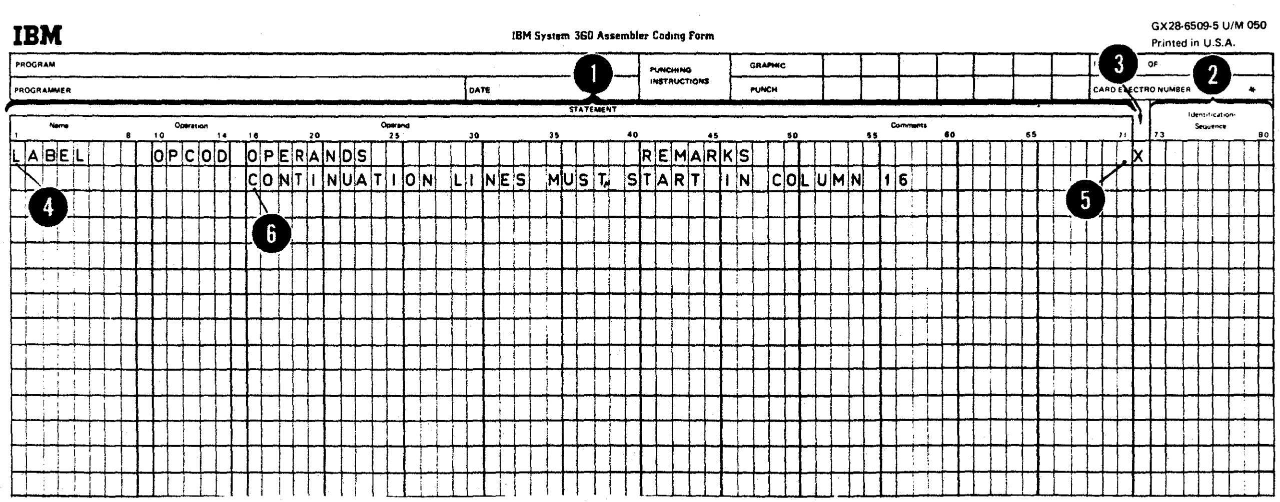
1. The Statement field,
2. The Identification -Sequence field, and
3. The Continuation Indicator field.

The Statement Field

The instructions and comments statements must be written in the statement field. The statement field starts in the "begin" column and ends in the "end" column. Any continuation lines needed must start in the "continue" column and end in the "end" column. The assembler assumes the following standard values for these columns:

4. • The "begin" column is column 1
5. • The "end" column is column 71, and
6. • The "continue" column is column 16.

These standard values can be changed by using the ICTL instruction. However, all references to the "begin", "end", and "continue" columns in this manual refer to the standard value described above.



The 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.

NOTE: The field the assembler normally checks lies in columns 73 through 80. However, if the ICTL instruction has been used to change the begin and end columns, the boundaries for the identification-sequence field can be affected.

The 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.

Field Positions

The statement field always lies between the begin and the end columns. The continuation indicator field always lies in the column after the end column. The identification-sequence field usually lies in the field after the continuation indicator field. However, 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.

B1B - CONTINUATION LINES

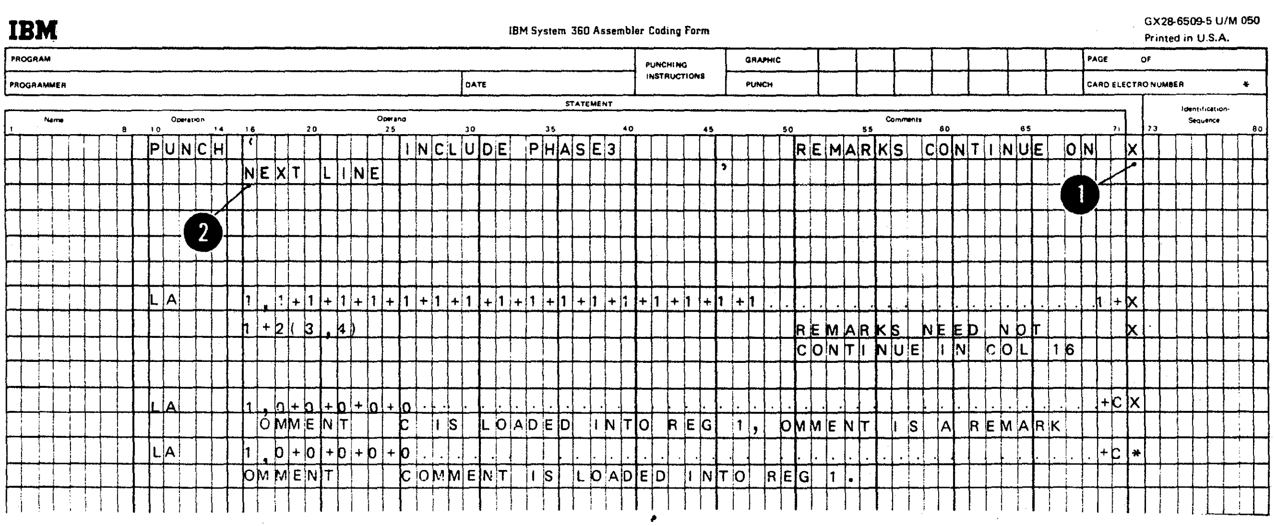
To continue a statement on another line, the following applies:

1 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 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 the user tries 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.

Only two continuation lines are allowed for a single assembler language statement. However, macro instruction statements and the prototype statement of macro definitions can have as many continuation lines as needed.



B1C - COMMENTS STATEMENT FORMAT

Comments

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:

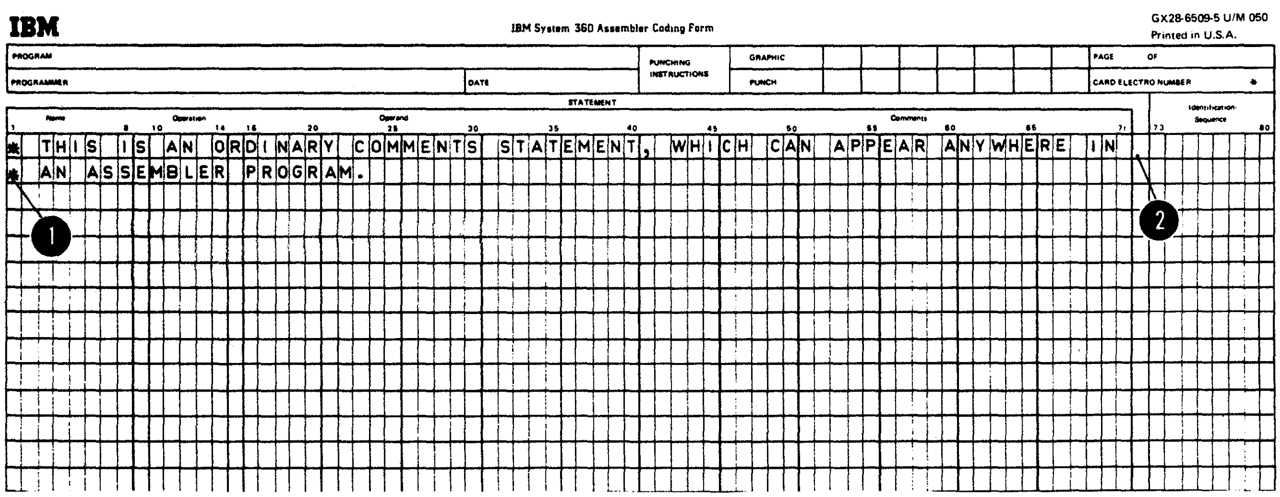
(1) 1. Comments statements require an asterisk in the begin column.

NOTE: Internal macro definition comments statements require a period in the begin column, followed by an asterisk (for details see J6A).

2. Any characters, including blanks and special characters, of the IBM System/370 Character Set (see C3) can be used.

(2) 3. 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.

4. Comments statements must not appear between an instruction statement and its continuation lines.



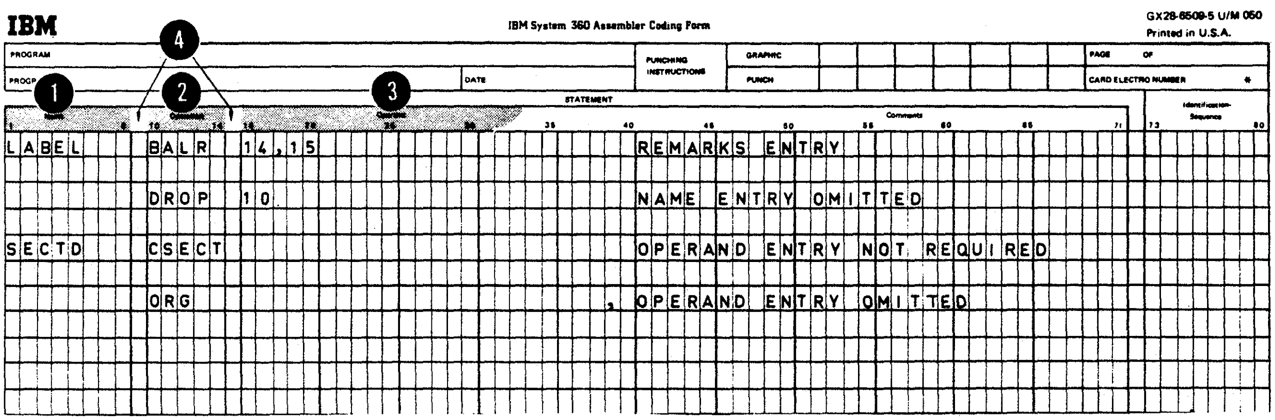
B1D -- INSTRUCTION STATEMENT FORMAT

The statement field of an instruction statement must be formatted to include from one to four of the following entries:

1. A name entry
2. An operation entry
3. An operand entry
4. A remarks entry.

Fixed Format

The standard coding form is divided into fields that provide fixed positions for the first three entries, as follows:



1. An 8-character name field starting in column 1.
2. A 5-character operation field starting in column 10.
3. An operand field that begins in column 16.
4. Note that 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.

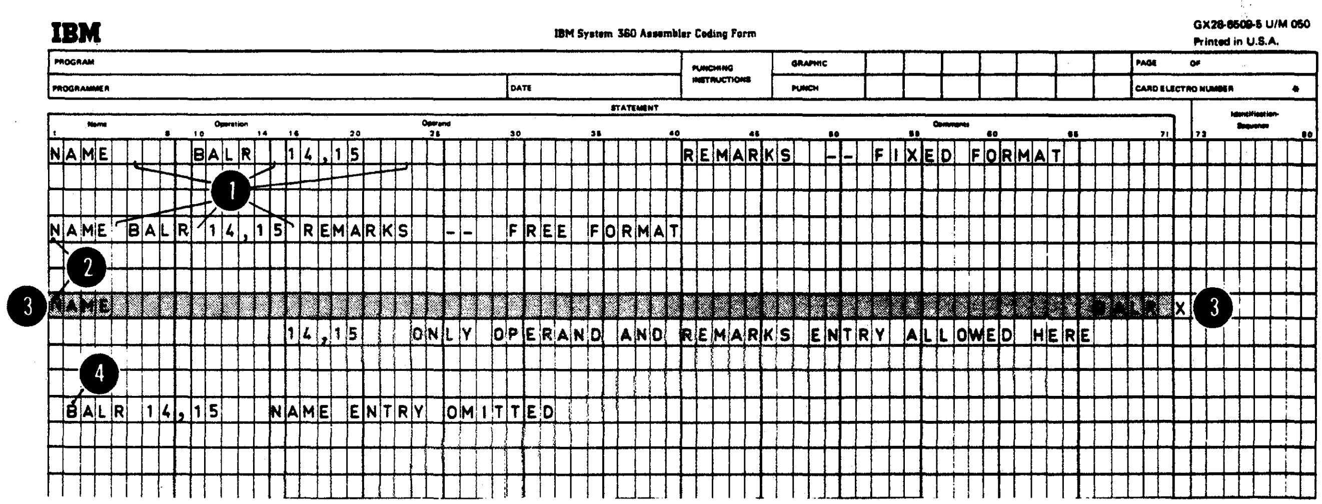
2. The entries must be contained in the begin column (1) through the end column (71) of the first line and, if needed, in the continue column (16) through the end cclurn (71) of any continuation lines.

(1) 3. The entries must be separated from each other by one or more blanks.

(2) 4. If used, the name entry must start in the begin column.

(3) 5. The name and operation entries, each followed by at least one blank, must be contained in the first line of an instruction statement.

(4) 6. The operation entry must start at least one column to the right of the begin column.



THE NAME ENTRY: The name entry identifies an instruction statement.

The following applies to the name entry:

1. It is usually optional.

2. It must be a valid symbol at assembly time (after substitution for variable symbols, if specified); for an exception see the TITLE instruction (H3E).

THE OPERATION ENTRY: The operation entry provides the symbolic operation code that specifies the machine, assembler, or macro instruction to be processed. The following applies to the operation entry:

1. It is mandatory.

2. 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 Appendixes IV and V).

OS only **NOTE: The standard set of codes can be changed by OPSYN instructions (as described in H5).**

3. For macro instructions it can be any valid symbol that is not identical to the operation codes described in 2 above.

THE OPERAND ENTRY: The operand entry has one or more operands that identify and describe the data used by an instruction. The following applies to operands:

1. One or more operands are usually required, depending on the instruction.

2. Operands must be separated by commas. No blanks are allowed between the operands and the commas that separate them.

3. 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 apostrophes (for example, C'J N') or in logical expressions (see L4C).

THE REMARKS ENTRY: The remarks entry is used to describe the current instruction. The following applies to the remarks entry:

1. It is optional.

2. It can contain any of the 256 characters (or punch combinations) of the IBM System/370 character set, including blanks and special characters.

(1) 3. It can follow any operand entry.

4. If an optional operand entry is omitted, remarks are allowed if the absence of the operand entry is indicated (2) by a comma, preceded and followed by one or more blanks.

IBM System 360 Assembler Coding Form — GX28-6509-5 U/M 050, Printed in U.S.A.

```
ALWAYS   LR    10,8                         REMARKS MUST BE SEPARATED FROM
         SR    10,9                         AN OPERAND ENTRY BY ONE OR MORE
         PUNCH 'LABEL OPCODE OPR1,OPR2'     BLANKS.

OMIT     START                            , COMMA INDICATES ABSENCE OF OPND
NONO1    CSECT                            , REMARKS
NONO2    END                              , REMARKS
```

Section C: Assembler Language Structure

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

C1 -- The Source Module

A source module is a sequence of assembler language statements that constitute the input to the assembler. The figure on the opposite page shows an overall picture of the structure of the assembler language.

C2 - Instruction Statements

The instruction statements of a source module are composed of one to four entries that are contained in the statement field. Other entries outside the statement field are discussed in B1A. The four statement entries are:

1. A name entry (usually optional)

2. An operation entry (mandatory)

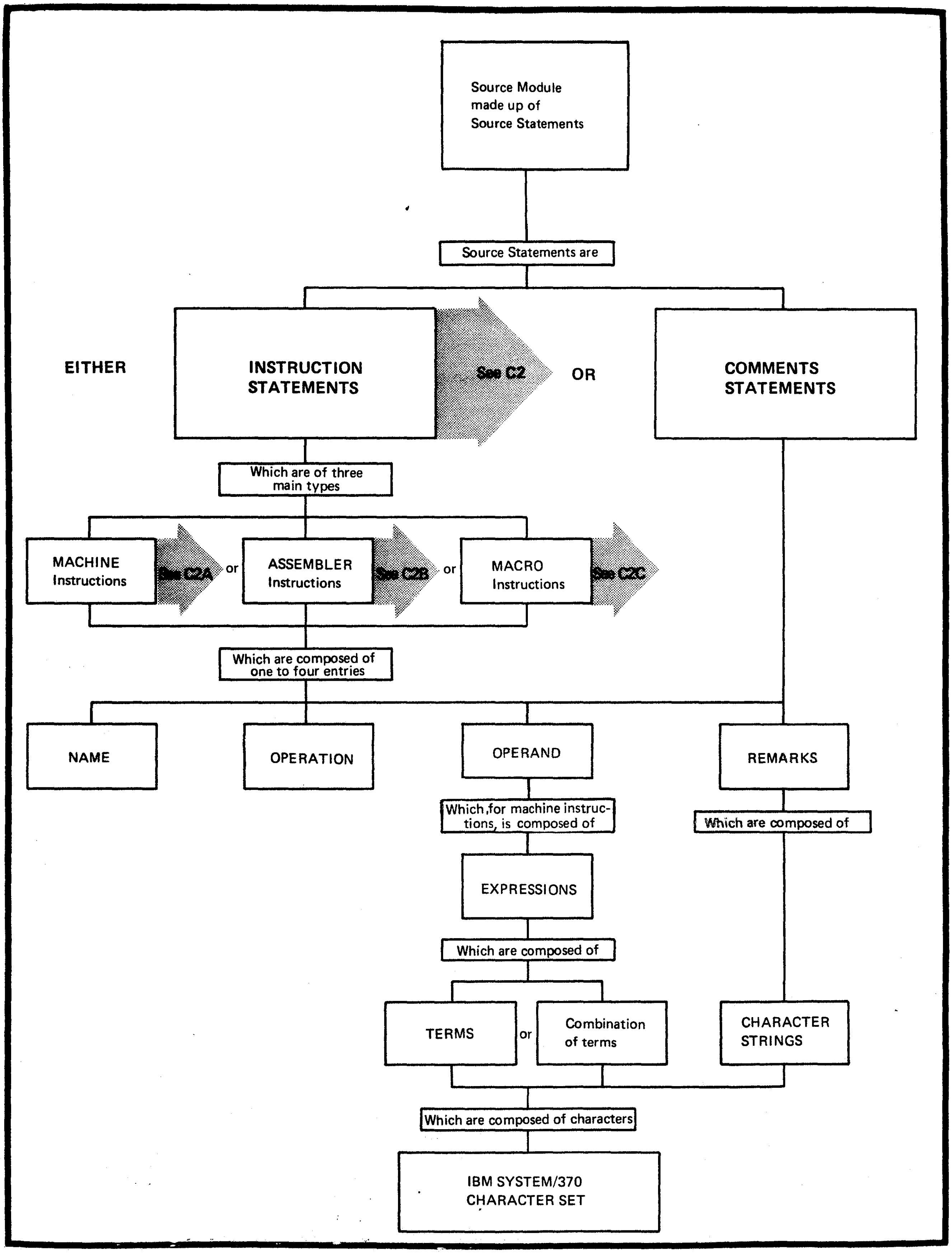
3. An operand entry (usually required)

4. A remarks entry (optional).

NOTES:

1. The figures in this subsection show the overall structure of the statements that represent the assembler language instructions and are not specifications for these instructions. The individual instructions, their purposes, and their specifications are described in other sections of this manual (as cross-referenced in the figures). Model statements, used to generate assembler language statements, are described in J4.

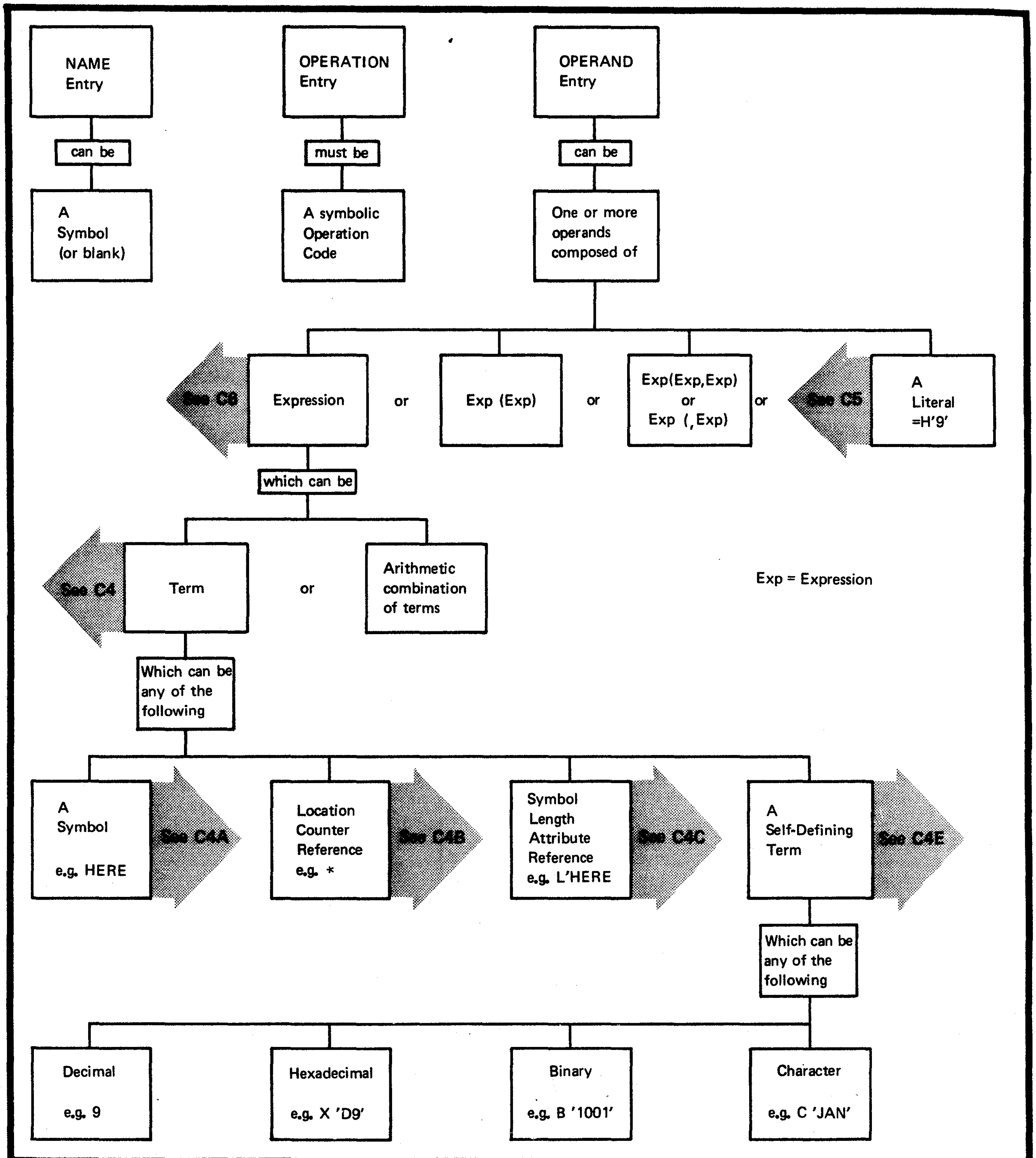
2. The remarks entry is not processed by the assembler, but only copied into the listings of the program. It is therefore not shown except in the overview opposite.


Source Module made up of Source Statements
Source Statements are
EITHER
INSTRUCTION STATEMENTS
See C2
OR
COMMENTS STATEMENTS
Which are of three main types
MACHINE Instructions
See C2A
or
ASSEMBLER Instructions
See C2B
or
MACRO Instructions
See C2C
Which are composed of one to four entries
NAME
OPERATION
OPERAND
REMARKS
Which,for machine instructions, is composed of
Which are composed of
EXPRESSIONS
Which are composed of
TERMS
or
Combination of terms
CHARACTER STRINGS
Which are composed of characters
IBM SYSTEM/370 CHARACTER SET

C2A -- MACHINE INSTRUCTIONS

The machine instruction statements are described in the figure below.

The instructions themselves are discussed in Part II of this manual and summarized in Appendix IV.



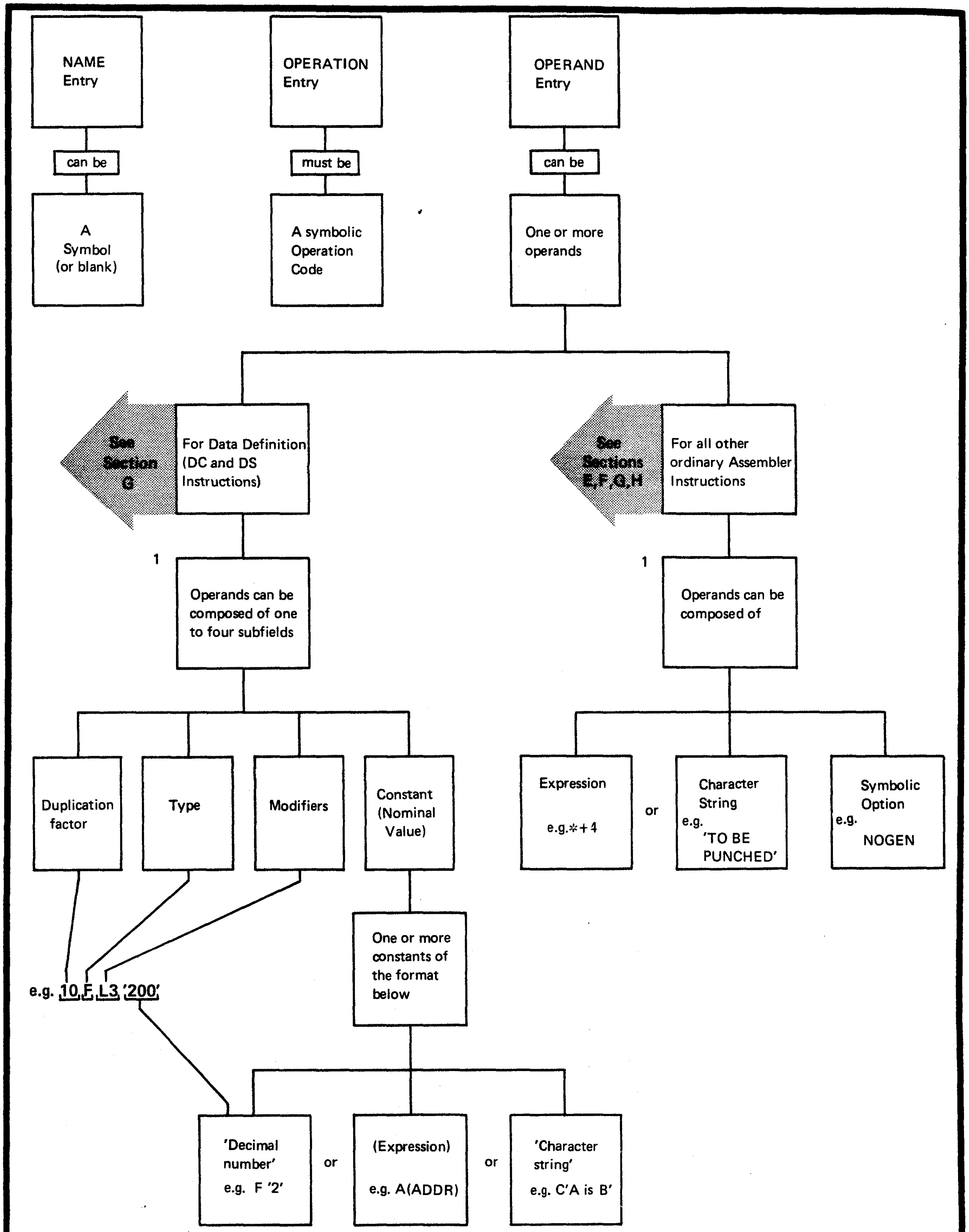
C2B -- ASSEMBLER INSTRUCTIONS

The assembler instruction statements can be divided into two main groups: ordinary assembler instructions and conditional assembly instructions.

Ordinary Assembler Instructions

Ordinary assembler instruction statements are described in the figure on the opposite page.

These instructions are discussed in Part III of this manual and summarized in Appendix V.

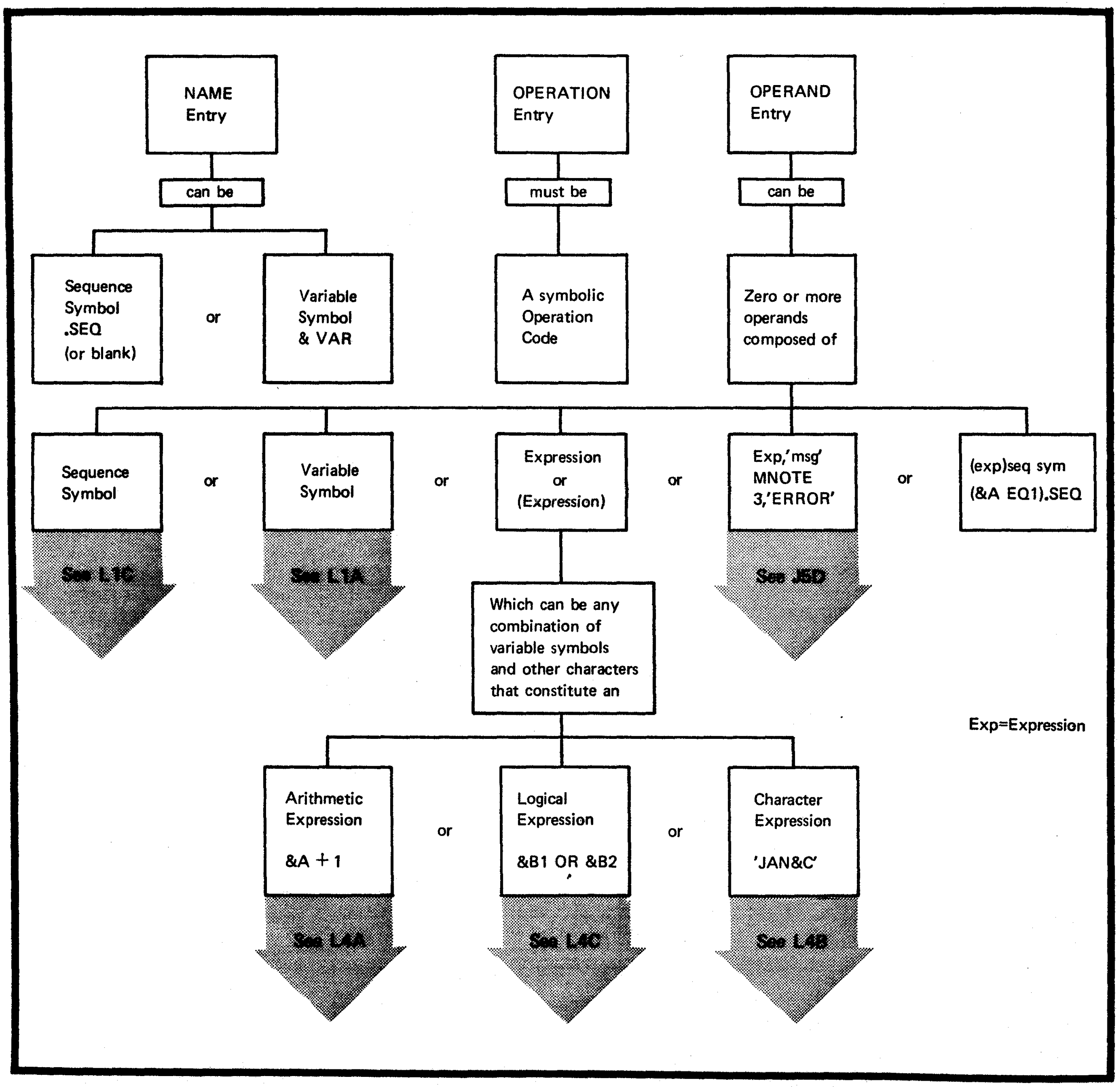


[1] Discussed more fully where individual instructions are described

Conditional Assembly Instructions

Conditional assembly instruction statements and the macrc processing statements (MACRO, MEND, MEXIT, MNOTE) are described in the figure below.

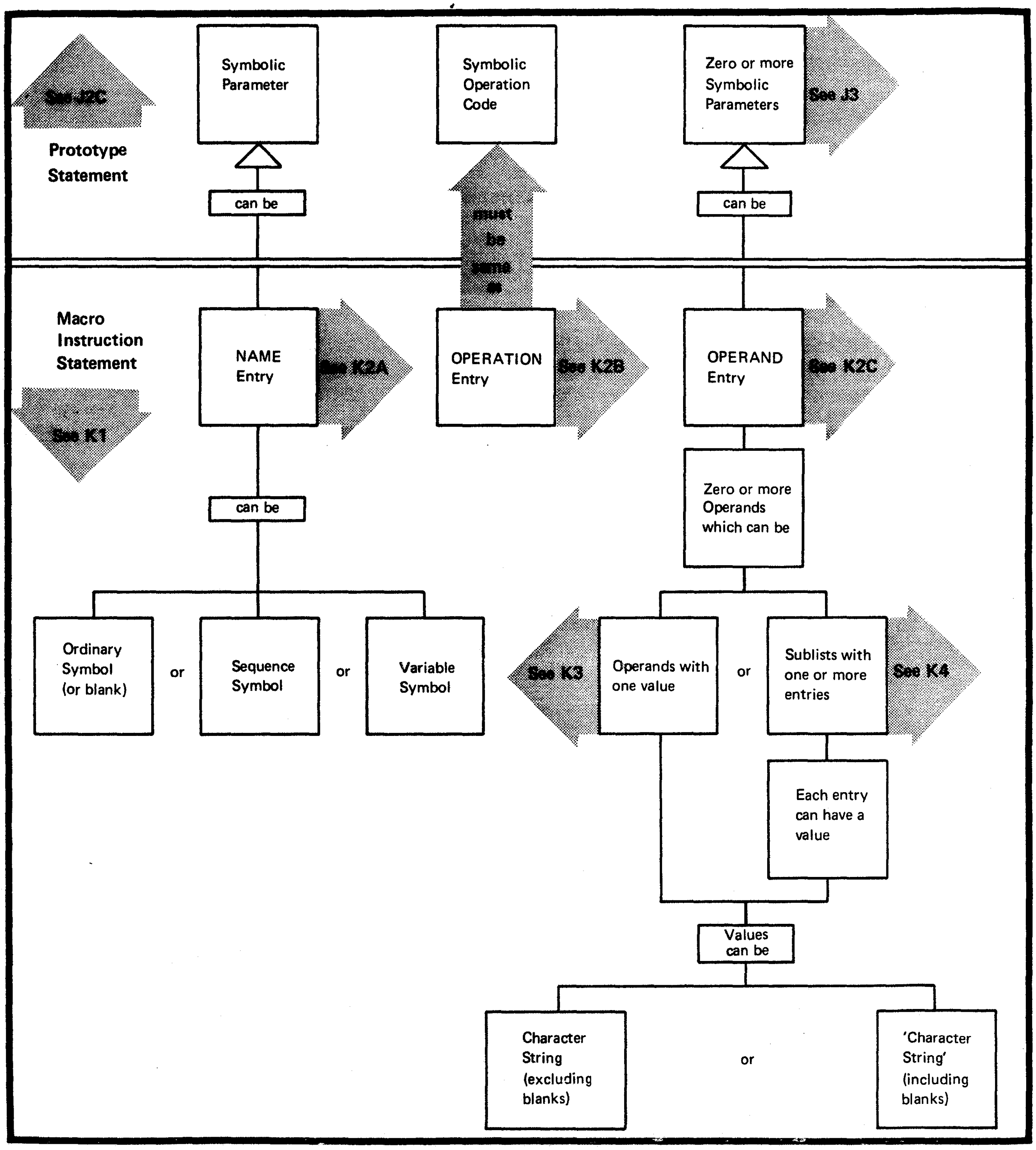
The conditional assembly instructions are discussed in Section L and macro processing instructions in Section J; both types are summarized in Appendix V.



C2C -- MACRC INSTRUCTIONS

Macro instruction statements are described in the figure below; the prototype statement of a macro definition, which serves as a model for the macro instruction statement, is also shown.

Macro instruction statements are discussed in Section K of this manual and the prototype statement is discussed in Section J2.



C3 - Character Set

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

1. Alphameric Characters

   Alphabetic characters (or letters): A through Z, and $, #, @

   Digits (or numerals): 0 through 9

2. Special characters

   + - , = . * ( ) ' / & blank

Examples, showing the use of the above characters are given in the figure below.

Normally, you would use strings of alphameric characters to represent data (terms, see C4), and special characters as:

   a. Arithmetic operators in expressions

   b. Data or field delimiters

   c. Indicators to the assembler for specific handling.

Characters are represented by the card-punch combinations and internal bit configurations listed in Appendix I. In addition to the printable characters listed above, any of the 256 combinations for punched cards listed in Appendix I can be used:

1. Between paired apostrophes

2. As statement remarks

3. In comments statements

4. In macro instruction operands (for restrictions see K5).

Characters	Usage	Example	Constituting
Alphameric	In symbols	LABEL NINE#01	Terms
Digits	As decimal self-defining terms	01 9	Terms
Special Characters	As Operators		
+	Addition	NINE+FIVE	Expressions
-	Subtraction	NINE-5	Expressions
*	Multiplication	9*FIVE	Expressions
/	Division	TEN/3	Expressions
+ or -	(Unary)	+NINE -FIVE	Terms
	As Delimiters		
Blanks	Between fields	LABEL AR 3,4	Statement
Comma	Between operands	OPND1,OPND2	Operand field
Apostrophes	Enclosing character strings	C'STRING'	String
Parentheses	Enclosing subfields or subexpressions	MOVE MVC TO(80),FROM (A+B*(C-D))	Statement Expression
	As indicators for		
Ampersand	Variable symbol	&VAR	Term
Period	Sequence symbol	.SEQ	(label)
	Comments statement in Macro definition	.*THIS IS A COMMENT	Statement
	Concatenation	&VAR.A	Term
	Bit-length specification	DC CL.7'AB'	Operand
	Decimal point	DC F'1.7E4'	Operand
Asterisk	Location counter reference	*+72	Expression
	Comments statement	* THIS IS A COMMENT	Statement
Equal sign	Literal reference	L 6,=F'2'	Statement
	Keyword	&KEY=D	Keyword Parameter

C4 -- 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 have absolute or relocatable values that are assigned by the assembler or 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. The various types of terms described below are summarized in the figure to the right.

Terms

Terms	Term Can Be		Value Is	
	Absolute	Relocatable	Assigned by Assembler	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

C4A -- SYMBOLS

Purpose

You can use a symbol to represent storage locations or arbitrary values.

SYMBOLIC REPRESENTATION: You can 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 D5. The advantages of symbolic over numeric representation are:

1. Symbols are easier to remember and use than numerical 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 numerical 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 (1) the number of the statement in which the symbol is defined (that is, used as the name entry) and (2) the numbers of all the statements in which the symbol is used in the operands.

<u>THE SYMBOL TABLE</u>: The assembler maintains an internal table called a symbol table. When the assembler processes your source statements for the first time, the assembler 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 (Attribute references used as terms are discussed in C4C and C4D below).

Specifications

Symbols

The three types of symbol recognized by the assembler are:

1. Ordinary symbols

2. Sequence symbols

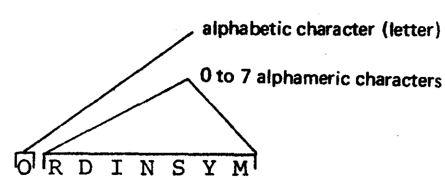
3. Variable symbols.

<u>ORDINARY SYMBOLS</u>: Ordinary symbols can be used in the name and operand field of machine and assembler instruction statements. They must be coded in the format shown in the figure to the right.

NOTES:

1. No special characters are allowed in an ordinary symbol.

2. No blanks are allowed in an ordinary symbol

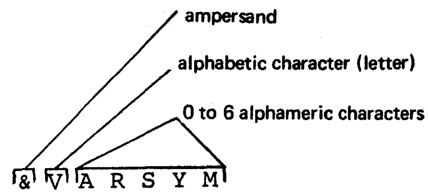


Examples:

HERE	#01	X
READER	#12	Y
A001	@33	Z
B002	$OPEN	F2A

Var. Sym.

VARIABLE SYMBOLS: Variable symbols can only be used in macro processing and conditional assembly instructions. They must be coded in the format shown in the figure to the right.

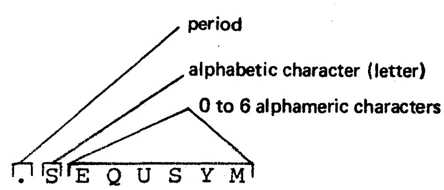


Examples:

&A &PARAM
&B &KEYWORD
&C &CHAR3

Seq. Sym.

SEQUENCE SYMBOLS: Sequence symbols can only be used in macro processing and conditional assembly instructions. They must be coded in the format shown in the figure to the right.



Examples:

.SEQ
.LOOP11
.EXIT20
.T0001

Symbol Definition

An ordinary symbol is considered defined when it appears as:

1. The name entry in a machine or assembler instruction of the assembler language.

2. One of the operands of an EXTRN or WXTRN instruction.

NOTE: Ordinary symbols that appear in instructions generated from model statements at pre-assembly time are also considered defined.

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:

   (1) a. Any machine or assembler instruction (except the EQU or OPSYN instructions)

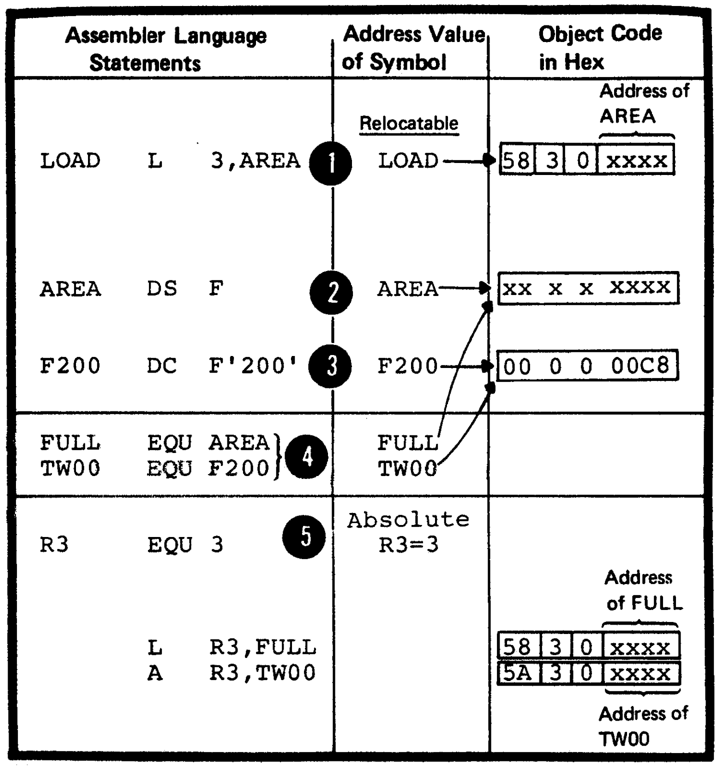
   (2) b. A storage area defined by the DS instruction

   (3) c. Any constant defined by the DC instruction

   d. A channel command word defined by the CCW instruction.

The address value thus assigned is relocatable, because the object code assembled from these items is relocatable; the relocatability of addresses is described in D5B.

2. According to the value of the first or only expression specified in the operand of an EQU instruction. This expression can have a (4) relocatable or absolute value, which is then assigned to the ordinary symbol. (5) The value of an ordinary symbol must lie in the range -2^{31} through $+2^{31}-1$.



Restrictions on Symbols

UNIQUE DEFINITION: A symbol must be defined only once in a source module:

1. either in the name field of a source statement
2. or in the operand field of an EXTRN or WXTRN instruction.

This is true even for a source module which contains two or more control sections.

OS only — NOTE: The ordinary symbol that appears in the name field of an **OPSYN** or 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.

CONTROL SECTION NAMES: A duplicate symbol can, however, be used as the name entry of a START, CSECT, DSECT, or COM instruction. The (3) first time a symbol is used to name these instructions, it identifies the beginning of the control section; (4) a duplicate use of the symbol identifies the resumption of an interrupted control section.

PREVIOUSLY DEFINED SYMBOL: In some instructions the symbols used in (5) their operands must have been defined in a previous instruction. Previously defined symbols are required for the operands of the following instructions:

EQU

CNOP

ORG

DC and DS (in modifier and duplication factor expressions).

```
(3) FIRST     START   0

              EXTRN   READER,PRINTER   (2)

    REG3      EQU     3                (5)
    INDEX     EQU     REG3
    REG4      EQU     4

(1) ENTRIES   L       REG4,TABLE(INDEX)

              B       SECOND

    TABLE     DS      CL256

    SECOND    CSECT

              L       REG3,ADRDR

              B       RESUME1

(1) ADRDR     DC      A(READER)

(4) FIRST     CSECT
(1) RESUME1   LA      INDEX,20

              END
```

C4B -- LOCATION COUNTER REFERENCE

Purpose

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.

THE LOCATION COUNTER: 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. (1)

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. (2)

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. (3)

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. (4)

The assembler maintains a location counter for each control section in a source module; for complete details about the location counter setting in control sections, see E2C. 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".

Location in Hex	Source Statements		
000004	DONE	DC	CL3'SOB'
(1) 000007 (2)	BEFORE	EQU	*
000008	(3) DURING	DC	F'200'
00000C (4)	AFTER	EQU	*
000010	NEXT	DS	D

NOTE: In the figure below, an example of a location counter overflow (or wrap-around) is shown.

1 The internal address value of the symbol B is carried as a 4-byte value, but the printed location only includes
2 the low-order 3 bytes.

3 The location counter value for instructions or constants is usually printed as a 3-byte value. However, the 4-byte
4 value, with up to 3 leading zeros suppressed, is printed for the addresses specified in the operands of the following instructions: EQU, ORG, and USING. Only 3-byte values
DOS are printed for the operands in the above instructions.

You can control the setting of the location counter in
5 a particular control section by using the START or ORG instructions.

Assembly Listings in Hexadecimal Representation

LOC	OBJECT CODE	ADDR1	ADDR2	STMT	SOURCE STATEMENT
000000				1	A START 0
000000			00FFFFFE	2	(5) ORG *+X'FFFFFE'
FFFFFE	58506004	00D08		3	L 5,4(,6)
	*** ERROR ***		(Location counter overflow)		
(2) 000002	07FF			4	B BR 15
000004	01000002 address of B			5	C DC A(B)
			01000004	6	D EQU C

(3) LOC column (1) object code (4) ADDR2 column

Up to 3 leading zeros are suppressed

Specifications

The lccaticn ccunter reference is specified by an asterisk (*). The asterisk can be specified as a relocatable term according to the following rules:

1. It can only be specified in the operands of:

   a. Machine instructions

   b. The DC and DS instructions

   c. The EQU, ORG, and USING instructions.

(1) 2. It can also be specified in literal constants (see C5).

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 (2) address of the first byte of the instruction in which it appears (for the value of the asterisk in address constants with duplication factors, see G3J).

Location in Hex	Source Statements				Address Value of *
000104	HERE	B	*+8	same effect	(2) HERE
000108		B	HERE+8		
00011C	CONSTANT	DC	A(*) (1)		CONSTANT
000120	THERE	L	3,=A(*)		THERE

C4C -- SYMBOL LENGTH ATTRIBUTE REFERENCE

Purpose

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:

1. Specify unknown storage area lengths

2. Cause the assembler to compute length specifications for you

3. Build expressions to be evaluated by the assembler.

Specifications

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:

1. For machine instructions, it assigns either 2, 4, or 6, depending on the format of the instruction.

2. For the DC and DS instructions, it assigns either the implicit or explicitly specified length. The length attribute is not affected by a duplication factor.

3. For the EQU instruction, it assigns the length attribute value of the leftmost or only term of the first expression in the first operand, unless a specific length attribute is supplied in a second operand.

DOS **Only one operand is allowed in the EQU instruction.**

Note the length attribute values of the following terms in an EQU instruction:

4. • self-defining terms

5. • location counter reference

6. • L'*

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

For the remaining assembler instructions, see the specifications for the individual instructions.

Length Attr.

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

C4D -- OTHER ATTRIBUTE REFERENCES

There are other attributes which 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'), scaling (S'), integer (I'), count (K'), and number (N') attributes.

NOTE: You can refer to these attributes only in conditional assembly instructions and expressions; for full details, see L1B.

C4E -- SELF-DEFINING TERMS

Purpose

A self-defining term allows you to specify a value explicitly. With self-defining terms, you can specify decimal, binary, hexadecimal, or character data. 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.

GENERAL RULES: Self-defining terms:

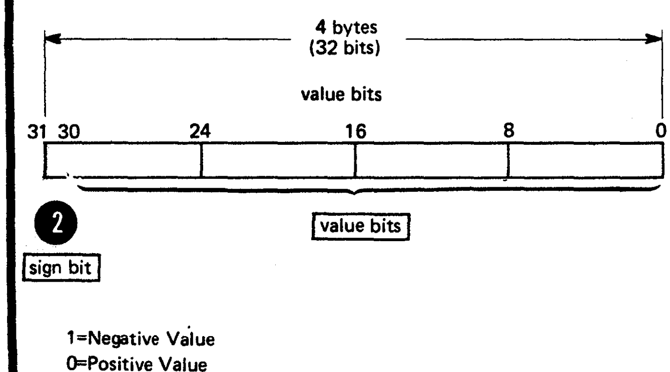
1 • Represent machine language binary values

• Are absolute terms; their values do not change upon program relocation.

Self-Defining Term	Decimal Value	Binary Value (1)
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

The assembler carries the values represented by self-defining terms to 4 bytes or 32-bits; the high-
2 order bit is the sign bit.

DOS Values are carried to 3 bytes or 24 bits.



DECIMAL: A decimal self-defining term is an unsigned decimal number. The assembler allows:

1 • High-order zeros

2 • A maximum of 10 decimal digits

• A range of values from 0 through
3 2,147,483,647.

DOS • A maximum of 8 decimal digits.

• A range of values from 0 through 16,777,215.

(1) 0002

(2) 2147483647 = $2^{31}-1$ (3)

BINARY: A binary self-defining term must be coded in the format shown in the figure to the right. The assembler:

1 • Assembles each binary digit as it is specified

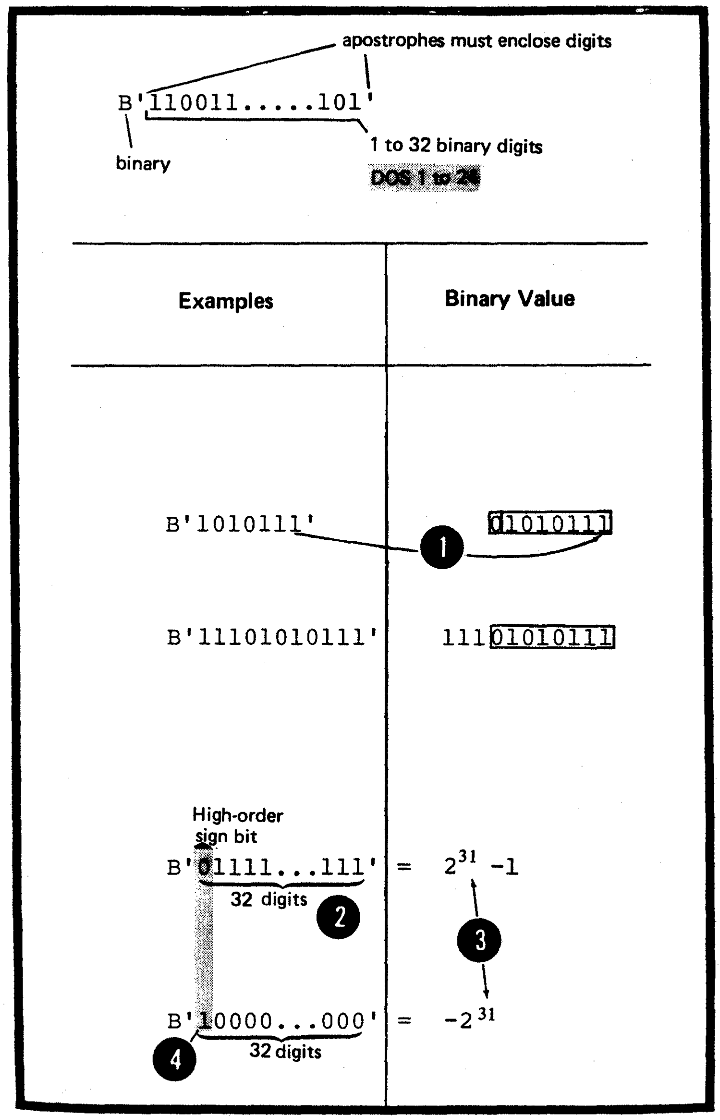
2 • Allows a maximum of 32 binary digits

3 • Allows a range of values from -2,147,483,648 through 2,147,483,647.

DOS • Allows a maximum of 24 binary digits.

• Allows a range of values from 0 through 16,777,215.

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



HEXADECIMAL: A hexadecimal self-defining term must be coded as shown in the figure to the right. The assembler:

1 • Assembles each hexadecimal digit into its 4-bit binary equivalent (listed in the figure to the right)

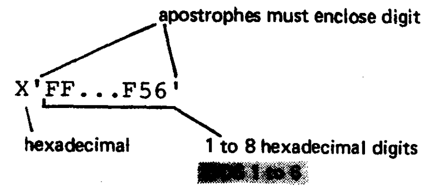
2 • Allows a maximum of 8 hexadecimal digits

3 • Allows a range of values from -2,147,483,648 through 2,147,483,647.

DOS • Allows a maximum of 6 hexadecimal digits.

• Allows a range of values from 0 through 16,777,215.

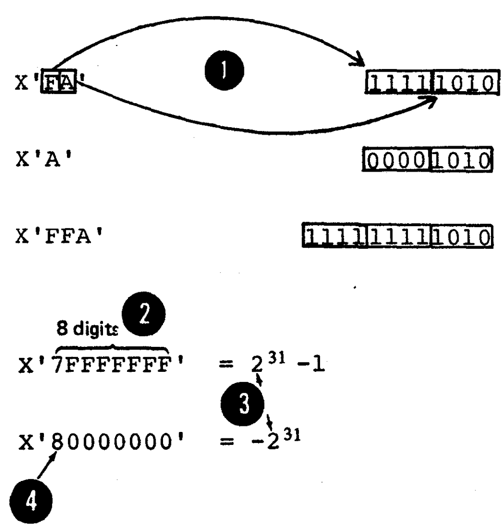
4 NOTE: When used as an absolute term in an expression, a hexadecimal self-defining term has a negative value if the high-order bit is 1.



Conversion Table:

Hexadecimal Digit	Decimal Equivalent	4-bit Binary Representation
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111

Examples: Binary Value



CHARACTER: A character self-defining term must be coded as shown in the figure to the right. The assembler:

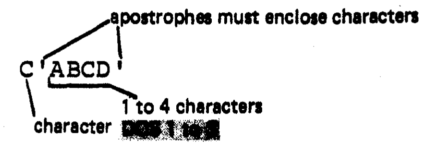
- Allows any of the 256 punch combinations when using punched cards as input. This includes the printable characters, that is, blanks and special characters.

1 • Assembles each character into its 8-bit binary equivalent. (A table of characters and their binary equivalents can be found in Appendix I).

2 • Requires that two ampersands or apostrophes be specified in the character sequence for each ampersand (3) or apostrophe required in the assembled term.

4 • Allows a maximum of 4 characters.

DOS • Allows a maximum of 3 characters.



Examples:

Character self-defining term	Characters Assembled	Hexadecimal Value	Binary Value
C'A'	A	X'C1'	11000001
C'1'	1	X'F1'	11110001
C' '	(blank)	X'40'	01000000
C'#'	#	X'7B'	01111011
C'@'	@	X'7C'	01111100
C'&&'	&	X'50'	01010000
C''''	'	X'7D'	01111101
C'L''A'	L'A	X'D37DC1'	
C''''''	''	X'7D7D'	
C'FOUR'	FOUR	X'C6D6E4D9'	

C5 – Literals

Purpose

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

1. The data you enter as numbers for computation, addresses, or messages to be printed is visible in the instruction in which the literal appears, and

2. You avoid defining constants elsewhere in your source module and then using their symbolic names in machine instruction operands.

```
L       1,=F'200'
L       2,=A(SUBRTN)
MVC     MESSAGE(16),=C'THIS IS AN ERROR'
```

The assembler assembles the data specified in a literal into a "literal pool" (fully described in H1B). 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: Do not confuse literals with constants or self-defining terms. They differ in three important ways:

1. In where you can specify them in machine instructions, that is, whether they represent data or an address of data.

2. In whether they have relocatable or absolute values.

3. In what is assembled into the object code of the machine instruction in which they appear.

The figure to the right illustrates the first two points.

(1) • A literal represents data.

(2) • A constant is represented by its relocatable address. Note that a symbol with an absolute value does not represent the address of (3) a constant, but represents immediate data (see D5D) or an absolute (4) address.

(5) • A self-defining term represents data and has an absolute value.

Compare:

A literal with a relocatable address

```
         L    3,=F'33'  (1) }
         L    3,F33     (2) } same effect
         .
         .
         .
F33      DC   F'33'
```

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

```
         MVC  FLAG,=X'00'  (1) }
         MVI  FLAG,X'00'   (5) } same effect
         MVI  FLAG,ZERO    (3) }
         .
         .
         .
FLAG     DS   X
ZERO     EQU  X'00'
```

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

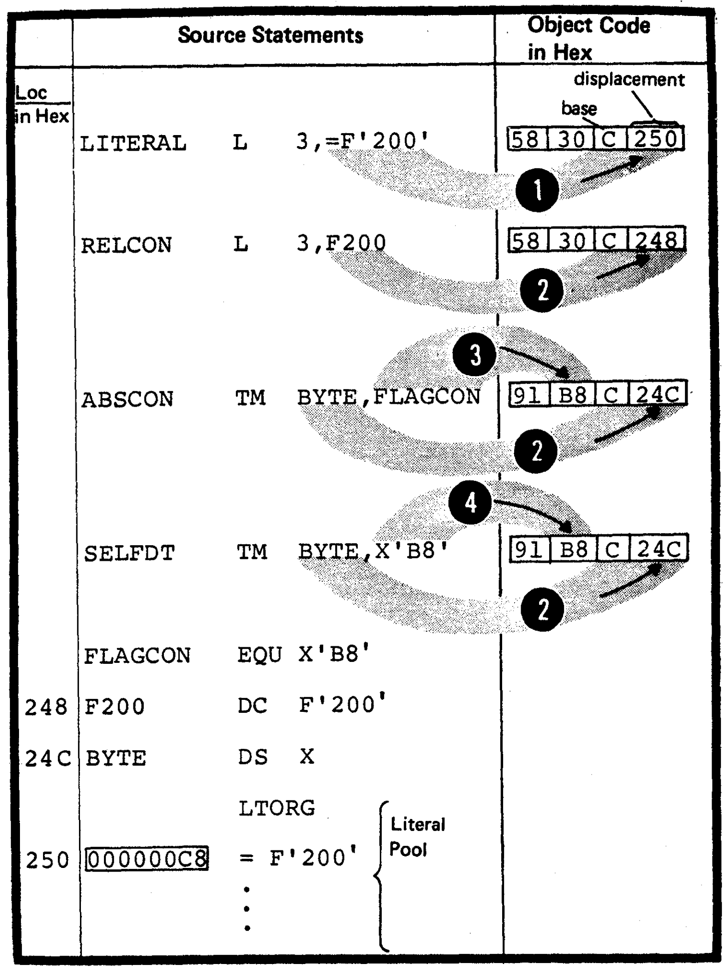
```
          LA  4,LOCORE  (4) }
          LA  4,1000    (5) } same effect
          .
          .
          .
LOCORE EQU 1000
```

The figure to the right illustrates the third point.

1 • The address of the literal, rather than the literal data itself is assembled into the object code.

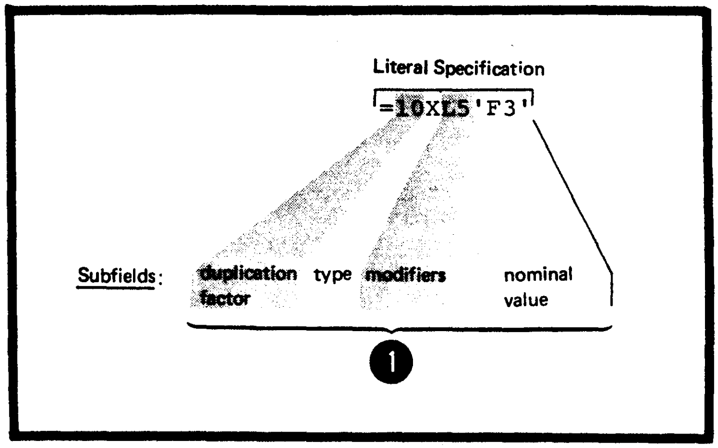
2 • The address of a constant is assembled into the object code. Note that when a symbol with an
3 absolute value represents immediate data, it is the absolute value that is assembled into the object code.

4 • The absolute value of a self-defining term is assembled into the object code.



Specifications

A literal must be coded as shown in the figure to the right.



1 The literal is specified in the same way as the operand of a DC instruction (for restrictions see G3C).

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:

1. In operands that represent the receiving field of an instruction that modifies storage

2. In any shift or I/O instructions.

C6 – Expressions

C6A -- PURPOSE

You can use an expression to specify:

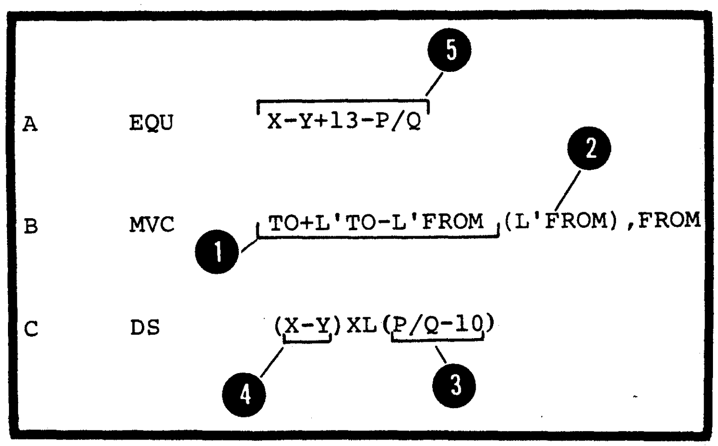
1 An address

2 An explicit length

3 A modifier

4 A duplication factor

5 A complete operand



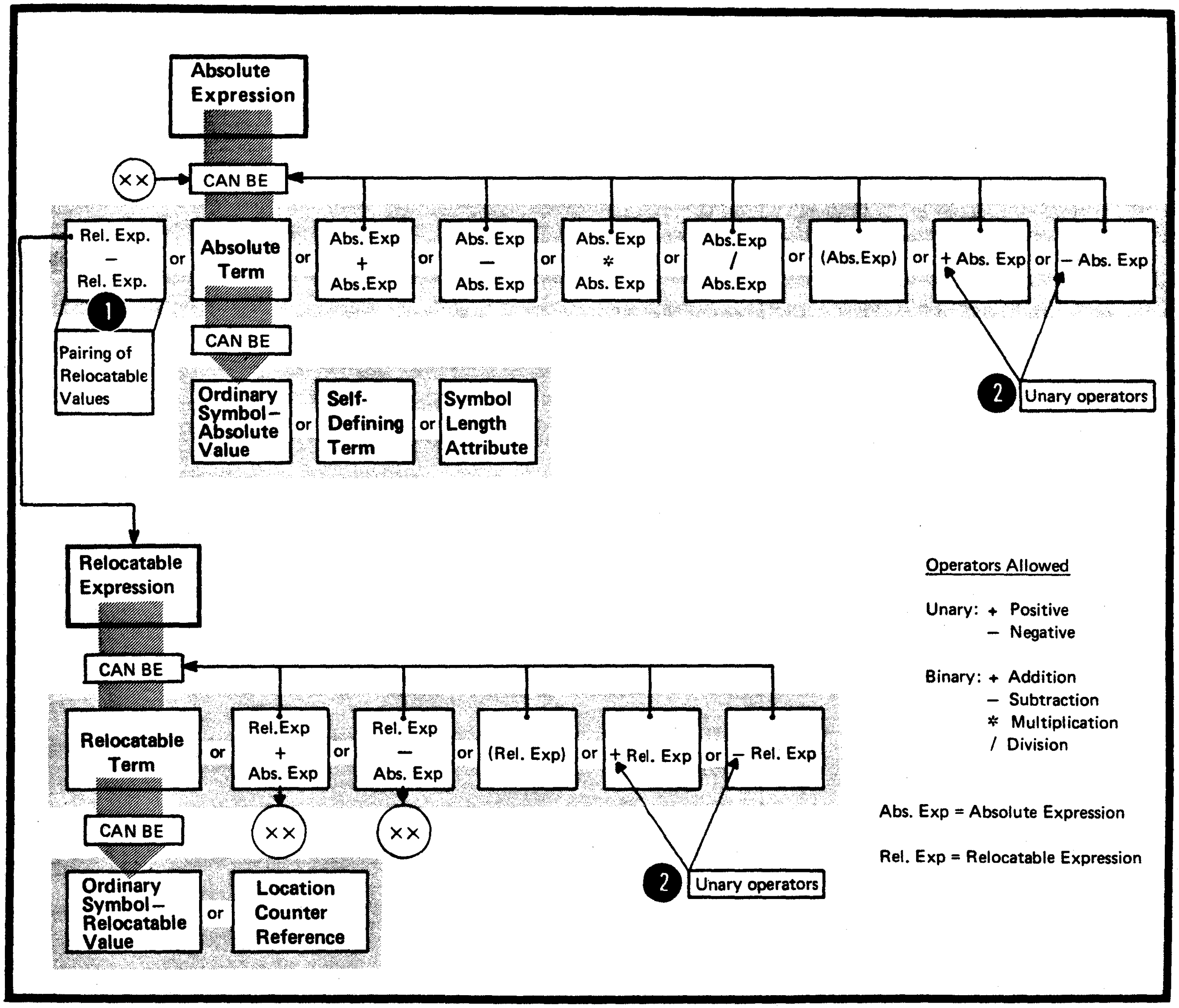
You can write an expression with a simple term or as an arithmetic combination of terms. The assembler reduces multiterm expressions to single values. Thus, you do not have to compute these values yourself.

Expressions have absolute or relocatable values. Whether an expression is absolute or relocatable depends on the value of the terms it contains. You can use the absolute or relocatable expression described in this subsection in a machine instruction or any assembler instruction other than a conditional assembly instruction. The assembler evaluates relocatable and absolute expressions at assembly time. Throughout this manual, the word "expression" refers to these types of expression.

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 pre-assembly time. In this manual they will always be referred to by their full names; they are described in detail in L4.

The figure below defines both absolute and relocatable expressions.

1 NOTE: The relocatable values that are paired must have the opposite sign after the resolution of all unary
2 operators.



Absolute and Relocatable Expressions

An expression is absolute if its value is not changed by program relocation; it is relocatable if its value is changed upon program relocation. A description of the factors that determine whether an expression is absolute or relocatable follows.

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:

- 1 The paired terms must be defined in the same control section of a source module (that is, have the same relocatability attribute).
- 2 The paired terms must have opposite signs after all unary operators are resolved. In an expression, the paired terms do 3 not have to be contiguous, that is, other terms can come between the paired terms.
- 4 The value represented by the paired terms is absolute.

Source Module

```
                      FIRST     CSECT
                                  .
Can be     1  {       A         DS        F
paired                B         DS        F
                      C         DS        F
                      LOCTREF   EQU       *
                      ABSA      EQU       X'F'
                      ABSB      EQU       300
                      ABSC      EQU       C'A'

                      SECOND    CSECT
                                  .
Can be     1  {       D         DS        X
paired                E         DS        X
                      F         DS        X
                                END
```

Examples:

Paired Relocatable Terms 4	Absolute Expressions
B-A C-A +B-+C ⟹ B-C (2) -A--B ⟹ -A+B LOCTREF-C D-E F-D	(3) A+ABSA-B D-E+ABSC F-D+B-C paired paired
Unpaired Relocatable Terms	**Relocatable Expressions**
B C LOCTREF D	Unpaired B+ABSA C+X'FF' F-5*(B-C) paired 4

Absolute Expressions

The assembler reduces an absolute expression to a single absolute value if the expression:

(1) 1. Is composed of a symbol with an absolute value, a self-defining term, or a symbol length attribute reference, or any arithmetic (2) combination of absolute terms.

(3) 2. If it contains relocatable terms, alone or in combination with absolute terms, and if all these relocatable (4) terms are paired.

Source Module

```
FIRST   CSECT
          .
          .
A       DC      F'2'
B       DC      F'3'
C       DC      F'4'
ABSA    EQU     100
ABSB    EQU     X'FF'
ABSC    EQU     B-A
          .     Paired
          .
ABSD    EQU     *-A
        END
```

Absolute Expressions

(1)
```
ABSA
15
L'A
```

(2)
```
ABSA+ABSC-ABSC*15
```

(3) (4 — paired terms)
```
B-A
ABSA+15-B+C-ABSD/(C-A+ABSA)
```

Relocatable Expressions

A relocatable expression is one whose value changes, for example, by a 1000, if the object module into which it is assembled is relocated 1000 bytes away from its originally assigned storage area. 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:

   (1) 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
   (2) 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.

COMPLEX RELOCATABLE EXPRESSIONS: Complex relocatable expressions, unlike relocatable expressions, can contain:

   a. Two or more unpaired relocatable terms or

   b. An unpaired relocatable term preceded by a negative sign.

Complex relocatable expressions can be used only in A-type and Y-type address constants (see G3J).

Source Module

```
FIRST   CSECT
          .
          .
A       DC        H'2'
B       DC        H'3'
C       DC        H'4'
          .
          .
          .
ABSA    EQU       10
ABSB    EQU       *-A
ABSC    EQU       10*(B-A)
          .
          .
        END
```

Relocatable Expresssions:

(Belong to control section named FIRST and have same relocatable attribute as A, B and C)

(2) A

A+ABSA+10

B-+A+C-10*ABSC

(1) (2)

B-A+C+100*ABSA+ABSA/(C-A)

(1)

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.

❶ 2. An expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression.

❷ 3. An expression must not begin with a binary operator, nor can ❸ it contain two binary operators in succession.

❹ 4. An expression must not contain two terms in succession.

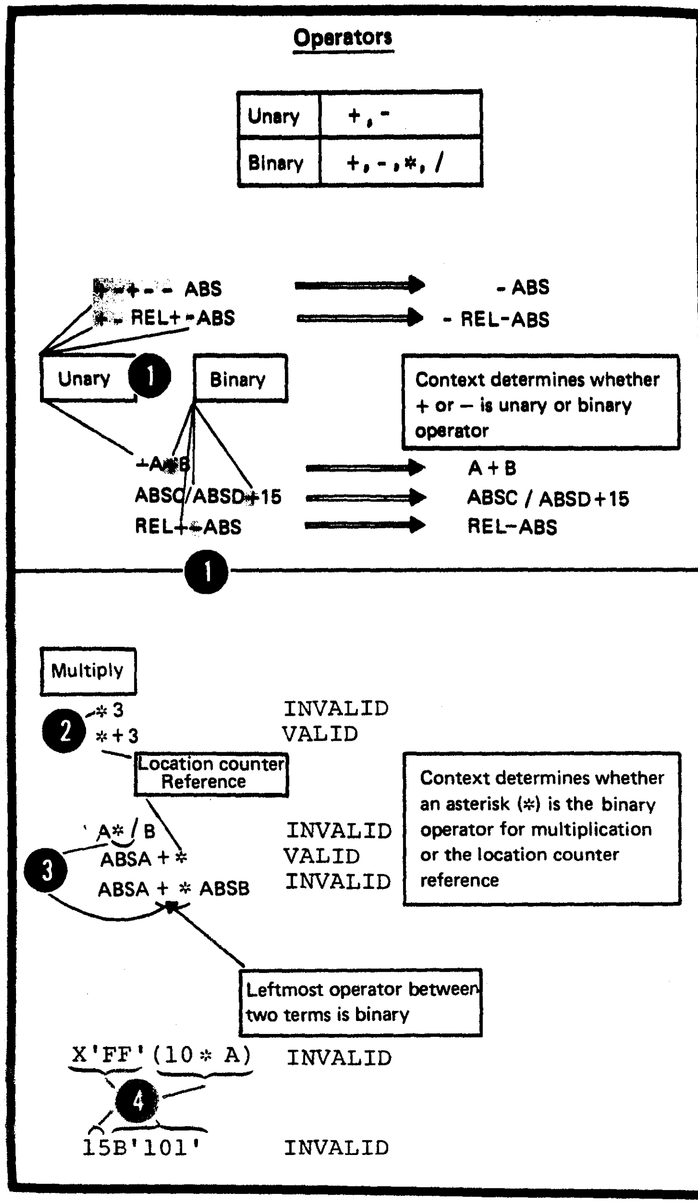
5. No blanks are allowed between an operator and a term nor between two successive operators.

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.

DOS 6. An expression can contain up to 15 unary and binary operators and up to 5 levels of parentheses.

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.



Evaluation of Expressions

The assembler reduces a multiterm expression to a single value as follows:

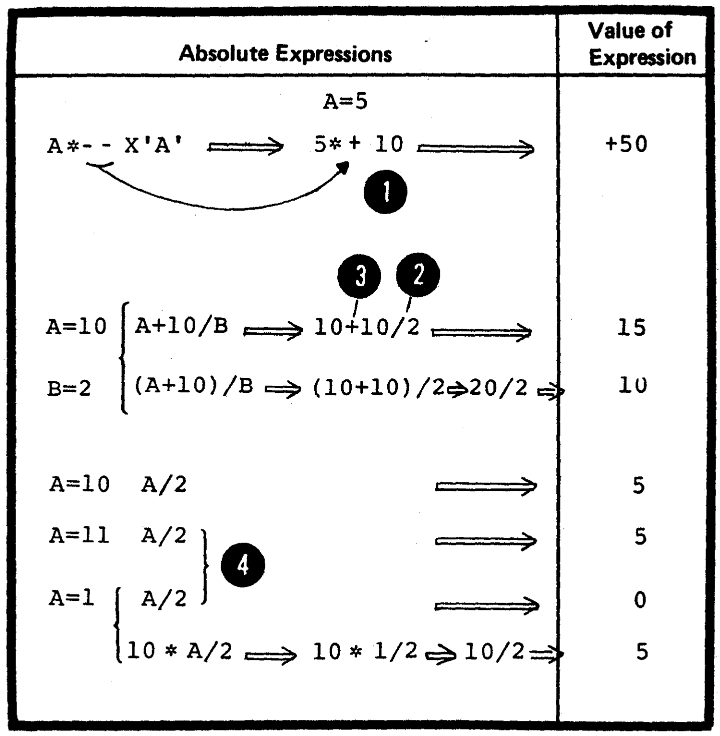
1. It evaluates each term.

2. It performs arithmetic operations from left tc right. Hcwever:

   (1) a. It performs unary operations before binary cperations, and

   (2) b. It performs the binary operations of multiplication and division befcre the binary (3) operations of addition and subtraction.

(4) 3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives 0.

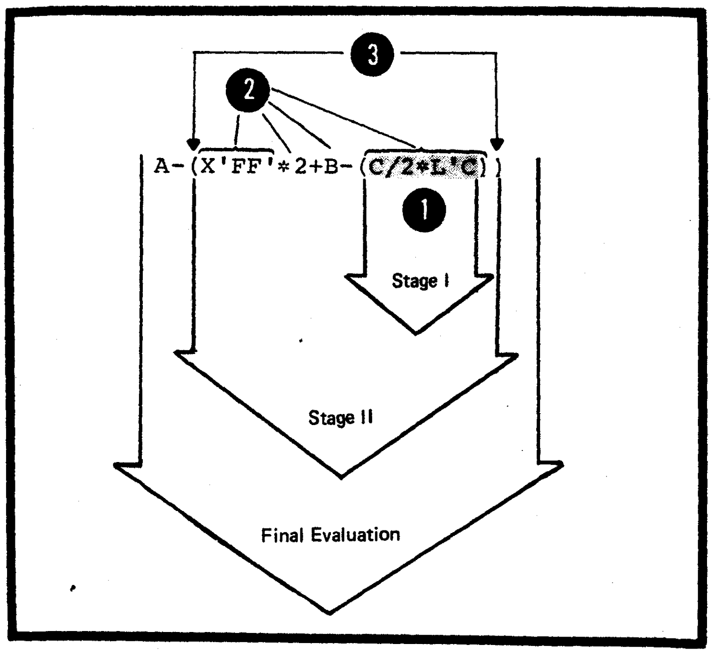


4. In parenthesized expressions, (1) the assembler evaluates the inner most expressions first and then considers them as terms in the next (2) outer level of expressions. It (3) continues this process until the outermost expression is evaluated.

5. A term or expression's intermediate value and computed result must lie in the range of -2^{31} through $+2^{31}-1$.

DOS 6. The computed result is then truncated to a 24-bit value that lies between 0 and 16,777,215.

NOTE: It is assured that the assembler evaluates paired relocatable terms at each level of expression nesting.



Part II: Functions and Coding of Machine Instructions

SECTION D: MACHINE INSTRUCTIONS

Section D: Machine Instructions

This section 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 (see GLOSSARY) areas see the Principles of Operation manuals:

- IBM System/360 Principles of Operation, Order No. GA22-6821

- IBM System/370 Principles of Operation, Order No. GA22-7000

D1 - Functions

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 G3). At execution time, the machine instructions can refer to these constants and areas, but the constants themselves are not executed.

Purpose

You use fixed-point instructions when you wish to perform arithmetic operations on data represented in binary form. These instructions treat all numbers as integers. If they are to operate upon data representing mixed numbers (such as 3.14 and 0.235) you must keep track of the decimal point yourself. For your constants you must provide the necessary number of binary positions to represent the fractional portion of the number specified by using the scale modifier (see G3B).

Operations Performed

Fixed-point instructions allow you to perform the operations listed in the figure to the right.

Data Constants Used

In fixed-point instructions, you can refer to the constants listed in the figure to the right.

❶ NOTE: Except for the conversion operations, fixed-point arithmetic is performed on signed binary values.

Fixed - Point Operations	Mnemonic Operation Codes
Add	AR , A, AH, ALR, AL
Subtract	SR, S, SH, SLR, SL
Multiply	MR, M, MH
Divide	DR, D
Arithmetic Compare (taking sign into account)	CR, C, CH
Load into registers	LR, L, LH, LTR, LCR, LPR, LNR, LM
Store into areas	ST, STH, STM
Arithmetic Shift of binary contents of registers to left or right (retaining sign)	SLA, SRA, SLDA, SRDA
Convert (packed) decimal data to binary ❶	CVB
Convert binary data to (packed) decimal data ❶	CVD

Constants Used	Type
Fixed-Point	H and F
Binary	B
Hexadecimal	X
Character	C
Decimal (packed)	P
Address	Y, A, S, V and Q

D1B -- DECIMAL ARITHMETIC

Purpose

You use the decimal instructions when you wish to perform arithmetic operations on data that has the binary equivalent of decimal representation, either in packed or zoned form. These instructions treat all numbers as integers. For example, 3.14, 31.4, and 314 are all processed as 314. You must keep track of the decimal point yourself.

Operations Performed

Decimal instructions allow you to perform the operations listed in the figure to the right.

Data Constants Used

In decimal instructions you can refer to the constants listed in the figure to the right.

(1) NOTE: Except for the conversion operations, decimal arithmetic is performed on signed packed decimal values.

Decimal Operations	Mnemonic Operation Codes
Add	AP
Subtract	SP
Multiply	MP
Divide	DP
Arithmetic Compare (taking sign into account)	CP
Move decimal data with a 4-bit offset	MVO
Shift decimal data in fields to left or right	SRP
Set a field to zero and add contents of another field	ZAP
Convert zoned to packed decimal data (1)	PACK
Convert packed to zoned decimal data (1)	UNPK

Constants Used	Type
Decimal (packed)	P
(zoned)	Z

D1C -- FLOATING-POINT ARITHMETIC

Purpose

You use floating-point instructions when you wish to perform arithmetic operations on binary data that represents both integers and fractions. Thus, you do not have to keep track of the decimal point in your computations. Floating-point instructions also allow you to perform arithmetic operations on both very large numbers and very small numbers, with greater precision than with fixed-point instructions.

Operations Performed

Floating-point instructions allow you to perform the operations listed in the figure to the right.

Data Constants Used

In floating-point instructions, you can refer to the constants listed in the figure to the right.

NOTE: Floating-point arithmetic is performed on signed values that must have a special floating-point format. The fractional portion of floating-point numbers, when
❶ used in addition and subtraction, can have a normalized (no leading zeros) or unnormalized format.

Floating - Point Operations	**Mnemonic Operation Codes**
Add ❶	ADR, AD, AER, AE, AWR AW, AUR, AU, AXR
Subtract ❶	SDR, SD, SER, SE, SWR, SW, SUR, SU, SXR
Multiply	MDR, MD, MER, ME, MXR, MXDR,MXD
Divide	DDR, DD, DER, DE
Halve (division by 2)	HDR, HER
Arithmetic Compare (taking sign into account)	CDR, CD, CER, CE
Load into floating - point registers	LDR, LD, LER, LE, LTDR, LTER, LCDR, LCER, LPDR, LPER,LNDR,LDER,LRDR, LRER
Store into areas	STD, STE

Constants Used	**Type**
Floating - Point	E, D, and L

D1D -- LOGICAL OPERATIONS

Purpose

You can use the logical instructions to introduce data, move data, or inspect and change data.

Operations Performed

The logical instructions allow you to perform the operations listed in the figure to the right.

Logical Operations	Mnemonic Operation Codes
Move	MVI, MVC, MVN, MVZ, MVCL
Logical Compare (unsigned binary values)	CLR, CL, CLI, CLC, CLCL, CLM
AND (logical multiplication)	NR, N, NI, NC
OR (logical addition)	OR, O, OI, OC
Exclusive OR (either........ or, but not both)	XR, X, XI, XC
Testing binary bit patterns	TM
Inserting characters into registers	IC, ICM
Store characters into areas	STC, STCM
Load address into register	LA
Logical Shift of unsigned binary contents of registers to left or right	SLL, SRL, SLDL, SRDL
Replace argument values by corresponding function values from table (translate)	TR, TRT
Edit (packed and zoned decimal data) values in preparation for printing	ED, EDMK

D1E -- BRANCHING

Purpose

You can use several types of branching instructions, combined with the logical instructions listed in D1D, to code and control loops, subroutine linkages, and the sequence of processing.

Operations Performed

The branching instructions allow you to perform the operations listed in the figure to the right.

❶ NOTE: Additional mnemonics for branching on condition are described in section D1H below.

Branching Operations	Mnemonic Operation Codes
Branch depending on the results of the preceding operation (that sets the condition code) ❶	BCR, BC
Branch to a subroutine with a return link to current code	BALR, BAL
Branch according to a count contained in a register (count is decremented by one before determining course of action)	BCTR, BCT
Branch by comparing index value to fixed comparand, (index incremented or decremented before determining course of action)	BXH, BXLE
Temporary Branch in order to execute a specific machine instruction	EX

D1F -- STATUS SWITCHING

Purpose

You can use the status switching instructions to communicate between your program and the system control program. However, some of these instructions are privileged instructions and you can use them only when the CPU is in the supervisor state, but not when it is in the problem state. The privileged instructions are marked with a "p" in the figure to the right.

Operations Performed

The status switching instructions allow you to perform the operations listed in the figure to the right.

Status Switching Operations		Mnemonic Operation Codes
Load program status information	P	LPSW
Load sequence of control registers	P	LCTL
Set bit patterns for condition code and interrupts for program		SPM
Set bit patterns for channel usage by system	P	SSM
Set protection key for a block of storage	P	SSK
Set time-of-day clock	P	SCK
Insert protection key for storage into a register	P	ISK
Store time-of-day clock		STCK
Store identification of channel or CPU	P	STIDC, STIDP
Store (save) sequence of control registers	P	STCTL
Call supervisor for system interrupt		SVC
Call monitor for interrupts depending on contents of control register		MC
Test bit which is subsequently set to 1		TS
Write or Read directly to or from other CPU's	P	WRD, RDD
Set Clock Comparator	P	SCKC
Store Clock Comparator	P	STCKC
Set CPU Timer	P	SPT
Store CPU Timer	P	STPT
Store Then AND System Mask	P	STNSM
Store Then OR System Mask	P	STOSM

D1G -- INPUT/OUTPUT

Purpose

You can use the input/output instructions, instead of the IBM-supplied system macro instructions, when you wish to control your input and output operations more closely.

Operations Performed

The input or output instructions allow you to identify the channel, or the device on which the input or output operation is to be performed. The operations performed are listed in the figure to the right. However, these are privileged instructions, and you can only use them when the CPU is in the supervisor state, but not when it is in the problem state.

Input or Output Operations	Mnemonic Operation Codes
Start I/O	SIO, SIOF
Halt I/O	HIO
Test state of channel or device being used	TIO,TCH
Halt Device	HDV

D1H -- BRANCHING WITH EXTENDED MNEMONIC CODES

Purpose

The branching instructions described below allow you to specify a mnemonic code for the condition on which a branch is to occur. Thus, you avoid having to specify the mask value required by the BC and BCR branching instructions. The assembler translates the mnemonic code that represents the condition into the mask value, which is then assembled in the object code of the machine instruction.

Specifications

The extended mnemonic codes are given in the figure on the opposite page.

They can be used as operation codes for branching
1 instructions, replacing the BC and BCR machine instruction
codes. Note that the first operand of the BC and BCR 2
3 instructions must not be present in the operand field of
the extended mnemonic branching instructions.

4 NOTE: The addresses represented are explicit addresses; however, implicit addresses can also be used in this type of instruction.

Extended Code		Meaning	Format	(Symbolic) Machine Instruction Equivalent	
	❸ ❹			❶	❷
B	D2(X2,B2)	Unconditional Branch	RX	BC	15,D2(X2,B2)
BR	R2		RR	BCR	15,R2
NOP	D2(X2,B2)	No Operation	RX	BC	0,D2(X2,B2)
NOPR	R2		RR	BCR	0,R2

Used After Compare Instructions

BH	D2(X2,B2)	Branch on High	RX	BC	2,D2(X2,B2)
BHR	R2		RR	BCR	2,R2
BL	D2(X2,B2)	Branch on Low	RX	BC	4,D2(X2,B2)
BLR	R2		RR	BCR	4,R2
BE	D2(X2,B2)	Branch on Equal	RX	BC	8,D2(X2,B2)
BER	R2		RR	BCR	8,R2
BNH	D2(X2,B2)	Branch on Not High	RX	BC	13,D2(X2,B2)
BNHR	R2		RR	BCR	13,R2
BNL	D2(X2,B2)	Branch on Not Low	RX	BC	11,D2(X2,B2)
BNLR	R2		RR	BCR	11,R2
BNE	D2(X2,B2)	Branch on Not Equal	RX	BC	7,D2(X2,B2)
BNER	R2		RR	BCR	7,R2

Used After Arithmetic Instructions

BO	D2(X2,B2)	Branch on Overflow	RX	BC	1,D2(X2,B2)
BOR	R2		RR	BCR	1,R2
BP	D2(X2,B2)	Branch on Plus	RX	BC	2,D2(X2,B2)
BPR	R2		RR	BCR	2,R2
BM	D2(X2,B2)	Branch on Minus	RX	BC	4,D2(X2,B2)
BMR	R2		RR	BCR	4,R2
BNP	D2(X2,B2)	Branch on Not Plus	RX	BC	13,D2(X2,B2)
BNPR	R2		RR	BCR	13,R2
BNM	D2(X2,B2)	Branch on Not Minus	RX	BC	11,D2(X2,B2)
BNMR	R2		RR	BCR	11,R2
BNZ	D2(X2,B2)	Branch on Not Zero	RX	BC	7,D2(X2,B2)
BNZR	R2		RR	BCR	7,R2
BZ	D2(X2,B2)	Branch on Zero	RX	BC	8,D2(X2,B2)
BZR	R2		RR	BCR	8,R2
BNO	D2(X2,B2)	Branch on No Overflow	RX	BC	14,D2(X2,B2)
BNOR	R2		RR	BCR	14,R2

Used After Test Under Mask Instructions

BO	D2(X2,B2)	Branch if Ones	RX	BC	1,D2(X2,B2)
BOR	R2		RR	BCR	1,R2
BM	D2(X2,B2)	Branch if Mixed	RX	BC	4,D2(X2,B2)
BMR	R2		RR	BCR	4,R2
BZ	D2(X2,B2)	Branch if Zeros	RX	BC	8,D2(X2,B2)
BZR	R2		RR	BCR	8,R2
BNO	D2(X2,B2)	Branch if Not Ones	RX	BC	14,D2(X2,B2)
BNOR	R2		RR	BCR	14,R2
BNM	D2(X2,B2)	Branch if Not Mixed	RX	BC	11,D2(X2,B2)
BNMR	R2		RR	BCR	11,R2
BNZ	D2(X2,B2)	Branch if Not Zeros	RX	BC	7,D2(X2,B2)
BNZR	R2		RR	BCR	7,R2

D2=displacement,X2=index register,B2=base register,R2=register containing branch address

D1I -- RELOCATION HANDLING

Purpose

You use the relocation instructions in connnection with the relocate feature of IBM System/370.

Operations Performed

The relocation instructions allow you to perform the operations listed in the figure to the right. However, these instructions are privileged instructions, and you can use them only when the CPU is in the supervisor state, but not when it is in the problem state.

Relocation Operations	Mnemonic Operation Code
Load Real Address	LRA
Purge Translation Lookaside Buffer	PTLB
Reset Reference Bit	RRB
Set Clock Comparator	SCKC
Store Clock Comparator	STCKC
Set CPU Timer	SPT
Store CPU Timer	STPT
Store and AND System Mask	STNSM
Store and OR System Mask	STOSM

D2 - Alignment

Purpose

The assembler automatically aligns the object code of all machine instructions on halfword boundaries. For execution of the IBM System/370 machines, the constants and areas do not have to lie on specific boundaries to be addressed by the machine instructions.

However, if the assembler option ALIGN is set, you can cause the assembler to align constants and areas; for example, on fullword boundaries. This allows faster execution of the fullword machine instructions.

If the NOALIGN option is set, you do not need to align constants and areas. They will be assembled at the next available byte, which allows you to save space (no bytes are skipped for alignment).

Specifications

MACHINE INSTRUCTIONS: When the assembler aligns machine instructions on halfword boundaries, it sets
1 any bytes skipped to zero.

CONSTANTS AND AREAS: One of the assembler options that can be set in the job control language (that initiates execution of the assembler program) concerns the alignment of constants and areas; it can be specified as ALIGN or NOALIGN.

If ALIGN is specified, the following applies:

2 • The assembler aligns constants and areas on the boundaries implicit in their type, if no length specification is supplied.

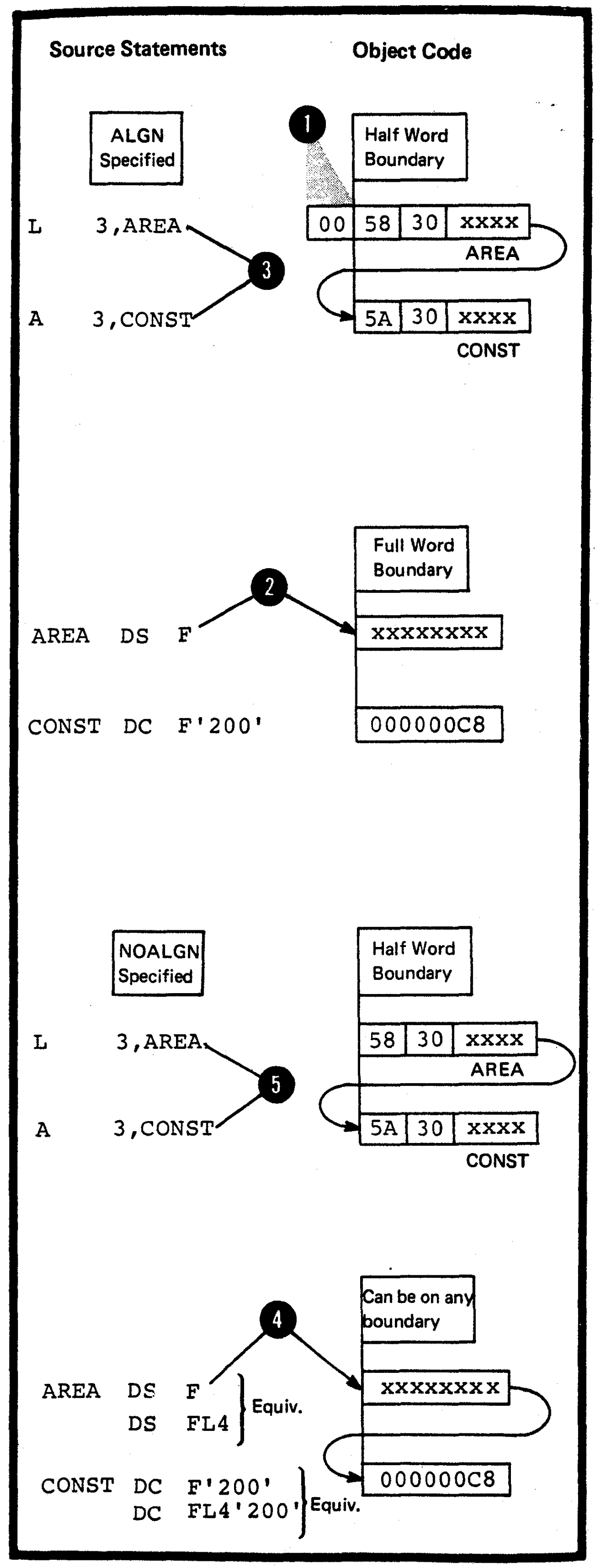
3 • The assembler checks all expressions that represent storage addresses to ensure that they are aligned on the boundaries required by the instructions. If they are not, the assembler issues a warning message.

If NOALIGN is specified, the following applies:

4 • The assembler does not align constants and areas on special boundaries, even if the length specification is omitted. Note that the CCW instruction, however, always causes the alignment of the channel command word on a doubleword boundary.

5 • The assembler does not check storage addresses for boundary alignment.

NOTE 1: The assembler always forces alignment if a duplication factor of 0 is specified in a constant or area without a length modifier (for an example, see G3N). Alignment occurs when either ALIGN or NOALIGN is set.



NOTE 2: When NOALIGN is specified, the CNOP assembler instruction can be used to ensure the correct alignment of data referred to by the privileged instructions that require specific boundary alignment. The mnemonic operation codes for these instructions are listed in the figure to the right.

Mnemonic Operation Codes for Privileged Operations	Meaning
LPSW	Load program status word.
ISK	Insert Storage Key.
SSK	Set Storage Key.
LCTL	Load Control registers.
SCK	Set Clock.
STIDP	Store CPU Identification
STCTL	Store Control registers.
(Diagnose - not handled by assembler)	

D3 -- Statement Formats

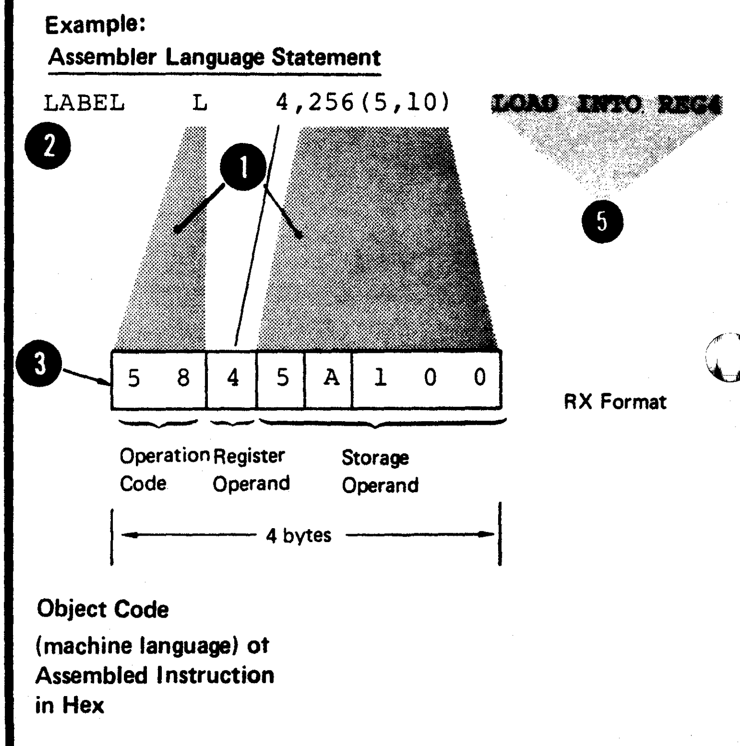
Machine instructions are assembled into object code according to one of the six formats given in the figure to the right.

When you code machine instructions you use symbolic formats that correspond to the actual machine language formats. Within each basic format, you can also code variations of the symbolic representation (Examples of coded machine instructions, divided into groups according to the six basic formats, are illustrated in D6 below).

The assembler converts only the **1** operation code and the operand entries of the assembler language statement into object code. The assembler assigns to the symbol **2** you code as a name entry the value of the address of the leftmost **3** byte of the assembled instruction. When you use this same symbol in the operand of an assembler language statement, the assembler uses this address value in converting the symbolic operand into its object **4** code form. The length attribute assigned to the symbol depends on the basic machine language format of the instruction in which the symbol appears as a name entry (for details on the length attribute see C4C).

5 A remarks entry is not converted into object code.

Format	Length of Object Code Required for the Assembled Instruction in Bytes
RR	2
RX	4 **4** (L'LABEL=4)
RS	4
SI	4
S	4
SS	6



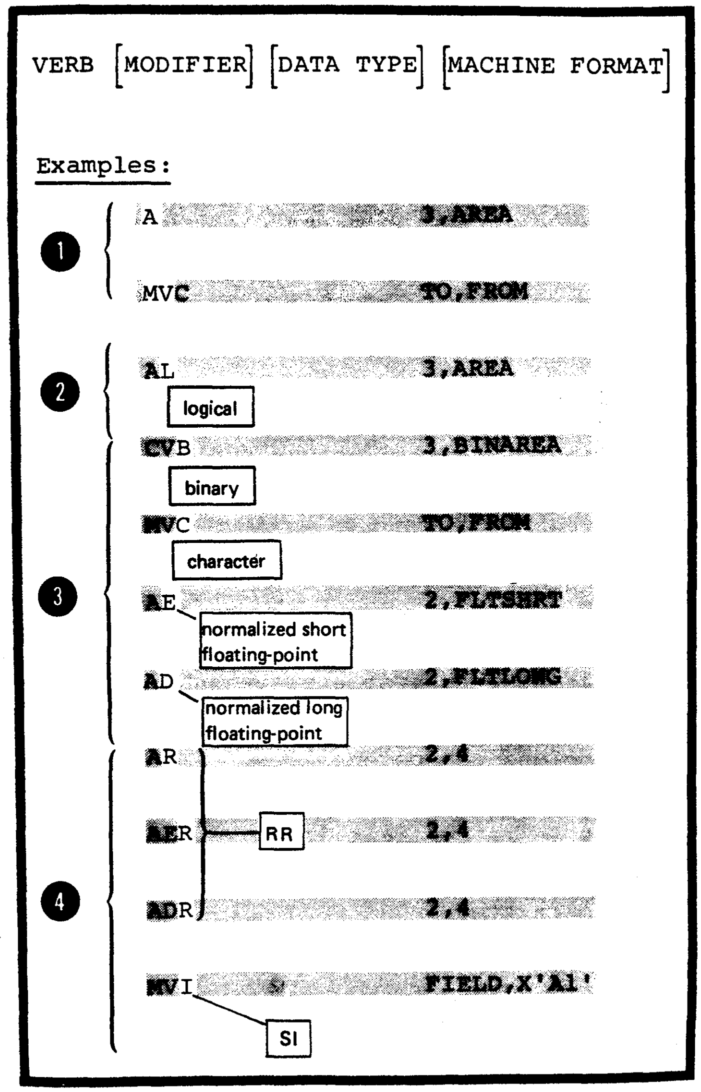
D4 – Mnemonic Operation Codes

Purpose

You must specify an operation code for each machine instruction statement. The mnemonic operation code indicates the type of operation to be performed; for example, "A" indicates the "addition" operation. Appendix IV contains a complete list of mnemonic operation codes and the formats of the corresponding machine instructions.

Specifications

The general format of the machine instruction operation code is shown in the figure to the right.



1 The verb must always be present. It usually consists of one or two characters and specifies the operation to be performed. The other items in the operation code are not always present. They include:

2 • The modifier which further defines the operation

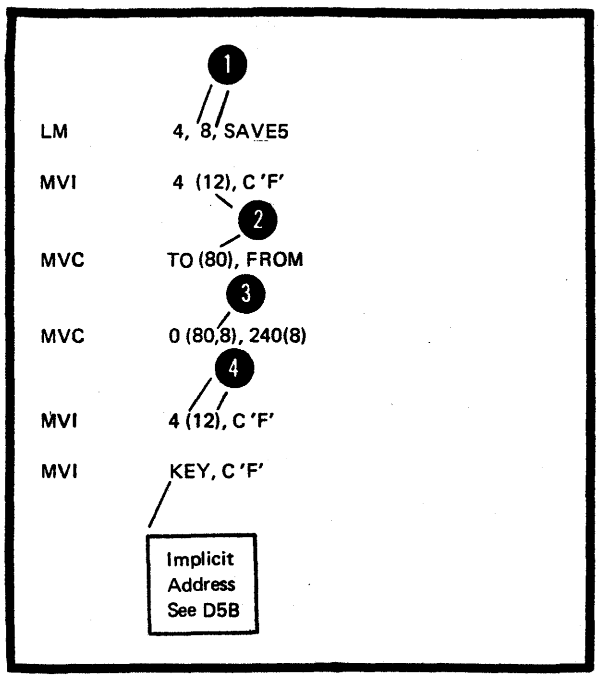
3 • The type qualifier, which indicates the type of data used by the instruction in its operation, and

4 • The format qualifier, R or I, which indicates that an RR or SI machine instruction format is assembled.

Furpose

You rust specify cne cr mcre operands in each machine instructicn staterent to prcvide the data cr the location of the data upon which the machine operaticn is tc be perfcrred. The operand entries consist of cne cr rore fields cr subfields depending on the format of the instruction teing ccded. They can specify a register, an address, a length, and inrediate data.

You can code an operand entry either with syrtcls cr with self-defining terms. You can omit length fields or sutfields, which the assembler will compute for you from the cther cperand entries.



General Specifications for Coding Operand Entries

The rules for coding cperand entries are as fcllcws:

1 A comma must separate operands.

2 Farentheses rust enclcse subfields.

3 A corra rust separate subfields enclosed in parentheses.

If a sutfield is critted because it is in implicit in a syrtclic address, the parentheses that would 4 have enclosed the sutfield rust te critted.

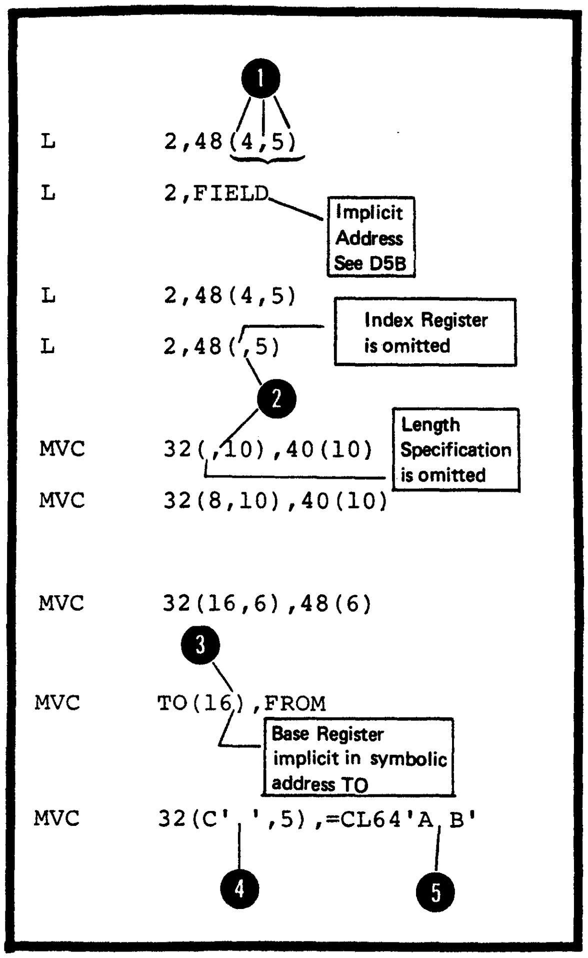
If two subfields are enclosed in parentheses and separated by commas, the following applies:

If both subfields are omitted because they are implicit in a symbolic entry, the separating comma and the parentheses that would have been needed must also be omitted. (1)

If the first subfield is omitted, the comma that separates it from the second subfield must be written as well as the enclosing parentheses. (2)

If the second subfield is omitted, the comma that separates it from the first subfield must be omitted, however, the enclosing parentheses must be written. (3)

NOTE: Blanks must not appear within the operand field, except as part of a character self-defining term (4) or in the specification of a character literal (5).



Purpose and Usage

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

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

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

Specifications

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

1. The floating-point registers (0, 2, 4, or 6) must be specified
❶ for floating-point instructions:

2. The even numbered registers (0, 2, 4, 6, 8, 10, 12, 14) must be specified for the following groups of instructions:

❷ a. The double-shift instructions

❸ b. The fullword multiply and divide instructions

❹ c. The move long and compare logical long instructions.

3. The floating-point registers 0 and 4 must be specified for the
❺ instructions that use extended floating-point data:
AXR, SXR, LRDR, MXR, MXDR, MXD.

NOTE: The assembler checks the registers specified in the instruction statements of the above groups. If the specified register does not comply with the stated restrictions, the assembler issues a diagnostic message and does not assemble the instruction.

Registers

		Operation Code	Register Operand
Examples:		L	3,AREA
	❶	LE	4,FLTAREA
	❷	SLDA	4,1
	❷	SRDA	6,2
	❷	SLDL	8,3
	❷	SRDL	12,3
	❸	M	6,MULTIP
	❸	D	8,DIVIDER
	❹	MVCL	6,8
	❹	CLCL	2,4
	❺	AXR	0,4

Both register operands must be even-numbered (MVCL, CLCL)

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

1. The base registers that are implicit in the symbolic addresses specified. These implicit addresses are described in detail in D5B. The registers can be identified (1) by examining the object code of the assembled machine instruction (2) or the USING instruction(s) that assigns base registers for the source module.

2. The registers that are used by machine instructions in their operations, but do not appear even in the assembled object code. They are as follows:

(3) a. For the double shift and fullword multiply and divide instructions, the odd-numbered register whose number is one greater than the even-numbered register specified as the first operand.

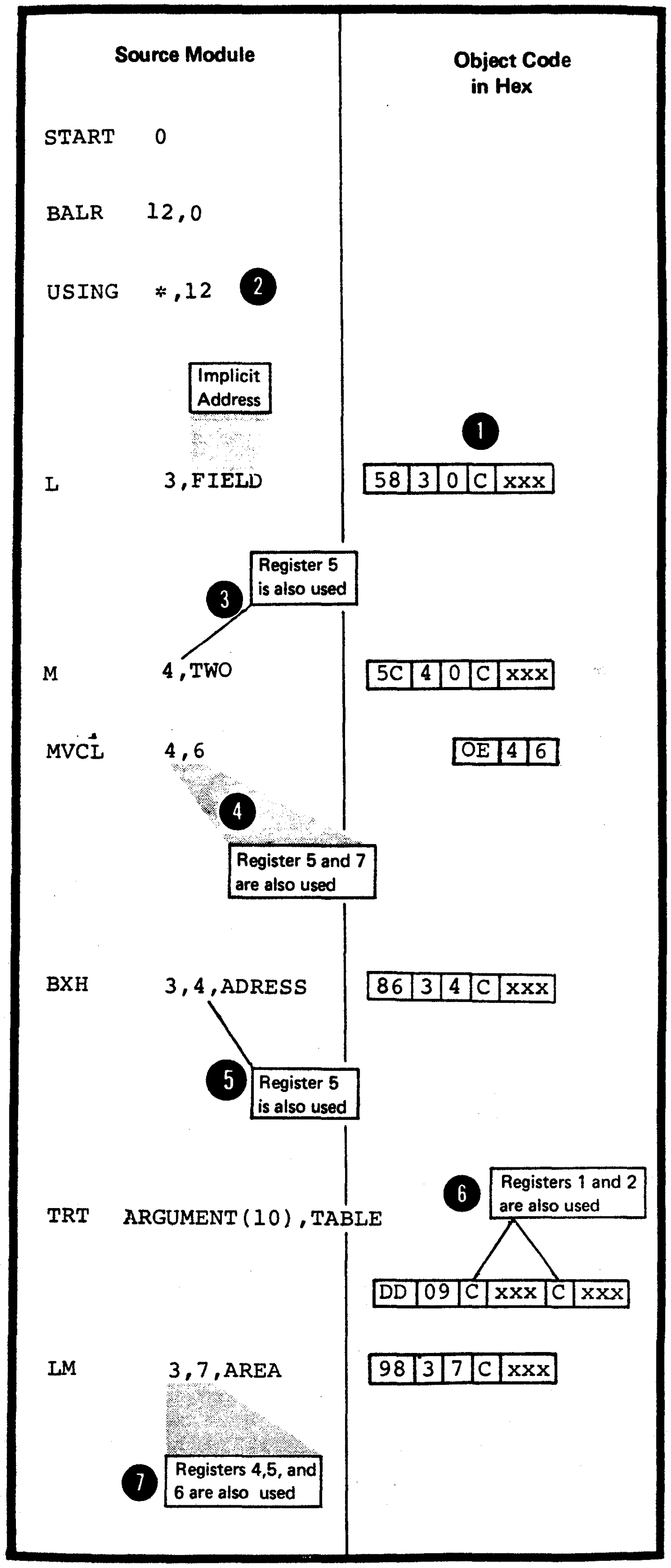
(4) b. For the Move Long and Compare Logical Long instructions, the odd-numbered registers whose number is one greater than the even numbered registers specified in the two operands.

(5) c. For the Branch on Index High (BXH) and the Branch on Index Low or Equal (BXLE) instructions; if the register specified for the second operand is an even-numbered register, the next higher odd-numbered register is used to contain the value to be used for comparison.

(6) d. For the Translate and Test (TRT) instruction, registers 1 and 2 are also used.

(7) e. For the Load Multiple (LM) and Store Multiple (STM) instructions, the registers that lie between the registers specified in the first two operands.

REGISTER USAGE BY SYSTEM: The control program of the IBM System/370 uses registers 0, 1, 13, 14, and 15.



Purpose and Definition

You can code a symbol in the name field of a machine instruction statement to represent the address of that instruction. You can then refer to the symbol in the operands of other machine instruction statements. The object code for the IBM System/370 requires that all addresses be assembled in a numeric base-displacement format. This format allows you to specify addresses that are relocatable or absolute.

You must not confuse the concept of relocatability with the actual addresses that are coded as relocatable, nor with the format of the addresses that are assembled.

DEFINING SYMBOLIC ADDRESSES: You define symbols to represent either relocatable or absolute addresses. You can define relocatable addresses in two ways:

1 By using a symbol as the label in the name field of an assembler language statement or

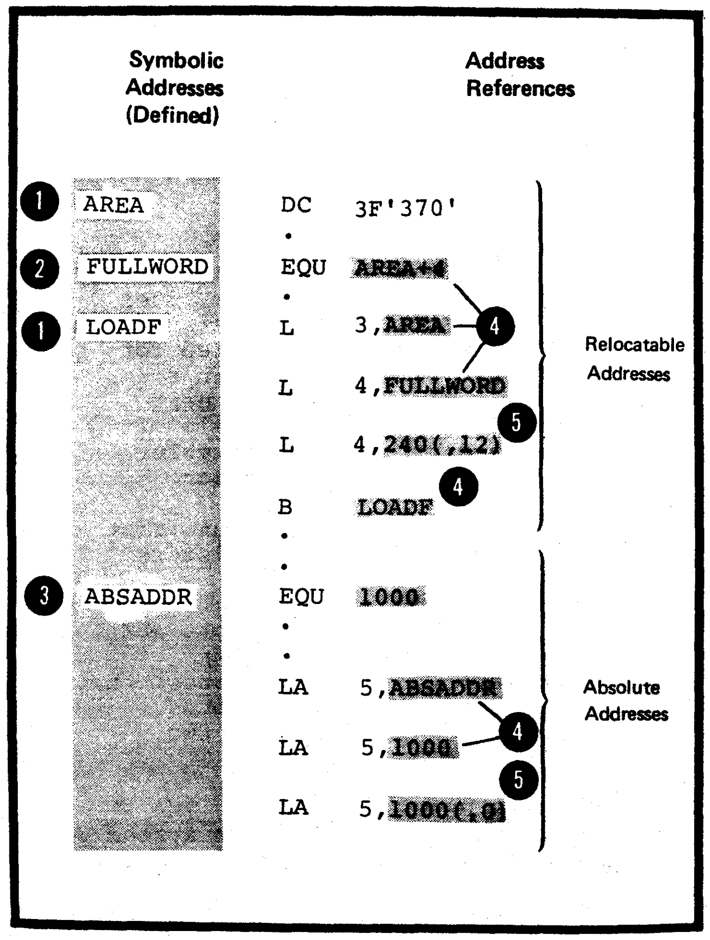
2 By equating a symbol to a relocatable expression.

3 You can define absolute addresses (or values) by equating a symbol to an absolute expression.

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

4 Implicitly: that is, in a form that the assembler must first convert into an explicit base-displacement form before it can be assembled into object code.

5 Explicitly: that is, in a form that can be directly assembled into object code.



Relocatability of Addresses

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

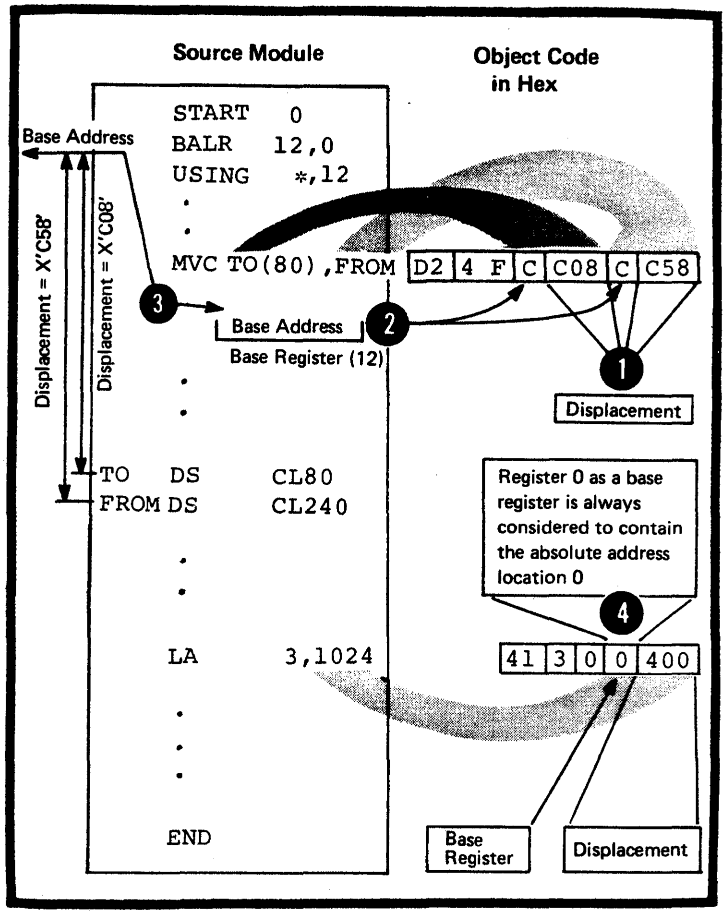
- 1 2 Each relocatable address is assembled as a displacement from a base address and a base register.

- 3 The base register contains the base address.

- If the object module assembled from your source module is relocated, only the contents of the base register need reflect this relocation. This means that the location in virtual storage of your base has changed and that your base register must contain this new base address.

- Your addresses have been assembled as relative to the base address; therefore, the sum of the displacement and the contents of the base register will point to the correct address after relocation.

NOTE: Absolute addresses are also assembled in the base-displacement form, but always indicate a fixed location in virtual storage. This 4 means that the contents of the base register must always be a fixed absolute address value regardless of relocation.



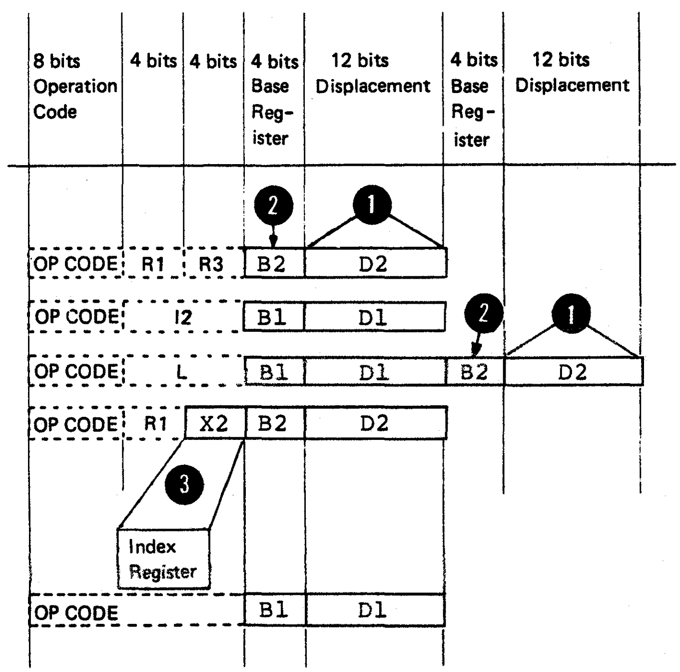
Page of GC33-4010-4
Revised July 31, 1976
By TNL: GN33-8207

Specifications

MACHINE OR OBJECT CODE FORMAT: All addresses assembled into the object code of the IBM System/370 machine instructions have the format given in the figure below.

Format	Coded or Symbolic Representation of Explicit Addresses
RS	D2(B2)
SI	D1(B1)
SS	D1(,B1),D2(B2)
RX	D2(X2,B2)
S	D1(B1)

Object Code Representation of Addresses



R1 and R3 represent registers
I2 represents an immediate value
L represents a length value

The addresses represented have a value which is the sum of:

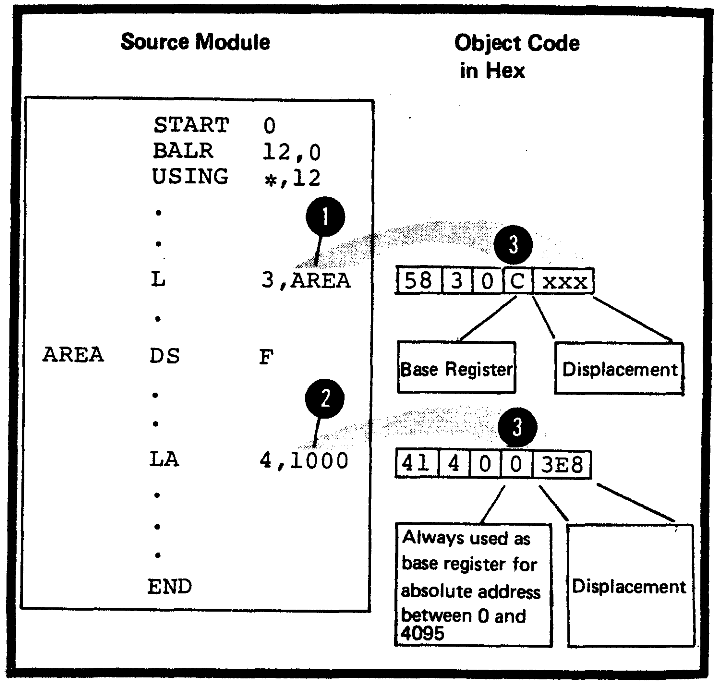
1 • A displacement and

2 • The contents of a base register.

NOTE: In RX instructions, the address represented has a value which is the sum of a displacement, the contents of a base register, and the contents of an index register. (3)

Implicit Address

An implicit address is specified by coding one expression. The expression can be relocatable (1) or absolute (2). The assembler converts all implicit addresses into their base-displacement form (3) before it assembles them into object code. The assembler converts implicit addresses into explicit addresses only if a USING instruction has been specified. The USING instruction assigns both a base address, from which the assembler computes displacements, and a base register, to contain the base address. The base register must be loaded with the correct base address at execution time. For details on how the USING instruction is used when establishing addressability, thus allowing implicit references, see F1.



Explicit Address

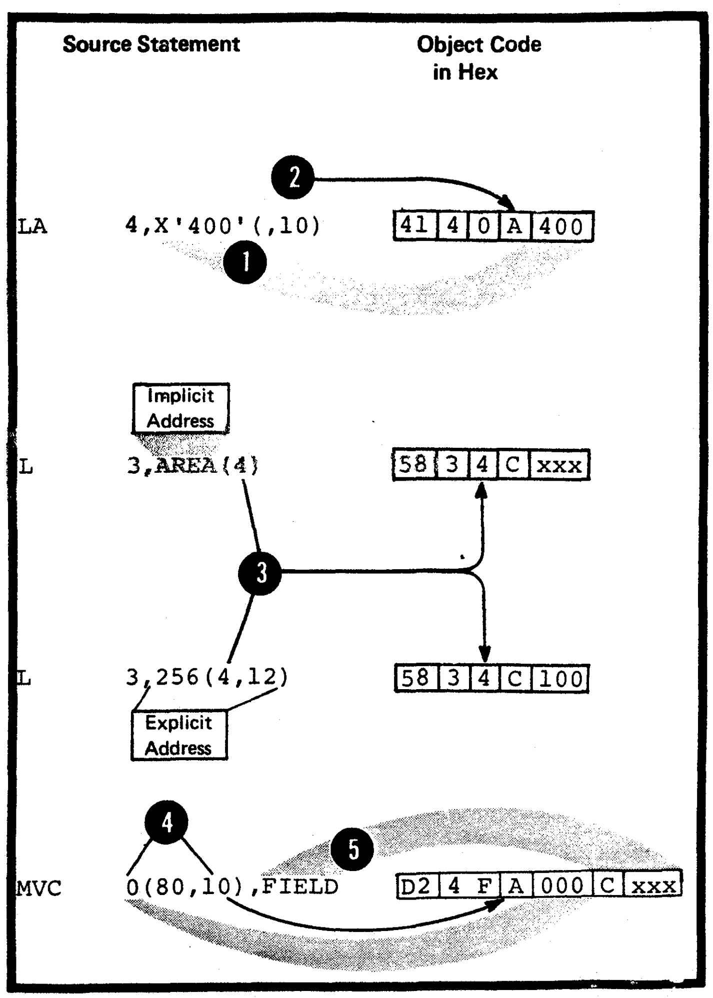
An explicit address is specified by coding two absolute expressions as follows:

(1) The first is an absolute expression for the displacement, whose value must lie in the range 0 through 4095 (4095 is the maximum value that can be represented by the 12 binary bits available for the displacement in the object code).

(2) The second (enclosed in parentheses) is an absolute expression for the base register, whose value must lie in the range 0 through 15.

If the base register contains a value that changes when the program is relocated, the assembled address is relocatable. If the base register contains a fixed absolute value that is unaffected by program relocation, the assembled address is absolute.

NOTES (for implicit and explicit addresses):
1. An explicit base register designation must not accompany an implicit address.
2. However, in RX instructions an index register (3) can be coded with an implicit address as well as with an explicit address.
3. When two addresses are required, one address can be coded as an explicit address (4) and the other as an implicit address (5).



D5C -- LENGTHS

Purpose

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

Specifications

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

1. For an implicit address (see D5B above), it is the
❶ length attribute of the first or only term in the expression representing the implicit address.

2. For an explicit address (see D5B above), it is the
❷ length attribute of the first or only term in the expression that represents the displacement.

For details on the length attribute of symbols and other terms see C4C.

EXPLICIT LENGTH: When a length subfield is specified in
❸ an SS-type machine instruction, the explicit length thus defined always overrides the implicit length.

NOTES:

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

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

Assembler Language Statement	Length Attribute of term (symbols)	Object Code in Hex L= Length Value (4)
Implicit Lengths		Address: L TO FROM
MVC TO,FROM	L'TO = 80	D2 4F xxxx xxxx
(1)		L
MVC TO+80,FROM	L'TO = 80	D2 4F xxxx xxxx
AP AREA,TWO	L'AREA = 8 L'TWO = 4	L1 L2 FA 7 3 xxxx xxxx
(1) MVC 0(,10),80(10)	1	L D2 00 A000 A050
(2) MVC FROM-TO(,10),80(10)	L'FROM =240	L D2 EF A0A0 A050
Explicit Lengths	(3)	Address: TO FROM
MVC TO(160),FROM	L'TO = 80	D2 9F xxxx xxxx
	(3)	L
MVC 0(80,10),80(10)	1	D2 4F A000 A050
CLC 0(1,10),256(10)	1 (5)	D5 00 A000 A100
CLC 0(0,10),256(10)	1 (5)	D5 00 A000 A100

```
TO   DS  CL80
FROM DS  CL240
AREA DS  PL8
TWO  DC  PL4'2'
```

D5D -- IMMEDIATE DATA

Purpose

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

You should be careful to specify immediate data only where it is required. Do not confuse it with address references to constants and areas or with any literals you specify as the operands of machine instruction (for a comparison between constants, literals, and immediate data, see C5).

Specifications

Immediate data must be specified as absolute expressions whose range of values depends on the machine instruction for which the data is required. The immediate data is
1 assembled into its 4-bit or 8-bit binary representation 2
according to the figure on the opposite page.

Machine Instructions in which immediate data is required (Op codes in Appendix IV)		Range of Values allowed for immediate data	Examples	Object Code in Hex
SRP	(SS)	0 through 9	SRP A,B,3, ❶	F0 7 3 xxxx xxxx 7 = Length of Field A; xxxx xxxx = A B Addresses
All BCR All BC	(RR) (RX)	0 through 15 0 through 15	BCR 8,3 ❶ BC 11,AAA	07 8 3 47 B 0 xxxx (xxxx = AAA Address)
ICM STCM CLM	(RS)	0 through 15	STCM 3,X'F',BBB ❶	BE 3 F xxxx (xxxx = BBB Address)
NI CLI XI MVI OI TM RDD WRD	(SI)	0 through 255	CLI SLOT,C'A' ❷ TM KEY,X'7F' ❷	95 C1 xxxx (xxxx = Address SLOT) 91 7F xxxx (xxxx = Address KEY)
SVC	(RR)	0 through 255	SVC 128 ❷	0A 80

D6 – Examples of Coded Machine Instructions

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

The object code assembled from at least one coded statement per group is also included. A complete summary of machine instruction formats with the coded assembler language variants can be found in Appendix III and IV.

RR Format

You use the instructions with the RR format mainly to move data between registers. The operand fields must thus designate registers, with the following exceptions:

(1) In BCR branching instructions when a 4-bit branching mask replaces the first register specification

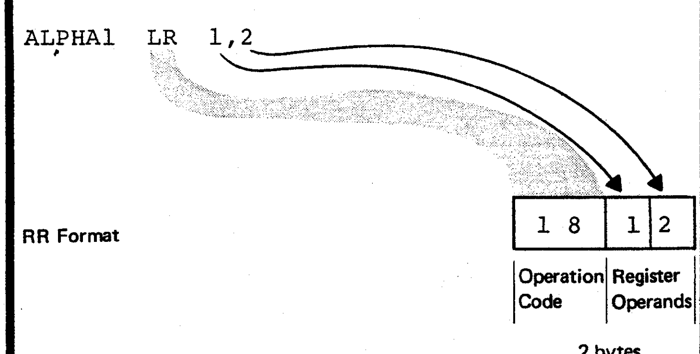
(2) In SVC instructions, where an immediate value (between 0 and 255) replaces both registers.

(3) NOTE: Symbols used in RR instructions are assumed to be equated to absolute values between 0 and 15.

Name	Operation	Operand
ALPHA1	LR	1,2
ALPHA2	LR	INDEX,REG2 (3)
GAMMA1	BCR	(1) 8,12
DELTA1	SVC	200 (2)
DELTA2	SVC	TEN

Assembly Examples:

Assembler Language Statement | **Object Code of Machine Instruction in Hex**



RX Format

You use the instructions with the RX format mainly to move data between a register and virtual storage. By adjusting the contents of the index register in the RX-instructions you can change the location in virtual storage being addressed. The operand fields must therefore designate registers, including index registers, and virtual storage addresses, with the following exception:

(1) In BC branching instructions a 4-bit branching mask, with a value between 0 and 15, replaces the first register specification.

NOTES:

(2) 1. Symbols used to represent registers are assumed to be equated to absolute values between 0 and 15.

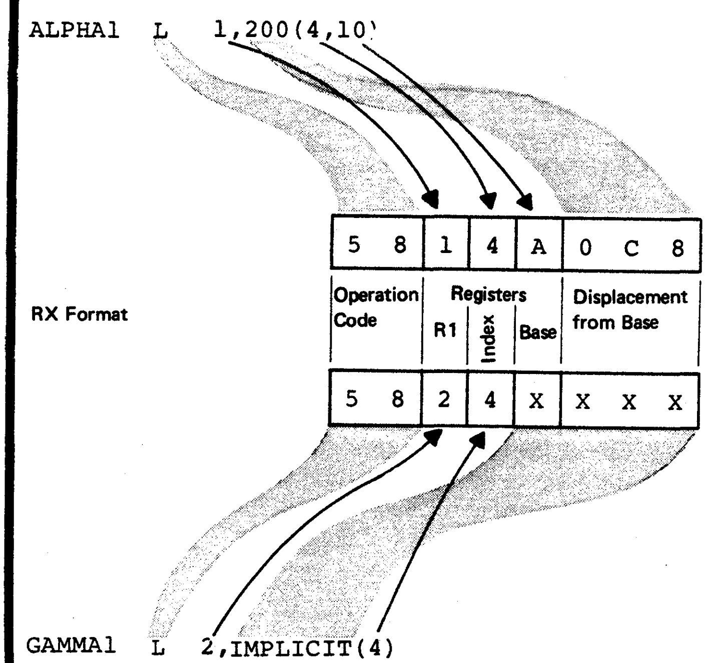
(3) 2. Symbols used to represent implicit addresses can be either relocatable or absolute.

(4) 3. Symbols used to represent displacements in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

Name	Operation	Operand
ALPHA1	L	1,200(4,10)
ALPHA2	L	REG1,200(INDEX,BASE) (2)
BETA1	L	2,200(,10) — No Indexing
BETA2	L	REG2,DISPL(,BASE) (4) — No Indexing
GAMMA1	L	3,IMPLICIT (3)
GAMMA2	L	3,IMPLICIT(INDEX) (3)
DELTA1	L	4,[redacted] — Literal Specification See C5
LAMDA1	BC	7,DISPL(,BASE) (4)
LAMDA2	BC	TEN,ADDRESS (1)

Assembly Examples:

Assembler Language Statement | **Object Code of Machine Instruction in Hex**



RS Format

You use the instructions with the RS format mainly to move data between one or more registers and virtual storage or to compare data in one or more registers (see the BXH and BXLE operations in Appendix IV).

In the Insert Characters under Mask (ICM) and the Store Characters Under Mask (STCM) instructions, when a (1) 4-bit mask, with a value between 0 and 15, replaces the second register specification.

NOTES:

(2) 1. Symbols used to represent registers are assumed to be equated to absolute values between 0 and 15.

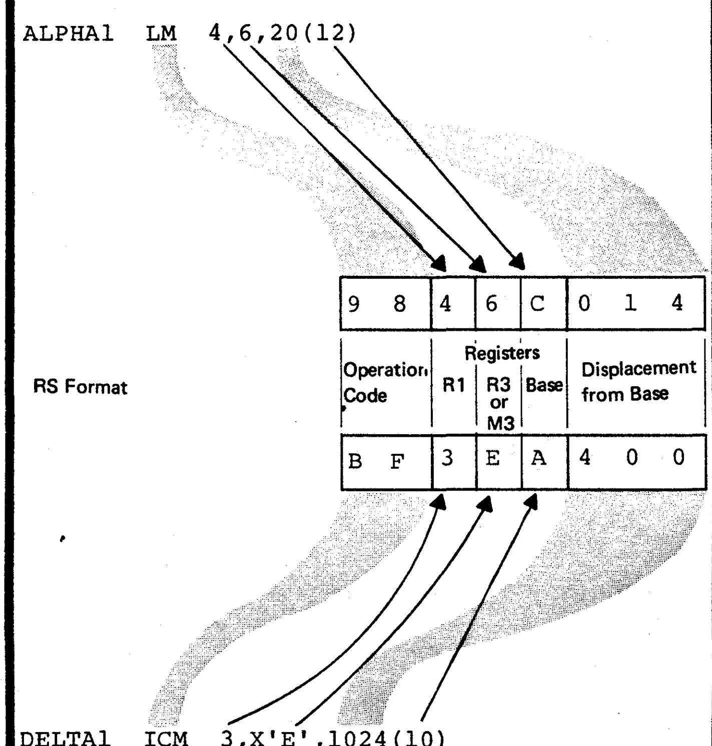
(3) 2. Symbols used to represent implicit addresses can be either relocatable or absolute.

(4) 3. Symbols used to represent displacements in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

Name	Operation	Operand
ALPHA1	LM	4,6,20(12)
ALPHA2	LM	REG4,REG6,20(BASE) (2)
BETA1	STM	4,6,AREA (3)
BETA2	STM	4,6,DISPL(BASE) (4)
GAMMA1	SLL	2,15
GAMMA2	SLL	2,0(15)
DELTA1	ICM	3,X'E',1024(10)
DELTA2	ICM	REG3,MASK,IMPLICIT (3) (1)

Assembly Examples:

Assembler Language Statement | **Object Code of Machine Instruction In Hex**



SI Format

You use the instructions with the SI format mainly to move immediate data into virtual storage. The operand fields must therefore designate immediate data and virtual storage addresses, with the following exception:

❶ An immediate field is not needed in the statements whose operation codes are: LPSW, SSM, TS, TCH, and TIO.

NOTES:

1. Symbols used to represent ❷ immediate data are assumed to be equated to absolute values between 0 and 255.

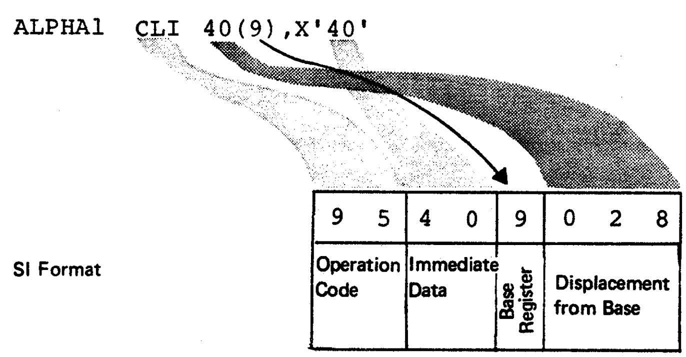
2. Symbols used to represent implicit ❸ addresses can be either relocatable or absolute.

3. Symbols used to represent ❹ displacements in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

Name	Operation	Operand
ALPHA1	CLI	40(9),X'40'
ALPHA2	CLI	❹ DISPL40(NINE),HEX40 ❷
BETA1	CLI	❸ IMPLICIT,TEN ❷
BETA2	CLI	KEY,C'E'
❶ GAMMA1	LPSW	0(9)
❶ GAMMA2	LPSW	NEWSTATE ❸

Assembly Examples:

Assembler Language Statement — **Object Code of Machine Instruction In Hex**



S Format

You use the instructions with the S format to perform I/O and other system operations and not to move data in virtual storage.

The operation codes for these instructions are given in the figure to the right. They are assembled into two bytes.

Mnemonic Operation Codes	Assembled Operation Code in Hex	Description
SIO	9C00	Start I/O
SIOF	9C01	Start I/O fast release
HIO	9E00	Halt I/O
HDV	9E01	Halt Device
STIDP	B202	Store CPU ID
STIDC	B203	Store Channel ID
SCK	B204	Set Clock
STCK	B205	Store Clock
SCKC	B206	Set Clock Comparator
STCKC	B207	Store Clock Comparator
SPT	B208	Set CPU Timer
STPT	B209	Store CPU Timer
PTLB	B20D	Purge Translation Lookaside Buffer
RRB	B213	Reset Reference Bit

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 the following exception:

(1) In the Shift and Round Decimal (SRP) instruction a 4-bit immediate data field, with a value between 0 and 9, is specified as a third operand.

NOTES:

(2) 1. Symbols used to represent base registers in explicit addresses are assumed to be equated to absolute values between 0 and 15.

(3) 2. Symbols used to represent explicit lengths 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.

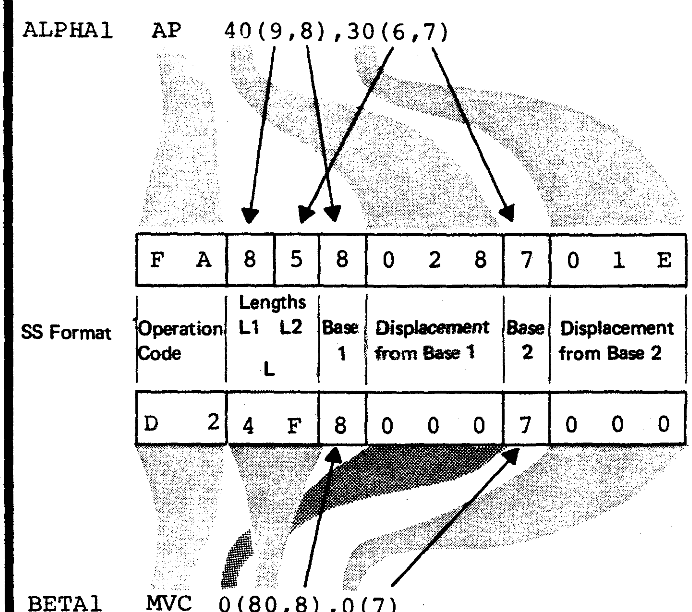
(4) 3. Symbols used to represent implicit addresses can be either relocatable or absolute.

(5) 4. Symbols used to represent displacements in explicit addresses are assumed to be equated to absolute values between 0 and 4095.

Name	Operation	Operand
ALPHA1	AP	40(9,8),30(6,7)
ALPHA2	AP	40(NINE,BASE8),30(SIX,BASE7) (3) (2)
ALPHA3	AP	FIELD1,FIELD2 (4)
ALPHA4	AP	AREA(9),AREA2(6)
ALPHA5	AP	DISP40(,8),DISP30(,7) (5)
BETA1	MVC	0(80,8),0(7)
BETA2	MVC	DISP0(,8),DISP0(7)
BETA3	MVC	TO,FROM
	SRP	FIELD1,X'8',3 (4) (1)

Assembly Examples:

Assembler Language Statement | **Object Code of Machine Instruction in Hex**



Part III: Functions of Assembler Instructions

SECTION E: PROGRAM SECTIONING

SECTION F: ADDRESSING

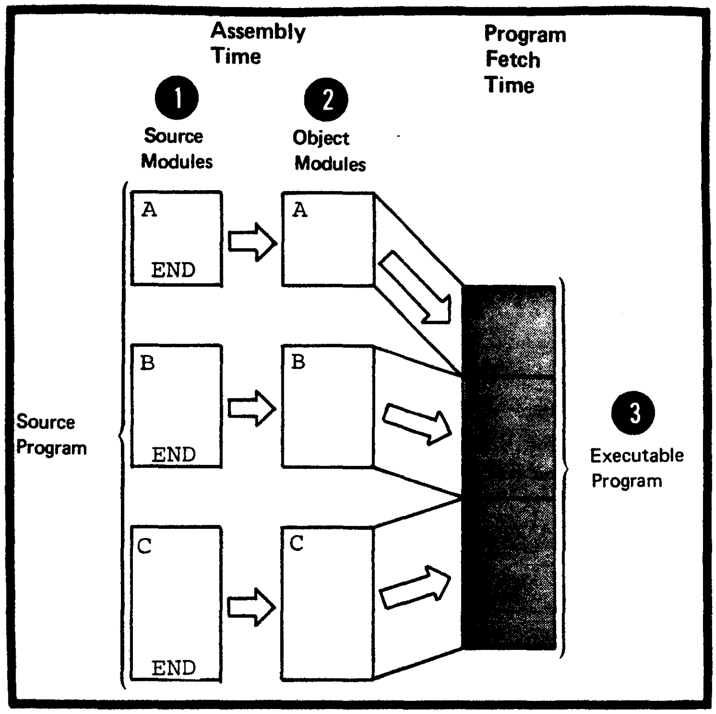
SECTION G: SYMBOL AND DATA DEFINITION

SECTION H: CONTROLLING THE ASSEMBLER PROGRAM

This page left blank intentionally.

Section E: Program Sectioning

This section 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.

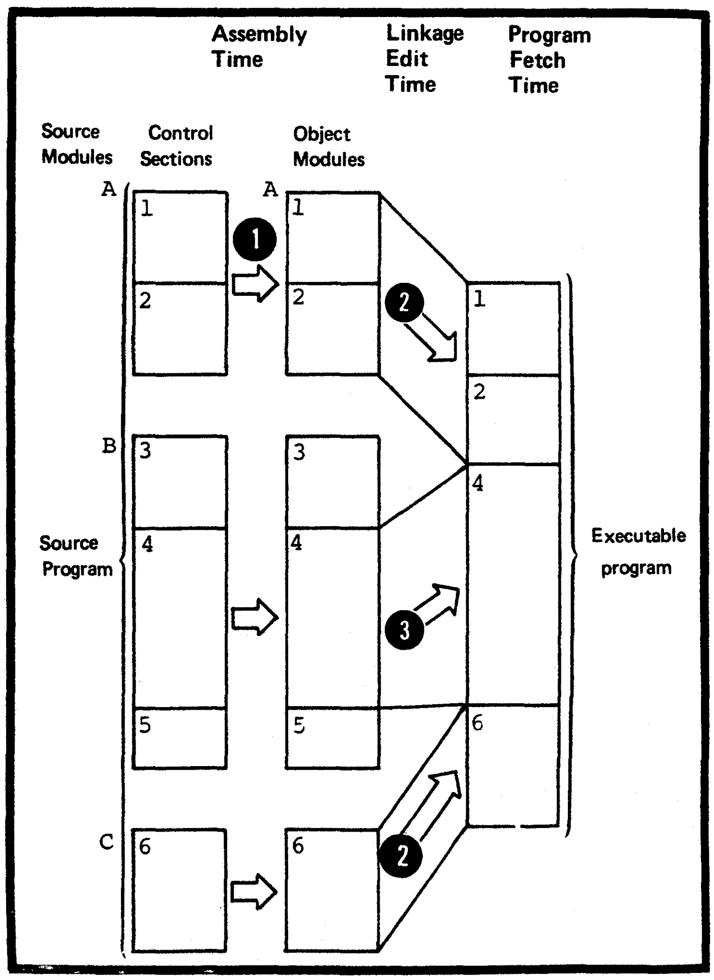


You should consider two different subdivisions when writing an assembler language program:

1. The source module

2. The control section.

You can divide a program into two
1 or more source modules. Each <u>source
module</u> is assembled into a separate
2 <u>object module</u>. The object modules
can then be combined into load
modules to form an <u>executable
3 program</u>.



You can also divide a source module
into two or more control sections.
1 Each control section is <u>assembled</u>
as part of an object module. By
writing the proper linkage edit
control statements, you can select
2 a complete <u>object module</u> or any
individual <u>control section</u> of the 3
object module to be linkage edited
and later loaded as an executable
program.

<u>SIZE OF PROGRAM PARTS</u>: If a source module becomes so large that its logic is not easily comprehensible, break it up 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 (for the assignment of base registers to control sections, see F1A).

<u>COMMUNICATION BETWEEN PROGRAM PARTS</u>: 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; symbolic linkage is described in F2.

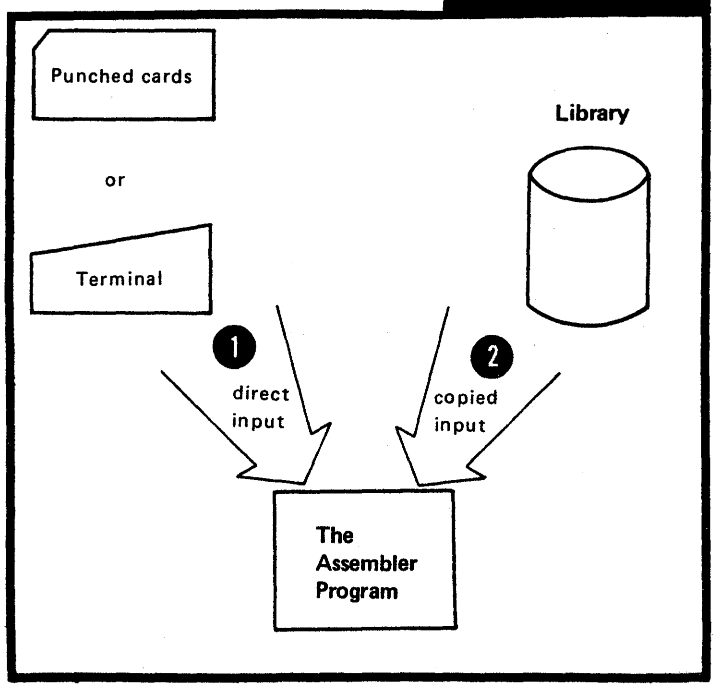
To communicate between two or more control sections within a source module, you must establish the addressability of each control section; establishing addressability is described in F1.

E1 -- The 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, for example, 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.



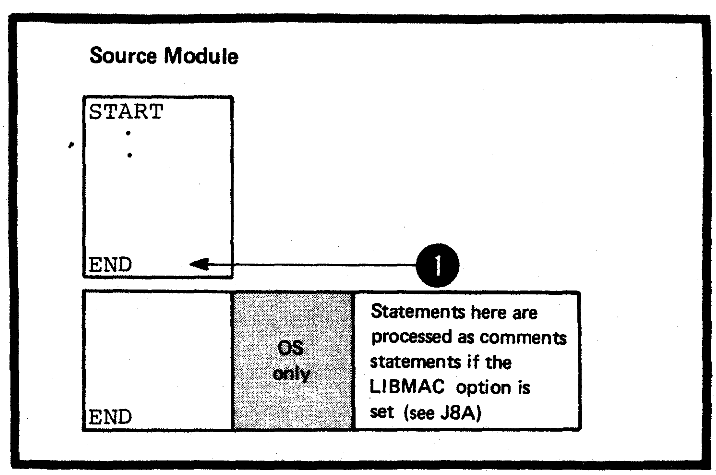
The Beginning of a Source Module

The first statement of a source module can be any assembler language statement, except MEXIT and MEND, that is described in this manual. You can initiate the first control section of a source module by using the START instruction. However, you can or must write some source statements before the beginning of the first control section (for a list of these statements see E2D).

The End of a Source Module

OS only

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.



OS only — NOTE: Conditional assembly processing can determine which of several substituted END instructions is to be processed. The conditional assembly language is described in Section L.

DOS — Only one END instruction is allowed. The assembler does not process any instruction that follows the END instruction.

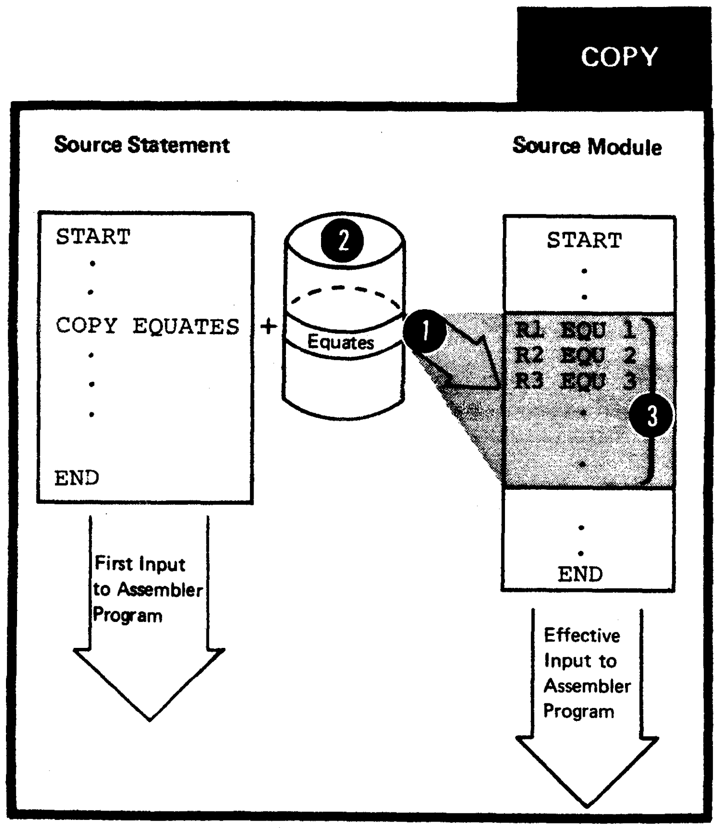
E1A -- THE COFY INSTRUCTICN

Purpose

1 2 3 The CCPY instruction allows you to copy predefined source statements from a library and include them in a source module. Ycu thereby avoid:

1. Writing the same, often-used sequence of code over and over

2. Keypunching and handling the punched cards for that code.



Specifications

The format of the COPY instruction statement is shown in the figure to the right.

Name	Operation	Operand
Blank	COPY	One ordinary Symbol

The symbol in the operand field must identify a part of a library called:

- A member of a partitioned data set

DOS A book in the source statement library

This member (or book) contains the coded source statements to be copied.

The source coding that is copied into a source module:

1 • Is inserted immediately after the COPY instruction

2 • Is inserted and processed according to the standard instruction statement coding format (described in B1D), even if an ICTL instruction has been specified

• Must not contain either an ICTL or ISEQ instruction

3 • Can contain a COPY instruction. Up to 5 levels of nesting of the COPY instruction are allowed.

DOS Up to 3 levels of nesting are allowed.

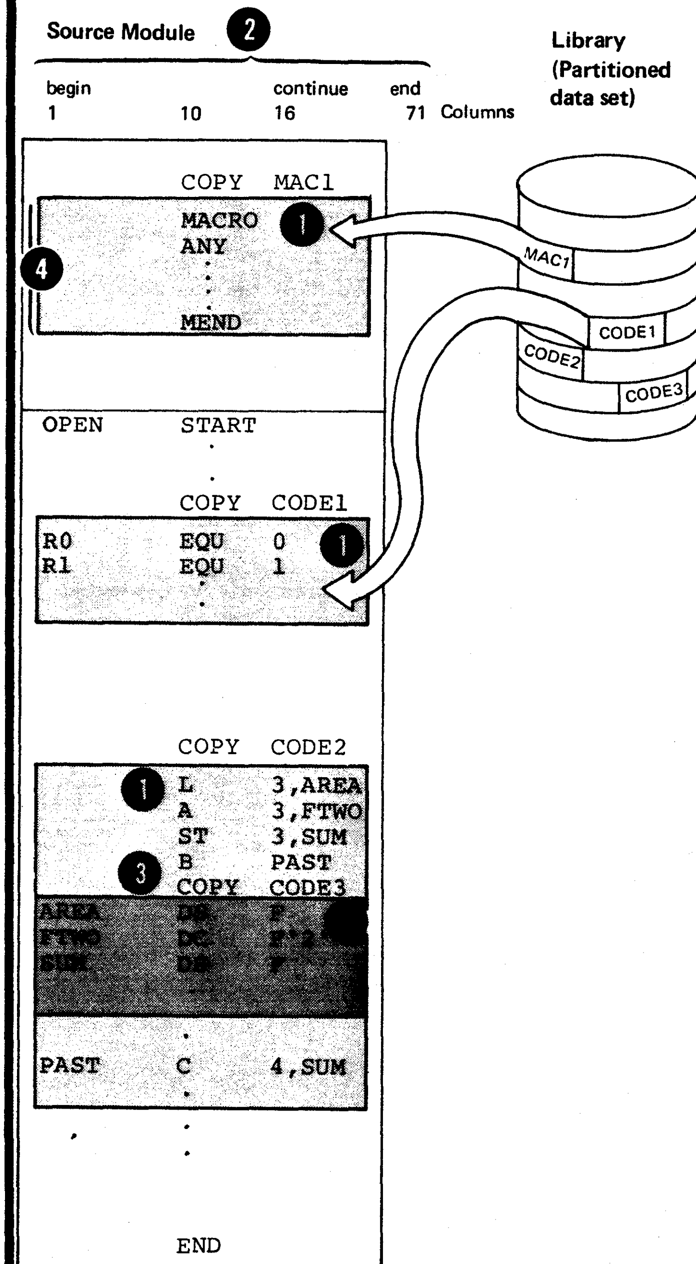
4 • Can contain macro definitions (see Section J).

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 (see J5C).

2. The rules that govern the occurrence of assembler language statements in a source module also govern the statements copied into the source module.



E1B -- THE END INSTRUCTION

Purpose

You use the END instruction to mark the end of a source module. It indicates to the assembler where to stop assembly processing. You can also supply an address in the operand field to which control can be passed when your program is loaded. This is usually the address of the first executable instruction in a source module.

Specifications

The format of the END instruction statement is shown in the figure to the right.

END

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

If specified, the operand entry can be generated by substitution into variable symbols. However, after substitution, that is, at assembly time:

(1) 1. It must be a relocatable expression representing an address in the source module delimited by the END instruction, or

(2) 2. If it contains an external symbol, the external symbol must be the only term in the expression, or (3) the remaining terms in the expression must reduce to zero.

3. It must not be a literal.

Source Module A

```
A       START   0
ENTERA  BALR    12,0
        USING   *,12
        ENTRY   ENTERA
          .
          .
                     (1)
        END     ENTERA
```

Source Module B

```
B       START   0
        BALR    11,0
        USING   *,11
        EXTRN   ENTERA
          .
          .
                 (2)             (3)
        END     ENTERA + (Subexpression)
```

This page left blank intentionally.

E2 - General Information About 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.

E2A -- AT DIFFERENT PROCESSING TIMES

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

1 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.

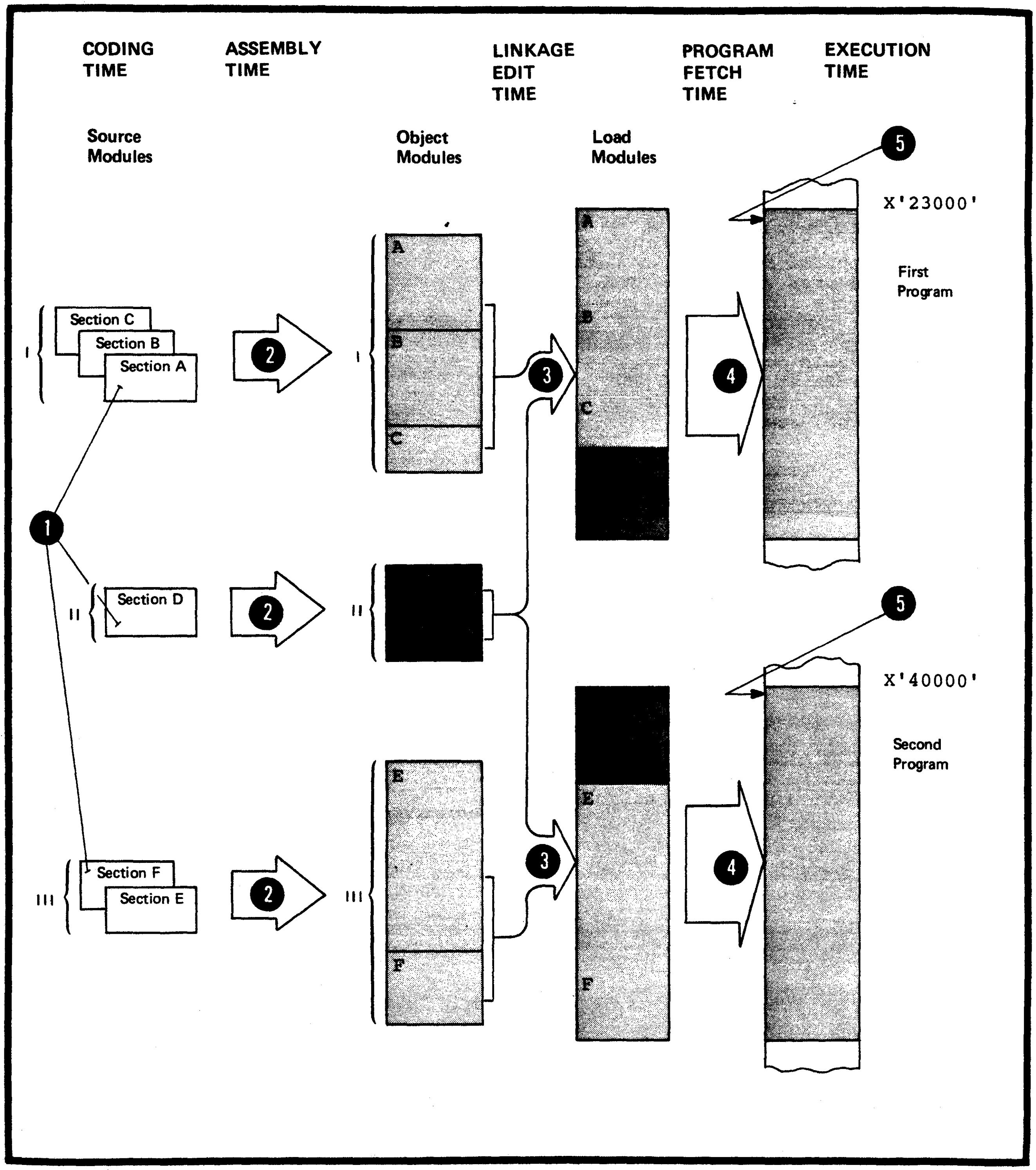
2 AT ASSEMBLY TIME: The assembler translates the source statements in the control section into object code. Each source module is assembled into one object module. The entire object module and each of the control sections it contains is relocatable.

3 AT LINKAGE 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 (see E4).

4 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.

5 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 linkage editor control statement or in the operand field of an END statement.


CODING TIME
ASSEMBLY TIME
LINKAGE EDIT TIME
PROGRAM FETCH TIME
EXECUTION TIME
Source Modules
Object Modules
Load Modules
X'23000'
First Program
Section C
Section B
Section A
Section D
Section F
Section E
X'40000'
Second Program
A
B
C
E
F
I
II
III
1
2
3
4
5

Executable Control Sections

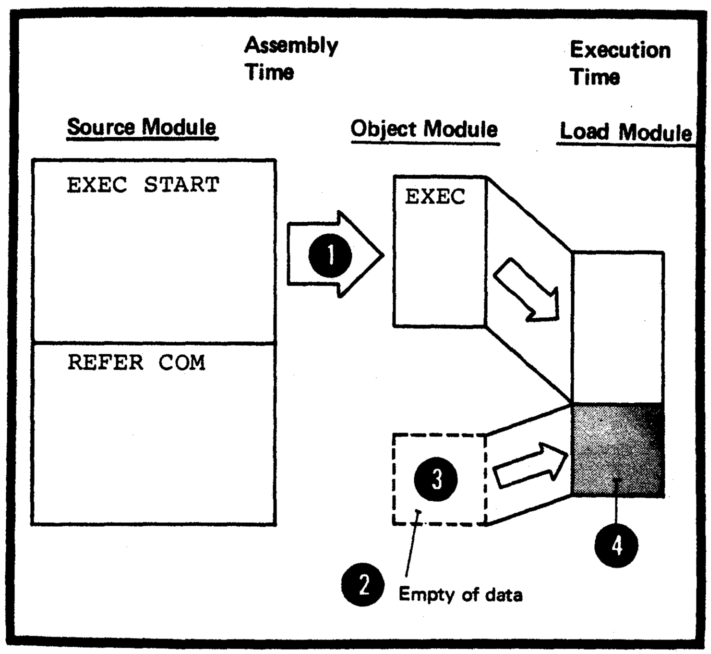
An executable control section is one you initiate by using the START or CSECT instructions and is 1 assembled into object code. At execution time, an executable control section contains the binary data assembled from your coded instructions and constants and is therefore executable.

An executable control section can also be initiated as "private code", without using the START or CSECT instruction (see E2E).

Reference Control Sections

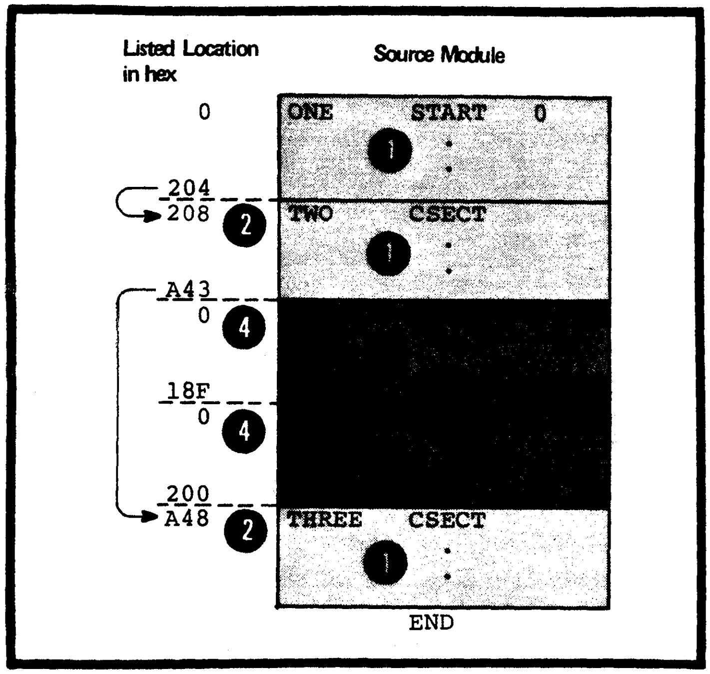
OS only

A reference control section is one you initiate by using the DSECT, COM, or DXD instruction and is not 2 assembled into object code. You can use a reference control section 3 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 4 binary data to which they refer is not entered until execution time.



E2C -- 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.



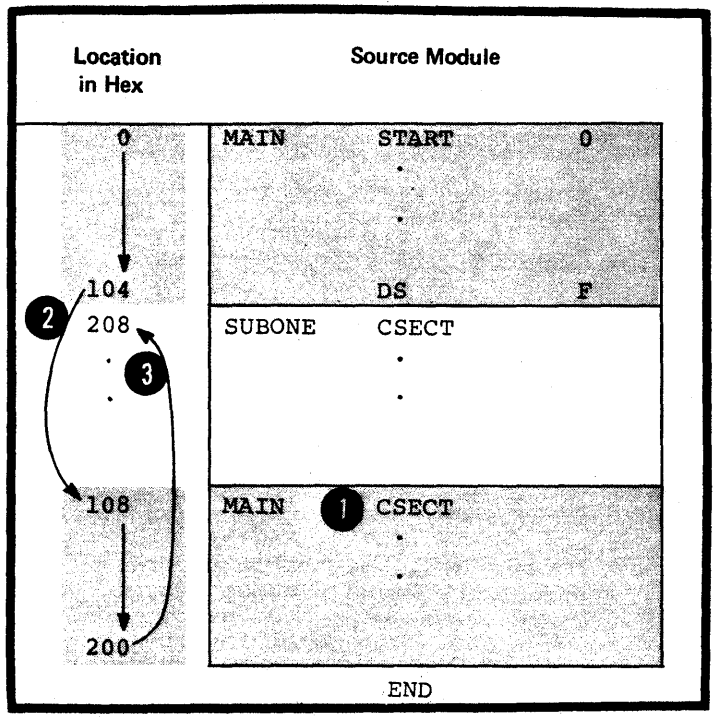
(1) However, for executable control sections, the location values that (2) appear in the listings do not restart at 0 for each subsequent executable control section. They carry on from the end of the previous control section. Your executable control sections are usually loaded into storage in the order 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.

DOS For executable control sections, the location values that appear in the listings always start from 0, except the control section initiated by a START instruction with a non-zero operand entry.

(3) For reference control sections, the location values that appear (4) in the listings always start from 0.

1 You can continue a control section that has been discontinued by another control section and thereby intersperse code sequences from different control sections. Note 2 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.

OS only 3 The location values listed for the next control section defined begin after the last location value assigned to the preceding control section.



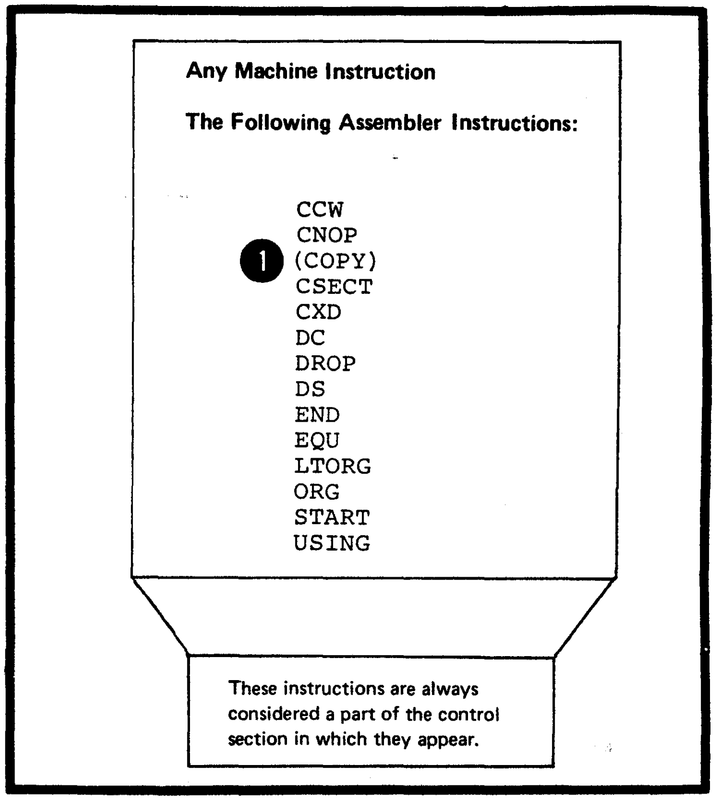
E2D -- FIRST CONTROL SECTION - SPECIFICATIONS

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 are listed in the figure to the right.

1 The statements copied into a source module by a COPY instruction, if specified, determine whether or not it will initiate the first control section.

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



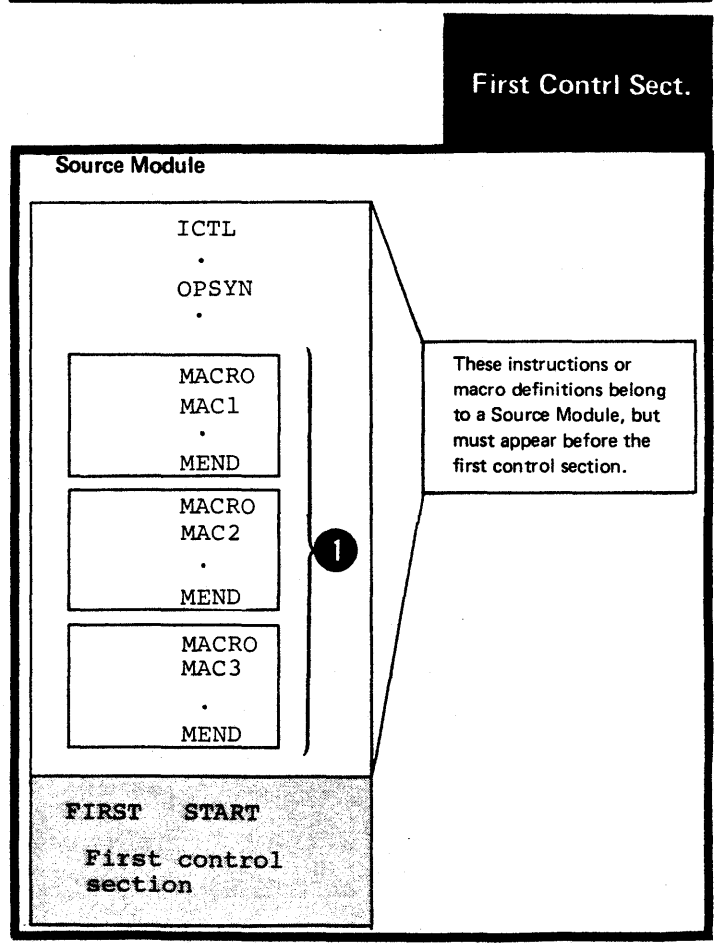
WHAT MUST COME BEFORE THE FIRST CONTROL SECTION: The following instructions or groups of instructions, if specified, must appear before the first control section, as shown in the figure to the right.

- The ICTL instruction, which, if specified, must be the first statement in a source module

OS only
- **The OPSYN instruction**

1
- Any source macro definitions (see J1B)

- 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 shown in the figure to the right.

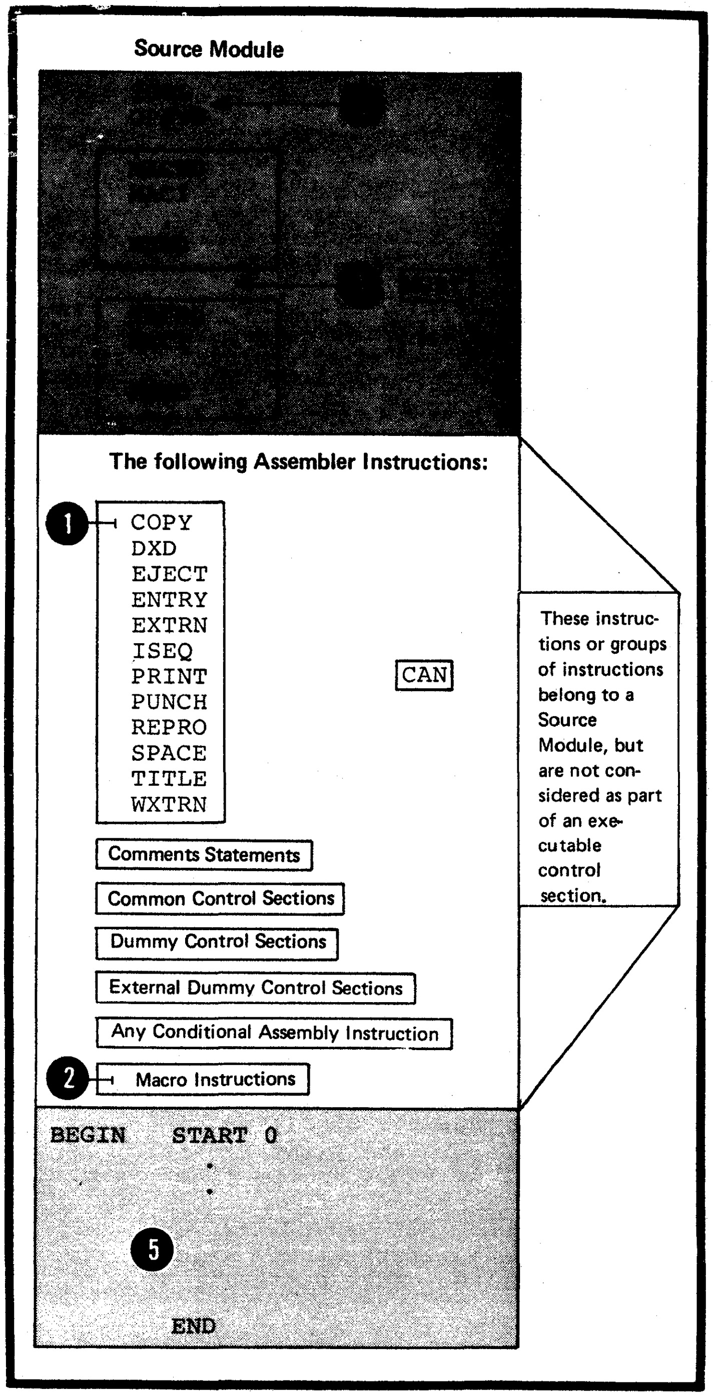
(1) Any instructions copied by a COPY instruction or generated by the (2) processing of a macro instruction before the first control section must belong exclusively to one of the groups of instructions shown in the figure to the right.

NOTES:

1. The EJECT, ISEQ, PRINT, SPACE, or TITLE instructions and comments (3) statements must follow the ICTL instruction, if specified. However, (4) they can precede or appear between source macro definitions. The OPSYN instruction must (1) follow the ICTL instruction, if specified, and (2) precede any source macro definition specified.

2. All the other instructions of the assembler language must follow any source macro definitions specified.

3. All the instructions or groups of instructions listed in the figure to the right can also appear as (5) part of a control section.



E2E -- THE UNNAMED CONTROL SECTION

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

(1) 1. By coding a <u>START or CSECT</u> instruction without a name entry

(2) 2. By coding any instruction, other than the START or CSECT instruction, that <u>initiates the first executable control section</u>.

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:

1. Within a source module

2. In EXTRN and WXTRN instructions and linkage editor control statements for linkage between source modules.

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

DOS **Only unnamed common control sections (initiated by the COM instruction) and named dummy control sections (initiated by the DSECT instruction) are allowed.**

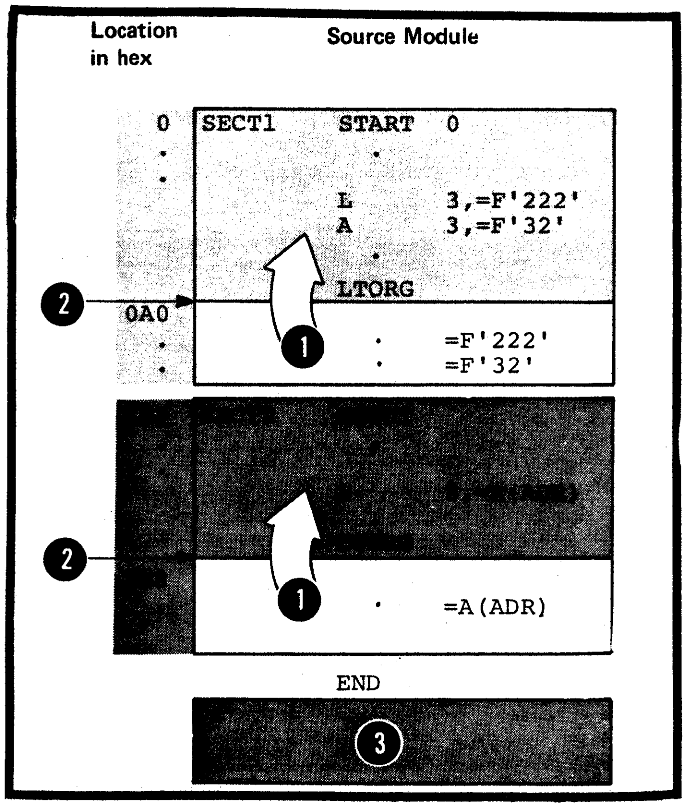
Type Code Assigned for External Symbol Dictionary	Unnamed Control Sections in separate Source Modules	Notes
PC	(1) START . . END	Unnecessary unless dictated by specific programming purpose
PC	(1) CSECT . . END	
PC	(2) BALR 12,0 USING*,12 . END	Inadvertent and inadvisable initiation of first control section: instead, precede with a named START instruction

PC signifies "private code"

E2F -- LITERAL POOLS IN CONTROL SECTIONS

Literals, collected into pools by the assembler, are assembled as (1) part of the <u>executable control section</u> 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 (2) pool <u>starting at the LTORG instruction</u>. 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 <u>after the</u> (3) <u>END instruction</u>.

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. (For a complete discussion of the literal pool see H1B.)



E2G -- 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.

1. Its symbolic name, if one is specified

2. Its type code

3. Its individual identification

4. Its starting address.

The figure to the right lists:

1. The assembler instructions that ❶ define control sections and dummy control sections or identify entry and external symbols, ❷

2. The type code that the assembler assigns to the control sections or dummy control sections and to the entry and external symbols.

NOTE: The total number of entries identifying separate control sections, dummy control sections, entry symbols, and external symbols in the external symbol dictionary must not exceed 399. External symbols identified in a Q-type address constant and specified as the ❸ name entry of a DSECT instruction are counted twice in determining this total.

DOS **The maximum number of external symbol dictionary entries (control sections, dummy control sections, and external symbols) allowed is 511. The maximum allowable number of symbols identified by the ENTRY instruction is 200.**

Name Entry		Instruction	Type code entered into external symbol dictionary
optional		START	SD } if name entry is present
		CSECT	SD
		START	PC } if name entry is omitted
		CSECT	PC
		Any instruction that initiates the unnamed control section	PC
optional DOS blank	❶	COM	CM
optional DOS mandatory		DSECT	none
OS only mandatory		DXD ❸	XD
mandatory		(external DSECT)	XD
	❷	ENTRY	LD
		EXTRN	ER
		DC(V-type address constant)	ER
		WXTRN	WX

E3 – Defining a Control Section

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

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

E3A -- THE START INSTRUCTION

Purpose

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

1. 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.

2. 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.

3. To specify the initial setting of the location counter for the first or only control section.

Specifications

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.

The format of the START instruction statement is given in the figure to the right.

START

Name	Operation	Operand
Any Symbol or blank	START	A self-defining term, or blank

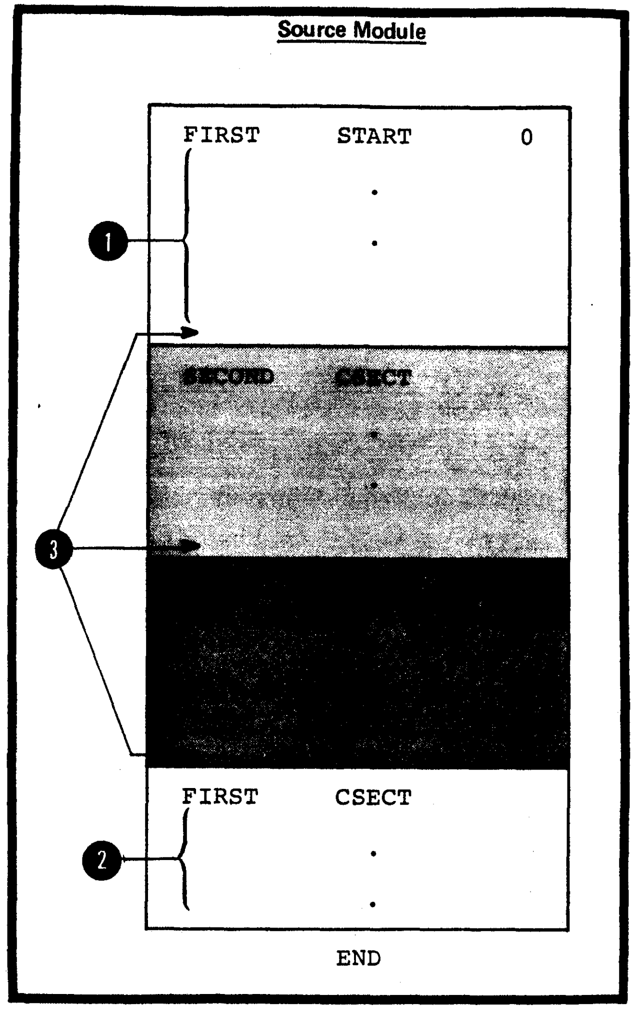
1 The symbol in the name field, if specified, identifies the first control section. It must be used in the name field of any CSECT
2 instruction that indicates the continuation of the first control section. This symbol represents the address of the first byte of the control section and has a length attribute value of 1.

3 The assembler uses the value of the self-defining term 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 eight, the assembler sets the initial value of the
4 location counter to the next higher doubleword boundary. If the operand
5 entry is omitted, the assembler sets the initial value to 0.

Location In Hex		Source Module	
	(1)		
000000	FIRST	START	0
		.	
		.	
000D00	BREAK	DS	F
	SECOND	CSECT	
		.	
		.	
000D04	(2) FIRST	CSECT	
000D04	CONTINUE	DS	F
		.	
		.	
		END	

Further Examples:

(3)			
001000	A	START	X'1000'
001000	B	START	4096
000020	C	START	30
(4)			
(5)			
000000	D	START	



❶ 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 ❷ follow this CSECT instruction are also assembled into the first control section.

Any instruction that defines a new or continued control section marks ❸ the end of the preceding control section or portion of a control section. The END instruction marks the end of the control section in effect.

E3B -- THE CSECT INSTRUCTION

Purpose

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

Specifications

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 causes the first control section to be initiated.

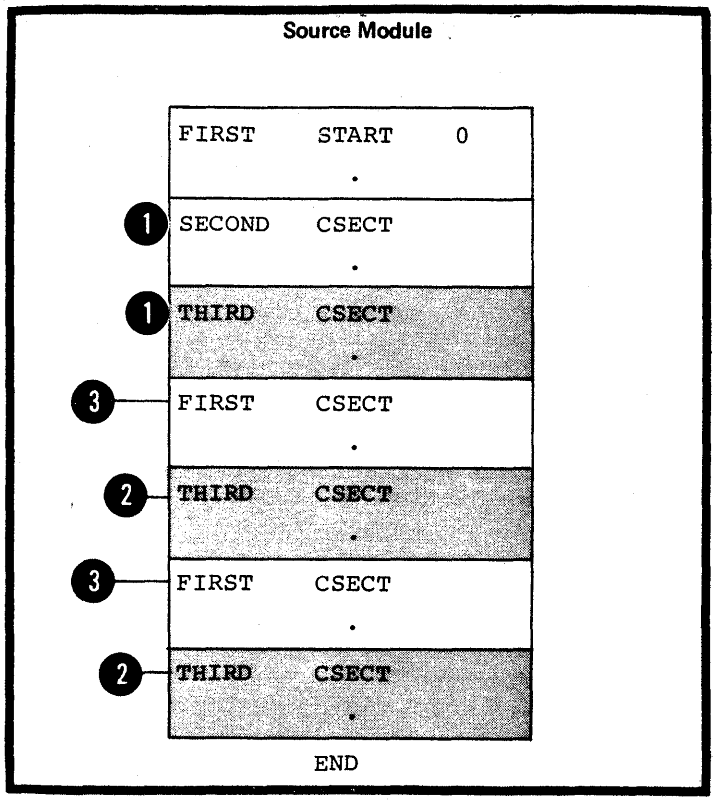
The format of the CSECT instruction statement is shown in the figure to the right.

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 (see E2E).



1. 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.

2. The beginning of a control section is aligned on a doubleword boundary. However, the continuation of a
3. control section begins at the next available location in that control section.

4. The source statements that follow a CSECT instruction that either initiates or indicates the continuation of a control section are assembled into the object code of the control section identified by that CSECT instruction.

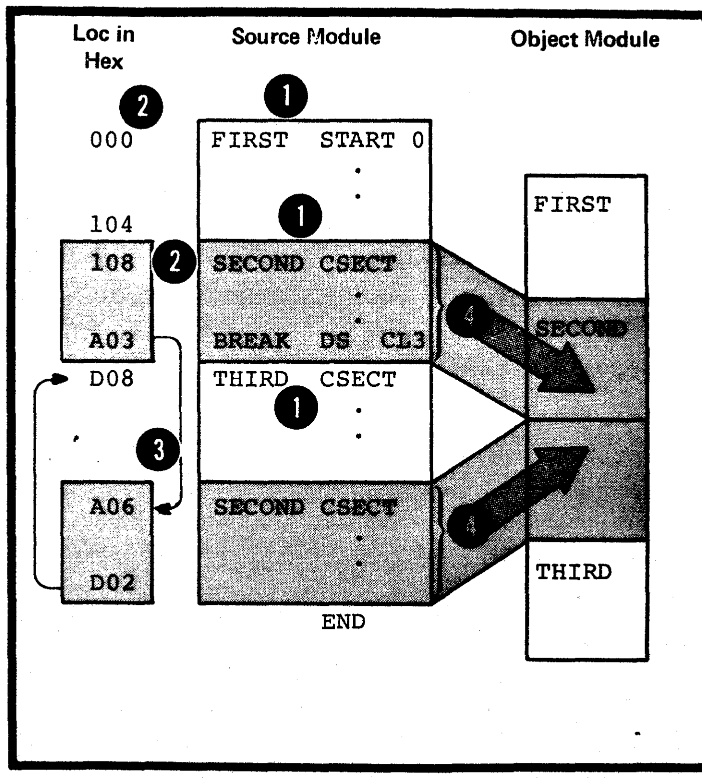
NOTES:

1. 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.

DOS 2. The location counter is reset to zero each time the DOS/VS assembler encounters a CSECT instruction. (The figure on the right illustrates location counter settings when using the OS/VS assembler.)



E3C -- THE DSECT INSTRUCTION

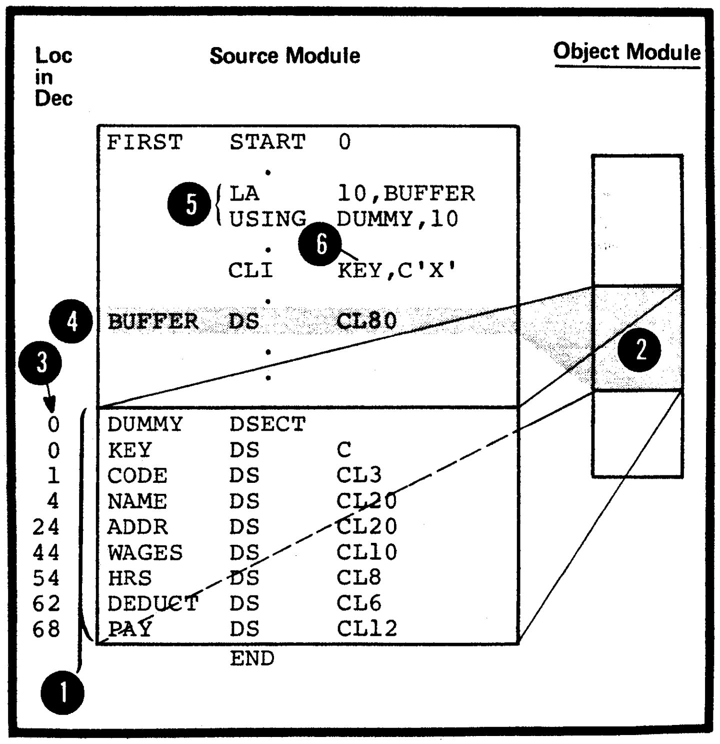
Purpose

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.

How to Use a Dummy Control Section

The figure to the right illustrates a dummy control section.



A dummy control section (dummy section) allows you to write a (1) sequence of assembler language statements to describe the layout of (2) unformatted data located elsewhere in your program. The assembler produces no object code for statements in a dummy control section and it reserves no storage for the dummy section. Rather, the dummy section provides a symbolic format that is empty of data. However, the assembler (3) assigns location values to the symbols you define in a dummy section, relative to the beginning of that dummy section.

Therefore, to use a dummy section you must:

- (4) 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

- (5) Establish the addressability of the dummy section in combination with the storage area (see F1A).

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

Specifications

The DSECT instruction identifies the beginning or continuation of a dummy control section (dummy 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.

The format of the DSECT instruction statement is given in the figure to the right.

DSECT

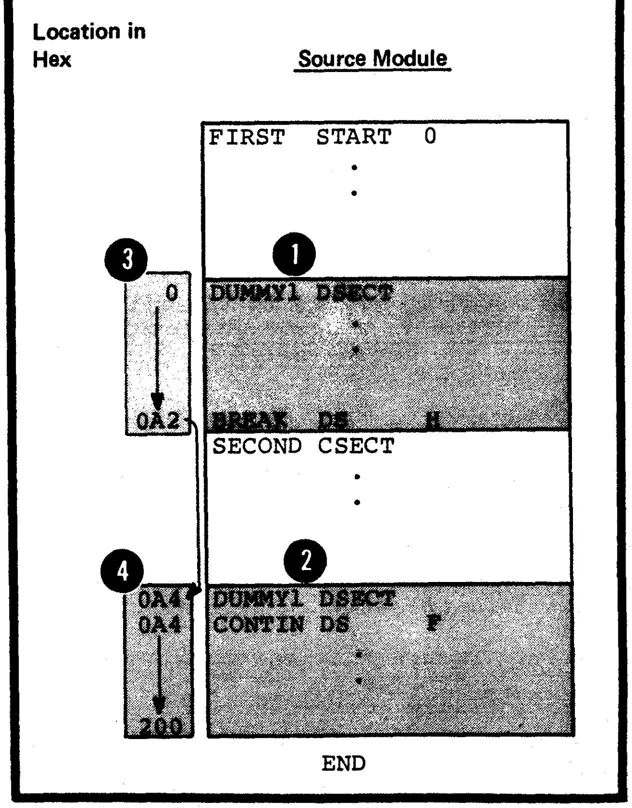
Name	Operation	Operand
Any Symbol or blank DOS Ordinary Symbol or Variable Symbol	DSECT	Not required

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 ❶ initiates the dummy section and the rest ❷ 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 ❸ initial value of 0. However, the continuation of a dummy section begins at the ❹ next available location in that dummy section.



(1) 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 control section are not assembled into object code.

2. When establishing the addressability of a dummy section, (2) the symbol in the name field of (3) 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 (4) address constant only if the symbol is paired with another symbol from the same dummy section, and if the symbols have the opposite sign.

```
FIRST    START    0
           .
         USING    DUMMY1,10   or USING A,10
           .        (2)                (3)
           .

DUMMY1   DSECT
A        DS       H        }
B        DS       H        }
C        DS       CL30     } (1)
           .               }
           .               }

SECOND   CSECT     (4)
           .
ADCON    DC       A(FROM-TO)
           .
           .

DUMMY1   DSECT
           .               }
TO       DS       CL80     }
FROM     DS       CL240    } (1)
           .               }
           .               }

         END
```

Purpose

You can use the COM instruction to initiate a common control section or to indicate its continuation. 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

The figure to the right illustrates a common control section.

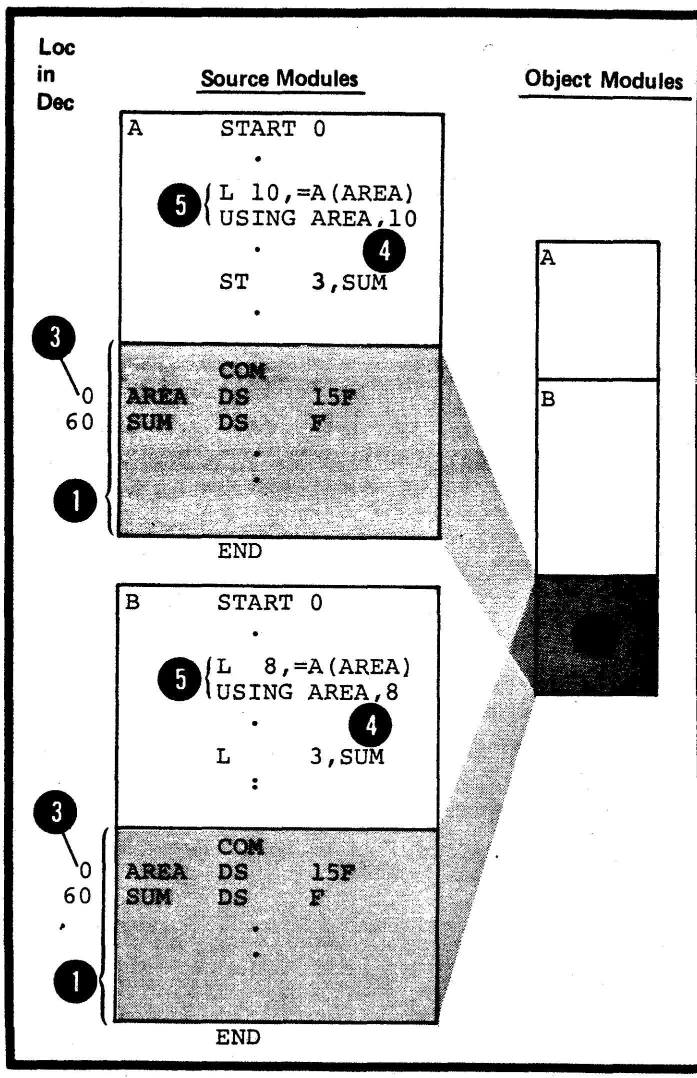
A common control section (common section) allows you to describe
1 a common storage area in one or more source modules.

When the separately assembled object modules are linked as one program,
2 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.

3 The assembler assigns locations to the symbols you define in a common section relative to the beginning of that common section.
4 This allows you to refer symbolically to the data that will be loaded at execution time. Note that you
5 must establish the addressability of a common control section in every source module in which it is specified (see F1A). 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.



Specifications

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

One or more common sections can be defined in a source module.

[illegible] Only one common section can be defined.

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.

The format of the COM instruction statement is given in the figure to the right.

COM

Name	Operation	Operand
Any Symbol or blank DOS Must be blank	COM	Not required

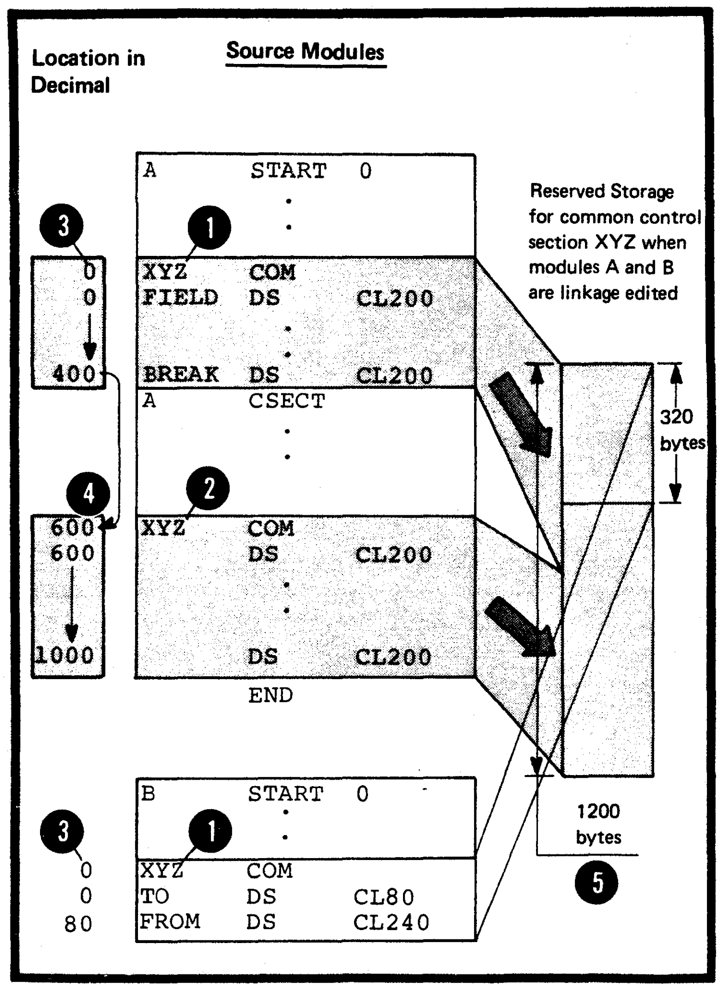
OS only The symbol in the name field, if specified, identifies the common control section. If several COM instructions within a source module have the same symbol in the name field, the first occurrence initiates the common section and the rest indicate the continuation of the common section.

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.

3 The location counter for a common section is always set to an initial value of 0. However, the continuation of a common section begins at the next available location in that common section.

5 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.



❶ 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, ❷ 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.

DOS Because the name entry of the COM instruction must be blank, a symbol ❸ defined in the common section must be used as the base address in a USING instruction.

Source Module

```
FIRST     START   0
            .                ❷                ❸
          USING   COMMON,11         or USING A,11
            .
            .

COMMON    COM
A         DS      3D    ┐
B         DS      3F    │
C         DS      30H   │ ❶
            .           │
            .           ┘
SECOND    CSECT
            .
            .
COMMON    COM
D         DS      CL20  ┐
E         DS      CL30  │
F         DS      XL2000│ ❶
            .           │
            .           ┘
          END
```

E4 – External Dummy Sections

OS only

Purpose

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:

1. Work areas for each source module or

2. Communication areas between two or more source modules.

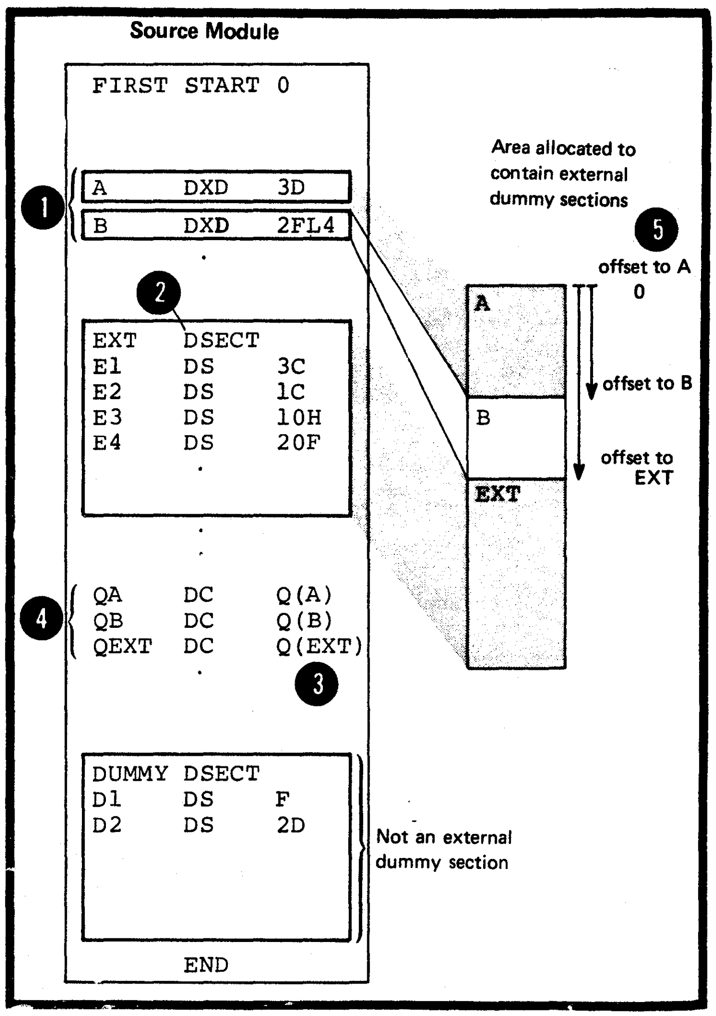
When the assembled object modules are linked and loaded, you can dynamically allocate the storage required for all your external dummy sections at one time from one source module (for example, by using the GETMAIN macro instruction). This is not only convenient but you save space and prevent fragmentation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

To generate and use external dummy sections, you need to specify a combination of the following:

1. The DXD or DSECT instruction

2. The Q-type address constant

3. The CXD instruction.



Generating an External Dummy Section

1 2 An external dummy section is generated when you specify a DXD instruction or a DSECT instruction in combination with a Q-type address 3 constant that contains the name of the DSECT instruction.

4 5 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. The Q-type address constant is described in G3M.

OS only

How to Use External Dummy Sections

To use an external dummy section, you must do the following (as illustrated in the figure below):

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 (for this example, register 11). Note that register 11 must contain this address throughout the whole program.

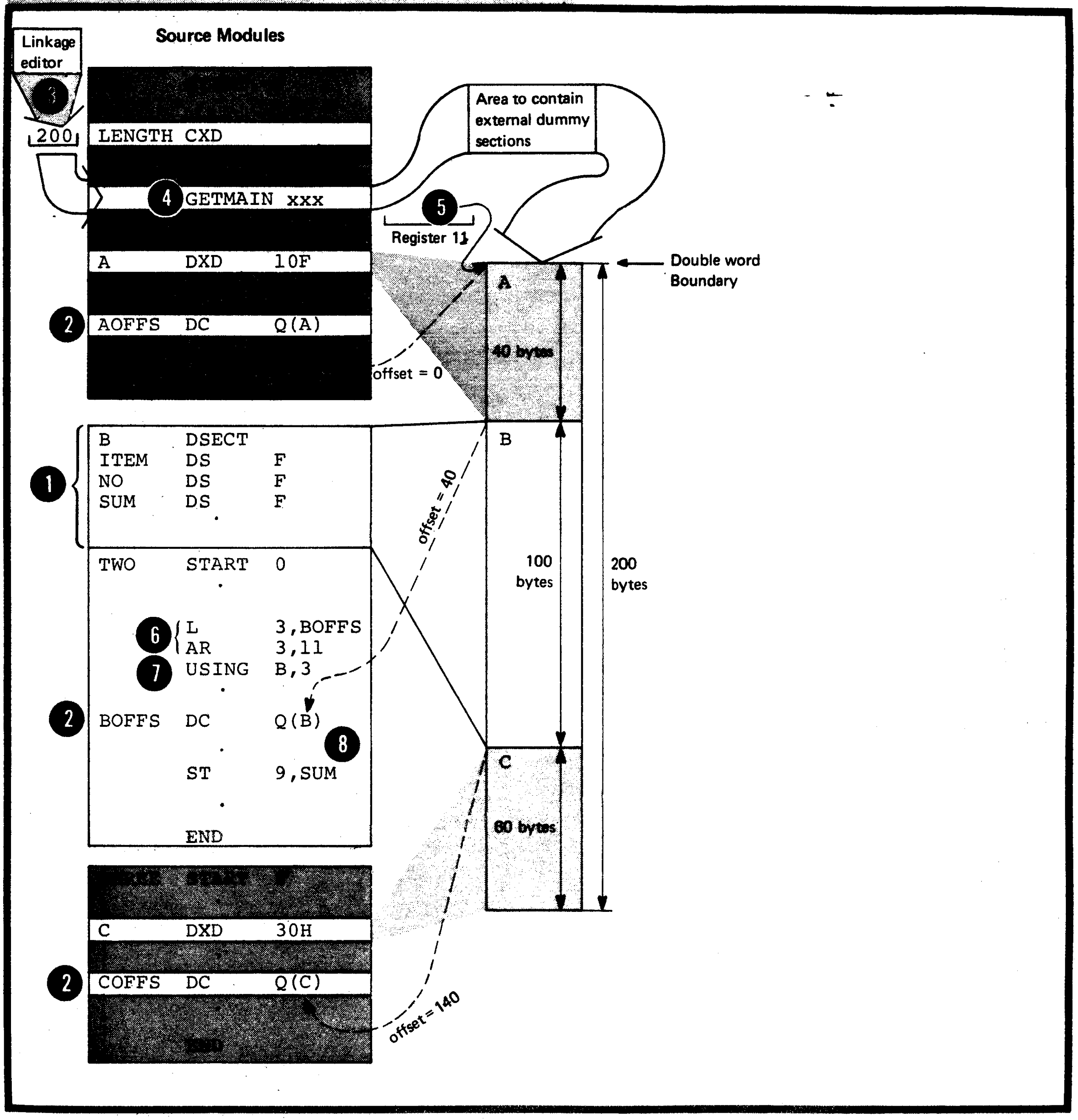
6. Add, to the address in register 11, the offset into the allocated area of the desired external dummy section. The linkage editor inserts this offset into the fullword area reserved by the appropriate Q-type address constant.

7. Establish the addressability of the external dummy section in combination with the portion of the allocated area reserved for the external dummy section.

8. 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.

OS only


Linkage editor
Source Modules
200
LENGTH CXD
GETMAIN xxx
A DXD 10F
AOFFS DC Q(A)
Area to contain external dummy sections
Register 11
Double word Boundary
offset = 0
A
40 bytes
B DSECT
ITEM DS F
NO DS F
SUM DS F
TWO START 0
L 3,BOFFS
AR 3,11
USING B,3
BOFFS DC Q(B)
ST 9,SUM
END
offset = 40
B
100 bytes
200 bytes
C
60 bytes
C DXD 30H
COFFS DC Q(C)
offset = 140
1
2
3
4
5
6
7
8

E5 -- Defining an External Dummy Section

OS only

E5A -- THE DXD INSTRUCTION

Purpose

The DXD instruction allows you to identify and define an external dummy section.

Specifications

The DXD instruction defines 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.

NOTE: 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.

The format of the DXD instruction is given in the figure to the right.

1 The symbol in the name field must appear in the operand of a Q-type address constant. This symbol represents the address of the first byte of the external dummy section defined and has a length attribute value of 1.

2 The subfields in the operand field are specified in the same way as in the 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.

DXD

Name	Operation	Operand
A symbol	DXD	Same format as the operand of a DS instruction

2 Subfields: Duplication Factor | Type | Modifiers | Nominal Value

Example:

```
A          DXD      10FL3
 .
 .
AOFFSET    DC       Q(A)
```

(1: the symbol A in the name field of the DXD instruction is referenced in Q(A))

E5B -- THE CXD INSTRUCTION

Purpose

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 together into one program.

Specifications

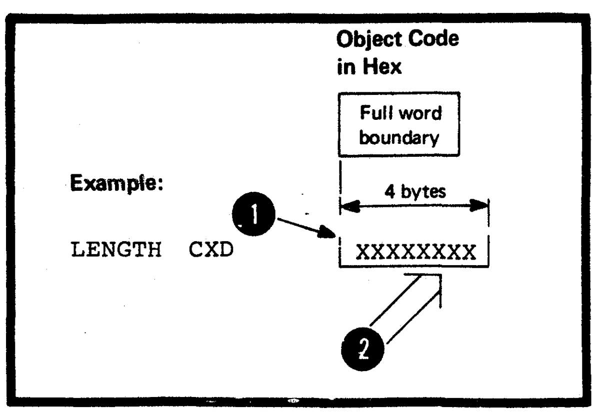
The CXD instruction reserves a fullword area in storage, and it can appear in one or more of the source modules assembled and combined by the linkage editor into one program.

The format of the CXD instruction statement is given in the figure to the right.

CXD

Name	Operation	Operand
A 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.



This page left blank intentionally.

Section F: Addressing

This section 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.

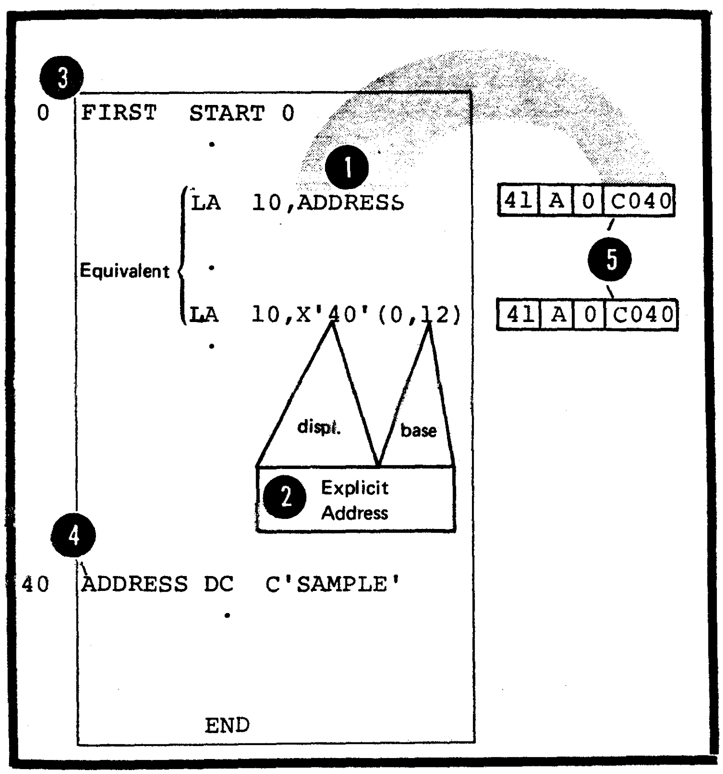
F1 — Addressing Within Source Modules: Establishing Addressability

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 (1) addresses, and the addresses in the base-displacement form are (2) called explicit addresses, both of which are fully described in D5B.

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:

(3) 1. A base address from which it can compute displacements to the (4) addresses within a control section and

(5) 2. A base register to hold this base address.



How to Establish Addressability

To establish the addressability of a control section, you must, at coding time:

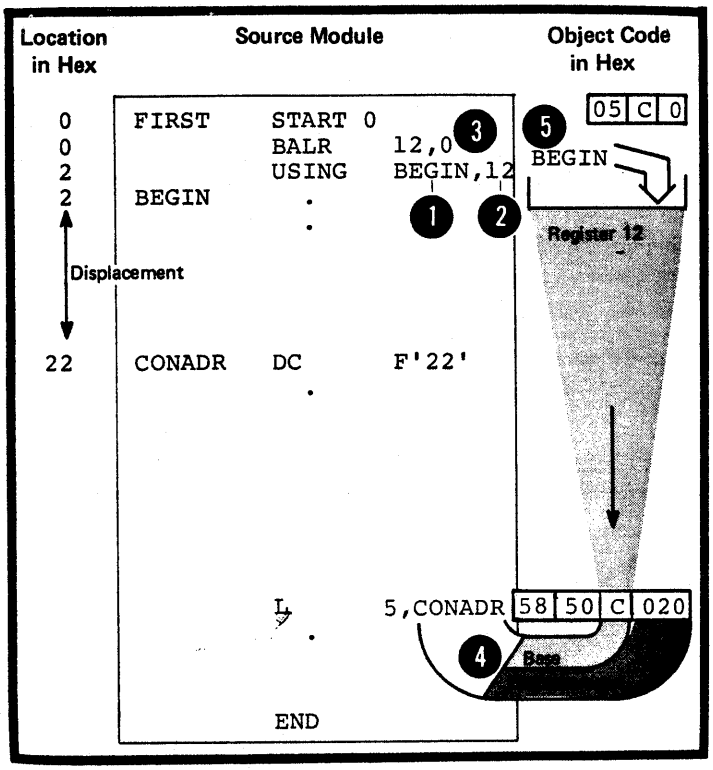
1 • Specify a base address from which the assembler can compute displacements

2 • Assign a base register to contain this base address

3 • Write the instruction that loads the base register with the base address.

At assembly time, 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. 4

At execution time, the base address is loaded into the base register and should remain there throughout the execution of your program. 5



F1A - THE USING INSTRUCTION

Purpose

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:

1. Know which locations in a control section are made addressable by the USING instruction

2. Know where in a source module you can use these established addresses as implicit addresses in instruction operands.

The Range of a USING Instruction

The range of a USING instruction (called the USING range) is the 4,096 bytes beginning at the (1) base address specified in the USING instruction. Addresses that lie within the USING range can be (2) converted from their implicit to their explicit form; those outside (3) 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 (4) 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. If register 0 is specified as the base register, the USING range will be location 0 through 4095 regardless of the base address specified.

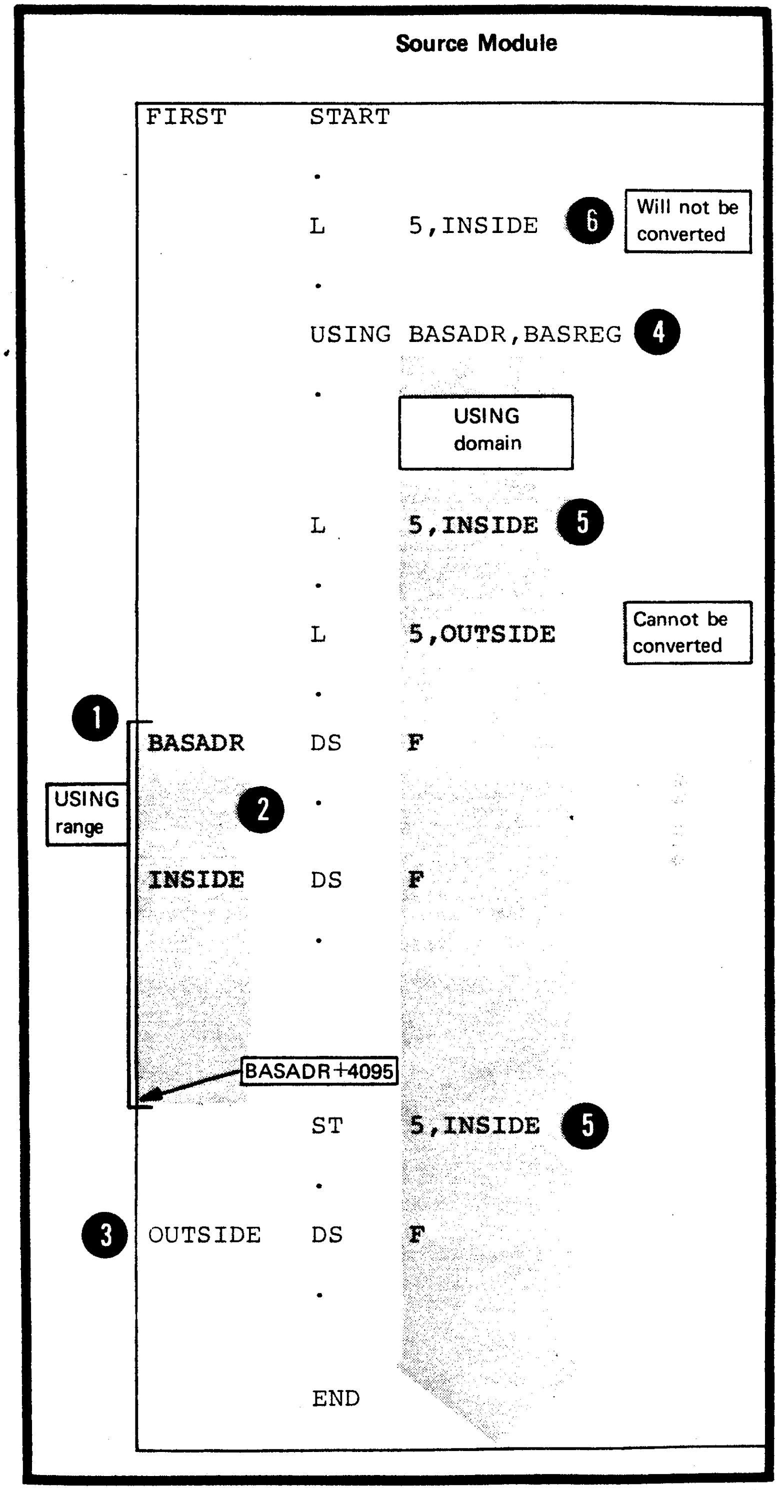
The Domain of a USING Instruction

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 in this subsection, under NOTES ABOUT THE USING DOMAIN.) The (5) assembler converts implicit address references into their explicit form:

1. If the address reference appears in the domain of a USING instruction and

2. If the addresses referred to lie within the range of the same USING instruction.

(6) 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.



This page left blank intentionally.

How to Use the USING Instruction

You should specify your USING instructions so that:

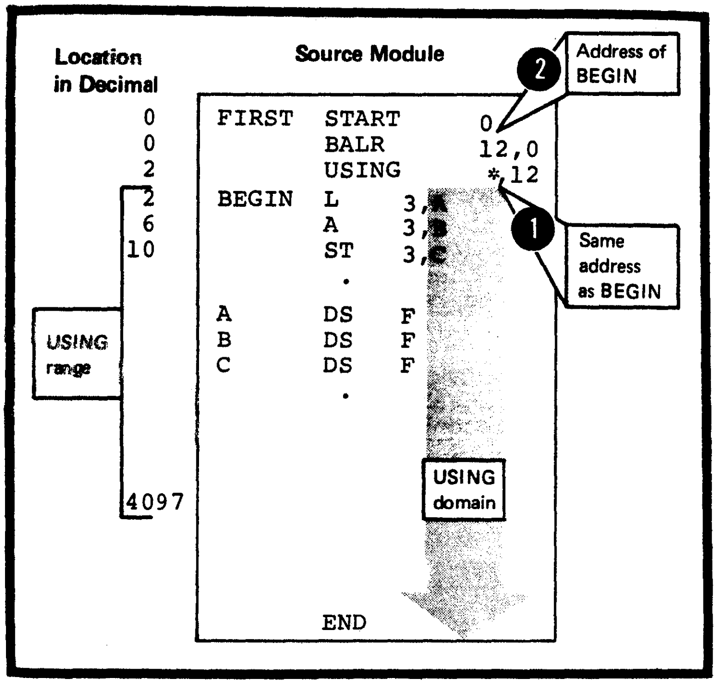
1. All the addresses in each control section lie within a USING range and

2. 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: The figure to the right illustrates a way of establishing the addressability of an executable control section (defined by a START or CSECT instruction). You specify (1) a base address and assign a base register in the USING instruction. (2) At execution time the base register is loaded with the correct base address.

Note that for this particular combination of the BALR and USING instructions, you should code them exactly as shown in the figure to the right.



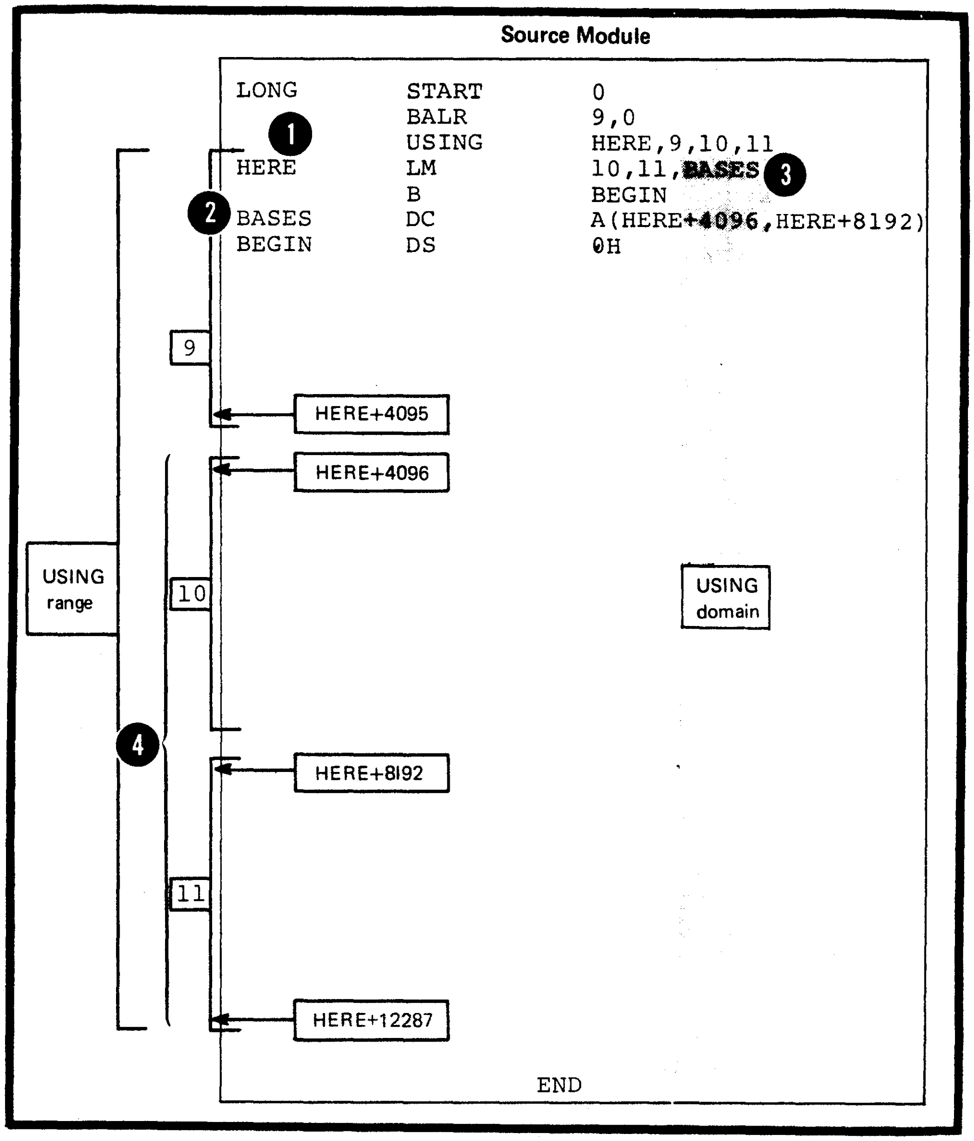
If a control section is longer than 4096 bytes, you must assign more than one base register. This allows you to establish the addressability of the entire control section with one USING instruction as shown in the figure on the opposite page.

The assembler assumes that the base registers that you assign contain the correct base addresses. The address
1 of HERE is loaded into the first base register. The addresses HERE+4096 and HERE+8192 are loaded into the second and third base registers respectively.

2 Note that you must define the address, BASES, within the first part of the total USING range, that is, the addresses covered by base register 9. This is because the explicit
3 address converted from the implicit address reference, is assembled into the LM instruction. At execution time, the assembled address must have a base register which already contains a base address at this point; the only base register loaded with its base address is register 9.

4 The addressability of addresses in the USING range covered by the second and third base registers is not completely established until after the LM instruction.

NOTE: Addresses specified in address constants (except the S-type) are not converted to their base-displacement form.

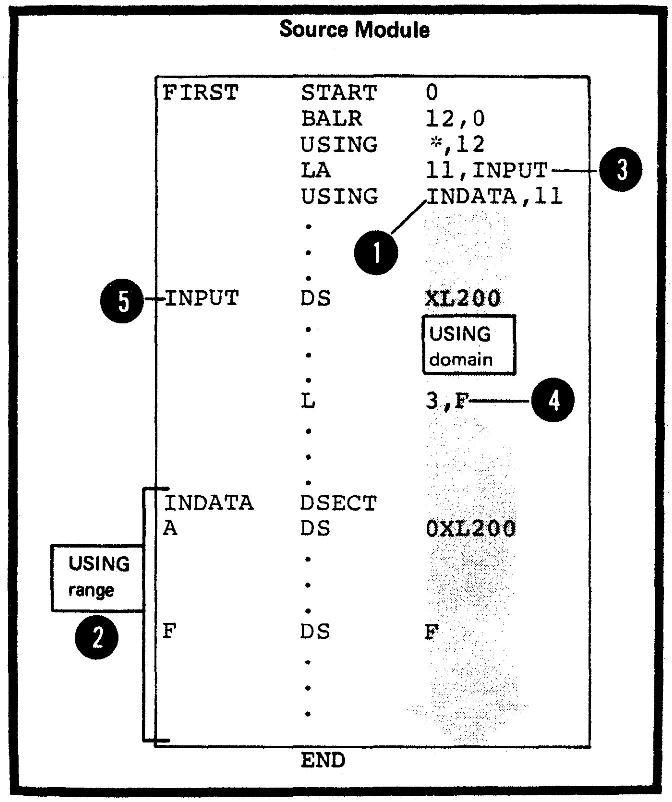

Source Module
LONG START 0
BALR 9,0
USING HERE,9,10,11
HERE LM 10,11,BASES
B BEGIN
BASES DC A(HERE+4096,HERE+8192)
BEGIN DS 0H
1
2
3
4
9
10
11
HERE+4095
HERE+4096
HERE+8192
HERE+12287
USING range
USING domain
END

FOR REFERENCE CONTROL SECTIONS: The figure to the right illustrates how to establish the addressability of a dummy section. A dummy section is a reference control section defined by the DSECT instructions. Examples of establishing addressability for the other reference control sections are given in E3D and E4.

(1) As the base address, you should specify the address of the first byte of the dummy section, so that all its addresses lie within the (2) pertinent USING range.

(3) The address you load into the base register must be the address of the storage area being formatted by the dummy section.

Note that the assembler assumes that you are referring to the (4) symbolic addresses of the dummy section, and it computes displacements accordingly. However, at execution time, the assembled addresses refer to the location (5) of real data in the storage area.



Specifications for the USING Instruction

The USING instruction must be coded as shown in the figure to the right.

Name	Operation	Operand
Sequence symbol or blank	USING	BASE,BASREG1 [,BASEREG2] ... (2) (1)

Example:

```
USING    BASE,9,10,11
```

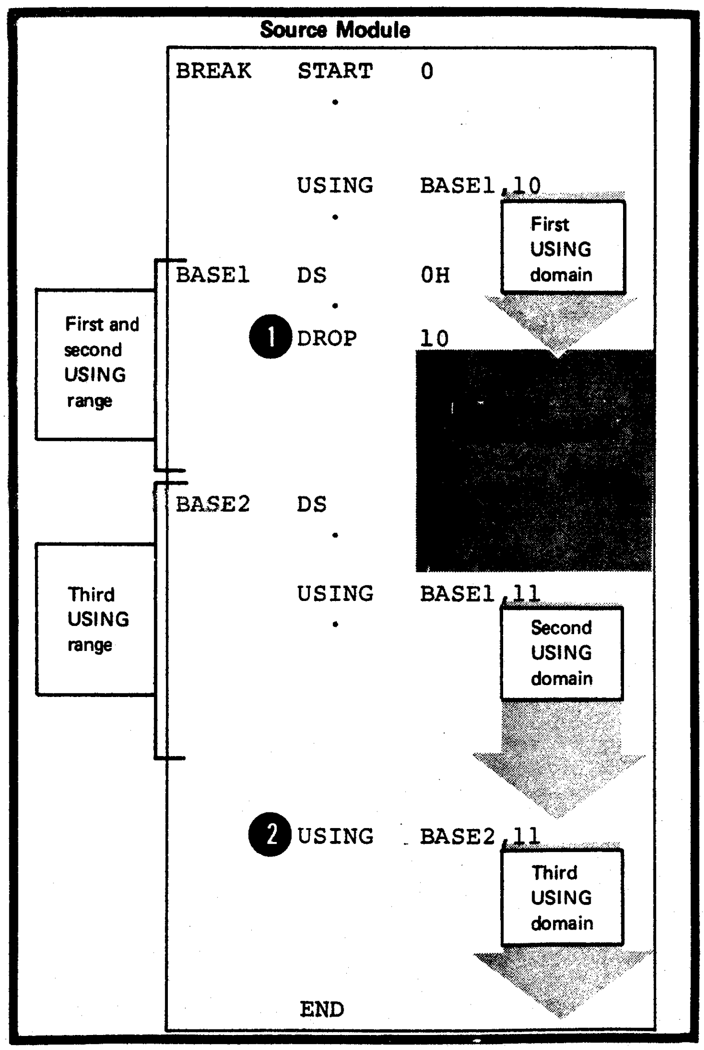
Logical Equivalent

```
USING    BASE,9
USING    BASE+4096,10
USING    BASE+8192,11
```

The operand, BASE, specifies a base address, which can be a relocatable or 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.

(1) (2) The assembler assumes that the first base register (BASREG1) contains the base address BASE at execution time. If present, the subsequent operands, BASREG2, BASREG3,..., represent registers that the assembler assumes will contain the address values, BASE+4096, BASE+8192,..., respectively.

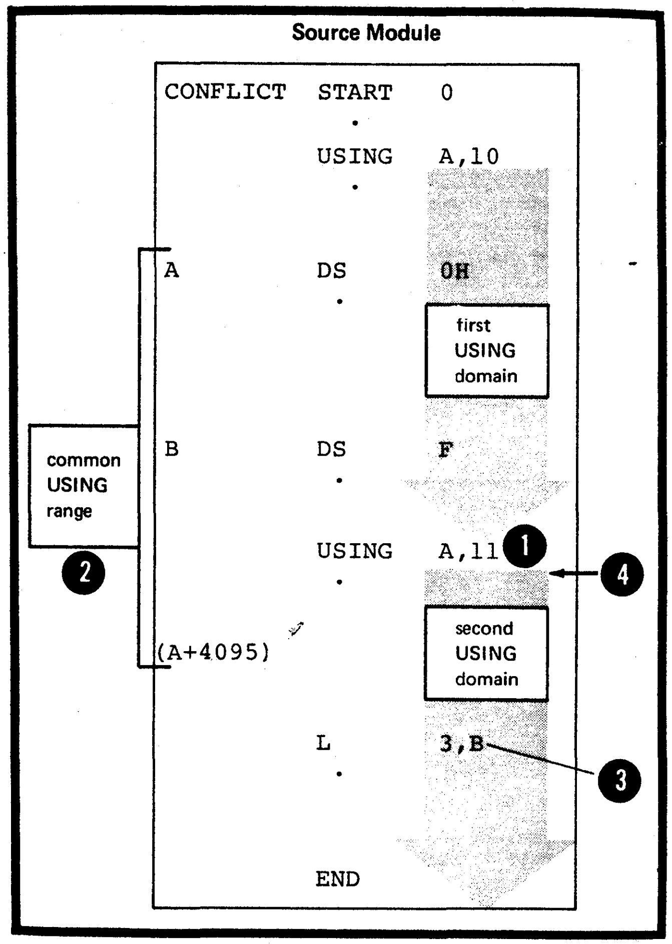


NOTES ABOUT THE USING DOMAIN: The domain of a USING instruction continues until the end of a source module except when:

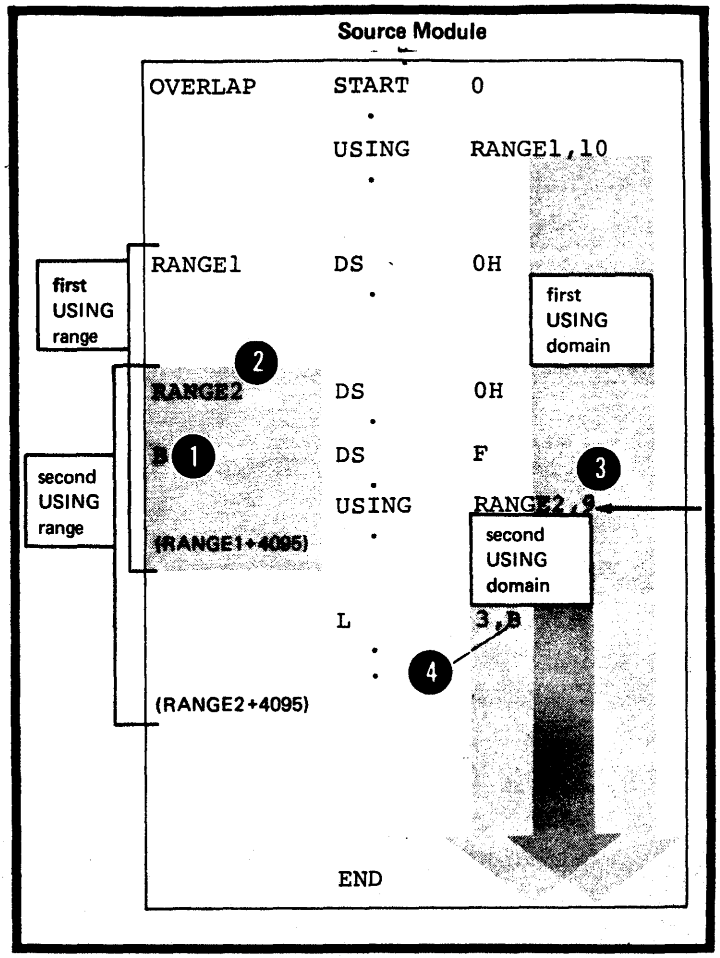
(1) • A subsequent DROP instruction specifies the same base register or registers assigned by the preceding USING instruction.

(2) • 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
1 higher numbered register for assembling the addresses within
2 the common USING range. In the example, this applies only to the
3 implicit addresses that appear after the second USING instruction. In effect, the first USING domain is
4 terminated after the second USING instruction.

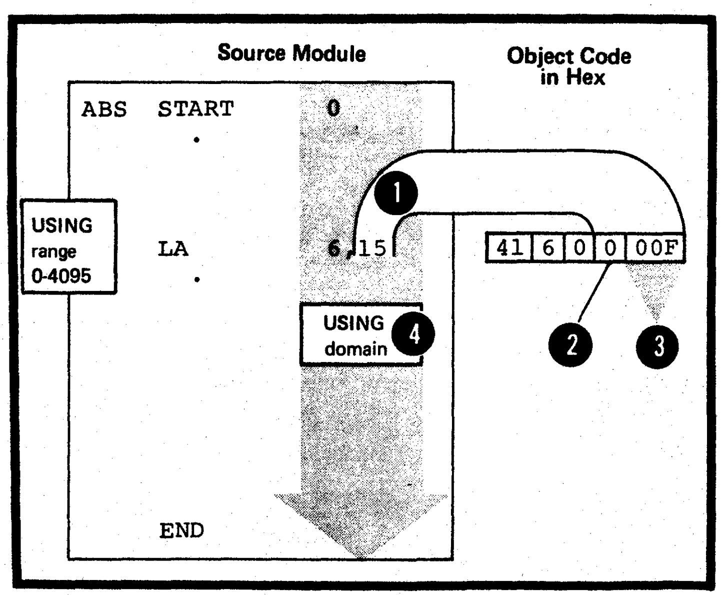


Two USING ranges overlap when the
base address of one USING instruction
lies within the range of another
1 USING instruction. When two ranges
overlap, the assembler computes
2 displacements from the base address
that gives the smallest displacement;
3 it uses the corresponding base
register when it assembles the
addresses within the range overlap.
This applies only to implicit
4 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
1 absolute implicit addresses in the
range 0 through 4,095 to their
explicit form. The assembler uses
2 register 0 as a base register.
3 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
4 domain for this automatic base
register assignment is the whole
of a source module.

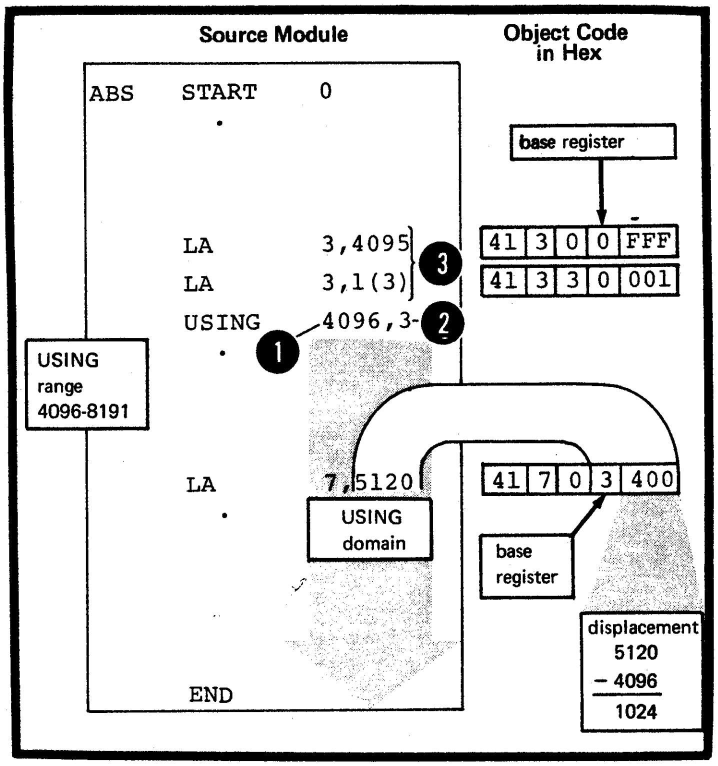


For absolute implicit addresses greater than 4095, a USING instruction must be specified according to the following:

1 • With a base address representing an absolute expression, and

2 • With a base register that has not been assigned by a USING instruction in which a relocatable base address is specified.

3 This base register must be loaded with the base address specified.



F1B - THE DROP INSTRUCTION

Purpose

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:

1. To free base registers for other programming purposes

2. 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 (as described above in F1A, Notes about the USING range).

Specifications

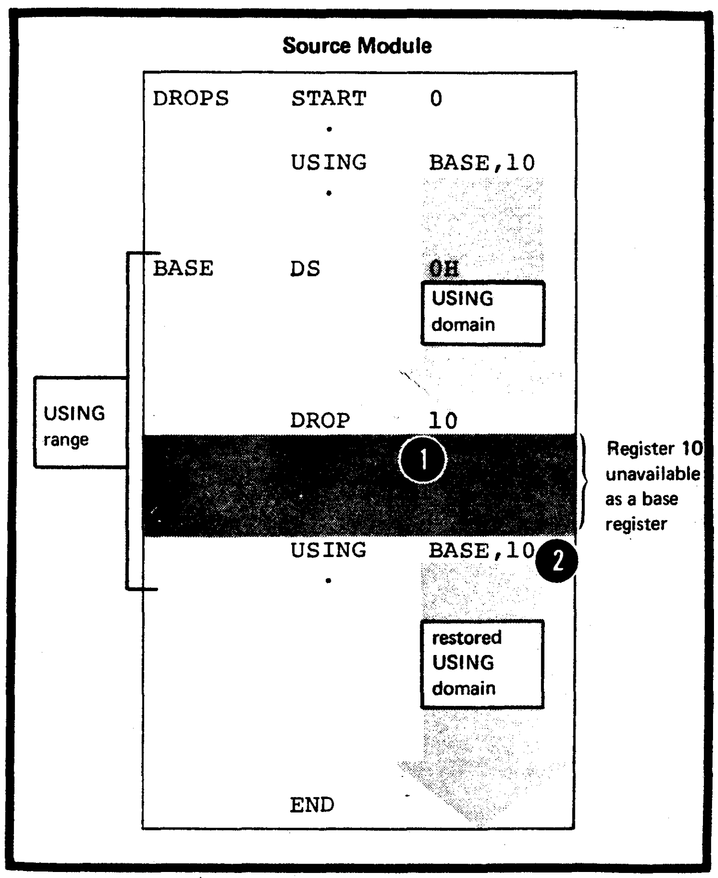
The DROP instruction must be coded as shown in the figure to the right.

Up to 16 operands can be specified. They must be absolute expressions whose values represent the general registers 0 through 15. A DROP instruction with a blank operand field causes all currently active base registers assigned by USING instructions to be dropped.

DROP

Name	Operation	Operand
Sequence symbol or blank	DROP	BASREG1 [,BASREG2] ... or blank

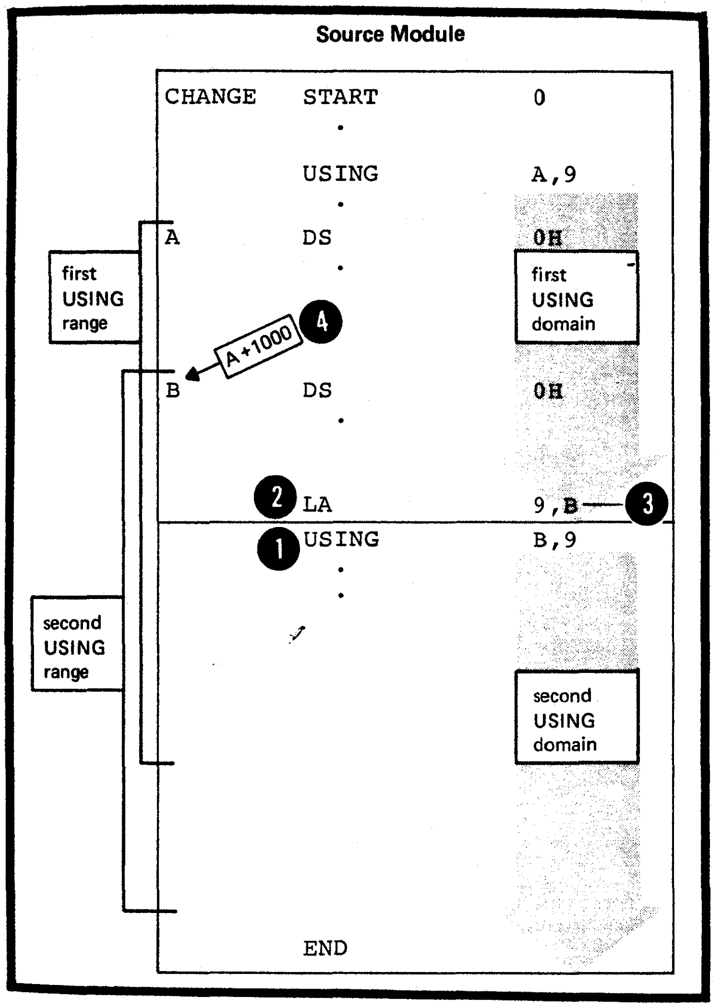
❶ After a DROP instruction, the assembler will not use the registers specified in a DROP instruction as base registers. A register made unavailable as a base register by ❷ a DROP instruction can be reassigned as a base register by a subsequent USING instruction.



A DROP instruction is not needed:

- If the base address is being
1 changed by a new USING instruction, and the same base register is assigned. However, the new base
2 address must be loaded into the base register. Note that the
3 implicit address "B" lies within the first USING domain, and that
4 the base address to which it refers lies within the first USING range.

- At the end of a source module.



F2 — Addressing Between Source Modules: Symbolic Linkage

This section describes symbolic linkage, that is, using symbols to communicate between different source modules that are separately assembled and then linked together by the linkage editor.

How to Establish Symbolic Linkage

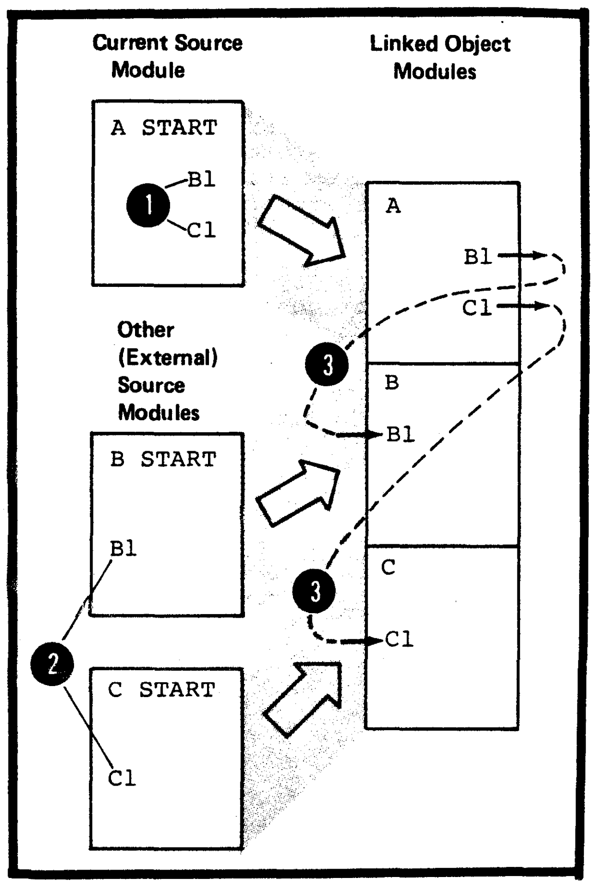
You must establish symbolic linkage between source modules so that you can refer or branch to symbolic locations defined in the control sections of external source modules. To establish symbolic linkage with an external source module you must do the following:

1. 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
(1) 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.

2. 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
(2) 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.

3. You must provide the A-type or Y-type address constants needed by the assembler to reserve storage for the addresses represented by the external symbols.

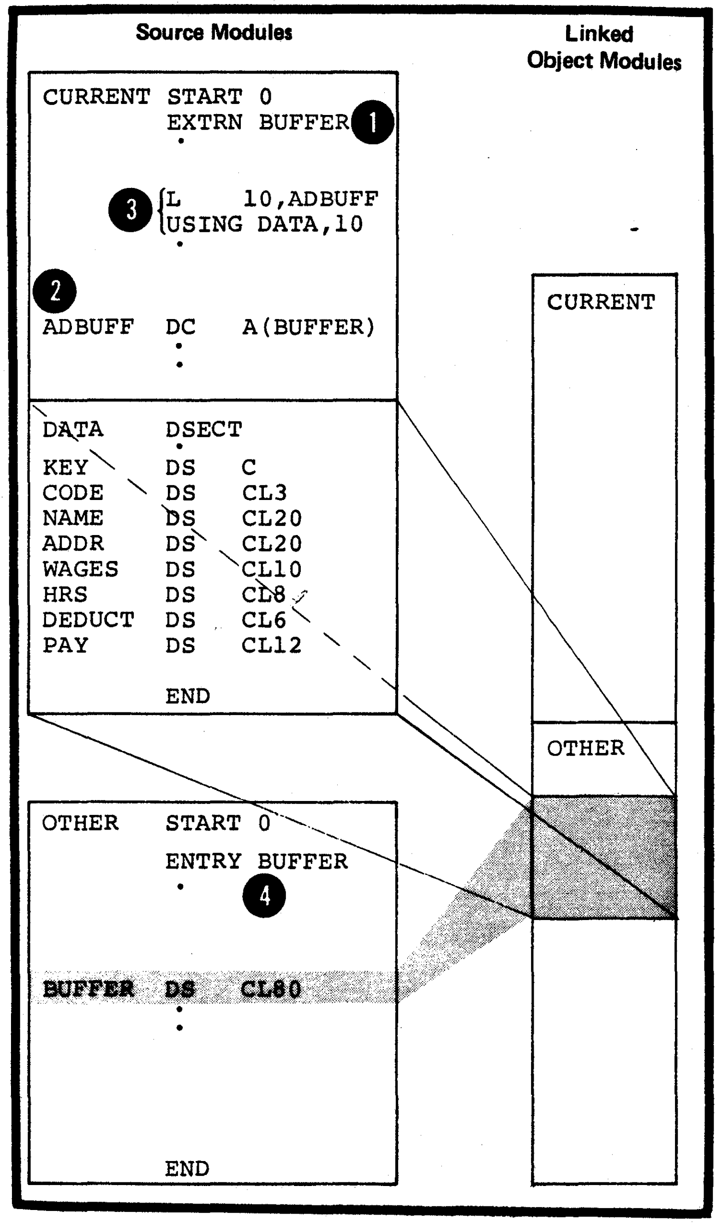
The assembler places information about entry and external symbols in the External Symbol Dictionary. The linkage editor uses this
(3) information to resolve the linkage addresses identified by the entry and external symbols.



TO REFER TO EXTERNAL DATA: You
should use the EXTRN instruction
1 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
2 symbol and load the address constant
specifying this symbol into a base
register. Then, you use this base
register when establishing the
3 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
4 as an entry symbol.



TO BRANCH TO AN EXTERNAL ADDRESS: You should use the V-type address (1) constant to identify the external symbol that represents the address in an external source module to which you wish to branch. For the specifications of the V-type address constant, see G3L.

For example, you can load into a (2) register the V-type address constant that identifies the external symbol. Using this register, you can then (3) 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 (4) identified as an entry symbol. If the symbol represents an address in the middle of a control section, you must, however, identify it as (5) 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:

1. You do not have to use an EXTRN instruction.

2. 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.

Source Modules

```
CURRENT START 0
          .
        L 3,EXTADR
    (3) BR 3
          .
(2)              (1)
EXTADR  DC V(OTHER)
          .

        END
```

```
(4)
OTHER START 0
      ENTRY SUBRTN (5)
          .
SUBRTN DS 0H
          .

      END
```

F2A - THE ENTRY INSTRUCTION

Purpose

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

Specifications

The format of the ENTRY instruction is shown in the figure to the right.

ENTRY

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

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

- They must be valid symbols.
- They must be defined in an (1) executable control section.
- They must not be defined in a (2) 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 (3) 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 (4) External Symbol Dictionary along with entries for external symbols, common control sections, and external control sections. The maximum number of External Symbol Dictionary entries for each source module is 399.

DOS The maximum number of external symbol dictionary entries (control sections and external symbols) allowed is 511. The maximum allowable number of entry symbols identified by the ENTRY instruction is 200.

(5) NOTE: A symbol identified in an ENTRY instruction counts towards this maximum, even though it may not be used in the name field of a statement in the source module nor constitute a valid entry point.

Source Module			Entry in External Symbol Dictionary (4) Symbol	Type Code
(3) FIRST	START	0	FIRST	SD
	•			
	ENTRY	SUBRTN,INVALID	SUBRTN	LD
	•		INVALID (5)	LD
(1) SUBRTN	DS	0H		
	•			
DUMMY	DSECT		DUMMY	none
	•			
(2) INVALID	DS	F	INVALID	-
	•			
	END			

F2B - THE EXTRN INSTRUCTION

Purpose

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.

Specifications

The format of the EXTRN instruction statement is shown in the figure to the right.

EXTRN

Name	Operation	Operand
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:

(1) • 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.

(2) • They must be used alone and cannot be paired when used in an expression (for pairing of terms see C6).

(3) 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. The maximum number of External Symbol Dictionary entries for each source module is 399.

DOS The maximum number of external symbol dictionary entries (control sections and external symbols) allowed is 511. The maximum allowable number of entry symbols identified by the ENTRY instruction is 200.

(4) NOTE: The symbol specified in a V-type address constant is implicitly identified as an external symbol and counts towards this maximum.

Source Modules			Entry in External Symbol Dictionary (3): Symbol	Type Code
CURRENT	START	0	CURRENT	SD
	EXTRN	OTHER (1)	OTHER	ER
	•			
	L	3,EXTAD		
	BR	3		
	•			
	L	4,ADSUBRT		
	BR	4		
	•			
EXTAD	DC	A(OTHER) (2)		
ADSUBRT	DC	V(SUBRTN) (4)	SUBRTN	ER
	•			
	END			
OTHER	START	0	OTHER	SD
	ENTRY	SUBRTN	SUBRTN	LD
	•			
SUBRTN	DS	0H		
	•			
	END			

Purpose

The WXTRN instruction 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; then the module is 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:

1. In a module that is linked and loaded along with the object module assembled from your source module or

2. In a module brought in from a library due to the presence of an EXTRN instruction in another module linked and loaded with yours.

Specifications

The format of the WXTRN instruction statement is shown in the figure to the right.

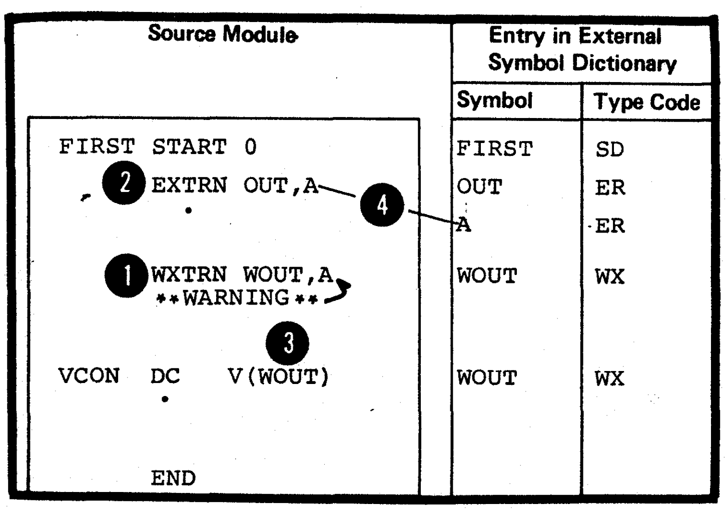
WXTRN

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

1 EXTERNAL SYMBOLS: The external symbols identified by a WXTRN instruction have the same properties as the external symbols identified 2 by the EXTRN instruction. However, the type code assigned to these external symbols differs.

3 NOTE: If a symbol, specified in a V-type address constant, is also identified by a WXTRN instruction in the same source module, 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 4 module, the first declaration takes precedence, and subsequent declarations are flagged with warning messages.



Section G: Symbol and Data Definition

This section describes the assembly time facilities which you can use to:

1. Assign values to symbols
2. Define constants and storage areas
3. Define channel command words.

By assigning an absolute value to a symbol and then using that symbol to represent, for example, a register or a length, you can code machine instructions entirely in symbolic form.

G1 – Establishing Symbolic Representation

You define symbols to be used as elements in your programs. This (1) symbolic representation is superior (2) to numeric representation because:

- You can give meaningful names to the elements;
- You can debug a program more easily, because the symbols are cross-referenced to where they are defined and used in your program. The cross-referenced statement numbers containing the symbols are printed in your assembly listing.
- You can maintain a program more easily, because you can change a symbolic value in one place and its value will be changed throughout a program.

(3) Some symbols represent absolute values, while others represent (4) relocatable address values. The relocatable addresses are of:

(5) instructions

(6) constants

(7) storage areas.

You can use these defined symbols in the operand fields of instruction statements to refer to the (8) instructions, constants, or areas represented by the symbol.

Source Module

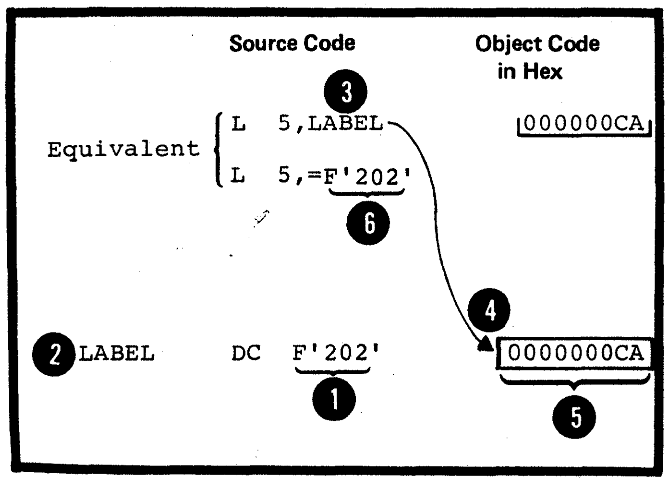
```
    FIRST      START
                 .
    DATAREG    EQU    10
    EIGHTY     EQU    80
    TWO40      EQU    240
(3) BASREG     EQU    12
(4) RELOC      EQU    *
                 .    (2)
               BALR   12,0   (1)
               USING  *,BASREG
                 .
               B      INSTR              (8)
                 .
(5) INSTR      MVC    TO,FROM            (8)
               L      DATAREG,DATACON    (8)
                 .
(6) DATACON    DC     F'3'
(7) TO         DS     CL(EIGHTY)
    FROM       DS     CL(TWO40)
                 .
                 .
               END
```

Assigning Values

You can create symbols and assign them absolute or relocatable values anywhere in a source module with an EQU instruction (see G2A). You can use these symbols instead of the numeric value they represent in the operand of an instruction.

Defining and Naming Data

DATA CONSTANTS: You can define a data constant at assembly time that will be used by the machine instructions in their operations at execution time. The three steps for creating a data constant and introducing it into your program in symbolic form are:



1 • define the data

2 • provide a label for the data

3 • refer to the data by its label.

4 The symbol used as a label represents the address of the constant; it is not to be confused with the

5 assembled object code of the actual constant.

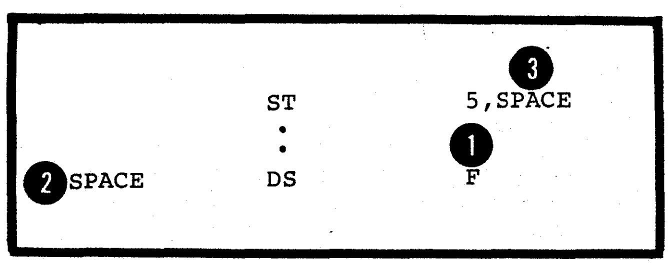
Defining data constants is discussed in G3.

LITERALS: You can also define data at its point of reference in the operand of a machine instruction

6 by specifying a literal.

Literal constants are discussed in G3C.

STORAGE AREAS: You must usually reserve space in virtual storage at assembly time for insertion and manipulation of data at execution time. The three steps for reserving virtual storage and using it in your program are:



1 • define the space

2 • provide a label for the space

3 • refer to the space by its label.

Defining storage areas is discussed in G3N.

CHANNEL COMMAND WORDS: When you define a channel command word at assembly time you create a command for an input or output operation to be performed at execution time. You should:

- define the channel command word
- provide a label for the word.

Channel command words are discussed in subsection G3C.

G2 – Defining Symbols

G2A -- THE EQU INSTRUCTION

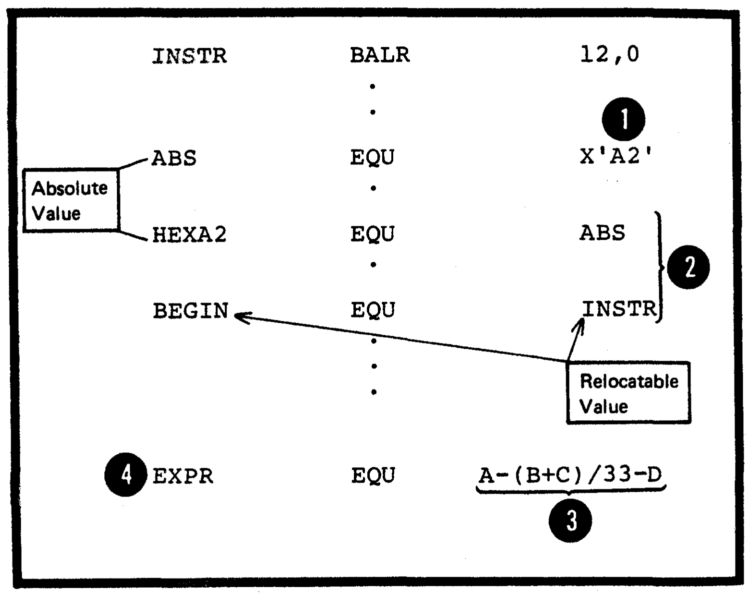
Purpose

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

❶ 1. To assign single absolute values to symbols

❷ 2. To assign the values of previously defined symbols or expressions to new symbols, thus allowing you to use different mnemonics for different purposes.

❸ 3. To compute expressions whose values are unknown at coding time or difficult to calculate. The value of the expression is then ❹ assigned to a symbol.



Specifications

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).

The format of the EQU instruction statement is given in the figure to the right.

DOS Only one operand (expression 1) is allowed.

EQU

Name	Operation	Operand
An ordinary symbol or a variable symbol	EQU	4 options: Expression 1 Expression 1, Expression 2 Expression 1, Expression 2, Expression 3 Expression 1,, Expression 3 OS only (Indicates the absence of Expression 2)

Expression 1 represents a value. It must always be specified and can have a relocatable or absolute value. The assembler carries this value as a signed four-byte (32-bit) number; all four bytes are printed in the program listings opposite the symbol.

OS only Expression 2 represents a length attribute. It is optional, but, if specified, it must have an absolute value in the range of 0 through 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 through 255.

Any symbols appearing in these three expressions 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 (described in C4C) thus assigned is as follows (see figure on following page):

1. If the leftmost term is a location counter reference (*), a self-defining term or a symbol length attribute
1 value reference, the length attribute value 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
2 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
3 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 one. However, the name of a CCW instruction has a length
4 attribute value of eight.

NOTE: The length attribute value assigned in cases 2-4 only
5 applies to the assembly-time value of the attribute. Its
6 value at pre-assembly time, during conditional assembly processing, is always 1.

Further, if expression 3 is omitted, the assembler assigns a type attribute value of "U" to the symbol in the name field.

Value assigned to symbol is:	Source Module				Length Attribute Value assigned to symbol in name field: At Assembly Time 5	At Pre-assembly Time 6
	SECTA	START	0			
		•				
	RR	LR	3,4			
	RX	A	3,FULL			
	SS	MVC	TO,FROM			
		•				
	FULL	DC	F'33'			
	AREA	DS	XL2000			
	TO	DS	CL240			
	FROM	DS	CL80			
		•				
	ADCONS	DC	AL1(A),AL2(B),AL3(C)			
		•				
	ADCCW	CCW	2,READER,X'48',80			
		•				
Absolute	A	EQU	X'FF'	1	1	1
Absolute	B	EQU	L'FROM		1	1
Relocatable	C	EQU	*+4		1	1
Absolute	D	EQU	A*10		1	1
Relocatable	E	EQU	FULL	2	4	1
Relocatable	F	EQU	AREA+1000		2000	1
Relocatable	G	EQU	TO		240	1
Absolute	H	EQU	FROM-TO		80	1
Relocatable	I	EQU	ADCONS		1	1
Relocatable	J	EQU	RR	3	2	1
Relocatable	K	EQU	RX		4	1
Relocatable	L	EQU	SS		6	1
Relocatable	M	EQU	SECTA		1	1
Relocatable	N	EQU	ADCCW	4	8	1

OS only **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. (1)

If expression 2 is a self-defining term, the assembler also assigns the length attribute value to the symbol at pre-assembly time (during conditional assembly processing). (2)

OS only **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. (3) Note that the type attribute value is the EBCDIC character equivalent of the value of expression 3. (4)

Value assigned	Source Module				Length Attribute Value assigned: At Assembly Time	Length Attribute Value assigned: At Pre-assembly Time (2)	Type Attribute Value assigned
	FIRST	START					
		.					
	AREA	DS	XL2000	Implicit Attribute Values	2000	2000	X
		.					
	SDT	EQU	X'FF'	Implicit Attribute Values	1	1	U
		.					
	ASTERISK	EQU	*	Implicit Attribute Values	1	1	U
Value of AREA	A	EQU	AREA,1000	(1)	1000	1000	U
255	B	EQU	SDT,4	(1)	4	4	U
Value of Location Counter at ASTERISK	C	EQU	ASTERISK,4	(1)	4	4	U
	D	EQU	AREA,,C'F'		2000	1	F (3)
	E	EQU	SDT,,C'N'		1	1	N (3)
	F	EQU	ASTERISK,,C'A'		1	1	A (3)
	G	EQU	AREA,1000,C'1'		1000	1000	1
	H	EQU	SDT,4,C'F'		4	4	F
	I	EQU	ASTERISK,4,C'A'		4	4	A
	J	EQU	AREA,100,198	(4)	100	100	F

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.

OS only 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

THE 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 pre-assembly time.

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

1. The symbol in the name field must be an ordinary symbol.

2. Expression 2 and Expression 3 must be single self-defining terms.

G3 – Defining Data

This section describes the DC, DS, and CCW instructions; these instructions 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.

G3A -- THE DC INSTRUCTION

Purpose

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.

TYPES OF CONSTANTS: The DC instruction can generate the following types of constants:

1. Binary constants -- to define bit patterns
2. Character constants -- to define character strings or messages
3. Hexadecimal constants -- to define large bit patterns
4. Fixed-Point constants -- for use by the fixed-point and other instructions of the standard set
5. Decimal constants -- for use by the decimal instructions
6. Floating-Point constants -- for use by the floating-point instruction set
7. Address constants -- to define addresses mainly for the use of the fixed-point and other instructions in the standard instruction set.

```
1   FLAG     DC   B'00010000'

2   CHAR     DC   C'STRING OF CHARACTERS'

3   PATTERN  DC   X'FF00FF00'

4            L    3,FCON
    FCON     DC   F'100'

5            AP   AREA,PCON
    PCON     DC   P'100'
    AREA     DS   P

6            LE   2,ECON
    ECON     DC   E'100.50'

7            L    5,ADCON
    ADCON    DC   A(SOMWHERE)
```

G3B -- GENERAL SPECIFICATIONS FOR CONSTANTS

DC

Name	Operation	Operand
Any Symbol or blank	DC	One or more Operands separated by commas (In the format described in the next figure)

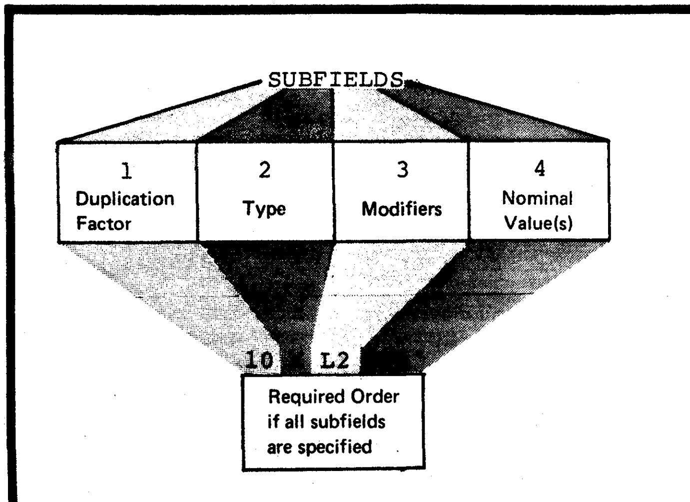
The general format of the DC instructions statements is shown in the figure to the right.

```
                      (1)
 FIRSTCON     DC     F'2',X'A1',C'HUM'

                     Logical Equivalent

(2)FIRSTCON   DC     F'2'
              DC     X'A1'
              DC     C'HUM'

              LA     3,FIRSTCON

              CLI    5,4(3)
                       (3)
```

The symbol in the name field represents the address of the first byte of the assembled constant. (1) If several operands are specified, the first constant defined is (2) addressable by the symbol in the name field. The other constants can be (3) reached by relative addressing.

Each operand in a DC instruction statement consists of four subfields. The format of a DC instruction operand is given in the figure to the right.

The first three subfields describe the constant, and the fourth subfield specifies the nominal value of the constant to be generated.



Rules for the DC Operand

(1) (2) 1. The type subfield and the nominal value must always be specified.

2. The duplication factor and modifier subfields are optional.

(3) 3. When multiple operands are specified, they can be of different types.

(4) 4. When multiple nominal values are specified in the fourth subfield, they must be separated by commas and be of the same type.

(5) 5. The descriptive subfields apply to all the nominal values.

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

```
MUST      DC   F'200'                    (1) type F, (2) nominal value 200

OPRNDS    DC   C'FIRST',H'99',FL3'101'   (3)

SEVERAL   DC   A(FIRST,SECOND,THIRD)
VALUES    DC   F'100,200,300'

          Logical Equivalent

SEVERAL   DC   A(FIRST)
          DC   A(SECOND)
          DC   A(THIRD)
VALUES    DC   F'100'
          DC   F'200'
          DC   F'300'
```

(4) Multiple nominal values not allowed for character constant

```
MIXED     DC   AL3(ONE,TWO),2F'1,2,3'    (5)

          Logical Equivalent

MIXED     DC   AL3(ONE)
          DC   AL3(TWO)
          DC   F'1'
          DC   F'2'
          DC   F'3'
          DC   F'1'
          DC   F'2'
          DC   F'3'
```

6. No blanks are allowed:

(1) a. Between subfields

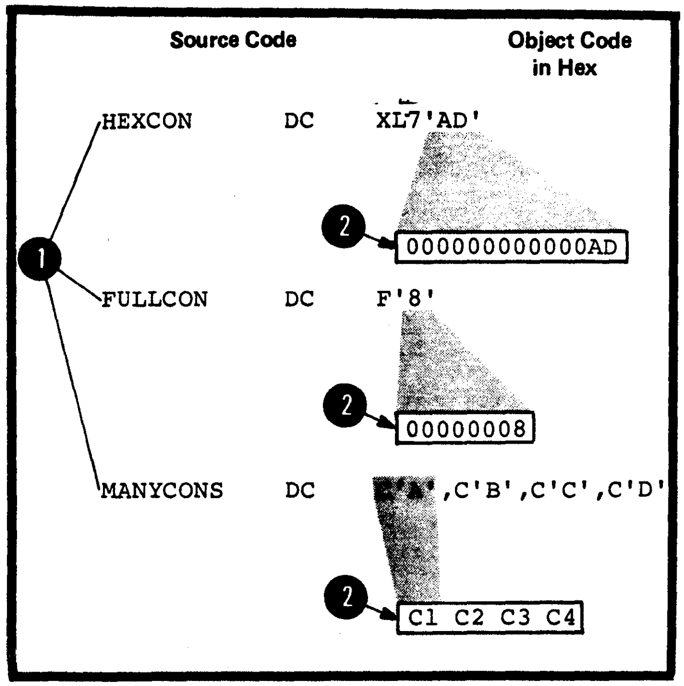
(2) b. Between multiple operands

(3) (4) c. 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.

```
BETWEEN   DC   10FL3'+456'               (1)

SEVERAL   DC   C'BOO HOO',F'95',H'2'     (2)

WITHIN    DC   C'MESSAGE HAS BLANKS'     (3)

          DC   XL(A+B-C'N O'+3)'F0'      (4)
```

Information about Constants

SYMBOLIC ADDRESSES OF CONSTANTS: Constants defined by the DC instruction are assembled into an object module at the location where 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 below under Alignment of Constants). ❶ The value of the symbol that names the DC instruction ❷ is the address of the leftmost byte (after alignment) of the first or only constant.



THE LENGTH ATTRIBUTE VALUE OF SYMBOLS NAMING CONSTANTS: The length attribute value assigned to the symbols in the name field of constants is equal to:

❶ The implicit length of the constant when no explicit length is specified in the operand of the constant, or

❷ The explicitly specified length 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.

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

[1] Depends on type

[2] Depends on whether or not an explicit length is specified in constant

ALIGNMENT OF CONSTANTS: The assembler aligns constants on different boundaries according to the following:

(1) On boundaries implicit to the type of constant, when no length specification is supplied.

(2) On byte boundaries 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. Note that the automatic alignment of constants and areas does not occur if the NOALIGN assembler option has been specified in the job control language which invoked the assembler.

NOTE: Alignment can be forced to any boundary by a preceding DS (or DC) instruction with a zero duplication factor (see G3N). This occurs when either the ALIGN or NOALIGN option is set.

Type of Constant	Implicit Boundary Alignment[1]	Examples	Boundary Alignment
B	byte		
C	byte		
X	byte		
H	halfword	DC H'25'	halfword
		DC HL3'25' (2)	byte
F	fullword	DC F'225'	fullword
		DC FL7'225' (2)	byte
P	byte	DC P'2934'	byte
Z	byte	DC Z'1235'	byte
		DC ZL2'1235' (2)	byte
E	fullword	DC E'1.25'	fullword
		DC EL5'1.25' (2)	byte
D	doubleword	DC 8D'95'	doubleword
		DC 8DL7'95' (2)	byte
L	doubleword	DC L'2.57E65'	doubleword
Y	halfword	DC Y(HERE)	halfword
A	fullword	DC AL3(THERE) (2)	byte
S	halfword		
V	fullword		
Q	fullword		

[1] Depends on type (1)

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:

1. By the explicit length specified in the second operand subfield, or

2. If no explicit length is specified, by the implicit length according to the type of constant defined (see Appendix VI).

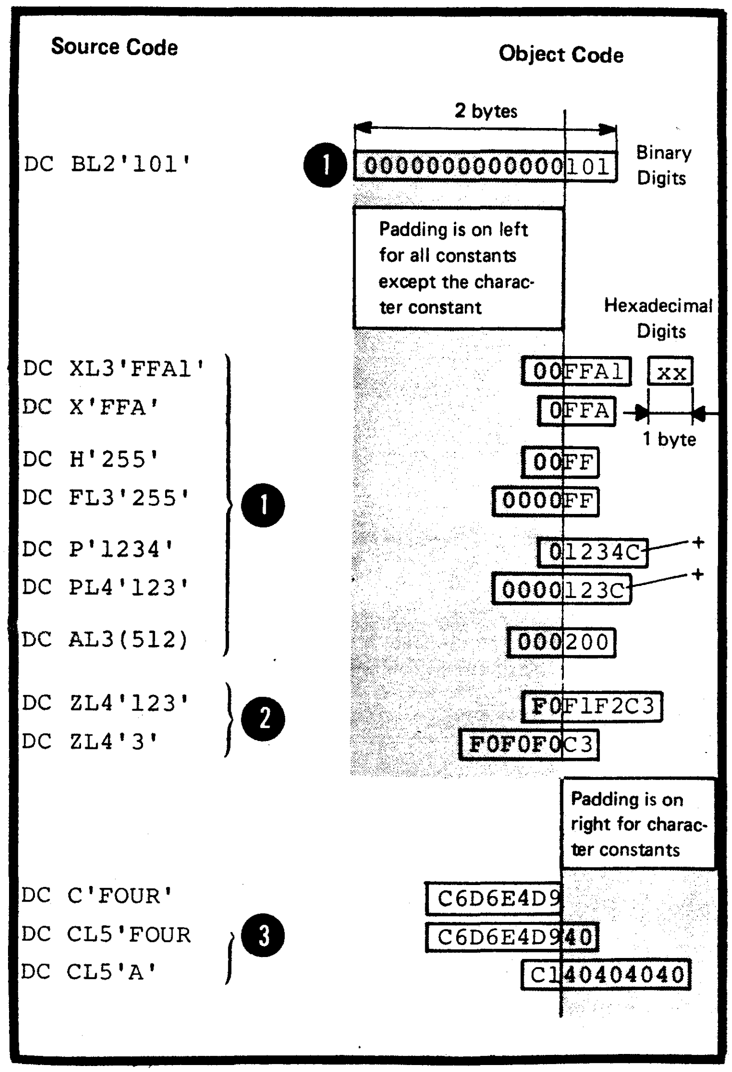
<u>PADDING:</u> If more space is available than is needed to accommodate the binary representation of the nominal value, the extra space is padded:

1 <u>With binary zeros</u> 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

2 <u>With EBCDIC zeros</u> on the left (X'F0') for the zoned decimal (Z) constants

3 <u>With EBCDIC blanks</u> on the right (X'40') for the character (C) constant

NOTE: Floating-point constants (E,D,L) are also padded on the right with zeros (see G3I).



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:

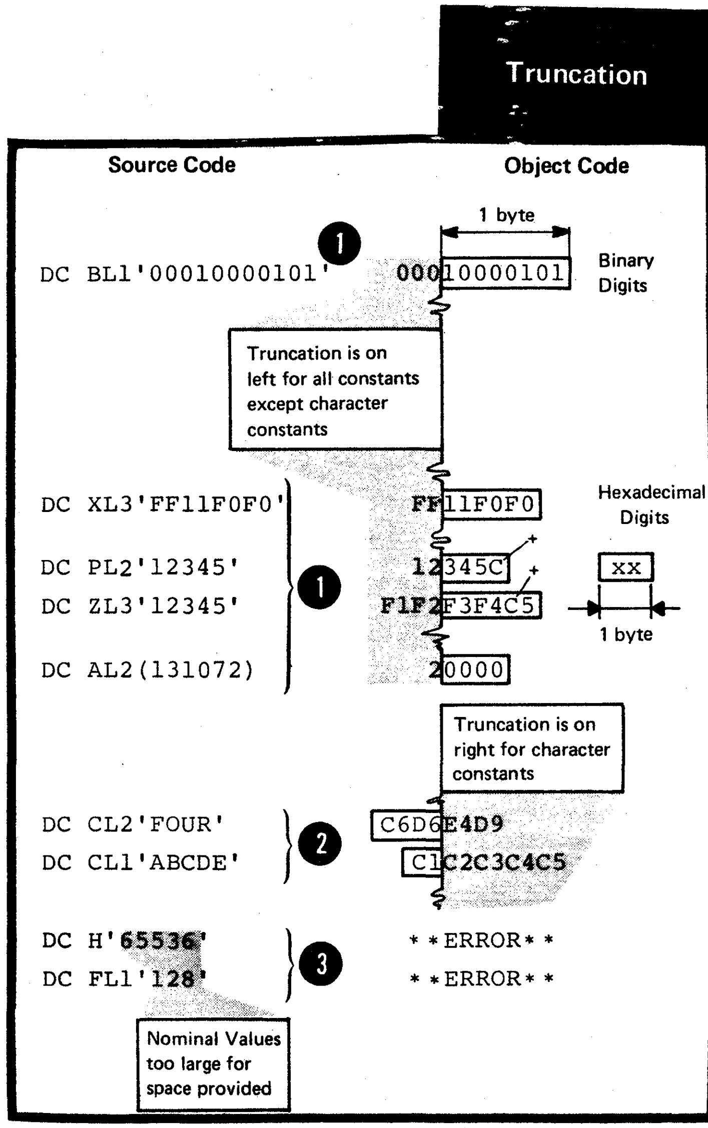
1. On the left for the binary (B), hexadecimal (X), decimal (P and Z), and address (A and Y) constants.

2. On the right for the character (C) constant.

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

NOTE: Floating-point constants (E,D,L) are not truncated; they are rounded (see G3I).

NOTE: The above rules for padding and truncation also apply when the bit-length specification is used (see below under Subfield 3: Modifiers).



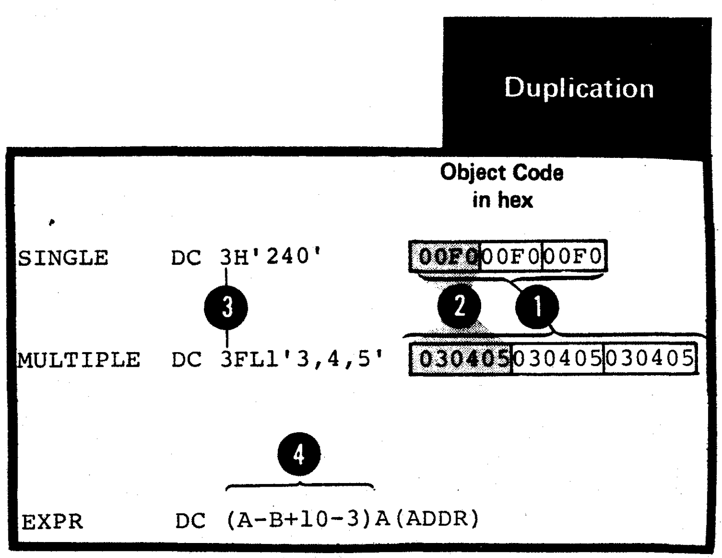
Subfield 1: Duplication Factor

The duplication factor, if specified, causes the nominal value or multiple nominal values specified in a
(1) constant to be generated the number of times indicated by the factor.
(2) It is applied after the nominal value or values are assembled into the constant.

The factor can be specified by an
(3) unsigned decimal self-defining term or by an absolute expression enclosed
(4) in parentheses.

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

Any symbols used in the expression must be previously defined.



NOTES:

1. The value of a location counter reference in a duplication factor is the value before any align to boundaries, according to the type of constant specified.

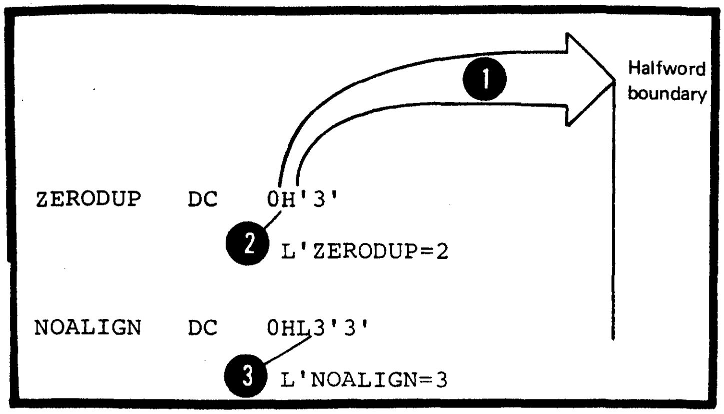
2. A duplication factor of zero is permitted with the following results:

a. No value is assembled.

1 b. Alignment is forced according to the type of constant specified, if no length attribute is present (see above under Alignment of Constants).

c. The length attribute of the symbol naming the constant is established according to the 2 implicitly or 3 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 G3J).



Subfield 2: Type

The type subfield must be specified. It defines the type of constant to be generated and is specified by a single letter code as in the figure to the right.

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 1 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 above, under Padding and Truncation of Constants).

Type

Code	Type of Constant	Machine Format
C	Character	8-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

OS only

		1 Object Code in hex
Examples:	DC P'+234'	234C
	DC C'ABC'	C1C2C3
	DC X'F0'	F0
	DC H'2'	0002

Subfield 3: Modifiers

The three modifiers that can be specified to describe a constant are:

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

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

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.

4. If multiple modifiers are used, they must appear in the sequence: length, scale, exponent.

```
                    (1)
LENGTH    DC   XL10'FF'
SCALE     DC   FS8'35.92'
                    (2)
EXPON     DC   EE3'3.414'
                    (3)
ALL3      DC   DL7S3E50'2.7182'
                    (4)
```

Length

LENGTH MODIFIER: The length modifier indicates the number of bytes of storage into which the constant is to be assembled. It is written as Ln, where n is either of the following:

1. A decimal self-defining term

2. An absolute expression enclosed in parentheses. It must have a positive value and any symbols it
3. contains must be previously defined.

DOS/VS NOTE: Location counter reference must not be used in the modifier subfield.

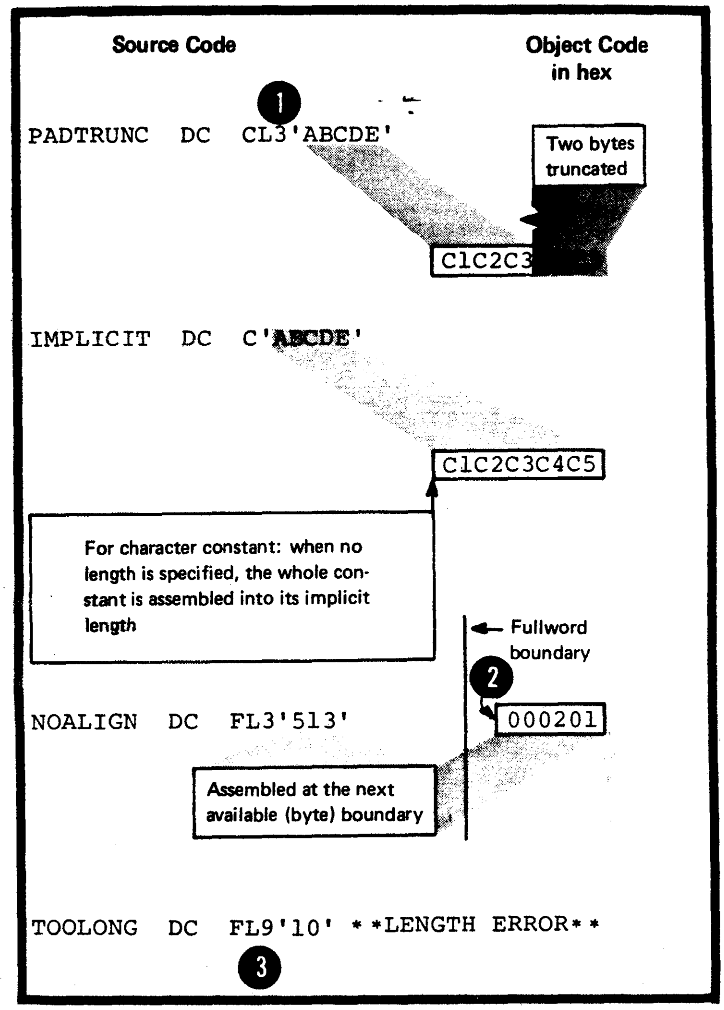
```
                 (1)
DECSDT    DC   FL3'9999'
                          (2)
EXPR      DC   XL(SYMBOL-ADDR+3*B-L'D)'F7A'
                          (3)
```

When the length modifier is specified:

1 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 above under Padding and Truncation of Constants).

2 No boundary alignment, according to constant type, is provided (see above under Alignment of Constants).

3 Its value must not exceed the maximum length allowed for the various types of constant defined. (For the allowable range of length modifiers, see the specifications for the individual constants and areas from G3D through G3N.)



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 of the following:

A decimal self-defining term

An absolute expression enclosed in parentheses. It must have a positive value and any symbols it contains must be previously defined.

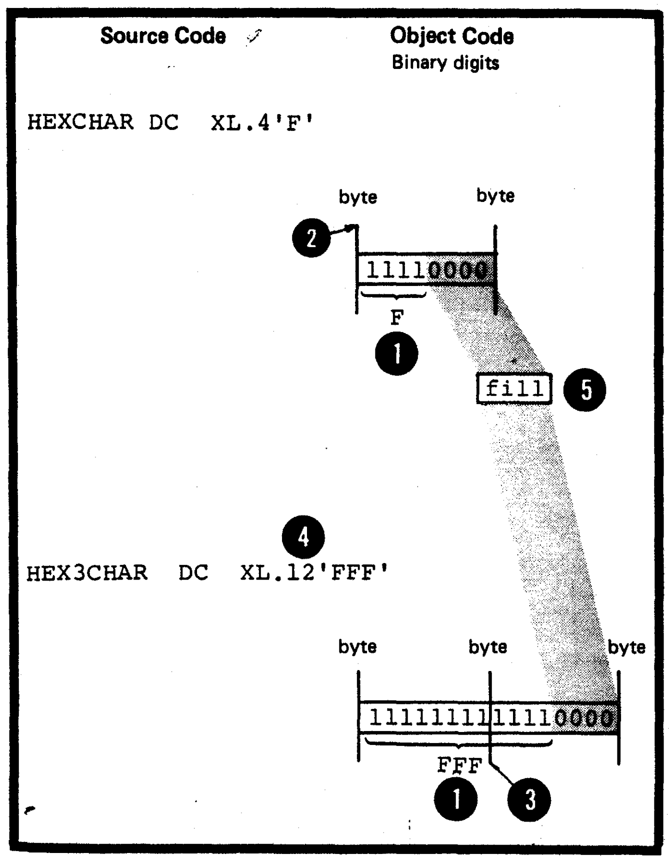
The value of n must lie between 1 and the number of bits (a multiple of 8) that are required to make up the maximum number of bytes allowed in the type of constant being defined. The bit length-specification cannot be used with the S, V, and Q-type constants.

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

1 1. The bit-length specification allocates a field into which a constant is to be assembled.

2 The field starts at a byte boundary,
and can run over one or more byte
3 boundaries, if the bit-length
4 specified is greater than 8.

If the field does not end at a byte boundary, if the bit-length specified
5 is not a multiple of 8, the remainder of the last byte is filled with zeros.



1 2. The nominal value of the constant is assembled into the field:

2 Starting at the high order end for the C, E, D, and L type constants.

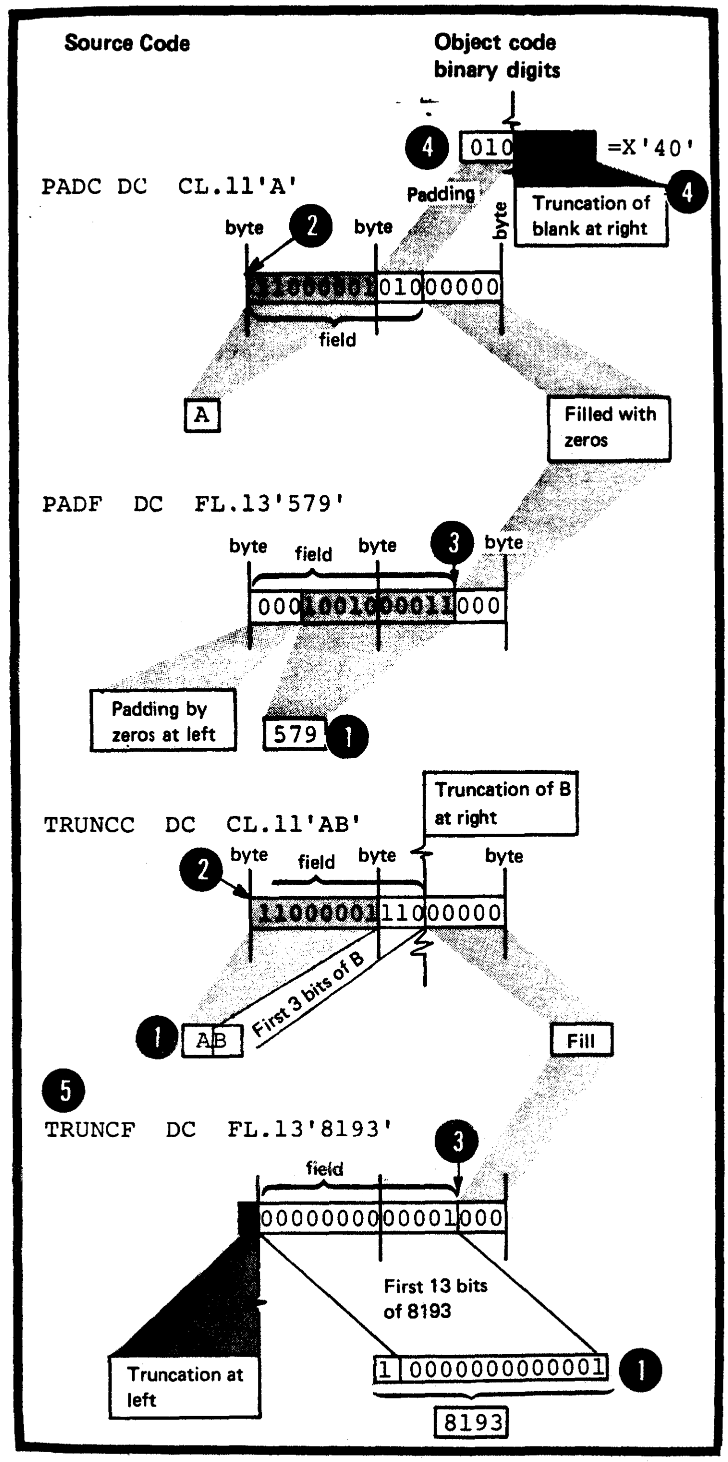
3 Starting at the low order end for the remaining types of constants that allow bit-length specification.

The nominal value is padded or truncated to fit the field (see above under Padding or Truncation of Constants).

4 Padding of character constants is 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

5 reference to TRUNCF would address the entire two bytes that are assembled.



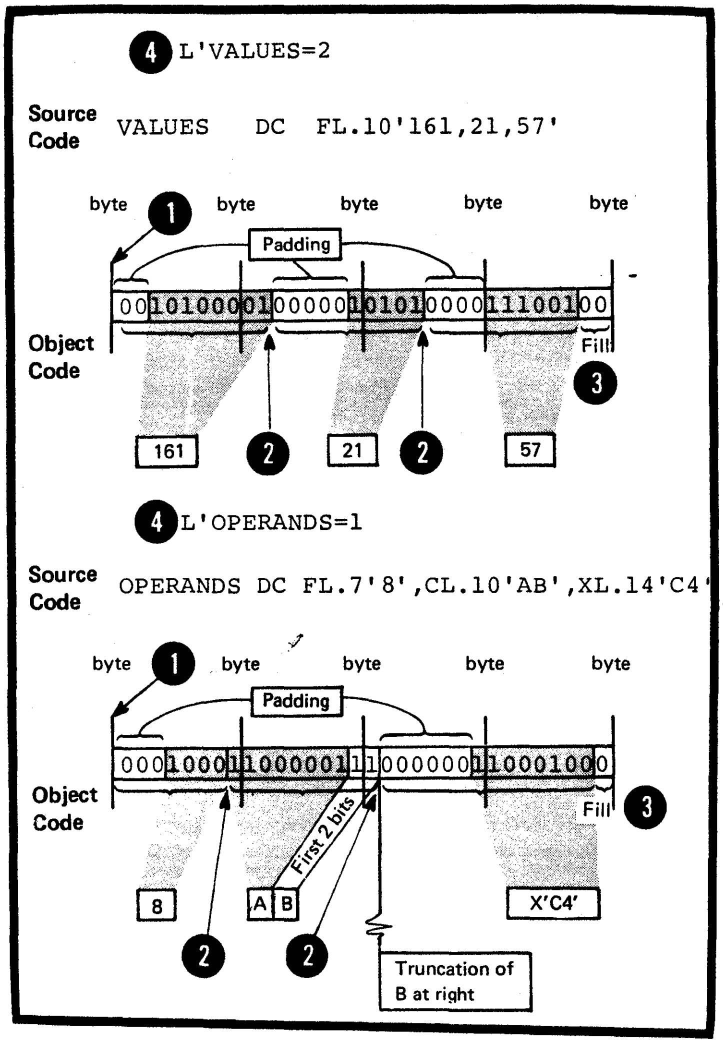
When more than one operand is specified in a DC instruction or more than one nominal value in a DC operand, the above rules about bit-length specifications also apply, except:

1 1. The first field allocated starts at a byte boundary, but the succeeding fields start at the next 2 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 3 zeros.

NOTE: If duplication is specified, filling with zeros occurs once at the end of all the fields occupied by the duplicated constants.

4 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.



STORAGE REQUIREMENT FOR CONSTANTS: The total amount of storage required to assemble a DC instruction is the sum of:

(1) 1. The requirements for the individual DC operands specified in the instruction.

The requirement of a DC operand is the product of:

(2) a. The length (implicit or explicit),

(3) b. The number of nominal values, and

(4) c. The duplication factor, if specified.

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

```
SPACE  DC  10H'3,4,5',10FL3'6,7,8'
```

Space for		Storage Requirements							
(1)	OPERAND 1	2	x	3	x	10	=	60
(1)	OPERAND 2	3	x	3	x	10	=	90
(5)	ALIGNMENT	(2)		(3)		(4)	=	0

Second operand not aligned due to presence of length specification

TOTAL 150 Bytes

```
ALIGN DC  C'ABC',F'9,10,11'
```

		(2)		(3)	(4)		
(1)	OPERAND 1	3	x	1	(x 1)	=	3
(1)	OPERAND 2	4	x	3	(x 1)	=	12
(5)	ALIGNMENT						0-3

First operand can end on any byte boundary

TOTAL 15-18 Bytes

SCALE MODIFIER: The scale modifier specifies the amount of internal scaling that is desired:

Binary digits for fixed-point (H,F) constants

Hexadecimal digits for floating-point (E,D,L) constants

It can only be used with the above types of constant.

The scale modifier is written as Sn, where n is either:

(1) A decimal self-defining term or

(2) An absolute expression enclosed in parentheses.

DOS Any symbols used in the expression must be previously defined.

Both types of specification can be preceded by a sign; if no sign is present, a plus sign is assumed.

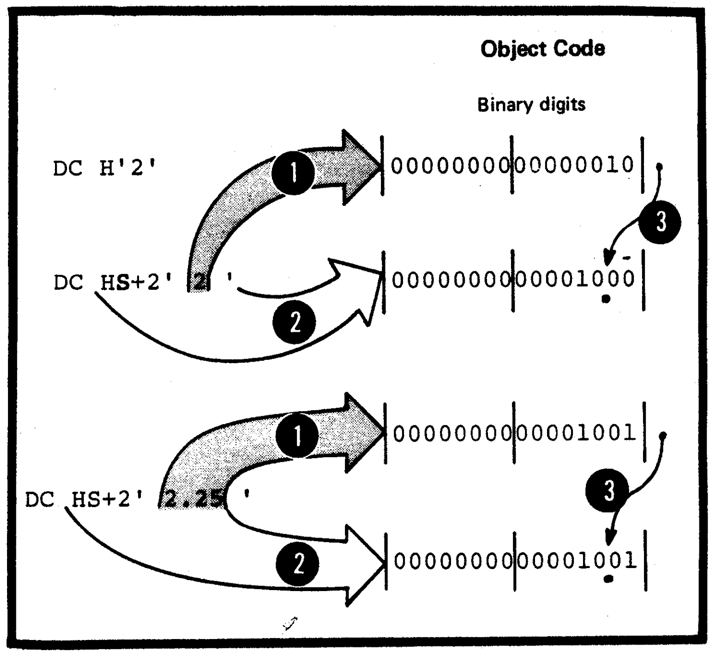
Scale

Examples:

```
DC  HS-132'5.55'      (1)
DC  HS3'2.25'         (2)
DC  FS(A+B-C*3)'2.3'
```

Examples	Allowable Range for Scale Modifier	
DC HS-132'5.55' / DC HS3'2.25' / DC FS(A+B-C*3)'2.3'	Fixed-point Constants (H,F)	− 187 through +346
	Floating-point Constants	
DC ES12'19.3'	(E,D)	0 through 14
DC LS22'3.414'	(L)	0 through 28

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 ❶ converted to its binary representation, but before it is ❷ assembled in its final "scaled" form. Scaling causes the ❸ binary point to move from its assumed fixed position at the right of the rightmost bit position.

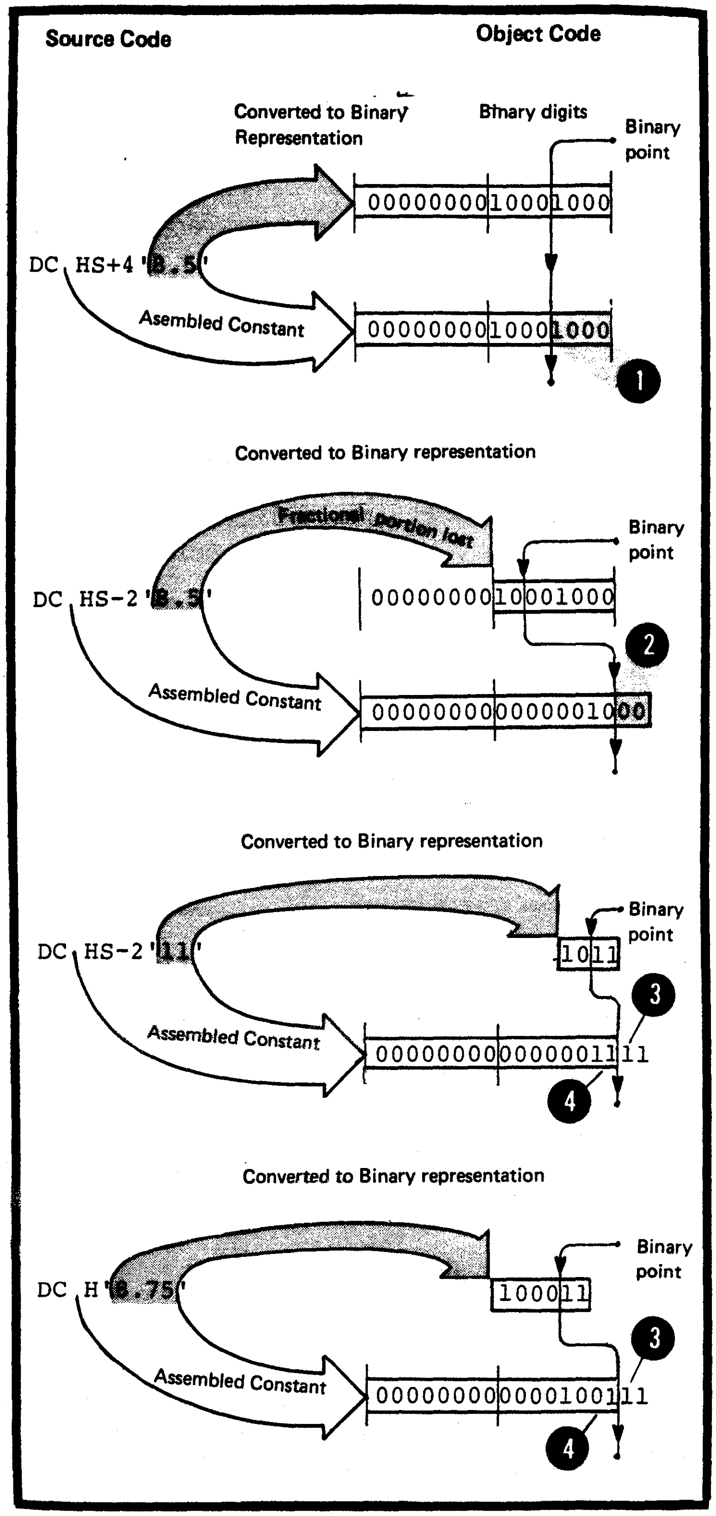


NOTES:

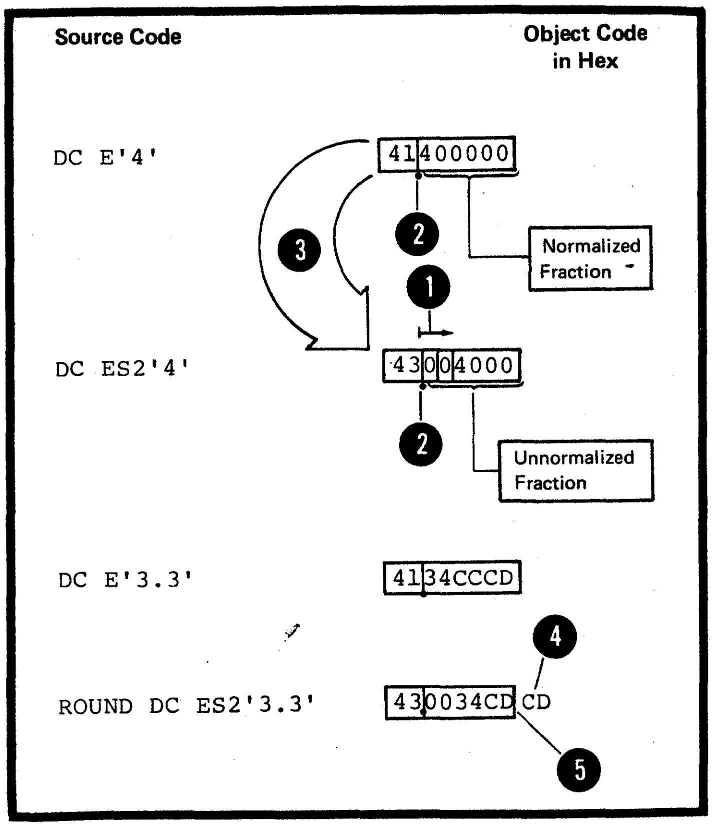
1 1. When the scale modifier has a positive value, it indicates the number of binary positions to be occupied by the fractional portion of the binary number.

2 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 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
4 rightmost position saved.



SCALE MODIFIER FOR FLOATING-POINT CONSTANTS: The scale modifier for floating-point constants must have a positive value. It specifies
1 the number of hexadecimal positions that the fractional portion of the binary representation of a floating-point constant is to be shifted
2 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
3 the characteristic portion of the constant is adjusted upward accordingly. When hexadecimal
4 positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost
5 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,F) and floating-point (E,D,L) constants. The exponent modifier is written as En, where n can be either of the following:

1 A decimal self-defining term.

2 An absolute expression enclosed in parentheses.

DOS **Any symbols used in the expression must be previously defined.**

3 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 through +75.

Exponent

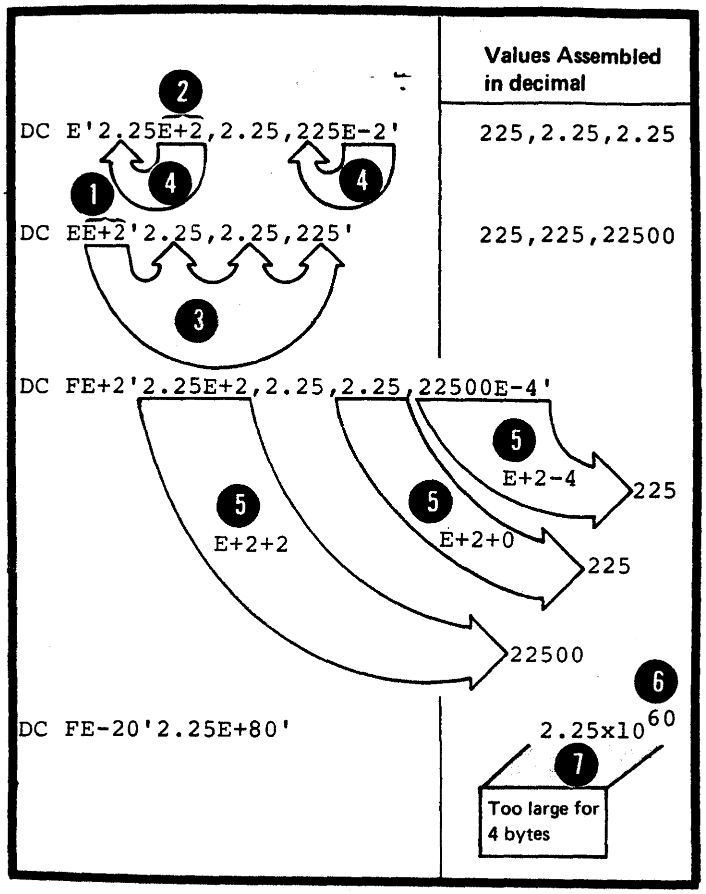
Source Code	Decimal Value before conversion to binary form	Object Code Binary digits
DC H'4'	4	00000000 00000100
(1) DC HE2'4'	400	00000001 10010000
(2) DC FE(A-B*3)'4'	-	
(3) DC HE-2'400'	4	00000000 00000100

NOTES:

1 1. The exponent modifier is not to be confused with the exponent that can be specified in the nominal
2 value subfield of fixed-point and floating-point constants (see sections G3G and G3I).

3 The exponent modifier affects each nominal value specified in the operand, whereas the exponent written as part of the nominal value subfield
4 only affects the nominal value it follows. If both types of exponent specification are present in a DC operand, their values are
5 algebraically added together before the nominal value is converted to binary form. However, this sum must lie within the permissible
6 range -85 through +75.

7 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.



Nom. Value

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 nominal values are described in the figure to the right.

DOS Only one nominal value is allowed in binary (B) and hexadecimal (X) constants.

How nominal values are specified and interpreted by the assembler is explained in the subsections that describe each individual constant, beginning at G3D.

Constant Type	Formats of Nominal Value Subfields: Single Nominal Values	Multiple Nominal Values
C	'Value'	Not allowed
B, X, H, F, P, Z, E, D, L	'Value'	'Value, value,......val ue,' (multiple values must be separated by commas)
A, Y, S, Q, V (Address Constants)	(Value)	(Value, value,.......value)

G3C -- LITERAL CONSTANTS

Purpose

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 C5.

Specifications

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 (as described in G3B above) also apply to the subfield of a literal constant. Moreover, the rules that apply to the individual types of constants, as described in G3D through G3M, apply to literal constants.

However, literal constants differ from DC operands in the following ways:

1 • Literals must be preceded by an equal sign.

2 • Multiple operands are not allowed.

3 • The duplication factor must not be zero.

DOS • Q-type and S-type address constants are not allowed.

```
      1
L   3,=F'32'

LM 3,5,=A(BASE,BASE+4096,BASE+8192)

                  3
MVC   FIELD(24),=6CL4'CANT'
```

2 Multiple Nominal Values are allowed

G3D -- BINARY CONSTANT (B)

Purpose

The binary constant allows you to specify the precise bit pattern you want assembled into storage.

Specifications

The constants of the subfields defining a binary constant are described in the figure below.

1 NOTE: Each binary constant is assembled into the integral number of bytes required to contain the bits specified.

Subfield	Binary Constants		
	3. Constant Type		
	Binary (B)		
1. Duplication Factor allowed	Yes		

2. Modifiers
Implicit Length: (Length Modifier not present)
	As needed
	`B DC B'10101111'`
	`C DC B'101'`
		`L'B = 1` 1
		`L'C = 1`	

Alignment: (Length Modifier not present)	Byte		
Range for Length:	1 through 256 (byte length) .1 through .2048 (bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed		
4. Nominal Value Represented by:	Binary digits (0 or 1)		
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of Values per Operand:	Multiple Only one DOS		
Padding:	With zeros at left		
Truncation of Assembled Value:	At left		

G3E -- CHARACTER CONSTANT (C)

Purpose

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.

Specifications

The contents of the subfields defining a character constant are described in the figure on the opposite page.

1 Each character specified in the nominal value subfield is assembled into one byte.

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

NOTE: When apostrophes or ampersands are to be included in the assembled constant, double apostrophes or double ampersands must be specified. They are assembled as single
3 apostrophes and ampersands.

Subfield	Character Constants		
	3. Constant Type		
	Character (C)		
1. Duplication Factor allowed	Yes		
2. Modifiers Implicit Length: (Length Modifier not present)	As needed C DC C'LENGTH' ❶	L'C = 6	
Alignment: (Length Modifier not present)	Byte		
Range for length:	1 through 256 (byte length) .1 through .2048 (bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed		
4. Nominal Value Represented by:	Characters (All 256 8-bit combinations)	DC C'A''B' Assembled A'B ❸ A&B DC C'A&&B'	Object Code (hex). C1 7D C2 C1 50 C2
Enclosed by:	Apostrophes		
Exponent allowed:	No		
Number of values per Operand:	One	DC C'A,B' Assembled A,B ❷	C1 6B C2
Padding:	With blanks at right (X'40')		
Truncation of Assembled value:	At right		

G3F -- HEXADECIMAL CONSTANT (X)

Purpose

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.

Specifications

The contents of the subfields defining a hexadecimal constant are described in the figure on the opposite page.

1 Each hexadecimal digit specified in the nominal value subfield is assembled into four bits (their binary patterns
2 can be found in C4E). 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
3 to an odd number of digits).

X

Subfield	Hexadecimal Constants		
	3. Constant Type		
	Hexadecimal (X)		
1. Duplication Factor allowed	Yes		

2. Modifiers
Implicit Length: (Length Modifier not present)
	As needed
	`X DC X'FF00A2'`
	`Y DC X'F00A2'`
		`L'X = 3`
		`L'Y = 3` ❷	

Alignment: (Length Modifier not present)	Byte		
Range for Length:	1 through 256 (byte length) .1 through .2048 (bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed	❶	
4. Nominal Value Represented by:	Hexadecimal digits (0 through 9 and A through F)	`DC X'1F'` `DC X'91F'`	Object Code (hex) 0001 1111 0000 1001 0001 1111 ← 1 byte →
Enclosed by:	Apostrophes	❸	
Exponent allowed:	No		
Number of Values per Operand:	Multiple (Only one DOS)		
Padding:	With zeros at left		
Truncation of Assembled value:	At left		

G3G -- FIXED-POINT CONSTANTS (H AND F)

Purpose

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 standard 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 NOALGN option has been specified; see D2). You can perform algebraic functions using this type of constant because they can have positive or negative values.

Specifications

The contents of the subfields defining fixed-point constants are described in the figure on the opposite page.

(1) The nominal value can be a signed (plus is assumed if the
number is unsigned) integer, fraction, or mixed number (2)
(3) followed by an exponent (positive or negative). The
exponent must lie within the permissible range. If an
(4) exponent modifier (see G3B) is also specified, the algebraic
sum of the exponent and the exponent modifier must lie (5)
within the permissible range.

Subfield	Fixed-Point Constants 3. Constant Type		
	Fullword(F)	Halfword (H)	
1. Duplication Factor Allowed	Yes	Yes	
2. Modifiers Implicit Length: (Length Modifier not present)	4 bytes	2 bytes	
Alignment: (Length Modifer not present)	Full word	Half word	
Range for Length:	1 through 8 (byte length) .1 through .64 (bit length)	1 through 8 (byte length) .1 through .64 (bit length)	
Range for Scale:	− 187 through + 346	− 187 through + 346	
Range for Exponent:	− 85 through + 75 ❹	− 85 through + 75	DC HE+90'2E-88' value = $2x10^2$ ❺
4. Nominal Value Represented by:	Decimal digits (0 through 9) DC F'-200' ❶ DC FS4'2.25' ❷	Decimal digits (0 through 9) DC H'+200' DC HS4'.25'	
Enclosed by:	Apostrophes	Apostrophes	
Exponent allowed:	Yes DC F'2E6' ❸	Yes DC H '2E-6'	
Number of Values per Operand:	Multiple	Multiple	
Padding:	With zeros at left	With zeros at left	
Truncation of Assembled value:	Not allowed (error message issued)	Not allowed	

Some examples of the range of values that can be assembled into fixed-point constants are given in the figure to the right.

(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.

Length (1)	Range of Values that can be Assembled
8	-2^{63} through $2^{63}-1$
4	-2^{31} " $2^{31}-1$
2	-2^{15} " $2^{15}-1$
1	-2^{7} " $2^{7}-1$

A fixed-point constant is assembled as follows:

1. The specified number, multiplied by any exponents, is converted to a binary number.

2. Scaling (see G3E) 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 2's complement form.

5. Duplication is applied after the constant has been assembled.

G3H -- DECIMAL CONSTANTS (P AND Z)

Purpose

The decimal constants allow you to introduce data that is 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 instruction set. The zoned decimal constants (Z-type) are in the form (EBCDIC representation) that you can use as a print image (except the digits in the rightmost byte).

Specifications

The contents of the subfields defining decimal constants are described in the figure on the opposite page.

The nominal value can be a signed (plus is assumed if the number is unsigned) decimal number. A decimal point can be written anywhere in the number, but it does not affect the assembly of the constant in any way. The specified (1) digits are assumed to constitute an integer. Decimal constants are assembled as follows:

(2) PACKED DECIMAL CONSTANTS: Each digit is converted into its 4-bit binary equivalent. The sign indicator (3) is assembled into the rightmost four bits of the constant.

(4) ZONED DECIMAL CONSTANTS: Each digit is converted into its 8-bit EBCDIC representation. The sign indicator (5) replaces the first four bits of the low-order byte of the constant.

Subfield	Decimal Constants 3. Constant Type		
	Packed (P)	Zoned (Z)	
1. Duplication Factor Allowed	Yes	Yes	

2. Modifiers
Implicit Length: (Length Modifier not present)
	As needed
	`P DC P'+593'`
	`L'P = 2`
		As needed
		`Z DC Z'-593'`
		`L'Z = 3`	

Alignment: (Length Modifer not present)	Byte	Byte	
Range for Length:	1 through 16 (byte length) .1 through .128 (bit length)	1 through 16 (byte length) .1 through .128 (bit length)	
Range for Scale:	Not allowed	Not allowed	
Range for Exponent:	Not allowed	Not allowed	

4. Nominal Value
Represented by:
	Decimal digits (0 through 9)
	`DC P'+555'`
	(2) → [5 | 5 | 5 | C] (3)
		Decimal digits (0 through 9)
		`DC Z'-555'`
		(4) → [F | 5 | F | 5 | D | 5] (5)
			`DC P'5.5'`
			[0 | 5 | 5 | C] (1)
			`DC P'55'`

Enclosed by:	Apostrophes	Apostrophes	
Exponent allowed :	No	No	
Number of Values per Operand:	Multiple	Multiple	
Padding:	With Binary zeros at left	With EBCDIC zeros `(X'F0')` at left	
Truncation of Assembled value:	At left	At left	

The range of values that can be assembled into a decimal constant is shown in the figure to the right.

Type of Decimal Constant	Range of Values that can be Specified
PACKED	$10^{31}-1$ through -10^{31}
ZONED	$10^{16}-1$ through -10^{16}

G3I -- FLOATING-POINT CONSTANTS (E, D, and L)

Purpose

Floating-point constants allow you to introduce data that is in a form suitable for the operations of the floating-point feature instruction set. These constants have the following advantages over fixed-point constants.

1. 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.

2. You can specify both much larger and much smaller values.

3. You retain greater processing precision, that is, your values are carried in more significant figures.

Specifications

The contents of the subfields defining floating-point constants are described in the figure on the opposite page.

❶ The nominal value can be a signed (plus is assumed if the number is unsigned) integer, fraction, or mixed number ❷ followed by an exponent (positive or negative). The ❸ exponent must lie within the permissible range. If an exponent modifier (see G3E under Modifiers) is also specified, the algebraic sum of the exponent and the exponent modifier must lie within the permissible range.

Subfield	Floating Point Constants 3. Constant Type		
	SHORT (E)	LONG (D)	EXTENDED (L)
1. Duplication Factor Allowed	Yes	Yes	Yes
2. Modifiers Implicit Length: (Length Modifier Not Precent)	4 Bytes	8 Bytes	I6 Bytes
Alignment: (Length Modifier Not Present)	Full Word	Double Word	Double Word
Range for Length:	1 through 8 (byte length) .1 through .64 (bit length)	1 through 8 (byte length) .1 through .64 (bit length)	1 through 16 (byte length) .1 through .128 (bit length)
Range for Scale:	0 through 14	0 through 14	0 through 28
Range for Exponent:	– 85 through + 75	– 85 through + 75	– 85 through + 75
4. Nominal Value Represented by:	Decimal Digits (0 through 9) (1) `DC E'+525'` `DC E'5.25'` (2)	Decimal Digits (0 through 9) `DC D'-525'` `DC D'+.001'` (2)	Decimal Digits (0 through 9) `DC L'525'` `DC L'3.414'` (2)
Enclosed by:	Apostrophes	Apostrophes	Apostrophes
Exponent Allowed:	Yes `DC E'1E+60'` (3)	Yes `DC D'-2.5E10'` (3)	Yes (3) `DC L'3.712E-3'`
Number of Values per Operand:	Multiple	Multiple	Multiple
Padding:	With hexadecimal zeros at right	With hexadecimal zeros at right	With hexadecimal zeros at right
Truncation of Assembled Value:	Not applicable (Values are rounded)	Not Applicable (Values are Rounded)	Not applicable (Values are Rounded)

The range of values that can be assembled into floating-point constants is given in the figure to the right.

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

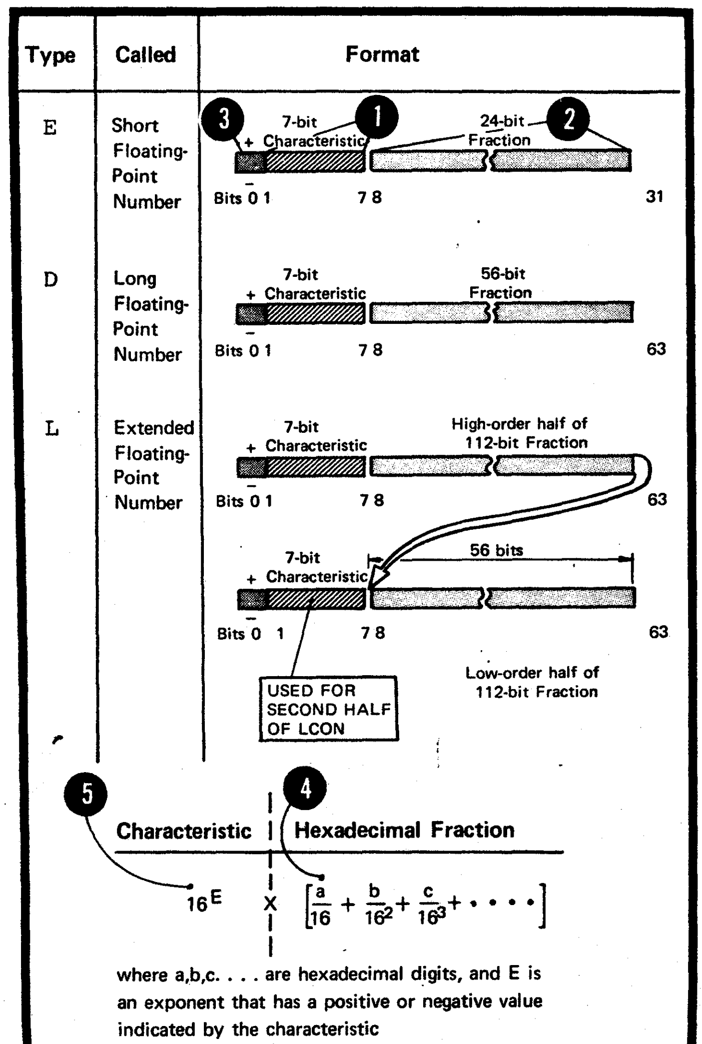
Type of Constant	Range of Magnitude (M) of Values (Positive and Negative)
E	$16^{-65} \leq M \leq (1-16^{-6}) \times 16^{63}$
D	$16^{-65} \leq M \leq (1-16^{-14}) \times 16^{63}$
L	$16^{-65} \leq M \leq (1-16^{-28}) \times 16^{63}$
	(For all Three) Approximately $5.4 \times 10^{-79} \leq M \leq 7.2 \times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

FORMAT: The format of the floating-point constants is described below. The value of the constant is represented by two parts:

(1) 1. An exponent portion, followed by

(2) 2. A fractional portion.

(3) A sign bit 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 (4) fraction and the number 16 raised to a power. (5)



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:

(1) 1. A fractional portion represented by hexadecimal digits and the sign (2) indicator. The fraction is then entered into the leftmost part of the fraction field of the constant (after rounding).

(3) 2. An exponent portion represented (4) 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.

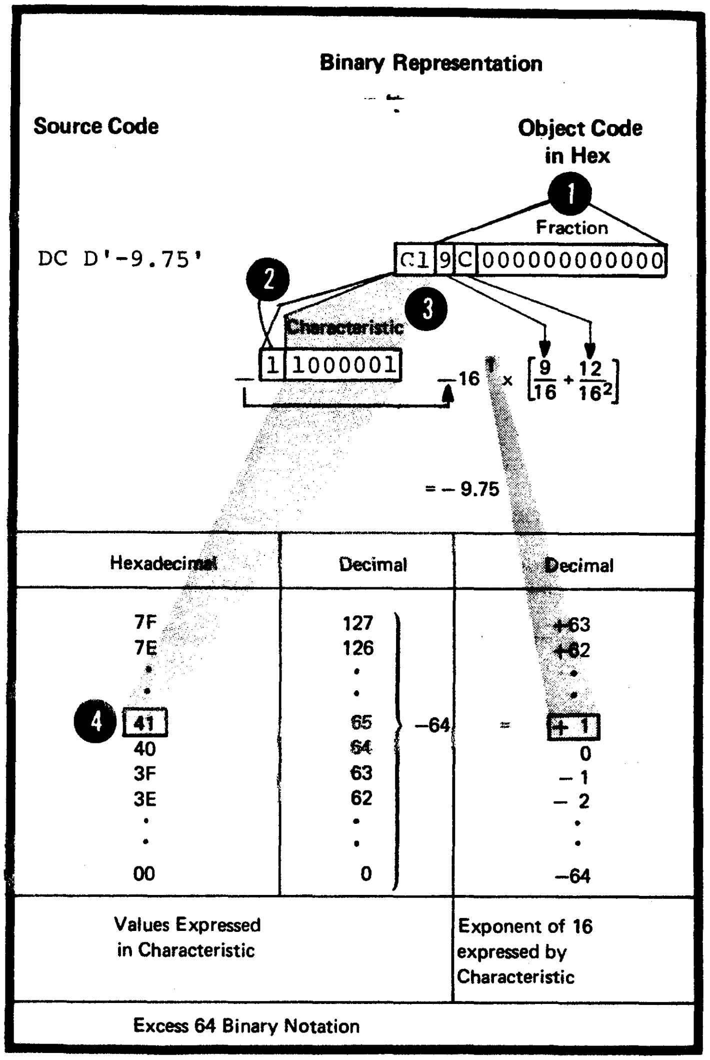
The characteristic for the second doubleword is equal to the characteristic of the first doubleword minus 14 (the number of hexadecimal digits in the fractional portion of the first doubleword).

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 G3B under Modifiers).

3. Rounding of the fraction is performed according to the implicit or explicit length of the constant. The resulting number will not differ from the exact number specified by more than one in the last place.

4. Negative fractions are carried in true representation, not in the 2's complement form.

5. Duplication is applied after the constant has been assembled.



G3J -- THE A-TYPE AND Y-TYPE ADDRESS CONSTANTS

This subsection and the three following subsections 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.

Purpose

In the A-type and Y-type address constant, you can specify any of the three types of assembly-time expressions (see C6), whose value the assembler then computes and assembles into object code. You use this expression computation as follows:

1. Relocatable expressions for addressing

2. Absolute expressions for addressing and value computation.

3. Complex relocatable expressions to relate addresses in different source modules.

Specifications

The contents of the subfields defining the A-type and Y-type address constants are described in the figure on the opposite page.

NOTES:

(1) 1. No bit-length specification is allowed when a relocatable
(2) or complex relocatable expression is specified. The only explicit lengths that can be specified with these addresses are:

   a. 3 or 4 bytes for A-type constants

   b. 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:

   a. Multiple operands

(3)   b. Multiple nominal values

(4)   c. A duplication factor.

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	Address Constants (A and Y)		
	3. Constant Type		
	A – Type	Y – Type	4
1. Duplication Factor allowed	Yes	Yes Object Code in Hex ——→	A DC 5AL1(*-A) 0001020304
2. Modifiers Implicit Length: (Length Modifer not present)	4 bytes	2 bytes	
Alignment: (Length Modifier not present)	Full word	Half word	
Range for Length:	1 through 4 (byte length) 1 .1 through .32 (bit length)	1 through 2 (byte length) .1 through .16 (bit length)	
Range for Scale:	Not allowed	Not allowed	
Range for Exponent:	Not allowed	Not allowed	
4. Nominal Value Represented by:	Absolute, relocatable, or complex relocatable expressions 2 DC A(ABSOL+10)	Absolute, relocatable, or complex relocatable expressions DC Y(RELOC+32)	3 A DC Y(*-A,*+4) 0 A+6 values
Enclosed by:	Parentheses	Parentheses	
Exponent allowed:	No	No	
Number of Values per Operand:	Multiple	Multiple	
Padding:	With zeros at left	With zeros at left	
Truncation of Assembled value:	At left	At left	

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.

G3K -- THE S-TYPE ADDRESS CONSTANT

Purpose

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 (for details on implicit and explicit addresses, see D5B).

Specifications

The contents of the subfields defining the S-type address constants are described in the figure on the opposite page.

The nominal values can be specified in two ways:

(1) 1. As one absolute or relocatable expression representing an implicit address

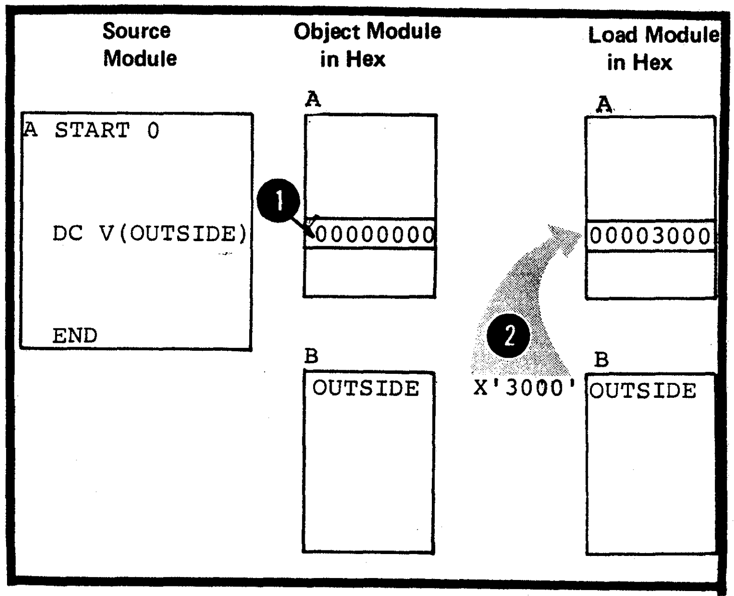
(2) 2. As two absolute expressions, the first of which represents the displacement (3) and the second, the base register. (4)

Subfield	Address Constants (S)		
	3. Constant Type		
	S – Type		
1. Duplication Factor Allowed	Yes		
2. Modifiers Implicit Length: (Length Modifier not present)	2 bytes		
Alignment: (Length Modifier not present)	Half word		
Range for length: (in bytes)	2 only (no bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed		
4. Nominal Value Represented by:	Absolute or relocatable expression (1) Two absolute expressions (2)	DC S(RELOC) DC S(1024) DC S(512(12)) (3) (4)	C XXX 0 400 Base Displacement C 200
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of Values per operand :	Multiple		
Padding:	Not applicable		
Truncation of Assembled value:	Not applicable		

Purpose

The V-type address 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 (for a comparison, see F2).

Because you specify a symbol in a V-type address constant, the assembler assumes that it is an
1 external symbol. A value of zero is assembled into the space reserved for the V-type constant; the correct
2 relocated value of the address is inserted into this space by the linkage editor before your object program is loaded.



Specifications

The contents of the subfields defining the V-type address constants are described in the figure on the opposite page.

1 The symbol specified 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.

V

Subfield	Address Constants (V)		
	3. Constant Type		
	V – Type		
1. Duplication Factor allowed	Yes		
2. Modifiers Implicit Length: (Length Modifier not present)	4 bytes		
Alignment: (Length Modifier not present)	Full word		
Range for Length: (in bytes)	4 or 3 only (no bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed		
4. Nominal Value Represented by:	A single relocatable symbol	DC V(MODA) ❶ DC V(EXTADR)	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of values per Operand:	Multiple		
Padding:	With zeros at left		
Truncation of assembled value:	Not applicable		

OS G3M -- THE Q-TYPE ADDRESS CONSTANT
only

Purpose

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 E4.)

Specifications

The contents of the subfields defining the Q-type address constant are described in the figure below.

(1) The symbol specified in the nominal value subfield must be previously defined as the label of a DXD or DSECT statement.

Q

Subfield	Address Constants (Q)		
	3. Constant Type		
	Q-Type		
1. Duplication Factor allowed	Yes		
2. Modifiers Implicit Length: (Length Modifier not present)	4 bytes		
Alignment: (Length Modifier not present)	Fullword		
Range for Length: (in bytes)	1-4 bytes (no bit length)		
Range for Scale:	Not allowed		
Range for Exponent:	Not allowed		
4. Nominal Value Represented by	A single relocatable symbol	DC Q(DUMMYEXT) DC Q(DXDEXT) (1)	
Enclosed by:	Parentheses		
Exponent allowed:	No		
Number of Values per Operand:	Multiple		
Padding:	With zeros at left		
Truncation of Assembled Value	At left		

G3N -- THE DS INSTRUCTION

Purpose

The DS instruction allows you to:

1. Reserve areas of storage
2. Provide labels for these areas
3. Use these areas by referring to the symbols defined as labels.

The DS instruction causes no data to be assembled. Unlike the DC instruction (see G3B), 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.

How to Use the DS Instruction

TO RESERVE STORAGE: If you wish to take advantage of ❶ automatic boundary alignment (if the ALIGN option is specified) and ❷ 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 (See individual types in sections G3D through G3M).

Named (Mnemonic) Areas for Fixed-Point Instructions	Areas Aligned on Boundary ❶	Length Attribute of Symbols Naming Areas same as Implicit Length of Areas ❷
FAREA DS F	Full word	4
HAREA DS H	Half word	2
AAREA DS A	Full word	4
DUPF DS 10F (10 full words of storage reserved)	Full word	L'DUPF=4 (Duplication has no effect on implicit length)
Named Areas for Floating-Point Instructions		
EAREA DS 3E	Full word	4
DEAREAS DS 9D (9 double words reserved)	Double word	8
LAREA DS L	Double word	16

(1) Using a length modifier can give you the advantage of explicitly (2) specifying the length attribute value assigned to the label naming the area reserved. However, your areas will not be aligned automatically according to their (3) type. If you omit the nominal value in the operand, you should use a length modifier for the binary (B), character (C), hexadecimal (X), and decimal (P and Z) type areas; otherwise their labels will be given (4) a length attribute value of 1.

Area Specified (1)			Area Reserved in in Bytes	Length Attribute (2)
TEN	DS	CL10 (1)	10	10
TWO56	DS	XL256	256	256
F3	DS	FL3	3	3
D7	DS	DL7	7	7
A2	DS	AL2	2	2
C1	DS	CL16 (1)	16	16
C2	DS	16C (3)	16	1 (4)
C3	DS	C (3)	1	1 (4)
X1	DS	XL200 (1)	200	200
X2	DS	X (3)	1	1 (4)
X3	DS	200X (3)	200	1 (4)

Duplication factor has no effect on length attribute

❶ When you need to reserve large areas you can use a duplication factor. However, you can only refer to the ❷ first area by the label in this case. You can also use the character (C) and hexadecimal (X) field types to specify ❸ large areas using the length modifier.

Area Specified	Area Reserved in Bytes	Automatic Boundary Alignment	Length Attribute of symbol used as Label
LARGEF DS 10FL3 ❶ ❷	30	NONE	3
LARGED DS 100D ❷	800	DOUBLE WORD	8
LARGEA DS 60A ❷	240	FULL WORD	4
LARGEC DS 1000C	1000	NONE	1
C2 DS CL1000	1000	NONE	1000
XLARGE DS XL2000	2000	NONE	2000
LARGERX DS 2XL2000	4000	NONE	2000

Duplication has no effect

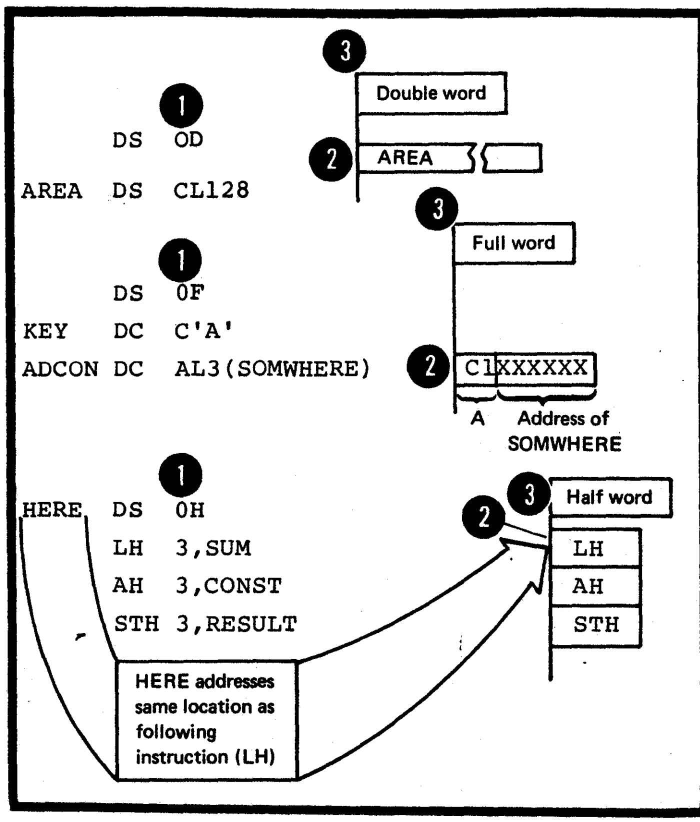
❸ Maximum of 65,535 allowed

Although the nominal value is optional for a DS instruction, you can put it to good use by letting
❶ the assembler compute the length for areas of the B, C, X, and decimal (P or Z) type areas. You achieve this by specifying the general
❷ format of the nominal value that will be placed in the area at execution time.

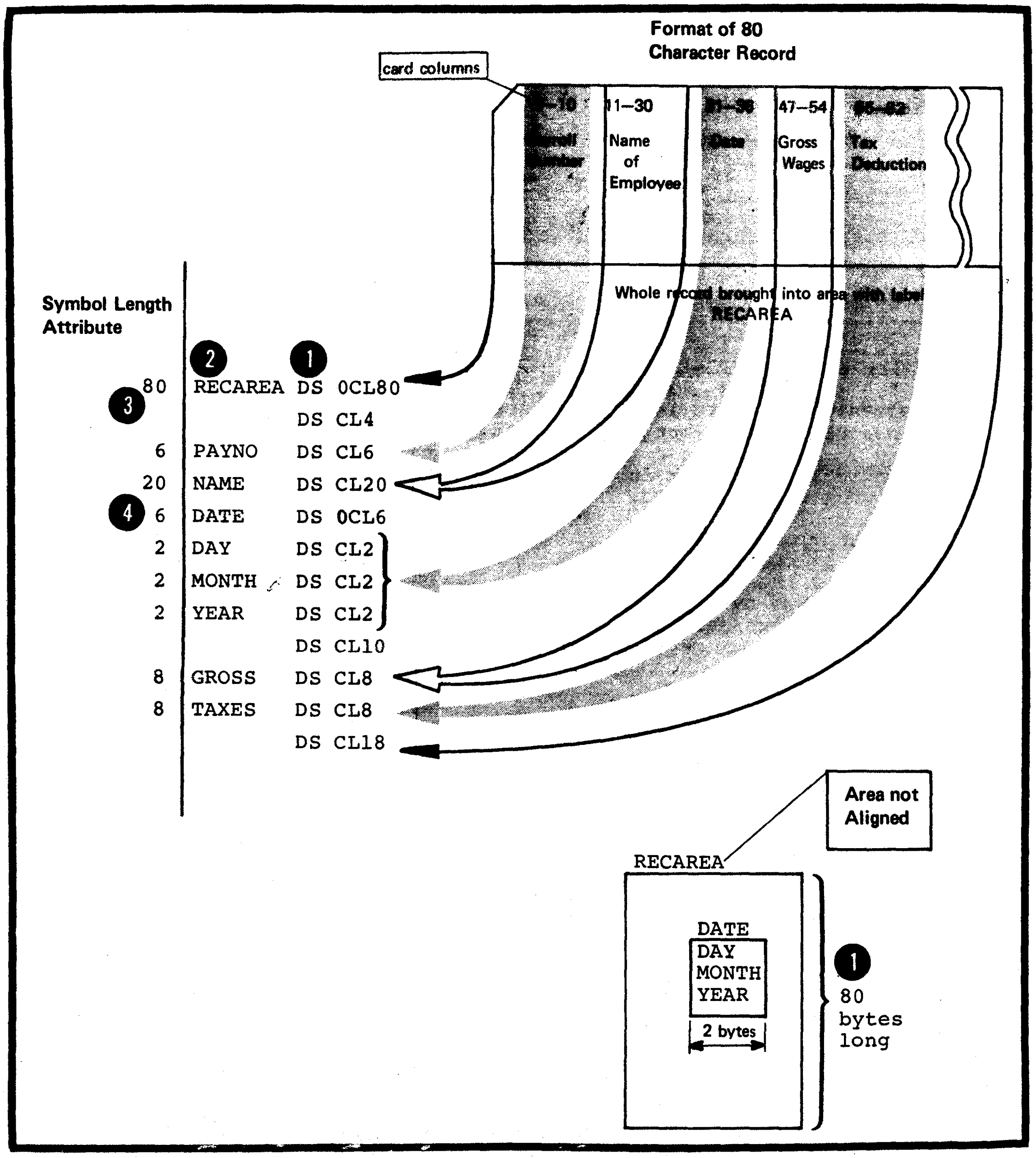
Area Specified	Area Reserved in bytes	Length Atribute or computed implicit length of area (duplication disregarded) ❶
C1 DS C'THIS IS AN ERROR'	16	16
X1 DS X'0A1'	2	2
X2 DS 30X'F1F2'	60	2
P1 DS P'99999'	3	3
P2 DS 5P'99999'	15	3
Z1 DS Z'99999'	5	5

TO FORCE ALIGNMENT: You can use the DS instruction to force alignment to a boundary that otherwise would not be provided. You do this by
❶ using a duplication factor of zero. No space is reserved for such an
❷ instruction, yet the data that follows is aligned on the desired
❸ boundary.

NOTE: Alignment is forced when either the ALIGN or NOALIGN assembler option is set (see D2).



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 E3C.) 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. The DATE field has the same address as the subfield DAY. However, DATE addresses 6 bytes, while DAY addresses only 2 bytes.

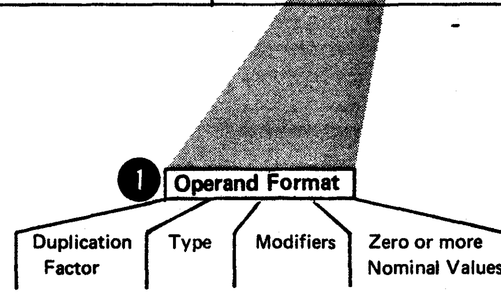


Specifications

The format of the DS instruction statement is given in the figure to the right.

1 The format of the operand of a DS instruction is identical to that of the DC operand (see G3B).

Name	Operation	Operand
Any Symbol or blank	DS	One or more Operands separated by commas



The two differences in the specification of subfields are:

1 The nominal value subfield is optional in a DS operand, but it is mandatory in a DC operand. If
2 a nominal value is specified in a DS operand, it must be valid.

3 The maximum length that can be specified in a DS operand for the character (C) and hexadecimal (X) type areas is 65,535 bytes, rather than 256 bytes for the same DC operands.

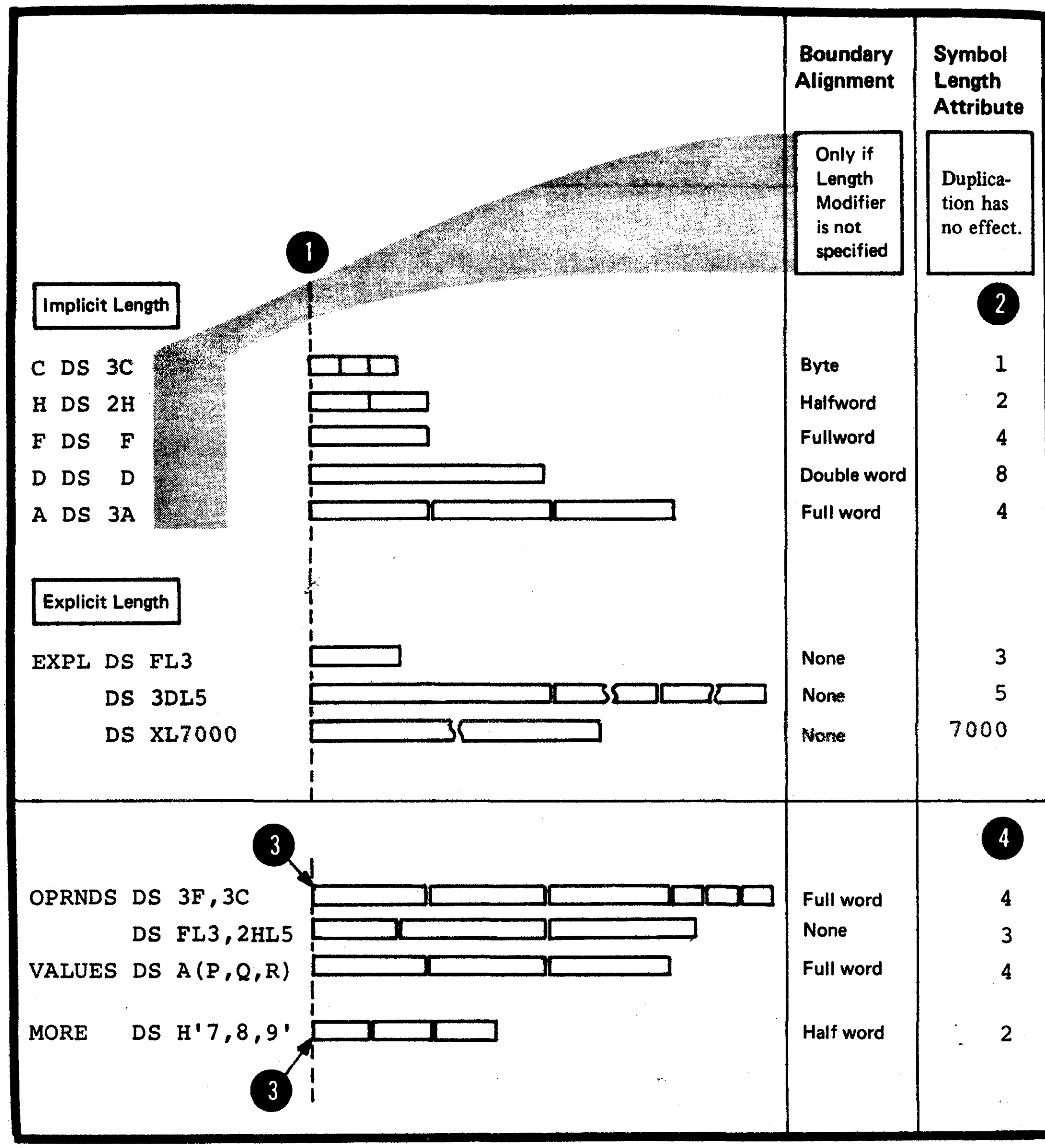
```
OPTION1   DS   3FL3        1
OPTION2   DS   3FL3'3'
AMUST     DC   3FL3'3'
                  2

LONGC     DS   CL65535     3
LONGX     DS   XL65535

LIMITEDC  DC   CL256'A'
LIMITEDX  DC   XL256'00'
```

The label used in the name entry of a DS instruction, like the label for a DC instruction (see G3B):

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

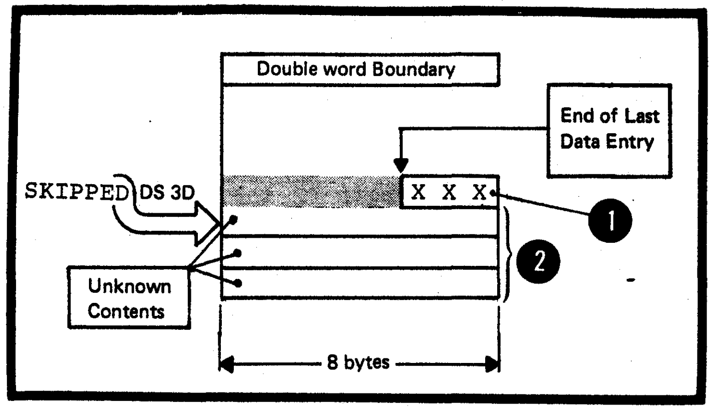
❷ 2. Has a length attribute value, depending on the implicit or explicit length of the type of area reserved.

If the DS instruction is specified with more than one operand or more than one nominal value in the operand, ❸ the label addresses the area reserved for the field that corresponds to the first nominal value of the first operand. ❹ The length attribute value is equal to the length explicitly specified or implicit in the first operand.



❶ ❷ NOTE: Unlike the DC instruction, bytes skipped for alignment are not set to zero. Also, ncthing 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, whichever is smaller.

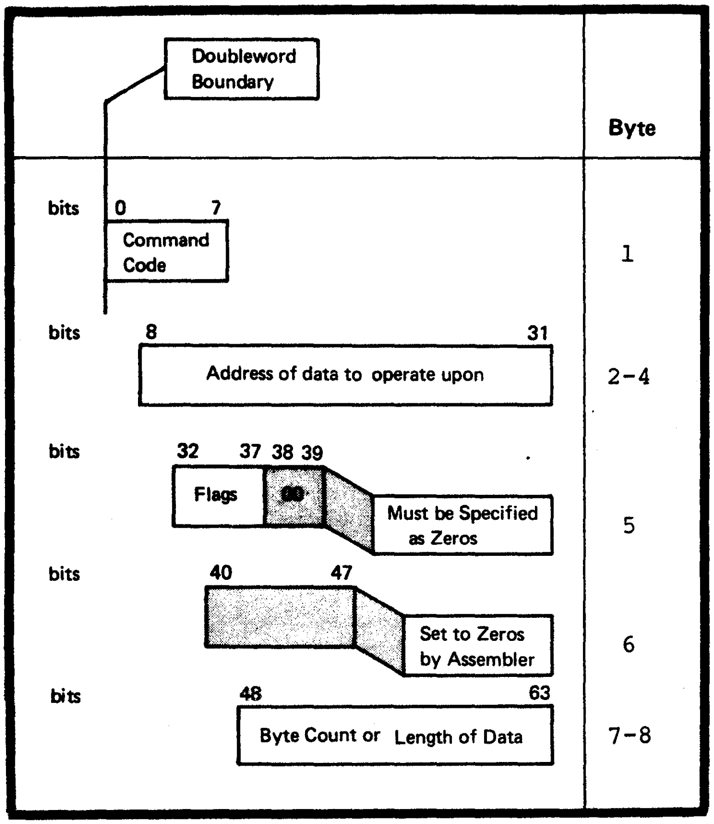


G30 -- THE CCW INSTRUCTION

Purpose

You can use the CCW instruction to define and generate an eight-byte channel command word for input/output operations.

The channel command word is an eight-byte field aligned at a doubleword boundary, and contains the information described in the figure to the right.



Specifications

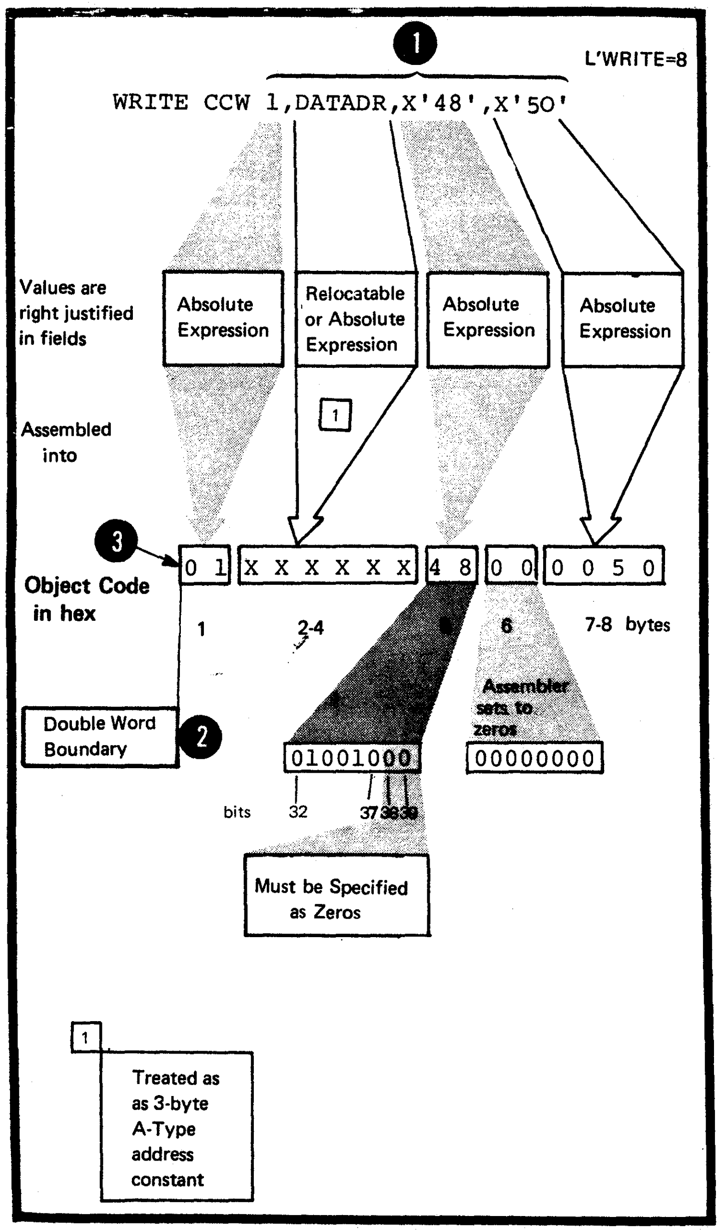
The format of the CCW instruction statement is given in the figure to the right.

CCW

Name	Operation	Operand
Any symbol or blank	CCW	Four operands separated by commas

❶ All four operands must be specified in the order described in the figure to the right. The generated channel command word is aligned on a ❷ doubleword boundary. Any bytes skipped are set to zero.

❸ 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.



Section H: Controlling the Assembler Program

This section describes the assembler instructions that request the assembler to perform certain functions that it would otherwise perform in a standard predetermined way. You can use these instructions to:

1. Change the standard coding format for writing your source statements

2. Control the final structure of your assembled program

3. Alter the format of the source module and object code printed on the assembler listing

4. Produce punched card output in addition to the object deck

5. Substitute your own mnemonic operation codes for the standard codes of the assembler language

6. Save and restore programming environments, such as the status of the PRINT options and the USING base register assignment.

H1 -- Structuring a Program

The instructions described in this subsection affect the location counter and thereby the structure of a control section. You can use them to interrupt the normal flow of assembly and redefine portions of a control section or to reserve space to receive literal constants. Also, you can use them to align data on any desired boundary.

H1A -- THE ORG INSTRUCTION

Purpose

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.

For example, if you wish to build a translate table (to convert EBCDIC character code into some other internal code):

❶ 1. You define the table as being filled with zeros.

❷ 2. You use the ORG instruction to alter the location counter so that its counter value indicates a desired location within the table.

❸ 3. You redefine the data to be assembled into that location.

❹ 4. After repeating the first three steps until your translate table is complete, you use an ORG instruction with a blank operand field to alter the location counter so that the counter value indicates ❺ the next available location 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 counter 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.

Source Module | **Object Code**

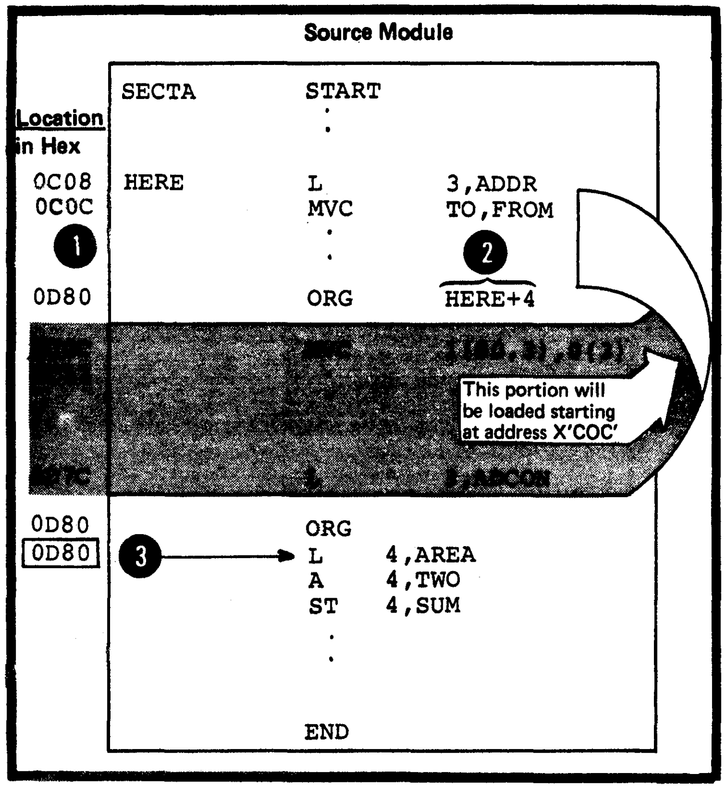
```
                                                     Object Code
FIRST        START    0                              TABLE
             .                      ❶                (in Hex)
             .
TABLE        DC       XL256'00'
             ORG      TABLE+0   ❷            +0
           { DC       C'0'  ❸                          F0
           { DC       C'1'                             F1
             .                                         .
             .                                         .
             ORG      TABLE+13               +13
             .
             DC       C'D'                             C4
             DC       C'E'                             C5
      ❹      .
             .
             ORG      TABLE+C'D'             +196
             DC       AL1(13)                          0D
             DC       AL1(14)                          0E
             .
             ORG      TABLE+C'0'             +240
             DC       AL1(0)                           00
      ❺      DC       AL1(1)                           01
TABLE+256    .
  |                                          +255
  v          ORG
GOON         DS       0H
             .
             TR       INPUT,TABLE
             .
INPUT        DS       CL20
             .
             .
             END
```

ORG

Specifications

The format of the ORG instruction is shown in the figure to the right.

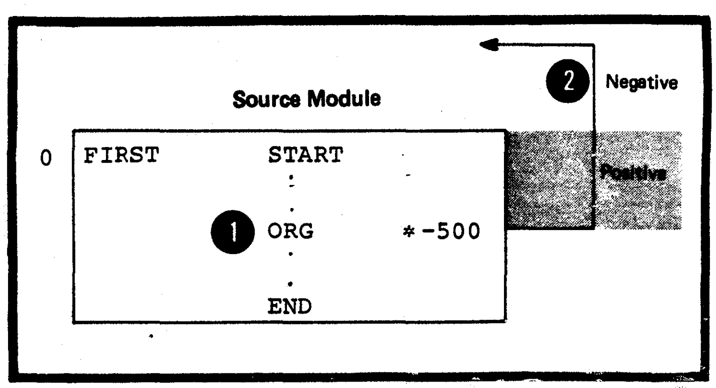
Name	Operation	Operand
OS Any symbol or blank [redacted]	ORG	A relocatable expression or blank



The symbols in the expression in the operand field must be previously defined. The unpaired relocatable term of the expression (see C6B) must be defined in the same control section in which the ORG statement appears.

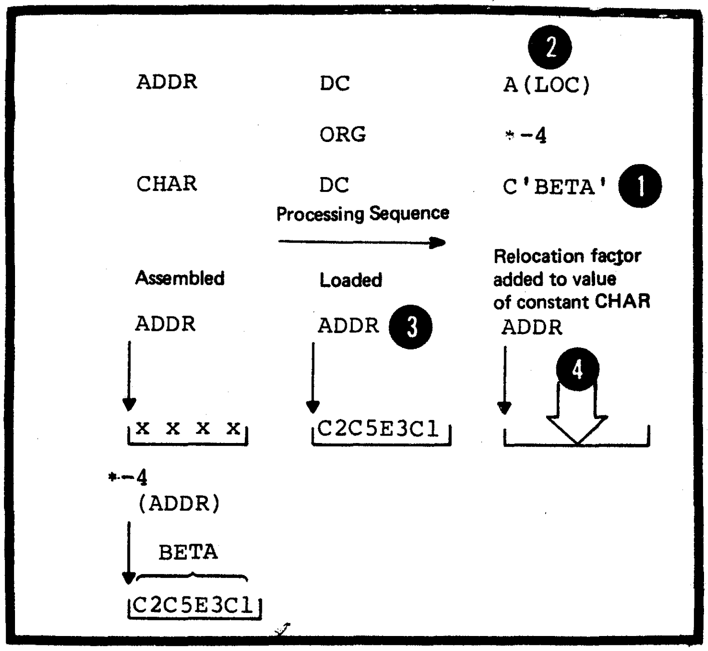
1 The location counter is set to the
2 value of the expression in the operand. If the operand is omitted, the location counter is set to the
3 next available location for the current control section.

The expression in the operand of an ORG instruction must not specify a location before the beginning of the control section in which it appears. In the example to the
1 right, the ORG instruction is invalid if it appears between the beginning of the current control section and 500 bytes from the beginning of the same control section. This
2 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 section C4B).



NOTE: Using the ORG instruction to insert data assembled later at the same location as earlier data will not always work.

1 In the example to the right, it appears as if the character constant
2 will be loaded over the address constant. However, after the character constant is loaded into
3 the same location as the address constant, the relocation factor required for the address constant
4 is added to the value of the constant. This sum then constitutes the object code that resides in the four bytes with the address ADDR.



H1B -- THE LTORG INSTRUCTION

Purpose

You use the LTORG statement 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:

1 After the preceding LTORG instruction or

2 After the beginning of the source module.

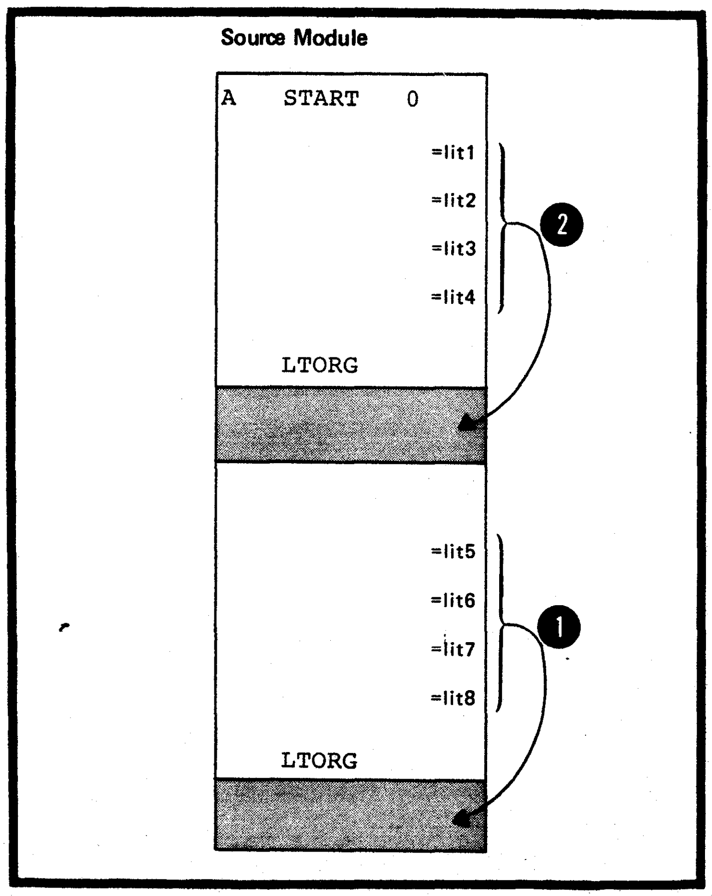
The assembler ignores the borders between control sections when it collects literals into pools. Therefore, you must be careful to include the literal pools in the control sections to which they belong (for details see Addressing Considerations below).

The creation of a literal pool gives the following advantages:

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

2. Assembling of duplicate data into the same area

3. Because all literals are cross-referenced, you can find the literal constant in the pool into which it has been assembled.



The 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 (1) in the order that the literals are specified and (2) according to their assembled lengths, which, for each literal, is the total explicit or implicit length, as described below.

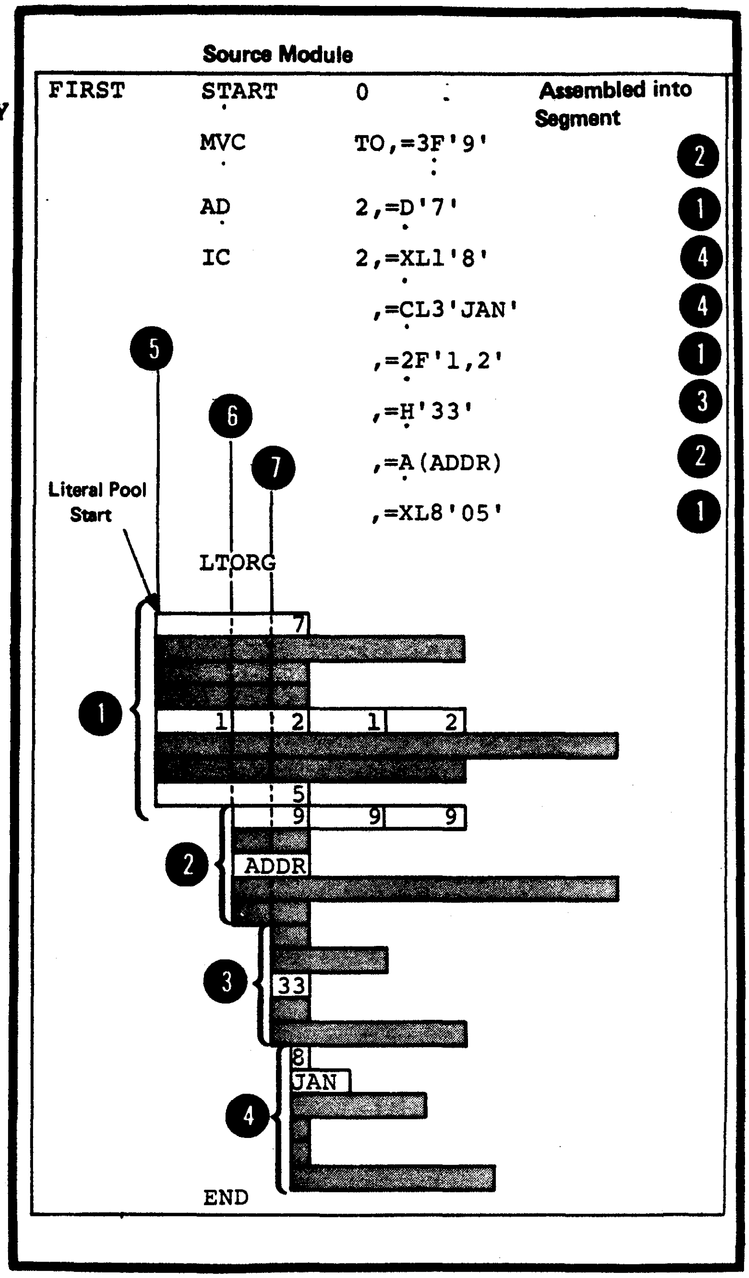
1 The first segment contains all literal constants whose assembled lengths are a multiple of eight.

2 The second segment contains those whose assembled lengths are a multiple of four, but not of eight.

3 The third segment contains those whose assembled lengths are even, but not a multiple of four.

4 The fourth segment contains all the remaining literal constants whose assembled lengths are odd.

5 The beginning of each literal pool is aligned on a doubleword boundary. Therefore, the literals in the first segment are always aligned on a doubleword boundary, those in the 6 second segment on a fullword boundary, and those in the third 7 segment on a halfword boundary.



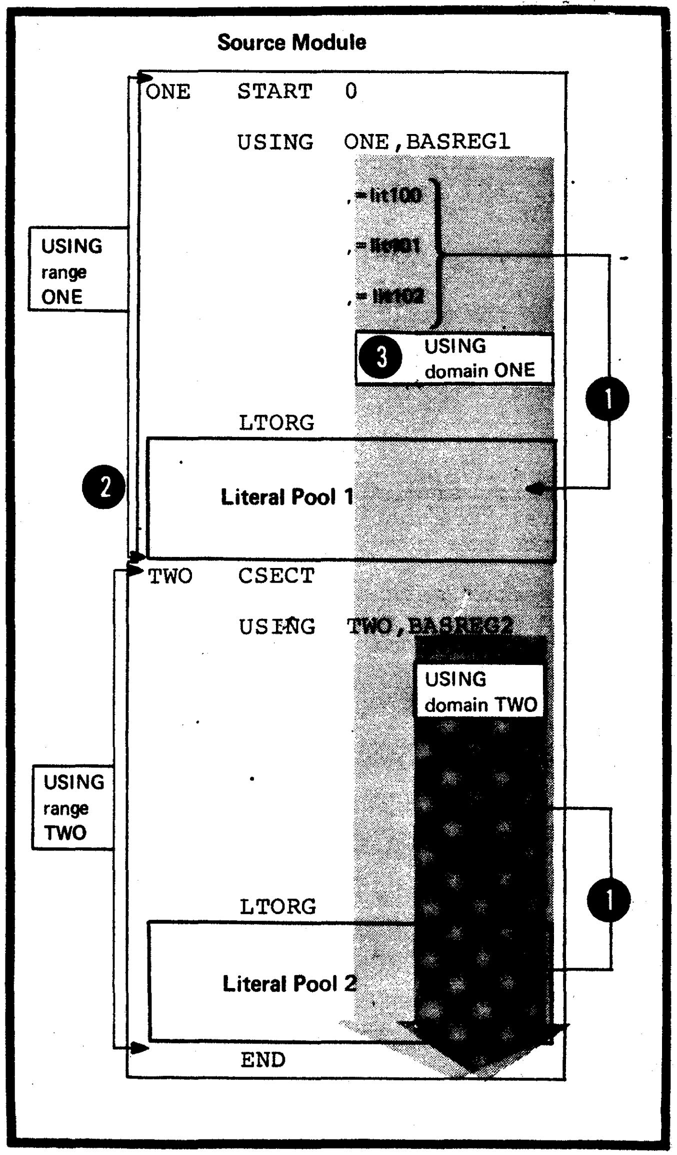
Addressing Considerations

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

1. Write a LTORG instruction at the end of each control section, so that all the literals specified in the section are assembled into
(1) 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.

2. When establishing the addressability of each control section, make sure (a) that the entire literal pool for that section is also addressable, by including
(2) it within a USING range, and (b) that the literal specifications are within the corresponding USING
(3) domain. The USING range and domain are described in F1A.

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.

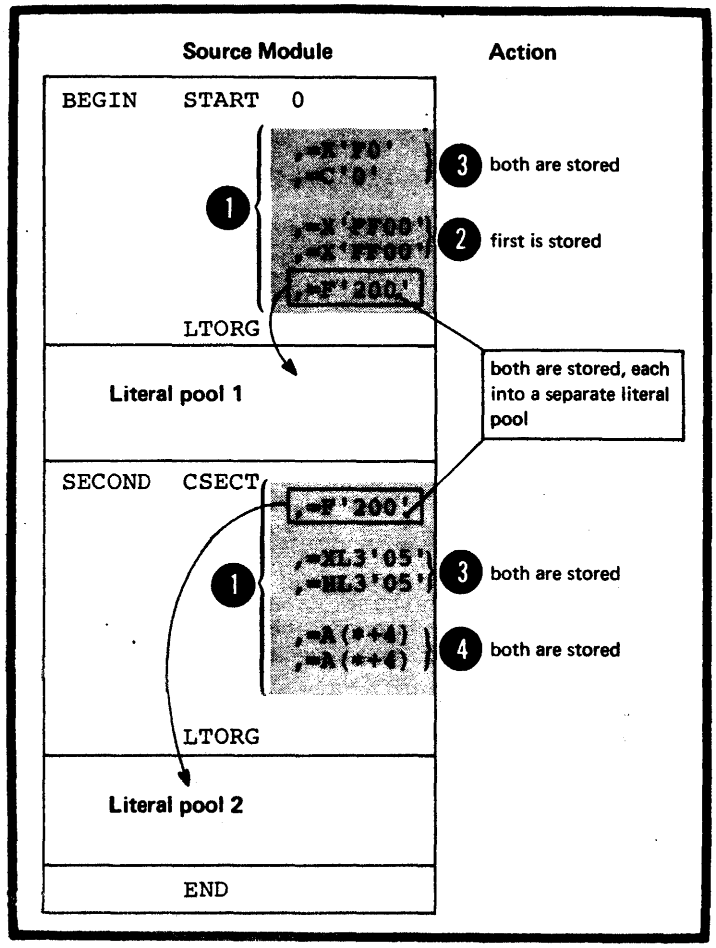


Duplicate Literals

If you specify duplicate literals within the part of the source module (1) 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.

(2) Literals are duplicates only if their specifications are identical, (3) not if the object code assembled happens to be identical.

When two literals specifying (4) 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.



Specifications

The format of the LTORG instruction is given in the figure to the right.

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 one. 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.

LTORG

Name	Operation	Operand
Any symbol or blank	LTORG	Not required

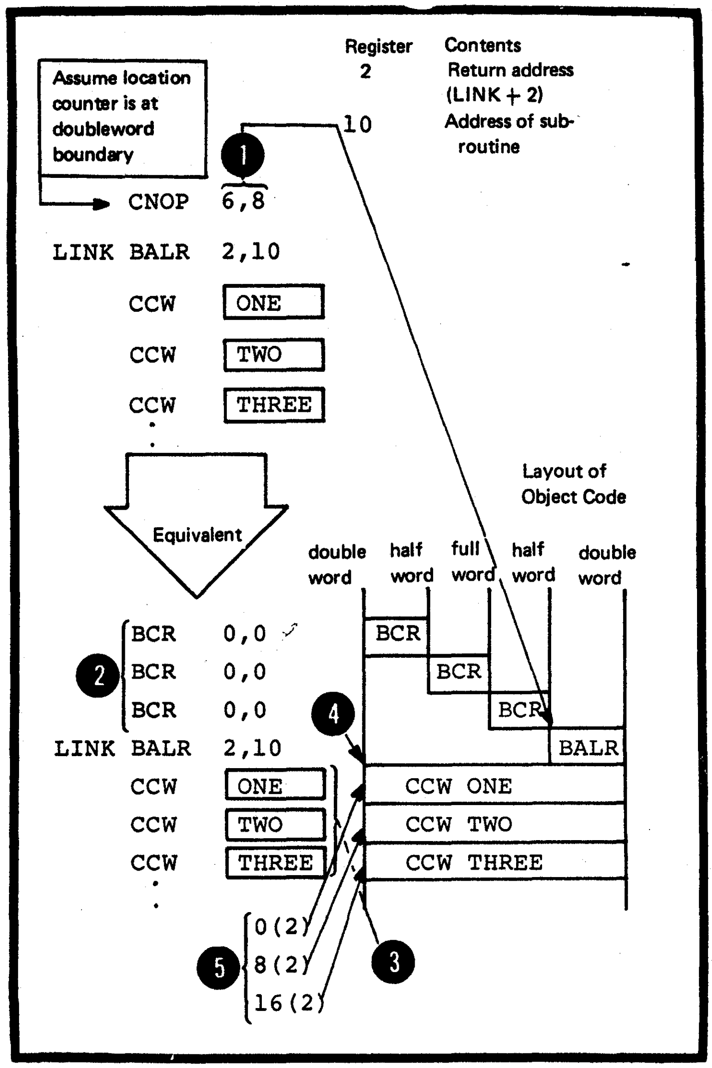
H1C -- THE CNOP INSTRUCTION

Purpose

You can use the CNOP instruction to align any instruction or other **1** data on a specific halfword boundary. The CNOP instruction ensures an unbroken flow of executable instructions by generating **2** 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 **3** wish to pass parameters to the subroutine in fields immediately following the **4** branch and link instruction. These parameters, for instance, channel command words (see G30), can require alignment on a specific boundary.

The subroutine can then **5** address the parameters you pass through the register with the return address.



Specifications

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.

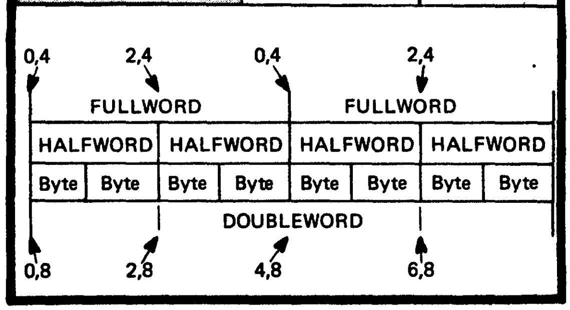
The format of the CNOP instruction statement is given in the figure to the right.

The operands must be absolute expressions, and any symbols must have been previously defined.
(1) The first operand, b, specifies at which even-numbered byte in a fullword or doubleword the location counter is set. (2) 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 as indicated in the figure to the right.

NOTE: Both 0,4 and 2,4 specify two locations in a doubleword.

CNOP

	Name	Operation	Operand
OS	Any symbol or blank	CNOP	b,w (1) (2)
DOS	Sequence symbol or blank		



H2 -- Determining Statement Format and Sequence

You can change the standard coding conventions for the assembler language statements or check the sequence of source statements by using the following instructions.

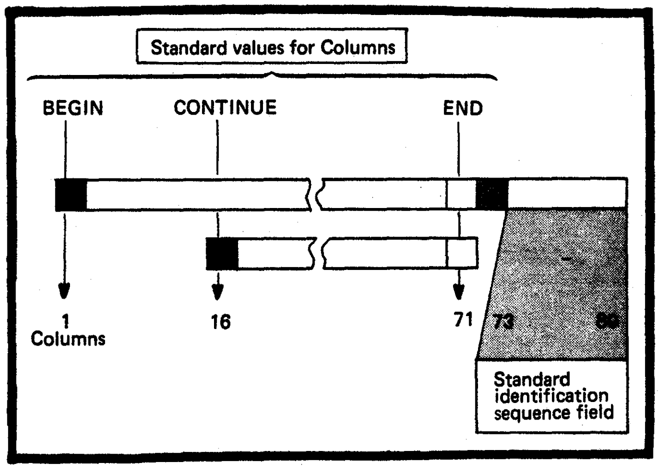
H2A -- THE ICTL INSTRUCTION

Purpose

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 module. If you do not use it, the assembler recognizes the standard values for the begin, end, and continue columns.



Specifications

The ICTL instruction, if specified, must be the first statement in a source module.

The format of the ICTL instruction statement is shown in the figure to the right.

The operand entry must be one to three decimal self-defining terms.
(1) There are only <u>three possible ways</u> of specifying the operand entry.

(2) The <u>operand b</u> must always be
(3) specified. The <u>operand e</u>, when not specified, is assumed to be 71.
(4) If the <u>operand c</u> is not specified, or if e is specified as 80, the assembler assumes that continuation lines are not allowed. The values
(5) specified for the <u>three operands depend on each other</u>.

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 Section B1A.

ICTL

Format

Name	Operation	Operand
Blank	ICTL	b or b,e or b,e,c (1)

Operands

	Specifies	Allowable range
(2) b	Begin column	1 through 40
(3) e	End column	41 through 80
(4) c	Continue column	2 through 40

(5) Rules for interaction of b, e and c

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	e ≥ b + 5 e > c
The position of the Continue column must be greater than that of the Begin column	c > b

H2B -- THE ISEQ INSTRUCTION

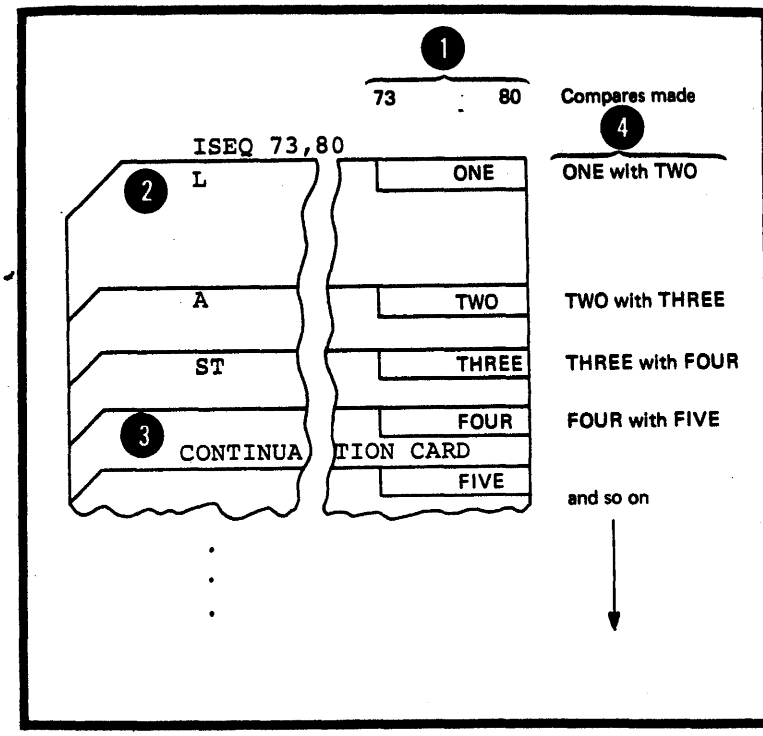
Purpose

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 (1) columns between which the assembler is to check for sequence numbers.

The assembler begins sequence checking with the (2) first statement line following the ISEQ instruction. The assembler also checks (3) continuation lines.

Sequence numbers on adjacent (4) 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 non-blank sequence field to make its comparison.



Specifications

The ISEQ instruction initiates or terminates the checking of the sequence of statements in a source module.

The format of the ISEQ instruction is shown in the figure to the right.

(1) The first option in the operand entry must be two decimal self-defining terms. This format of the ISEQ instruction initiates sequence checking, beginning at the statement or line following the (2) the ISEQ instruction. Checking begins at the column represented (3) by l and ends at the column represented by r . The second (4) option of the ISEQ format terminates the sequence checking operation.

ISEQ

Name	Operation	Operand
Blank	ISEQ	l, r (1) or blank (4)

Column	Specifies	Rules for interaction
(2) l	leftmost column of field to be checked	l ≤ r — l must not be greater than r
	l and r not allowed to lie between begin and end columns	
(3) r	rightmost column of field to be checked	r ≥ l — r must not be less than l

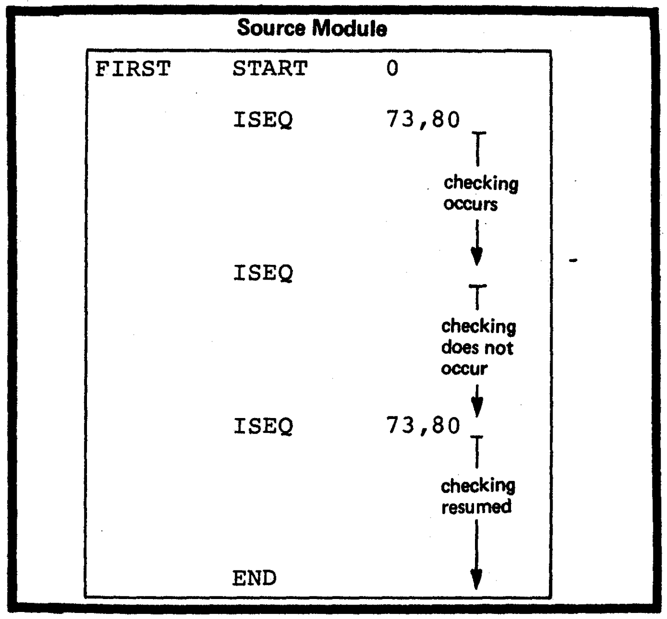
NOTE: The assembler checks only those statements that are specified in the coding of a source module. This includes any COPY instruction statement or macro instruction.

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 Section J)

3. Statements in library macro definitions.



H3 -- Listing Format and Output

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.

H3A -- THE PRINT INSTRUCTION

Purpose

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 figure to the right.

They 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.

Hierarchy	Description	PRINT 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 (Note: The MNOTE instruction always causes a message to be printed)	NOGEN
3	Constants are printed in full in the listing	DATA
	Only the leftmost eight bytes of constants are printed in the listing	NODATA

Specifications

PRINT

Name	Operation	Operand
A sequence symbol or blank	PRINT	{[ON / OFF] [,GEN / ,NOGEN] [,NODATA / ,DATA]} (1) Any sequence of specification allowed

The format of the PRINT instruction statement is shown in the figure to the right.

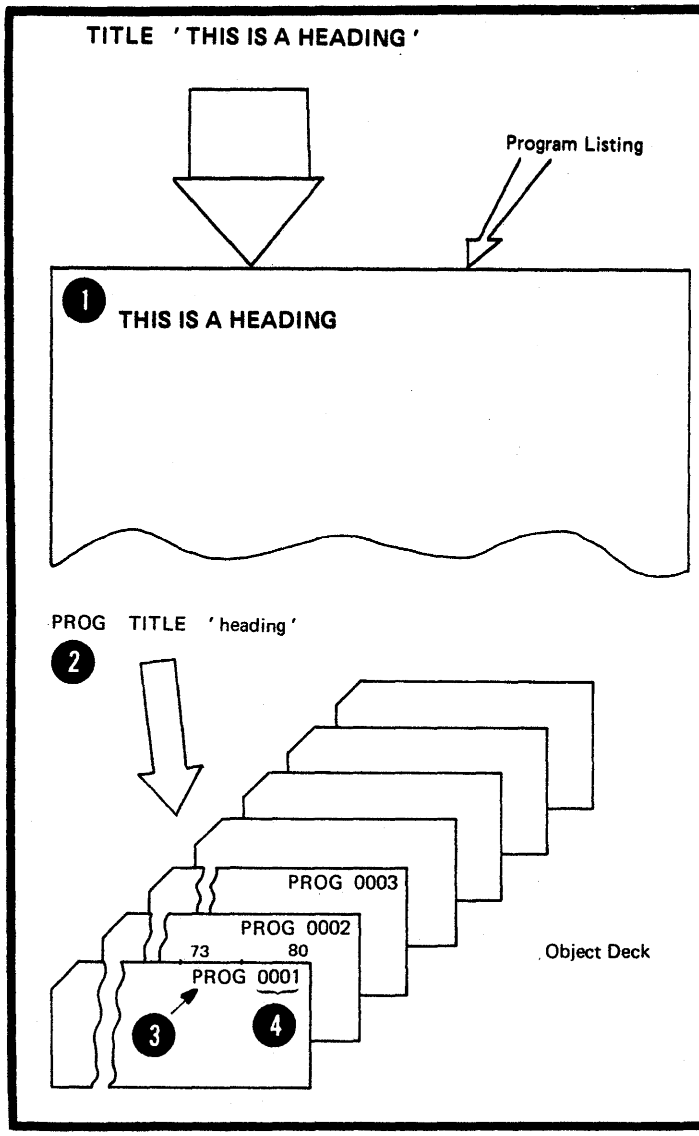
(1) At least one of the operands must be specified, and at most one of the options from each group. The PRINT instruction can be specified any number of times in a source module, but only those print options actually specified in the instruction change the current print status.

PRINT options can be generated by macro processing, at pre-assembly time. However, at assembly time, all options are in force until the assembler encounters a new and opposite option in a PRINT instruction.

OS only **The PUSH and POP instructions, described in H6, 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 in the job control language, the entire listing for the assembly is suppressed.

H3B -- THE TITLE INSTRUCTION



Purpose

The TITLE instruction allows you to:

(1) 1. Provide headings for each page of the assembly listing of your source modules.

2. Identify the assembly output cards of your object modules. You can specify up to 8 identification (2) characters that the assembler will punch into all the output cards, (3) beginning at column 73.

DOS **Up to 4 identification characters are allowed.**

(4) The assembler punches sequence numbers into the columns that are left, up to column 80.

NOTE: The name field of the TITLE instruction is generated throughout the assembly listing, preceding the generation of the operand data from any TITLE instruction.

Specifications

The format of the TITLE instruction statement is given in the figure to the right.

Any of the five options can be specified in the name field.

(1) 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.

TITLE

option	Name	Operation	Operand
(1) 1	A string of alphameric characters	TITLE	A character string up to 100 characters, enclosed in apostrophes
2	A variable symbol		
3	A combination of 1 and 2		
4	A sequence symbol		
5	blank		

For the first TITLE instruction of a source module that has a non-blank name entry that is not a sequence symbol, the following applies:

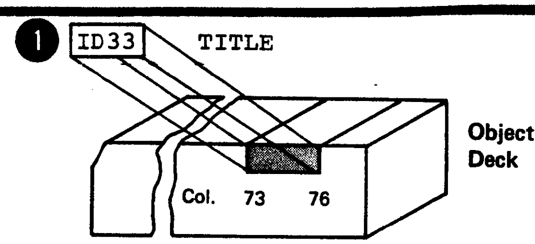
1 Up to eight alphameric characters can be specified in any combination in the name field.

DOS Up to four alphameric characters can be specified.

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.

2 The assembler substitutes the current value into a variable symbol and uses the generated result as identification characters.

3 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.



Examples of TITLE instructions in separate source modules:

Source Statement	Value of variable symbol	Punched into cards beginning at col. 73
&ID (2) TITLE	MOD99A	MOD99A
PGM&N TITLE	200	PGM200
1234 TITLE		1234
SYMBOL TITLE (3)		SYMBOL

(1) 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.

Each TITLE statement causes the listing to be advanced to a new page (before the heading is printed) except when PRINT NOGEN is in use.

Any printable character specified will appear in the heading, including blanks. Variable symbols are allowed. However, the following rules apply to ampersands and apostrophes:

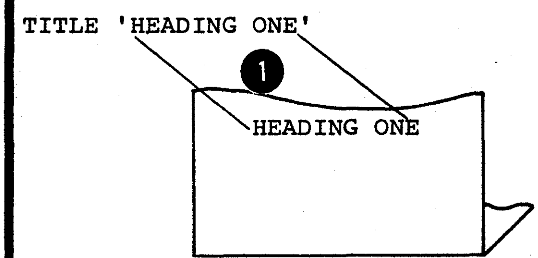
- (2) A single ampersand initiates an attempt to identify a variable symbol and to substitute its current value.

- (3) Double ampersands or apostrophes specified, print as single ampersands or apostrophes in the heading.

- (4) A single apostrophe followed by one or more blanks simply terminates the heading prematurely. If a non-blank 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.



Examples of headings:

Source Statement	Value of Variable Symbol	Printed Heading
(2) TITLE 'HEADING &N'	TWO	HEADING TWO
TITLE 'HEADING && '' ' (3)		HEADING & '
TITLE 'HEADING FOUR' FIVE' (4)		HEADING FOUR
TITLE 'HEADING FOUR'REMARKS **ERROR**		

H3C -- THE EJECT INSTRUCTION

Purpose

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

Specifications

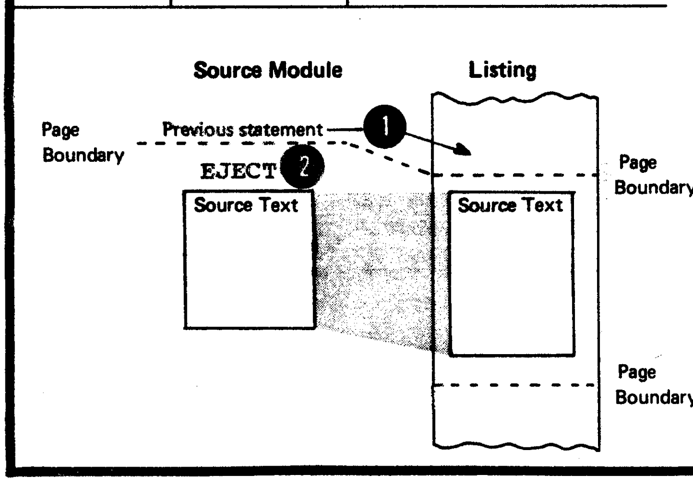
The format of the EJECT instruction statement is shown in the figure to the right.

The EJECT instruction causes the next line of the assembly listing to be printed at the top of a new
1 page. If the line before the EJECT instruction appears at the bottom
2 of a page, the EJECT instruction 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.

EJECT

Name	Operation	Operand
A sequence symbol or blank	EJECT	Not required



H3D -- THE SPACE INSTRUCTION

Purpose

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.

Specifications

The format of the SPACE instruction statement is given in the figure to the right.

The operand entry specifies the number of lines to be left blank. A blank operand entry 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 listed.

SPACE

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

H4 - Punching Output Cards

The instructions described in this section produce punched cards as output from the assembly in addition to those produced for the object module (object deck).

H4A -- THE PUNCH INSTRUCTION

Purpose

The PUNCH instruction allows you to punch source or other statements into a single card. With this feature you can:

1. 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.

2. Code PUNCH statements in macro definitions to produce, for example, source statements in other computer languages or for other processing phases.

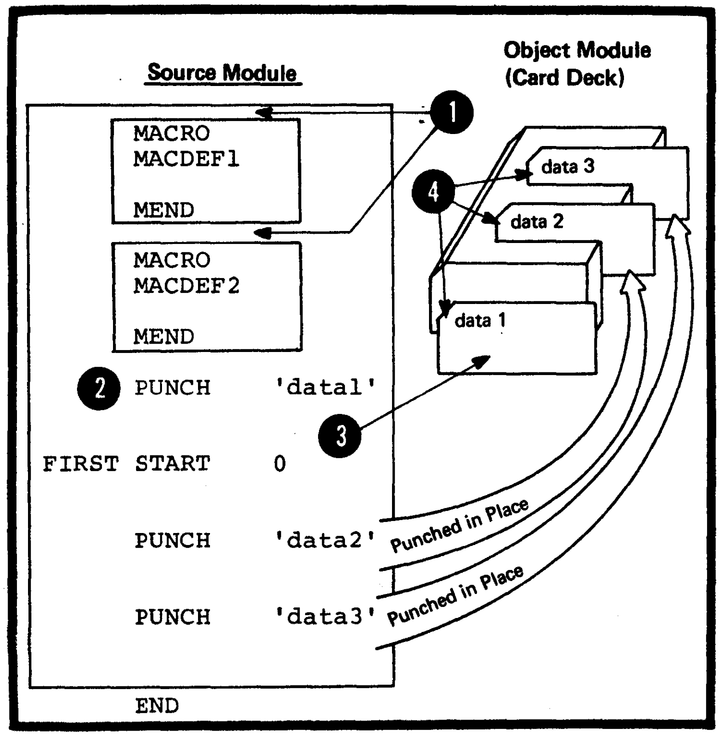
The card that is punched has a physical position immediately after the PUNCH instruction and before any other TXT cards of the object decks that are to follow.

Specifications

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 (1) except before and between source macro definitions. If a PUNCH instruction occurs (2) before the first control section, the (3) resultant card punched will precede all other cards in the object deck.

(4) 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.



The format of the PUNCH instruction statement is shown in the figure to the right.

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.

PUNCH

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

(1) The position of each character specified in the PUNCH statement (2) 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 (3) symbol and to substitute its current value.

(4) 2. Double ampersands or apostrophes are punched as single ampersands or apostrophes.

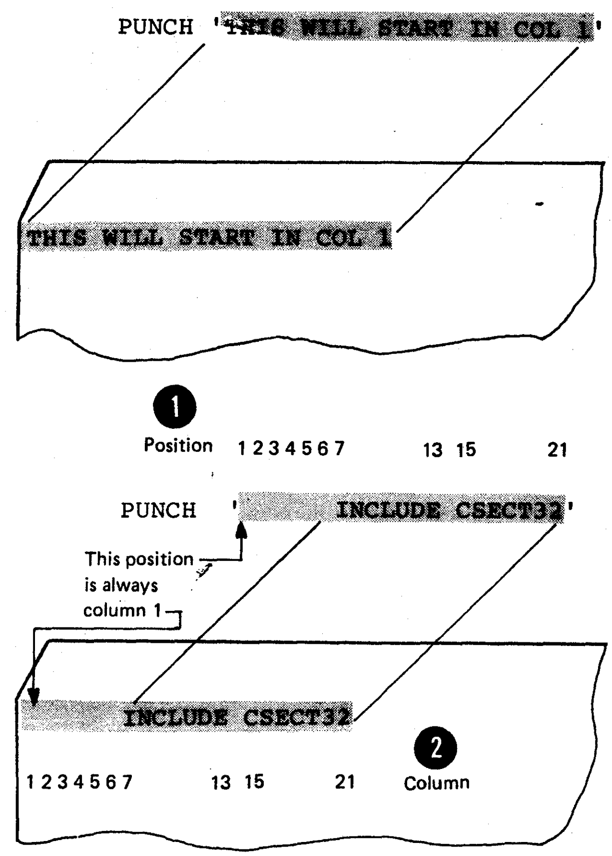
(5) 3. A single apostrophe followed by one or more blanks simply terminates the string of characters punched. If a non-blank character follows a single apostrophe, an error message is issued and nothing is punched.

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 in the EXEC statement of 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.



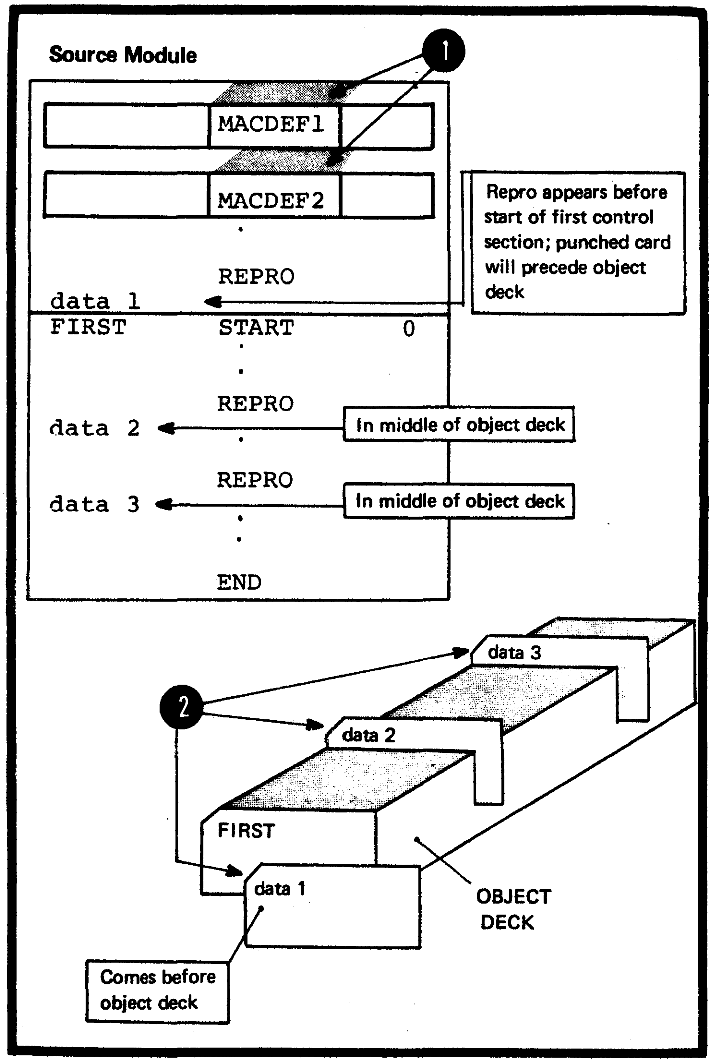
Examples:

Source Statement	Value of Variable Symbol	Characters Punched
(3) PUNCH 'CHARS &VAR'	ABC	CHARS ABC
(4) PUNCH 'CHARS &&'''		CHARS &'
PUNCH 'CHARS A' B'		CHARS A
(5) PUNCH 'CHARS A'REMARKS *** ERROR ***		
PUNCH 'CHARS A' REMARKS		CHARS A

H4B -- THE REPRO INSTRUCTION

Purpose

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.



Specifications

The REPRO instruction causes data on the statement line that follows it to be punched into the corresponding columns of a card. One REPRO instruction produces one punched card.

The REPRO instruction can appear anywhere in a source module except
1 before and between source macro
definitions. The punched cards 2
are not part of the object deck, even though they can be physically interspersed in the object deck.

The format of the REPRO instruction statement is shown in the figure to the right.

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.

REPRO

Name	Operation	Operand
A sequence symbol or blank	REPRO	Not required

NOTES:

1. No sequence numbers or identification is 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.

H5A -- THE OPSYN INSTRUCTION

Purpose

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

1. Machine and extended mnemonic branch instructions.

2. Assembler instructions including conditional assembly instructions.

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

Specifications

The OPSYN instruction must be written after the ICTL instruction and can be preceded only by the EJECT, ISEQ, PRINT, SPACE, and TITLE instructions. The OPSYN instruction must precede any source macro definitions that may be specified.

The OPSYN instruction has two basic formats as shown in the figure to the right.

(1) The operation code specified in the name field or the operand field (2) must represent either:

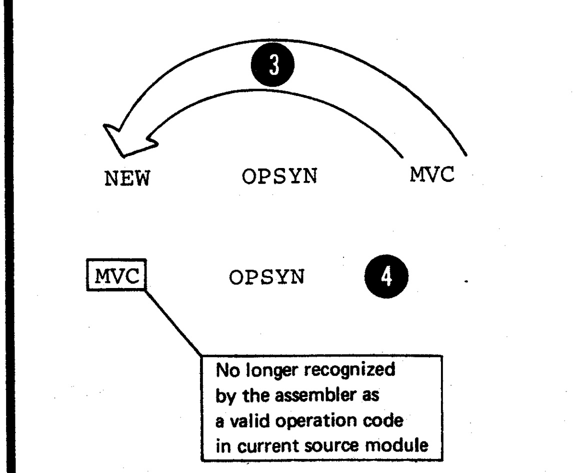
1. The operation code of one of the machine or assembler instructions as described in PARTS II, III, and PART IV of this manual, or

2. The operation code defined by a previous OPSYN instruction.

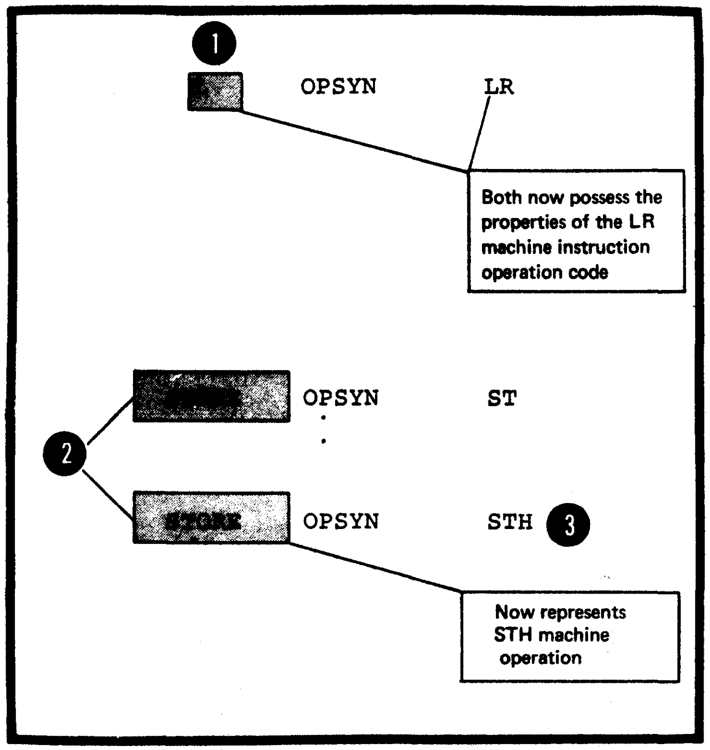
(3) The OPSYN instruction assigns the properties of the operation code specified in the operand field to the symbol in the name field. A (4) blank in the operand field causes the operation code in the name field to lose its properties as an operation code.

OPSYN

Name	Operation	Operand
(1) Any symbol or operation code	OPSYN	An operation code (2)
	or	
An operation code	OPSYN	blank



1 NOTE: The <u>symbol in the name field</u> 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. Further, when the same symbol appears in the <u>name</u>
2 <u>field of two OPSYN</u> instructions the <u>latest definition</u> takes
3 precedence.



H6 -- Saving and Restoring Programming Environments

OS only

The instructions described in this subsection can save and restore the status of PRINT options and the base register assignment of your program.

H6A -- THE PUSH INSTRUCTION

Purpose

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.

Specifications

The format of the PUSH instruction statement is shown in the figure to the right.

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.

PUSH

Name	Operation	Operand	Options
A sequence symbol or blank	PUSH	PRINT	1
		USING	2
		PRINT,USING	3
		USING,PRINT	4

H6B -- THE POP INSTRUCTION

Purpose

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

Specifications

The format of the POP instruction is given in the figure to the right.

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.

POP

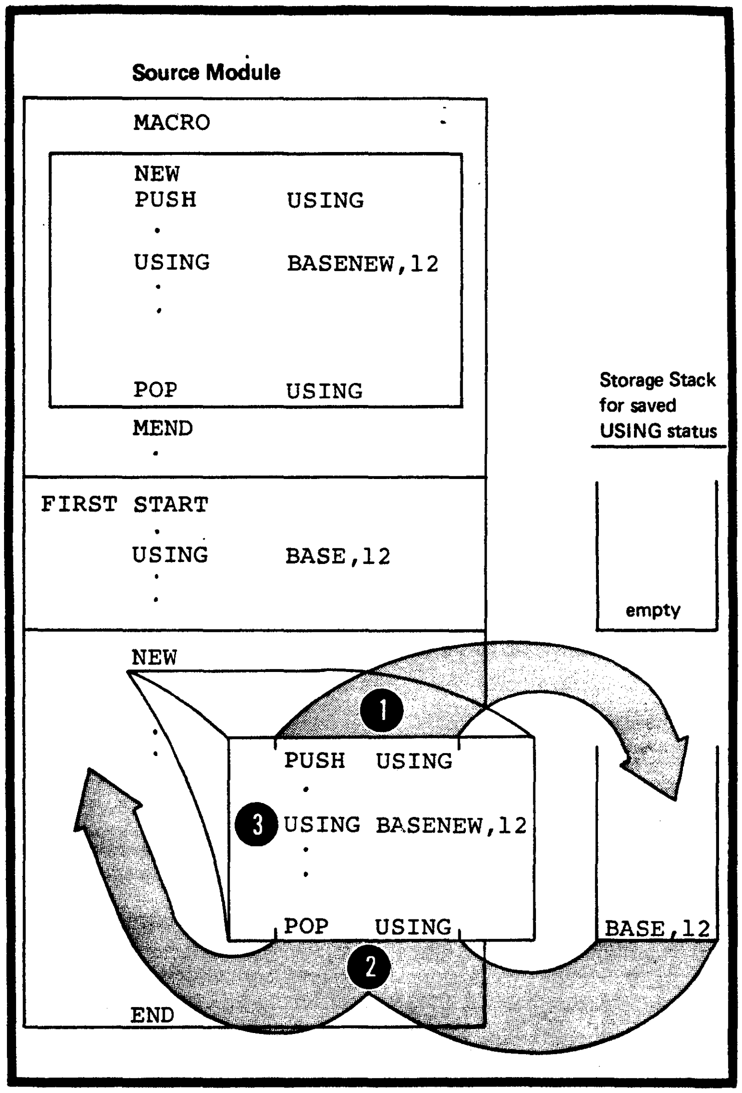
Name	Operation	Operand	Options
A sequence symbol or blank	POP	PRINT	1
		USING	2
		PRINT,USING	3
		USING,PRINT	4

H6C -- COMBINING PUSH AND POP

OS only

In the opposite example, you can see how the USING environment is (1) saved and restored (2) by a combination of PUSH and POP instructions.

NOTE: The PUSH instruction does not change the current USING status; (3) you must do this yourself.



Part IV: The Macro Facility

This page left blank intentionally.

Section I: Introducing Macros

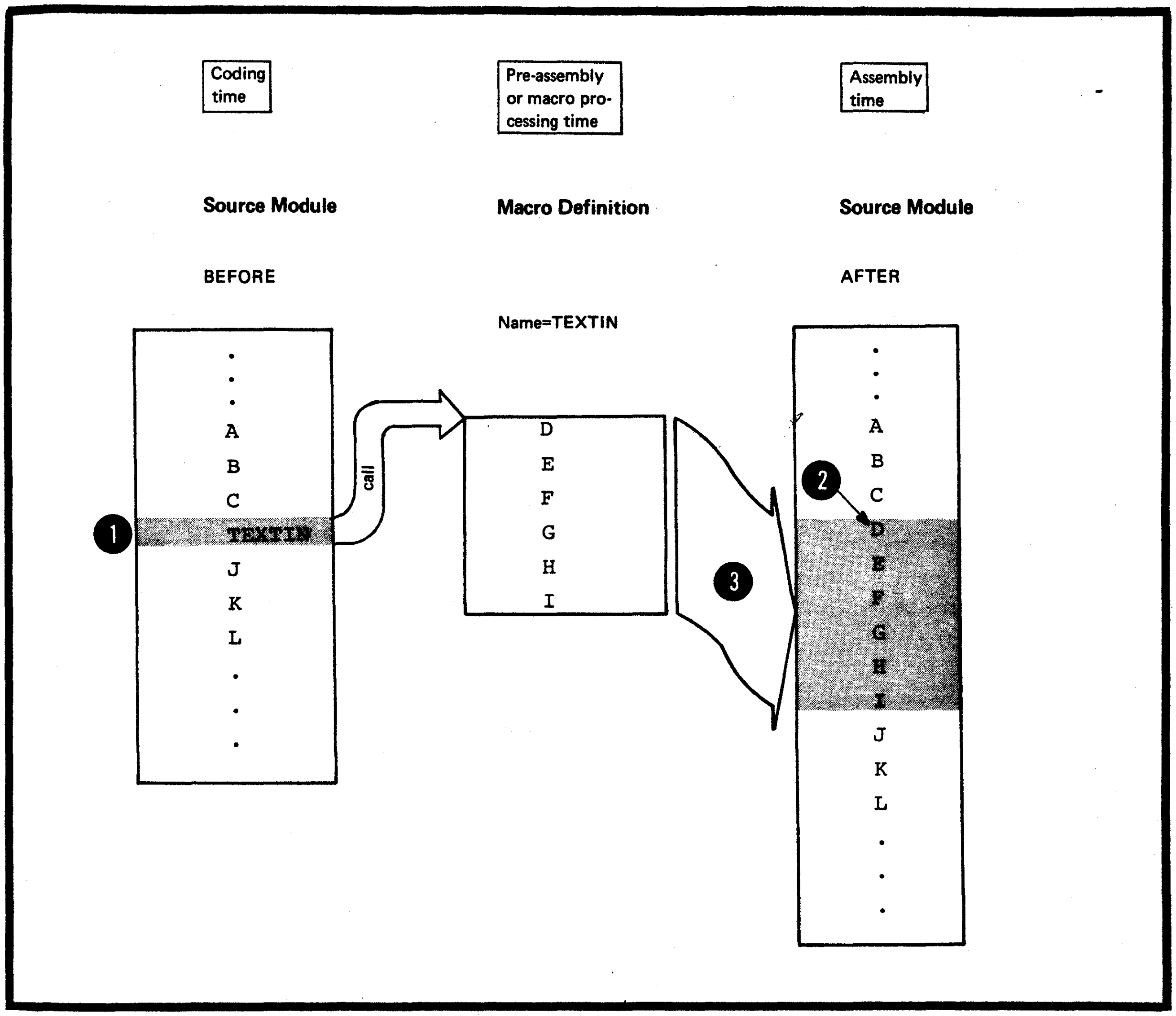
This section introduces the basic macro concept; what you can use the macro facility for, how you can prepare your own macro definitions, and how you call these macro definitions for processing by the assembler.

Read this section straight through before referring to the detailed descriptions identified by the cross-reference arrows.

NOTE: IBM supplies macro definitions in system libraries for input/output and other control program services, such as the dynamic allocation of main storage areas. To process these macro definitions you only have to write the macro instruction that calls the definition.

Using Macros

FOR TEXT INSERTION: The main use of macros is to insert assembler language statements into a source program.



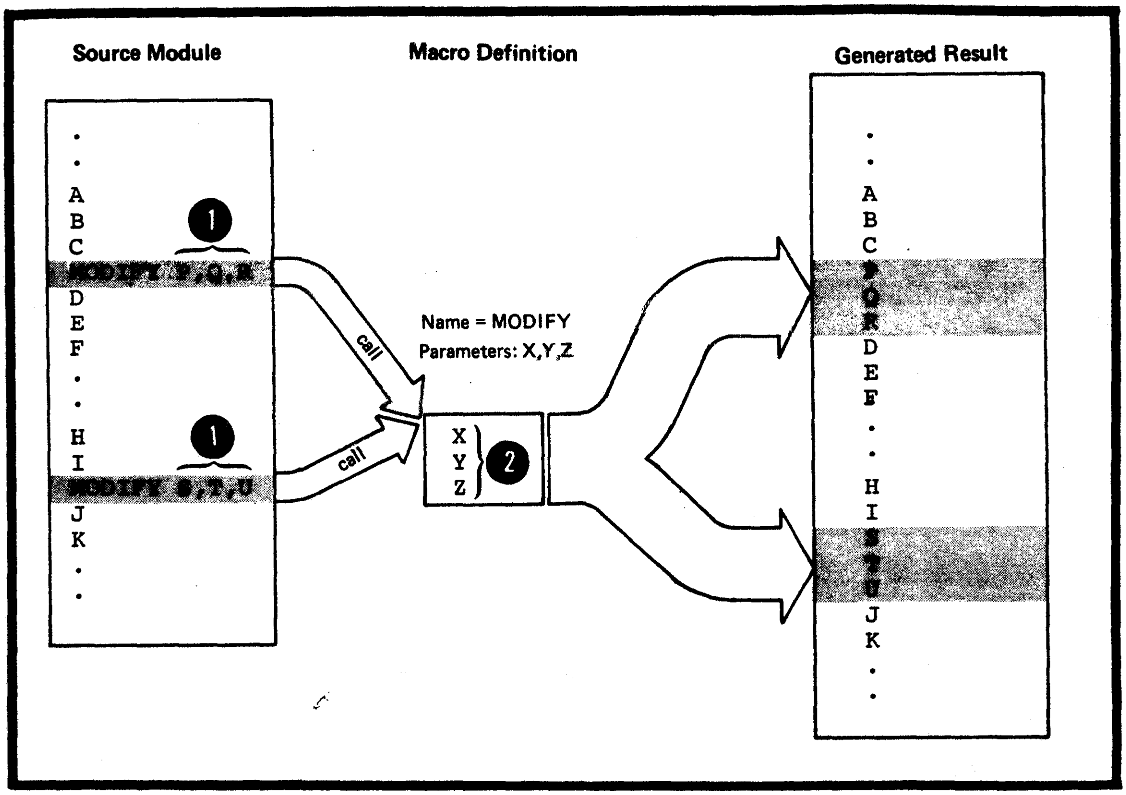
❶ You call a named sequence of statements (the macro definition) by using a macro instruction, or macro call. The assembler replaces the macro call by the statements from the macro definition and inserts them into the source ❷ module at the point of call. The process of inserting ❸ the text of the macro definition is called macro generation or macro expansion. The assembler expands a macro at pre-assembly 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 intc object code.

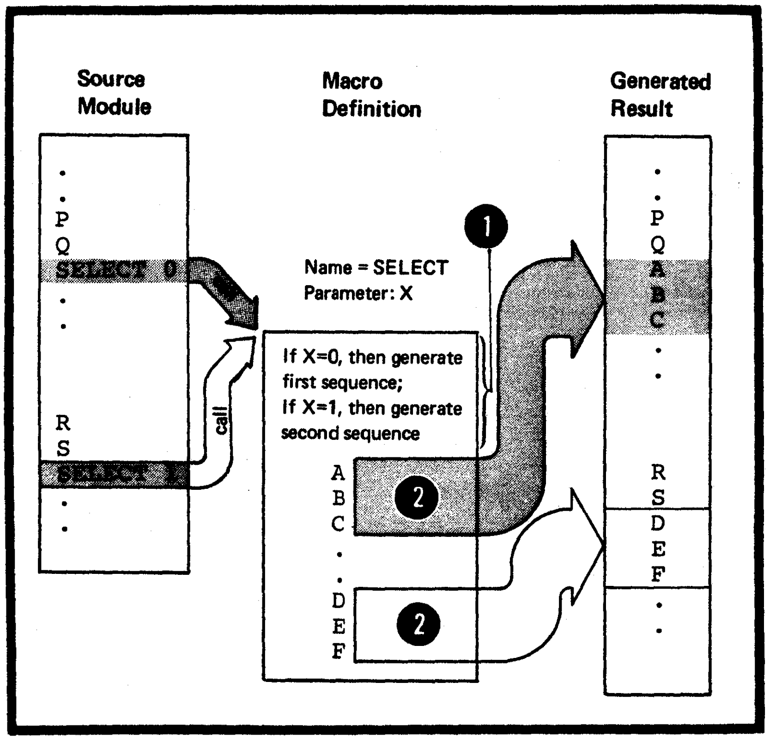
FOR TEXT MODIFICATION: You may want to modify the statements in a macro definition before they are generated.

1 2 You can do this by supplying character string values as operands in a macro call. These values replace parameters in the statement to be generated. This means that you can change the content of the generated statements each time you call the macro definition.

1 See K 3
2 See J 3



FOR TEXT MANIPULATION: You can also select and reorder the statements to be generated from a macro definition by using the conditional assembly language described later in this section.



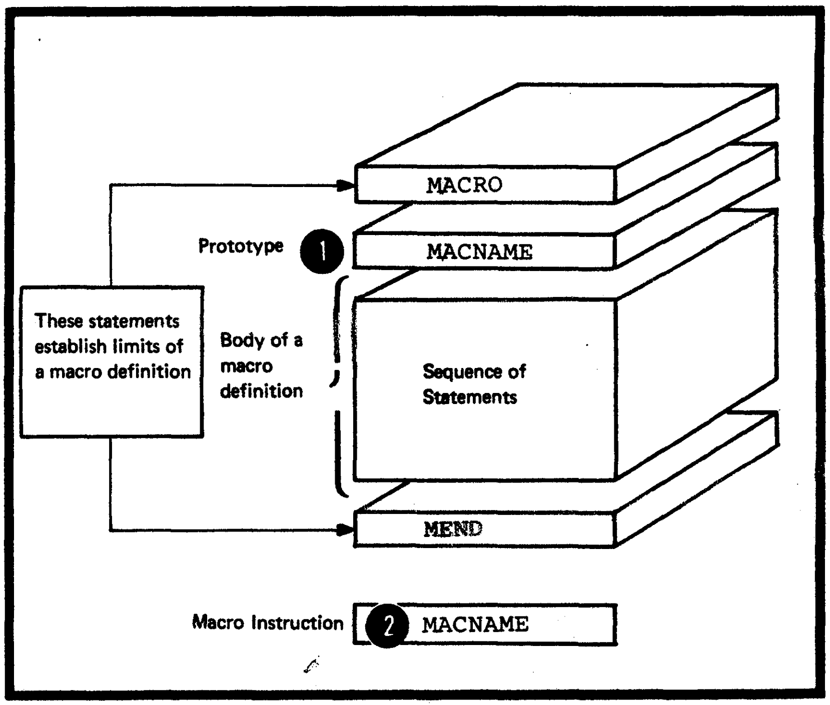
❶ The conditional assembly language allows you to manipulate text generation, for example, by branching upon the result of a condition test. You can choose exactly which ❷ statements will or will not be generated by varying the values you specify in the macro call.

❶ See SECTION L

The Basic Macro Concept

To use the complete macro facility provided by the assembler you must:

- Prepare a macro definition and
- Call this definition using a macro instruction.



You can create a macro definition by enclosing any sequence of assembler language statements between the MACRO and (1) MEND statements, and by writing a prototype statement in which you give your definition a name. This name is then (2) the operation code that you must use in the macro instruction to call the definition.

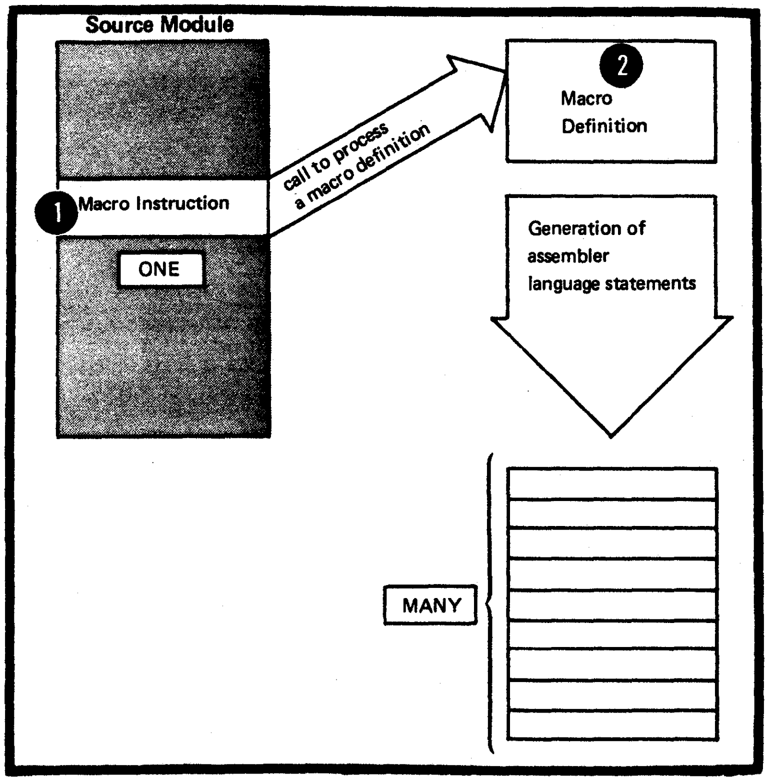
(1) See J2C

(2) See K2B

(1) When you write a macro instruction in your source module, you tell the assembler to process a particular macro (2) definition. The assembler produces assembler language statements from this macro definition for each macro instruction that calls the definition.

(1) See SECTION K

(2) See SECTION J



By using the macro facility you reduce programming effort, because:

1. You write and test the code a macro definition contains once. You and other programmers can then use the same code as often as you like by calling the definition; which means that you do not have to reconstruct the coding logic each time you use the code.

2. You need write only one macro instruction to call for the generation of many assembler language statements from the macro definition.

When you are designing and writing large assembler language programs, the above features allow you to:

- Prepare macro definitions, containing difficult code, for your less experienced colleagues. They can then call your definitions to generate the appropriate statements, without having to learn the code in the definition.

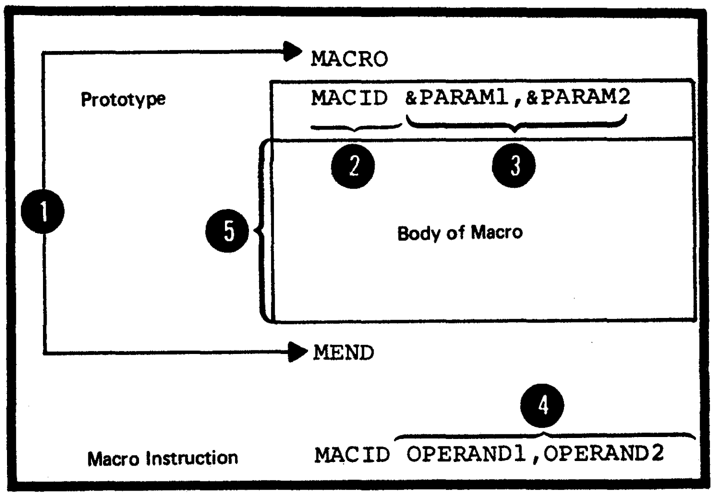
- Change the code in one place when updating or making corrections, that is, in the macro definition. Each call gets the latest version automatically, thus providing standard coding conventions and interfaces.

- Describe the functions of a complete macro definition rather than the function of each individual statement it contains, thus providing more comprehensible documentation for your source module.

Defining a Macro

Defining a macro means preparing the statements that constitute a macro definition. To define a macro you must:

1. Give it a name
2. Declare any parameters to be used
3. Write the statements it contains.
4. Establish its boundaries



(1) The MACRO and MEND instructions establish the boundaries of a macro definition.

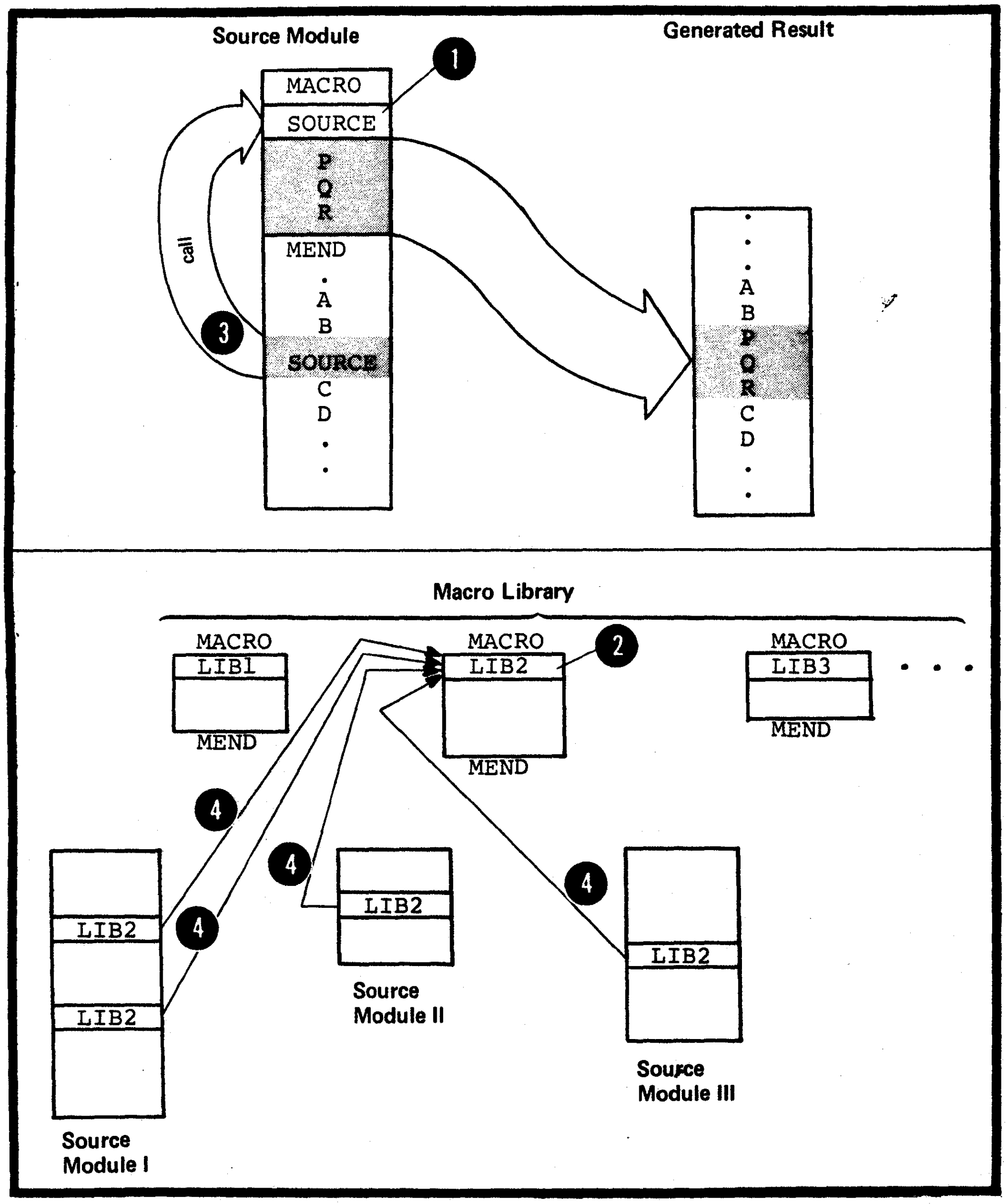
(2)(3)(4) You use the prototype statement to name the macro and to declare its parameters. In the operand field of the macro instruction, you can assign values to the parameters declared for the called macro definition.

(5) The body of a macro definition 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 cther processing statements.

(2) See J2D
(3) See J3
(4) See K2C
(5) See J2E

WHERE YOU CAN PLACE A MACRO DEFINITION: You can include a macro definition at the beginning of a source module. ❶ This type of definition is called a source macro definition. ❶ See J 1 B

You can also insert a macro definition in a system or user library (located, for example, on disk) by using the appropriate utility program. This type of definition is ❷ called a library macro definition. The IBM-supplied macro definitions mentioned earlier are examples of library macro definitions.

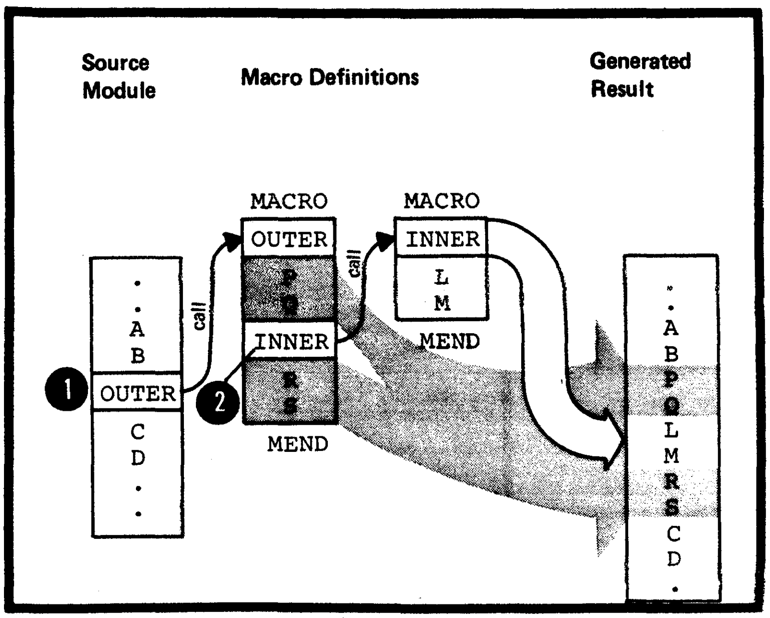


Calling a Macro

❸ You can call a source macro definition only from the source ❹ module in which it is included. You can call a library macro definition from any source module.

1 WHERE YOU CAN CALL A MACRO DEFINITION: You can call a macro definition by specifying a macro instruction anywhere in a source module, except before or between any source macro definitions that may be specified.

1 See K 1 B

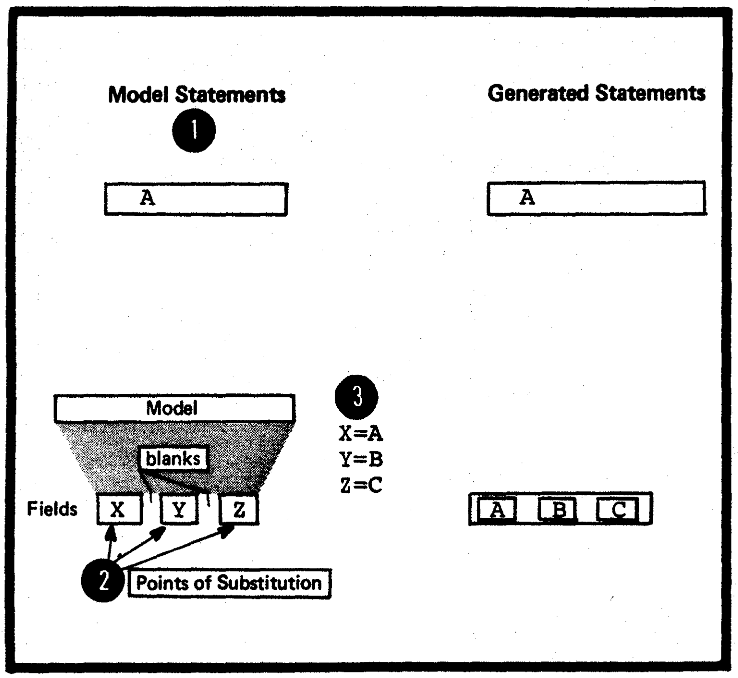


2 You can also call a macro definition from within another macro definition. This type of call is an inner macro call; it is said to be nested in the macro definition.

2 See K6 A

The Contents of a Macro Definition

The body of a macro definition can contain a combination of model statements, processing statements, and comments statements.



1 MODEL STATEMENTS: You can write assembler language statements as model statements. The assembler copies them exactly as they are written when it expands the macro.
2 You can also use variable symbols as points of substitution
3 in a model statement. The assembler will enter values in place of these points of substitution each time the macro is called.

1 See J4

The three types of variable symbol in the assembler language are

1. Symbolic parameters, declared in the prototype statement

2. System variable symbols (see J7)

3. SET symbols, which are part of the conditional assembly language (see L1A).

The assembler processes the generated statements, with or without value substitution, at assembly time.

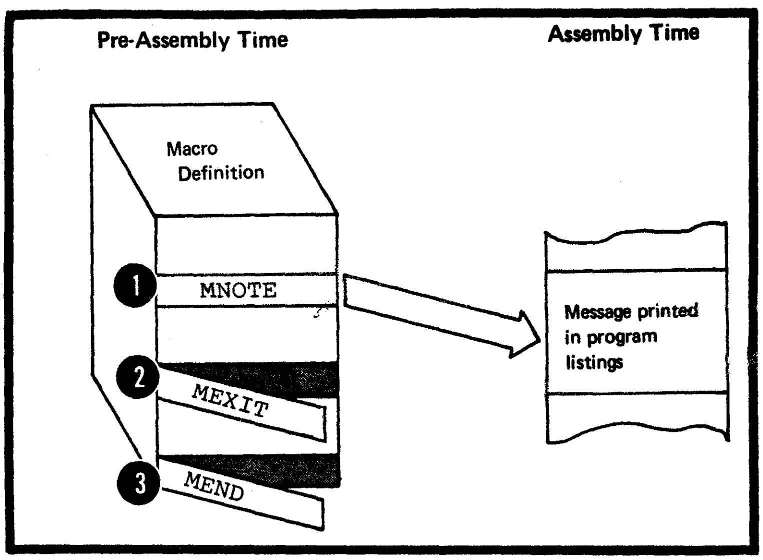
PROCESSING STATEMENTS: Processing statements perform functions at pre-assembly time when macros are expanded, but they are not themselves generated for further processing at assembly time. The processing statements are:

1. Conditional assembly instructions

2. Inner macro calls

3. The MNOTE instruction

4. The MEXIT instruction.



❶ 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 pre-assembly computation.

❶ See J5D

❷ 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.

❷ See J5E

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

COMMENTS STATEMENTS: One type of comments statement describes pre-assembly operations and is not generated. The other type describes assembly-time operations and is therefore generated (for details see J6).

The Conditional Assembly Language

The conditional assembly language is a programming language with most of the features that characterize such a language. For example, it provides:

1. Variables

2. Data attributes

3. Expression computation

4. Assignment instructions

5. Labels for branching

6. Branching instructions

7. 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 fully described in Section L.

Section J: The Macro Definition

This section describes macro definitions: where they can be placed in order to be available to call, how they are specified, and what they can contain.

J1 -- Using a Macro Definition

J1A -- PURPOSE

A macro definition is a named sequence of statements which 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:

1. Copied directly from the definition,

2. Modified by parameter values before generation, or

3. 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 do not generate assembler language statements, but perform only internal processing, like some of the macro definitions used for system generation.

Where to Define a Macro In a Source Module

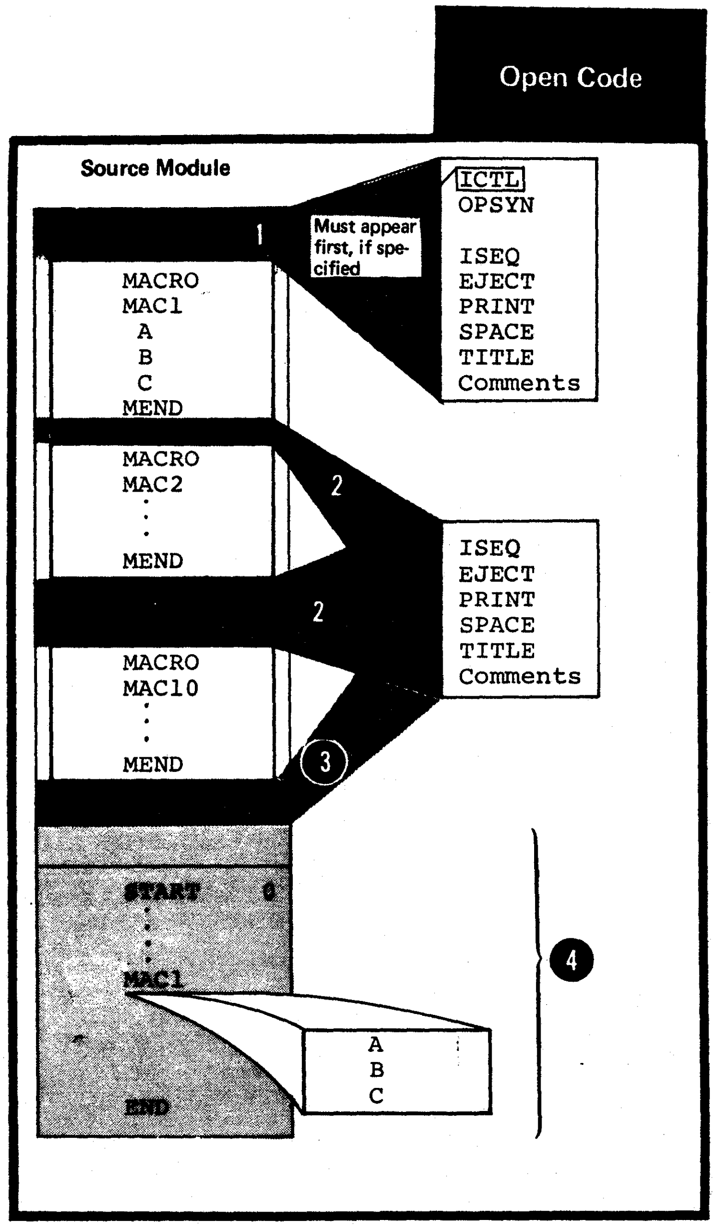
A macro definition within a source module must be specified at the beginning of that source module. This type of macro definition is called a source macro definition. A macro definition can also reside in a system library; this type of macro is called a library macro definition. Either type can be called from the source module by the appropriate macro instruction.

NOTE: A source macro definition can be entered into a library and thereby become a library macro definition. A library macro definition can be included at the beginning of a source module and thereby become a source macro definition.

Some control and comments statements can appear at the beginning of a source module along with the source macro definitions. They can be used:

1. Before all macro definitions.
2. Between macro definitions.
3. After macro definitions and before open code

All other statements of the assembler language must appear after any source macro definitions that are specified.



Open Code

4. Open code is that part of a source module that lies outside of and after any source macro definition. Open code is initiated by any statement of the assembler language that appears outside of a macro definition, except the ICTL, OPSYN, ISEQ, EJECT, PRINT, SPACE, or TITLE instruction, or a comments statement.

At coding time, it is important to distinguish between source statements that lie in open code and those that lie inside macro definitions.

NOTES:

1. The ISEQ, EJECT, PRINT, SPACE, and TITLE instructions, and one or more comments statements, can appear between source macro definitions and the start of open code. However, in this position, the above instructions must not contain any variable symbols.

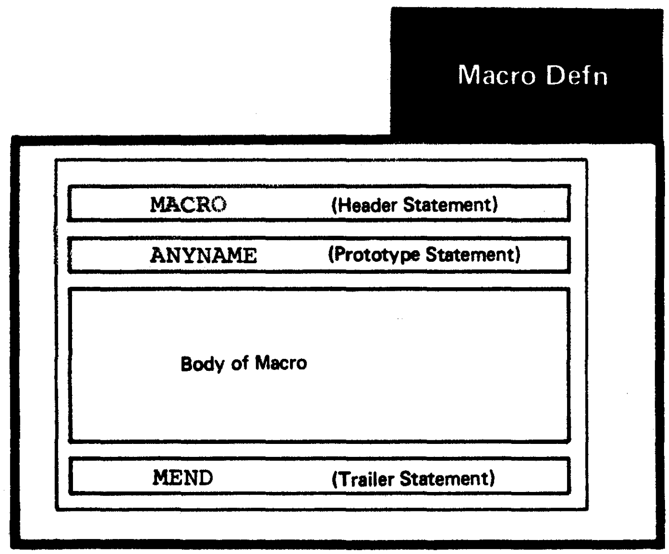
2. After the start of open code, variable symbols are allowed in any statement.

3. A macro definition must not be specified after the start of open code.

The Format of a Macro Definition

The general format of a macro definition is shown in the figure to the right.

The four parts are described in detail below.



J2 -- Parts of a Macro Definition

J2A -- THE MACRO DEFINITION HEADER

Purpose

The macro definition header instruction indicates the beginning of a macro definition.

Specifications

The MACRO instruction is the macro definition header; it must be the first statement of every macro definition. Its format is given in the figure to the right.

Header

Name	Operation	Operand
Not used, must not be present	MACRO	Not required

J2B -- THE MACRO DEFINITION TRAILER

Purpose

The macro definition trailer instruction indicates the end of a macro definition. It also provides an exit when it is processed during macro expansion.

Specifications

The MEND instruction statement is the macro definition trailer; it must be the last statement of every macro definition. Its format is given in the figure to the right.

Trailer

Name	Operation	Operand
A sequence symbol, or not used	MEND	Not required

J2C -- THE MACRO PROTOTYPE STATEMENT: CODING

Purpose

The prototype statement in a macro definition serves as a model (prototype) of the macro instruction you use to call the macro definition.

Specifications

The prototype statement must be the second statement in every macro definition. It comes immediately after the MACRO instruction.

The format of the prototype statement statement is given in the figure to the right.

The maximum number of symbolic parameters allowed in the operand field is not fixed. It depends on the amount of virtual storage available to the program.

[redacted]

If no parameters are specified in the operand field, remarks are allowed, if the absence of the operand entry is indicated by a comma preceded and followed by one or more blanks.

Prototype

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

Alternate Ways of Coding the Prototype Statement

The prototype statement can be specified in one of the following three ways:

1 The normal way, with all the symbolic parameters preceding any remarks.

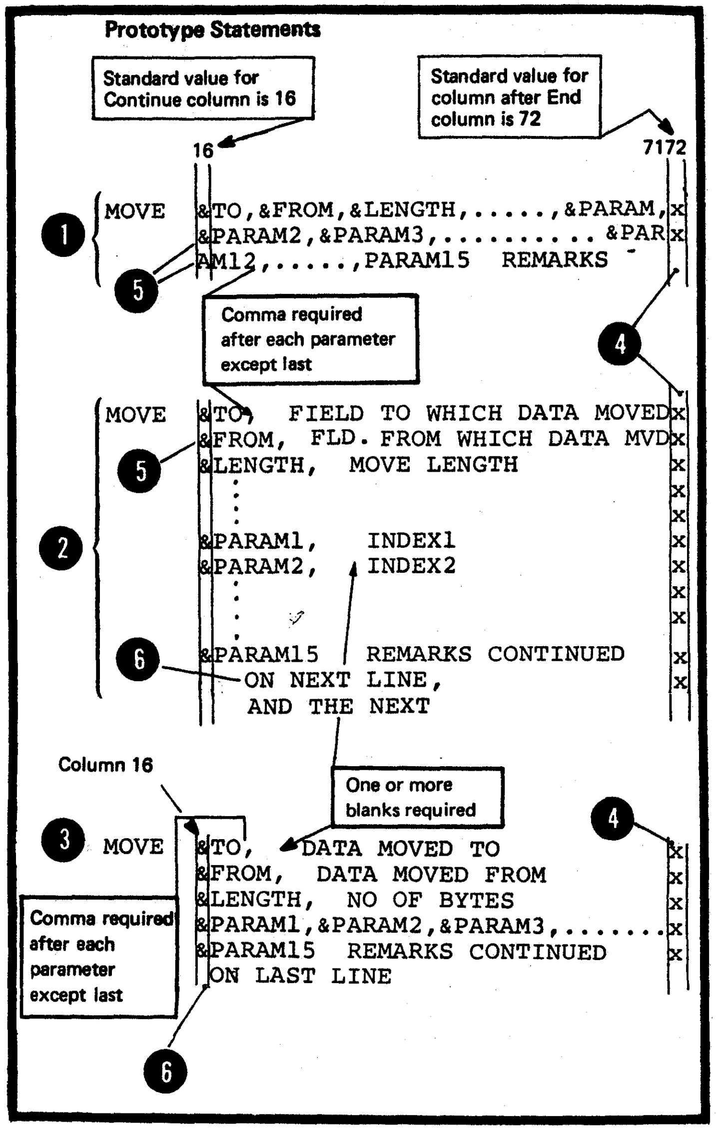
2 An alternate way, allowing remarks for each parameter.

3 A combination of the first two ways.

NOTES:

1. Any number of continuation lines is allowed. However, each
4 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
5 parameters) must begin in the continue column otherwise the whole line and any lines that follow will be
6 considered to contain remarks.



J2D -- THE MACRO PROTOTYPE STATEMENT: ENTRIES

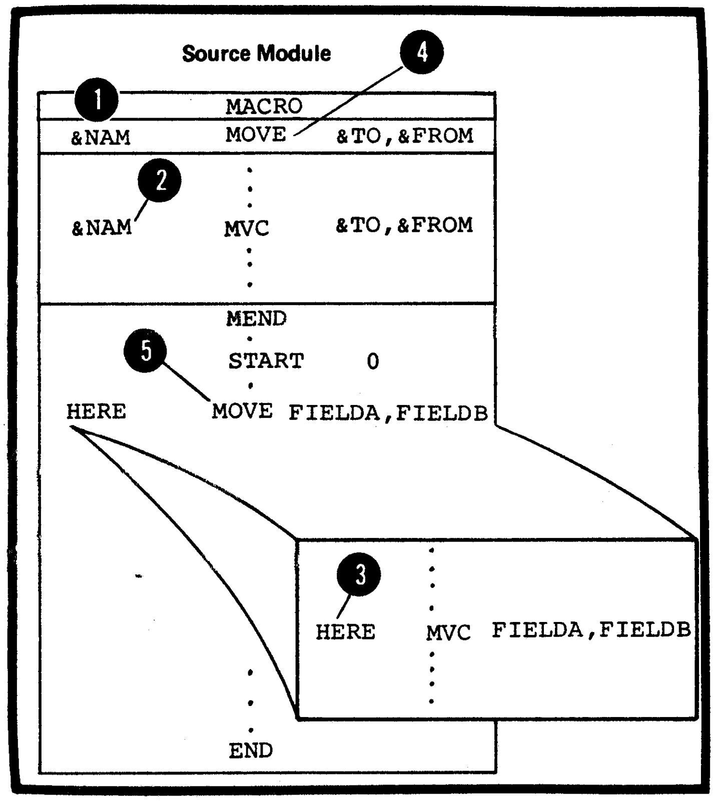
The Name Entry

Purpose

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.

Specifications

1 2 3 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 K2A.



The Operation Entry

Purpose

The operation entry is a symbol that identifies the macro definition. When you specify it in the operation field of a macro instruction, the appropriate macro definition is called and processed by the assembler.

Specifications

4 The symbol in the operation field of the prototype statement establishes the name by which a macro definition must be called.
5 This name becomes the operation code required in any macro instruction that calls the macro.

OS only **NOTE: Unless operation codes have been changed by the OPSYN** instruction, the operation code specified in the prototype statement must not be the same as that specified in:

1. A machine instruction.

2. An assembler instruction.

3. The prototype statement of another source (or library) macro definition.

The Cperand Entry

Purpose

The operand entry 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.

Specifications

The operands of the macro prototype statement must be symbolic parameters separated by commas. They can be positional parameters or keyword parameters or both (see J3).

NOTE: The operands must be symbolic parameters; parameters in sublists are not allowed. For a discussion of sublists in macro instruction cperands, see K4.

J2E -- THE BODY OF A MACRO DEFINITION

Purpose

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.

Specifications

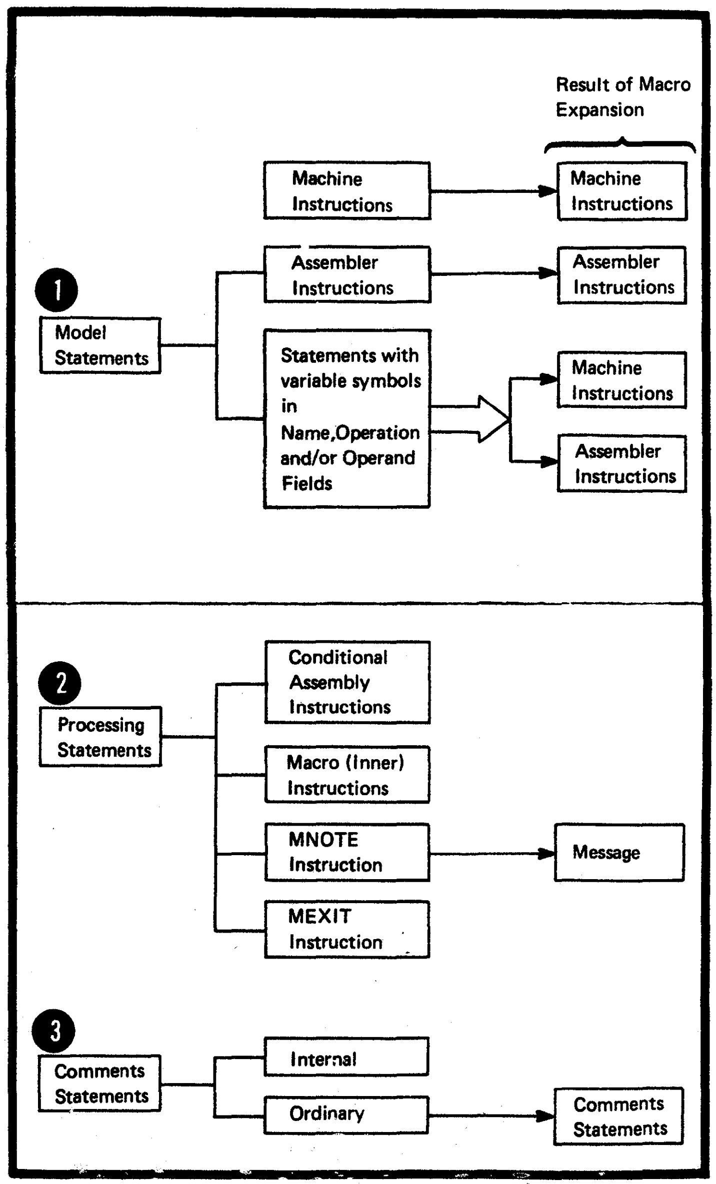
The statements in the body of a macro definition must appear between the macro prototype statement and the MEND statement of the definition. The three main types of statements allowed in the body of a macro are:

1 • Model statements (see J4),

2 • Processing statements (see J5), and

3 • Comments statements (see J6).

NOTE: The body of a macro definition can be empty, that is, contain no statements.

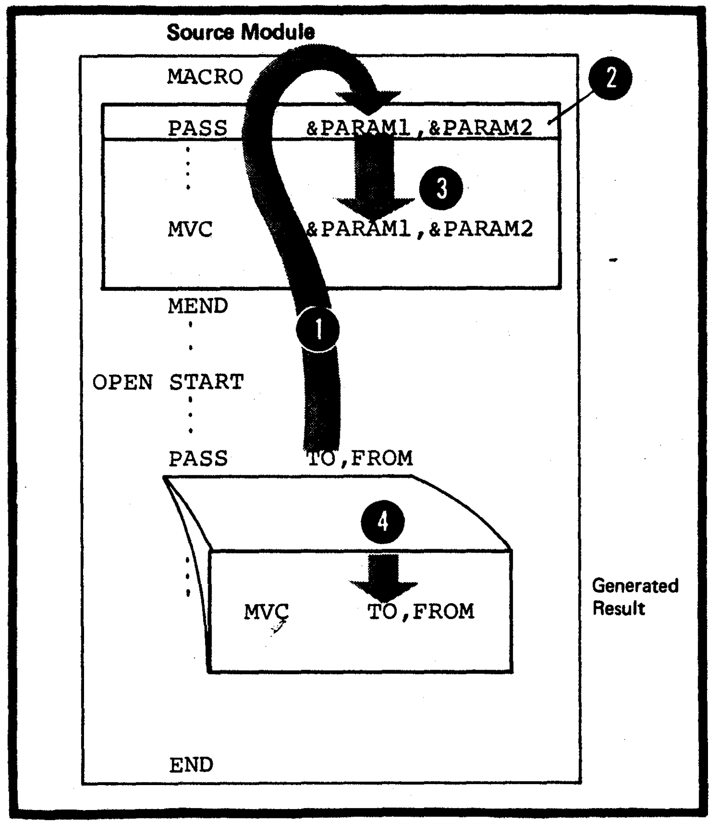


J3 — Symbolic Parameters

Purpose

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.

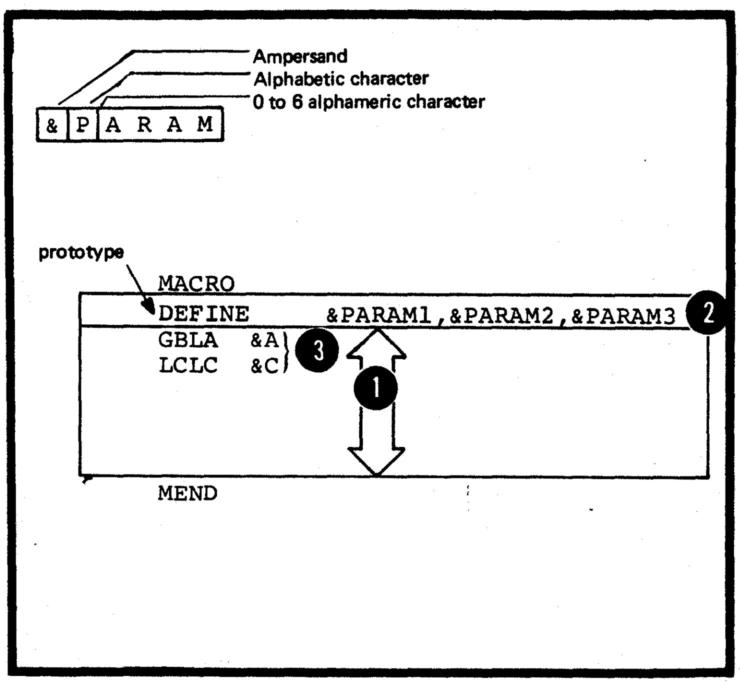


General Specifications

Symbolic parameters must be valid variable symbols, as shown in the figure to the right.

They 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 be multiply defined or identical to any other variable symbols within the given local scope. This applies to the system variable symbols described in J7, and local and global SET symbols described in L1A.

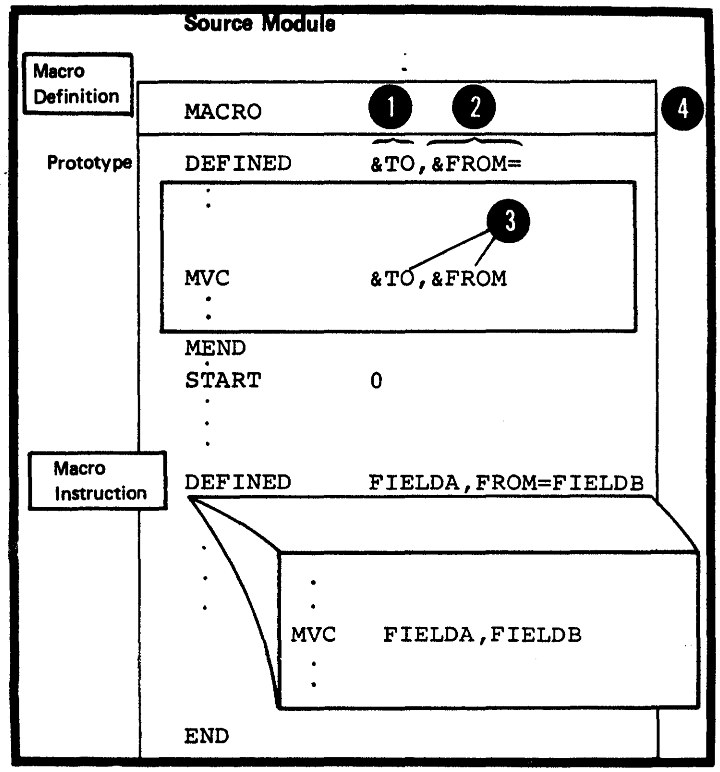


The two kinds of symbolic parameters are:

1 • Positional parameters

2 • Keyword parameters.

3 Each positional or keyword parameter used in the body of a macro definition must be declared in the
4 prototype statement.

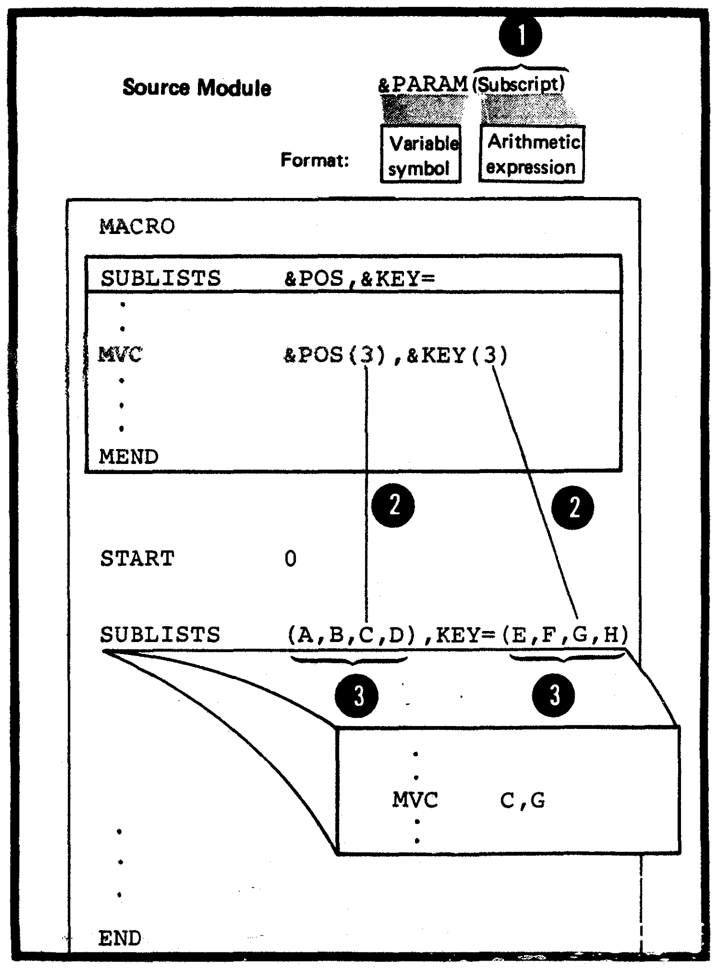


Subscripted Symbolic Parameters

Subscripted symbolic parameters must be coded in the format shown in the figure to the right.

1 The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (arithmetic expressions are discussed in L4A). The arithmetic expression can contain subscripted variable symbols. Subscripts can be nested up to 5 levels of nesting.

The value of the subscript must be greater than or equal to one.
2 The subscript indicates the position of the entry in the sublist that
3 is specified as the value of the subscripted parameter (sublists as values in macro instruction operands are fully described in K4).



J3A -- POSITIONAL PARAMETERS

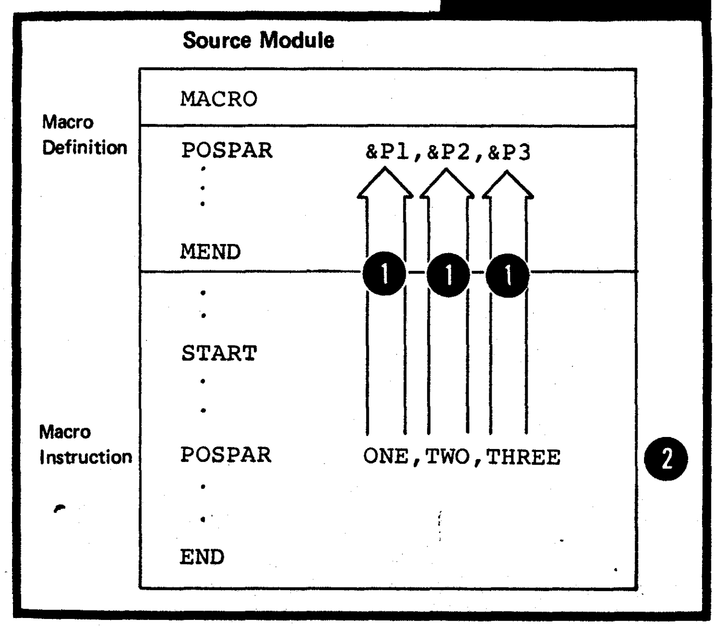
Purpose

You should use a positional parameter in a macro definition if you wish 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 wish the parameter to have in the proper position in the operand of the calling macro instruction.

For keyword (described below) parameters, 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.

Specifications

The general specifications for symbolic parameters described in J3 also apply to positional parameters. Note that the specification for each positional parameter declared in the prototype statement definition must be a valid
1 variable symbol. Values are assigned to the positional parameters by
2 the corresponding positional operands specified in the macro instruction that calls the definition.



J3B -- KEYWORD PARAMETERS

Purpose

You should use a keyword parameter in a macro definition for a value that changes infrequently. By specifying a standard default value to be assigned to the keyword parameter, you can omit the corresponding keyword operand in the calling macro instruction.

Keyword parameters are also convenient because:

1. You can specify the corresponding keyword operands in any order in the calling macro instruction.

2. The keyword, repeated in the operand, reminds you which parameter is being given a value and for which purpose the parameters is being used.

Specifications

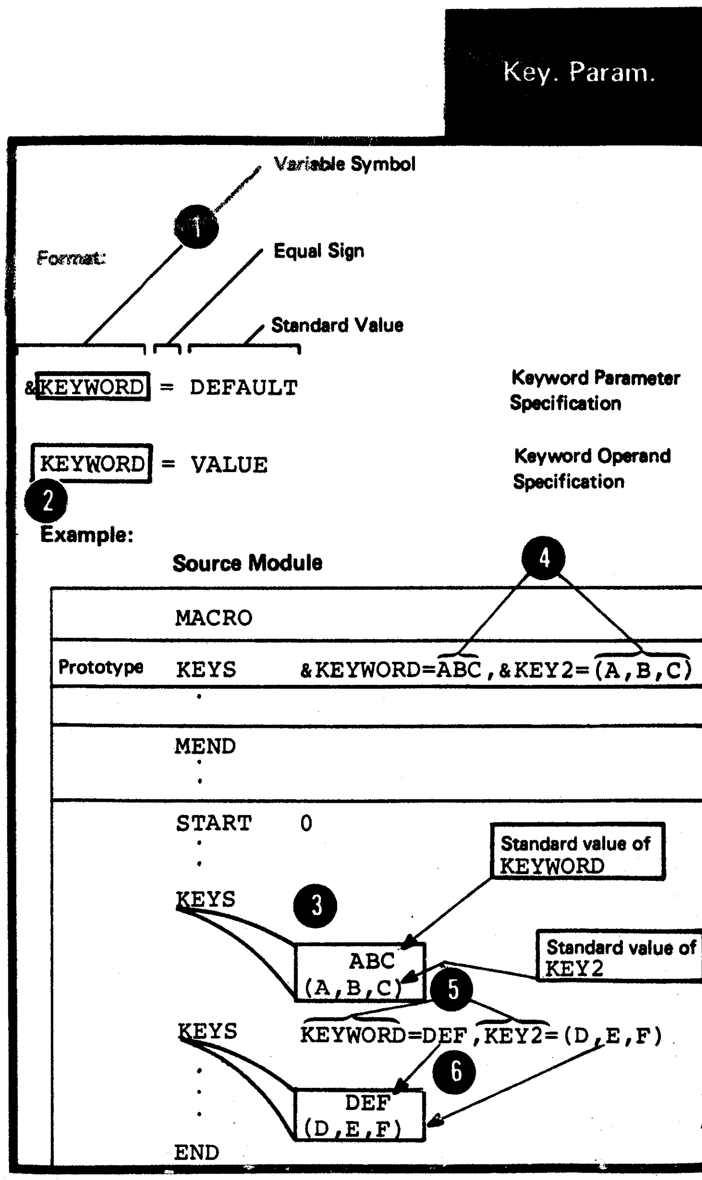
The general specifications for symbolic parameters described in J3 also apply to keyword parameters. Each keyword parameter must be in the format shown in the figure to the right.

1 The actual parameter must be a valid variable symbol.

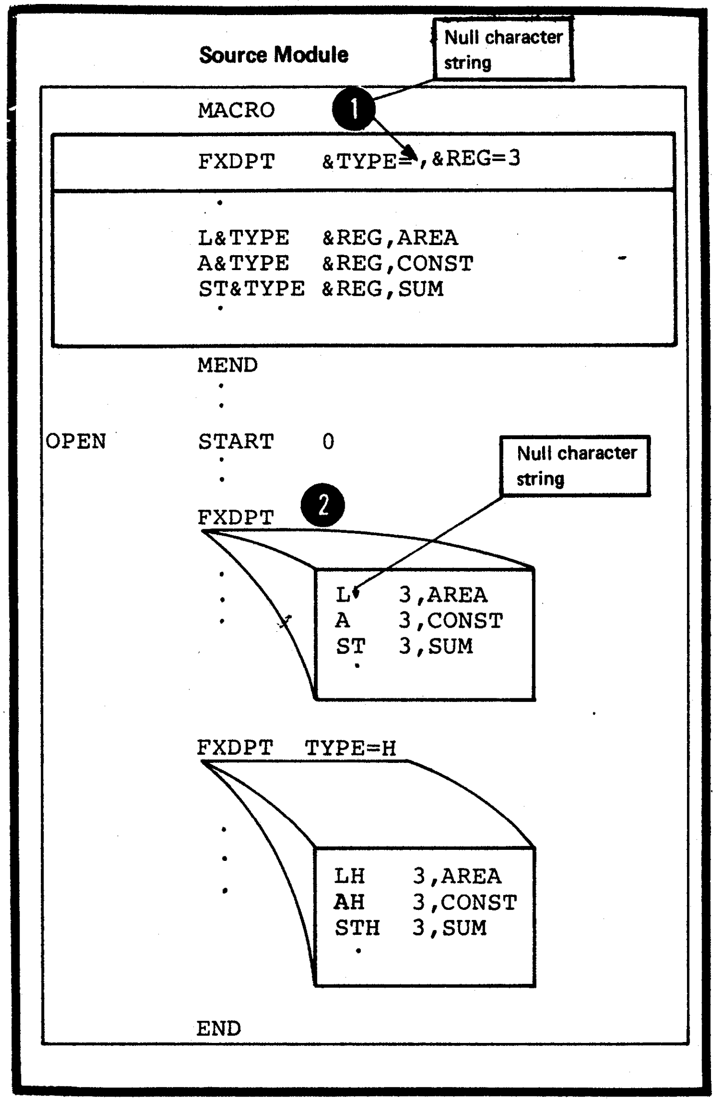
2 A value is assigned to a keyword parameter by the corresponding keyword operand through the name of the keyword as follows:

3 4 1. If the corresponding keyword operand is omitted, the standard value specified in the prototype statement becomes the value of the parameter for that call (for full details on values passed see K5).

5 6 2. If the corresponding keyword operand is specified, the value after the equal sign overrides the standard value in the prototype and becomes the value of the parameter for that call (see K5).



❶ NOTE: A null character string can be specified as the standard value of a keyword parameter, and will be generated if the corresponding ❷ keyword operand is omitted.



J3C -- COMBINING POSITIONAL AND KEYWORD PARAMETERS

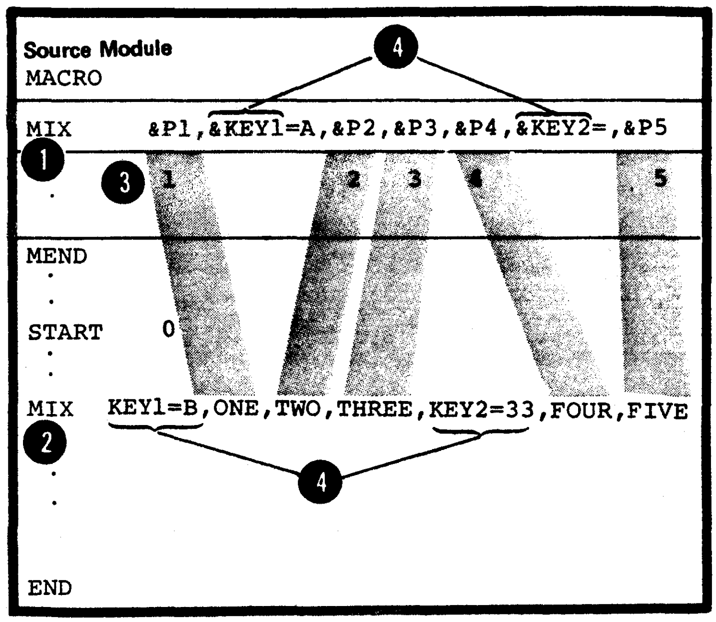
Purpose

By using positional and keyword parameters in a prototype statement, you combine the benefits of both. You can use positional parameters in a macro definition for passing values that change frequently and keyword parameters for passing values that do not change often.

Specifications

(1) Positional and keyword parameters can be mixed freely in the macro prototype statement. The same applies to the positional and keyword (2) operands of the macro instruction (see K3C). Note, however, that (3) the order in which the positional parameters appear determines the order in which the positional operands must appear. Interspersed (4) keyword parameters or operands do not affect this order.

DOS All positional parameters must precede any keyword parameters, if specified. The same applies to positional and keyword operands of a macro instruction (see K3C).



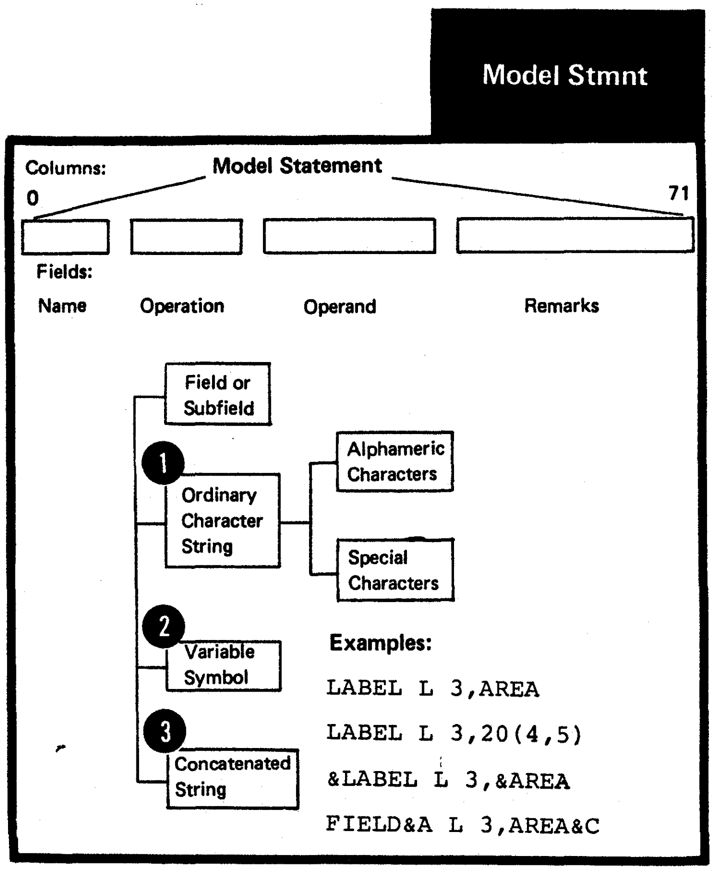
J4 - Model Statements

J4A -- PURPOSE

Model statements are statements from which assembler language statements are generated at pre-assembly 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 content of the statements generated from that model statement. You can also use model statements into which you substitute values in open code.

J4B -- SPECIFICATIONS

The following specifications also apply to model statements in open code. Exceptions are noted where applicable.



Format of Model Statements

A model statement consists of one or more fields separated by one or more blanks.

Each field or subfield can consist of:

1. An ordinary character string
2. A variable symbol as a point of substitution
3. Any combination of ordinary character strings and variable symbols to form a concatenated string.

The statements generated at pre-assembly 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 Section B or they will be flagged as an error at assembly time.

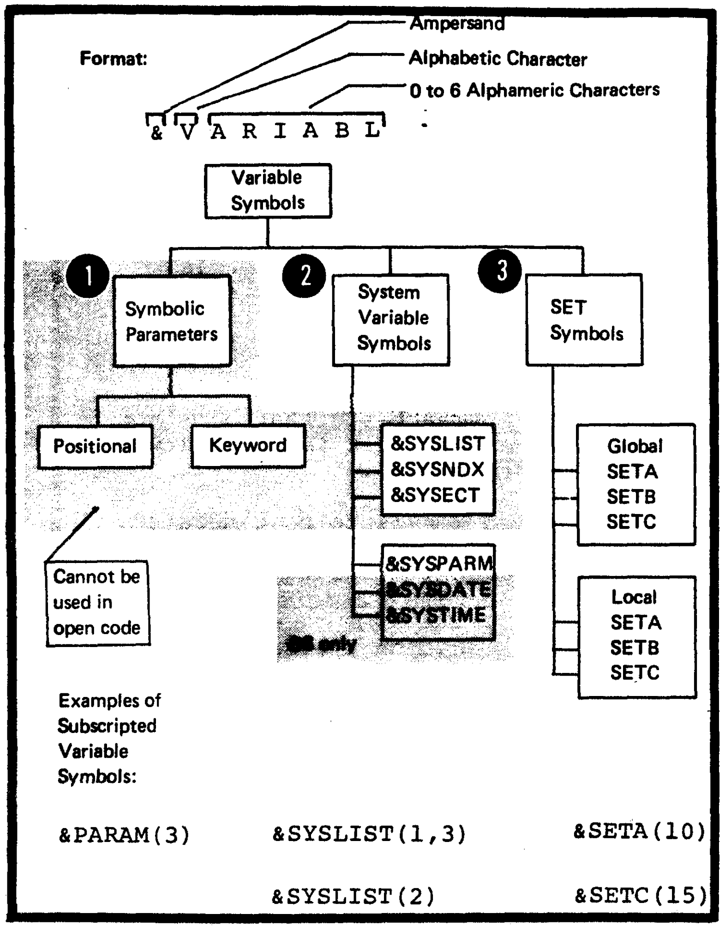
Variable Symbols as Points of Substitution

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:

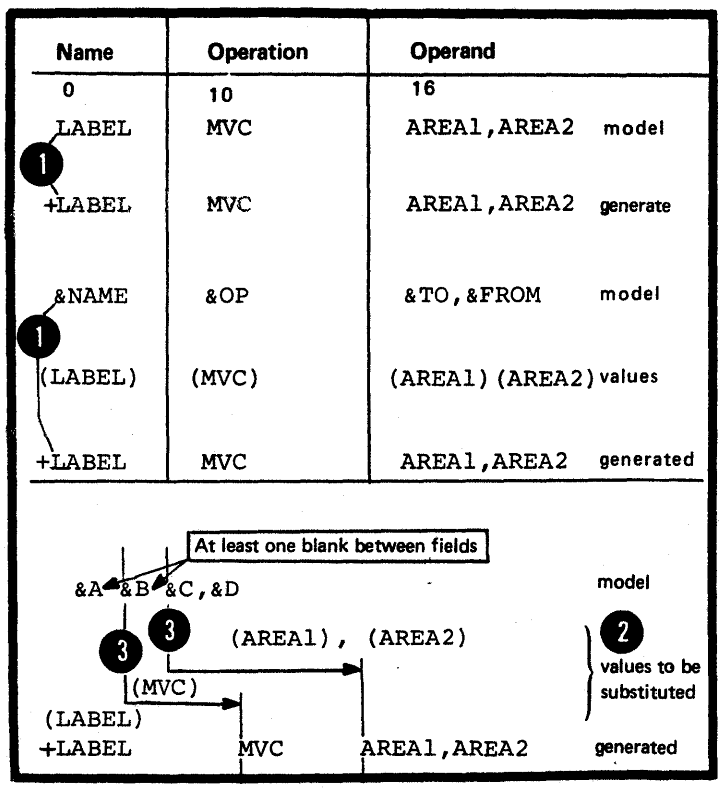
1. Symbolic parameters (described in J3 above),
2. System variable symbols (described in J7 below), and
3. SET symbols (described in L1A).

NOTES:

1. Symbolic parameters, SET symbols, and the system variable symbol, &SYSLIST, can all be subscripted. The remaining system variable symbols &SYSNDX, &SYSECT, &SYSPARM, &SYSDATE, and &SYSTIME cannot be subscripted.



2. The fields in a statement generated from a model statement
   1. appear in the listings in the same columns as in the model statement.
   2. However, when values are substituted for variable symbols the generated
   3. fields can be displaced to the right.



Rules for 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 • A period (.) is to be generated.
4 Two periods must be specified in the concatenated string following a variable symbol.

The concatenation character is not necessary when:

5 • An ordinary character string precedes a variable symbol.

6 • A special character, except left parenthesis or 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.

Concatenated String	Values to be Substituted: Variable symbol	Values to be Substituted: Value	Generated Result
&FIELD.A (1)	&FIELD	AREA	AREAA
&FIELDA	&FIELDA	SUM	SUM
(2) &DISP.(&BASE)	&DISP	100	100(10)
	&BASE	10	
Concatenation character is not generated			
DC D'&INT..&FRACT' (4)	&INT	99	DC D'99.88' (3)
	&FRACT	88	
DC D'&INT&FRACT' (7)			DC D'9988'
DC D'&INT.&FRACT' (optional; Concatenation character is not generated)			DC D'9988'
(5) FIELD&A	&A	A	FIELDA
&A+&B*3-D (6)	&A	A	A+B*3-D
	&B	B	
&A&B (7)			AB
&SYM(&SUBSCR) (8)	&SUBSCR	10	ENTRY
	&SYM(10)	ENTRY	

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.

THE NAME FIELD: The entries allowed in the name field of a model statement are given in the figure to the right, including the allowable results of generation.

❶ Variable symbols must not be used to generate comments statement indicators.

NOTE: Restrictions on the name entry are further specified where each individual assembler language instruction is described in this manual.

Name Field	Allowed	Not Allowed
In Model Statements (before generation)	▶ blank ▶ ordinary symbol ▶ sequence symbol ▶ variable symbol ▶ any combination of variable symbols and other character strings concatenated together	
In Generated Statements (generated results)	▶ blank ▶ valid ordinary symbol	* .* } ❶

THE OPERATION FIELD: The entries allowed in the operation field of a model statement are given in the figure to the right, including the allowable results of generation.

(1) The operation codes ICTL and OPSYN are not allowed inside a macro (2) definition. The MACRO and MEND operation codes are not allowed in model statements; they are used only for delimiting macro definitions.

DOS (3) The END operation code is not allowed inside a macro definition.

(4) If the REPRO operation code is specified in a model statement, no substitution is performed for (5) 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:

(6) • Any machine instruction, or

(7) • The assembler instructions listed.

(8) NOTE: The MNOTE and MEXIT statements are not model statements; they are described in J5D and J5E respectively.

The generated operation code must not be an operation code for the following (or their OPSYN equivalents):

(9) • A macro instruction,

(10) • A conditional assembly instruction, or

(11) • The assembler instructions listed.

DOS (12) • The END operation code must not be generated.

Operation Field	Allowed	Not Allowed
In Model Statements (Before Generation)	▶ An ordinary symbol that represents the operation code for: – any machine instruction – a macro instruction – the following Assembler instructions: CCW, CNOP, COM, COPY, CSECT, CXD, DC, DROP, DS, DSECT, DXD, EJECT, END, ENTRY, EQU, EXTRN, ISEQ, LTORG, ORG, POP, PRINT, PUNCH, PUSH, (4) REPRO, SPACE, START, TITLE, USING, WXTRN, MEXIT, MNOTE (8) (5) ▶ A variable symbol ▶ A combination of variable symbols and other character strings concatenated together (OS only)	▶ blank ▶ The assembler operation codes: (1) ICTL, OPSYN (2) MACRO, MEND (3) END DOS
In Generated Statements (Generated Results)	▶ An ordinary symbol that represents the operation code for: (6) – any machine instruction – the following assembler instructions: (7) CCW, CNOP, COM, CSECT, CXD, DC, DROP, DS, DSECT, DXD, EJECT, END, ENTRY, EQU, EXTRN, LTORG, ORG, POP, PRINT, PUNCH, PUSH, SPACE, TITLE, USING, WXTRN, (MNOTE) (8) OS only	(9) ▶ blank ▶ a macro instruction operation code ▶ a conditional assembly operation code: (10) ACTR, AGO, AGOB, AIF, AIFB, ANOP, GBLA, GBLB, GBLC, LCLA, LCLB, LCLC, SETA, SETB, SETC ▶ the following assembler operation codes: (11) COPY, ICTL, ISEQ, MACRO, MEND, MEXIT, OPSYN, REPRO (12) END DOS

THE OPERAND FIELD: The entries allowed in the operand field of a model statement are given in the figure to the right, including the allowable results of generation.

1 NOTE: Variable symbols must not be used in the operand field of a COPY, ICTL, ISEQ, or OPSYN instruction.

Operand Field	Allowed	Not Allowed
In Model Statements (Before Generation)	➤ 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	
In Generated Statements (Generated Results)	➤ blank (if valid) ➤ Character String that represents a valid assembler or machine instruction operand field	➤ operand field of a: COPY, ICTL, ISEQ or OPSYN statement 1

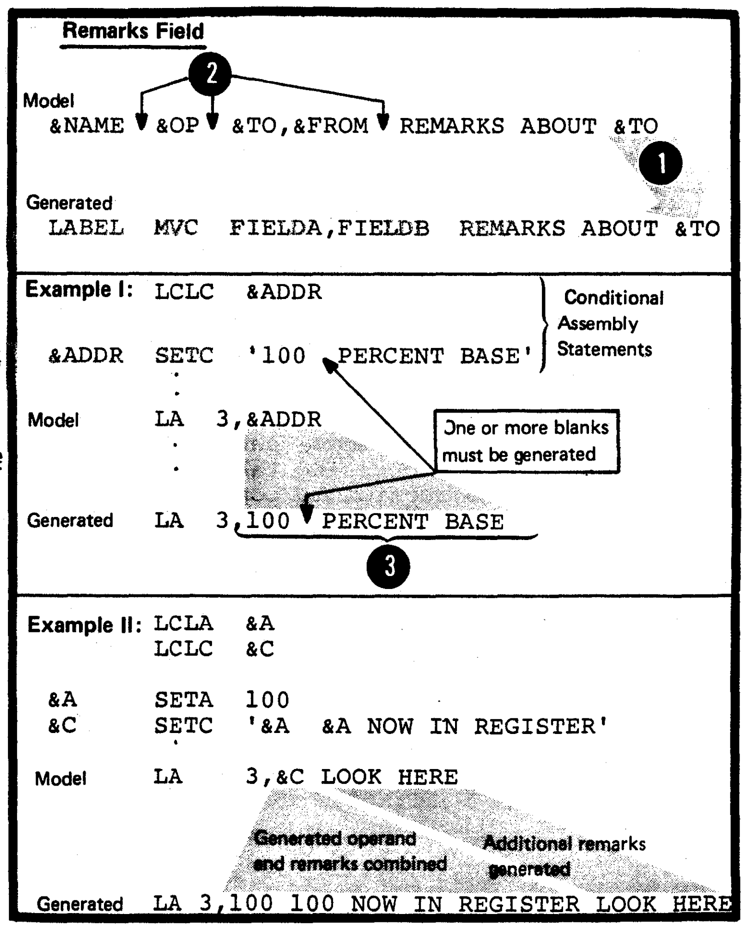
THE REMARKS FIELD: Any combination of characters can be specified in the remarks field of a model

1 statement. No values are substituted into variable symbols in this field.

NOTE: One or more blanks must be used in a model statement to separate

2 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.

3 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.



J5 -- Processing Statements

J5A -- CONDITIONAL ASSEMBLY INSTRUCTIONS

Conditional assembly instructions allow you to determine at pre-assembly time the content of the generated statements and the sequence in which they are generated. The instructions and their functions are given in the figure to the right.

Conditional assembly instructions can be used both inside macro definitions and in open code. They are fully described in Section L.

Conditional Assembly Instruction	Function Performed
GBLA,GBLB,GBLC LCLA,LCLB,LCLC	Declaration of initial values of variable symbols (global and local SET symbols)
SETA,SETB,SETC	Assignment of values to variable symbols (SET symbols)
	Branching
AIF	– Conditional (based on logical test)
AGO	– Unconditional
ANOP	– To next Sequential instruction (No operation)
ACTR	Setting Loop Counter

J5B -- INNER MACRO INSTRUCTIONS

Macro instructions can be nested inside macro definitions, allowing you to call other macros from within your own definitions. Nesting of macro instructions is fully described in K6.

J5C -- THE COPY INSTRUCTION

Purpose

The COPY instruction, inside macro definitions, allows you to copy into the macro definition any sequence of statements allowed in the body of a macro definition. These statements become part of the body of the macro before macro processing takes place. You can also use the COPY instruction to copy complete macro definitions into the beginning of a source module.

The specifications for the COPY instruction, which can also be used in open code, are described in E1A.

J5D -- THE MNOTE INSTRUCTION

Purpose

You can use the MNOTE instruction to generate your own error messages or display intermediate values of variable symbols computed at pre-assembly time.

Specifications

The MNOTE instruction can be used inside macro definitions [redacted] and its operation code can be created by substitution. The MNOTE instruction causes the generation of a message which is given a statement number in the printed listing.

The format of the MNOTE instruction statement is given in the figure to the right.

❶ The n stands for a severity code. The rules for specifying the contents of the severity code subfield are as follows:

❷ 1. The severity code can be specified as any arithmetic expression allowed in the operand field of a SETA instruction. The expression must have a value in the range 0 through 255.

❸ 2. If the severity code is omitted, but the comma separating it from the message is present, the assembler assigns a default value of 1 as the severity code.

❹ 3. An asterisk in the severity code subfield causes the message and the asterisk to be generated as a comments statement.

❺ 4. If the entire severity code subfield is omitted, including the comma separating it from the message, the assembler generates the message as a comments statement.

MNOTE

Name	Operation	Operand
A sequence symbol or blank	MNOTE	One of four options allowed: ❶ n, 'message' / ,'message' } error message *, 'message' / 'message' } comments

Examples:

Source Statements	Generated Result
❷ MNOTE 2,'ERROR IN SYNTAX'	2,ERROR IN SYNTAX
❸ MNOTE ,'ERROR, SEV 1'	,ERROR, SEV 1
❹ MNOTE *,'NO ERROR'	*,NO ERROR
❺ MNOTE 'NO ERROR'	NO ERROR

NOTES:

1. An MNOTE instruction causes a message to be printed, if the current PRINT option is ON, even if the PRINT NOGEN option is specified.

2. The statement number of the message generated from an MNOTE instruction with a severity code is listed among any other error messages for the current source module. However, the message is printed only if the severity code specified is greater than or equal to the severity code "nnn" in the assembler option, FLAG(nnn), contained in the EXEC statement that invokes the assembler.

DOS The assembler option FLAG does not exist, and the severity code is not used by the DOS control program.

3. The statement number of the comments generated from an MNOTE instruction without a severity code is not listed among other error messages.

Any combination of up to 256 characters enclosed in apostrophes can be specified in the message subfield. The rules that apply to this character string are as follows:

1 • Variable symbols are allowed (NOTE: variable symbols can have a value that includes even the enclosing apostrophes).

2 • Double ampersands and double apostrophes are needed 3 to generate one ampersand or one apostrophe. If variable symbols have ampersands or apostrophes as values, the
4 values must have double ampersands or apostrophes.

NOTE:

Any remarks for the MNOTE instruction statement must be separated from the apostrophe that ends the message by one or more blanks.

MNOTE Operand (Severity Code)	Value of Variable Symbol	Generated Result
3,'THIS IS A MESSAGE'		3,THIS IS A MESSAGE
3,&PARAM (1)	&PARAM='ERROR'	3,ERROR
3,'VALUE OF &&A IS &A' (2) (1)	&A=10	3,VALUE OF &A IS 10
3,'L''&AREA' (3)	&AREA=FIELD1	3,L'FIELD1
3,'DOUBLE &AMPS'	&AMPS=&& (4)	3,DOUBLE &
3,'DOUBLE L&APOS&AREA'	&APOS='' &AREA=FIELD1	3,DOUBLE L'FIELD1
3,'MESSAGE STOP'PED' — Invalid remarks, must be separated from operand by one or more blanks		
3'MESSAGE STOP' RMRKS — Valid Remarks entry		3,MESSAGE STOP RMRKS

J5E -- THE MEXIT INSTRUCTION

Purpose

The MEXIT instruction allows you to provide an exit for the assembler from any point in the body of a macro definition. The MEND instruction provides an exit only from the end of a macro definition (see J2B).

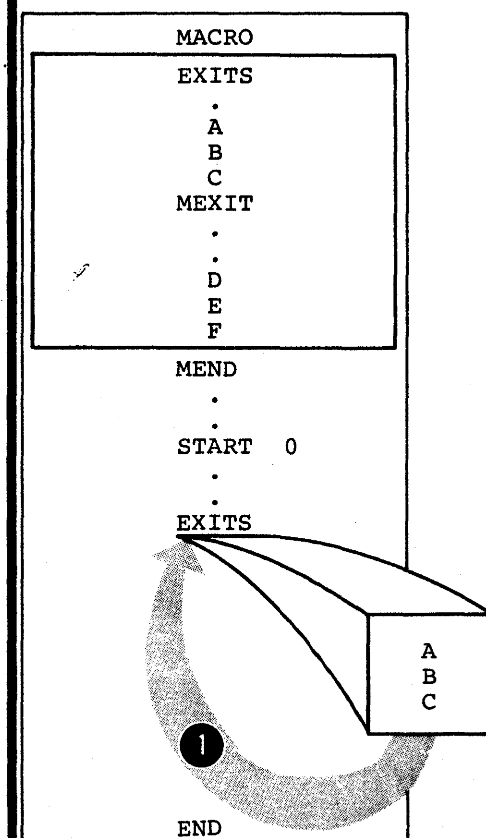
Specifications

The MEXIT instruction statement can be used only inside macro definitions. It has the format given in the figure to the right.

The MEXIT instruction causes the assembler to exit from a macro definition to the next sequential instruction after the macro instruction that calls the definition. (This also applies to nested macro instructions, which are described in K6.) ❶

MEXIT

Name	Operation	Operand
Sequence symbol or blank	MEXIT	Not required

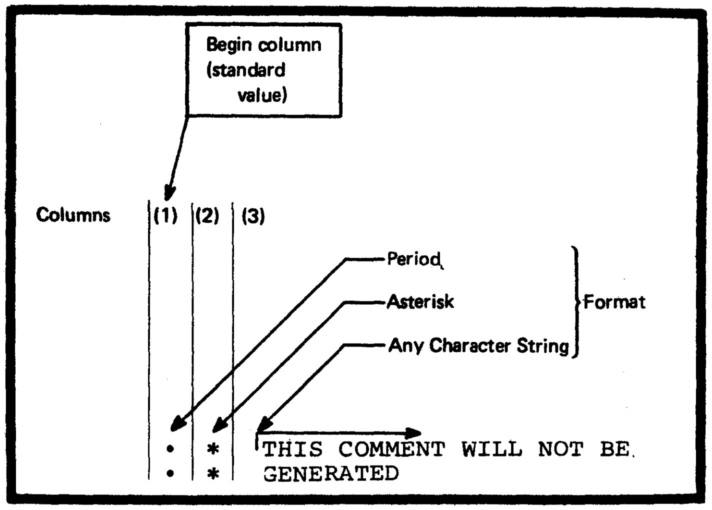


J6 – Comments Statements

J6A -- INTERNAL MACRO COMMENTS STATEMENTS

Purpose

You write internal macro comments in the body of a macro definition, to describe the operations performed at pre-assembly time when the macro is processed.



Specifications

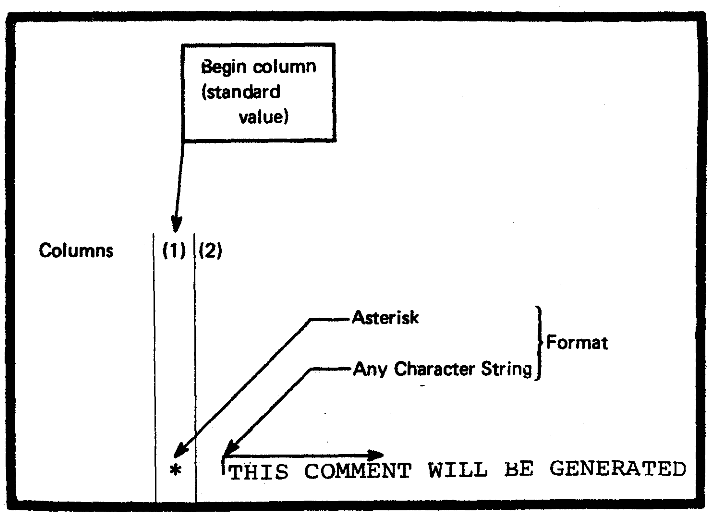
Internal macro comments statements can be used only inside macro definitions. An example of their correct use is given in the figure to the right.

No values are substituted for any variable symbols that are specified in macro comments statements.

J6B -- ORDINARY COMMENTS STATEMENTS

Purpose

Ordinary comments statements (described in B1C) allow you to make descriptive remarks about the generated output from a macro definition.



Specifications

Ordinary comments statements can be used in macro definitions and in open code. An example of their correct use is shown in the figure to the right.

Even though this type of statement is generated along with the model statements of a macro definition, values are not substituted for any variable symbols specified.

J7 -- System Variable Symbols

Purpose

System variable symbols are variable symbols whose values are set by the assembler according to specific rules. You can use these symbols as points of substitution in model statements and conditional assembly instructions.

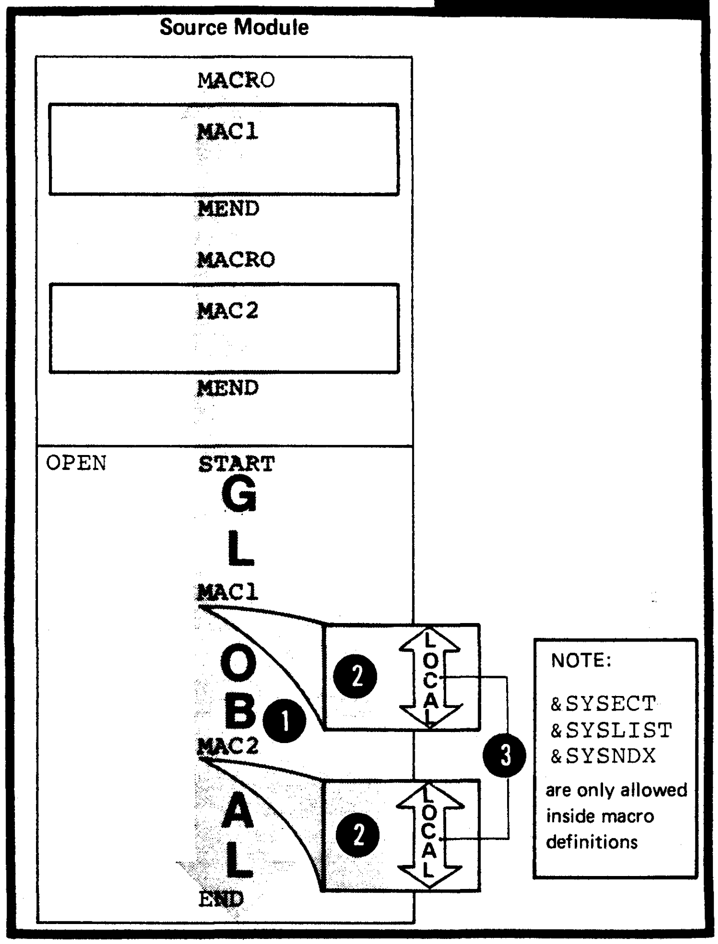
General Specifications for System Variable Symbols

OS only The system variable symbols: &SYSDATE, &SYSPARM, and &SYSTIME, can be used as points of substitution both inside macro definitions and in open code. The remaining system variable symbols: &SYSECT, &SYSLIST, and &SYSNDX, can be used only inside macro definitions. All system variable symbols are subject to the same rules of concatenation and substitution as other variable symbols (see J4B).

System variable symbols must not be used as symbolic parameters in the macro prototype statement. Also, they must not be declared as SET symbols (see L2).

The assembler assigns read-only values to system variable symbols; they cannot be changed by using the SETA, SETB, or SETC instructions (see L3).

Syst. Var. Sym.



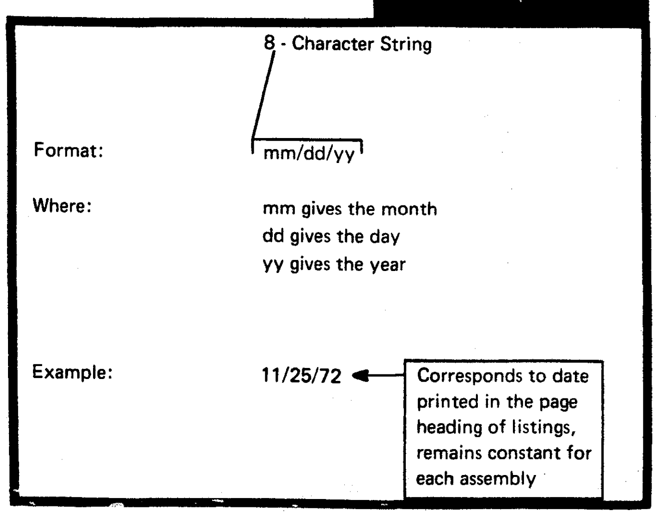
THE SCOPE OF SYSTEM VARIABLE SYMBOLS: The system variable symbols: **&SYSDATE**, &SYSPARM, and **&SYSTIME**, have a global scope. This means that they are assigned a read-only value for an entire source module; a value that is the same throughout (1) open code and (2) inside any macro definitions called. The system variable symbols: &SYSECT, &SYSLIST, and &SYSNDX, have a local scope. They are assigned a read-only value each time a macro is called, and have that value only within the (3) expansion of the called macro.

OS only

J7A -- &SYSDATE

OS only

&SYSDATE



Purpose

You can use &SYSDATE to obtain the date on which your source module is assembled.

Specifications

The global system variable symbol &SYSDATE is assigned a read-only value of the format given in the figure to the right.

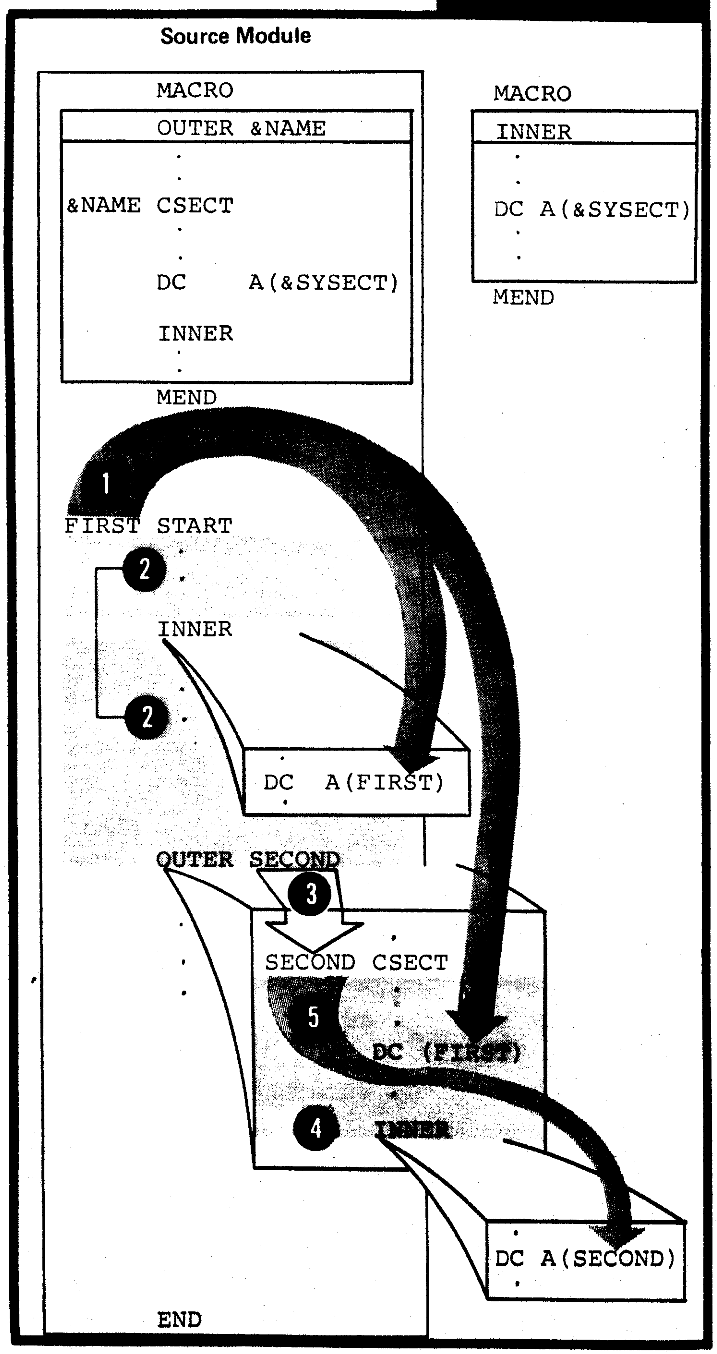
NOTE: The value of the type attribute of &SYSDATE (T'&SYSDATE) is always U and the value of the count attribute (K'&SYSDATE) is always eight. (Attributes are fully described in L1B.)

J7B -- &SYSECT

&SYSECT

Purpose

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.



Specifications

The local system variable symbol &SYSECT is assigned a read-only value each time a macro definition is called.

1 The value assigned is the symbol that represents the name of the
2 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
3 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
4 subsequent macro call. Nested macros cause the assembler to assign
5 a value to &SYSECT that depends on the control section in force inside the outer macro when the inner macro is called (see K6).

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
OS only attribute (K'&SYSECT) is equal to the number of characters assigned as a value to &SYSECT. (Attributes are fully described in L1E.)

J7C -- &SYSLIST

Purpose

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 positional operand or sublist entry in a macro call. &SYSLIST allows you to refer to positional operands for which no corresponding positional parameter is specified in the macro prototype statement.

Specifications

The local system variable symbol &SYSLIST is assigned a read-only value each time a macro definition is called.

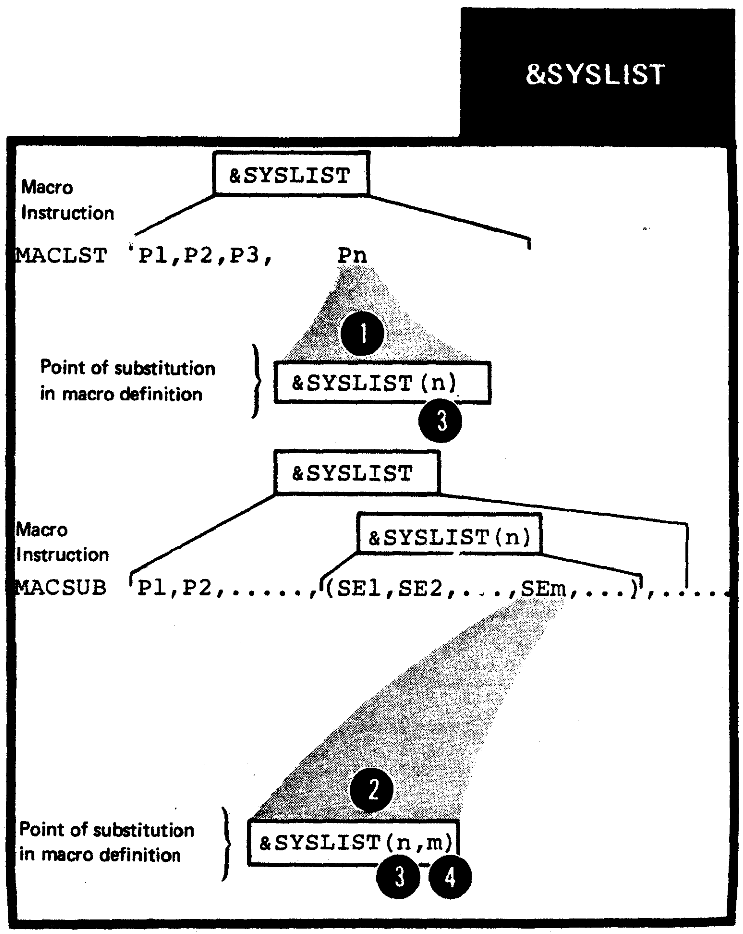
&SYSLIST refers to the complete list of positional operands specified in a macro instruction. &SYSLIST does not refer to keyword operands.

However, &SYSLIST cannot be specified as &SYSLIST alone. One of the two forms given in the figure to the right must be used as a point of substitution:

❶ 1. To refer to a positional operand

❷ 2. To refer to a sublist entry of a positional operand (sublists are fully described in K4 below).

❸ ❹ The subscript n indicates the position of the operand referred to. The subscript m, if specified, indicates the position of an entry in the sublist specified in the operand whose position is indicated by the first subscript n.



The subscripts n and m can be any arithmetic expression allowed in the operand of a SETA instruction (see L3A). The subscript n must be greater than or equal to 0. The subscript m must be greater than or equal to 1.

The figure to the right shows examples of the values assigned to &SYSLIST according to the value of its subscript, m and n.

❶ If the position indicated by n refers to an omitted operand or ❷ refers past the end of the list of positional operands specified, the null character string is substituted for &SYSLIST(n). If the position (in a sublist) indicated by the second subscript, m, refers ❸ to an omitted entry or ❹ 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). Further, if the nth positional operand is not a sublist, ❺ &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.

❻ NOTE: If the value of subscript n is zero, then &SYSLIST(n) is assigned the value specified in the name field of the macro instruction, except when it is a sequence symbol.

Macro Instruction:

```
NAME   MACALL ONE,TWO,(3,4,,6),,EIGHT
```

Point of substitution in macro definition	Value Substituted
&SYSLIST(2)	TWO
&SYSLIST(3,2)	4
❶ &SYSLIST(4)	Null
❷ &SYSLIST(9)	Null
❸ &SYSLIST(3,3)	Null
❹ &SYSLIST(3,5)	Null
❺ &SYSLIST(2,1)	TWO
&SYSLIST(2,2)	Null
❻ &SYSLIST(0)	NAME
&SYSLIST(3)	(3,4,,6)

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. (Attributes are fully described in L1B.) However, the number attribute of &SYSLIST, N'&SYSLIST, is different from the number attribute described in L1B. One of the two forms given in the figure to the right can be used for the number attribute:

(1) • To indicate the number of positional operands specified in a call

(2) • To indicate the number of sublist entries that have been specified in a positional operand indicated (3) by the subscript.

NOTES:

(4) 1. For N'&SYSLIST, positional operands are counted if specifically omitted by specifying the comma that would normally have followed the operand.

(5) 2. For N'&SYSLIST(n), sublist entries are counted if specifically omitted by specifying the comma that would normally have followed the entry.

(6) 3. If the operand indicated by n is not a sublist, N'&SYSLIST(n) (7) is 1. If it is omitted, N'&SYSLIST(n) is zero.

(1) N'&SYSLIST

Macro Instruction		Value of N'&SYSLIST
MACLST	1,2,3,4	4
MACLST	A,B,,D,E	5 (4)
MACLST	,A,B,C,D	5 (4)
MACLST	(A,B,C),(D,E,F)	2 Counts sublists as one operand
MACLST		0
MACLST	KEY1=A,KEY2=B	0 Keyword operands are not counted
MACLST	A,B,KEY1=C	2

(2) N'&SYSLIST(n)

Macro Instruction (3) (n=2)		Value of N'&SYSLIST (2)
MACSUB	A,(1,2,3,4,5),B	5
MACSUB	A,(1,,3,,5),B	5 (5)
MACSUB	A,(,2,3,4,5),B	5 (5)
MACSUB	A,B,C (6)	1
MACSUB	A,,C (7)	0
MACSUB	A,KEY=(A,B,C)	0
MACSUB	(7)	0 Keyword sublists are not counted

J7D -- &SYSNDX

Purpose

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.

&SYSNDX

Specifications

The local system variable symbol &SYSNDX is assigned a read-only value each time a macro definition is called from a source module.

1 The value assigned to &SYSNDX is a 4-digit number, starting at 0001 for the first macro called by a
2 program. It is incremented by one for each subsequent macro call (including nested macro calls, described in K6).

NOTES:

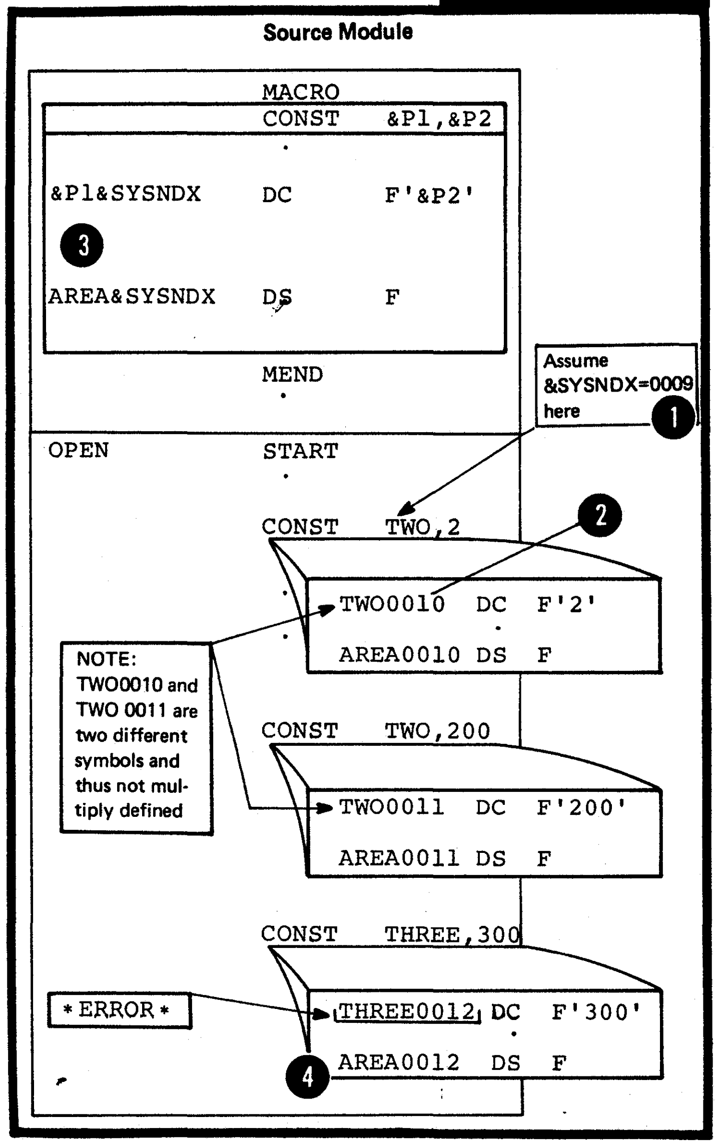
1. &SYSNDX does not generate a valid symbol, and it must:

3 a. Follow the symbol to which it is concatenated

4 b. Be concatenated to a symbol containing four characters or less.

2. The value of the type attribute of &SYSNDX (T'&SYSNDX) is always N, and the value of the count attribute (K'&SYSNDX) is always four.

(Attributes are fully described in L1B.)



J7E -- &SYSPARM

Purpose

You can use &SYSPARM to communicate with an assembler source module through the job control language. Through &SYSPARM, you pass a character string into the source module to be assembled from a job control language statement or from a program that dynamically invokes the assembler. Thus, you can set a character value from outside a source module and then examine it as part of the source module at pre-assembly time, during conditional assembly processing.

&SYSPARM

Specifications

❶ The global system variable symbol &SYSPARM is <u>assigned a read-only value</u> in a job control statement or in a field set up by a program that dynamically invokes the assembler. It is treated as a global SETC symbol in a source module, except that its value cannot be changed.

The largest value that &SYSPARM can hold when you code your own procedure is 91 characters, which can be specified by an invoking program. However, if the PARM field of the EXEC statement is used to specify its value, the PARM field restrictions reduce its maximum possible length.

CMS Under CMS, the option line of the ASSEMBLE command cannot exceed 100 characters, thus limiting the number of characters you can specify for &SYSPARM.

DOS The largest value &SYSPARM can hold is 8 characters.

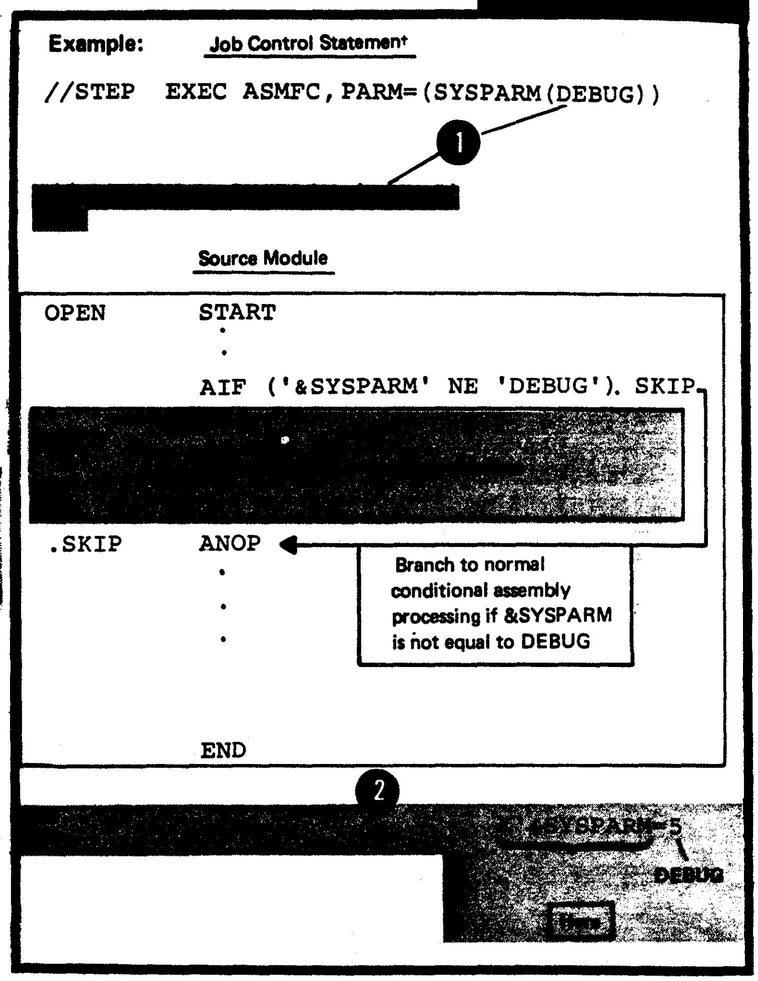
NOTES:

1. No values are substituted for variable symbols in the specified value; however, double ampersands must be used to represent single ampersands in the value.

CMS Since CMS does not strip ampersands from the variable symbol, you need not specify double ampersands for CMS.

2. Double apostrophes are needed to represent single apostrophes because the entire PARM field specification is enclosed in apostrophes.

CMS Since CMS does not strip single apostrophes from the variable symbol, you need not specify double apostrophes for CMS.



3. If SYSPARM is not specified in a job control statement outside the source module, &SYSPARM is assigned a default value of the null character string.

OS only 4. The value of the type attribute of &SYSPARM (T'&SYSPARM) is always U, while the value of the count attribute (K'&SYSPARM) is the number of characters specified for SYSPARM in a job control statement or in a field set up by a program that dynamically invokes the assembler. Double apostrophes and double ampersands count as one character.

CMS 5. CMS parses the command line, breaking the input into eight-character tokens; therefore, the SYSPARM option field under VM/370 is limited to an eight-character field. If you want to enter larger fields or if you want to enter parentheses or embedded blanks, you must enter the special symbol "?" (the question mark symbol) in the option field. When CMS encounters this symbol in the command line, it will prompt you with the message ENTER SYSPARM:, after which you may enter any characters you want up to the option line limit of 100 characters. The following code is an example of how to use the ? symbol in the SYSPARM field:

```
assemble test (load deck sysparm (?)
ENTER SYSPARM:
&&am,'bo).fy
      .
      .
      .
R;
```

J7 -- &SYSTIME

Purpose

You can use &SYSTIME to obtain the time at which your source module is assembled.

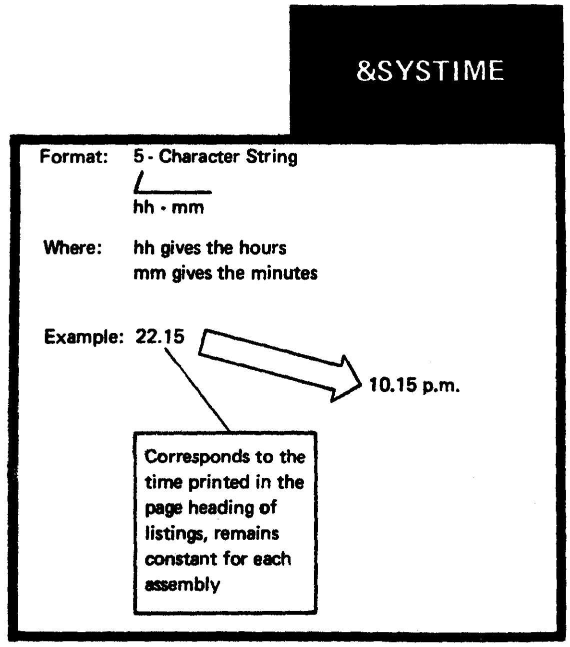
Specifications

The global system variable symbol &SYSTIME is assigned a read-only value of the format given in the figure to the right.

NOTES:

1. The value of the type attribute of &SYSTIME (T'&SYSTIME) is always U and the value of the count attribute (K'&SYSTIME) is always 5.

2. For systems without the internal timer feature, &SYSTIME is a 5-character string of blanks.



J8 -- Listing Options

In addition to the PRINT options that you can set from inside a source module, you can set other listing options from outside a source module by using the job control language. These options can be specified in the PARM field of the EXEC statement or by a program that dynamically invokes the assembler.

J8A -- LIBMAC

Purpose

The LIBMAC option allows you to print in the program listings the library macro definitions called from your source module, and any statements in open code following the first END statement (coded or generated) that is processed by the assembler.

Specifications

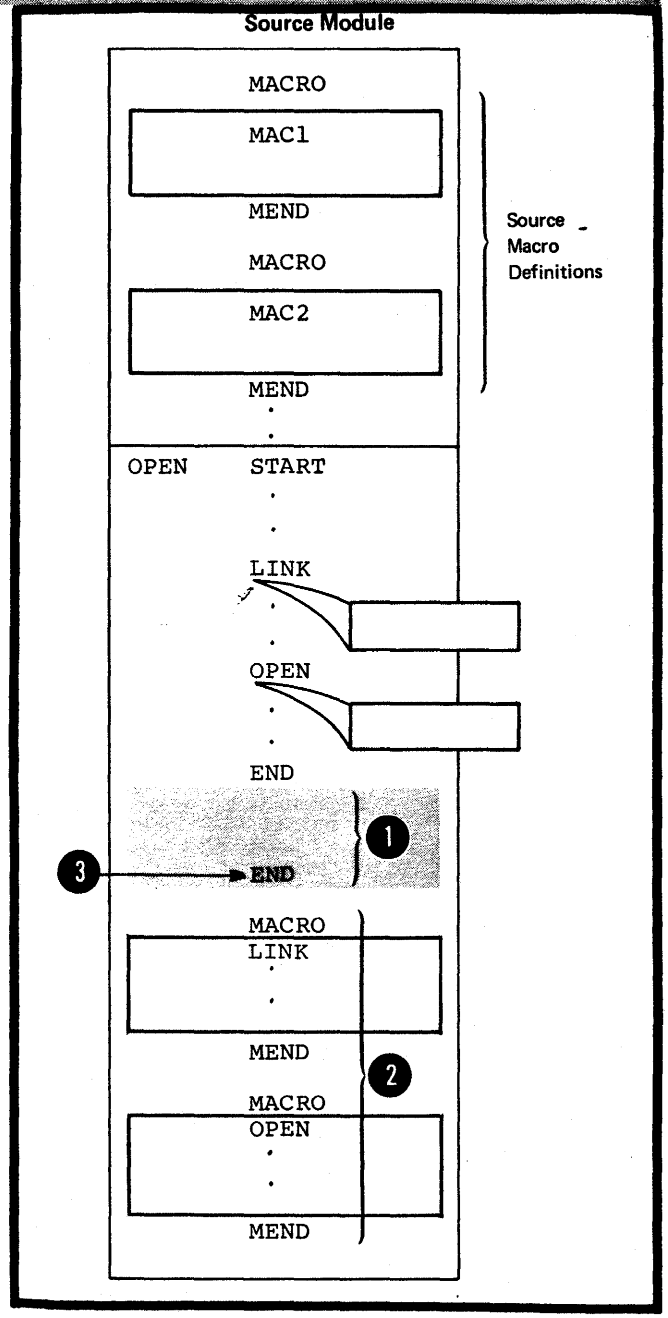
The LIBMAC option, when set, causes:

1. Any statements in open code that follow the first END statement and

2. All library macro definitions called to be printed in the program listings after the first (or only) END statement of the source module.

3. NOTE: Multiple END statements can be coded or generated and are printed, but the first END statement processed ends the assembly.

The option NOLIBMAC suppresses the listing of the items mentioned above. It is the default option that applies to the assembling of source modules.



J8B -- MCALL

Purpose

The MCALL option allows you to list all the inner macro instructions that the assembler processes.

Specifications

The MCALL option, when set, causes all inner macro instructions processed by the assembler to be listed. The NOMCALL option suppresses the listing of inner macro instructions. It is the default option that applies to the assembling of source modules.

NOTE: The MLOGIC and ALOGIC options concern the listing of conditional assembly statements. They are discussed in L8.

Section K: The Macro Instruction

This section 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.

After studying this section, you should be able to use the macro instructions correctly to call the macro definitions that you write. You will also have a better understanding of what to specify when you call a macro and what will be generated as a result of that call.

K1 -- Using a Macro Instruction

K1A -- PURPOSE

The macro instruction provides the assembler with:

1. The name of the macro definition to be processed.

2. 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:

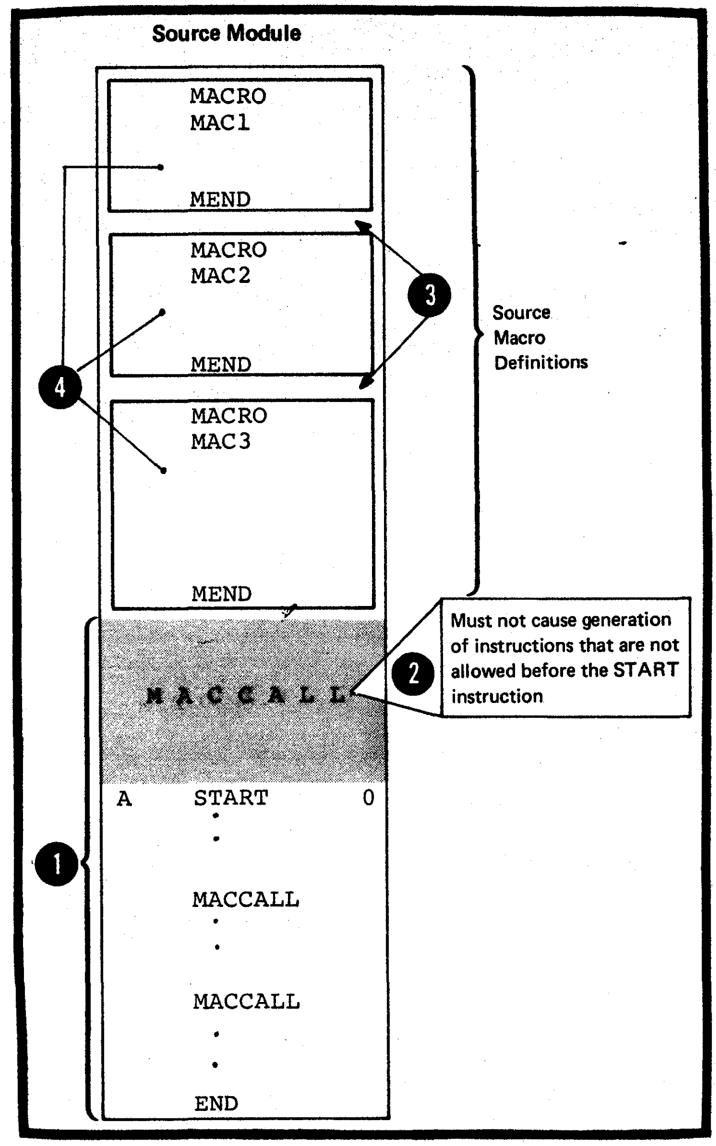
1. A sequence of statements generated from the model statements of the macro for further processing at assembly time.

2. Values assigned to global SET symbols. These values can be used in other macro definitions and in open code (see L1A).

K1B -- SPECIFICATIONS

Where Macro Instructions Can Appear

(1) A macro instruction can be written anywhere in the open code portion of a source module. However, the statements generated from the called macro definition must be valid assembler language instructions (2) and allowed where the calling macro instruction appears. A macro instruction is not allowed before (3) or between any source macro definitions, if specified, but it can be nested inside a macro (4) definition (see K6).



Macro Instruction Format

The format of a macro instruction statement is given in the figure to the right.

The maximum number of operands allowed is not fixed. It depends on the amount of virtual storage available to the program.

DOS Only 200 operands are allowed in the operand field.

If no operands are specified in the operand field, remarks are allowed if the absence of the operand entry is indicated by a comma preceded and followed by one or more blanks.

The entries in the name, operation, and operand fields correspond to entries in the prototype statement of the called macro definition (see K2).

Macro Inst.

Name	Operation	Operand
Any symbol or blank	Symbolic Operation Code	Zero or more operands separated by commas

Alternate Ways of Coding a Macro Instruction

A macro instruction can be specified in one of the three following ways:

1. The normal way, with the operands preceding any remarks.

2. The alternate way, allowing remarks for each operand.

3. A combination of the first two ways.

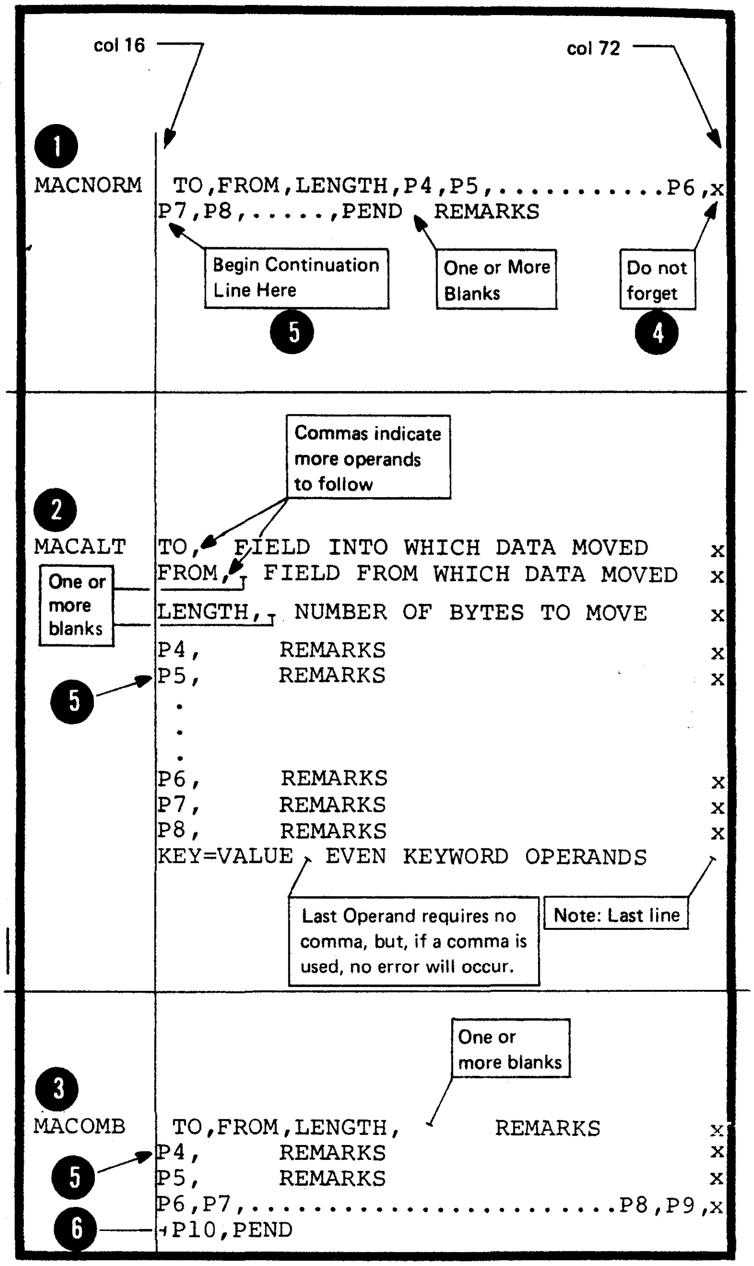
NOTES:

(4) 1. Any number of continuation lines are allowed. However, each continuation line must be indicated by a non-blank character in the column after the end column of the previous statement line (see B1B).

(5) 2. Operands on continuation lines must begin in the continue column, or

(6) 3. Otherwise, the assembler assumes that any lines that follow contain remarks.

NOTE: If any entries are made in the columns before the continue column in continuation lines, the assembler issues an error message and the whole statement is not processed.



K2 -- Entries

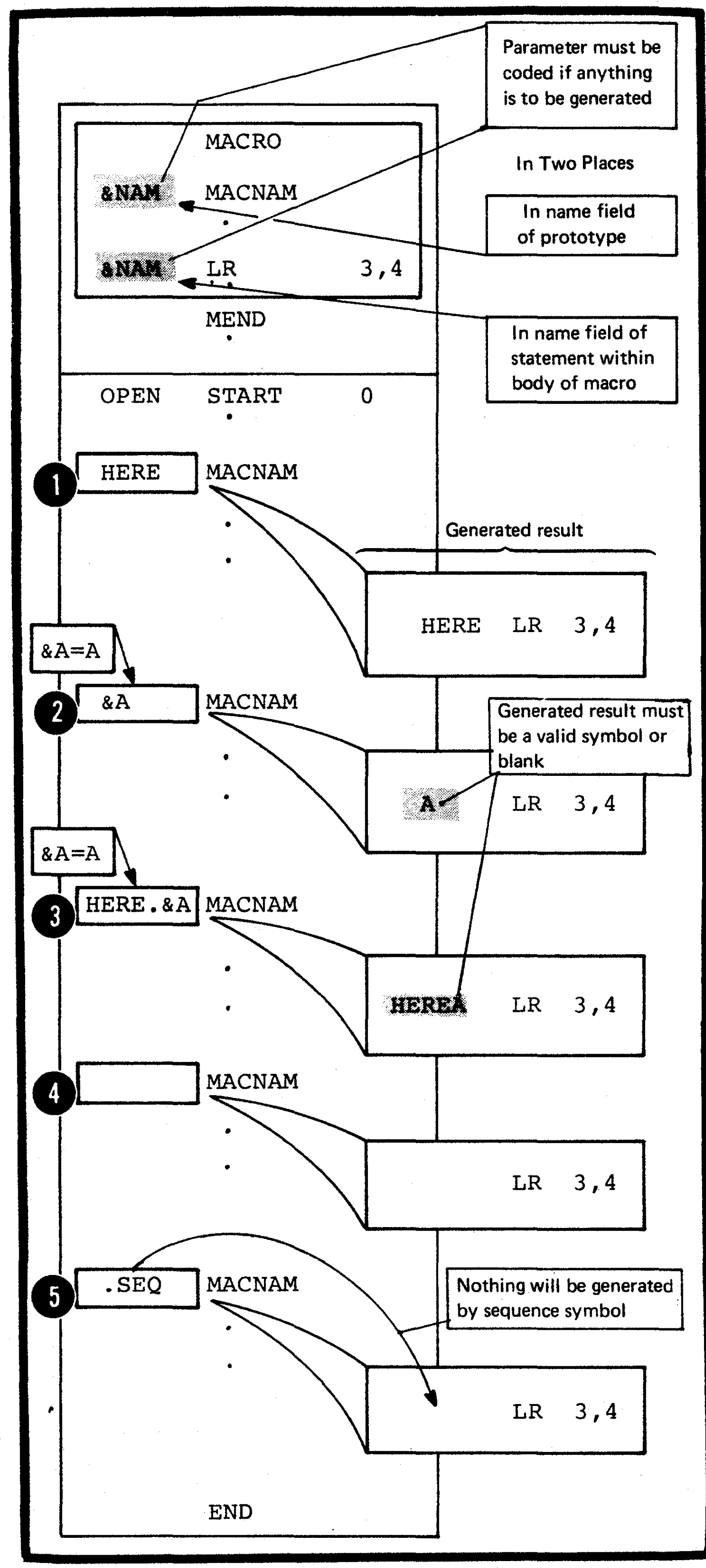
K2A -- THE NAME ENTRY

Purpose

You can use the name entry of a macro instruction:

1. Either to generate an assembly-time label for a machine or assembler instruction.

2. Or to provide a conditional assembly label (see sequence symbol in L1C) so that you can branch to the macro instruction at pre-assembly time if you want the called macro definition expanded.



Specifications

The name entry of a macro instruction can be:

1. an ordinary symbol
2. a variable symbol
3. a character string in which a variable symbol is concatenated to other characters
4. a blank
5. a sequence symbol, which is never generated.

K2B -- THE OPERATION ENTRY

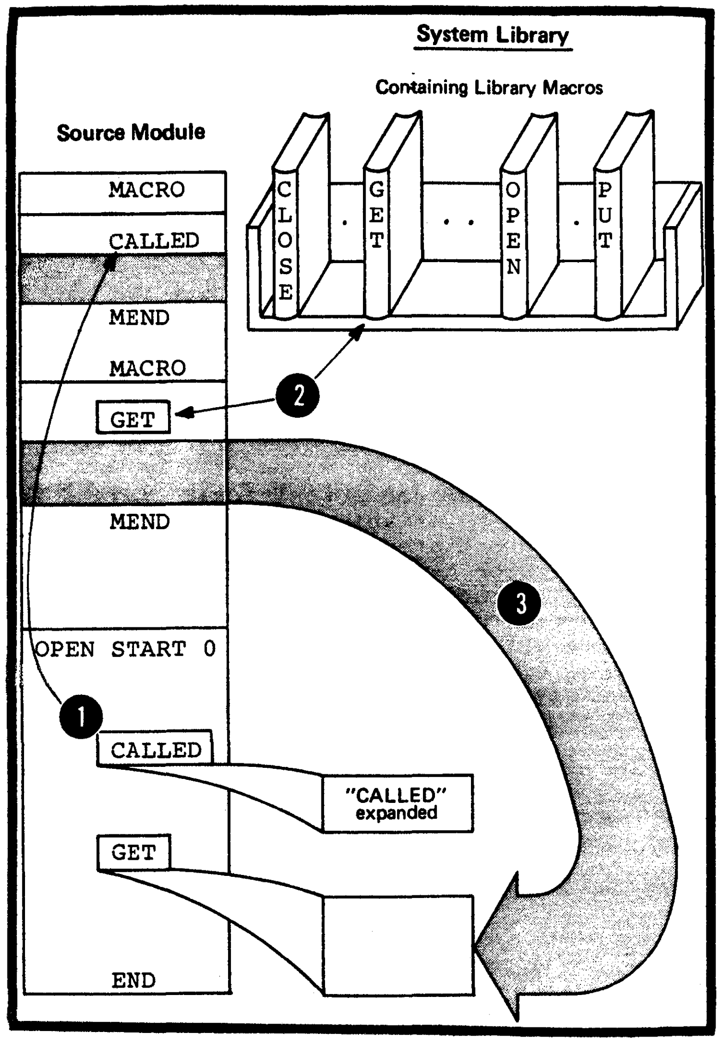
Purpose

The symbolic operation code you specify identifies the macro definition you wish the assembler to process.

Specifications

(1) The operation entry for a macro instruction must be a valid symbol that is identical to the symbolic operation code specified in the prototype statement of the macro definition called.

(2) NOTE: If a source macro definition with the same operation code as a library macro definition is called, the assembler processes the source (3) macro definition.



K2C -- THE OPERAND ENTRY

Purpose

You can use the operand entry of a macro instruction to pass values into the called macro definition. These values can be passed through:

1. The symbolic parameters you have specified in the macro prototype, or

2. The system variable symbol &SYSLIST if it is specified in the body of the macro definition (see J7C).

The two types of operands allowed in a macro instruction are the positional operand and the keyword operand (see K3). You can specify a sublist with multiple values in both types of operands (see K4). Special rules for the various values you can specify in operands are given in K5.

K3– Operands

K3A -- POSITIONAL OPERANDS

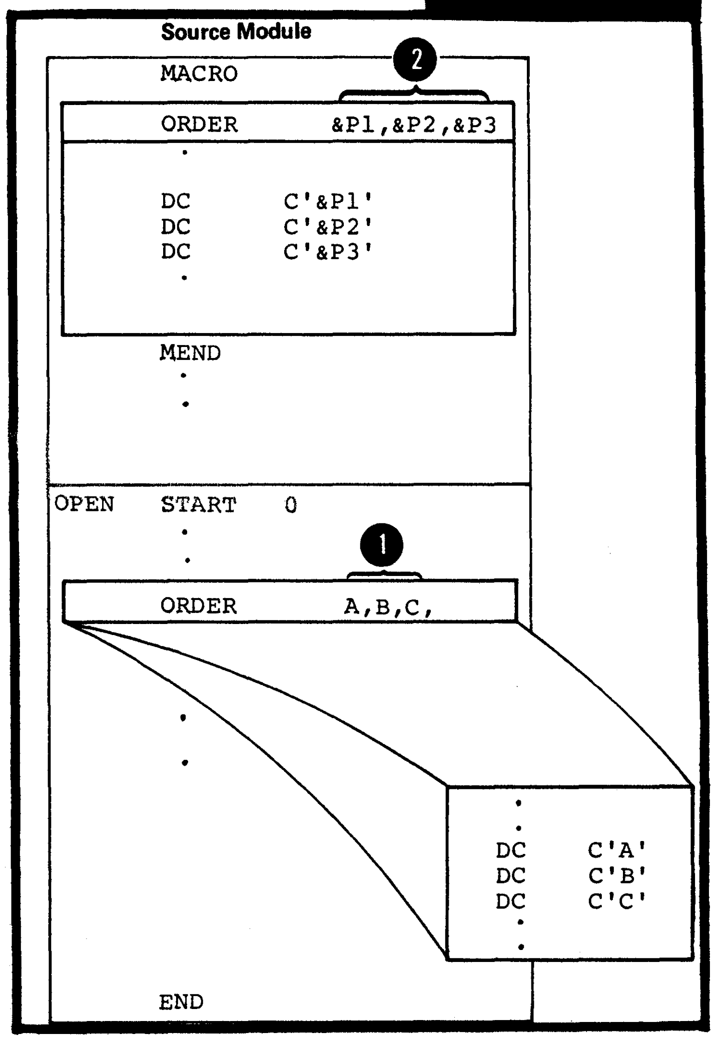
Purpose

You can use a positional operand to pass a value into a macro definition through the corresponding positional parameter declared for the definition. You should declare a positional parameter in a macro definition when you wish to change the value passed at every call to that macro definition.

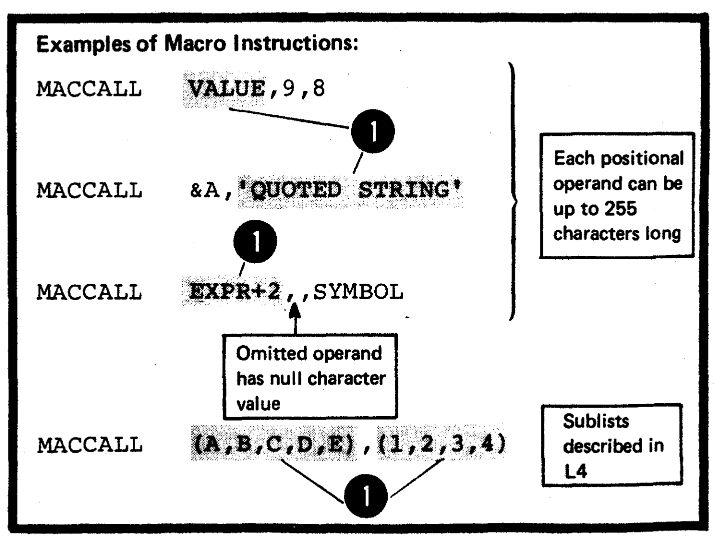
You can also use a positional operand to pass a value to the system variable symbol &SYSLIST. If &SYSLIST, with the appropriate subscripts, is specified in a macro definition, you do not need to declare positional parameters in the prototype statement of the macro definition. You can thus use &SYSLIST to refer to any positional operand. This allows you to vary the number of operands you specify each time you call the same macro definition. The use of &SYSLIST is described in J7C.

Specifications

❶ The positional operands of a macro instruction must be specified in the same order as the positional ❷ parameters declared in the called macro definition.



❶ Each positional operand constitutes a character string. It is this character string that is the value passed through a positional parameter into a macro definition.



The figure to the right illustrates what happens when the number of positional operands in the macro instruction differs from the number of positional parameters declared in the prototype statement of the called macro definition.

	Number of Positional Operands in macro instruction		
	EQUAL	GREATER THAN	LESS THAN
Number of positional parameters in Prototype of macro definition	Valid, if Operands are correctly specified		
		Meaningless, unless &SYSLIST is specified in definition to refer to excess operands	
			Omitted operands give null character values to corresponding parameters (or &SYSLIST specification)

K3B -- KEYWORD OPERANDS

Purpose

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.

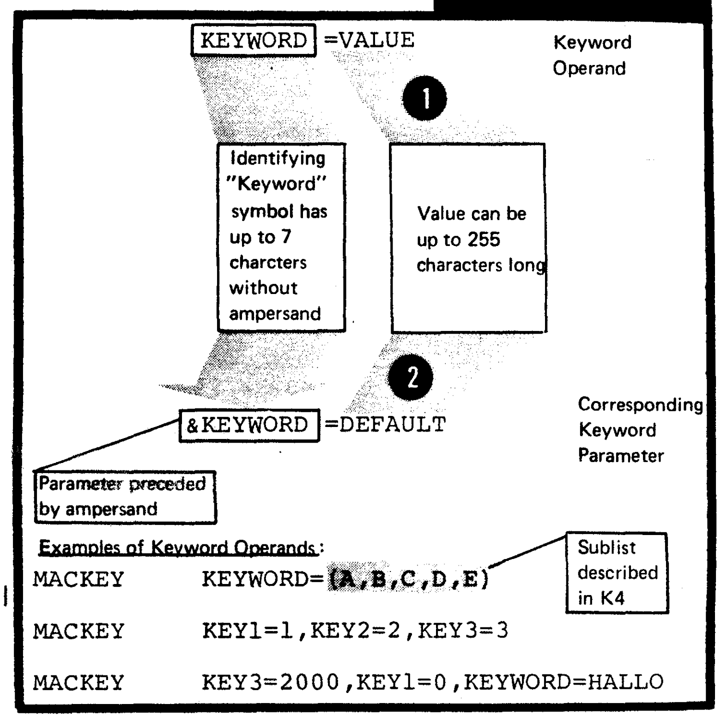
Page of GC33-4010-5
As Updated 28 Dec 1981
By TNL GN20-9372

Specifications

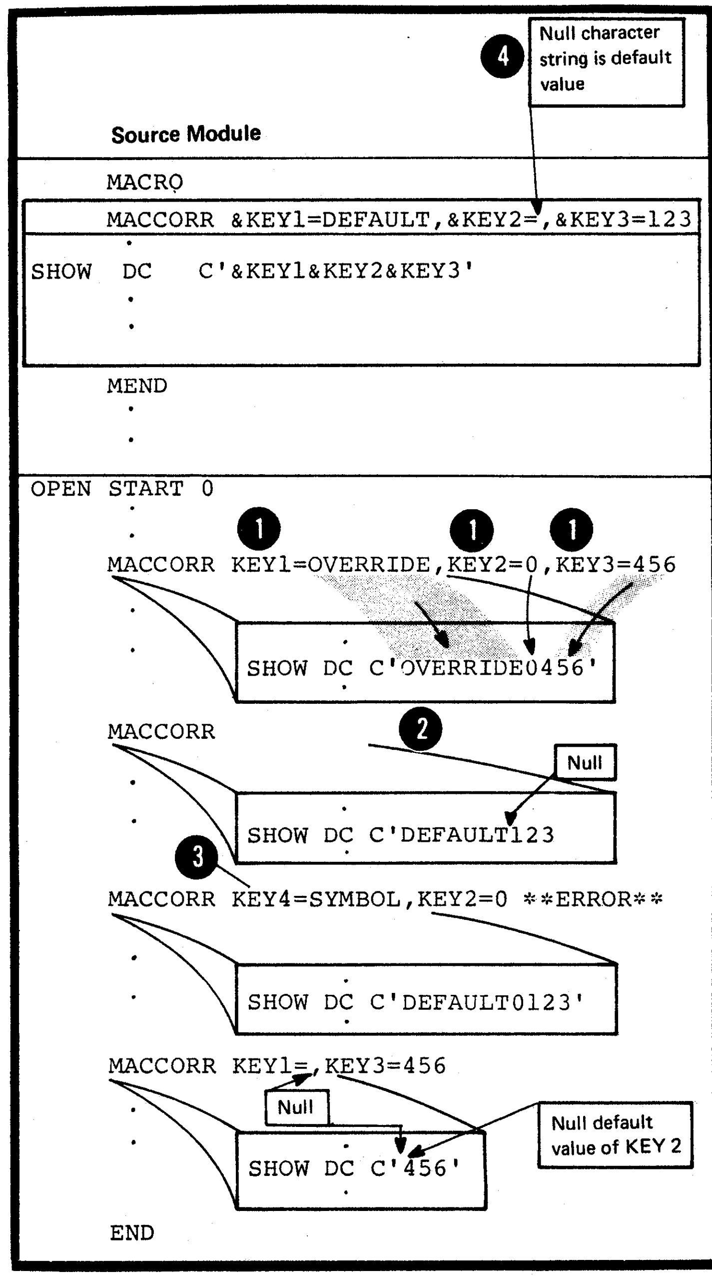
Any keyword operand specified in a macro instruction must correspond to a keyword parameter in the macro definition called. However, keyword operands do not have to be specified in any particular order.

A keyword operand must be coded in the format shown in the figure to the right. If a keyword operand (1) is specified, its value overrides the (2) default value specified for the keyword parameter.

The standard default value obeys the same rules as the value specified in the keyword operand (see K5).



The following examples describe the relationship between keyword operands and keyword parameters and the values that the assembler assigns to these parameters under different conditions.



(1) The keyword of the operand corresponds to a keyword parameter. The value in the operand overrides the default value of the parameter.

(2) The keyword operand is not specified. The default value of the parameter is used.

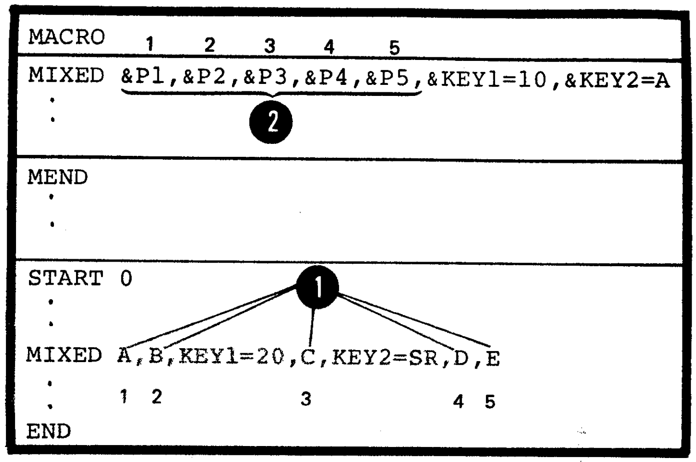
(3) The keyword of the operand does not correspond to any keyword parameter. The assembler issues an error message, but the macro is generated using the default values of the other parameters.

NOTE: The default value specified for a keyword parameter can be the (4) null character string. 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.

K3C -- COMBINING POSITIONAL AND KEYWORD OPERANDS

Purpose

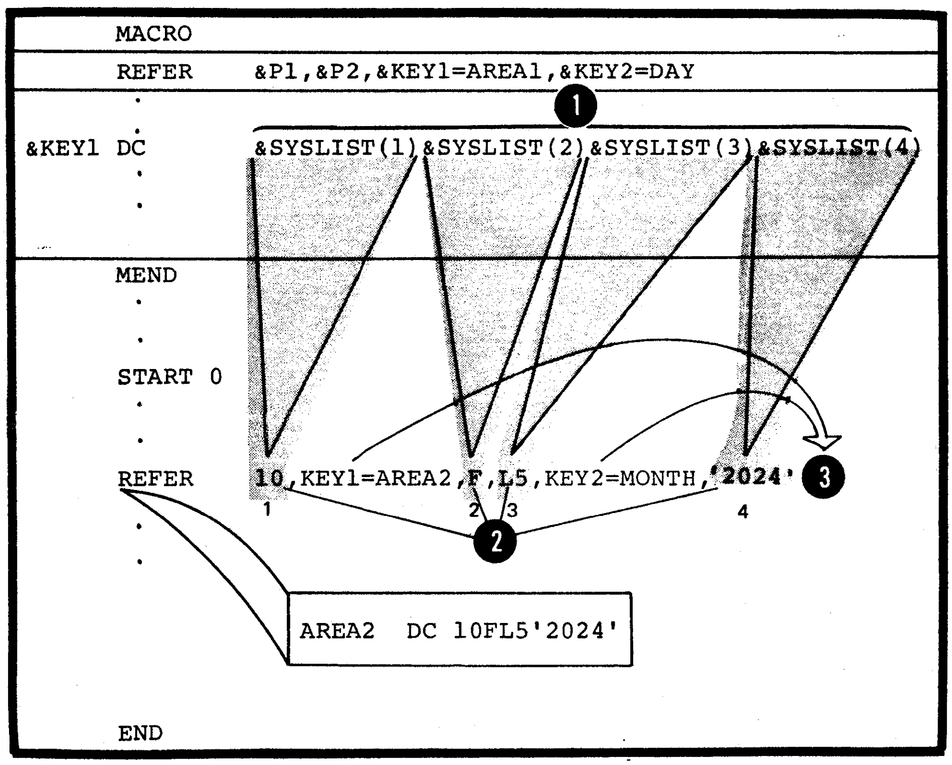
You can use positional and keyword operands in the same macro instruction: use a positional operand for a value that you change often and a keyword operand for a value that you change infrequently.



Specifications

Positional and keyword operands can be mixed in the macro instruction operand field. However, the (1) positional operands must be in the same order as the corresponding (2) positional parameters in the macro prototype statement.

DOS All positional operands must precede any keyword operands, if specified.



NOTE: The system variable symbol (1) &SYSLIST(n) refers only to the (2) positional operands in a macro instruction.

DOS All keyword operands must follow (3) any positional operands specified.

K4 -- Sublists in Operands

Purpose

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:

1. The corresponding symbolic parameter with an appropriate subscript or

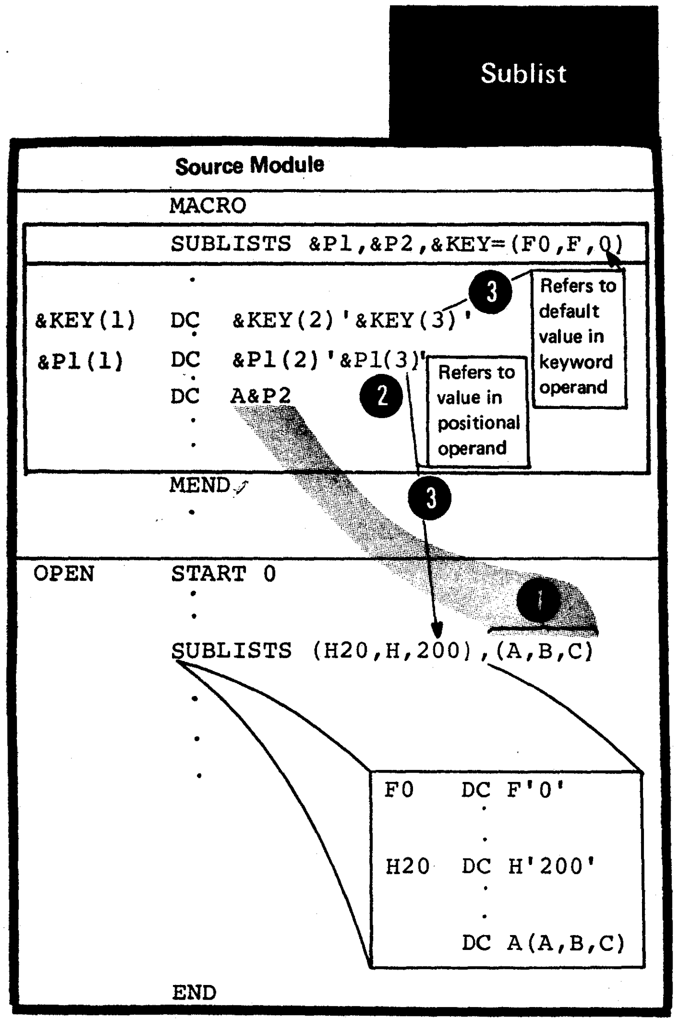
2. The system variable symbol &SYSLIST with appropriate subscripts, the first to refer to the positional operand and the second to refer to the sublist entry in the operand.

&SYSLIST can refer only to sublists in positional operands.

Specifications

The value specified in a positional or keyword operand can be a sublist.

1 A symbolic parameter can refer to the entire sublist or to an individual entry of the sublist. To refer to an individual entry, 2 the symbolic parameter must have a subscript whose value indicates 3 the position of the entry in the sublist. The subscript must have a value greater than or equal to one.



The format of a sublist is given in the figure to the right. A sublist, including the enclosing parentheses, must not contain more than 255 characters.

Format: One or more entries separated by commas and enclosed in parentheses

Examples: (A,B,C,D,E)

()

Valid sublist with the null character string as the only entry

The figure to the right shows the relationship between subscripted parameters and sublist entries if:

1. A sublist entry is omitted,
2. The subscript refers past the end of the sublist,
3. The value of the operand is not a sublist,
4. The parameter is not subscripted.

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)	1	(1,2,,4)	Null character string
&PAR(5)	2	(1,2,3,4)	Null character string
&PAR	3	A	A
&PAR(1)		A	A
&PAR(2)		A	Null character string
&PAR	4	(A)	(A)
&PAR(1)		(A)	A
&PAR(2)	2	(A)	Null character string
&PAR		()	()
&PAR(1)		()	Null character string
&PAR(3)		() (Considered as Sublists)	Null character string
&PAR(2)		(A, ,C,D) (This blank indicates end of operand field)	Nothing (*ERROR* Unmatched left parentheses)
&PAR(1)		( )	Nothing
		Positional Operands	
&POSPAR(3)		A,(1,2,3,4)	3
&SYSLIST(2,3)		A,(1,2,3,4)	3

K5 -- Values in Operands

Purpose

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:

1. Explicit values or the actual character strings you specify in the operand.

2. Implicit values, or the attributes inherent in the data represented by the explicit values.

Attributes are fully described in L1B.

Specifications

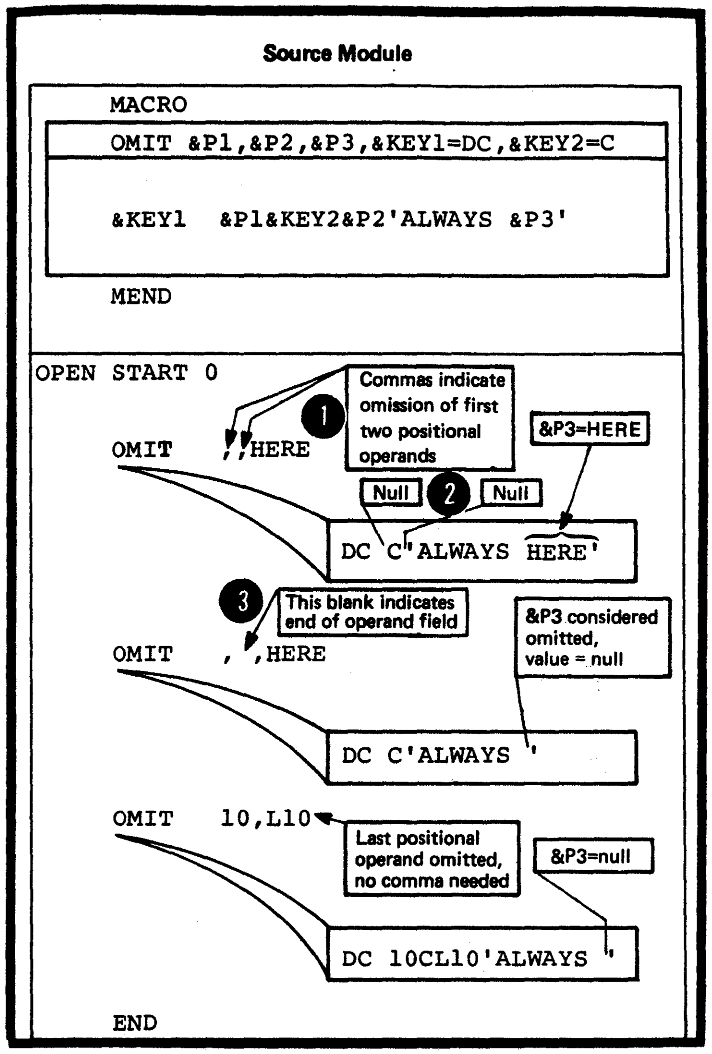
The explicit value specified in a macro instruction operand is a character string that can contain one or more variable symbols.

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 (see K4).

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 ❶ a positional operand or sublist entry is omitted, the null character ❷ string is assigned to the parameter.

❸ NOTE: Blanks appearing between commas do not signify an omitted positional operand or an omitted sublist entry.



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 ❶ the value of the variable symbol into the character string specified in a macro instruction operand. ❷ The resultant string is then the value passed into the macro definition. If the variable symbol is undefined, an error message is issued.

❸ Double ampersands must be specified if they are to be passed to the macro definition.

Value Specified In Operand	Value Of Variable Symbols ❶	Character String Value Passed ❷
&VAR	XYZ	XYZ
&A+&B+3+&C*10	&A=2 &B=X &C=COUNT	2+X+3+COUNT*10
'&MESSAGE' Quoted string described below	BLANK BETWEEN	'BLANK BETWEEN'
[redacted]REGISTR ❸		&&REGISTER
NOTE[redacted]		NOTE&&&&

<u>APOSTROPHES:</u> A single apostrophe is used: (1) to indicate the beginning and end of a quoted string, and (2) in a length attribute notation that is not within a quoted string.

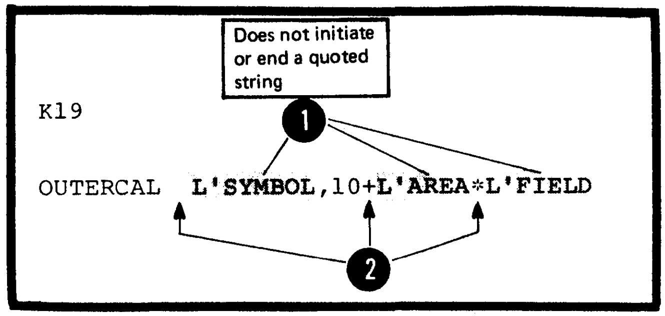
<u>QUOTED STRINGS:</u> A quoted string is any sequence of
❶ characters that <u>begins and ends</u> with a single apostrophe (compare with conditional assembly character expressions
❷ described in L4B). <u>Double apostrophes</u> must be specified inside each quoted string. This includes <u>substituted</u>
❸ <u>apostrophes.</u>

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 <u>one or more</u>
❹ <u>characters,</u> and each must contain an even number of apostrophes.

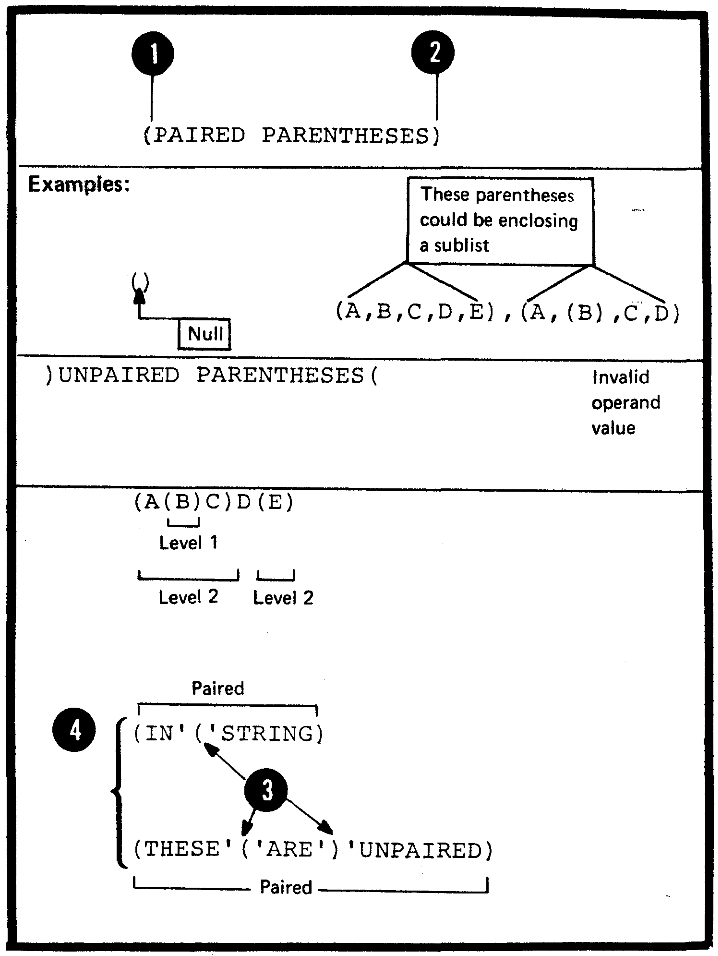
❶
'QUOTED STRING'

Value specified in Operand	Value of Variable Symbol	Value Passed
'&&NOTATION'		'&&NOTATION'
'&MESSAGE'	BLANKS OK	'BLANKS OK'
''		''
❷ 'L''SYMBOL'		'L''SYMBOL'
❷ 'L''&VAR'	SYMBOL	'L''SYMBOL'
'&QUOTES'	❸ '' ''	''' '''
'L'SYMBOL' (Indicates end of quoted string) (Indicates beginning of a new quoted string)		INVALID OPERAND VALUE
'QUOTE1'AND'QUOTE2' ❹ No apostrophes, single ampersands, commas, blanks, or equal signs allowed between quoted strings in one operand		'QUOTE1'AND'QUOTE2'
'AB''CD'E'FGH&&' (Quoted strings)		'AB''CD'E'FGH&&'

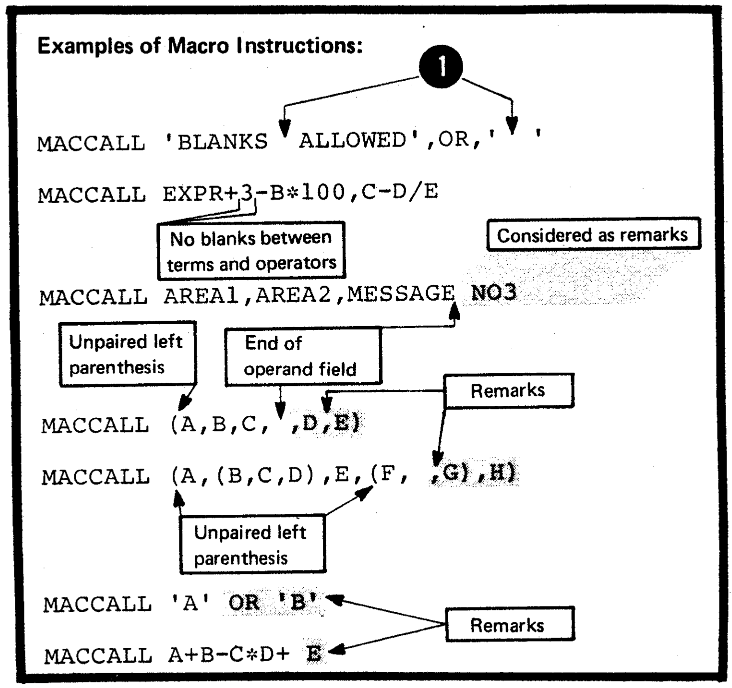
LENGTH ATTRIBUTE NOTATION: In macro instruction operand values, the
❶ length attribute notation with ordinary symbols can be used outside of quoted strings, if the length attribute notation is preceded by
❷ any special character except the ampersand.



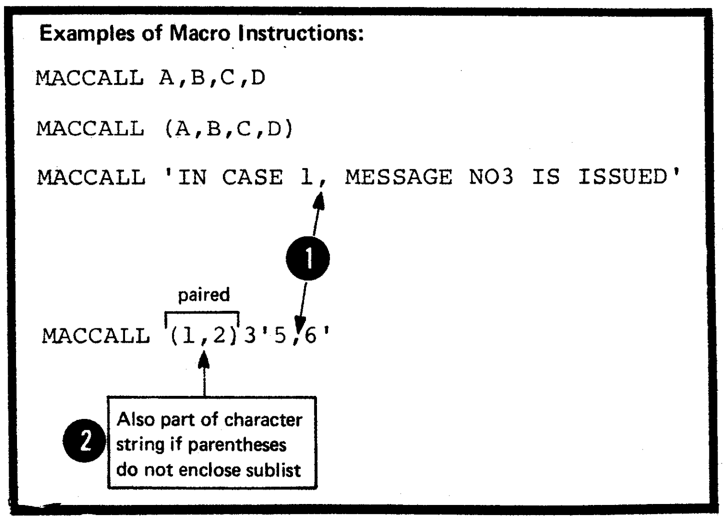
PARENTHESES: In macro instruction operand values, there must be an equal number of left and right parentheses. They must be paired,
❶ that is, to each left parenthesis
❷ belongs a following right parenthesis at the same level of nesting. An unpaired (single) left or right
❸ parenthesis can appear only in a
❹ quoted 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
❶ strings.



COMMAS: A comma outside a quoted string indicates the end of an operand value or sublist entry. Commas that do not delimit values
❶ can appear inside quoted strings
❷ or paired parentheses that do not enclose sublists.



EQUAL SIGNS: An equal sign can appear in the value of a macro instruction operand or sublist entry:

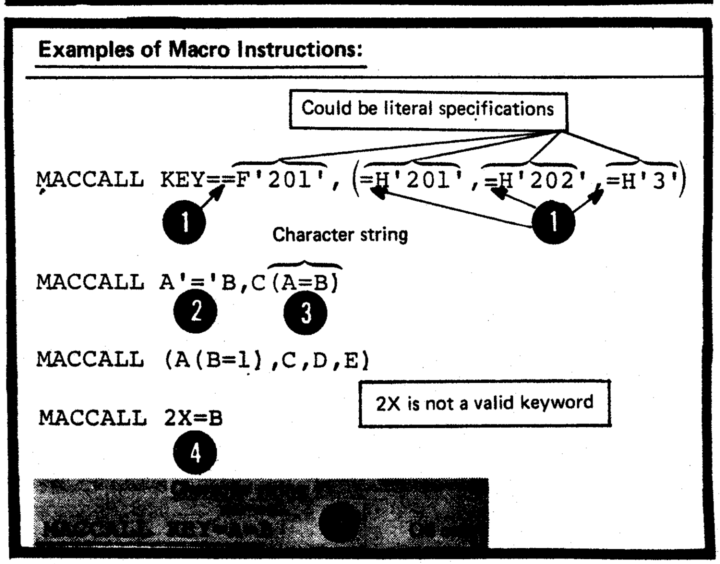
❶ As the first character,

❷ Inside quoted strings,

❸ Between paired parentheses, or

❹ In a positional parameter, provided that the parameter does not resemble a keyword parameter.

OS only ❺ In a keyword parameter



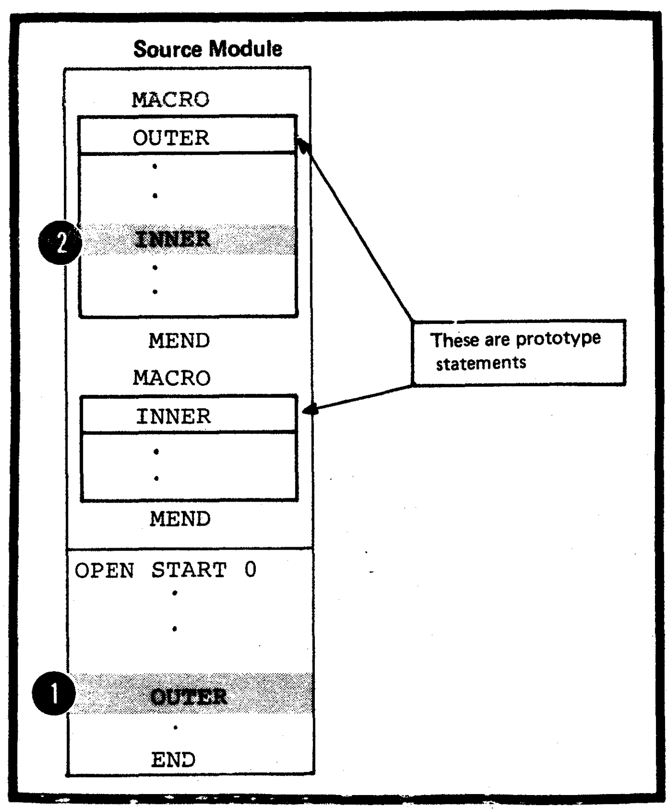
PERIODS: A period (.) can be used in the value of an operand or sublist entry. It will be passed as a period.
1 However, if it is used immediately after a variable symbol it becomes
2 a concatenation character. Then, two periods are required if one
3 is to be passed as a character.

Character String specified as value of Operand or Sublist Entry	Value of Variable Symbol	Value Passed
3.4		3.4
(3.4,3.5,3.6)		3.4 3.5 3.6
&A.1 (1)	FIELD	FIELD1
&A.1	3	31
&A..1 (2)	3	3.1 (3)
&A&B	&A=AREA	AREA200
&A.&B (1)	&B=200	AREA200
&DISP.(&BASE)	&DISP=1000 &BASE=10	1000(10)

K6 – Nesting in Macro Definitions

K6A -- PURPOSE

A nested macro instruction is a macro instruction that 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.



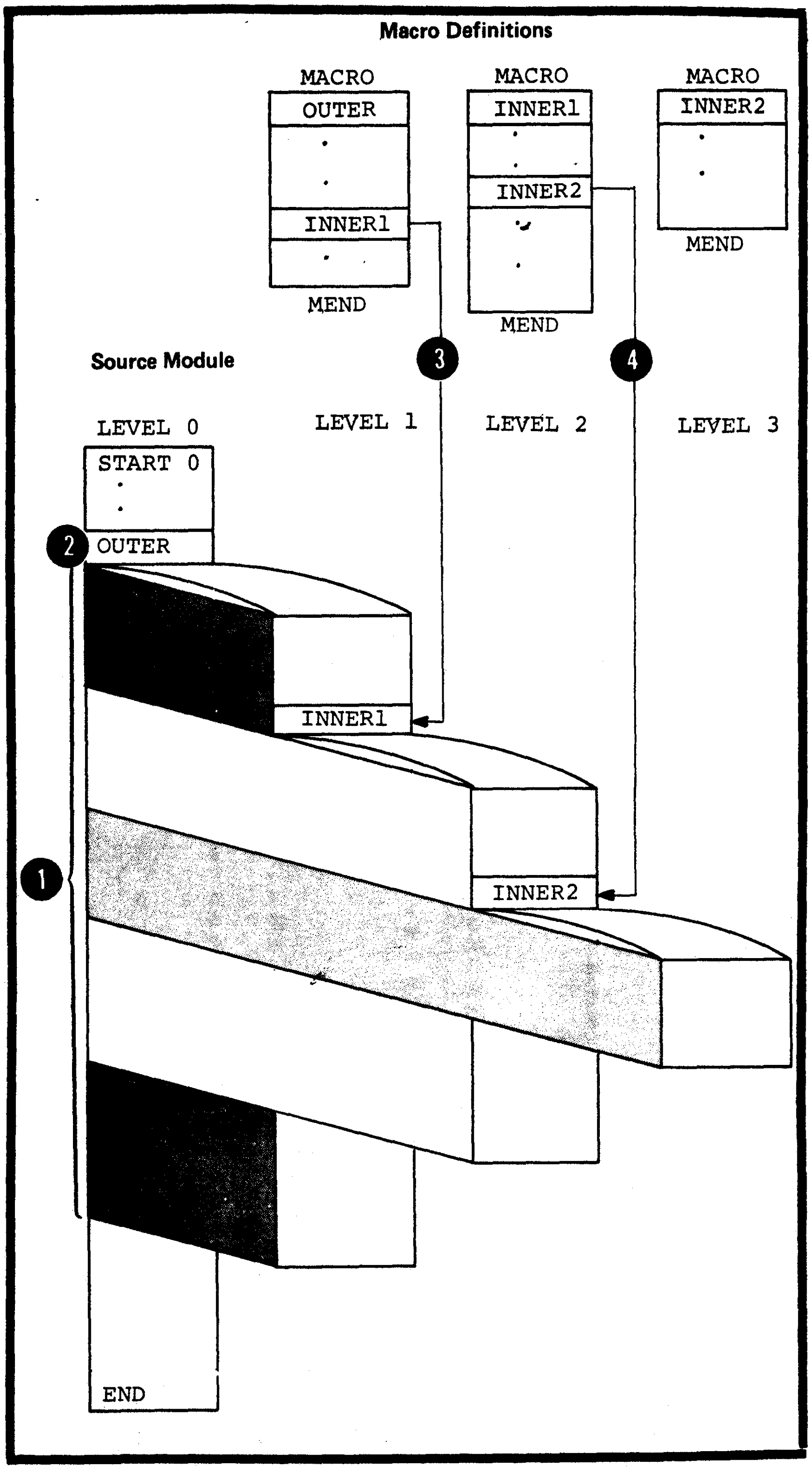
Inner and Outer Macro Instructions

Any macro instruction you write in the open code of a source module
1 is an outer macro instruction or call. Any macro instruction that appears within a macro definition
2 is an inner macro instruction or call.

Levels of Nesting

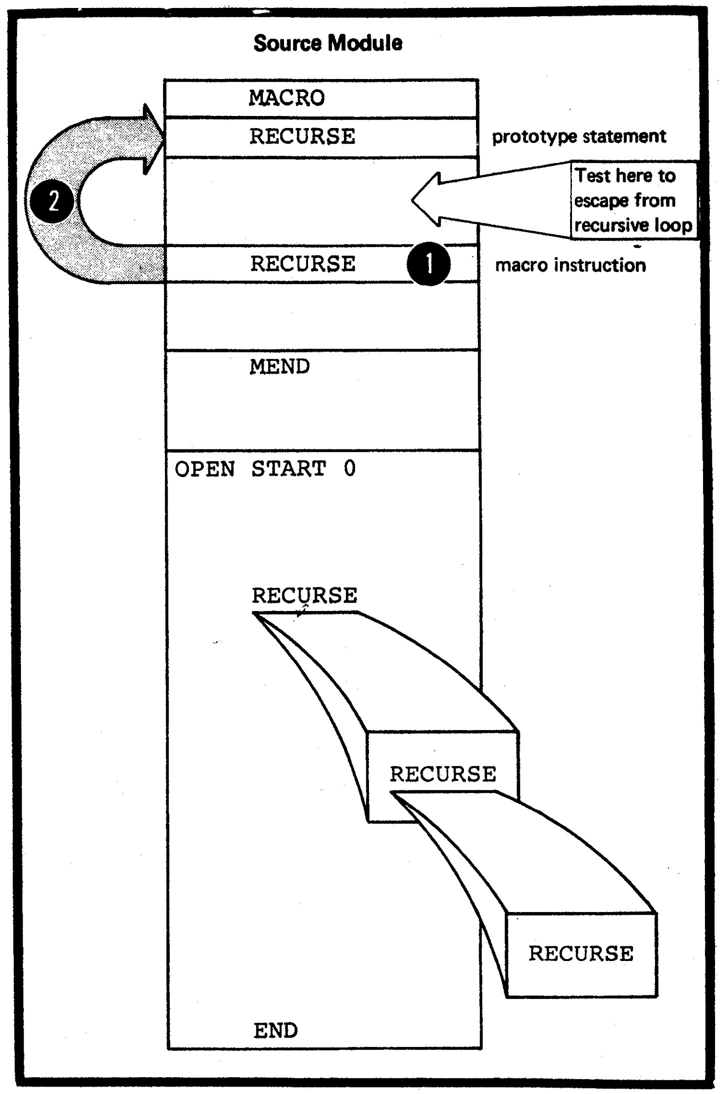
1 The code generated by a macro definition called by an inner macro call is nested inside the code generated by the macro definition that contains the inner macro call. In the macro definition called by an inner macro call, you can include a macro call to another macro definition. Thus, you can nest macro calls at different levels.

2 The zero level includes outer macro calls, calls that appear in open code; 3 the first level of nesting includes inner macro calls that appear inside macro definitions called from the zero level; 4 the second level of nesting includes inner macro calls inside macro definitions that are called from the first level, etc.


Macro Definitions
MACRO
OUTER
INNER1
MEND
MACRO
INNER1
INNER2
MEND
MACRO
INNER2
MEND
Source Module
3
4
LEVEL 0
LEVEL 1
LEVEL 2
LEVEL 3
START 0
2
OUTER
INNER1
1
INNER2
END

Recursion

You can also call a macro definition recursively, that is, you can write
1 macro instructions inside macro
2 definitions that are calls to the containing definition. This allows you to define macros to process recursive functions.



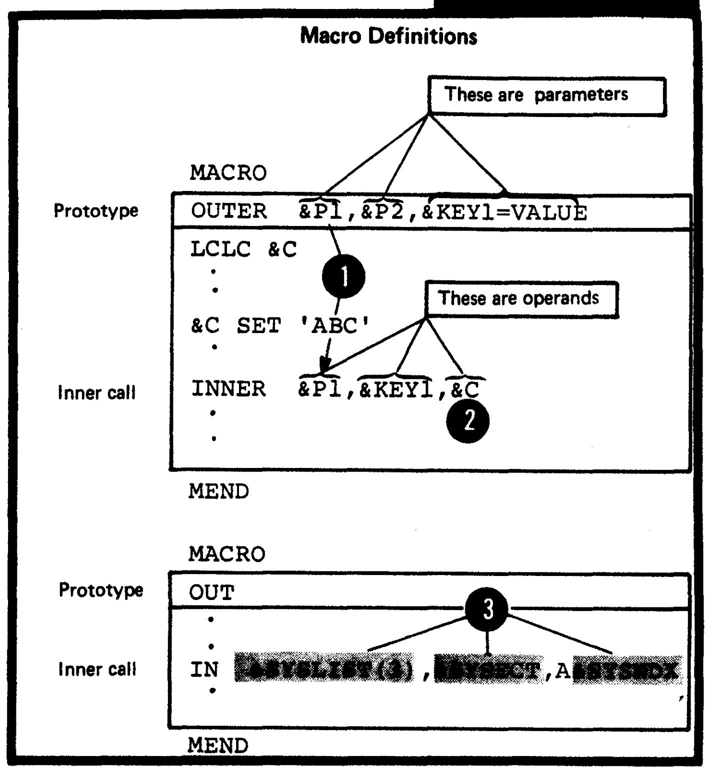
General Rules and Restrictions

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 K5 above, nested macro calls can specify values that include:

1. Any of the symbolic parameters specified in the prototype statement of the containing macro definition

2. Any SET symbols declared in the containing macro definition

3. Any of the system variable symbols (OS only) (&SYSDATE, &SYSTIME).

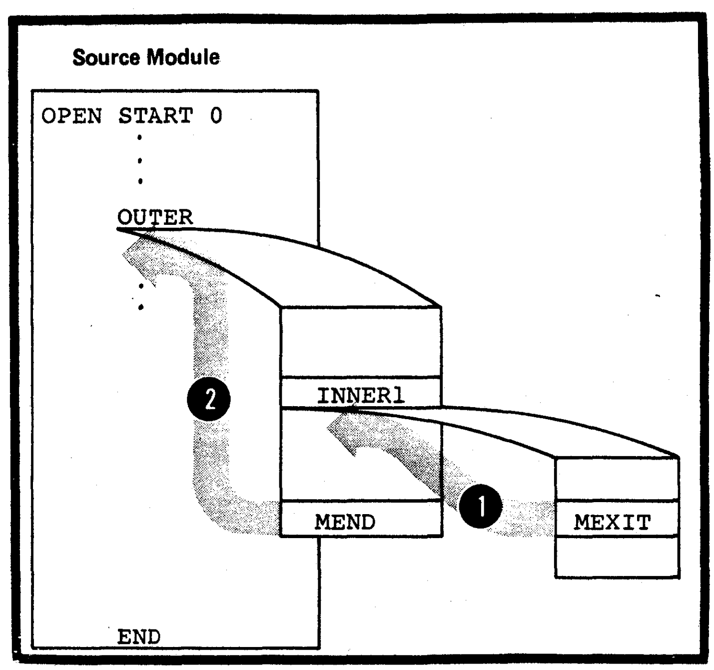


The number of nesting levels permitted depends on the complexity and size of the macros at the different levels, that is: the number of operands specified, the number of local and global SET symbols declared (see L1A) and the number of sequence symbols used.

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. (1)

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. (2)

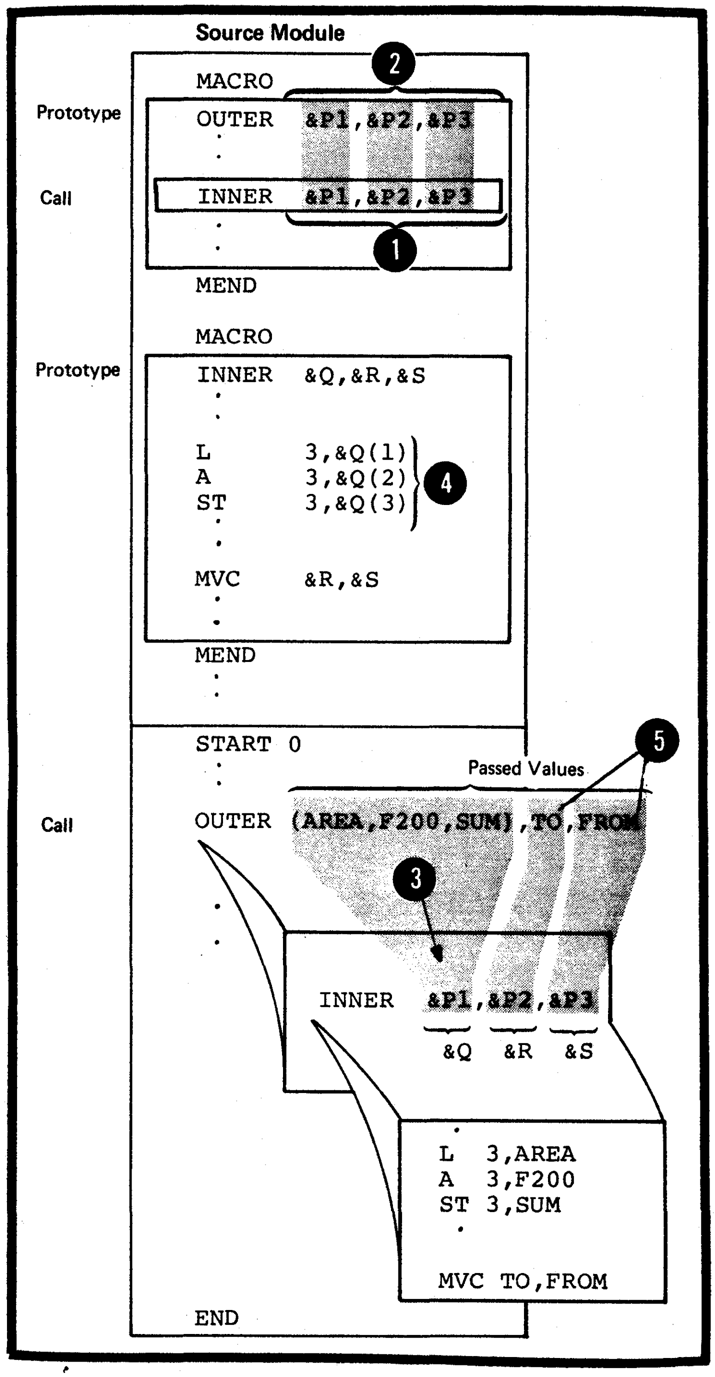


Passing Values through Nesting Levels

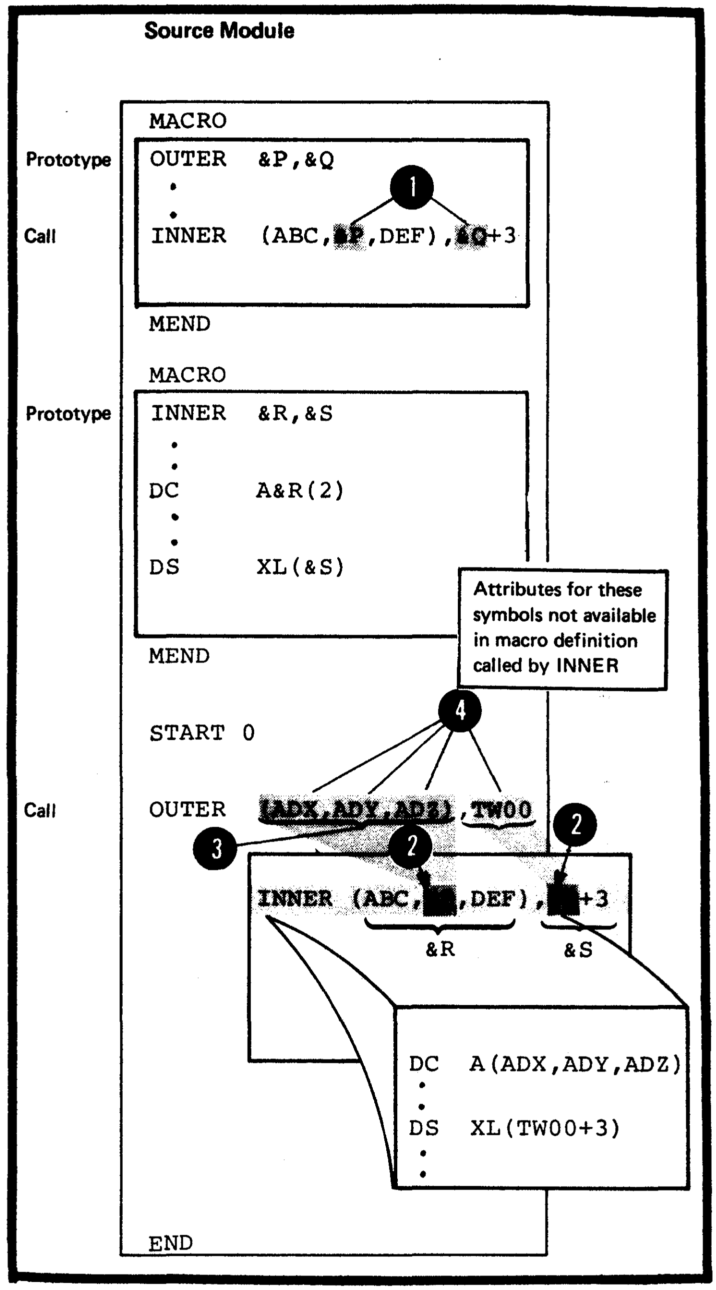
The value contained in an outer macro instruction operand can be passed through one or more levels of nesting. However, the value specified in the inner macro instruction operand must be identical to the corresponding symbolic parameter declared in the prototype of the containing macro definition.

Thus, a sublist can be passed and referred to as a sublist in the macro definition called by the inner macro call. Also, any symbol that is passed will carry its inherent attribute values through the nesting levels.

Values can be passed from open code through several levels of macro nesting if inner macro calls at each level are specified with symbolic parameters as operand values.

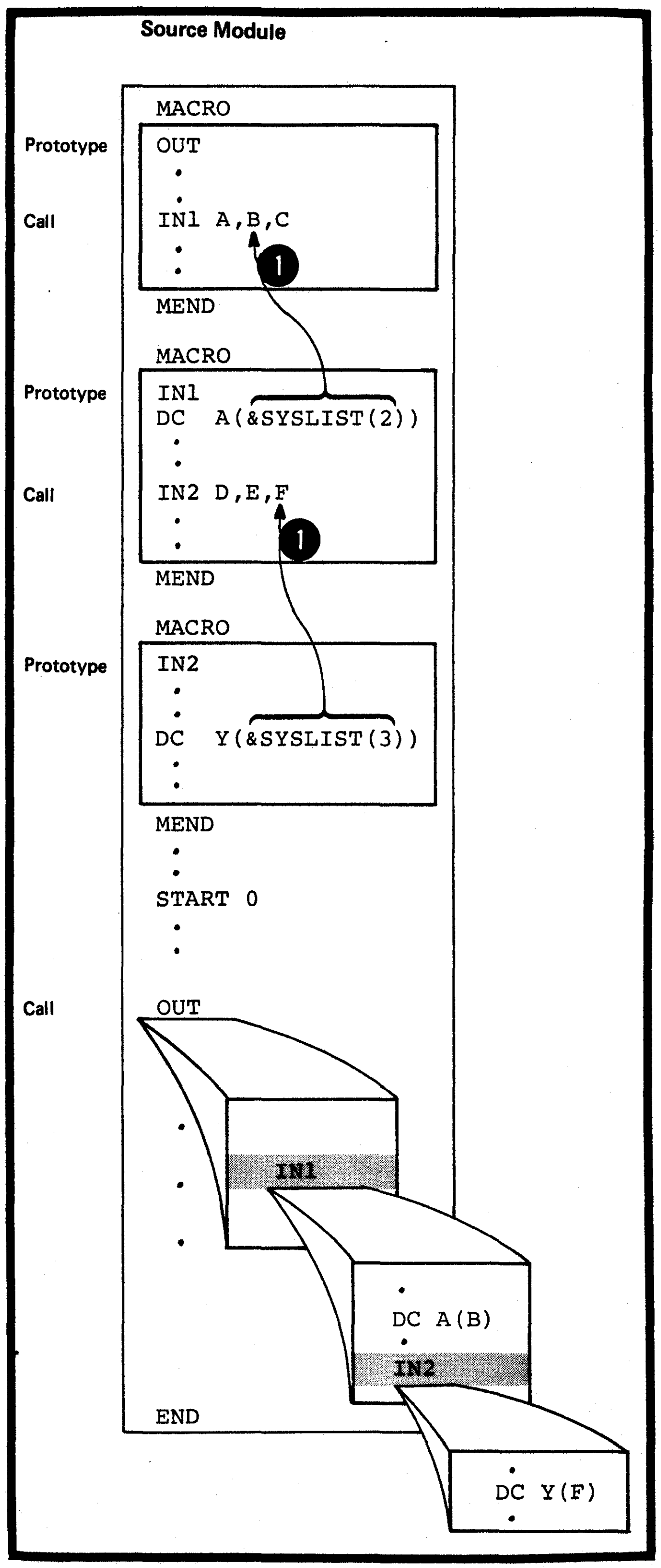


❶ NOTE: If a symbolic parameter is only a part of the value specified in an inner macro instruction ❷ operand, only the character string value given to the parameter by an outer call is passed through ❸ the nesting level. Inner sublist entries and attributes of symbols ❹ are not available for reference in the inner macro.



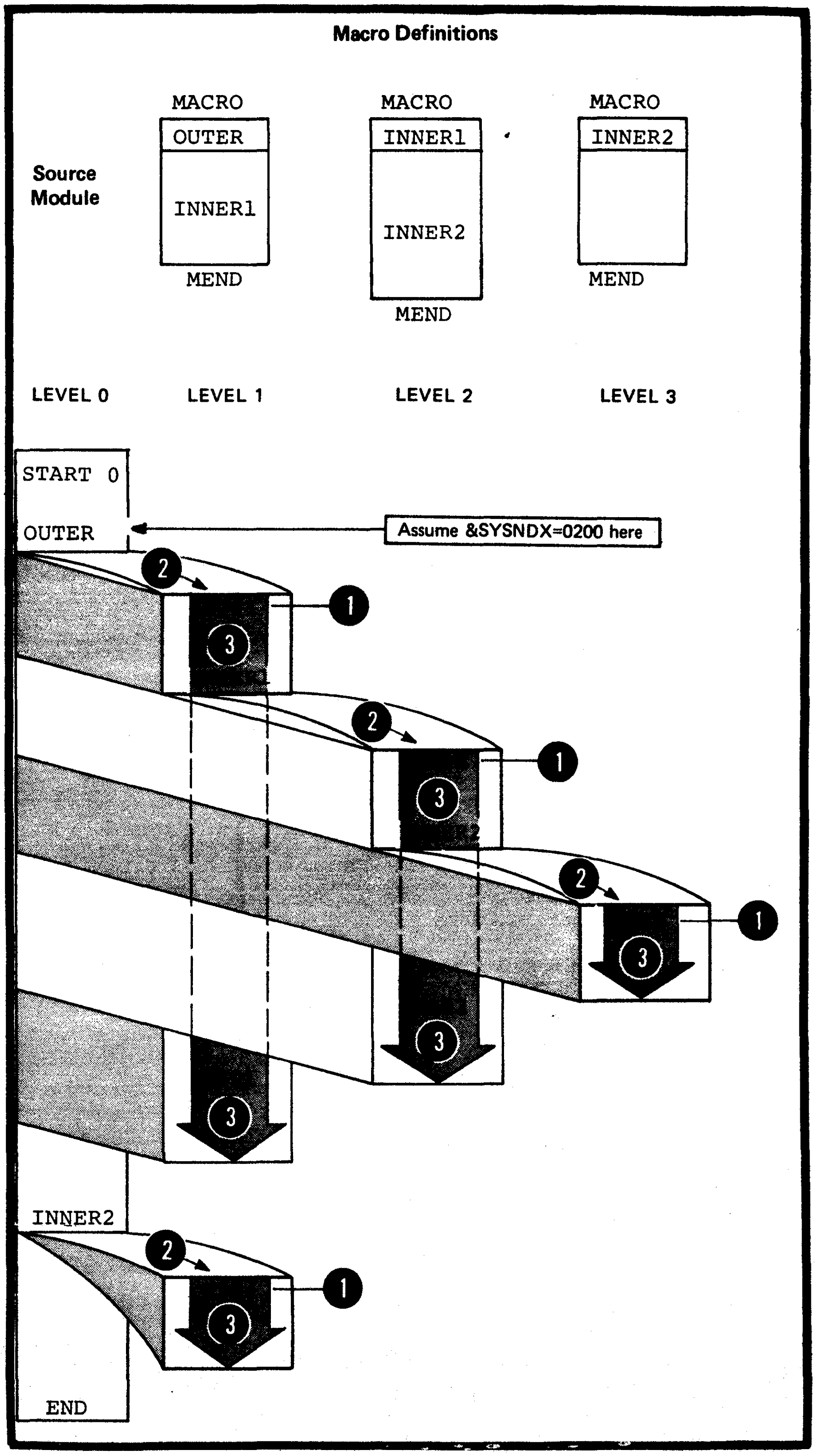
System Variable Symbols in Nested Macros

OS only 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.



1 If &SYSLIST is specified in a macro definition called by an inner macro instruction, then &SYSLIST refers to the positional operands of the inner macro instruction.

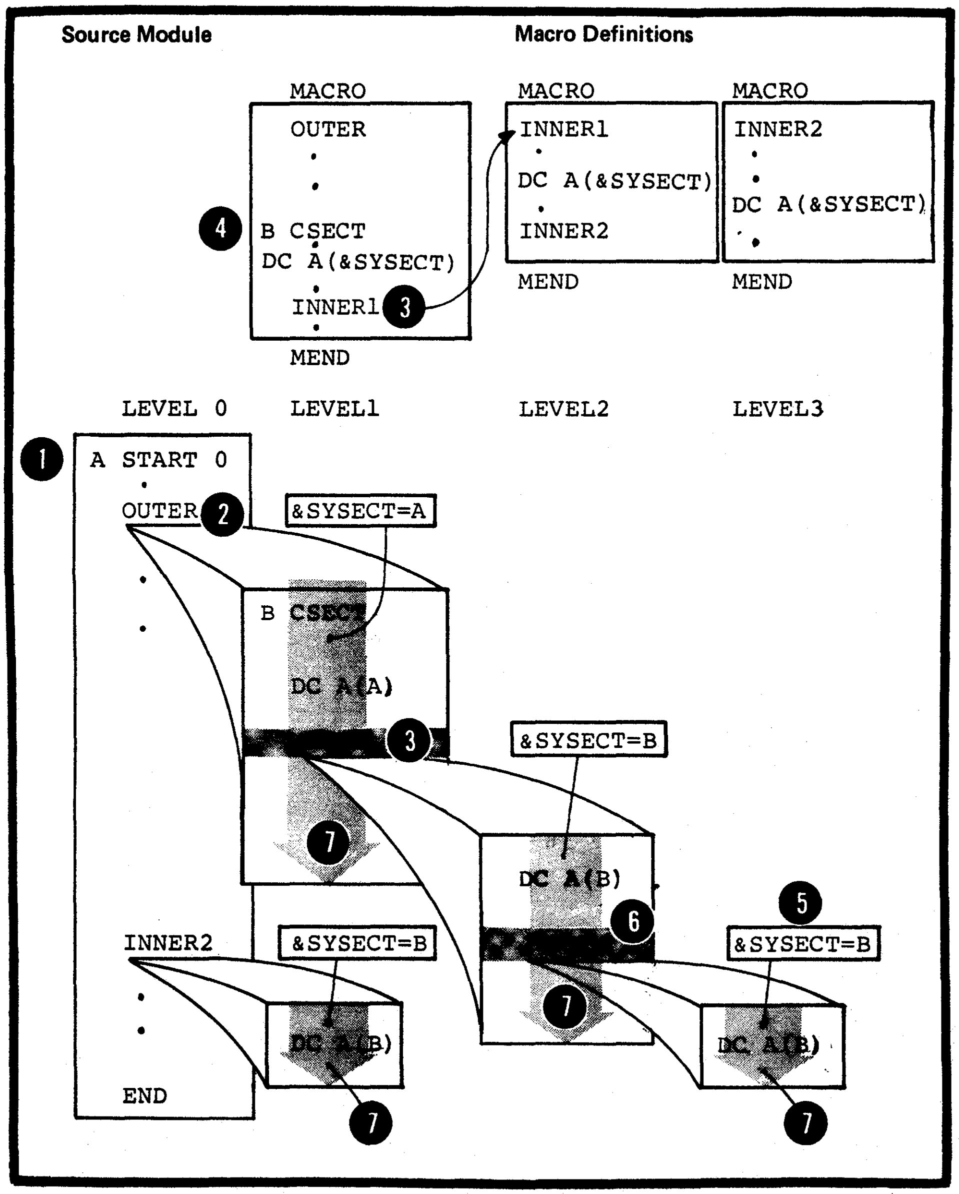
1 The assembler increments &SYSNDX by one each time it encounters a macro call. It retains the incremented value 2 throughout the expansion of the macro definition that is 3 called, that is, within the local scope of the nesting level.



1 The assembler gives &SYSECT the character string value of the name of the control section in force at the point 2 where a macro call is made. For a macro definition called by an inner macro call, the assembler will assign &SYSECT 3 the name of the control section generated in the macro definition that contains the inner macro call. The control 4 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. 5 It is the same for the next level of macro definition called by an inner macro instruction. 6

7 &SYSECT has a local scope; its read-only value remains constant throughout the expansion of the called macro definition.



Section L: The Conditional Assembly Language

This section 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:

1. Select sequences of these source statements, called model statements, from which machine and assembler instructions are generated

2. Vary the contents of these model statements during generation

The assembler processes the instructions and expressions of the conditional assembly language at pre-assembly time. Then, at assembly time, it processes the generated instructions. Conditional assembly instructions, however, are not processed after pre-assembly 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 as described in L7 below.

L1 -- Elements and Functions

The elements of the conditional assembly language are

1. SET symbols that represent data (see L1A)

2. Attributes that represent different characteristics of data (see L1B)

3. Sequence symbols that act as labels for branching to statements at pre-assembly time (see L1C).

The functions of the conditional assembly language are:

1. Declaring SET symbols as variables for use by the conditional assembly language in its computations (see L2)

2. Assigning values to the declared SET symbols (see L3)

3. Evaluating conditional assembly expressions used as values for substitution, as subscripts for variable symbols, or as condition tests for branch instructions (see L4)

4. Selecting characters from strings for substitution in and concatenation to other strings, or for inspection in condition tests (see L5)

5. Branching and exiting from conditional assembly loops (see L6).

L1A - SET SYMBOLS

Purpose

SET symbols are variable symbols that provide you with arithmetic, binary, or character data, whose values you can vary at pre-assembly time.

You can use SET symbols as:

1. Terms in conditional assembly expressions

2. Counters, switches, and character strings

3. Subscripts for variable symbols

4. Values for substitution.

Thus, SET symbols allow you to control your conditional assembly logic and to generate many different statements from the same model statement.

SUBSCRIPTED SET SYMBOLS: You can use a SET symbol to represent an array of many values. You can then refer to any one of the values of this array by subscripting the SET symbol.

The Scope of SET Symbols

You must declare a SET symbol before you can use it. The scope of a SET symbol is that part of a program for which the SET symbol has been declared.

1 If you declare a SET symbol to have a local scope, you can use it only in the statements that are part of:

2 • The same macro definition or

3 • Open code.

4 If you declare a SET symbol to have a global scope, you can use it in the statements that are part of:

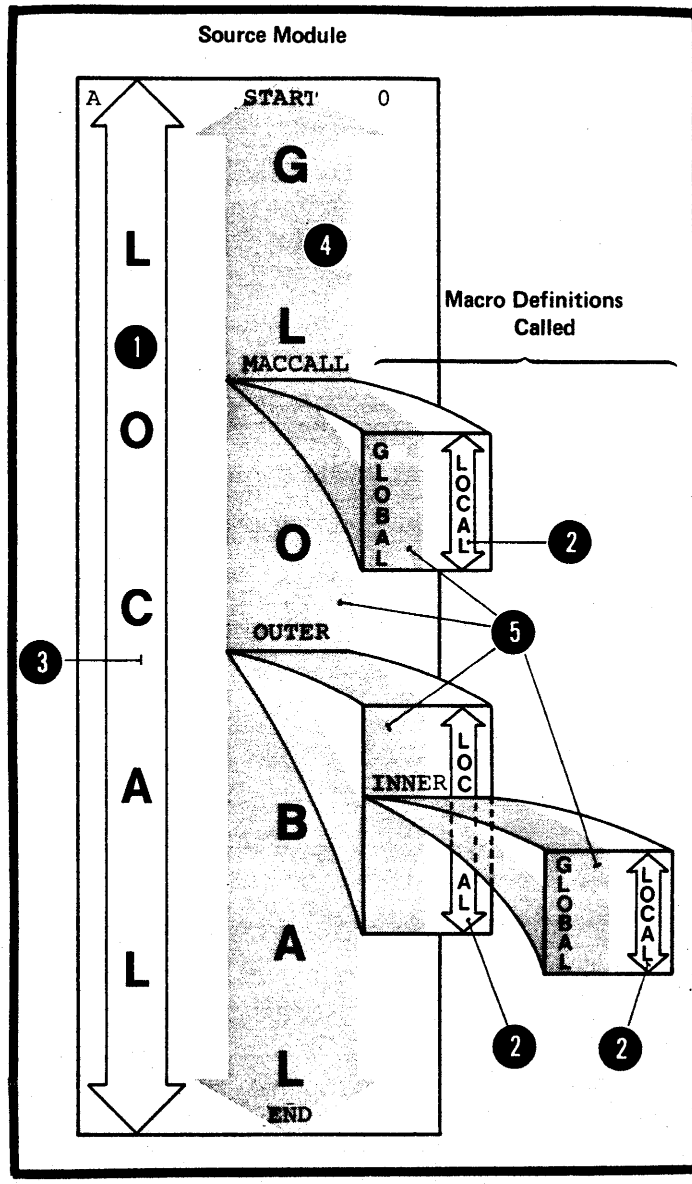
- The same macro definition, and
- A different macro definition, and
- Open code.

You must, however, declare the SET symbol as global for each part of 5 the program (a macro definition or open code) in which you use it.

You can change the value assigned to a SET symbol without affecting the scope of this symbol.

THE SCOPE OF OTHER VARIABLE SYMBOLS: A symbolic parameter has a local scope. You can use it only in the statements that are part of the macro definition for which the parameter is declared. You declare a symbolic parameter in the prototype statement of a macro definition.

The system variable symbols, &SYSLIST, &SYSECT, and &SYSNDX have a local scope; you can use them only inside macro definitions. OS only However, the system variable symbols, &SYSPARM, **&SYSDATE, and &SYSTIME** have a global scope; you can use them in both open code and inside any macro definition.

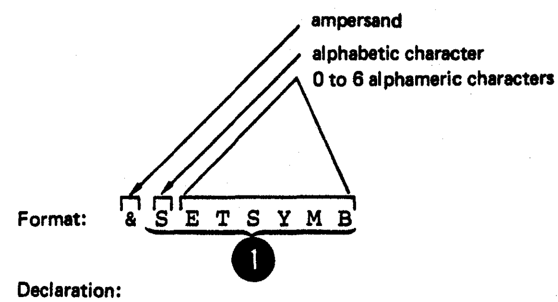


Specifications

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. (1) A SET symbol must be a valid variable symbol, as shown in the figure to the right.

(2) A SET symbol must be declared before it can be used. The instruction that declares a SET symbol determines its scope and type (see L2).

SET Symbols



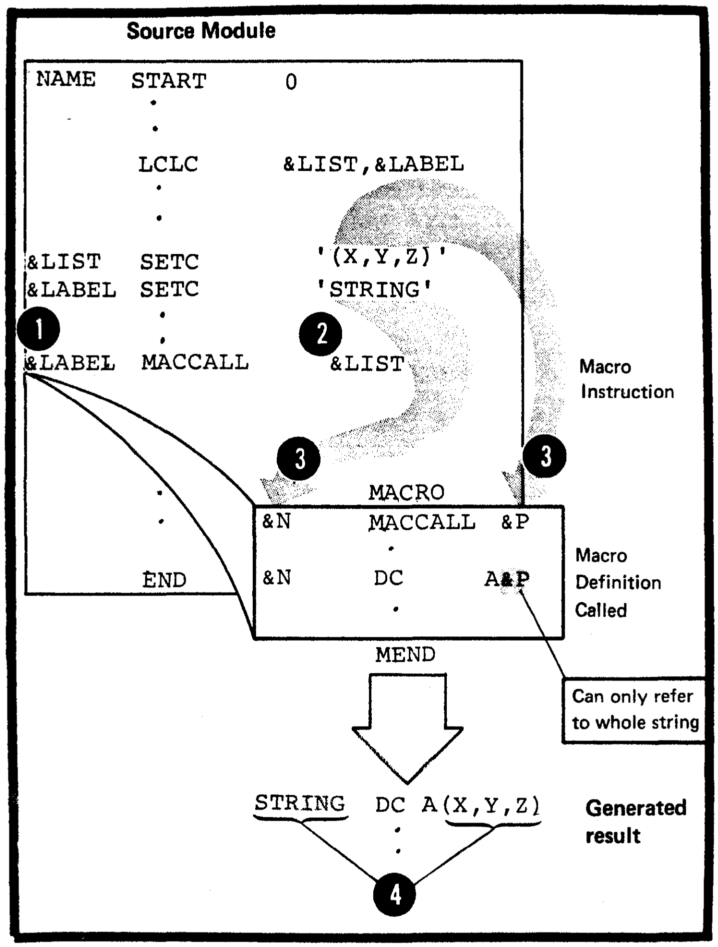
Declaration:

Instruction (2) Operation	Operand	Type	Scope
LCLA	&ARITH	SETA	local
LCLB	&BOOLEAN	SETB	local
LCLC	&CHAR	SETC	local
GBLA	&A	SETA	global
GBLB	&B	SETB	global
GBLC	&C	SETC	global

The features of SET symbols and other types of variable symbol are compared in the figure to the right.

The value assigned to a SET symbol (1) can be changed by using the SETA, SETB, or SETC instruction within the declared scope of the SET symbol. However, a symbolic parameter and the system variable symbols are (2) 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.

Feature	Types of Variable Symbol: SETA, SETB, or SETC Symbols	Symbolic Parameters	System Variable Symbols
Can be used: In open code	YES	NO	only: &SYSPARM OS only &SYSDATE &SYSTIME
In macro definitions	YES	YES	All
Scope: Local or	YES	YES	&SYSLIST &SYSECT &SYSNDX
Global	YES	NO	&SYSPARM OS &SYSDATE &SYSTIME
Values can be changed within scope of symbol	(1) YES	(2) NO: read only value	(2) NO: read only value



(1) NOTE: SET symbols can be used in (2) the name and operand field of macro instructions. However, the value (3) thus passed through a symbolic parameter into a macro definition (4) is considered as a character string and is generated as such.

[redacted] The "LCLC &LIST,&LABEL" instruction must precede the START instruction.

Subscripted SET Symbols - Specifications

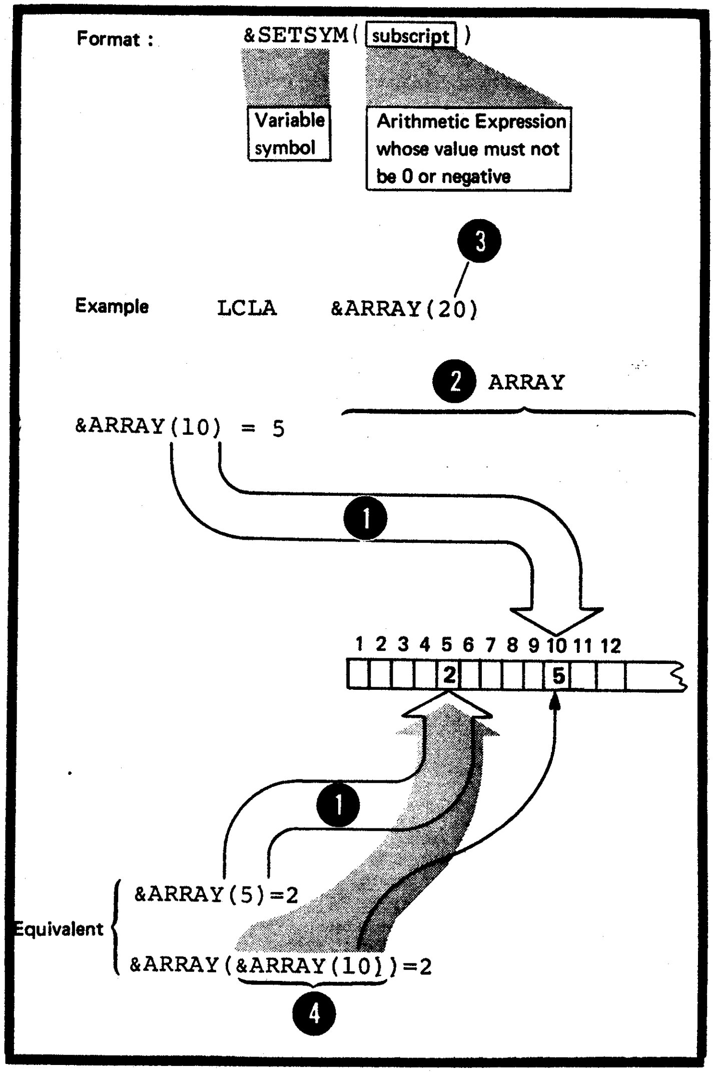
A subscripted SET symbol must be specified as shown in the figure to the right.

The subscript can be any arithmetic expression allowed in the operand field of a SETA instruction (see L4A).

A subscripted SET symbol can be used anywhere an unsubscripted SET symbol is allowed. However, subscripted SET symbols must be declared as subscripted by a previous local or global declaration instruction.

1 2 3 The subscript refers to one of the many positions in an array of values identified by the SET symbol. The value of the subscript must not exceed the dimension declared for the array in the corresponding LCLA, LCLB, LCLC, GBLA, GBLB, or GBLC instruction.

4 NOTE: The subscript can be a subscripted SET symbol. Five levels of subscript nesting are allowed.



L1B - DATA ATTRIBUTES

What Attributes Are

The data, such as instructions, constants, and areas, which you define in a source module can be described in terms of:

1. Type, which distinguishes one form of data from another: for example, fixed-point constants from floating-point constants, or machine instructions from macro instructions.

2. Length, which gives the number of bytes occupied by the object code of the data.

3. Scaling, which indicates the number of positions occupied by the fractional portion of fixed-point and decimal constants in their object code form.

4. Integer, which indicates the number of positions occupied by the integer portion of fixed-point and decimal constants in their object code form.

5. 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.

6. Number, which gives the number of sublist entries in a macro instruction operand.

These six characteristics are called the attributes of the data. The assembler assigns attribute values to the ordinary symbols and variable symbols that represent the data.

Purpose

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 in the figure to the right.

(1) NOTE: The number attribute of &SYSLIST(m) and &SYSLIST(m,n) is described in J7C.

Attribute	Purpose	Main Uses
Type	Gives a letter that identifies type of data represented	- In tests to distinguish between different data types - For value substitution - In macros to discover missing operands
Length	Gives number of bytes that data occupies in storage	- For substitution into length fields - For computation of storage requirements
Scaling	Refers to the position of the decimal point in decimal, fixed-point and floating-point constants	- For testing and regulating the position of decimal points - For substitution into a scale modifier
Integer	Is a function of the length and scaling attributes of decimal, fixed-point, and floating-point constants	- To keep track of significant digits (integers)
Count	Gives the number of characters required to represent data	- For scanning and decomposing of character strings - As indexes in substring notation
Number (1)	Gives the number of sublist entries in a macro instruction operand sublist	- For scanning sublists - As counter to test for end of sublist

Specifications

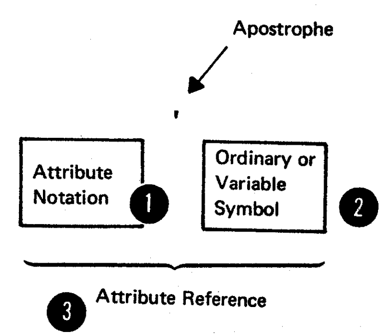
FORMAT: The format for an attribute reference is shown in the figure to the right.

(1) The attribute notation indicates the attribute whose value is desired.
(2) The ordinary or variable symbol represents the data which possesses the attribute. The assembler substitutes the value of the attribute for the attribute reference.
(3)

WHERE ALLOWED: 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 (for details on this use see C4C).

Attributes

Format:



Examples:

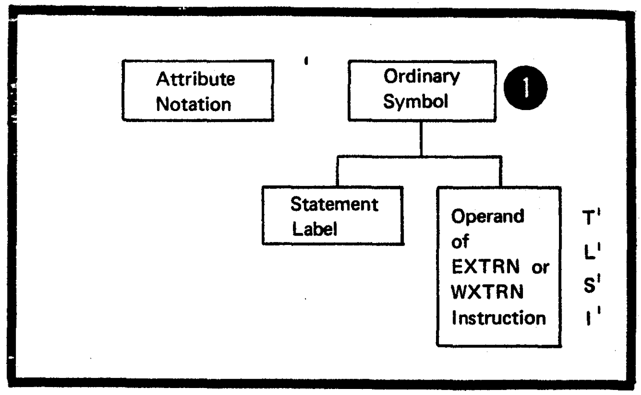
```
T'SYMBOL
L'&VAR
K'&PARAM
```

COMBINATION WITH SYMBOLS: The figure below shows the six kinds of attributes and the type of symbol with which the attributes can be combined.

NOTE: Whether or not an attribute reference is allowed in open code, in macro definitions, or in both, depends on the type of symbol specified.

	Symbols Specified	ATTRIBUTES SPECIFIED Type T'	Length L'	Scaling S'	Integer I'	Count K'	Number N'
IN OPEN CODE	Ordinary Symbols	YES	YES	YES	YES	YES	YES
	SET Symbols	YES	NO	NO	NO	YES	NO
	System Variable Symbols: &SYSPARM, &SYSDATE, &SYSTIME	YES	NO	NO	NO	YES	NO
IN MACRO DEFINITIONS	Ordinary Symbols	YES	YES	YES	YES	NO	NO
	SET Symbols	YES	NO	NO	NO	YES	NO
	Symbolic Parameters	YES	YES	YES	YES	YES	YES
	System Variable Symbols. &SYSLIST	YES	YES	YES	YES	YES	YES
	&SYSNDX, &SYSPARM, &SYSDATE, &SYSECT, &SYSTIME	YES	NO	NO	NO	YES	NO
DOS IN OPEN CODE	Ordinary Symbols	YES	YES	YES	YES	NO	NO
DOS IN MACRO DEFINITIONS	Ordinary Symbols	NO	YES	NO	NO	NO	NO
	Symbolic Parameters	YES	YES	YES	YES	YES	YES
	System Variable Symbol &SYSLIST	YES	YES	YES	YES	YES	YES

ORIGIN OF VALUES: The value of
1 an attribute for an ordinary symbol specified in an attribute reference comes from the data represented by the symbol, as shown in the figure to the right.



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:

1. Must appear in open code

2. Must not contain any variable symbols, and

3. Must not be a generated instruction.

1 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 (see also the figure to the right):

OS only 1. **For SET symbols and the system variable symbols &SYSECT, &SYSNDX, &SYSPARM, &SYSDATE, and &SYSTIME, the attribute values come from the**
2 **current data value of these symbols.**

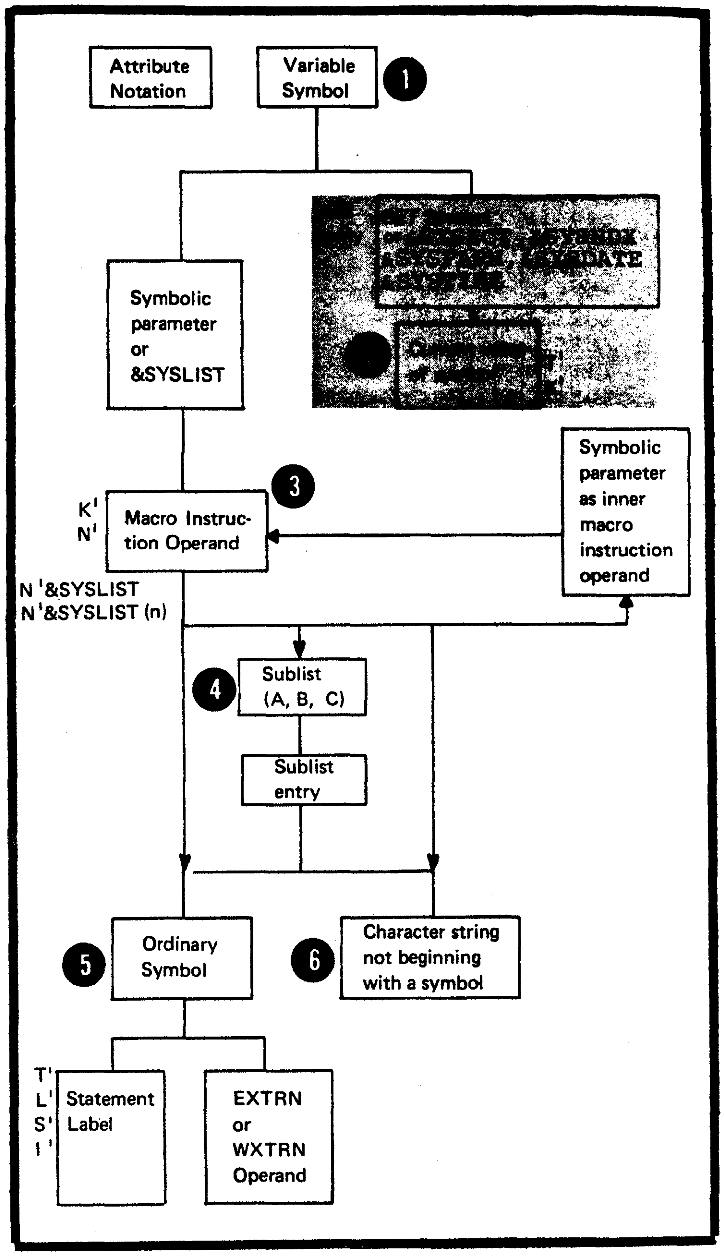
2. For symbolic parameters and the system variable symbol, &SYSLIST, the values of the count and number attributes come from the operands
3 of macro instructions.

The values of the type, length, scaling, and integer attributes, however, come from the values represented by the macro instruction operands, as follows:

4 a. If the operand is a sublist, the sublist as a whole has 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).

5 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 values of the attributes for the operand are the same as for the ordinary symbol.

6 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.



VALUES: Because attribute references are allowed only in conditional assembly instructions, their values are available only at pre-assembly time, except for the length attribute, which can be referred to outside conditional assembly instructions and is therefore also available at assembly time (see C4C).

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.

THE TYPE ATTRIBUTE (T'): The type attribute has a value of a single alphabetic character that indicates the type of data represented by:

❶ • An ordinary symbol

DOS only NOTE: An ordinary symbol outside a macro cannot be used as the operand of T' inside a macro in DOS assembler.

❷ • A macro instruction operand

NOTE: The type attribute of a sublist is set to the same value as the type attribute of the first element of the sublist.

OS only • A SET symbol. ❸

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.

NOTE: Ordinary symbols used in the name field of an EQU instruction have the type attribute value "U".

OS only **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.**

T'

Type Attribute	Data Characterized
	❶ For ordinary symbols and outer macro instructions that are symbols
	: Defined as labels for DC and DS instructions
A	A-type constant, implicit length, aligned (also **CXD** **OS only** 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	Full-word fixed-point constant, implicit length, aligned
G	Fixed-point constant, explicit length
H	Half-word fixed-point constant, implicit length, aligned
K	Floating-point constant, explicit length
L	Extended floating-point constant, implicit length, aligned
P	Packed decimal constant
OS only **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
	: Defined as labels for assembler language statements
I	Machine instruction
M	Macro Instruction
W	CCW instruction
J	: Identified as control section name
T $	: Identified as external symbol by EXTRN or WXTRN instruction
❷	A macro Instruction Operand that is:
N	A self-defining term
O	Omitted (has a value of a null character string)
OS only **N** ❸	**The value of a SETA or SETB variable**

When a symbol or macro instruction operand cannot be assigned any of the type attribute values listed in the preceding figure, the data represented is considered to be undefined and its type attribute is U. Specific cases of where U is assigned as a type attribute value are given in the figure to the right.

1 The type attribute will be set to U when the same ordinary symbol is used to define a label in more than one place, even though only one label will be generated by conditional assembly statements.

The Type Attribute Value=U is assigned to the following:
Ordinary symbols that are used as labels: ► for the LTORG instruction ► for the EQU instruction without a third operand ► that are defined more than once **1** ► for DC and DS statements that contain variable symbols **Example:** `U1 DC &X'1'` **DOS only** ► **for DC and DS statements that contain expressions as duplication factors** **Example:** `DC (AA BB)F'15'`
The SETC variable symbol
OS only **The system variable symbols: &SYSPARM, &SYSDATE, and &SYSTIME**
Macro instruction operands that specify literals. **Inner macro instruction operands that are ordinary symbols.**

1 <u>THE LENGTH ATTRIBUTE (L'):</u> The length attribute has a <u>numeric value</u> equal to the number of bytes occupied by the data that is represented by the symbol specified in the attribute reference.

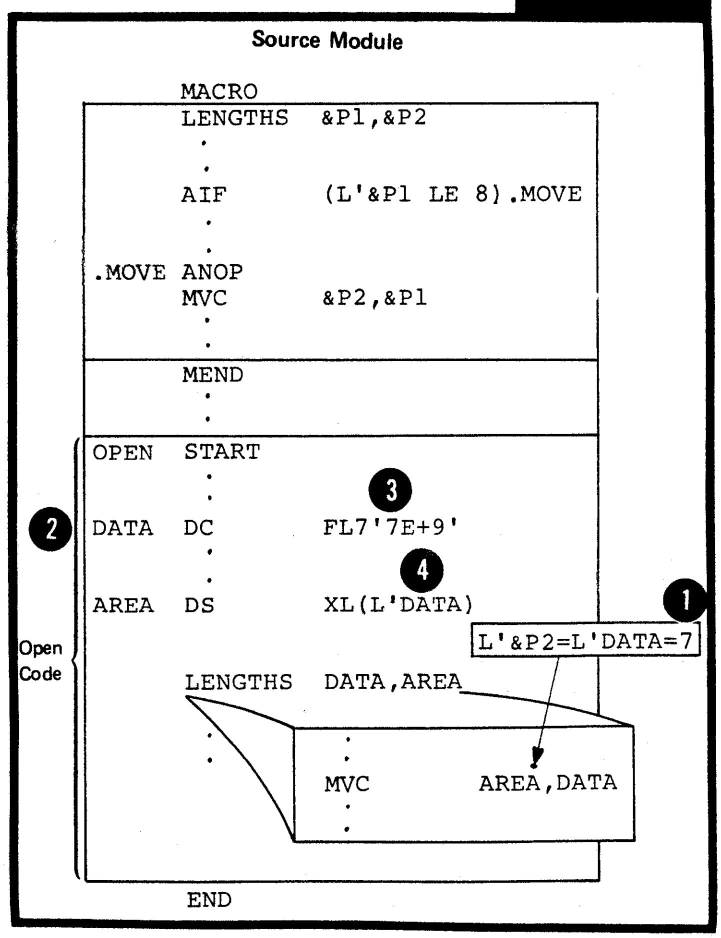
2 If the length attribute value is desired for pre-assembly processing, the symbol specified in the attribute reference must ultimately represent the name entry of a <u>statement in open code.</u> In such a statement, the length modifier (for DC and DS instructions) or the length field (for a machine instruction), if specified, must be a <u>self-defining term.</u> **3** 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.

4 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 <u>assembly time.</u>

At pre-assembly 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.

OS only **However, the second operand of an EQU instruction can be used to assign a length attribute value to the symbol in the name field.**

L



NOTES:

1. The length attribute reference, when used in conditional assembly processing, can be specified only in arithmetic expressions (see L4).

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.

THE 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 in the figure to the right.

NOTES:

1. The scaling attribute reference can be used only in arithmetic expressions (see L4).

2. When no scaling attribute value can be determined, the reference is flagged and the scaling attribute is given the value of 1.

S'

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 modifer (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)

Examples:

```
PACKED   DC   P'+12.345'     S'PACKED=3
ZONED    DC   Z'+12.345'     S'ZONED=3
```

THE 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 the figure below.

NOTE: The integer attribute reference can be used only in arithmetic expressions (see L4).

I'

Constant Type Allowed (attribute value)	Formula Relating the Integer to the Length and Scaling Attributes	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' ≤ 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

THE 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 that is equal to the number of characters:

- (1) That constitute the macro instruction operand, or
- (2) (3) OS only: 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 (see L4).

2. The count attribute of an omitted macro instruction operand has a default value of 0.

Macro Instruction Operands (1) — All characters of operand are included	Value of Count Attribute
ALPHA	5
(SUB,LIST,ALL)	14
2(10,12)	8
'A''B'	6
' ' blank	3
'' null character string (omitted operand)	2
(omitted operand)	0

SET Symbols (2) — Delimiting apostrophes not included

			OS only	
&C	SETC	'ALPHA'	K'&C=	5
&C	SETC	' '	K'&C=	1
&C	SETC	''	K'&C=	0
&B	SETB	1	K'&B=	1
&B	SETB	0	K'&B=	1
&A	SETA	399	K'&A=	3
&A	SETA	X'FF' (255)	K'&A=	3
&A	SETA	0100 (leading zeros are not counted)	K'&A=	3

System Variable Symbols (3)

&SYSNDX=	0912 (leading zeros are counted)	K'&SYSNDX	4	OS only

THE NUMBER ATTRIBUTE (N'): The number attribute applies only to the operands of macro instructions. It has a numeric value that is equal
1 to the number of sublist entries in the operand.

NOTES:

1. The number attribute reference can be used only in arithmetic expressions (see L4).

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 (for further details on the number attribute of &SYSLIST, see J7C).

Macro Instruction Operand Sublist	Value of Number Attribute
	1+ number of commas separating the entries (1)
(A,B,C,D,E)	5
(A,,B,C,D,E) (omitted entry)	6
(,B,C,D) (omitted entry)	4
(A)	1
A (When operand is not a sublist)	1
(No operands)	0

L1C - SEQUENCE SYMBOLS

Seq. Sym.

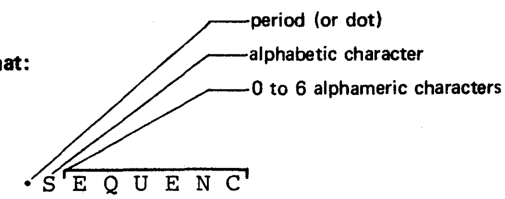
Purpose

You can use a sequence symbol in the name field of a statement to branch to that statement at pre-assembly time, thus altering the sequence in which the assembler processes your conditional assembly and macro instructions. You can thereby select the model statements from which the assembler generates assembler language statements for processing at assembly time.

Specifications

Sequence symbols must be specified as shown in the figure to the right.

Format:



Examples: .SEQ

.A1234

.#924

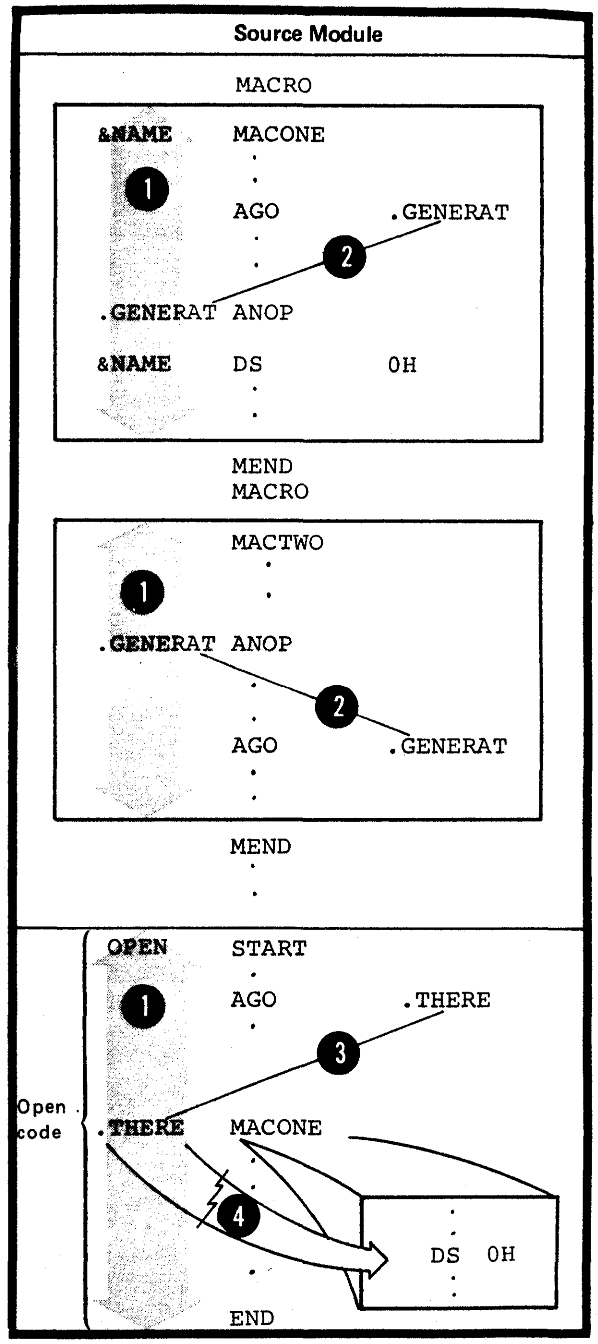
Sequence symbols can be specified in the name field of assembler language statements and model statements, except as noted in the figure to the right.

Statements in which sequence symbols must not be used as name entries
The following assembler instructions: COPY EQU GBLA GBLB GBLC ICTL ISEQ LCLA LCLB LCLC MACRO OPSYN DOS DSECT
The Macro prototype instruction
Any instruction that already contains an ordinary symbol or 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.

1 A sequence symbol has a local scope. Thus, if a sequence symbol is used in an AIF or 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
2 the same macro definition or in
3 open code.

NOTE: A sequence symbol in the name field of a macro instruction
4 is not substituted for the parameter, if specified, in the name field of the corresponding prototype statement (for specifications about the name entry of macro instructions see K2A).



L2 – Declaring Set Symbols

You must declare a SET symbol before you can use it. In the declaration, you specify whether it is to have a global or local scope. The assembler assigns an initial value to a SET symbol at its point of declaration.

L2A -- THE LCLA, LCLB, AND LCLC INSTRUCTIONS

Purpose

You use the LCLA, LCLB, and LCLC instructions to declare the local SETA, SETB, and SETC symbols you need.

Specifications

The format of the LCLA, LCLB, and LCLC instruction statements is given in the figure to the right.

These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

DOS The LCLA, LCLB, and LCLC instructions, if specified, must appear immediately following any GBLA, GBLB, or GBLC instructions that may be specified.

If specified inside a macro definition, the global declaration instructions must appear immediately following the macro prototype statement. If specified outside a macro definition, the global declarations must appear first in open code; that is, they must follow any source macro definitions specified and precede the beginning of the first control section.

LCLA
LCLB
LCLC

Name	Operation	Operand
Blank	LCLA, LCLB,or LCLC	One or more variable symbols separated by commas

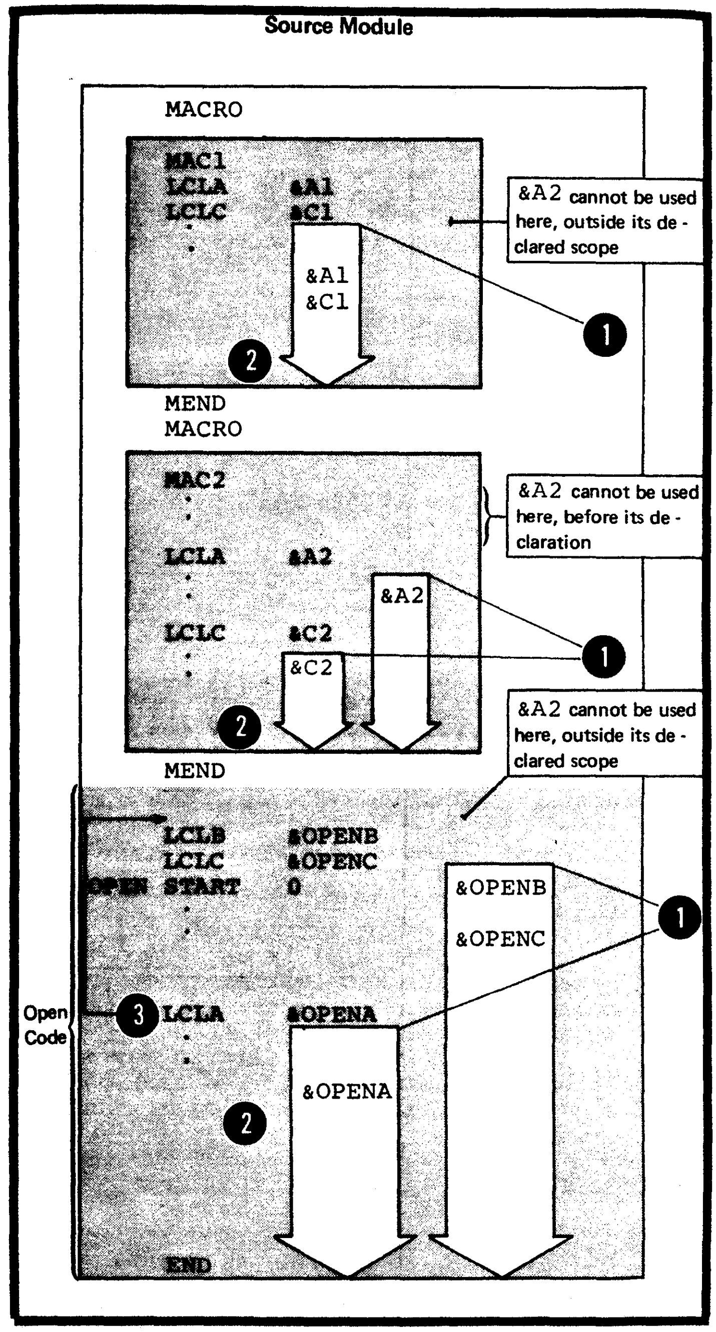
Any variable symbols declared in the operand field have a local scope. They can be used as SET

1 symbols anywhere after the pertinent LCLA, LCLB, or LCLC instructions,

2 but only within the declared local scope.

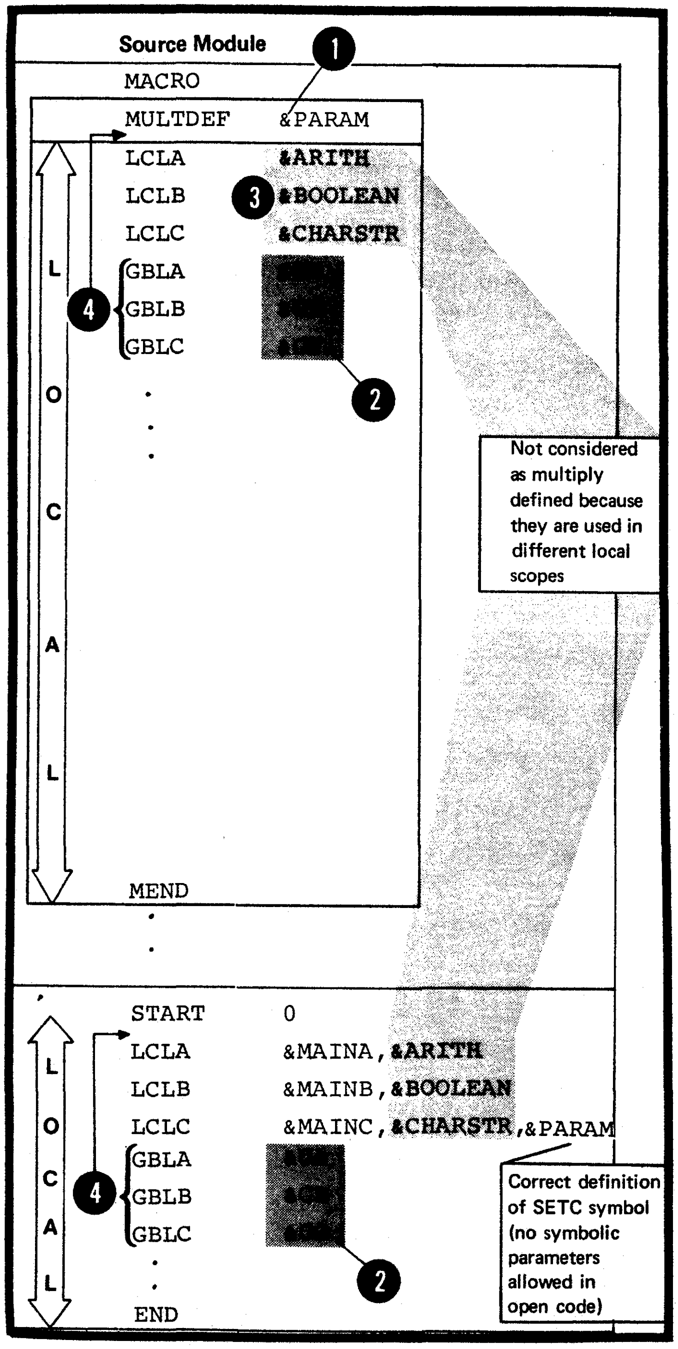
DOS NOTE: The "LCLA &OPENA" instruction must precede the START

3 instruction.



The assembler assigns initial values to these SET symbols as shown in the figure to the right.

Instruction	Initial Value assigned to SET variable symbols in operand fields
LCLA	0
LCLB	0
LCLC	Null character string



<u>LOCAL VARIABLE SYMBOLS MUST NOT BE MULTIPLY DEFINED:</u> A local SET variable symbol declared by the LCLA, LCLB, or LCLC instruction must not be identical to any other variable symbol used within the same local scope. The following rules apply to a local SET variable symbol:

1. Within a macro definition, it must not be the same as any <u>symbolic parameter</u> declared in the prototype statement. **(1)**

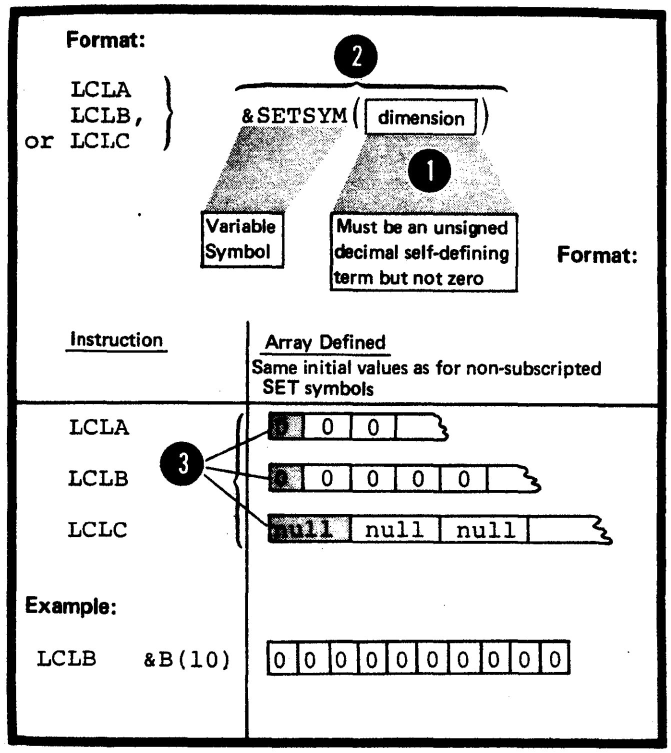
2. It must not be the same as any <u>global variable symbol</u> (see L2B) declared within the same local scope. **(2)**

3. The same variable symbol must not be declared or used as two <u>different types of SET symbols,</u> for example, as a SETA and a SETB symbol, within the same local scope. **(3)**

NOTE 1: A local SET symbol should not begin with the four characters &SYS, which are reserved for system variable symbols (see J7).

DOS NOTE 2: The <u>global declarations</u> must precede the local declarations. **(4)**

SUBSCRIPTED LOCAL SET SYMBOLS: A local subscripted SET symbol is declared by the LCLA, LCLB, or LCLC instruction. This declaration must be specified as shown in the figure to the right.



1 The maximum dimension allowed is 32,767.

DOS The maximum dimension allowed is 4095.

The dimension indicates the number 2 of SET variables associated with the subscripted SET symbol. The 3 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.

L2B -- THE GBLA, GBLB, AND GBLC INSTRUCTIONS

GBLA GBLB GBLC		
Name	**Operation**	**Operand**
Blank	GBLA, GBLB, or GBLC	One or more variable symbols separated by commas

Purpose

You use the GBLA, GBLB, and GBLC instructions to declare the global SETA, SETB, and SETC symbols you need.

Specifications

The format of the GBLA, GBLB, and GBLC instruction statements is given in the figure to the right.

These instructions can be used anywhere in the body of a macro definition or in the open code portion of a source module.

DOS If specified inside a macro definition, the GBLA, GBLB, and GBLC instructions must appear immediately following the macro prototype statement. If specified outside a macro definition, the global declarations must appear first in open code; that is, they must follow any source macro definitions specified and precede the beginning of the first control section.

Any variable symbols declared in the operand field have a global scope. They can be used as SET symbols anywhere after the pertinent (1) 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 (2) any macro definition and in open (3) code.

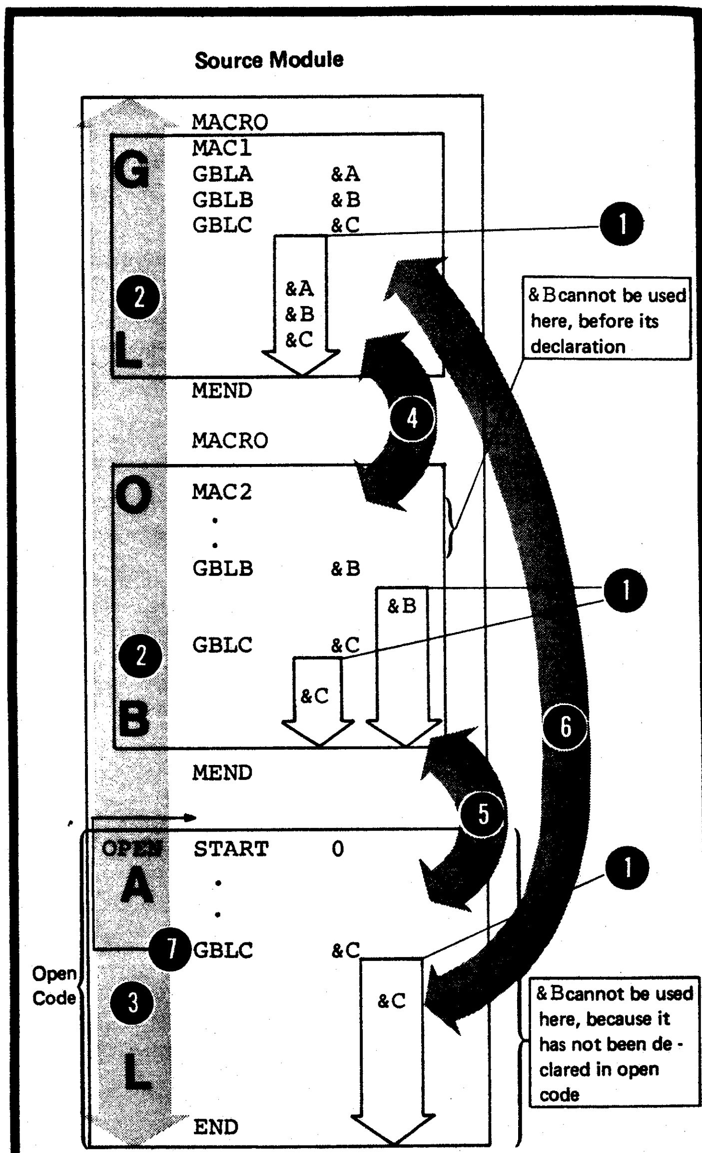
NOTE: Values can be passed between:

(4) • The macro definitions, MAC1, and MAC2, only by using the variable symbols &B and &C.

(5) • The macro definition, MAC2, and open code, only by using the variable symbol &C.

(6) • The macro definition, MAC1, and open code, only by using the variable symbol &C.

DOS NOTE: The "GBLC &C" instruction (7) must precede the START instruction.

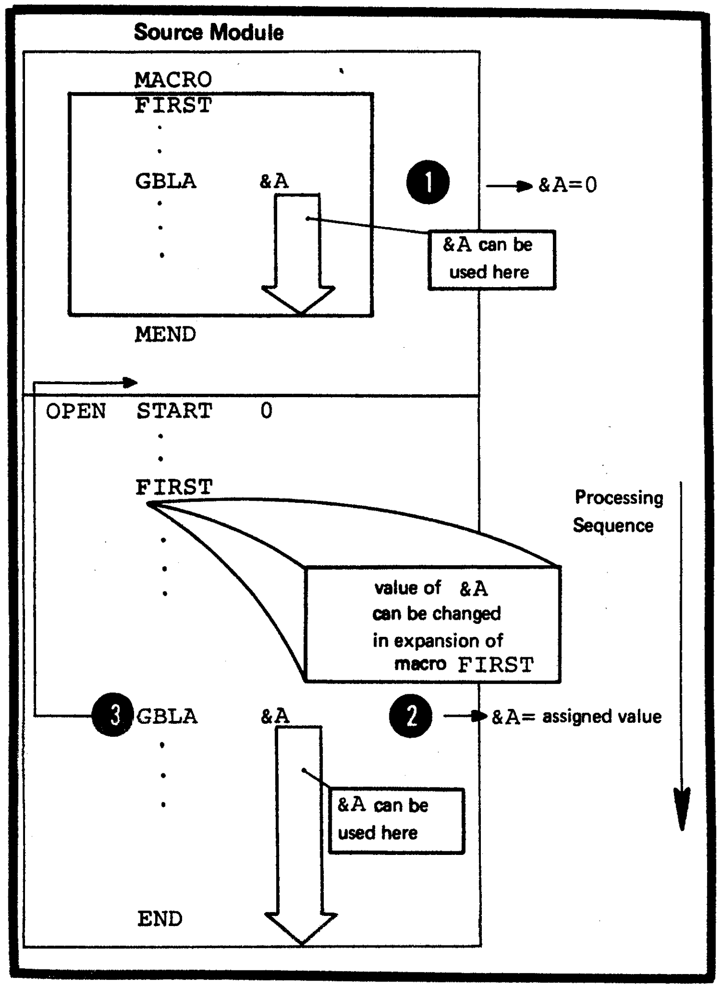


The assembler assigns initial values to these SET symbols as shown in the figure to the right.

Instruction	Initial Value assigned to SET variable symbols in operand field
GBLA	0
GBLB	0
GBLC	Null character string

The assembler assigns this initial value to the SET symbol only when (1) it processes the first GBLA, GBLB, or GBLC instruction in which the symbol appears. (2) Subsequent GBLA, GBLB, or GBLC instructions do not reassign an initial value to the SET symbol.

DOS NOTE: (3) The "GBLA &A" instruction must precede the START instruction.



GLOBAL VARIABLE SYMBOLS MUST NOT BE MULTIPLY DEFINED: A global SET variable symbol declared by the GBLA, GBLB, or GBLC instruction must not be identical to any other variable symbol used in open code or within the same macro definition. The following rules apply to a global SET variable symbol:

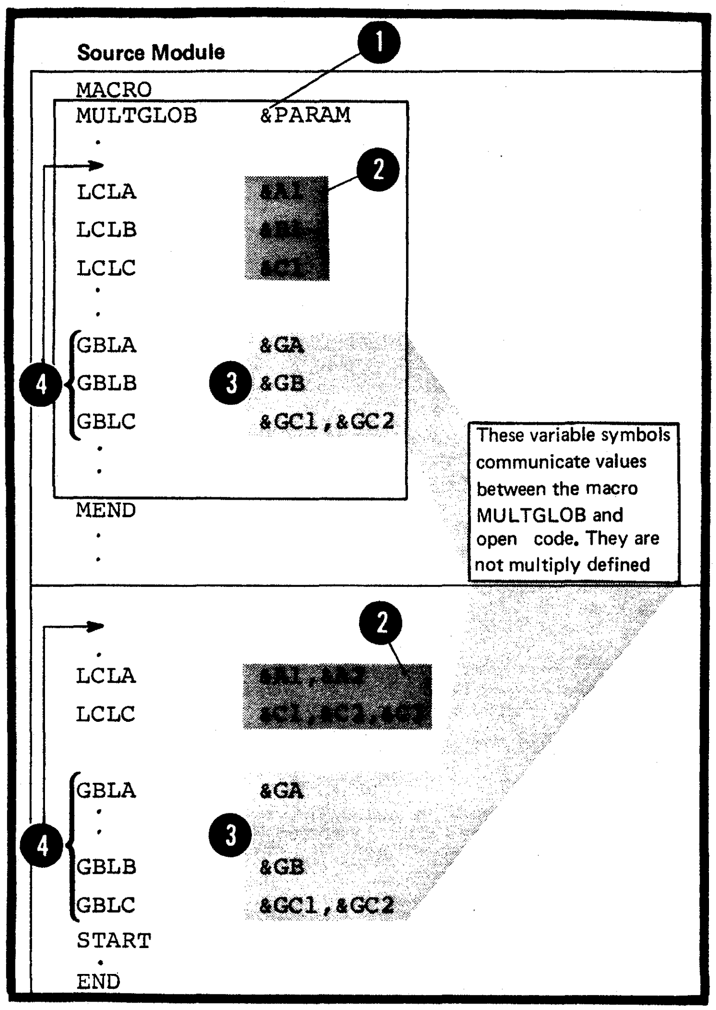
1. Within a macro definition, it must not be the same as any symbolic
(1) parameter declared in the prototype statement.

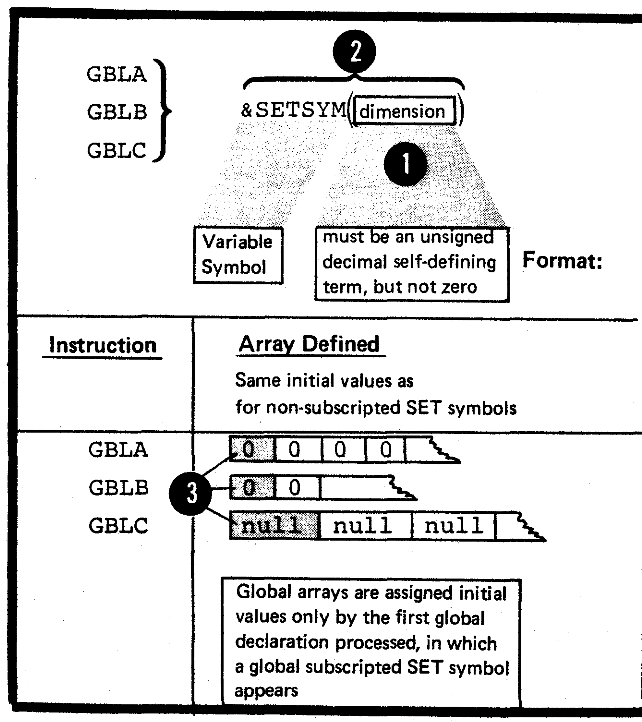
2. It must not be the same as any
(2) local variable symbol (see L2A) declared within the same local scope.

3. The same variable symbol must not be declared or used as two
(3) different types of global SET symbol, for example, as a SETA or SETB symbol.

NOTE 1: A global SET symbol should not begin with the four characters &SYS, which are reserved for system variable symbols (see J7).

DOS **NOTE 2: The global declarations**
(4) **must precede the local declarations.**





SUBSCRIPTED GLOBAL SET SYMBOLS: A global subscripted SET symbol is declared by the GBLA, GBLB, or GBLC instruction. This declaration must be specified as shown in the figure to the right.

(1) The maximum dimension allowed is 32,767.

DOS **The maximum dimension allowed is 4095.**

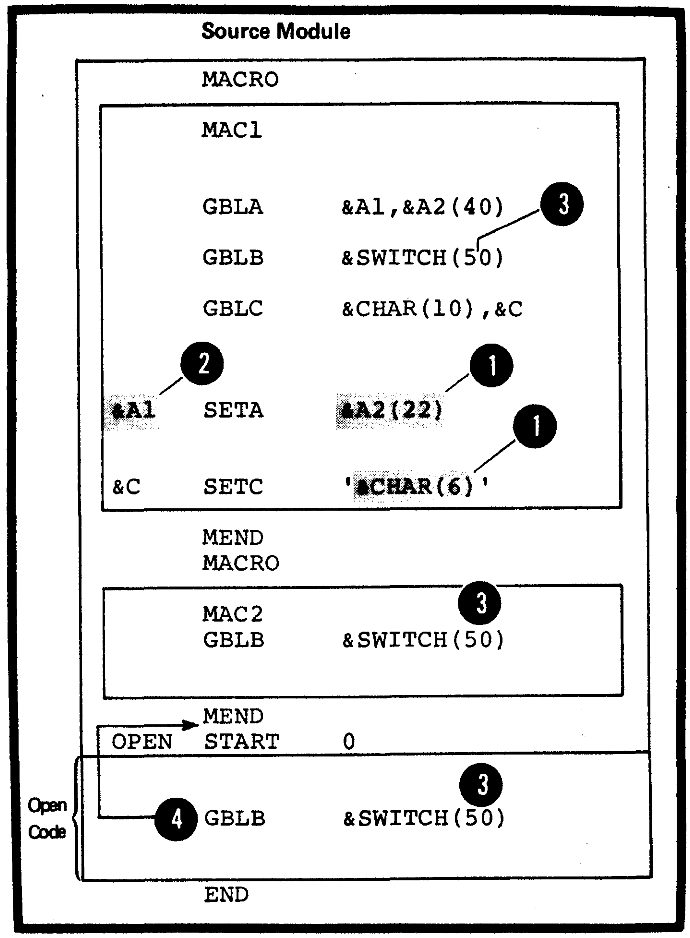
The dimension indicates the number of SET variables associated with
(2) the subscripted SET symbol. The assembler assigns an initial value
(3) to every variable in the array thus declared.

NOTES:

(1) 1. A subscripted global SET symbol can be used only if the declaration has a subscript, which represents (2) a dimension; a nonsubscripted global SET symbol can be used only if the declaration had no subscript.

2. Wherever a particular global SET symbol is declared with a dimension as a subscript, the (3) dimension must be the same in each declaration.

DOS NOTE: The "GBLB &SWITCH(50)" instruction must precede the START (4) instruction.



L3 -- Assigning Values to Set Symbols

L3A -- THE SETA INSTRUCTION

Purpose

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.

Specifications

The format of the SETA instruction statement is given in the figure to the right.

SETA

Name	Operation	Operand
A variable symbol ❶	SETA	An arithmetic expression ❷ Allowable range of values -2^{31} through $2^{31}-1$

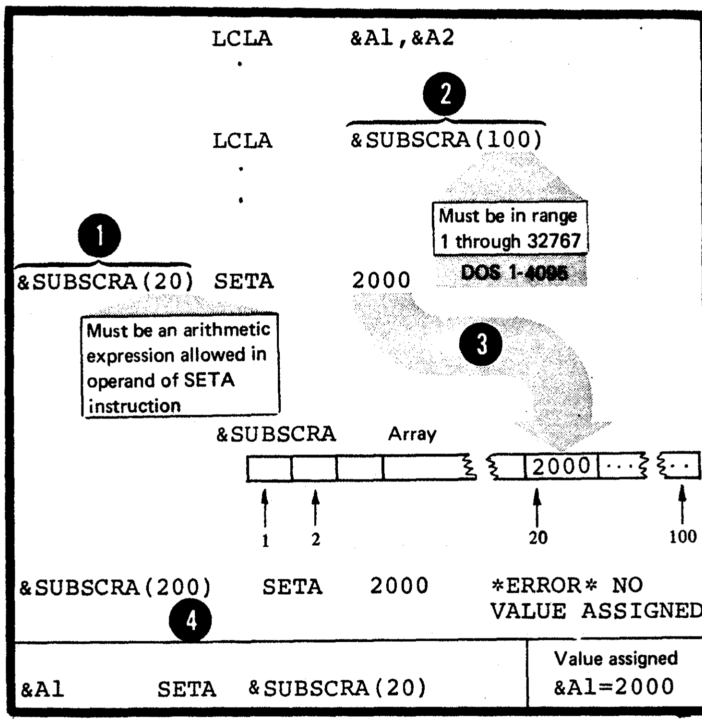
❶ The variable symbol in the name field must have been previously declared as a SETA symbol in a GBLA or LCLA instruction.

OS only: **The variable symbol is assigned a type attribute value of N.**

❷ The assembler evaluates the arithmetic expression in the operand field as a signed 32-bit arithmetic value and assigns this value to the SETA symbol in the name field. An arithmetic expression is described in L4A.

❶ SUBSCRIPTED SETA SYMBOLS: The SETA symbol in the name field can be subscripted, but only if the same SETA symbol has been ❷ previously declared in a GBLA or LCLA instruction with an allowable dimension.

❸ 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 ❹ value, or exceed the dimension actually specified in the declaration.



L3B -- THE SETC INSTRUCTION

Purpose

The SETC instruction allows you to assign a character string 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 (see L5).

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.

Specifications

The format of the SETC instruction statement is given in the figure to the right.

The variable symbol in the name field must have been previously declared as a SETC symbol in a GBLC or LCLC instruction.

OS only: The variable symbol is assigned a type attribute value of U.

The four options that can be specified in the operand field are:

1 1. A type attribute reference

2 2. A character expression (see L4B)

3 3. A substring notation (see L5)

4 4. A concatenation of substring notations, or character expressions, or both.

5 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.

SETC

Format:

Name	Operation	Operand
A variable symbols	SETC	One of four options, exemplified below

5 Value	Examples:		
–	&C1	SETC	T'&DATA or T'SYMBOL (1) — Must appear alone and must not be enclosed in apostrophes
ABC	&C2	SETC	'ABC' (2) — Up to 255 characters enclosed in apostrophes
ABC	&C3	SETC	'ABCDE'(1,3) (3) — Up to 255 characters enclosed in apostrophes
ABCDEF	&C4	SETC	(4) 'ABC'.'DEF'
ABCDEF			or 'ABC'.'ABCDEF'(4,3)

NOTE: When a SETA or SETB symbol is specified in a character
1 expression, the unsigned decimal value of the symbol (with leading
zeros removed) is the character
2 value given to the symbol.

Examples:	Value of &A1	Character Value Assigned to SETC symbols
&C1 SETC '&A1'	200	1 200
&C2 SETC '&A1'	00200	200
&C3 SETC '&A1'	-200	200
&C4 SETC '-200'		-200
&C5 SETC '&A1'	0 (Not considered as leading zero)	0
&C6 SETC '00200' (Part of string represented)		00200
&C7 SETC '&A1+1'	30	30+1
&C8 SETC '1-&A1'	-30	1-30

1 A duplication factor can precede any of the first three options,
2 or any of the parts (character expression or substring notation)
3 that make up the fourth option of the SETC instruction operand. The duplication factor can be any
4 arithmetic expression allowed in the operand of a SETA instruction.

NOTE: The assembler evaluates the character string represented (in particular the substring) before
5 applying the duplication factor.
6 The resulting character string is then assigned to the SETC symbol in the name field.

Format:

(duplication factor) 1 4 — Arithmetic Expression must have a value in the range 1 through 32767

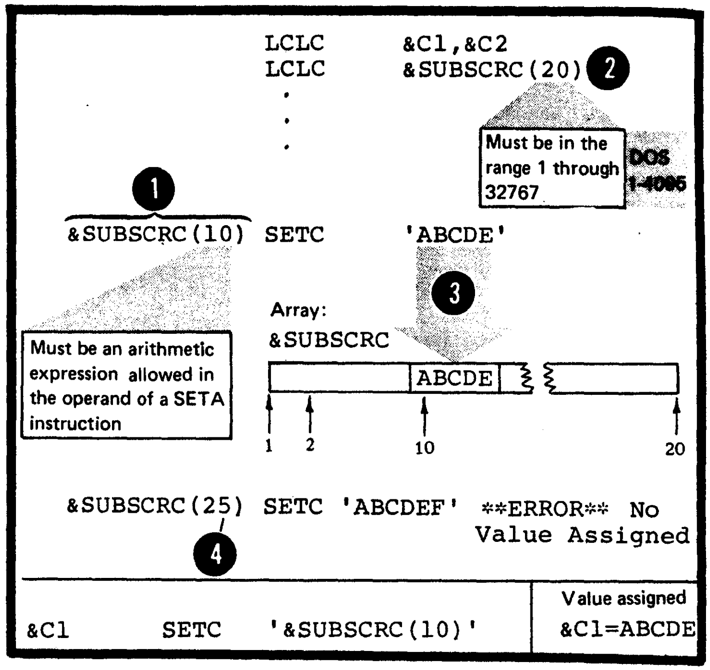
Value Assigned to SETC symbol — Must be in the range 1 through 255

Examples:

Examples		Value Assigned to SETC symbol
&C1 SETC (3)T'&FULLWRD	2	FFF
&C2 SETC (3)'ABC'		ABCABCABC
&C3 SETC (3)'ABCDE'(1,3)		6 [redacted]ABCABC 5
&C4A SETC (3)'ABC'.'DEF' 3		ABCABCABCDEF
&C4B SETC 'ABC'.(3)'ABCDEF'(4,3)		ABC[redacted]DEFDEF 5

❶ SUBSCRIPTED SETC SYMBOLS: The SETC symbol in the name field can be subscripted, but only if the same ❷ SETC symbol has been previously declared in a GBLC or LCLC instruction with an allowable dimension.

The assembler assigns the character value represented in the operand ❸ 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 ❹ value, or exceed the dimension actually specified in the declaration.



L3C -- THE SETB INSTRUCTION

Purpose

The SETB instruction allows you 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 expression (see L4C) in the operand field, the assembler evaluates this expression to determine whether it is true or false and then assigns the values 1 or 0 respectively to the SETB symbol. You can use this computed value in condition tests or for substitution.

Specifications

The format of the SETB instruction statement is given in the figure to the right.

The variable symbol in the name field must have been previously declared as a SETB symbol in a GBLB or LCLB instruction.

OS only: The variable symbol is assigned a type attribute value of N.

The three options that can be specified in the operand field are:

(1) 1. A binary value (0 or 1)

(2) 2. A binary value enclosed in parentheses

OS only: 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.

(3) 3. A logical expression enclosed in parentheses (see L4C).

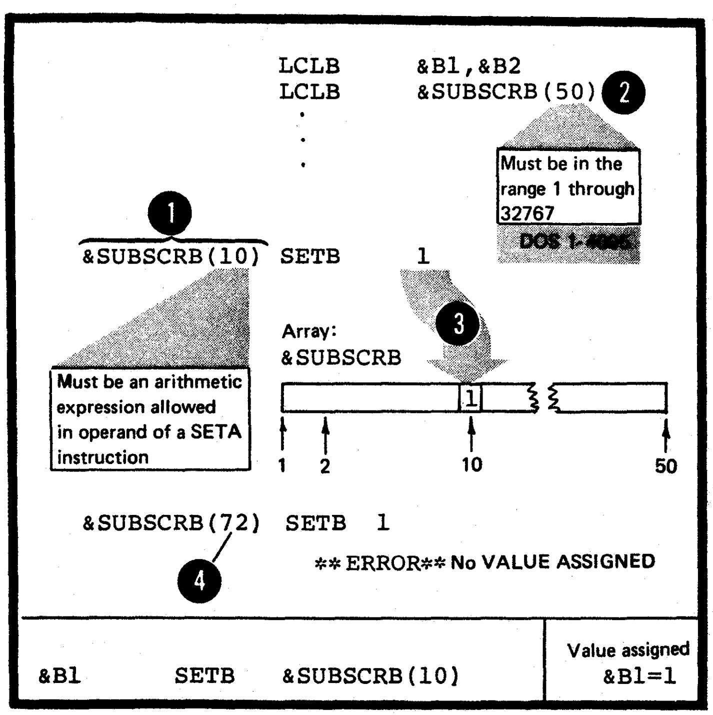
The assembler evaluates the logical expression, if specified, to determine if it is true or false. If it is true, it is given a value of 1; if it is false, a value of 0. (4) The assembler assigns the explicitly specified binary value (0 or 1) or the computed logical value (0 or 1) to the SETB symbol in the name field.

(1) SUBSCRIPTED SETB SYMBOLS: The SETB symbol in the name field can be subscripted, but only if the same SETB symbol has been previously (2) 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 (3) 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, (4) or exceed the dimension actually specified in the declaration.

SETB

Format: Name	Operation	Operand	
A variable symbol	SETB	One of three options, exemplified below	
Examples:			(4) **Values Assigned**
&B1	SETB	0 (1)	0
&B2	SETB	(1) (2)	1
&B3A	SETB	(2 GT 3) false [Greater than] (3)	0
&B3B	SETB	(2 LT 3) true [Less than]	1



L4 – Using Expressions

There are three types of expressions that you can use only in conditional assembly instructions: arithmetic, character, and logical. The assembler evaluates these conditional assembly expressions at pre-assembly time.

Do not confuse the conditional assembly expressions with the absolute or relocatable expressions used in other assembler language instructions and described in C6. The assembler evaluates absolute and relocatable expressions at assembly time.

L4A -- ARITHMETIC (SETA) EXPRESSIONS

Purpose

You can use an arithmetic expression for assigning an arithmetic value to a SETA symbol, or for computing a value used during conditional assembly processing.

An arithmetic expression can contain one or more SET symbols, which allows you to use arithmetic expressions wherever you wish to specify varying values, for example as:

1. Subscripts for SET symbols, symbolic parameters, and &SYSLIST, and in substring notation.

OS only 2. Duplication factors in the operand of the SETC instruction.

You can then control loops, vary the results of computations, and produce different values for substitution into the same model statement.

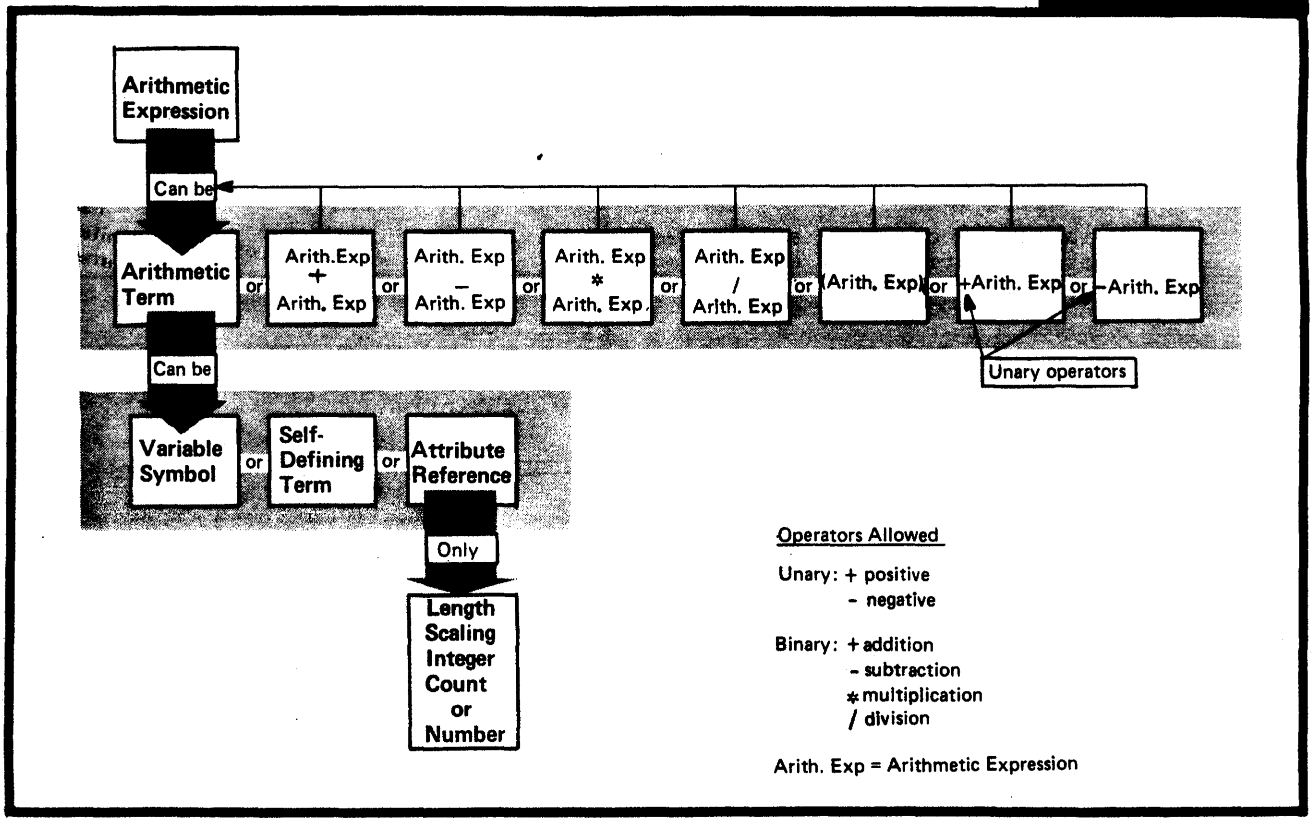
Specifications

Arithmetic expressions can be used as shown in the figure to the right.

NOTE: When an arithmetic expression is used in the operand field of
1 a SETC instruction, the assembler assigns the character value representing the arithmetic expression to the SETC symbol, after
2 substituting values into any variable symbols. It does not evaluate the arithmetic expression.

Can be Used In	Used As	Example
SETA instruction	operand	&A1 SETA &A1+2
AIF instruction 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	sublist (A,B,C,D) when &A=1 &PARAM(&A+1)=B
&SYSLIST	subscript	&SYSLIST(&M+1,&N-2) &SYSLIST(N'&SYSLIST)
SETC instruction	character string in operand	1 &C SETC '5-10*&A' if &A=10 → 2 then &C=5-10*10

The figure below defines an arithmetic expression (self-defining terms are described in C4E).



The variable symbols that are allowed as terms in an arithmetic expression are given in the figure to the right.

Variable Symbol	Restrictions	Example	Value
SETA	none	—	—
SETB	none	—	—
SETC &SYSPARM DOS	value must be an unsigned decimal self-defining term in the range 0 through 2,147,483,647 0-99,999,999	&C &SYSPARM	123 2000
Symbolic Parameters	value must be a self-defining term	&PARAM &SUBLIST(3)	X'A1' C'Z'
&SYSLIST(n) &SYSLIST(n,m)	corresponding operand or sublist entry must be a self-defining term	&SYSLIST(3) &SYSLIST(3,2)	24 B'101'
&SYSNDX	none	—	—

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.

❶ 2. An arithmetic expression can have one or more unary operators preceding any term in the expression or at the beginning of the expression.

❷ 3. An arithmetic expression must not begin with a binary operator, ❸ and it must not contain two binary operators in succession.

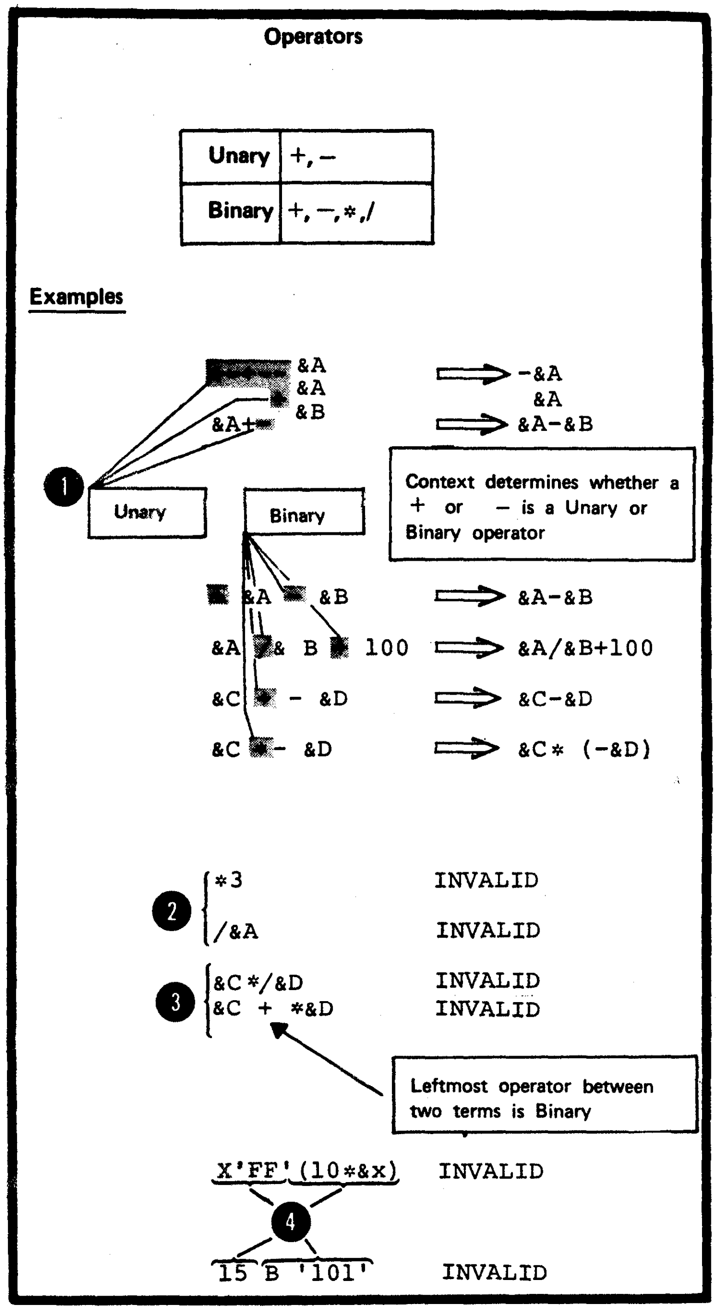
❹ 4. An arithmetic expression must not contain two terms in succession.

5. An arithmetic expression must not contain blanks between an operator and a term nor between two successive operators.

6. An arithmetic expression can contain up to 24 unary and binary operators and up to 11 levels of parentheses.

DOS An arithmetic expression can contain up to 16 unary and binary operators and up to 5 levels of parentheses.

Note that 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 pre-assembly time as follows:

1. It evaluates each arithmetic term.

2. It performs arithmetic operations from left to right. However:

1 a. It performs unary operations before binary operations, and

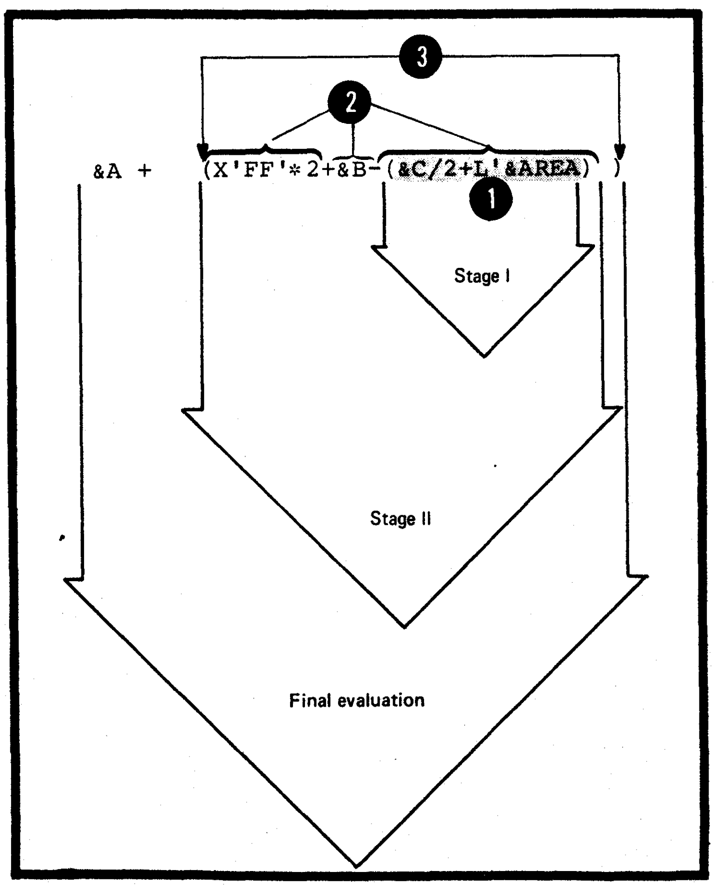
2 b. It performs the binary operations of multiplication and division before the binary
3 operations of addition and subtraction.

4 3. In division, it gives an integer result; any fractional portion is dropped. Division by zero gives a 0 result.

Examples of Arithmetic Expressions		Value of Arithmetic Expression
&A*--X'A' ⇒	&A=5 5*+10 (1)	+50
&A+10/&B ⇒	&A=10,&B=2 10+(10/2) (3) (2) ⇒	15
(&A+10)/&B ⇒	20/2 ⇒	10
&A/2 ⇒	&A=10 10/2	5
(4) &A/2 ⇒	&A=11 11/2	5
&A/2 ⇒	&A=1 1/2	0
10*&A/2 ⇒	&A=1 10/2	5

1 4. In parenthesized arithmetic expressions, the assembler evaluates the innermost expressions first and then considers them as arithmetic
2 terms in the next outer level of 3 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 $+2^{31}-1$.



L4B -- CHARACTER (SETC) EXPRESSIONS

Purpose

The main purpose of a character expression is to assign a character value to a SETC symbol. You can then use the SETC symbol to substitute the character string into a model statement.

You can also use a character expression as a value for comparison in condition tests and logical expressions (see L4C). In addition, a character expression provides the string from which characters can be selected by the substring notation (see L5).

Substitution of one or more character values into a character expression allows you to use the character expression wherever you need to vary values for substitution or to control loops.

Specifications

Character (SETC) expressions can be used only in conditional assembly instructions as shown in the figure to the right.

Char. Exp.

Can be Used in	Used As	Example
SETC instruction	operand	&C SETC 'STRINGO'
AIF instruction or SETB instruction	character string in character relation	AIF ('&C' EQ 'STRING1').B
Substring notation (See L5)	first part of notation	'SELECT'(2,5)=ELECT (character expression: 'SELECT')

A character expression consists of any combination of characters enclosed in apostrophes. Variable symbols are allowed. The assembler substitutes the representation of their values as character strings into the character expression before evaluating the expression.

Up to 255 characters are allowed in a character expression.

NOTE: Attribute references are not allowed in character expressions.

Must not contain more than 255 characters (including blanks)

Variable Symbol	Restrictions	Example	Value Substituted
SETA	sign and leading zeros are suppressed; stand alone zero is used	&A SETA -0201 &C SETC '&A' &ZERO SETA 0 &C SETC '&ZERO'	 201 0
SETB	none	&B SETB 1	1
SETC	none	&C1 SETC 'ABC' &C2 SETC '&C1'	 ABC
Symbolic Parameters	none	&PARAM=(ABC) &C1 SETC '&PARAM'	 (ABC)
System Variable symbols	none	&NUM SETC '&SYSNDX' if &SYSNDX=0201 (leading zeros are not suppressed)	0201

EVALUATION OF CHARACTER EXPRESSIONS: 1 The value of a character expression is the character string within the enclosing apostrophes, after the assembler performs any substitution for variable symbols.

2 Character strings, including variable symbols, can be concatenated to each other within a character 3 expression. The resultant string is the value of the expression used in conditional assembly operations: for example, the value assigned to a SETC symbol.

4 A double apostrophe must be used to generate a single apostrophe as part of the value of a character expression.

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)).

NOTE: To generate a period, two 5 periods must be specified after a variable symbol, or the variable symbol must have a period as part of its value.

Examples (Concatenation operator is a period (.))	Value of Variable Symbols Used	Value of Character Expression
'ABC'		1 ABC
'&PARAM'	SYMBOL	SYMBOL
'A+B-C*D'		A+B-C*D
'&A+10'	10	10+10 (Not 20)
'&A&A'	15	1515
2 '&C.ABC' (mandatory)	DEF	DEFABC 3
'&C.&C' (optional)	DEF	DEFDEF
'&C.+10*&A'	&A=200 &C=AREA	AREA + 10*200
'ABC&C'	&C= .	ABC .
'&C'	null	null character string
'ABC&C.DEF'	&C=null	ABCDEF
4 'L''SYMBOL'		L'SYMBOL
5 '&C..505'	2	2.505
'&C.505'	2.	2.505

Resultant Value must be in the range 0 through 255 characters

DOS 9-127

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.

(1) The resultant value is a character string composed of the concatenated parts.

(2) NOTE: The concatenation character (a period) is needed to separate the apostrophe that ends one character expression from the apostrophe that begins the next.

Concatenated String	Value of Variable Symbol	Resultant Character String Value
'ABC'.'DEF'		ABCDEF (1)
[Mandatory (2)]		
'ABC'.'ABCDEF'(4,3)		ABCDEF
[Substring notation (See L 6)]		
'&C'(4,3).'DEF'	ABCDEF	DEFDEF
[optional]		
'&C'(1,3).'&C'(4,3)	ABCDEF	ABCDEF
'ABC'.'&C'(4,3)'GHI'	ABCDEF	ABCDEFGHI
'ABC'.'&C'.'GHI'	null	ABCGHI
'ABC'.''.'GHI'		ABCGHI
[null character string]		[Value must be in the range 0 through 255 characters]

L4C -- LOGICAL (SETB) EXPRESSIONS

Purpose

You can use a logical (Boolean) expression to assign the binary value 1 or 0 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.

Specifications

Logical (SETB) expressions can be used only in conditional assembly instructions as shown in the figure to the right.

Logical Exp.

Can be used in	Used As	Example
SETB instruction	operand	&B1 SETB (&B2 OR 8 GT 3)
AIF instruction	condition test part of operand	AIF (NOT &B1 OR 8 EQ 3).A

The figure on the opposite page 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)

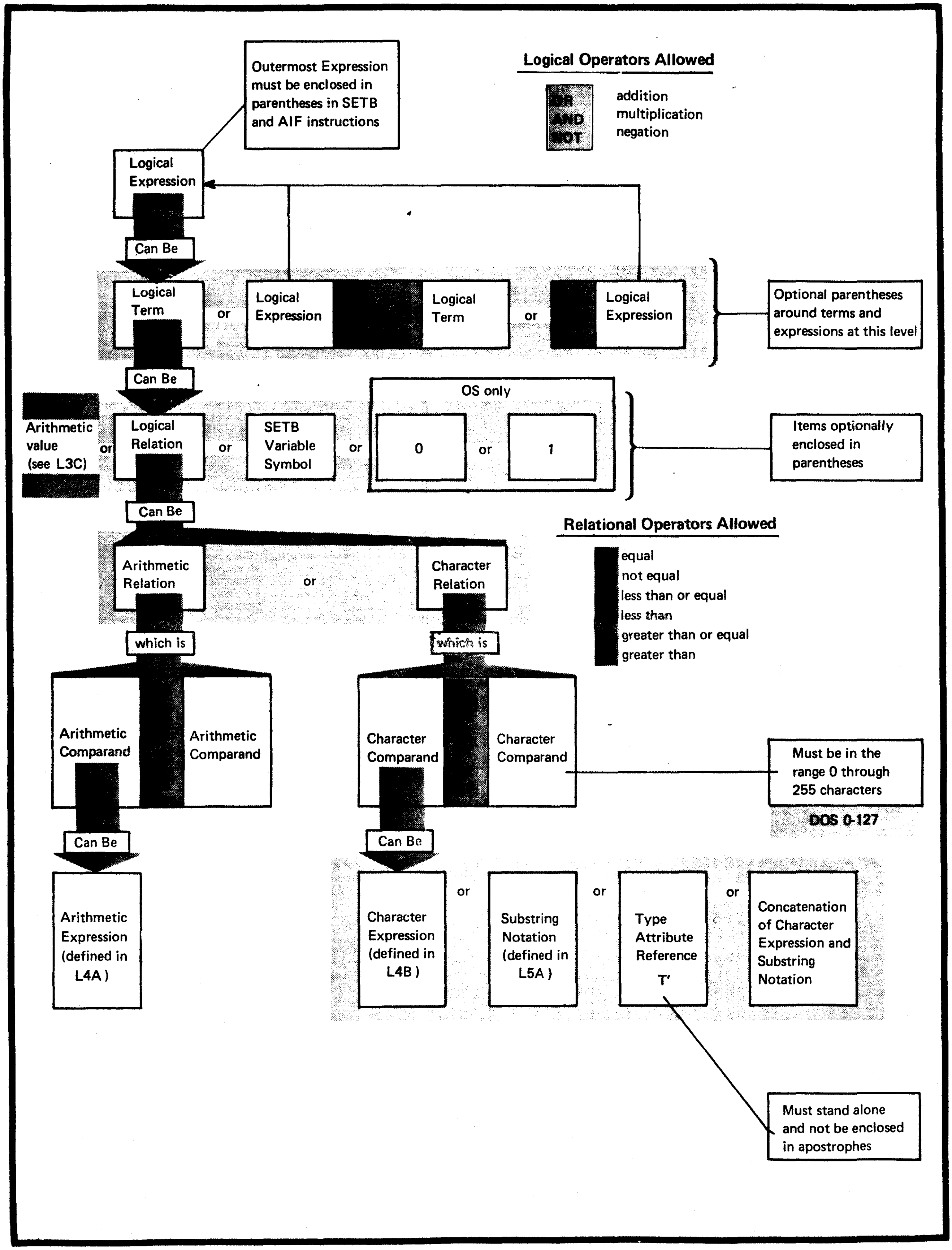
NE (not equal)

LE (less than or equal)

LT (less than)

GE (greater than or equal)

GT (greater than)


Outermost Expression must be enclosed in parentheses in SETB and AIF instructions
Logical Operators Allowed
OR
AND
NOT
addition
multiplication
negation
Logical Expression
Can Be
Logical Term
or
Logical Expression
Logical Term
or
Logical Expression
Optional parentheses around terms and expressions at this level
Can Be
Arithmetic value (see L3C)
or
Logical Relation
or
SETB Variable Symbol
or
OS only
0
or
1
Items optionally enclosed in parentheses
Can Be
Relational Operators Allowed
equal
not equal
less than or equal
less than
greater than or equal
greater than
Arithmetic Relation
or
Character Relation
which is
which is
Arithmetic Comparand
Arithmetic Comparand
Character Comparand
Character Comparand
Must be in the range 0 through 255 characters
DOS 0-127
Can Be
Can Be
Arithmetic Expression (defined in L4A)
Character Expression (defined in L4B)
or
Substring Notation (defined in L5A)
or
Type Attribute Reference T'
or
Concatenation of Character Expression and Substring Notation
Must stand alone and not be enclosed in apostrophes

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.

(1) 2. A logical expression can begin with the logical operator NOT.

(2) 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.

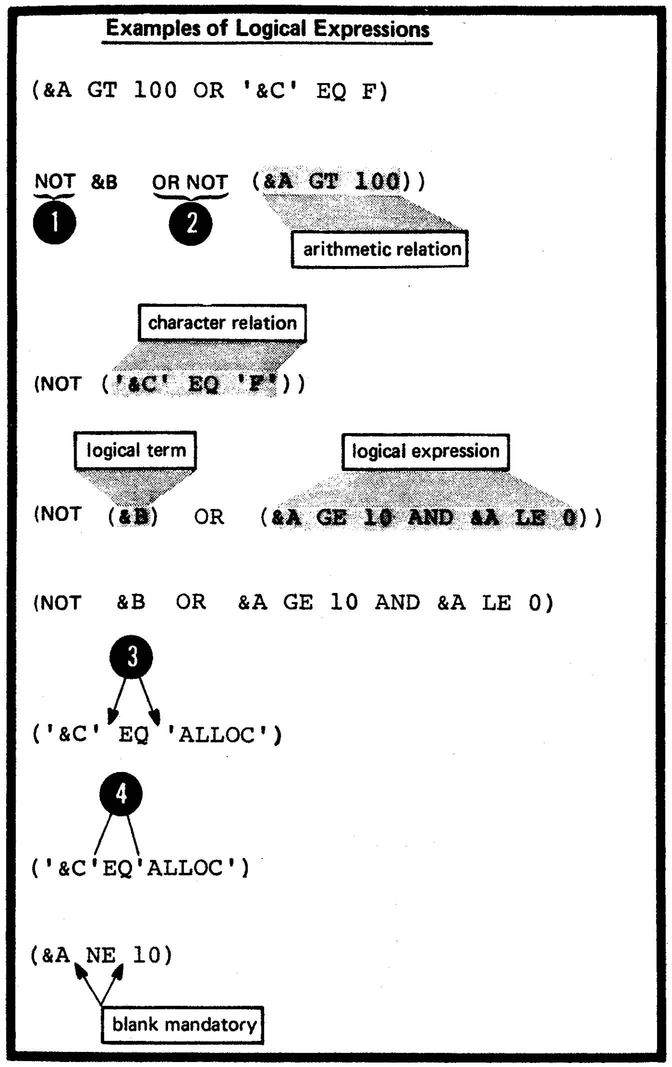
4. Any logical term, relation, or inner logical expression can be optionally enclosed in parentheses.

(3) 5. The relational and logical operators must be immediately preceded and followed by at least one blank or other special character. (4)

6. A logical expression can contain up to 18 logical operators and up to 17 levels of parentheses.

DOS A logical expression can contain up to 18 logical operators and up to 5 levels of parentheses.

Note that the relational and other operators used by the arithmetic and character expressions in relations do not count toward this total.



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:

   (1) a. The arithmetic or character expression specified as values for comparison in these relations, and then

   (2) 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.

   (3) NOTE: 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:

   (4) a. It performs logical NOTs before logical ANDs and ORs, and

   (5) b. It performs logical ANDs before logical ORs.

4. In parenthesized logical expressions, the assembler evaluates (1) the innermost expressions first and then considers them as logical (2) terms in the next outer level of (3) expressions. It continues this process until the outermost expression is evaluated.

Examples of Logical Expressions

```
((&A NE 100) OR T'&AREA EQ '&PARAM'(3,4))
```

(1) values for comparison: &A, 100; T'&AREA, '&PARAM'(3,4) — (2) relations: &A NE 100; T'&AREA EQ '&PARAM'(3,4)

```
('ABC' LT 'ABCD')
```

(3) Always true (given a value of 1)

```
(&B AND NOT (5 GT 3))
```

equivalent

```
(&B AND (NOT (5 GT 3)))
```

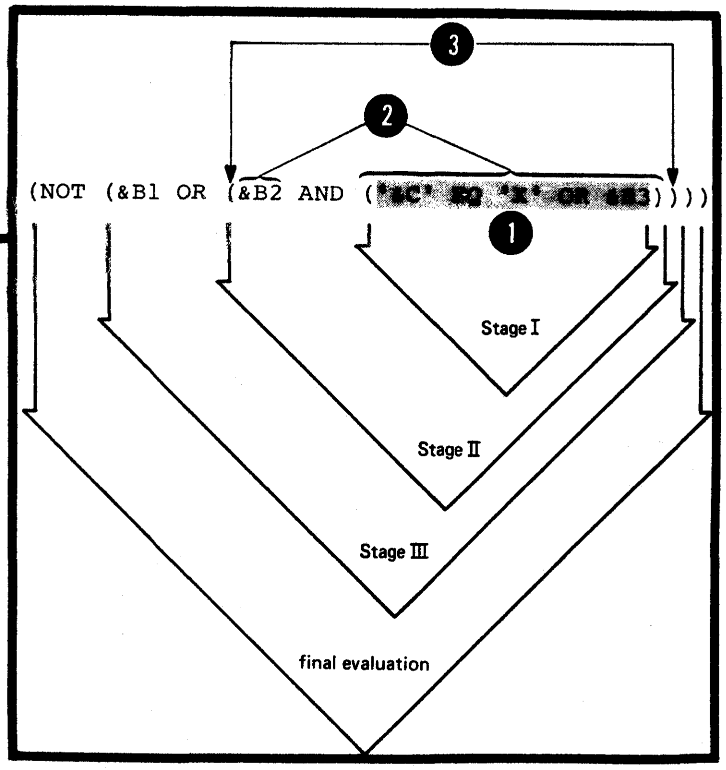
(4)

```
(&B OR &A AND ('&C' EQ 'B'))
```

equivalent

```
(&B OR (&A AND ('&C' EQ 'B')))
```

(5)



L5 -- Selecting Characters from a String

L5A -- SUBSTRING NOTATION

Purpose

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.

Specifications

The substring notation can be used only in conditional assembly instructions as shown in the figure below.

Can be 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
SETB or AIF 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')`	

Substring

The substring notation must be specified as shown in the figure to the right.

1. The character string is a character expression from which the substring is to be extracted. The first
2. subscript indicates the first character that is to be extracted from the character string. The
3. 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
4. length of the resulting substring.

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

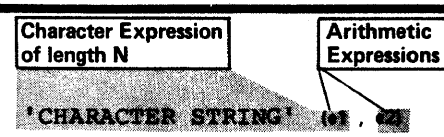
4 Must be in range 0 through 255 characters

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 0-255.

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 below:

1. In the usual case, the assembler generates a correct substring of the specified length.
2. When e1 has a value of zero 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 (OS only) and generates a substring which includes only the characters up to the end of the character expression specified.



Examples: Assume 0<N≤255		Character Value of Substring
1 0<e1≤N, 0<e2≤N, and e1+e2≤N+1		
'ABCDEF'(2,5)	N=6	BCDEF
2 e1≤0		
'ABCDEF'(0,5) (Value of e2 disregarded)	**ERROR**	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≤N, 0<e2≤N, but e1+e2>N+1		
'ABCDEF'(3,5)	N=6 *WARNING*	CDEF
'ABCDEF'(3,4)		CDEF

L6 - Branching

L6A -- The AIF INSTRUCTION

Purpose

The AIF instruction allows you to branch according to the result of a condition test. You can thus alter the sequence in which your assembler language statements are processed.

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.

Specifications

The AIF instruction statement must be specified as shown in the figure to the right.

AIF

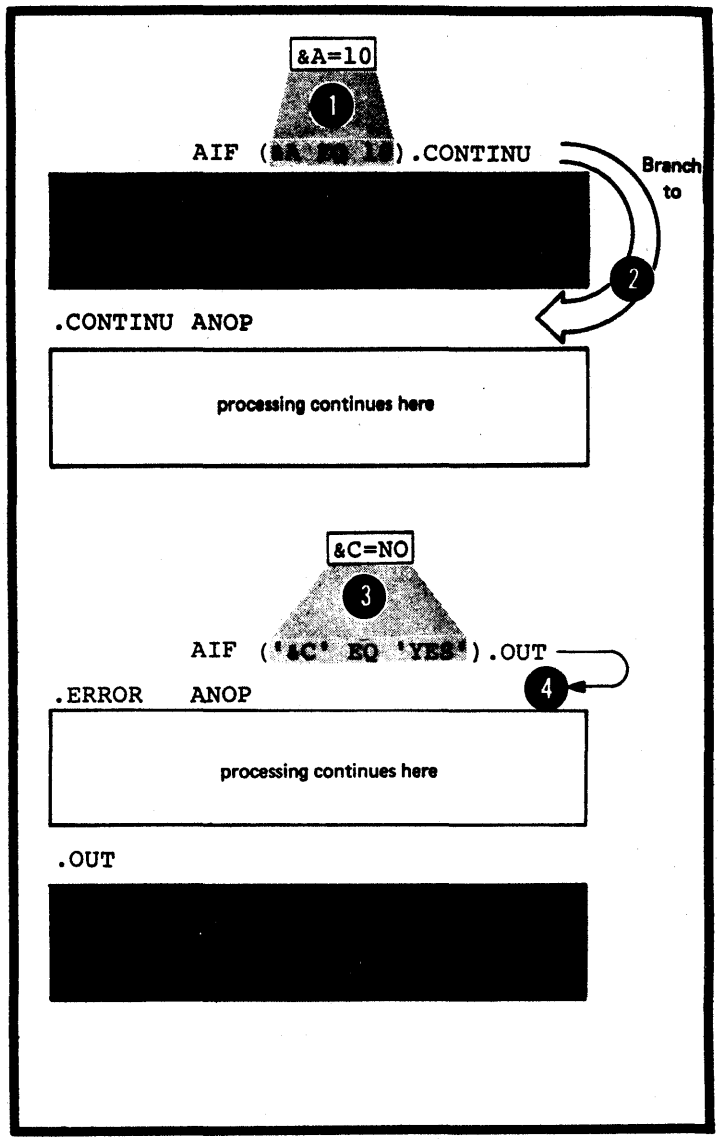
Name	Operation	Operand
A sequence Symbol or Blank	AIF	(logical expression).SEQUENC

(logical expression): Defined in L4C

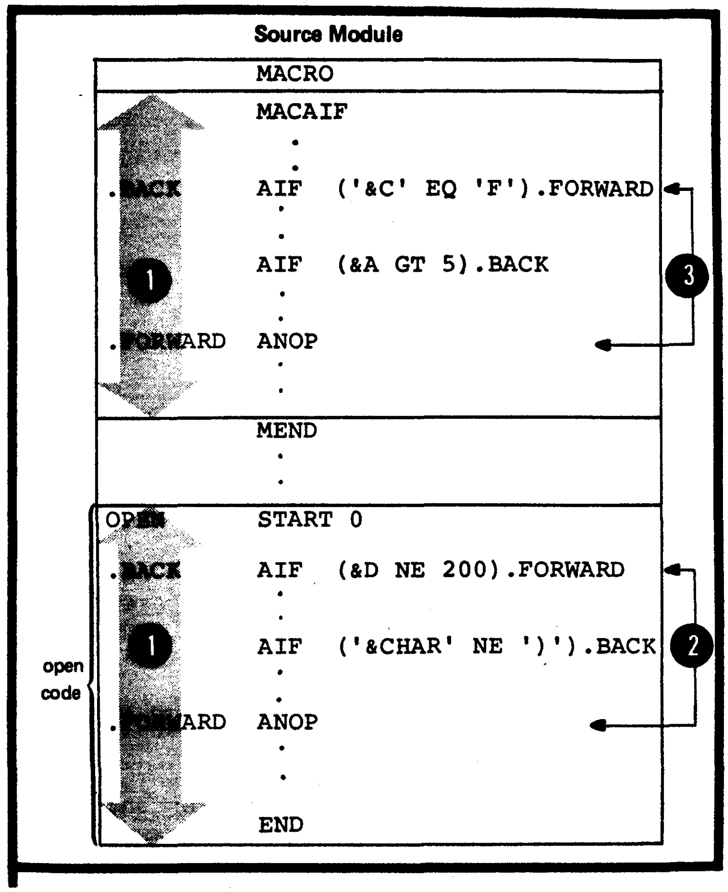
.SEQUENC: Sequence symbol described in L1C

No blanks allowed between right parenthesis and sequence symbol

The assembler evaluates the logical expression in the operand field
(1) at pre-assembly time. If the logical expression is true (logical value=1), the next statement processed by
(2) the assembler is the statement named by the sequence symbol. If it is
(3) false (logical value =0), the next sequential statement is processed. (4)



The sequence symbol in the operand field is a conditional assembly label that represents an address at pre-assembly 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 (1) within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear:

(2) • In open code, if the corresponding AIF instruction does or

(3) • In the same macro definition in which the corresponding AIF instruction appears.

The sequence symbols .BACK and .FORWARD are not multiply defined. No branch can be taken from open code into a macro definition or between macro definitions, regardless of nested calls to other macro definitions.

NOTE: For compatibility, the assemblers described in this manual will process the AIFB instruction (BOS/360) in the same way they process the AIF instruction.

L6B -- THE AGO INSTRUCTION

Purpose

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.

Specifications

The AGO instruction statement must be specified as shown in the figure to the right.

AGO

Name	Operation	Operand
A sequence symbol or blank	AGO	A sequence symbol described in L1C

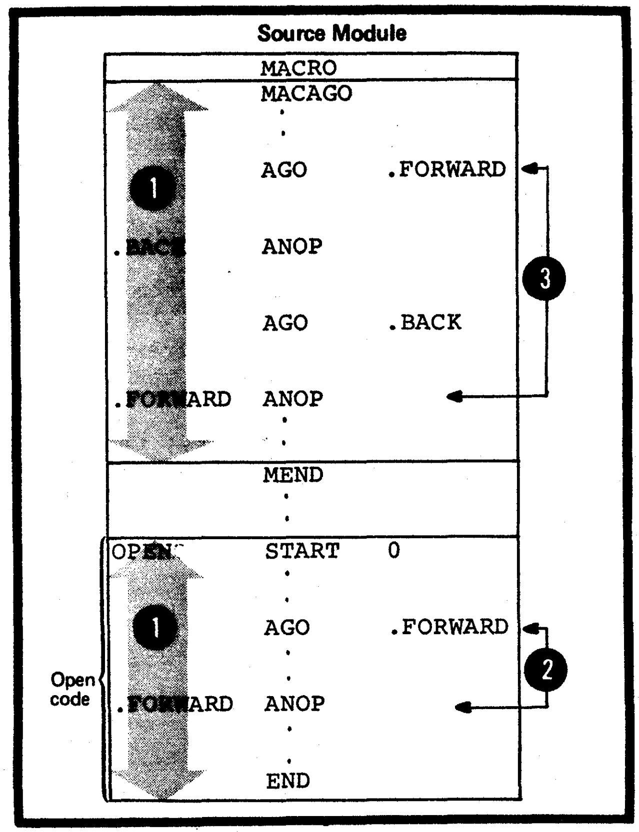
The statement identified by a sequence symbol referred to in the AGO instruction can appear before or after the AGO instruction. However, the statement must appear

❶ within the local scope of the sequence symbol. Thus, the statement identified by the sequence symbol must appear

❷ • In open code, if the corresponding AGO instruction does or

❸ • In the same macro definition in which the corresponding AGO instruction appears.

NOTE: For compatibility, the assemblers described in this manual will process the AGOB instruction (BOS/360) in the same way they process the AGO instruction.



L6C -- THE ACTR INSTRUCTION

Purpose

The ACTR instruction allows you to set a conditional assembly loop counter either within a macro definition or in open code.

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 pre-assembly time.

Specifications

The format of the ACTR instruction statement is given in the figure to the right.

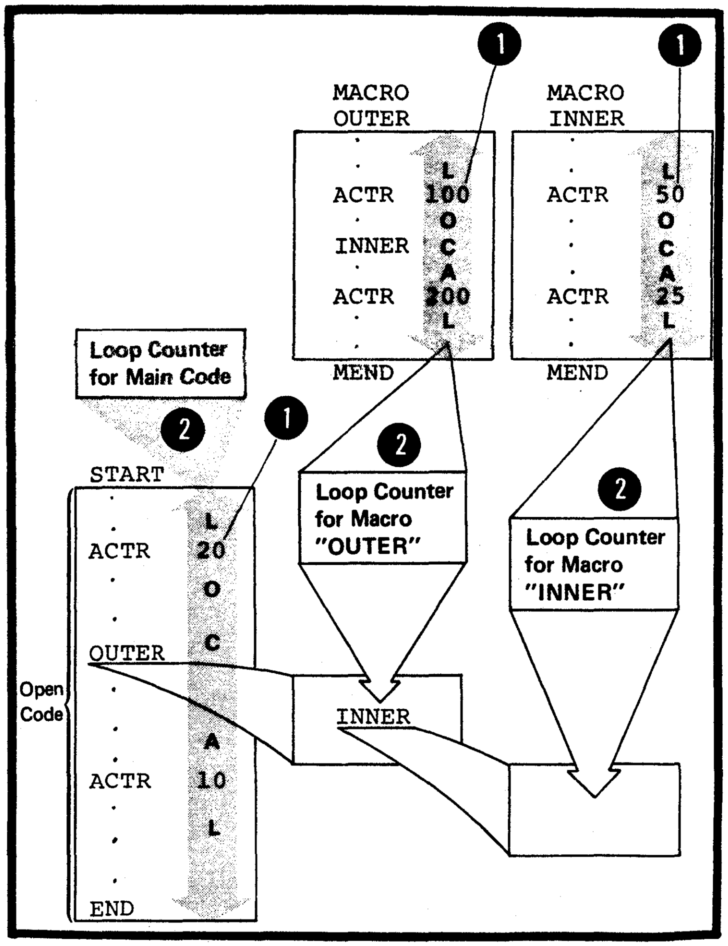
ACTR

Name	Operation	Operand
Sequence symbol or blank	ACTR	Any valid arithmetic (SETA) expression Defined in L4A

The ACTR instruction can appear anywhere in open code or within a macro definition.

A conditional assembly loop counter is set (or reset) to the value of (1) the arithmetic expression in the operand field. The loop counter (2) 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 (also AGOB or AIFB) branch is executed, the assembler checks the loop counter for zero or a negative value.

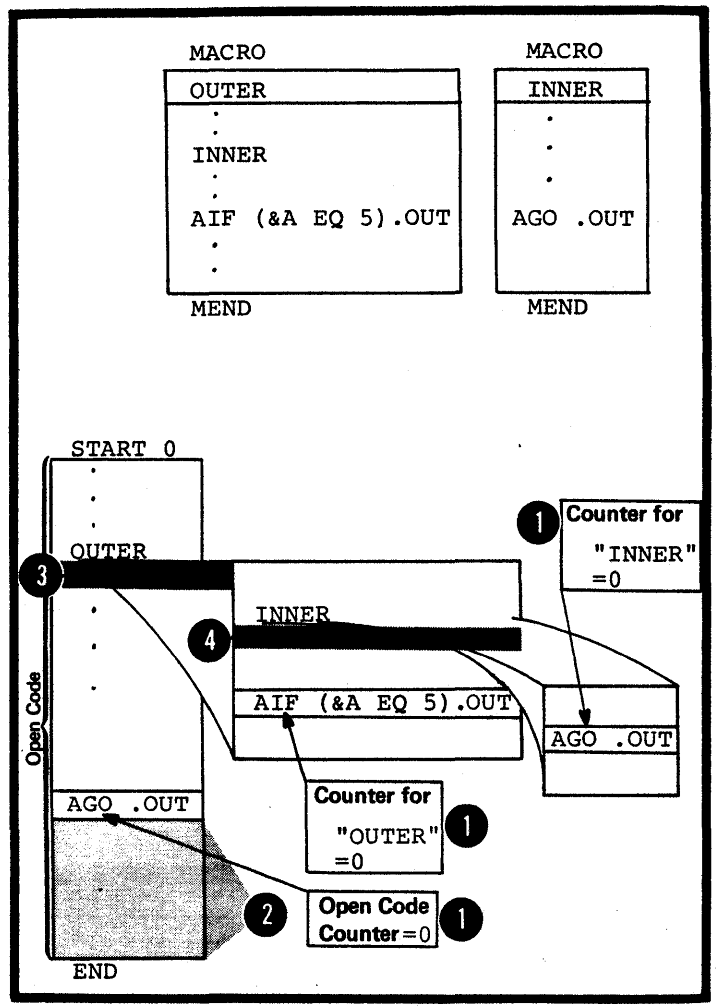
2. If the count is not zero or negative, it is decremented by one.

1 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
2 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
3 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
4 instruction after this inner call, that is, continues processing at the next outer level of nested macros (for levels of nesting see K6A).

NOTE: The assembler halves the ACTR counter value when it encounters serious syntax errors in conditional assembly instructions.



L6D -- THE ANOP INSTRUCTION

Purpose

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.

Specifications

The format of the ANOP instruction statement is given in the figure to the right.

No operation is performed by an ANOP instruction. Instead, if a
1 branch is taken to the ANOP instruction, the assembler processes
2 the next sequential instruction.

ANOP

Name	Operation	Operand
A sequence symbol or blank	ANOP	Not required

Example

```
         AGO    .SEQ   (1)
(2) .SEQ ANOP
    &A   SETA   10
```

L7 -- In Open Code

L7A -- PURPOSE

Conditional assembly instructions in open code allow ycu:

1. To select at pre-assembly time statements or groups of statements from the open code portion of a source nodule according to a pre-determined set of conditions. The assembler further processes the selected statements at assembly time.

2. To pass local variable information from open code through parameters into macro definitions.

3. To control the computation in and generation of macrc definitions using global SET symbols.

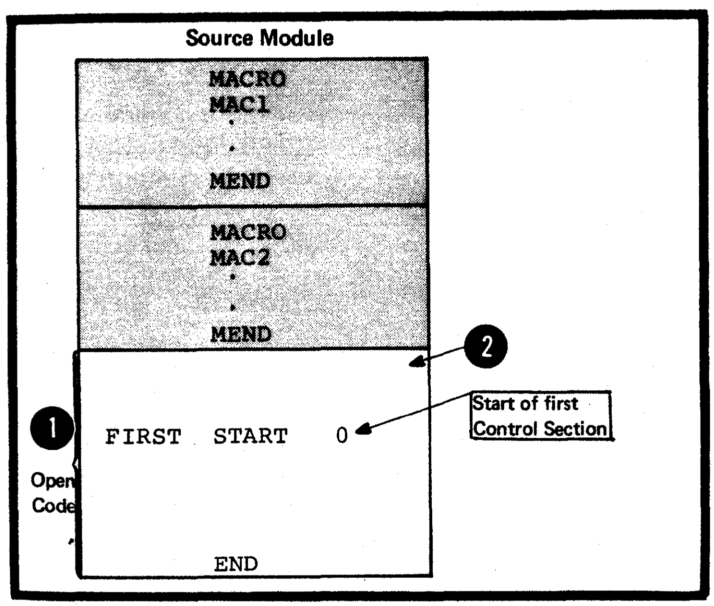
4. To substitute values into the model statements in the open code of a source module and control the sequence of their generation.

L7B -- SPECIFICATIONS

All the conditional assembly elements and instructions can be specified in open code.

Conditional assembly instructions (1) can appear anywhere in open code, but they must appear after any (2) source macro definitions that are specified.

DOS **The global and local declaration instructions (see L2) must appear first in open code; that is, they must follow any source macro definitions specified and precede the beginning of the first control section.**



The specifications for the conditional assembly language described in L1 through L6 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, as shown in the figure to the right.

Expression	Must not contain
Arithmetic (SETA)	► &SYSLIST ► Symbolic parameters ► Any attribute references to symbolic parameters, or &SYSLIST, &SYSECT, &SYSNDX
Character (SETC)	► &SYSLIST, &SYSECT, &SYSNDX ► Attribute references to &SYSLIST, &SYSECT, &SYSNDX, or to symbolic parameters ► Symbolic parameters
Logical (SETB)	• Arithmetic expressions with the items listed above • Character expressions with the items listed above

L8 -- Listing Options

OS only

Purpose

The listing options allow you to print the conditional assembly statements in the sequence they are processed. You can thus follow the conditional assembly logic in open code or in the code within any macro definition.

Specifications

Conditional assembly statements in the open code of a source module or in a macro definition can be printed in the program listings in the order in which they are processed, including iterations. This must be requested by specifying the desired options in the PARM field of the EXEC statement for the assembler program (job control language), or by specifying the options in fields set up by a program that dynamically invokes the assembler. The options are listed in the figure to the right.

NOTE: For other listing options see J8.

Option	Action
NOALOGIC	No conditional assembly statements in open code are printed
ALOGIC	All conditional assembly statements in open code that are processed are printed, including iterations
NOMLOGIC	No conditional assembly statements inside macro definitions, called from your program, are printed. NOTE: Conditional assembly statements in source macro definitions are always printed along with the rest of the code in a source module (assuming the PRINT option LIST)
MLOGIC	All conditional assembly statements inside macro definitions, that are processed when you call the macro, are printed, including iterations

Appendix I: Character Codes

8-Bit EBCDIC Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
00000000	12,0,9,8,1	0	00	
00000001	12,9,1	1	01	
00000010	12,9,2	2	02	
00000011	12,9,3	3	03	
00000100	12,9,4	4	04	
00000101	12,9,5	5	05	
00000110	12,9,6	6	06	
00000111	12,9,7	7	07	
00001000	12,9,8	8	08	
00001001	12,9,8,1	9	09	
00001010	12,9,8,2	10	0A	
00001011	12,9,8,3	11	0B	
00001100	12,9,8,4	12	0C	
00001101	12,9,8,5	13	0D	
00001110	12,9,8,6	14	0E	
00001111	12,9,8,7	15	0F	
00010000	12,11,9,8,1	16	10	
00010001	11,9,1	17	11	
00010010	11,9,2	18	12	
00010011	11,9,3	19	13	
00010100	11,9,4	20	14	
00010101	11,9,5	21	15	
00010110	11,9,6	22	16	
00010111	11,9,7	23	17	
00011000	11,9,8	24	18	
00011001	11,9,8,1	25	19	
00011010	11,9,8,2	26	1A	
00011011	11,9,8,3	27	1B	
00011100	11,9,8,4	28	1C	
00011101	11,9,8,5	29	1D	
00011110	11,9,8,6	30	1E	
00011111	11,9,8,7	31	1F	
00100000	11,0,9,8,1	32	20	
00100001	0,9,1	33	21	
00100010	0,9,2	34	22	
00100011	0,9,3	35	23	
00100100	0,9,4	36	24	
00100101	0,9,5	37	25	
00100110	0,9,6	38	26	
00100111	0,9,7	39	27	
00101000	0,9,8	40	28	
00101001	0,9,8,1	41	29	
00101010	0,9,8,2	42	2A	
00101011	0,9,8,3	43	2B	
00101100	0,9,8,4	44	2C	
00101101	0,9,8,5	45	2D	
00101110	0,9,8,6	46	2E	
00101111	0,9,8,7	47	2F	
00110000	12,11,0,9,8,1	48	30	
00110001	9,1	49	31	
00110010	9,2	50	32	

8-Bit EBCDIC Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
00110011	9,3	51	33	
00110100	9,4	52	34	
00110101	9,5	53	35	
00110110	9,6	54	36	
00110111	9,7	55	37	
00111000	9,8	56	38	
00111001	9,8,1	57	39	
00111010	9,8,2	58	3A	
00111011	9,8,3	59	3B	
00111100	9,8,4	60	3C	
00111101	9,8,5	61	3D	
00111110	9,8,6	62	3E	
00111111	9,8,7	63	3F	
01000000		64	40	blank
01000001	12,0,9,1	65	41	
01000010	12,0,9,2	66	42	
01000011	12,0,9,3	67	43	
01000100	12,0,9,4	68	44	
01000101	12,0,9,5	69	45	
01000110	12,0,9,6	70	46	
01000111	12,0,9,7	71	47	
01001000	12,0,9,8	72	48	
01001001	12,8,1	73	49	
01001010	12,8,2	74	4A	
01001011	12,8,3	75	4B	. (period)
01001100	12,8,4	76	4C	<
01001101	12,8,5	77	4D	(
01001110	12,8,6	78	4E	+
01001111	12,8,7	79	4F	
01010000	12	80	50	&
01010001	12,11,9,1	81	51	
01010010	12,11,9,2	82	52	
01010011	12,11,9,3	83	53	
01010100	12,11,9,4	84	54	
01010101	12,11,9,5	85	55	
01010110	12,11,9,6	86	56	
01010111	12,11,9,7	87	57	
01011000	12,11,9,8	88	58	
01011001	11,8,1	89	59	
01011010	11,8,2	90	5A	
01011011	11,8,3	91	5B	$
01011100	11,8,4	92	5C	*
01011101	11,8,5	93	5D	)
01011110	11,8,6	94	5E	
01011111	11,8,7	95	5F	
01100000	11	96	60	-
01100001	0,1	97	61	/
01100010	11,0,9,2	98	62	
01100011	11,0,9,3	99	63	
01100100	11,0,9,4	100	64	
01100101	11,0,9,5	101	65	
01100110	11,0,9,6	102	66	
01100111	11,0,9,7	103	67	
01101000	11,0,9,8	104	68	
01101001	0,8,1	105	69	
01101010	12,11	106	6A	
01101011	0,8,3	107	6B	, (comma)

8-Bit EBCDIC Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
01101100	0,8,4	108	6C	%
01101101	0,8,5	109	6D	
01101110	0,8,6	110	6E	
01101111	0,8,7	111	6F	
01110000	12,11,0	112	70	
01110001	12,11,0,9,1	113	71	
01110010	12,11,0,9,2	114	72	
01110011	12,11,0,9,3	115	73	
01110100	12,11,0,9,4	116	74	
01110101	12,11,0,9,5	117	75	
01110110	12,11,0,9,6	118	76	
01110111	12,11,0,9,7	119	77	
01111000	12,11,0,9,8	120	78	
01111001	8,1	121	79	
01111010	8,2	122	7A	
01111011	8,3	123	7B	#
01111100	8,4	124	7C	@
01111101	8,5	125	7D	' (apostrophe)
01111110	8,6	126	7E	=
01111111	8,7	127	7F	
10000000	12,0,8,1	128	80	
10000001	12,0,1	129	81	
10000010	12,0,2	130	82	
10000011	12,0,3	131	83	
10000100	12,0,4	132	84	
10000101	12,0,5	133	85	
10000110	12,0,6	134	86	
10000111	12,0,7	135	87	
10001000	12,0,8	136	88	
10001001	12,0,9	137	89	
10001010	12,0,8,2	138	8A	
10001011	12,0,8,3	139	8B	
10001100	12,0,8,4	140	8C	
10001101	12,0,8,5	141	8D	
10001110	12,0,8,6	142	8E	
10001111	12,0,8,7	143	8F	
10010000	12,11,8,1	144	90	
10010001	12,11,1	145	91	
10010010	12,11,2	146	92	
10010011	12,11,3	147	93	
10010100	12,11,4	148	94	
10010101	12,11,5	149	95	
10010110	12,11,6	150	96	
10010111	12,11,7	151	97	
10011000	12,11,8	152	98	
10011001	12,11,9	153	99	
10011010	12,11,8,2	154	9A	
10011011	12,11,8,3	155	9B	
10011100	12,11,8,4	156	9C	
10011101	12,11,8,5	157	9D	
10011110	12,11,8,6	158	9E	
10011111	12,11,8,7	159	9F	
10100000	11,0,8,1	160	A0	
10100001	11,0,1	161	A1	
10100010	11,0,2	162	A2	
10100011	11,0,3	163	A3	
10100100	11,0,4	164	A4	

8-Bit EBCDIC Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
10100101	11,0,5	165	A5	
10100110	11,0,6	166	A6	
10100111	11,0,7	167	A7	
10101000	11,0,8	168	A8	
10101001	11,0,9	169	A9	
10101010	11,0,8,2	170	AA	
10101011	11,0,8,3	171	AB	
10101100	11,0,8,4	172	AC	
10101101	11,0,8,5	173	AD	
10101110	11,0,8,6	174	AE	
10101111	11,0,8,7	175	AF	
10110000	12,11,0,8,1	176	B0	
10110001	12,11,0,1	177	B1	
10110010	12,11,0,2	178	B2	
10110011	12,11,0,3	179	B3	
10110100	12,11,0,4	180	B4	
10110101	12,11,0,5	181	B5	
10110110	12,11,0,6	182	B6	
10110111	12,11,0,7	183	B7	
10111000	12,11,0,8	184	B8	
10111001	12,11,0,9	185	B9	
10111010	12,11,0,8,2	186	BA	
10111011	12,11,0,8,3	187	BB	
10111100	12,11,0,8,4	188	BC	
10111101	12,11,0,8,5	189	BD	
10111110	12,11,0,8,6	190	BE	
10111111	12,11,0,8,7	191	BF	
11000000	12,0	192	C0	
11000001	12,1	193	C1	A
11000010	12,2	194	C2	B
11000011	12,3	195	C3	C
11000100	12,4	196	C4	D
11000101	12,5	197	C5	E
11000110	12,6	198	C6	F
11000111	12,7	199	C7	G
11001000	12,8	200	C8	H
11001001	12,9	201	C9	I
11001010	12,0,9,8,2	202	CA	
11001011	12,0,9,8,3	203	CB	
11001100	12,0,9,8,4	204	CC	
11001101	12,0,9,8,5	205	CD	
11001110	12,0,9,8,6	206	CE	
11001111	12,0,9,8,7	207	CF	
11010000	11,0	208	D0	
11010001	11,1	209	D1	J
11010010	11,2	210	D2	K
11010011	11,3	211	D3	L
11010100	11,4	212	D4	M
11010101	11,5	213	D5	N
11010110	11,6	214	D6	O
11010111	11,7	215	D7	P
11011000	11,8	216	D8	Q
11011001	11,9	217	D9	R
11011010	12,11,9,8,2	218	DA	
11011011	12,11,9,8,3	219	DB	
11011100	12,11,9,8,4	220	DC	
11011101	12,11,9,8,5	221	DD	

8-Bit EBCDIC Code	Character Set Punch Combination	Decimal	Hexa-Decimal	Printer Graphics
11011110	12,11,9,8,6	222	DE	
11011111	12,11,9,8,7	223	DF	
11100000	0,8,2	224	E0	
11100001	11,0,9,1	225	E1	
11100010	0,2	226	E2	S
11100011	0,3	227	E3	T
11100100	0,4	228	E4	U
11100101	0,5	229	E5	V
11100110	0,6	230	E6	W
11100111	0,7	231	E7	X
11101000	0,8	232	E8	Y
11101001	0,9	233	E9	Z
11101010	11,0,9,8,2	234	EA	
11101011	11,0,9,8,3	235	EB	
11101100	11,0,9,8,4	236	EC	
11101101	11,0,9,8,5	237	ED	
11101110	11,0,9,8,6	238	EE	
11101111	11,0,9,8,7	239	EF	
11110000	0	240	F0	0
11110001	1	241	F1	1
11110010	2	242	F2	2
11110011	3	243	F3	3
11110100	4	244	F4	4
11110101	5	245	F5	5
11110110	6	246	F6	6
11110111	7	247	F7	7
11111000	8	248	F8	8
11111001	9	249	F9	9
11111010	12,11,0,9,8,2	250	FA	
11111011	12,11,0,9,8,3	251	FB	
11111100	12,11,0,9,8,4	252	FC	
11111101	12,11,0,9,8,5	253	FD	
11111110	12,11,0,9,8,6	254	FE	
11111111	12,11,0,9,8,7	255	FF	

Special Graphic Characters

¢	Cent Sign	*	Asterisk	>	Greater-than Sign
.	Period, Decimal Point	)	Right Parenthesis	?	Question Mark
<	Less-than Sign	;	Semicolon	:	Colon
(	Left Parenthesis	¬	Logical NOT	#	Number Sign
+	Plus Sign	-	Minus Sign, Hyphen	@	At Sign
|	Vertical Bar, Logical OR	/	Slash	'	Prime, Apostrophe
&	Ampersand	,	Comma	=	Equal Sign
!	Exclamation Point	%	Percent	"	Quotation Mark
$	Dollar Sign	_	Underscore		

Examples	Type	Bit Pattern Bit Positions 01 23 4567	Hole Pattern Zone Punches	Hole Pattern Digit Punches
PF	Control Character	00 00 0100	12 -9	- 4
%	Special Graphic	01 10 1100	0	- 8 - 4
R	Upper Case	11 01 1001	11	- 9
a	Lower Case	10 00 0001	12 -0	- 1
	Control Character, function not yet assigned	00 11 0000	12 - 11 - 0 -9	- 8 - 1

This page left blank intentionally.

Appendix II: Hexadecimal-Decimal Conversion Table

The table in this appendix provides for direct conversion of decimal and hexadecimal numbers in these ranges:

Hexadecimal	Decimal
000 to FFF	0000 to 4095

Decimal numbers (0000-4095) are given within the 5-part table. The first two characters (high-order) of hexadecimal numbers (000-FFF) are given in the lefthand column of the table; the third character (x) is arranged across the top of each part of the table.

To find the decimal equivalent of the hexadecimal number 0C9, look for 0C in the left colum, and across that row under the column for x = 9. The decimal number is 0201.

To convert from decimal to hexadecimal, look up the decimal number within the table and read the hexadecimal number by a combination of the hex characters in the left column, and the value for x at the top of the column containing the decimal number. For example, the decimal number 123 has the hexadecimal equivalent of 07B; the decimal number 1478 has the hexadecimal equivalent of 5C6.

For numbers outside the range of the table, add the following values to the table

Hexadecimal	Decimal
1000	4096
2000	8192
3000	12288
4000	16384
5000	20480
6000	24576
7000	28672
8000	32768
9000	36864
A000	40960
B000	45056
C000	49152
D000	53248
E000	57344
F000	61440

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00x	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
01x	0016	0017	0018	0019	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	0030	0031
02x	0032	0033	0034	0035	0036	0037	0038	0039	0040	0041	0042	0043	0044	0045	0046	0047
03x	0048	0049	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	0060	0061	0062	0063
04x	0064	0065	0066	0067	0068	0069	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079
05x	0080	0081	0082	0083	0084	0085	0086	0087	0088	0089	0090	0091	0092	0093	0094	0095
06x	0096	0097	0098	0099	0100	0101	0102	0103	0104	0105	0106	0107	0108	0109	0110	0111
07x	0112	0113	0114	0115	0116	0117	0118	0119	0120	0121	0122	0123	0124	0125	0126	0127
08x	0128	0129	0130	0131	0132	0133	0134	0135	0136	0137	0138	0139	0140	0141	0142	0143
09x	0144	0145	0146	0147	0148	0149	0150	0151	0152	0153	0154	0155	0156	0157	0158	0159
0Ax	0160	0161	0162	0163	0164	0165	0166	0167	0168	0169	0170	0171	0172	0173	0174	0175
0Bx	0176	0177	0178	0179	0180	0181	0182	0183	0184	0185	0186	0187	0188	0189	0190	0191
0Cx	0192	0193	0194	0195	0196	0197	0198	0199	0200	0201	0202	0203	0204	0205	0206	0207
0Dx	0208	0209	0210	0211	0212	0213	0214	0215	0216	0217	0218	0219	0220	0221	0222	0223
0Ex	0224	0225	0226	0227	0228	0229	0230	0231	0232	0233	0234	0235	0236	0237	0238	0239
0Fx	0240	0241	0242	0243	0244	0245	0246	0247	0248	0249	0250	0251	0252	0253	0254	0255
10x	0256	0257	0258	0259	0260	0261	0262	0263	0264	0265	0266	0267	0268	0269	0270	0271
11x	0272	0273	0274	0275	0276	0277	0278	0279	0280	0281	0282	0283	0284	0285	0286	0287
12x	0288	0289	0290	0291	0292	0293	0294	0295	0296	0297	0298	0299	0300	0301	0302	0303
13x	0304	0305	0306	0307	0308	0309	0310	0311	0312	0313	0314	0315	0316	0317	0318	0319
14x	0320	0321	0322	0323	0324	0325	0326	0327	0328	0329	0330	0331	0332	0333	0334	0335
15x	0336	0337	0338	0339	0340	0341	0342	0343	0344	0345	0346	0347	0348	0349	0350	0351
16x	0352	0353	0354	0355	0356	0357	0358	0359	0360	0361	0362	0363	0364	0365	0366	0367
17x	0368	0369	0370	0371	0372	0373	0374	0375	0376	0377	0378	0379	0380	0381	0382	0383
18x	0384	0385	0386	0387	0388	0389	0390	0391	0392	0393	0394	0395	0396	0397	0398	0399
19x	0400	0401	0402	0403	0404	0405	0406	0407	0408	0409	0410	0411	0412	0413	0414	0415
1Ax	0416	0417	0418	0419	0420	0421	0422	0423	0424	0425	0426	0427	0428	0429	0430	0431
1Bx	0432	0433	0434	0435	0436	0437	0438	0439	0440	0441	0442	0443	0444	0445	0446	0447
1Cx	0448	0449	0450	0451	0452	0453	0454	0455	0456	0457	0458	0459	0460	0461	0462	0463
1Dx	0464	0465	0466	0467	0468	0469	0470	0471	0472	0473	0474	0475	0476	0477	0478	0479
1Ex	0480	0481	0482	0483	0484	0485	0486	0487	0488	0489	0490	0491	0492	0493	0494	0495
1Fx	0496	0497	0498	0499	0500	0501	0502	0503	0504	0505	0506	0507	0508	0509	0510	0511
20x	0512	0513	0514	0515	0516	0517	0518	0519	0520	0521	0522	0523	0524	0525	0526	0527
21x	0528	0529	0530	0531	0532	0533	0534	0535	0536	0537	0538	0539	0540	0541	0542	0543
22x	0544	0545	0546	0547	0548	0549	0550	0551	0552	0553	0554	0555	0556	0557	0558	0559
23x	0560	0561	0562	0563	0564	0565	0566	0567	0568	0569	0570	0571	0572	0573	0574	0575
24x	0576	0577	0578	0579	0580	0581	0582	0583	0584	0585	0586	0587	0588	0589	0590	0591
25x	0592	0593	0594	0595	0596	0597	0598	0599	0600	0601	0602	0603	0604	0605	0606	0607
26x	0608	0609	0610	0611	0612	0613	0614	0615	0616	0617	0618	0619	0620	0621	0622	0623
27x	0624	0625	0626	0627	0628	0629	0630	0631	0632	0633	0634	0635	0636	0637	0638	0639
28x	0640	0641	0642	0643	0644	0645	0646	0647	0648	0649	0650	0651	0652	0653	0654	0655
29x	0656	0657	0658	0659	0660	0661	0662	0663	0664	0665	0666	0667	0668	0669	0670	0671
2Ax	0672	0673	0674	0675	0676	0677	0678	0679	0680	0681	0682	0683	0684	0685	0686	0687
2Bx	0688	0689	0690	0691	0692	0693	0694	0695	0696	0697	0698	0699	0700	0701	0702	0703
2Cx	0704	0705	0706	0707	0708	0709	0710	0711	0712	0713	0714	0715	0716	0717	0718	0719
2Dx	0720	0721	0722	0723	0724	0725	0726	0727	0728	0729	0730	0731	0732	0733	0734	0735
2Ex	0736	0737	0738	0739	0740	0741	0742	0743	0744	0745	0746	0747	0748	0749	0750	0751
2Fx	0752	0753	0754	0755	0756	0757	0758	0759	0760	0761	0762	0763	0764	0765	0766	0767
30x	0768	0769	0770	0771	0772	0773	0774	0775	0776	0777	0778	0779	0780	0781	0782	0783
31x	0784	0785	0786	0787	0788	0789	0790	0791	0792	0793	0794	0795	0796	0797	0798	0799
32x	0800	0801	0802	0803	0804	0805	0806	0807	0808	0809	0810	0811	0812	0813	0814	0815
33x	0816	0817	0818	0819	0820	0821	0822	0823	0824	0825	0826	0827	0828	0829	0830	0831
34x	0832	0833	0834	0835	0836	0837	0838	0839	0840	0841	0842	0843	0844	0845	0846	0847
35x	0848	0849	0850	0851	0852	0853	0854	0855	0856	0857	0858	0859	0860	0861	0862	0863
36x	0864	0865	0866	0867	0868	0869	0870	0871	0872	0873	0874	0875	0876	0877	0878	0879
37x	0880	0881	0882	0883	0884	0885	0886	0887	0888	0889	0890	0891	0892	0893	0894	0895
38x	0896	0897	0898	0899	0900	0901	0902	0903	0904	0905	0906	0907	0908	0909	0910	0911
39x	0912	0913	0914	0915	0916	0917	0918	0919	0920	0921	0922	0923	0924	0925	0926	0927
3Ax	0928	0929	0930	0931	0932	0933	0934	0935	0936	0937	0938	0939	0940	0941	0942	0943
3Bx	0944	0945	0946	0947	0948	0949	0950	0951	0952	0953	0954	0955	0956	0957	0958	0959
3Cx	0960	0961	0962	0963	0964	0965	0966	0967	0968	0969	0970	0971	0972	0973	0974	0975
3Dx	0976	0977	0978	0979	0980	0981	0982	0983	0984	0985	0986	0987	0988	0989	0990	0991
3Ex	0992	0993	0994	0995	0996	0997	0998	0999	1000	1001	1002	1003	1004	1005	1006	1007
3Fx	1008	1009	1010	1011	1012	1013	1014	1015	1016	1017	1018	1019	1020	1021	1022	1023

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
40x	1024	1025	1026	1027	1028	1029	1030	1031	1032	1033	1034	1035	1036	1037	1038	1039
41x	1040	1041	1042	1043	1044	1045	1046	1047	1048	1049	1050	1051	1052	1053	1054	1055
42x	1056	1057	1058	1059	1060	1061	1062	1063	1064	1065	1066	1067	1068	1069	1070	1071
43x	1072	1073	1074	1075	1076	1077	1078	1079	1080	1081	1082	1083	1084	1085	1086	1087
44x	1088	1089	1090	1091	1092	1093	1094	1095	1096	1097	1098	1099	1100	1101	1102	1103
45x	1104	1105	1106	1107	1108	1109	1110	1111	1112	1113	1114	1115	1116	1117	1118	1119
46x	1120	1121	1122	1123	1124	1125	1126	1127	1128	1129	1130	1131	1132	1133	1134	1135
47x	1136	1137	1138	1139	1140	1141	1142	1143	1144	1145	1146	1147	1148	1149	1150	1151
48x	1152	1153	1154	1155	1156	1157	1158	1159	1160	1161	1162	1163	1164	1165	1166	1167
49x	1168	1169	1170	1171	1172	1173	1174	1175	1176	1177	1178	1179	1180	1181	1182	1183
4Ax	1184	1185	1186	1187	1188	1189	1190	1191	1192	1193	1194	1195	1196	1197	1198	1199
4Bx	1200	1201	1202	1203	1204	1205	1206	1207	1208	1209	1210	1211	1212	1213	1214	1215
4Cx	1216	1217	1218	1219	1220	1221	1222	1223	1224	1225	1226	1227	1228	1229	1230	1231
4Dx	1232	1233	1234	1235	1236	1237	1238	1239	1240	1241	1242	1243	1244	1245	1246	1247
4Ex	1248	1249	1250	1251	1252	1253	1254	1255	1256	1257	1258	1259	1260	1261	1262	1263
4Fx	1264	1265	1266	1267	1268	1269	1270	1271	1272	1273	1274	1275	1276	1277	1278	1279
50x	1280	1281	1282	1283	1284	1285	1286	1287	1288	1289	1290	1291	1292	1293	1294	1295
51x	1296	1297	1298	1299	1300	1301	1302	1303	1304	1305	1306	1307	1308	1309	1310	1311
52x	1312	1313	1314	1315	1316	1317	1318	1319	1320	1321	1322	1323	1324	1325	1326	1327
53x	1328	1329	1330	1331	1332	1333	1334	1335	1336	1337	1338	1339	1340	1341	1342	1343
54x	1344	1345	1346	1347	1348	1349	1350	1351	1352	1353	1354	1355	1356	1357	1358	1359
55x	1360	1361	1362	1363	1364	1365	1366	1367	1368	1369	1370	1371	1372	1373	1374	1375
56x	1376	1377	1378	1379	1380	1381	1382	1383	1384	1385	1386	1387	1388	1389	1390	1391
57x	1392	1393	1394	1395	1396	1397	1398	1399	1400	1401	1402	1403	1404	1405	1406	1407
58x	1408	1409	1410	1411	1412	1413	1414	1415	1416	1417	1418	1419	1420	1421	1422	1423
59x	1424	1425	1426	1427	1428	1429	1430	1431	1432	1433	1434	1435	1436	1437	1438	1439
5Ax	1440	1441	1442	1443	1444	1445	1446	1447	1448	1449	1450	1451	1452	1453	1454	1455
5Bx	1456	1457	1458	1459	1460	1461	1462	1463	1464	1465	1466	1467	1468	1469	1470	1471
5Cx	1472	1473	1474	1475	1476	1477	1478	1479	1480	1481	1482	1483	1484	1485	1486	1487
5Dx	1488	1489	1490	1491	1492	1493	1494	1495	1496	1497	1498	1499	1500	1501	1502	1503
5Ex	1504	1505	1506	1507	1508	1509	1510	1511	1512	1513	1514	1515	1516	1517	1518	1519
5Fx	1520	1521	1522	1523	1524	1525	1526	1527	1528	1529	1530	1531	1532	1533	1534	1535
60x	1536	1537	1538	1539	1540	1541	1542	1543	1544	1545	1546	1547	1548	1549	1550	1551
61x	1552	1553	1554	1555	1556	1557	1558	1559	1560	1561	1562	1563	1564	1565	1566	1567
62x	1568	1569	1570	1571	1572	1573	1574	1575	1576	1577	1578	1579	1580	1581	1582	1583
63x	1584	1585	1586	1587	1588	1589	1590	1591	1592	1593	1594	1595	1596	1597	1598	1599
64x	1600	1601	1602	1603	1604	1605	1606	1607	1608	1609	1610	1611	1612	1613	1614	1615
65x	1616	1617	1618	1619	1620	1621	1622	1623	1624	1625	1626	1627	1628	1629	1630	1631
66x	1632	1633	1634	1635	1636	1637	1638	1639	1640	1641	1642	1643	1644	1645	1646	1647
67x	1648	1649	1650	1651	1652	1653	1654	1655	1656	1657	1658	1659	1660	1661	1662	1663
68x	1664	1665	1666	1667	1668	1669	1670	1671	1672	1673	1674	1675	1676	1677	1678	1679
69x	1680	1681	1682	1683	1684	1685	1686	1687	1688	1689	1690	1691	1692	1693	1694	1695
6Ax	1696	1697	1698	1699	1700	1701	1702	1703	1704	1705	1706	1707	1708	1709	1710	1711
6Bx	1712	1713	1714	1715	1716	1717	1718	1719	1720	1721	1722	1723	1724	1725	1726	1727
6Cx	1728	1729	1730	1731	1732	1733	1734	1735	1736	1737	1738	1739	1740	1741	1742	1743
6Dx	1744	1745	1746	1747	1748	1749	1750	1751	1752	1753	1754	1755	1756	1757	1758	1759
6Ex	1760	1761	1762	1763	1764	1765	1766	1767	1768	1769	1770	1771	1772	1773	1774	1775
6Fx	1776	1777	1778	1779	1780	1781	1782	1783	1784	1785	1786	1787	1788	1789	1790	1791
70x	1792	1793	1794	1795	1796	1797	1798	1799	1800	1801	1802	1803	1804	1805	1806	1807
71x	1808	1809	1810	1811	1812	1813	1814	1815	1816	1817	1818	1819	1820	1821	1822	1823
72x	1824	1825	1826	1827	1828	1829	1830	1831	1832	1833	1834	1835	1836	1837	1838	1839
73x	1840	1841	1842	1843	1844	1845	1846	1847	1848	1849	1850	1851	1852	1853	1854	1855
74x	1856	1857	1858	1859	1860	1861	1862	1863	1864	1865	1866	1867	1868	1869	1870	1871
75x	1872	1873	1874	1875	1876	1877	1878	1879	1880	1881	1882	1883	1884	1885	1886	1887
76x	1888	1889	1890	1891	1892	1893	1894	1895	1896	1897	1898	1899	1900	1901	1902	1903
77x	1904	1905	1906	1907	1908	1909	1910	1911	1912	1913	1914	1915	1916	1917	1918	1919
78x	1920	1921	1922	1923	1924	1925	1926	1927	1928	1929	1930	1931	1932	1933	1934	1935
79x	1936	1937	1938	1939	1940	1941	1942	1943	1944	1945	1946	1947	1948	1949	1950	1951
7Ax	1952	1953	1954	1955	1956	1957	1958	1959	1960	1961	1962	1963	1964	1965	1966	1967
7Bx	1968	1969	1970	1971	1972	1973	1974	1975	1976	1977	1978	1979	1980	1981	1982	1983
7Cx	1984	1985	1986	1987	1988	1989	1990	1991	1992	1993	1994	1995	1996	1997	1998	1999
7Dx	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	2012	2013	2014	2015
7Ex	2016	2017	2018	2019	2020	2021	2022	2023	2024	2025	2026	2027	2028	2029	2030	2031
7Fx	2032	2033	2034	2035	2036	2037	2038	2039	2040	2041	2042	2043	2044	2045	2046	2047

x =	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
80x	2048	2049	2050	2051	2052	2053	2054	2055	2056	2057	2058	2059	2060	2061	2062	2063
81x	2064	2065	2066	2067	2068	2069	2070	2071	2072	2073	2074	2075	2076	2077	2078	2079
82x	2080	2081	2082	2083	2084	2085	2086	2087	2088	2089	2090	2091	2092	2093	2094	2095
83x	2096	2097	2098	2099	2100	2101	2102	2103	2104	2105	2106	2107	2108	2109	2110	2111
84x	2112	2113	2114	2115	2116	2117	2118	2119	2120	2121	2122	2123	2124	2125	2126	2127
85x	2128	2129	2130	2131	2132	2133	2134	2135	2136	2137	2138	2139	2140	2141	2142	2143
86x	2144	2145	2146	2147	2148	2149	2150	2151	2152	2153	2154	2155	2156	2157	2158	2159
87x	2160	2161	2162	2163	2164	2165	2166	2167	2168	2169	2170	2171	2172	2173	2174	2175
88x	2176	2177	2178	2179	2180	2181	2182	2183	2184	2185	2186	2187	2188	2189	2190	2191
89x	2192	2193	2194	2195	2196	2197	2198	2199	2200	2201	2202	2203	2204	2205	2206	2207
8Ax	2208	2209	2210	2211	2212	2213	2214	2215	2216	2217	2218	2219	2220	2221	2222	2223
8Bx	2224	2225	2226	2227	2228	2229	2230	2231	2232	2233	2234	2235	2236	2237	2238	2239
8Cx	2240	2241	2242	2243	2244	2245	2246	2247	2248	2249	2250	2251	2252	2253	2254	2255
8Dx	2256	2257	2258	2259	2260	2261	2262	2263	2264	2265	2266	2267	2268	2269	2270	2271
8Ex	2272	2273	2274	2275	2276	2277	2278	2279	2280	2281	2282	2283	2284	2285	2286	2287
8Fx	2288	2289	2290	2291	2292	2293	2294	2295	2296	2297	2298	2299	2300	2301	2302	2303
90x	2304	2305	2306	2307	2308	2309	2310	2311	2312	2313	2314	2315	2316	2317	2318	2319
91x	2320	2321	2322	2323	2324	2325	2326	2327	2328	2329	2330	2331	2332	2333	2334	2335
92x	2336	2337	2338	2339	2340	2341	2342	2343	2344	2345	2346	2347	2348	2349	2350	2351
93x	2352	2353	2354	2355	2356	2357	2358	2359	2360	2361	2362	2363	2364	2365	2366	2367
94x	2368	2369	2370	2371	2372	2373	2374	2375	2376	2377	2378	2379	2380	2381	2382	2383
95x	2384	2385	2386	2387	2388	2389	2390	2391	2392	2393	2394	2395	2396	2397	2398	2399
96x	2400	2401	2402	2403	2404	2405	2406	2407	2408	2409	2410	2411	2412	2413	2414	2415
97x	2416	2417	2418	2419	2420	2421	2422	2423	2424	2425	2426	2427	2428	2429	2430	2431
98x	2432	2433	2434	2435	2436	2437	2438	2439	2440	2441	2442	2443	2444	2445	2446	2447
99x	2448	2449	2450	2451	2452	2453	2454	2455	2456	2457	2458	2459	2460	2461	2462	2463
9Ax	2464	2465	2466	2467	2468	2469	2470	2471	2472	2473	2474	2475	2476	2477	2478	2479
9Bx	2480	2481	2482	2483	2484	2485	2486	2487	2488	2489	2490	2491	2492	2493	2494	2495
9Cx	2496	2497	2498	2499	2500	2501	2502	2503	2504	2505	2506	2507	2508	2509	2510	2511
9Dx	2512	2513	2514	2515	2516	2517	2518	2519	2520	2521	2522	2523	2524	2525	2526	2527
9Ex	2528	2529	2530	2531	2532	2533	2534	2535	2536	2537	2538	2539	2540	2541	2542	2543
9Fx	2544	2545	2546	2547	2548	2549	2550	2551	2552	2553	2554	2555	2556	2557	2558	2559
A0x	2560	2561	2562	2563	2564	2565	2566	2567	2568	2569	2570	2571	2572	2573	2574	2575
A1x	2576	2577	2578	2579	2580	2581	2582	2583	2584	2585	2586	2587	2588	2589	2590	2591
A2x	2592	2593	2594	2595	2596	2597	2598	2599	2600	2601	2602	2603	2604	2605	2606	2607
A3x	2608	2609	2610	2611	2612	2613	2614	2615	2616	2617	2618	2619	2620	2621	2622	2623
A4x	2624	2625	2626	2627	2628	2629	2630	2631	2632	2633	2634	2635	2636	2637	2638	2639
A5x	2640	2641	2642	2643	2644	2645	2646	2647	2648	2649	2650	2651	2652	2653	2654	2655
A6x	2656	2657	2658	2659	2660	2661	2662	2663	2664	2665	2666	2667	2668	2669	2670	2671
A7x	2672	2673	2674	2675	2676	2677	2678	2679	2680	2681	2682	2683	2684	2685	2686	2687
A8x	2688	2689	2690	2691	2692	2693	2694	2695	2696	2697	2698	2699	2700	2701	2702	2703
A9x	2704	2705	2706	2707	2708	2709	2710	2711	2712	2713	2714	2715	2716	2717	2718	2719
AAx	2720	2721	2722	2723	2724	2725	2726	2727	2728	2729	2730	2731	2732	2733	2734	2735
ABx	2736	2737	2738	2739	2740	2741	2742	2743	2744	2745	2746	2747	2748	2749	2750	2751
ACx	2752	2753	2754	2755	2756	2757	2758	2759	2760	2761	2762	2763	2764	2765	2766	2767
ADx	2768	2769	2770	2771	2772	2773	2774	2775	2776	2777	2778	2779	2780	2781	2782	2783
AEx	2784	2785	2786	2787	2788	2789	2790	2791	2792	2793	2794	2795	2796	2797	2798	2799
AFx	2800	2801	2802	2803	2804	2805	2806	2807	2808	2809	2810	2811	2812	2813	2814	2815
B0x	2816	2817	2818	2819	2820	2821	2822	2823	2824	2825	2826	2827	2828	2829	2830	2831
B1x	2832	2833	2834	2835	2836	2837	2838	2839	2840	2841	2842	2843	2844	2845	2846	2847
B2x	2848	2849	2850	2851	2852	2853	2854	2855	2856	2857	2858	2859	2860	2861	2862	2863
B3x	2864	2865	2866	2867	2868	2869	2870	2871	2872	2873	2874	2875	2876	2877	2878	2679
B4x	2880	2881	2882	2883	2884	2885	2886	2887	2888	2889	2890	2891	2892	2893	2894	2895
B5x	2896	2897	2898	2899	2900	2901	2902	2903	2904	2905	2906	2907	2908	2909	2910	2911
B6x	2912	2913	2914	2915	2916	2917	2918	2919	2920	2921	2922	2923	2924	2925	2926	2927
B7x	2928	2929	2930	2931	2932	2933	2934	2935	2936	2937	2938	2939	2940	2941	2942	2943
B8x	2944	2945	2946	2947	2948	2949	2950	2951	2952	2953	2954	2955	2956	2957	2958	2959
B9x	2960	2961	2962	2963	2964	2965	2966	2967	2968	2969	2970	2971	2972	2973	2974	2975
BAx	2976	2977	2978	2979	2980	2981	2982	2983	2984	2985	2986	2987	2988	2989	2990	2991
BBx	2992	2993	2994	2995	2996	2997	2998	2999	3000	3001	3002	3003	3004	3005	3006	3007
BCx	3008	3009	3010	3011	3012	3013	3014	3015	3016	3017	3018	3019	3020	3021	3022	3023
BDx	3024	3025	3026	3027	3028	3029	3030	3031	3032	3033	3034	3035	3036	3037	3038	3039
BEx	3040	3041	3042	3043	3044	3045	3046	3047	3048	3049	3050	3051	3052	3053	3054	3055
BFx	3056	3057	3058	3059	3060	3061	3062	3063	3064	3065	3066	3067	3068	3069	3070	3071

	x = 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C0x	3072	3073	3074	3075	3076	3077	3078	3079	3080	3081	3082	3083	3084	3085	3086	3087
C1x	3088	3089	3090	3091	3092	3093	3094	3095	3096	3097	3098	3099	3100	3101	3102	3103
C2x	3104	3105	3106	3107	3108	3109	3110	3111	3112	3113	3114	3115	3116	3117	3118	3119
C3x	3120	3121	3122	3123	3124	3125	3126	3127	3128	3129	3130	3131	3132	3133	3134	3135
C4x	3136	3137	3138	3139	3140	3141	3142	3143	3144	3145	3146	3147	3148	3149	3150	3151
C5x	3152	3153	3154	3155	3156	3157	3158	3159	3160	3161	3162	3163	3164	3165	3166	3167
C6x	3168	3169	3170	3171	3172	3173	3174	3175	3176	3177	3178	3179	3180	3181	3182	3183
C7x	3184	3185	3186	3187	3188	3189	3190	3191	3192	3193	3194	3195	3196	3197	3198	3199
C8x	3200	3201	3202	3203	3204	3205	3206	3207	3208	3209	3210	3211	3212	3213	3214	3215
C9x	3216	3217	3218	3219	3220	3221	3222	3223	3224	3225	3226	3227	3228	3229	3230	3231
CAx	3232	3233	3234	3235	3236	3237	3238	3239	3240	3241	3242	3243	3244	3245	3246	3247
CBx	3248	3249	3250	3251	3252	3253	3254	3255	3256	3257	3258	3259	3260	3261	3262	3263
CCx	3264	3265	3266	3267	3268	3269	3270	3271	3272	3273	3274	3275	3276	3277	3278	3279
CDx	3280	3281	3282	3283	3284	3285	3286	3287	3288	3289	3290	3291	3292	3293	3294	3295
CEx	3296	3297	3298	3299	3300	3301	3302	3303	3304	3305	3306	3307	3308	3309	3310	3311
CFx	3312	3313	3314	3315	3316	3317	3318	3319	3320	3321	3322	3323	3324	3325	3326	3327
D0x	3328	3329	3330	3331	3332	3333	3334	3335	3336	3337	3338	3339	3340	3341	3342	3343
D1x	3344	3345	3346	3347	3348	3349	3350	3351	3352	3353	3354	3355	3356	3357	3358	3359
D2x	3360	3361	3362	3363	3364	3365	3366	3367	3368	3369	3370	3371	3372	3373	3374	3375
D3x	3376	3377	3378	3379	3380	3381	3382	3383	3384	3385	3386	3387	3388	3389	3390	3391
D4x	3392	3393	3394	3395	3396	3397	3398	3399	3400	3401	3402	3403	3404	3405	3406	3407
D5x	3408	3409	3410	3411	3412	3413	3414	3415	3416	3417	3418	3419	3420	3421	3422	3423
D6x	3424	3425	3426	3427	3428	3429	3430	3431	3432	3433	3434	3435	3436	3437	3438	3439
D7x	3440	3441	3442	3443	3444	3445	3446	3447	3448	3449	3450	3451	3452	3453	3454	3455
D8x	3456	3457	3458	3459	3460	3461	3462	3463	3464	3465	3466	3467	3468	3469	3470	3471
D9x	3472	3473	3474	3475	3476	3477	3478	3479	3480	3481	3482	3483	3484	3485	3486	3487
DAx	3488	3489	3490	3491	3492	3493	3494	3495	3496	3497	3498	3499	3500	3501	3502	3503
DBx	3504	3505	3506	3507	3508	3509	3510	3511	3512	3513	3514	3515	3516	3517	3518	3519
DCx	3520	3521	3522	3523	3524	3525	3526	3527	3528	3529	3530	3531	3532	3533	3534	3535
DDx	3536	3537	3538	3539	3540	3541	3542	3543	3544	3545	3546	3547	3548	3549	3550	3551
DEx	3552	3553	3554	3555	3556	3557	3558	3559	3560	3561	3562	3563	3564	3565	3566	3567
DFx	3568	3569	3570	3571	3572	3573	3574	3575	3576	3577	3578	3579	3580	3581	3582	3583
E0x	3584	3585	3586	3587	3588	3589	3590	3591	3592	3593	3594	3595	3596	3597	3598	3599
E1x	3600	3601	3602	3603	3604	3605	3606	3607	3608	3609	3610	3611	3612	3613	3614	3615
E2x	3616	3617	3618	3619	3620	3621	3622	3623	3624	3625	3626	3627	3628	3629	3630	3631
E3x	3632	3633	3634	3635	3636	3637	3638	3639	3640	3641	3642	3643	3644	3645	3646	3647
E4x	3648	3649	3650	3651	3652	3653	3654	3655	3656	3657	3658	3659	3660	3661	3662	3663
E5x	3664	3665	3666	3667	3668	3669	3670	3671	3672	3673	3674	3675	3676	3677	3678	3679
E6x	3680	3681	3682	3683	3684	3685	3686	3687	3688	3689	3690	3691	3692	3693	3694	3695
E7x	3696	3697	3698	3699	3700	3701	3702	3703	3704	3705	3706	3707	3708	3709	3710	3711
E8x	3712	3713	3714	3715	3716	3717	3718	3719	3720	3721	3722	3723	3724	3725	3726	3727
E9x	3728	3729	3730	3731	3732	3733	3734	3735	3736	3737	3738	3739	3740	3741	3742	3743
EAx	3744	3745	3746	3747	3748	3749	3750	3751	3752	3753	3754	3755	3756	3757	3758	3759
EBx	3760	3761	3762	3763	3764	3765	3766	3767	3768	3769	3770	3771	3772	3773	3774	3775
ECx	3776	3777	3778	3779	3780	3781	3782	3783	3784	3785	3786	3787	3788	3789	3790	3791
EDx	3792	3793	3794	3795	3796	3797	3798	3799	3800	3801	3802	3803	3804	3805	3806	3807
EEx	3808	3809	3810	3811	3812	3813	3814	3815	3816	3817	3818	3819	3820	3821	3822	3823
EFx	3824	3825	3826	3827	3828	3829	3830	3831	3832	3833	3834	3835	3836	3837	3838	3839
F0x	3840	3841	3842	3843	3844	3845	3846	3847	3848	3849	3850	3851	3852	3853	3854	3855
F1x	3856	3857	3858	3859	3860	3861	3862	3863	3864	3865	3866	3867	3868	3869	3870	3871
F2x	3872	3873	3874	3875	3876	3877	3878	3879	3880	3881	3882	3883	3884	3885	3886	3887
F3x	3888	3889	3890	3891	3892	3893	3894	3895	3896	3897	3898	3899	3900	3901	3902	3903
F4x	3904	3905	3906	3907	3908	3909	3910	3911	3912	3913	3914	3915	3916	3917	3918	3919
F5x	3920	3921	3922	3923	3924	3925	3926	3927	3928	3929	3930	3931	3932	3933	3934	3935
F6x	3936	3937	3938	3939	3940	3941	3942	3943	3944	3945	3946	3947	3948	3949	3950	3951
F7x	3952	3953	3954	3955	3956	3957	3958	3959	3960	3961	3962	3963	3964	3965	3966	3967
F8x	3968	3969	3970	3971	3972	3973	3974	3975	3976	3977	3978	3979	3980	3981	3982	3983
F9x	3984	3985	3986	3987	3988	3989	3990	3991	3992	3993	3994	3995	3996	3997	3998	3999
FAx	4000	4001	4002	4003	4004	4005	4006	4007	4008	4009	4010	4011	4012	4013	4014	4015
FBx	4016	4017	4018	4019	4020	4021	4022	4023	4024	4025	4026	4027	4028	4029	4030	4031
FCx	4032	4033	4034	4035	4036	4037	4038	4039	4040	4041	4042	4043	4044	4045	4046	4047
FDx	4048	4049	4050	4051	4052	4053	4054	4055	4056	4057	4058	4059	4060	4061	4062	4063
FEx	4064	4065	4066	4067	4068	4069	4070	4071	4072	4073	4074	4075	4076	4077	4078	4079
FFx	4080	4081	4082	4083	4084	4085	4086	4087	4088	4089	4090	4091	4092	4093	4094	4095

This page left blank intentionally.

Appendix III: Machine Instruction Format

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
RR	8 Operation Code; 4 R1; 4 R2	R1,R2	All RR instructions except BCR,SPM, and SVC
	8 Operation Code; 4 M1; 4 R2	M1,R2	BCR
	8 Operation Code; 4 R1; (blank)	R1	SPM
	8 Operation Code; 8 I	I (See Notes 1,6,8, and 9)	SVC
RX	8 Operation Code; 4 R1; 4 X2; 4 B2; 12 D2	R1,D2(X2,B2) R1,D2(,B2) R1,S2(X2) R1,S2	All RX instructions except BC
	8 Operation Code; 4 M1; 4 X2; 4 B2; 12 D2	M1,D2(X2,B2) M1,D2(,B2) M1,S2(X2) M1,S2 (See Notes 1,6,8, and 9)	BC
RS	8 Operation Code; 4 R1; 4 R3; 4 B2; 12 D2	R1,R3,D2(B2) R1,R3,S2	BXH,BXLE,CDS,CS,LM,SIGP, STM,LCTL,STCTL
	8 Operation Code; 4 R1; 4; 4 B2; 12 D2	R1,D2(B2) R1,S2	All shift instructions
	8 Operation Code; 4 R1; 4 M3; 4 B2; 12 D2	R1,M3,D2(B2) R1,M3,S2 (See Notes 1-3,7, 8,and 9)	ICM,STCM,CLM

	BASIC MACHINE FORMAT	ASSEMBLER OPERAND FIELD FORMAT	APPLICABLE INSTRUCTIONS
SI	8 Operation Code | 8 I2 | 4 B1 | 12 D1	D1(B1),I2 S1,I2	All SI instructions except those listed for the other SI format.
	8 Operation Code | (blank) | 4 B1 | 12 D1	D1(B1) S1 (See Notes 2,3,6, 7,8 and 10)	LPSW,SSM,TIO,TCH,TS
S	16 Two-byte Operation Code | 4 B1 | 12 D1	D1(B1) S1 (See Notes 2, 3, and 7)	SCK,STCK,STIDP,SIOF,STIDC, SIO,HIO,HDV SCKC,STCKC,SPT,STPT,PTLB, RRB CLRIO,IPK,SPKA,SPX,STAP, STPX
SS	8 Operation Code | 4 L1 | 4 L2 | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L1,B1),D2(L2,B2) S1(L1),S2(L2)	PACK,UNPK,MVO,AP, CP,DP,MP,SP,ZAP
	8 Operation Code | 8 L | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L,B1),D2(B2) S1(L),S2	NC,OC,XC,CLC,MVC,MVN, MVZ,TR,TRT,ED,EDMK
	8 Operation Code | 4 L1 | 4 I3 | 4 B1 | 12 D1 | 4 B2 | 12 D2	D1(L1,B1),D2(B2),I3 S1(L1),S2,I3 S1,S2,I3 (See Notes 2,3,5,6, 7 and 10)	SRP

<u>Notes for Appendix III:</u>

1. R1, R2, and R3 are absolute expressions that specify general or floating-point registers. The general register numbers are 0 through 15; floating-point register numbers are 0, 2, 4, and 6.

2. D1 and D2 are absolute expressions that specify displacements. A value of 0 - 4095 may be specified.

3. B1 and B2 are absolute expressions that specify base registers. Register numbers are 0 - 15.

4. X2 is an absolute expression that specifies an index register. Register numbers are 0 - 15.

5. L, L1, and L2 are absolute expressions that specify field lengths. An L expression can specify a value of 1 - 256. L1 and L2 expressions can specify a value of 1 - 16. In all cases, the assembled value will be one less than the specified value.

6. I, I2, and I3 are absolute expressions that provide immediate data. The value of I and I2 may be 0 - 255. The value of I3 may be 0 - 9.

7. S1 and S2 are absolute or relocatable expressions that specify an address.

8. RR, RS, and SI instruction fields that are blank under BASIC MACHINE FORMAT are not examined during instruction execution. The fields are not written in the symbolic operand, but are assembled as binary zeros.

9. M1 and M3 specify a 4-bit mask.

10. In IBM System/370 the SIO, HIO, HDV and SIOF operation codes occupy one byte and the low order bit of the second byte. In all other systems the HIO and SIO operation codes occupy only the first byte of the instruction.

Appendix IV, pages 391-406, has been removed. The following manuals are the definitive publications for machine instructions:

IBM System/370 Principles of Operation

IBM 4300 Processors Principles of Operation for ECPS:VSE Mode

here 391 through page 406 retained for info

Appendix IV: Machine Instruction Mnemonic Codes

This appendix contains two tables of the mnemonic operation codes for all machine instructions that can be represented in assembler language, including extended mnemonic operation codes.

The first table is in alphabetic order by instruction. The second table is in numeric order by operation code.

In the first table is indicated: both the mnemonic and machine operation codes, explicit and implicit operand formats, program interruptions possible, and condition code set.

The column headings in the first table and the information each column provides follow:

Instruction: This column contains the name of the instruction associated with the mnemonic operation code.

Mnemonic Operation Code: This column contains the mnemonic operation code for the instruction. This is written in the operation field when coding the instruction.

Machine Operation Code: This column contains the hexadecimal equivalent of the actual machine operation code. The operation code will apppear in this form in most storage dumps and when displayed on the system control panel. For extended mnemonics, this column also contains the mnemonic code of the instruction from which the extended mnemonic is derived.

Operand Format: This column shows the symbolic format of the operand field in both explicit and implicit form. For both forms, R1, R2, and R3 indicate general registers in operand one, two, and three respectively. X2 indicates a general register used as an index register in the second operand. Instructions which require an index register (X2) but are not to be indexed are shown with a 0 replacing X2. L, L1, and L2 indicate lengths for either operand, operand one, or operand two respectively. M1 and M3 indicate a 4-bit mask in operands one and three respectively. I, I2, and I3 indicate immediate data eight bits long (I and I2) or four bits long (I3).

For the explicit format, D1 and D2 indicate a displacement and B1 and B2 indicate a base register for operands one and two.

For the implicit format, D1, B1, and D2, B2 are replaced by S1, and S2 which indicate a storage address in operands one and two.

Type of Instruction: This column gives the basic machine format of the instruction (RR, RX, SI, or SS). If an instruction is included in a special feature or is an extended mnemonic, this is also indicated.

Program Interruptions Possible: This column indicates the possible program interruptions for this instruction. The abreviations used are: A - Addressing, S - Specification, Ov - Overflow, P - Protection, Op - Operation (if feature is not installed), and Other - other interruptions which are listed. The type of overflow is indicated by: D - Decimal, E - Exponent, or F - Fixed Point.

Condition code set: The condition codes set as a result of this instruction are indicated in this column. (See legend following the table.)

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
Add	A	5A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add	AR	1A	R1, R2	
Add Decimal	AP	FA	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Add Halfword	AH	4A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Logical	AL	5E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Logical	ALR	1E	R1, R2	
Add Normalized, Extended	AXR	36	R1, R2	
Add Normalized, Long	AD	6A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Normalized, Long	ADR	2A	R1, R2	
Add Normalized, Short	AE	7A	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Normalized, Short	AER	3A	R1, R2	
Add Unnormalized, Long	AW	6E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Unnormalized, Long	AWR	2E	R1, R2	
Add Unnormalized, Short	AU	7E	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Add Unnormalized, Short	AUR	3E	R1, R2	
And Logical	N	54	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
And Logical	NC	D4	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2
And Logical	NR	14	R1, R2	
And Logical Immediate	NI	94	D1(B1), I2	S1, I2
Branch and Link	BAL	45	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Branch and Link	BALR	05	R1, R2	
Branch and Save	BAS	4D	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Branch and Save	BASR	0D	R1, R2	
Branch on Condition	BC	47	M1, D2(X2, B2) or M1, D2(, B2)	M1, S2(X2) or M1, S2
Branch on Condition	BCR	07	M1, R2	
Branch on Count	BCT	46	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Branch on Count	BCTR	06	R1, R2	
Branch on Equal	BE	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Equal	BER	07(BCR 8)	R2	
Branch on High	BH	47(BC 2)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on High	BHR	07(BCR 2)	R2	
Branch on Index High	BXH	86	R1, R3, D2(B2)	R1, R3, S2
Branch on Index Low or Equal	BXLE	87	R1, R3, D2(B2)	R1, R3, S2
Branch on Low	BL	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Low	BLR	07(BCR 4)	R2	
Branch if Mixed	BM	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Mixed	BMR	07(BCR 4)	R2	
Branch on Minus	BM	47(BC 4)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Minus	BMR	07(BCR 4)	R2	
Branch on Not Equal	BNE	47(BC 7)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Equal	BNER	07(BCR 7)	R2	
Branch on Not High	BNH	47(BC 13)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not High	BNHR	07(BCR 13)	R2	
Branch on Not Low	BNL	47(BC 11)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Low	BNLR	07(BCR 11)	R2	
Branch if Not Mixed	BNM	47(BC 11)	D2(X2,B2) or D2(,B2)	S2(X2) or S2
Branch if Not Mixed	BNMR	07(BCR 11)	R2	
Branch on Not Minus	BNM	47(BC 11)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Minus	BNMR	07(BCR 11)	R2	
Branch if Not Ones	BNO	47(BC 14)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Not Ones	BNOR	07(BCR 14)	R2	
Branch on No Overflow	BNO	47(BC 14)	D2(X2,B2) or D2(,B2)	S2(X2) or S2
Branch on No Overflow	BNOR	07(BCR 14)	R2	
Branch on Not Plus	BNP	47(BC 13)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Plus	BNPR	07(BCR 13)	R2	
Branch if Not Zeros	BNZ	47(BC 7)	D2(X2,B2) or D2(,B2)	S2(X2) or S2
Branch if Not Zeros	BNZR	07(BCR 7)	R2	
Branch on Not Zero	BNZ	47(BC 7)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Not Zero	BNZR	07(BCR 7)	R2	
Branch if Ones	BO	47(BC 1)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Ones	BOR	07(BCR 1)	R2	
Branch on Overflow	BO	47(BC 1)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Overflow	BOR	07(BCR 1)	R2	
Branch on Plus	BP	47(BC 2)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Plus	BPR	07(BCR 2)	R2	
Branch if Zeros	BZ	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch if Zeros	BZR	07(BCR 8)	R2	
Branch on Zero	BZ	47(BC 8)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch on Zero	BZR	07(BCR 8)	R2	
Branch Unconditional	B	47(BC 15)	D2(X2, B2) or D2(, B2)	S2(X2) or S2
Branch Unconditional	BR	07(BCR 15)	R2	
Clear I/O	CLRIO	9D01	D2(B2)	S2
DOS/VSE only Clear Storage Page	CLRP	B215	D2(B2)	S2
Compare Algebraic	C	59	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2 or R1, S2
Compare Algebraic	CR	19	R1, R2	
Compare and Swap	CS	BA	R1, R3, D2, (B2)	R1, R3, S2
Compare Decimal	CP	F9	D1(L1, B1), D2(L2, B2)	S1(L1), S2(L2) or S1, S2
Compare Double and Swap	CDS	BB	R1, R3, D2(B2)	R1, R3, S2
Compare Halfword	CH	49	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare Logical	CL	55	R1, D2(X2, B2) or R1, D2(, B2)	R1, S2(X2) or R1, S2
Compare Logical	CLC	D5	D1(L, B1), D2(B2)	S1(L), S2 or S1, S2

Instruction	Type of Instruction	Program Interruption Possible						Condition Code Set			
	Instruction	A	S	Ov	P	Op	Other	00	01	10	11
Add	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add	RR			F				Sum=0	Sum<0	Sum>0	Overflow
Add Decimal	SS, Decimal	x		D	x	x	Data	Sum=0	Sum<0	Sum>0	Overflow
Add Halfword	RX	x	x	F				Sum=0	Sum<0	Sum>0	Overflow
Add Logical	RX	x	x					Sum=0 (H)	Sum 0 (H)	Sum= 0 (I)	Sum 0 (I)
Add Logical	RR							Sum=0 (H)	Sum= 0 (H)	Sum= 0 (I)	Sum 0 (I)
Add Normalized, Extended	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Add Normalized, Long	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	
Add Normalized, Long	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Add Normalized, Short	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	
Add Normalized, Short	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Add Unnormalized, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	
Add Unnormalized, Long	RR, Floating Pt.		x	E		x	C	R	L	M	
Add Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	
Add Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	
And Logical	RX	x	x					J	K		
And Logical	SS	x			x			J	K		
And Logical	RR							J	K		
And Logical Immediate	SI	x			x			J	K		
Branch and Link	RX							N	N	N	N
Branch and Link	RR							N	N	N	N
Branch and Save	RX					x		N	N	N	N
Branch and Save	RR					x		N	N	N	N
Branch on Condition	RX							N	N	N	N
Branch on Condition	RR							N	N	N	N
Branch on Count	RX							N	N	N	N
Branch on Count	RR							N	N	N	N
Branch on Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Equal	RR, Ext. Mnemonic							N	N	N	N
Branch on High	RX, Ext. Mnemonic							N	N	N	N
Branch on High	RR, Ext. Mnemonic							N	N	N	N
Branch on Index High	RS							N	N	N	N
Branch on Index Low or Equal	RS							N	N	N	N
Branch on Low	RX, Ext. Mnemonic							N	N	N	N
Branch on Low	RR, Ext. Mnemonic							N	N	N	N
Branch if Mixed	RX, Ext. Mnemonic							N	N	N	N
Branch if Mixed	RR, Ext. Mnemonic							N	N	N	N
Branch on Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Minus	RR, Ext. Mnemonic							N	N	N	N
Branch on Not Equal	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Equal	RR, Ext. Mnemonic							N	N	N	N
Branch on Not High	RX, Ext. Mnemonic							N	N	N	N
Branch on Not High	RR, Ext. Mnemonic							N	N	N	N
Branch on Not Low	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Low	RR, Ext. Mnemonic							N	N	N	N
Branch if Not Mixed	RX, Ext. Mnemonic							N	N	N	N
Branch if Not Mixed	RR, Ext. Mnemonic							N	N	N	N
Branch on Not Minus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Minus	RR, Ext. Mnemonic							N	N	N	N
Branch if Not Ones	RX, Ext. Mnemonic							N	N	N	N
Branch if Not Ones	RR, Ext. Mnemonic							N	N	N	N
Branch on No Overflow	RX, Ext. Mnemonic							N	N	N	N
Branch on No Overflow	RR, Ext. Mnemonic							N	N	N	N
Branch on Not Plus	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Plus	RR, Ext. Mnemonic							N	N	N	N
Branch if Not Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch if Not Zeros	RR, Ext. Mnemonic							N	N	N	N
Branch on Not Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch on Not Zeros	RR, Ext. Mnemonic							N	N	N	N
Branch if Ones	RX, Ext. Mnemonic							N	N	N	N
Branch if Ones	RR, Ext. Mnemonic							N	N	N	N
Branch on Overflow	RX, Ext. Mnemonic							N	N	N	N
Branch on Overflow	RR, Ext. Mnemonic							N	N	N	N
Branch on Plus	RX, Ext. Mnemonic							N	N	N	N
Branch on Plus	RR, Ext. Mnemonic							N	N	N	N
Branch if Zeros	RX, Ext. Mnemonic							N	N	N	N
Branch if Zeros	RR, Ext. Mnemonic							N	N	N	N
Branch on Zero	RX, Ext. Mnemonic							N	N	N	N
Branch on Zero	RR, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RX, Ext. Mnemonic							N	N	N	N
Branch Unconditional	RR, Ext. Mnemonic							N	N	N	N
Clear I/O	S						A	AAX	CC	EE	KK
DOS/VSE only: Clear Storage Page	S	x			x		A,GB				
Compare Algebraic	RX	x	x					Z	AA	BB	
Compare Algebraic	RR							Z	AA	BB	
Compare and Swap	RS	x	x		x	x		Z	AAW		
Compare Decimal	SS, Decimal	x				x	Data	Z	AA	BB	
Compare Double and Swap	RS	x	x		x	x		Z	AAW		
Compare Halfword	RX	x	x					Z	AA	BB	
Compare Logical	RX	x	x					Z	AA	BB	
Compare Logical	SS	x	x					Z	AA	BB	

	Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
	Compare Logical	CLR	15	R1,R2	
	Compare Logical Characters under Mask	CLM	BD	R1,M3,D2(B2)	R1,M3,S2
	Compare Logical Immediate	CLI	95	D1(B1),I2	S1,I2
	Compare Logical Long	CLCL	0F	R1,R2	
	Compare, Long	CD	69	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
	Compare, Long	CDR	29	R1,R2	
	Compare, Short	CE	79	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
	Compare, Short	CER	39	R1,R2	
DOS/VSE only	Connect Page	CTP	B0	R1,D2(B2)	R1,S2
	Convert to Binary	CVB	4F	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
	Convert to Decimal	CVD	4E	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2)or R1,S2
DOS/VSE only	Deconfigure Page	DEP	B21B	D2(B2)	S2
DOS/VSE only	Disconnect Page	DCTP	B21C	D2(B2)	S2
	Divide	D	5D	R1,D2(X2,B2) or R1,D2(,B2)	R1, S2(X2) or R1,S2
	Divide	DR	1D	R1,R2	
	Divide Decimal	DP	FD	D1,(L1,B1),D2(L2,B2)	S1(L1),S2(L2)or S1,S2
	Divide, Long	DD	6D	R1,D2(X2,B2),or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Divide, Long	DDR	2D	R1,R2	
	Divide, Short	DE	7D	R1,D2(X2,B2)or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Divide, Short	DER	3D	R1,R2	
	Edit	ED	DE	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Edit and Mark	EDMK	DF	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Exclusive Or	X	57	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Exclusive Or	XC	D7	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Exclusive Or	XR	17	R1,R2	
	Exclusive Or Immediate	XI	97	D1(B1),I2	S1,I2
	Execute	EX	44	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) R1,S2
	Halve, Long	HDR	24	R1,R2	
	Halve, Short	HER	34	R1,R2	
	Halt Device	HDV	9E01[1]	D1,B1	S1
	Halt I/O	HIO	9E00[1]	D1(B1)	
DOS/VSE only	Insert Page Bits	IPB	B4	R1,D2 (B2)	R1,S2
	Insert Character	IC	43	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Insert Characters under Mask	ICM	BF	R1,M3,D2(B2)	R1,M3,S2
	Insert PSW Key	IPK	B20B		
	Insert Storage Key	ISK	09	R1,R2	
	Load	L	58	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load	LR	18	R1,R2	
	Load Address	LA	41	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load and Test	LTR	12	R1,R2	
	Load and Test, Long	LTDR	22	R1,R2	
	Load and Test, Short	LTER	32	R1,R2	
	Load Complement	LCR	13	R1,R2	
	Load Complement, Long	LCDR	23	R1,R2	
	Load Complement, Short	LCER	33	R1,R2	
	Load Control	LCTL	B7	R1,R3,D2(B2)	R1,R3,S2
DOS/VSE only	Load Frame Index	LFI	B8	R1,D2(B2)	R1,S2
	Load Halfword	LH	48	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load, Long	LD	68	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load, Long	LDR	28	R1,R2	
	Load Multiple	LM	98	R1,R3,D2(B2)	R1,R3,S2
	Load Negative	LNR	11	R1,R2	
	Load Negative, Long	LNDR	21	R1,R2	
	Load Negative, Short	LNER	31	R1,R2	
	Load Positive	LPR	10	R1,R2	
	Load Positive, Long	LPDR	20	R1,R2	
	Load Positive, Short	LPER	30	R1,R2	
	Load PSW	LPSW	82	D1(B1)	S1
Not DOS/VSE	Load Real Address	LRA	B1	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load Rounded, Extended to Long	LRDR	25	R1,R2	
	Load Rounded, Long to Short	LRER	35	R1,R2	
	Load, Short	LE	78	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Load, Short	LER	38	R1,R2	
DOS/VSE only	Make Addressable	MAD	B21D	D2(B2)	S2
DOS/VSE only	Make Unaddressable	MUN	B21E	D2(B2)	S2
	Monitor Call	MC	AF	D1(B1),I2	S1,I2
	Move Characters	MVC	D2	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Move Immediate	MVI	92	D1(B1),I2	S1,I2

[1] See Note 1 at end of this appendix

	Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
			A	S	Ov	P	Op	Other	00	01	10	11
	Compare Logical	RR	x						Z	AA	BB	
	Compare Logical Characters under Mask	RS	x			x	x		XX	YY	ZZ	
	Compare Logical Immediate	SI	x						Z	AA	BB	
	Compare Logical Long	RR	x	x		x	x		Z	AA	BB	
	Compare, Long	RX, Floating Pt.	x	x			x		Z	AA	BB	
	Compare, Long	RR, Floating Pt.	x	x			x		Z	AA	BB	
	Compare, Short	RX, Floating Pt.	x	x			x		Z	AA	BB	
	Compare, Short	RR, Floating Pt.		x			x		Z	AA	BB	
DOS/VSE only	Connect Page	RS	x	x			x	A, GC	ABA	ABB	ABC	
	Convert to Binary	RX	x	x				Data, F	N	N	N	N
	Convert to Decimal	RX	x	x		x			N	N	N	N
DOS/VSE only	Deconfigure Page	S	x	x			x	A, GC				
DOS/VSE only	Disconnect Page	S	x	x			x	A, GC	ADB	ABE		
	Divide	RX	x	x				F	N	N	N	N
	Divide	RR		x				F	N	N	N	N
	Divide Decimal	SS, Decimal	x	x		x	x	D, Data	N	N	N	N
	Divide, Long	RX, Floating Pt.	x	x	E		x	B, E	N	N	N	N
	Divide, Long	RR, Floating Pt.		x	E		x	B, E	N	N	N	N
	Divide, Short	RX, Floating Pt.	x	x	E		x	B, E	N	N	N	N
	Divide, Short	RR, Floating Pt.		x	E		x	B, E	N	N	N	N
	Edit	SS, Decimal	x			x	x	Data	S	T	U	
	Edit and Mark	SS, Decimal	x			x	x	Data	S	T	U	
	Exclusive Or	RX	x	x					J	K		
	Exclusive Or	SS	x			x			J	K		
	Exclusive Or	RR							J	K		
	Exclusive Or Immediate	SI	x			x			J	K		
	Execute	RX	x	x				G	(May be set by this instruction)			
	Halve, Long	RR, Floating Pt.		x			x		N	N	N	N
	Halve, Short	RR, Floating Pt.		x			x		N	N	N	N
	Halt Device	S						A	AAM	CC	AAL	
	Halt I/O	S						A	DD	CC	GG	KK
DOS/VSE only	Insert Page Bits	RS	x				x	A				
	Insert Character	RX	x						N	N	N	N
	Insert Characters under Mask	RS	x			x	x		UU	TT	SS	
	Insert PSW Key	S					x	A				
	Insert Storage Key	RR	x	x			x	A	N	N	N	N
	Load	RX	x	x					N	N	N	N
	Load	RR							N	N	N	N
	Load Address	RX							N	N	N	N
	Load and Test	RR							J	L	M	
	Load and Test, Long	RR, Floating Pt.		x			x		R	L	M	
	Load and Test, Short	RR, Floating Pt.		x			x		R	L	M	
	Load Complement	RR			F				P	L	M	O
	Load Complement, Long	RR, Floating Pt.		x			x		R	L	M	
	Load Complement, Short	RR, Floating Pt.		x			x		R	L	M	
	Load Control	RS	x	x		x	x	A	N	N	N	N
DOS/VSE only	Load Frame Index	RS					x	A	ABF	ABG	ABH	ABI
	Load Halfword	RX	x	x					N	N	N	N
	Load, Long	RX, Floating Pt.	x	x			x		N	N	N	N
	Load, Long	RR, Floating Pt.		x			x		N	N	N	N
	Load Multiple	RS	x	x					N	N	N	N
	Load Negative	RR							J	L		
	Load Negative, Long	RR, Floating Pt.		x			x		R	L		
	Load Negative, Short	RR, Floating Pt.		x			x		R	L		
	Load Positive	RR			F				J		M	O
	Load Positive, Long	RR, Floating Pt.		x			x		R	L	M	
	Load Positive, Short	RR, Floating Pt.		x			x		R	L	M	
	Load PSW	SI	x	x				A	QQ	QQ	QQ	QQ
Not DOS/VSE	Load Real Address	RX	x	x			x	A	AAV	AAU	AAP	AAO
	Load Rounded, Extended to Long	RR, Floating Pt.		x	E		x		N	N	N	N
	Load Rounded, Long to Short	RR, Floating Pt.		x	E		x		N	N	N	N
	Load, Short	RX, Floating Pt.	x	x			x		N	N	N	N
	Load, Short	RR, Floating Pt.		x			x		N	N	N	N
DOS/VSE only	Make Addressable	S	x				x	A, GC	ADB	ABJ		
DOS/VSE only	Make Unaddressable	S	x	x			x	A, GC	ABK	ABL		
	Move Characters	SS	x			x			N	N	N	N
	Move Immediate	SI	x			x			N	N	N	N

	Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
	Move Long	MVCL	0F	R1,R2	
	Move Numerics	MVN	D1	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Move with Offset	MVO	F1	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
	Move Zones	MVZ	D3	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Multiply	M	5C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Multiply	MR	1C	R1,R2	
	Multiply Decimal	MP	FC	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
	Multiply Extended	MXR	26	R1,R2	
	Multiply Halfword	MH	4C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Multiply, Long	MD	6C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Multiply, Long	MDR	2C	R1,R2	
	Multiply, Long to Extended	MXD	67	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Multiply, Long to Extended	MXDR	27	R1,R2	
	Multiply, Short	ME	7C	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Multiply, Short	MER	3C	R1,R2	
	No Operation	NOP	47(BC 0)	D2(X2,B2) or D2(,B2)	S2(X2) or S2
	No Operation	NOPR	07(BC 0)	R2	
	Or Logical	O	56	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Or Logical	OC	D6	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
	Or Logical	OR	16	R1,R2	
	Or Logical Immediate	OI	96	D1(B1),I2	S1,I2
	Pack	PACK	F2	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
	Purge Translation Lookaside Buffer	PTLB	B20D	-	-
Not DOS/VSE	Read Direct	RDD	85	D1(B1),I2	S1,I2
	Reset Reference Bit	RRB	B213	D1(B1)	S1
DOS/VSE only	Retrieve Status and Page	RSP	D8	D1(,B1),D2(B2) or D1(B1),D2(B2)	S1,S2
DOS/VSE only	Set Page Bits	SPB	B5	R1,D2(B2)	R1,S2
	Set Clock	SCK	B204	D1(B1)	S1
	Set Clock Comparator	SCKC	B206	D1(B1)	S1
	Set CPU Timer	SPT	B208	D1(B1)	S1
	Set Prefix	SPX	B210	D2(B2)	S2
	Set Program Mask	SPM	04	R1	
	Set PSW Key from Address	SPKA	B20A	D,(B,)	S1
	Set Storage Key	SSK	08	R1,R2	
	Set System Mask	SSM	80	D1(B1)	S1
	Shift and Round Decimal	SRP	F0	D1(L1,B1),D2(B2),I3	S1(L1),S2,I3 or S1,S2,I3
	Shift Left Double Algebraic	SLDA	8F	R1,D2(B2)	R1,S2
	Shift Left Double Logical	SLDL	8D	R1,D2(B2)	R1,S2
	Shift Left Single Algebraic	SLA	8B	R1,D2(B2)	R1,S2
	Shift Left Single Logical	SLL	89	R1,D2(B2)	R1,S2
	Shift Right Double Algebraic	SRDA	8E	R1,D2(B2)	R1,S2
	Shift Right Double Logical	SRDL	8C	R1,D2(B2)	R1,S2
	Shift Right Single Algebraic	SRA	8A	R1,D2(B2)	R1,S2
	Shift Right Single Logical	SRL	88	R1,D2(B2)	R1,S2
	Signal Processor	SIGP	AE	R1,R3,D2(B2)	R1,R3,S2
	Start I/O	SIO	9C00[1]	D1(B1)	S1
	Start I/O Fast Release	SIOF	9C01[1]	D1(B1)	S1
	Store	ST	50	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
DOS/VSE only	Store Capacity Counts	STCAP	B21F	D2(B2)	S2
	Store Channel ID	STIDC	B203	D1(B1)	S1
	Store Character	STC	42	R1,D2(X2,B2) or R1,D2(,B2)	R1,D2(X2) or R1,S2
	Store Characters under Mask	STCM	BE	R1,M3,D2(B2)	R1,M3,S2
	Store Clock	STCK	B205	D1(B1)	S1
	Store Clock Comparator	STCKC	B207	D1(B1)	S1
	Store Control	STCTL	B6	R1,R3,D2(B2)	R1,R3,S2
	Store CPU address	STAP	B212	D2(B2)	S2
	Store CPU ID	STIDP	B202	D1(B1)	S1
	Store CPU Timer	STPT	B209	D1(B1)	S1
	Store Halfword	STH	40	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Store Long	STD	60	R1,D2(X2,B2)	R1,S2(X2) or R1,S2
	Store Multiple	STM	90	R1,R2,D2(B2)	R1,R2,S2
	Store Prefix	STPX	B211	D2(B2)	S2
	Store Short	STE	70	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Store Then AND System Mask	STNSM	AC	D1(B1),I2	S1,I2
	Store Then OR System Mask	STOSM	AD	D1(B1),I2	S1,I2
	Subtract	S	5B	R1,D2(X2)	R1,S2(X2) or R1,S2
	Subtract	SR	1B	R1,R2	
	Subtract Decimal	SP	FB	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
	Subtract Halfword	SH	4B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Subtract Logical	SL	5F	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
	Subtract Logical	SLR	1F	R1,R2	

[1] See Note 2 at end of this appendix

	Instruction	Type of Instruction	Program Interruptions Possible						Condition Code Set			
			A	S	Ov	P	Op	Other	00	01	10	11
	Move Long	RR	x	x		x	x		AAA	AAB	AAC	AAD
	Move Numerics	SS	x			x			N	N	N	N
	Move with Offset	SS	x			x			N	N	N	N
	Move Zones	SS	x			x			N	N	N	N
	Multiply	RX	x	x					N	N	N	N
	Multiply	RR		x					N	N	N	N
	Multiply Decimal	SS, Decimal	x	x		x	x	Data	N	N	N	N
	Multiply Extended	RR, Floating Pt.		x	E		x	B	N	N	N	N
	Multiply Halfword	RX	x	x					N	N	N	N
	Multiply, Long	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
	Multiply, Long	RR, Floating Pt.		x	E		x	B	N	N	N	N
	Multiply, Long/ Extended	RX, Floating Pt.	x	x	E	x	x	B	N	N	N	N
	Multiply, Long/ Extended	RR, Floating Pt.		x	E		x	B	N	N	N	N
	Multiply, Short	RX, Floating Pt.	x	x	E		x	B	N	N	N	N
	Multiply, Short	RR, Floating Pt.		x	E		x	B	N	N	N	N
	No Operation	RX, Ext. Mnemonic							N	N	N	N
	No Operation	RR, Ext. Mnemonic							N	N	N	N
	Or Logical	RX	x	x					J	K		
	Or Logical	SS	x			x			J	K		
	Or Logical	RR							J	K		
	Or Logical Immediate	SI	x			x			J	K		
	Pack	SS	x			x			N	N	N	N
Not DOS/VSE	Purge Translation Lookaside Buffer	S					x	A	N	N	N	N
Not DOS/VSE	Read Direct	SI	x			x	x	A	N	N	N	N
	Reset Reference Bit	S					x	A	AAQ	AAR	AAS	AAT
DOS/VSE only	Retrieve Status and Page	SS	x				x	A	ABM			ABN
DOS/VSE only	Set Page Bits	RS	x				x	A	AAQ	AAR	AAS	AAT
	Set Clock	S	x	x		x	x	A	AAE	AAF		AAG
	Set Clock Comparator	S	x	x		x	x	A	N	N	N	N
	Set CPU Timer	S	x	x		x	x	A	N	N	N	N
	Set Prefix	S					x	A				
	Set Program Mask	RR							RR	RR	RR	RR
	Set PSW Key from Address	S					x	A				
	Set Storage Key	RR	x	x			x	A	N	N	N	N
	Set System Mask	SI	x					A	N	N	N	N
	Shift Left Double Algebraic	RS		x	F				J	L	M	O
	Shift and Round Decimal	SS	x		D	x	x	Data	J	L	M	O
	Shift Left Double Logical	RS		x					N	N	N	N
	Shift Left Single Algebraic	RS			F				J	L	M	O
	Shift Left Single Logical	RS							N	N	N	N
	Shift Right Double Algebraic	RS		x					J	L	M	
	Shift Right Double Logical	RS		x					N	N	N	N
	Shift Right Single Algebraic	RS							J	L	M	
	Shift Right Single Logical	RS							N	N	N	N
	Signal Processor	RS					x	A	AAY	AAZ	EE	HH
	Start I/O	S						A	MM	CC	EE	KK
	Start I/O Fast Release	S						A	MM	CC	EE	KK
	Store	RX	x	x		x			N	N	N	N
DOS/VSE only	Store Capacity Counts	S	x			x		A				
	Store Channel ID	S					x	A	AAH	CC	AAI	KK
	Store Character	RX	x			x			N	N	N	N
	Store Characters under Mask	RS	x			x	x		N	N	N	N
	Store Clock	S	x			x	x		AAJ	AAK	AAN	AAG
	Store Clock Comparator	S	x	x		x	x	A	N	N	N	N
	Store Control	RS	x	x		x	x	A	N	N	N	N
	Store CPU Address	S					x	A				
	Store CPU ID	S	x	x		x	x	A	N	N	N	N
	Store CPU Timer	S	x	x		x	x	A	N	N	N	N
	Store Halfword	RX	x	x		x			N	N	N	N
	Store Long	RX, Floating Pt.	x	x		x	x		N	N	N	N
	Store Multiple	RS	x	x		x			N	N	N	N
	Store Prefix	S					x	A				
	Store Short	RX, Floating Pt.	x	x		x	x		N	N	N	N
	Store Then AND System Mask	SI	x			x	x	A	N	N	N	N
	Store Then OR System Mask	SI	x			x	x	A	N	N	N	N
	Subtract	RX	x	x	F				V	X	Y	O
	Subtract	RR			F				V	X	Y	O
	Subtract Decimal	SS, Decimal	x		D	x	x	Data	V	X	Y	O
	Subtract Halfword	RX	x	x	F				V	X	Y	O
	Subtract Logical	RX	x	x						W, H	V, I	W, I
	Subtract Logical	RR								W, H	V, I	W, I

Instruction	Mnemonic Operation Code	Machine Operation Code	Operand Format Explicit	Operand Format Implicit
Subtract Normalized, Extended	SXR	37	R1,R2	
Subtract Normalized, Long	SD	6B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Normalized, Long	SDR	2B	R1,R2	
Subtract Normalized, Short	SE	7B	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Normalized, Short	SER	3B	R1,R2	
Subtract Unnormalized, Long	SW	6F	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Unnormalized, Long	SWR	2F	R1,R2	
Subtract Unnormalized, Short	SU	7F	R1,D2(X2,B2) or R1,D2(,B2)	R1,S2(X2) or R1,S2
Subtract Unnormalized, Short	SUR	3F	R1,R2	
Supervisor Call	SVC	0A	I	
Test and Set	TS	93	D1(B1)	S1
Test Channel	TCH	9F	D1(B1)	S1
Test I/O	TIO	9D	D1(B1)	S1
Test Under Mask	TM	91	D1(B1),I2	S1,I2
Translate	TR	DC	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Translate and Test	TRT	DD	D1(L,B1),D2(B2)	S1(L),S2 or S1,S2
Unpack	UNPK	F3	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2
Write Direct	WRD	84	D1(B1),I2	S1,I2
Zero and Add Decimal	ZAP	F8	D1(L1,B1),D2(L2,B2)	S1(L1),S2(L2) or S1,S2

Instruction	Type of Instruction	Program Interruption Possible						Condition Code Set			
		A	S	Ov	P	Op	Other	00	01	10	11
Subtract Normalized, Extended	RR, Floating Pt.		x	E		x	B,C	R	L	M	
Subtract Normalized, Long	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Long	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RX, Floating Pt.	x	x	E		x	B,C	R	L	M	Q
Subtract Normalized, Short	RR, Floating Pt.		x	E		x	B,C	R	L	M	Q
Subtract Unnormaliezd, Long	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Unnormailezd, Long	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RX, Floating Pt.	x	x	E		x	C	R	L	M	Q
Subtract Unnormalized, Short	RR, Floating Pt.		x	E		x	C	R	L	M	Q
Supervisor Call	RR							N	N	N	N
Test and Set	SI	x			x			SS	TT		
Test Channel	SI						A	JJ	II	FF	HH
Test I/O	SI						A	LL	CC	EE	KK
Test under Mask	SI	x						UU	VV		WW
Translate	SS	x			x			N	N	N	N
Translate and Test	SS	x						PP	NN	OO	
Unpack	SS	x			x			N	N	N	N
Write Direct	SI	x				x	A	N	N	N	N
Zero and Add Decimal	SS, Decimal	x		D	x	x	Data	J	L	M	O

Program Interruptions Possible

Under Ov:
- D = Decimal
- E = Exponent
- F = Fixed Point

Under Other:
- A Privileged Operation
- B Exponent Underflow
- C Significance
- D Decimal Divide
- E Floating Point Divide
- F Fixed Point Divide
- G Execute
- GA Monitoring

Condition Code Set

- H No carry
- I Carry
- J Result = 0
- K Result is not equal to zero
- L Result is less than zero
- M Result is greater than zero
- N Not changed
- O Overflow
- P Result exponent underflows
- Q Result exponent overflows
- R Result fraction = 0
- S Result field equals to zero
- T Result field is less than zero
- U Result field is greater than zero
- V Difference = 0
- W Difference is not euqal to zero
- X Difference is less than zero
- Y Difference is greater than zero
- Z First operand equals second operand
- AA First operand is less than second operand
- BB First operand is greater than second operand
- CC CSW stored
- DD Channel and subchannel not working
- EE Channel or subchannel busy
- FF Channel operating in burst mode
- GB Page state
- GC Page transision
- GG Burst operation terminated
- HH Channel not operational
- II Interruption pending on channel
- JJ Channel available
- KK Not operational
- LL Available
- MM I/O operation initiated and channel proceeding with its execution
- NN Nonzero function byte found before the first operand field is exhausted
- OO Last function bytes are zero
- PP All function bytes are zero
- QQ Set according to bits 34 and 35 of the new PSW loaded
- RR Set according to bits 2 and 3 of the register specified by R1
- SS Leftmost bit of byte specified = 0
- TT Leftmost bit of byte specified = 1
- UU Selected bits are all zeros; mask is all zeros
- VV Selected bits are mixed (zeros and ones)
- WW Selected bits are all ones
- XX Selected bytes are equal, or mask in zero
- YY Selected field of first operand is low
- ZZ Selected field of first operand is high
- AAA First operand and second operand counts are equal
- AAB First operand count is lower
- AAC First operand count is higher
- AAD No movement because of destructive overlap
- AAE Clock value set
- AAF Clock value secure
- AAG Clock not operational
- AAH Channel ID correctly stored
- AAI Channel activity prohibited during ID
- AAJ Clock value is valid
- AAK Clock value not necessarily valid
- AAL Channel working with another device
- AAM Subchannel busy or interruption pending
- AAN Clock in error state
- AAO Segment- or page-table length violation
- AAP Page-table entry invalid (I-bit one)
- AAQ Reference bit zero, change bit zero
- AAR Reference bit zero, change bit one
- AAS Reference bit one, change bit zero
- AAT Reference bit one, change bit one
- AAU Segment table entry invalid (I-bit one)
- AAV Translation available
- AAW First and second hand operands equal, second operand replaced by the third operand
- AAX No operation is in progress for the addressed device
- AAY Order code accepted
- AAZ Status stored
- ABA Successful, block was disconnected, index returned
- ABB Page was already disconnected, index returned
- ABC Not successful, index returned
- ABD Page was connected
- ABE Page was already disconnected
- ABF Index returned, page is addressable
- ABG Index returned, page is connected
- ABH Index not returned, page is disconnected
- ABI Index not returned, address is invalid
- ABJ Page was already addressable
- ABK Page was addressable
- ABL Page was already connected
- ABM Save valid
- ABN Save invalid

RR Format			
Operation Code	Name	Mnemonic	Remarks
00			
01			
02			
03			
04	Set Program Mask	SPM	
05	Branch and Link	BALR	
06	Branch on Count	BCTR	
07	Branch on Condition	BCR	
08	Set Storage Key	SSK	
09	Insert Storage Key	ISK	
0A	Supervisor Call	SVC	
0B			
0C			
0E	Move Long	MVCL	
0F	Compare Logical Long	CLCL	
10	Load Positive	LPR	
11	Load Negative	LNR	
12	Load and Test	LTR	
13	Load Complement	LCR	
14	AND	NR	
15	Compare Logical	CLR	
16	OR	OR	
17	Exclusive OR	XR	
18	Load	LR	
19	Compare	CR	
1A	Add	AR	
1B	Subtract	SR	
1C	Multiply	MR	
1D	Divide	DR	
1E	Add Logical	ALR	
1F	Subtract Logical	SLR	
20	Load Positive (Long)	LPDR	
21	Load Negative (Long)	LNDR	
22	Load and Test (Long)	LTDR	
23	Load Complement (Long)	LCDR	
24	Halve (Long)	HDR	
25	Load Rounded (Extended to Long)	LRDR	
26	Multiply (Extended)	MXR	
27	Multiply (Long to Extended)	MXDR	
28	Load (Long)	LDR	
29	Compare (Long)	CDR	
2A	Add Normalized (Long)	ADR	
2B	Subtract Normalized	SDR	
2C	Multiply (Long)	MDR	
2D	Divide (Long)	DDR	
2E	Add Unnormalized (Long)	AWR	
2F	Subtract Unnormalized (Long)	SWR	
30	Load Positive (Short)	LPER	
31	Load Negative (Short)	LNER	
32	Load and Test (Short)	LTER	
33	Load Complement (Short)	LCER	
34	Halve (Short)	HER	
35	Load Rounded (Long or Short)	LRER	
36	Add Normalized (Extended)	AXR	
37	Subtract Normalized (Extended)	SXR	
38	Load (Short)	LER	

RR Format			
Operation Code	Name	Mnemonic	Remarks
39	Compare (Short)	CER	
3A	Add Normalized (Short)	AER	
3B	Subtract Normalized (Short)	SER	
3C	Multiply (Short)	MER	
3D	Divide (Short)	DER	
3E	Add Unnormalized (Short)	AUR	
3F	Subtract Unnormalized (Short)	SUR	
RX Format			
40	Store Halfword	STH	
41	Load Address	LA	
42	Store Character	STC	
43	Insert Character	IC	
44	Execute	EX	
45	Branch and Link	BAL	
46	Branch on Count	BCT	
47	Branch on Condition	BC	
48	Load Halfword	LH	
49	Compare Halfword	CH	
4A	Add Halfword	AH	
4B	Subtract Halfword	SH	
4C	Multiply Halfword	MH	
4E	Convert to Decimal	CVD	
4F	Convert to Binary	CVB	
50	Store	ST	
51			
52			
53			
54	AND	N	
55	Compare Logical	CL	
56	OR	O	
57	Exclusive OR	X	
58	Load	L	
59	Compare	C	
5A	Add	A	
5B	Subtract	S	
5C	Multiply	M	
5D	Divide	D	
5E	Add Logical	AL	
5F	Subtract Logical	SL	
60	Store (Long)	STD	
61			
62			
63			
64			
65			
66			
67	Multiply (Long to Extended)	MXD	
68	Load (Long)	LD	
69	Compare (Long)	CD	
6A	Add Normalized (Long)	AD	
6B	Subtract Normalized (Long)	SD	
6C	Multiply (Long)	MD	
6D	Divide (Long)	DD	
6E	Add Unnormalized (Long)	AW	
6F	Subtract Unnormalized (Long)	SW	

RX Format			
Operation Code	**Name**	**Mnemonic**	**Remarks**
70	Store (Short)	STE	
71			
72			
73			
74			
75			
76			
77			
78	Load (Short)	LE	
79	Compare (Short)	CE	
7A	Add Normalized (Short)	AE	
7B	Subtract Normalized (Short)	SE	
7C	Multiply (Short)	ME	
7D	Divide (Short)	DE	
7E	Add Unnormalized (Short)	AU	
7F	Subtract Unnormalized (Short)	SU	
RS,SI, and S Format			
80	Set System Mask	SSM	
81			
82	Load PSW	LPSW	
83	Diagnose		
84	Write Direct	WRD	
85	Read Direct	RDD	
86	Branch on Index High	BXH	
87	Branch on Index Low or Equal	BXLE	
88	Shift Right Single Logical	SRL	
89	Shift Left Single Logical	SLL	
8A	Shift Right Single	SRA	
8B	Shift Left Single	SLA	
8C	Shift Right Double Logical	SRDL	
8D	Shift Left Double Logical	SLDL	
8E	Shift Right Double	SRDA	
8F	Shift Left Double	SLDA	
90	Store Multiple	STM	
91	Test under Mask	TM	
92	Move (Immediate)	MVI	
93	Test and Set	TS	
94	AND (Immediate)	NI	
95	Compare Logical (Immediate)	CLI	
96	OR (Immediate)	OI	
97	Exclusive OR (Immediate)	XI	
98	Load Multiple	LM	
99			
9A			
9B			
9C	Start I/O, Start I/O Fast Release	SIO,SIOF	See Note 2
9D	Test I/O	TIO	
9E	Halt I/O, Halt Device	HIO,HDV	See Note 1
9F	Test Channel	TCH	
A0			
A1			
A2			
A3			
A4			
A5			
A6			

RS,SI, and S Format			
Operation Code	Name	Mnemonic	Remarks
A7			
A8			
A9			
AA			
AB			
AC	Store Then AND System Mask	STNSM	
AD	Store Then OR System Mask	STOSM	
AE			
AF			
B0	Connect Page	CTP	
B1	Load Real Address	LRA	
B2	(First byte of two-byte operation codes)		
B3			
B4	Insert Page Bits	IPB	
B5	Set Page Bits	SPB	
B6	Store Control	STCTL	
B7	Load Control	LCTL	
B8	Load Frame Index	LFI	
B9			
BA			
BB			
BC			
BD	Compare Logical Characters under Mask	CLM	
BE	Store Characters under Mask	STCM	
BF	Insert Characters under Mask	ICM	
SS Format			
C0			
C1			
C2			
C3			
C4			
C5			
C6			
C7			
C8			
C9			
CA			
CB			
CC			
CD			
CE			
CF			
D0			
D1	Move Numerics	MVN	
D2	Move (Characters)	MVC	
D3	Move Zones	MVZ	
D4	AND (Characters)	NC	
D5	Compare Logical (Characters)	CLC	
D6	OR (Characters)	OC	
D7	Exclusive OR (Characters)	XC	
D8	Retrieve Status and Page	RSP	
D9			
DA			
DB			
DC	Translate	TR	

SS Format			
Operation Code	Name	Mnemonic	Remarks
DD	Translate and Test	TRT	
DE	Edit	ED	
DF	Edit and Mark	EDMK	
E0			
E1			
E2			
E3			
E4			
E5			
E6			
E7			
W8			
E9			
EA			
EB			
EC			
ED			
EE			
EF			
F0	Shift and Round Decimal	SRP	
F1	Move with Offset	MVO	
F2	Pack	PACK	
F3	Unpack	UNPK	
F4			
F5			
F6			
F7			
F8	Zero and Add Decimal	ZAP	
F9	Compare Decimal	CP	
FA	Add Decimal	AP	
FB	Subtract Decimal	SP	
FC	Multiply Decimal	MP	
FD	Divide Decimal	DP	
FE			
FF			

NOTES

1. Under the System/370 architecture, the machine operations for Halt Device and Halt I/O are as follows:

1001 1110 XXXX XXX0	Halt I/O	HIO
1001 1110 XXXX XXX1	Halt Device	HDV

(X denotes an ignored bit position)

2. Under the System/370 architecture, the machine operations for Start I/O and Start I/O Fast Release are as follows:

1001 1100 XXXX XXX0 Start I/O SIO

1001 1100 XXXX XXX1 Start I/O Fast Release SIOF

(X denotes an ignored bit position)

Operation Code	Name	Mnemonic
AE	Signal Processor	SIGP
BA	Compare and Swap	CS
BB	Compare Double and Swap	CDS
9D01	Clear I/O	CLRIO
B202	Store CPU ID	STIDP
B203	Store Channel ID	STIDC
B204	Set Clock	SCK
B205	Store Clock	STCK
B206	Set Clock Comparator	SCKC
B207	Store Clock Comparator	STCKC
B208	Set CPU Timer	SPT
B209	Store CPU Timer	STPT
B20A	Set PSW Key from Address	SPKA
B20B	Insert PSW Key	IPK
B20D	Purge Translation Lookaside Buffer	PTLB
B210	Set Prefix	SPX
B211	Store Prefix	STPX
B212	Store CPU Address	STAP
B213	Reset Reference Bit	RRB
B215	Clear Page	CLRP
B21B	Deconfigure Page	DEP
B21C	Disconnect Page	DCTP
B21D	Make Addressable	MAD
B21E	Make Unaddressable	MUN
B21F	Store Capacity Counts	STCAP

Appendix V: Assembler Instructions

Operation		Name Entry	Operand Entry
ACTR		A sequence symbol or blank	A SETA expression
AGO		A sequence symbol or blank	A sequence symbol
AIF		A sequence symbol or blank	A logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP		A sequence symbol or blank	Not required
CCW		Any symbol or blank	Four operands, separated by commas
CNOP		Any symbol or blank	Two absolute expressions, separated by a comma
COM	OS only	Any symbol or blank	Not required
	DOS only	Must be blank	Not required
COPY		Must not be present	One ordinary symbol
CSECT		Any symbol or blank	Not required
CXD (OS only)		**Any symbol or blank**	**Not required**
DC		Any symbol or blank	One or more operands, separated by commas
DROP		A sequence symbol or blank	One to sixteen absolute expressions, separated by commas; or blank
DS		Any symbol or blank	One or more operands, separated by commas
DSECT		Any symbol or blank	Not required
DXD (OS only)		**Any symbol**	**One or more operands, separated by commas**
EJECT		A sequence symbol or blank	Not required
END		A sequence symbol or blank	A relocatable expression or blank
ENTRY		A sequence symbol or blank	One or more relocatable symbols, separated by commas

Operation	Name Entry	Operand Entry
EQU	An ordinary symbol or a variable symbol	One to three operands, separated by commas **DOS Only one operand**
EXTRN	A sequence symbol or blank	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[2]
GBLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
GBLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
ICTL	Must not be present	One to three decimal values, separated by commas
ISEQ	Must not be present	Two decimal values, separated by commas
LCLA	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLB	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LCLC	Must not be present	One or more variable symbols that are to be used as SET symbols, separated by commas[2]
LTORG	Any symbol or blank	Not required
MACRO[1]	Must not be present	Not required
MEND[1]	A sequence symbol or blank	Not required
MEXIT[1]	A sequence symbol or blank	Not required
MNOTE	A sequence symbol or blank	A severity code followed by a comma (this much is optional) followed by any combination of characters enclosed in apostrophes

[1]Can be used only as part of a macro definition.
[2]SET symbols can be defined as subscripted SET symbols.

Operation	Name Entry	Operand Entry
OPSYN	An ordinary symbol	A machine instruction mnemonic code, an extended mnemonic code, a macro operation, an assembler operation, an operation code defined by a previous OPSYN instruction, or blank
or OPSYN	A machine instruction mnemonic code, an extended mnemonic code, an assembler operation, an operation code defined by a previous OPSYN instruction	Blank
ORG	OS only: Any symbol or blank DOS only: A sequence symbol or blank	A relocatable expression or blank A relocatable expression or blank
OS only POP	A sequence symbol or blank	One or more operands, separated by a comma
PRINT	A sequence symbol or blank	One to three operands
PUNCH	A sequence symbol or blank	One to eighty characters, enclosed in apostrophes
OS only PUSH	A sequence symbol or blank	One or more operands, separated by a comma
REPRO	A sequence symbol or blank	Not required
SETA	A SETA symbol	An arithmetic expression
SETB	A SETB symbol	A 0 or a 1, a SETB symbol, or a logical expression enclosed in parentheses
SETC	A SETC symbol	A type attribute, a character expression, a substring notation, or a concatenation of character expressions and substring notations. OS only: A duplication factor (a SETA expression enclosed in parentheses) can precede the above if desired.
SPACE	A sequence symbol or blank	A decimal self-defining term or blank
START	Any symbol or blank	A self-defining term or blank
TITLE	A string of alphameric characters. A variable symbol. A combination of the above. A sequence symbol. A blank	One to 100 characters, enclosed in apostrophes

Operation	Name Entry	Operand Entry
USING	A sequence symbol or blank	An absolute or relocatable expression followed by 1 to 16 absolute expressions, separated by commas
WXTRN	A sequence symbol or blank	One or more relocatable symbols, separated by commas

Instruction	Name Entry	Operand Entry
Model Statements[3]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or blank	Any combination of characters (including variable symbols)
Prototype Statement[2]	A symbolic parameter or blank	Zero or more operands that are symbolic parameters, separated by commas
Macro-Instruction Statement[2]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol,[2] or blank	Zero or more positional operands and/or zero or more keyword operands separated by commas[2]
Assembler Language Statement[3]	An ordinary symbol, a variable symbol, a sequence symbol, a combination of variable symbols and other characters that is equivalent to a symbol, or blank	Any combination of characters (including variable symbols)

[1] Can only be used as part of a macro definition.

[2] Variable symbols appearing in a macro instruction are replaced by their values before the macro instruction is processed.

[3] Restrictions on the use of variable symbols in statement fields are included in the descriptions for each individual statement and in "Rules for Model Statement Fields" (See J4B).

Appendix VI: Summary of Constants

TYPE	IMPLICIT LENGTH (BYTES)	ALIGN-MENT	LENGTH MODI-FIER RANGE	SPECIFIED BY	NUMBER OF CON-STANTS PER OPERAND	RANGE FOR EX-PONENTS	RANGE FOR SCALE	TRUN-CATION/ PADDING SIDE
C	as needed	byte	.1 to 256 (1)	characters	one			right
X	as needed	byte	.1 to 256 (1)	hexadecimal digits	multi-ple			left
B	as needed	byte	.1 to 256	binary digits	multi-ple			left
F	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 to +346	left (3)
H	2	half word	.1 to 8	decimal digits	multi-ple	-85 to +75	-187 +346	left (3)
E	4	word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right (3)
D	8	double word	.1 to 8	decimal digits	multi-ple	-85 to +75	0-14	right (3)
L	16	double word	.1 to 16	decimal digits	multi-ple	-85 to +75	0-28	right (3)
P	as needed	byte	.1 to 16	decimal digits	multi-ple			left
Z	as needed	byte	.1 to 16	decimal digits	multi-ple			left
A	4	word	.1 to 4 (2)	any expression	multi-ple			left
OS only Q	4	word	1-4	symbol nam-ing a DXD or DSECT	multi-ple			left
V	4	word	3,4	relocatable symbol	multi-ple			left
S	2	half word	2 only	one absolute or relocatab-le expression or two absol-ute express-ions: exp (exp)	multi-ple			
Y	2	half word	.1 to 2 (2)	any expression	multi-ple			left

(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.

This page left blank intentionally.

Appendix VII: Summary of Macro Facility

The four charts in this Appendix summarize the macro facility described in Part IV of this publication.

Chart 1 indicates which macro language elements can be used in the name and operand entries of each statement.

Chart 2 is a summary of the expressions that can be used in macro instruction statements.

Chart 3 is a summary of the attributes that may be used in each expression.

Chart 4 is a summary of the variable symbols that can be used in each expression.

Statement	Variable Symbols													Attributes						Sequence Symbol
		Global SET Symbols			Local SET Symbols			System Variable Symbols												
	Symbolic Parameter	SETA	SETB	SETC	SETA	SETB	SETC	&SYSNDX	&SYSECT	&SYSLIST	&SYSPARM	&SYSDATE	&SYSTIME	Type	Length	Scaling	Integer	Count	Number	
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	Operand	Operand							Name
SETA	Operand[2]	Name Operand	Operand[3]	Operand[9]	Name Operand	Operand[3]	Operand[9]	Operand		Operand[2]	Operand[9]				Operand	Operand	Operand	Operand	Operand	
SETB	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Name Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[6]			Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	
SETC	Operand	Operand[7]	Operand[8]	Name Operand	Operand[7]	Operand[8]	Name Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand						
AIF	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand	Operand[6]	Operand[6]	Operand[4]	Operand[6]	Operand[6]			Operand[4]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Operand[5]	Name Operand
AGO																				Name Operand
ACTR	Operand[2]	Operand	Operand[3]	Operand[2]	Operand	Operand[3]	Operand[2]	Operand		Operand[2]	Operand[2]				Operand	Operand	Operand	Operand	Operand	
ANOP																				Name
MEXIT																				Name
MNOTE	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	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

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 (from 0 through 2,147,483,647) OS
(from 0 through 99,999,999) DOS

Chart 2. Conditional Assembly Expressions

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions
Can contain	• Self-defining terms	• Any combination of characters enclosed in apostrophes	• A 0 or a 1
	• Length, scaling, integer, count, and number attributes		• SETB symbols
	• SETA and SETB symbols[1]	• Any variable symbol enclosed in apostrophes	• Arithmetic relations[1]
	• SETC symbols whose values are a decimal self-defining term[1]	• A concatenation of variable symbols and other characters enclosed in apostrophes	• Character relations[2] OS only • Arithmetic value
	• &SYSPARM if its value is a decimal self-defining term		
	• Symbolic parameters if the corresponding operand is a decimal self-defining term	• A type attribute reference	
	• &SYSLIST (n) if the corresponding operand is a decimal self-defining term		
	• &SYSLIST (n,m) if the corresponding operand is a decimal self-defining term		
	• &SYSNDX		

[1] Values must be:

OS from 0 through 2,147,483,647

DOS from 0 through 99,999,999

[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 the smaller one will always compare less than the larger.

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions
Operations are	+, - (unary and binary), *, and /; parentheses permitted	Concatenation, with a period (.)	AND, OR, and NOT parentheses permitted
Range of values	-2^{31} to $+2^{31}-1$	0 through 255 characters	0 (false) or 1 (true)
May be used in	• SETA operands • Arithmetic relations[1] • Subscripted SET symbols • &SYSLIST subscript(s) • Substring notation • Sublist notation	• SETC operands • Character relations[2]	• SETB operands • AIF operands

[1] An arithmetic relation consists of two arithmetic expressions related by the operators GT, LT, EQ, NE, GE, or LE.

[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 the smaller one will always compare less than the larger.

Chart 3. Attributes

Attribute	Notation	Can be used with:	Can be used only if type attribute is:	Can be used in
Type	T'	Ordinary Symbols defined in open code; symbolic parameters inside macro definitions; &SYSLIST(m), &SYSLIST(m,n), [redacted] SET symbols; &SYSTIME, &SYSPARM, &SYSDATE, &SYSECT, &SYSNDX	(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 [redacted] 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

NOTE: There are definite restrictions in the use of these attributes (see L1B).

Chart 4. Variable Symbols (Part 1 of 2)

Variable Symbol	Declared by:	Initialized, or set to:	Value changed by:	May be used in:
Symbolic[1] 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[1]	The assembler	Macro instruction index	(Constant throughout definition; unique for each macro instruction)	• Arithmetic expressions • Character expressions
&SYSECT[1]	The assembler	Control section in which macro instruction appears	(Constant throughout definition; set by CSECT, DSECT, START, and COM)	• Character expressions
&SYSLIST[1]	The assembler	Not applicable	Not applicable	• N'&SYSLIST in arithmetic expressions
&SYSLIST (n)[1] &SYSLIST (n,m)[1]	The assembler	Corresponding macro instruction operand	(Constant throughout definition)	• Arithmetic expressions if operand is decimal self-defining term
				• Character expressions

[1]Can be used only in macro definitions.

Chart 4. Variable Symbols cont. (Part 2 of 2)

Variable Symbol	Declared by:	Initialized, or set to:	Value changed by:	May be used in:
&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

[1]Can be used only in macro definitions.

This page left blank intentionally.

Glossary

This glossary has three main types of definitions that apply:

- To the assembler language in particular (usually distinguished by reference to the words "assembler", "assembly", etc.)
- To programming in general
- To data processing as a whole

If you do not understand the meaning of a data processing term used in any of the definitions below, refer to the <u>IBM Data Processing Glossary</u>, Order No. GC20-1699.

IBM is grateful to the American National Standards Institute (ANSI) for permission to reprint its definitions from the American National Standard Vocabulary for Information Processing, which was prepared by Subcommittee X3K5 on Terminology and Glossary of American National Standards Committee X3.

ANSI definitions are preceded by an asterisk (*).

★absolute address: A pattern of characters that identifies a unique storage location without further modification.

absolute expression: An assembly-time expression whose value is not affected by program relocation. An absolute expression can represent an absolute address.

absolute term: A term whose value is not affected by relocation.

★address:
1. An identification, as represented by a name, label, or number, for a register, location in storage, or any other data source or destination such as the location of a station in a communication network.
2. Loosely, any part of an instruction that specifies the location of an operand for the instruction. Synonymous with address reference.
3. See absolute address, base address, explicit address, implicit address, symbolic address.

address constant: A value, or an expression representing a value, used in the calculation of storage addresses.

address reference: Same as address (2).

alignment: The positioning of the beginning of a machine instruction, data constant, or area on a proper boundary in virtual storage.

alphabetic character: In assembler programming, the letters A through Z and $, #, @.

★alphameric: Same as alphanumeric.

★alphanumeric: Pertaining to a character set that contains letters, digits, and usually, other characters, such as punctuation marks. Synonymous with alphameric.

★AND: A logic operator having the property that if P is a statement, Q is a statement, R is a statement,..., then the AND of P, Q, R,... is true if all statements are true, false if any statement is false.

arithmetic expression: A conditional assembly expression that is a combination of arithmetic terms, arithmetic operators, and paired parentheses.

arithmetic operator:
1. In assembler programming, an operator that can be used in an absolute or relocatable expression, or in an arithmetic expression to indicate the actions to be performed on the terms in the expression. The arithmetic operators allowed are: +, -, *, /.
2. See binary operator, unary operator.

arithmetic relation: Two arithmetic expressions separated by a relational operator.

★arithmetic shift:
1. A shift that does not affect the sign position.
2. A shift that is equivalent to the multiplication of a number by a positive or negative integral power of the radix.

arithmetic term: A term that can be used only in an arithmeitc expression.

array: In assembler programming, a series of one or more values represented by a SET symbol.

★assemble: To prepare a machine language program from a symbolic language program by substituting absolute operation codes for symbolic operation codes and absolute or relocatable addresses for symbolic addresses.

★assembler: A computer program that assembles.

assembler instruction:
1. An assembler language statement that causes the assembler to perform a specific operation. Unlike the machine instructions, the assembler instructions are not translated into machine language.
2. See also conditional assembly instruction, macro processing instruction.

assembler language: A source language that includes symbolic machine language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. The assembler language also includes statements that represent assembler instructions and macro instructions.

assembly time: The time at which the assembler translates the symbolic machine language statements into their object code form (machine instructions). The assembler also processes the assembler instructions at this time, with the exception of the conditional assembly and macro processing instructions, which it processes at pre-assembly time.

attribute: A characteristic of the data defined in a source module. The assembler assigns the value of an attribute to the symbol or macro instruction operand that represents the data. Synonymous with data attribute.

*base:
1. A number that is multiplied by itself as many times as indicated by an exponent.
2. See floating-point base.

*base address: A given address from which an absolute address is derived by combination with a relative address. NOTE: In assembler programming, the relative address is synonymous with displacement.

base register: A register that contains the base address.

*binary: Pertaining to the number representation system with a radix of two.

*binary digit: In binary notation, either of the characters, 0 or 1.

binary operator: An arithmetic operator having two terms. The binary operators that can be used in absolute or relocatable expressions and arithmetic expressions are: addition (+), subtraction (-), multiplication (*), and division (/). Contrast with unary operator.

*bit: A binary digit.

bit-length modifier: A subfield in the DC assembler instruction that determines the length in bits of the area into which the defined data constant is to be assembled.

bit string: A string of binary digits in which the position of each binary digit is considered as an independent unit.

blank: In assembler programming, the same as space character.

*blank character: Same as space character.

boundary: In assembler programming, a location in storage that marks the beginning of an area into which data is assembled. For example, a fullword boundary is a location in storage whose address is divisible by four. The other boundaries are: doubleword (location divisible by eight), halfword (location divisible by two), and byte (location can be any number). See also alignment.

*branch: Loosely, a conditional jump.

buffer: An area of storage that is temporarily reserved for use in performing an input/output operation, and into which data is read or from which data is written.

*bug: A mistake or malfunction.

byte:
1. A sequence of adjacent binary digits operated upon as a unit and usually shorter than a computer word.
2. The representation of a character; eight binary digits (bits) in System/370.

call;
*1. To transfer control to a specified closed subroutine.
2. See also macro call.

*character:
1. A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes.
2. See blank character, character set, special character.

character expression: A character string enclosed by apostrophes. It can be used only in conditional assembly instructions. The enclosing apostrophes are not part of the value represented. Contrast with quoted string.

character relation: Two character strings separated by a relational operator.

character set:
*1. A set of unique representations called characters, for example, the 26 letters of the English alphabet, 0 and 1 of the Boolean alphabet, the set of signals in the Morse code alphabet, the 128 characters of the ASCII alphabet.
2. In assembler programming, the alphabetic characters A through Z and $, #, @; the digits, 0 through 9; and the special characters + - * / , ( ) = . ' & and the blank character.

*character string: A string consisting solely of characters.

closed subroutine: A subroutine that can be stored at one place and can be linked to one or more calling routines. Contrast with open subroutine.

★code:

1. A set of unambigous rules specifying the way in which data may be represented, for example, the set of correspondences in the standard code for information interchange.
2. In data processing, to represent data or a computer program in a symbolic form that can be accepted by a data processor.
3. To write a routine.
4. See condition code, object code, operation code.

★coding: See symbolic coding.

collating sequence: An ordering assigned to a set of items, such that any two sets in that assigned order can be collated.

★column: A vertical arrangement of characters or other expressions.

comments statement: A statement used to include information that may be helpful in running a job or reviewing an output listing.

★complement:

1. A number that can be derived from a specified number by subtracting it from a second specified number. For example, in radix notation, the second specified number may be given power of the radix or one less than the given power of the radix. The negative of the number is often represented by its complement.
2. See radix complement, twos complement.

complex relocatable expression: A relocatable expression that contains two or more unpaired relocatable terms or an unpaired relocatable term preceded by a minus sign, after all unary operators have been resolved. A complex relocatable expression is not fully evaluated until program fetch time.

★computer program: A series of instructions or statements, in a form acceptable to a computer, prepared in order to achieve a certain result.

★computer word: A sequence of bits or characters treated as a unit and capable of being stored in one computer location.

concatenation character: The period (.) that is used to separate character strings that are to be joined together in conditional assembly processing.

condition code: A code that reflects the result of a previous input/output, arithmetic, or logical operation.

conditional assembly: An assembler facility for altering at pre-assembly time the content and sequence of source statements that are to be assembled.

conditional assembly expression: An expression that the assembler evaluates at pre-assembly time.

conditional assembly instruction: An assembler instruction that performs a conditional assembly operation. Conditional assembly instructions are processed at pre-assembly time. They include: the LCLA, LCLB, LCLC, GBLA, GBLB, and the GBLC declaration instructions; the SETA, SETB, and SETC assignment instructions; the AIF, AGO, ANOP, and ACTR branching instructions.

★conditional jump: A jump that occurs if specified criteria are met.

★constant: See figurative constant.

continuation line: A line of a source statement into which characters are entered when the source statement cannot be contained on the preceding line or lines.

control program: A program that is designed to schedule and supervise the performance of data processing work by a computing system.

control section: That part of a program specified by the programmer to be a relocatable unit, all elements of which are to be loaded into adjoining virtual storage locations. Abbreviated CSECT.

control statement: See linkage editor control statement.

copy: To reproduce data in a new location or other destination, leaving the source data unchanged, although the physical form of the result may differ from that of the source. For example, to copy a deck of cards onto a magnetic tape.

count attribute (K'): An attribute that gives the number of characters that would be required to represent the data as a character string.

★counter:

1. A device such as a register or storage location used to represent the number of occurrences of an event.
2. See instruction counter, location counter.

CPU: Central processing unit.

CSECT: See control section.

data attribute: Same as attribute.

data constant: See figurative constant.

*debug: To detect, locate, and remove mistakes from a routine or malfunctions from a computer.

*decimal: Pertaining to the number representation system with a radix of ten.

declare: To identify the variable symbols to be used by the assembler at pre-assembly time.

*delimiter: A flag that separates and organizes items of data.

*device: See storage device.

*dictionary: See external symbol dictionary.

dimension: The maximum number of values that can be assigned to a SET symbol representing an array.

dimensioned SET symbol: A SET symbol, representing an array, followed by a decimal number enclosed in parentheses. A dimensioned SET symbol must be declared in a global (GBLA, GBLB, or GBLC) or local (LCLA, LCLB, LCLC) declaration instruction.

displacement:

*1. Same as relative address.
2. In assembler programming, the difference in bytes between a symbolic address and a specified base address.

doubleword: A contiguous sequence of bits or characters which comprises two computer words and is capable of being addressed as a unit.
NOTE: In assembler programming, the doubleword has a length of eight bytes and can be aligned on a doubleword boundary (a location whose address is divisible by eight). Contrast with fullword, halfword.

*dummy: Pertaining to the characteristic of having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy control section.

dummy control section: A control section that the assembler can use to format an area of storage without producing any object code. Synonymous with dummy section.

dummy section: Same as dummy control section.

duplication factor: In assembler programming, a value that indicates the number of times that the data specified immediately following the duplication factor is to be generated. For example, the first subfield of a DC or DS instruction is a duplication factor.

*dynamic storage allocation: A storage allocation technique in which the location of computer programs and data is determined by criteria applied at the moment of need.

EBCDIC: Extended binary coded decimal interchange code.

entry name: A name within a control section that defines an entry point and can be referred to by any control section.

*entry point: In a routine, any place to which control can be passed.

entry symbol:

1. An ordinary symbol that represents an entry name (identified by the ENTRY assembler instruction) or control section name (defined by the CSECT or START assembler instruction).
2. See also external symbol.

EQ: (equal to) See relational operator.

*error message: An indication that an error has been detected. Contrast with warning message.

ESD: External symbol dictionary.

excess sixty-four binary notation: In assembler programming, a binary notation in which each exponent of a floating-point number E is represented by the binary equivalent of E plus sixty-four.

execution time: The time at which the machine instructions in object code form are processed by the central processing unit of the computer.

explicit address: An address reference which is specified as two absolute expressions. One expression supplies the value of a base register and the other supplies the value of a displacement. The assembler assembles both values into the object code of a machine instruction.

exponent:
*1. In a floating-point representation, the numeral, of a pair of numerals representing a number, that indicates the power to which the base is raised.
2. See also excess sixty-four binary notation.

exponent modifier: A subfield in the operand of the DC assembler instruction that indicates the power of ten by which a number is to be multiplied before being assembled as a data constant.

expression:
1. One or more operations represented by a combination of terms, and paired parentheses.
2. See absolute expression, arithmetic expression, complex relocatable expression, relocatable expression.
3. See also character expression.

extended binary coded decimal interchange code: A set of 256 characters, each represented by eight bits.

external name: A name that can be referred to by any control section or separately assembled module; that is, a control section name or an entry name in another module.

external reference: A reference to a symbol that is defined as an external name in another module.

external symbol:
1. An ordinary symbol that represents an external reference. An external symbol is identified in a source module by the EXTRN or WXTRN assembler instruction, or by the V-type address constant.
2. Loosely, a symbol contained in the external symbol dictionary.
3. See also entry symbol.

external symbol dictionary: Control information associated with an object or load module which identifies the external symbols in the module. Abbreviated ESD.

EXTRN: External reference.

fetch:
*1. To locate and load a quantity of data from storage.
2. In the Operating System (OS), to obtain load modules from auxiliary storage and load them into virtual storage. See also loader (1).
3. In the Disk Operating System (DOS), to bring a program phase into virtual storage from the core image library for immediate execution.
4. A control program routine that accomplishes (1), (2), or (3). See also loader (2).
5. The name of the system macro instruction (FETCH) used to accomplish (1), (2), or (3).

* figurative constant: A preassigned, fixed, character string with a preassigned, fixed, data name in a particular programming language.
NOTE: In assembler programming, the two types of figurative constant are:
   a. data and address constants defined by the DC assembler instruction.
   b. symbols assigned values by the EQU assembler instruction.

flag:
*1. Any of various types of indicators used for identification. For example, in assembler programming, the paired apostrophes that enclose a character expression of a quoted string.
2. In assembler programming, to indicate the occurrence of an error.

* floating-point base: In floating-point representation, the fixed positive integer that is the base of the power. NOTE: In assembler programming, this base is 16.

fullword: A contiguous sequence of bits or characters which comprises a computer word and is capable of being addressed as a unit.
NOTE: In assembler programming, the fullword has a length of four bytes and can be aligned on a fullword boundary (a location whose address is divisible by four). Contrast with doubleword, halfword.

GE: (greater than or equal to) See relational operator.

generate:
*1. To produce a program by selection of subsets from a set of skeletal coding under the control of parameters.
2. In assembler programming, to produce assembler language statements from the model statements of a macro definition when the definition is called by a macro instruction.

global scope: Pertaining to that part of an assembler program that includes the body of any macro definition called from a source module and the open code portion of the source module. Contrast with local scope.

global variable symbol:
1. A variable symbol that can be used to communicate values between macro definitions and between a macro definition and open code.
2. Contrast with local variable symbol.

GT: (greater than) See relational operator.

★halfword: A contiguous sequence of bits or characters which comprises half a computer word and is capable of being addressed as a unit.
NOTE: In assembler programming, the halfword has a length of two bytes and can be aligned on a halfword boundary (a location whose address is divisible by two). Contrast with doubleword, fullword.

hexadecimal: Pertaining to a number system with a radix of sixteen; valid digits range from 0 through F, where F represents the highest units position (15).

immediate data: Data specified in an SI type machine instruction that represents a value to be assembled into the object code of the machine instruction.

implicit address: An address reference which is specified as one absolute or relocatable expression. An implicit address must be converted into its explicit base-displacement form before it can be assembled into the object code of a machine instruction.

index register:
★1. A register whose content may be added to or subtracted from the operand address prior to or during the execution of a computer instruction.
2. In assembler programming, a register whose content is added to the operand or absolute address derived from a combination of a base address with a displacement.

inner macro instruction: A macro instruction that is specified, that is, nested inside a macro definition. Contrast with outer macro instruction.

★instruction:
1. A statement that specifies an operation and the values or locations of its operands.
2. See assembler instruction, conditional assembly instruction, machine instruction, macro instruction.

★instruction counter: A counter that indicates the location of the next computer instruction to be interpreted.

instruction statement: See instruction (1).

integer attribute (I'): An attribute that indicates the number of digit positions occupied by the integer portion of fixed-point, decimal, and floating-point constants in their object code form.

★interrupt: To stop a process in such a way it can be resumed.

★I/O: An abbreviation for input/output.

★job control statement: A statement in a job that is used in identifying the job or describing its requirements to the operating system.

★jump:
1. A departure from the normal sequence of executing instructions in a computer.
2. See conditional jump.

keyword: In assembler programming, an ordinary symbol containing up to seven characters. A keyword is used to identify a parameter, called a keyword parameter, in a macro prototype statement and the corresponding macro instruction operand.

keyword operand; An operand in a macro instruction that assigns a value to the corresponding keyword parameter declared in the prototype statement of the called macro definition. Keyword operands can be specified in any order, because they identify the corresponding parameter by keyword and not by their position.
NOTE: In assembler programming, the specification of a keyword operand has the format: a keyword followed by an equal sign which, in turn, is followed by the value to be assigned to the keyword parameter.

keyword parameter: A symbolic parameter in which the symbol following the ampersand represents a keyword.
NOTE: In assembler programming, the declaration of keyword parameter has the format: a keyword parameter followed by an equal sign which, in turn, is followed by a standard (default) value.

label:
★1. One or more characters used to identify a statement or an item of data in a computer program.
2. In assembler programming, the entry in the name field of an assembler language statement. The three main types of name entry are:
   a. the ordinary symbol which represents a label at assembly time.
   b. the sequence symbol which represents a label at pre-assembly time and is used as a conditional assembly branching destination.
   c. the variable symbol that represents a pre-assembly time label for conditional assembly processing and from which ordinary symbols can be generated to create assembly-time labels.

★language:
1. A set of representations, conventions, and rules used to convey information.
2. See machine language, object language, source language.

LE: (less than or equal to) See relational operator.

★length: See word length.

length attribute (L'): An attribute that gives the number of bytes to be occupied by the object code for the data represented, such as machine instructions, constants, or areas.

length field: The operand entry or subentry in machine instructions that specifies the number of bytes at a specific address that are affected by the execution of the instruction.

length modifier: A subfield in the operand of the DS or DC assembler instruction that determines the length in bytes of the area to be reserved or of the area into which the data defined is to be assembled.

★level: The degree of subordination in a hierarchy.

library macro definition: A macro definition stored in a program library. The IBM-supplied supervisor and data management macro definitions (such as those called by GET or PUT) are examples of library macro definitions. A library macro definition can be included at the beginning of a source module: it then becomes a source macro definition.

★ linkage: In programming, coding that connects two separately coded routines.

linkage editor: A processing program that prepares the output of language translators for execution. It combines separately produced object or load modules; resolves symbolic cross references among them; replaces, deletes, and adds control sections, and generates overlay structures on request; and produces executable code (a load module) that is ready to be fetched into virtual storage.

linkage editor control statement: An instruction for the linkage editor.

literal: A symbol or a quantity in a source program that is itself data, rather than a reference to data. Contrast with figurative constant.

literal pool: An area in storage into which the assembler assembles the values of the literals specified in a source module.

★ load: In programming, to enter data into storage or working registers.

load module: The output of the linkage editor; a program in a format suitable for loading into virtual storage for execution.

loader:
1. Under the Operating System (OS), a processing program that combines the basic editing and loading functions of the linkage editor and program fetch in one job step. It accepts object modules and load modules created by the linkage editor and generates executable code directly in virtual storage. The loader does not produce load modules for program libraries.
2. Under the Disk Operating System (DOS), a supervisor routine that retrieves program phases from the core image library and loads them into virtual storage.

local scope: Pertaining to that part of an assembler program that is either the body of any macro definition called from a source module or the open code portion of the source module. Contrast with global scope.

local variable symbol:
1. A variable symbol that can be used to communicate values inside a macro definition or in the open code portion of a source module.

2. Contrast with global variable symbol.

★location: Any place in which data may be stored.

location counter: A counter whose value indicates the address of data assembled from a machine instruction or a constant, or the address of an area of reserved storage, relative to the beginning of a control section.

★logic shift: A shift that affects all positions.

logical expression: A conditional assembly expression that is combination of logical terms, logical operators, and paired parentheses.

logical operator: In assembler programming, an operator or pair of operators that can be used in a logical expression to indicate the action to be performed on the terms in the expression. The logical operators allowed are: AND, OR, NOT, AND NOT, and OR NOT.

logical relation:
1. A logical term in which two expressions are separated by a relational operator. The relational operators allowed are: EQ, GE, GT, LE, LT, and NE.
2. See arithmetic relation, character relation.

logical term: A term that can be used only in a logical expression.

loop:
★1. A sequence of instructions that is executed repeatedly until a terminal condition prevails.
2. See loop counter.

loop counter: In assembler programming, a counter to prevent excessive looping during conditional assembly processing.

LT: (less than) See relational operator.

★machine code: An operation code that a machine is designed to recognize.

machine instruction:
★1. An instruction that a machine can recognize and execute.
2. In assembler programming, (loosely) the symbolic machine language statements which the assembler translates into machine language instructions.

★machine language: A language that is used directly by a machine.

macro:
1. Loosely, a macro definition.
2. See also macro definition, macro generation, macro instruction, macro prototype statement.

macro call: Same as macro instruction.

macro definition: A set of assembler language statements that defines the name of, format of, and conditions for generating a sequence of assembler language statements from a single source statement.

★macro expansion: Same as macro generation.

macro generation: An operation in which the assembler produces a sequence of assembler language statements by processing a macro definition called by a macro instruction. Macro generation takes place at pre-assembly time. Synonymous with macro expansion.

macro instruction:
1. An instruction in a source language that is equivalent to a specified sequence of machine instructions.
2. In assembler programming, an assembler language statement that causes the assembler to process a predefined set of statements (called a macro definition). The statements normally produced from the macro definition replace the macro instruction in the source program. Synonymous with macro call.

macro instruction operand: An operand that supplies a value to be assigned to the corresponding symbolic parameter of the macro definition called by the macro instruction. This value is passed into the macro definition to be used in its processing.

macro library: See program library.

macro processing instruction: An assembler instruction that is used inside macro definitions and processed at pre-assembly time. These instructions are: MACRO, MEND, MEXIT, and MNOTE.

macro prototype: Same as macro prototype statement.

macro prototype statement: An assembler language statement that is used to give a name to a macro definition and to provide a model (prototype) for the macro instruction that is to call the macro definition.

main storage:

★ 1. The general purpose storage of a computer. Usually, main storage can be accessed directly by the operating registers.
2. See also real storage, virtual storage.

★ mask: A pattern of characters that is used to control the retention or elimination of portions of another pattern of characters.

mnemonic operation code: An operation code consisting of mnemonic symbols that indicate the nature of the operation to be performed, the type of data used, or the format of the instruction performing the operation.

mnemonic symbol:

★ 1. A symbol chosen to assist the human memory, for example, an abbreviation such as "mpy" for "multiply".
2. See also mnemonic operation code.

model statement: A statement in the body of a macro definition or in open code from which an assembler language statement can be generated at pre-assembly time. Values can be substituted at one or more points in a model statement; one or more identical or different statements can be generated from the same model statement under the control of a conditional assembly loop.

★ module:

1. A program unit that is discrete and identifiable with respect to compiling, combining with other units, and loading, for example, the input to, or output from, an assembler, compiler, linkage editor, or executive routine.
2. See load module, object module, source module.

name:

1. A 1- to 8-character alphameric term that identifies a data set, a control statement, an instruction statement, a program, or a cataloged procedure. The first character of the name must be alphabetic.
2. See entry name, external name.
3. See also name entry, label.

name entry: Usually synonymous with label (2). However, the name entry of a model statement can be any string of characters at pre-assembly time.

name field parameter: A symbolic parameter that is declared in the name field of a macro prototype statement. It is assigned a value from the entry in the name field of the macro instruction that corresponds to the macro prototype statement.

NE: (not equal to) See relational operator.

★ nest: To imbed subroutines or data in other subroutines or data at a different hierarchical level such that the different levels of routines or data can be executed or accessed recursively.

nesting level: In assembler programming, the level at which a term (or subexpression) appears in an expression, or the level at which a macro definition containing an inner macro instruction is processed by the assembler.

★ no OP: An instruction that specifically instructs the computer to do nothing, except to proceed to the next instruction in sequence.

★ NOT: A logic operator having the property that if P is a statement, then the NOT of P is true if P is false, false if P is true.

★ null character: A control character that serves to accomplish media fill or time fill, for example, in ASCII the all zeros character (not numeric zero). Null characters may be inserted into or removed from a sequence of characters without affecting the meaning of the sequence, but control of equipment or the format may be affected. Abbreviated NUL. Contrast with space character.

null character string: Same as null string.

null string:

★ 1. The notion of a string depleted of its entities, or the notion of a string prior to establishing its entities.
2. In assembler programming, synonymous with the null character string.

number attribute (N'):

1. An attribute of a symbolic parameter that gives the number of sublist entries in the corresponding macro instruction operand.
2. An attribute that gives the number of positional operands in a macro instruction (specified as N'&SYSLIST) or an attribute that gives the number of sublist entries in a specific positional operand (specified as N'&SYSLIST(n)).

★ object code: Output from an assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

★ object language: The language to which a statement is translated. The machine language for the IBM System/370 is an object language.

★ object module: A module that is the output of an assembler or compiler and is input to a linkage editor.

★ object program: A fully compiled or assembled program that is ready to be loaded into the computer. Contrast with source program.

open code: That portion of a source module that lies outside of and after any source macro definitions that may be specified.

open subroutine: A subroutine that is inserted into a routine at each place it is used. Contrast with closed subroutine. NOTE: In assembler programming, a macro definition is an open subroutine, because the statements generated from the definition are inserted into the source module at the point of call.

★ operand:
1. That which is operated upon.
2. See keyword operand, positional operand.

★ operating system: Software which controls the execution of computer programs and which may provide scheduling, debugging, input/output control, accounting, compilation. storage assignment, data management, and related services.

★ operation code: A code that represents specific operations.

★ operator:
1. In the description of a process, that which indicates the action to be performed on the operands. NOTE: In assembler programming, operands are referred to as terms.
2. See arithmetic operator, binary operator, logical operator, unary operator.
3. See also concatenation character.

★ OR: A logic operator having the property that if P is a statement, Q is a statement, R is a statement,..., then the OR of P, Q, R... is true if at least one statement is true, false if all statements are false.

ordinary symbol: A symbol that represents an assembly-time value when used in the name or operand field of an instruction in the assembler language. Ordinary symbols are also used to represent operation codes for assembler language instructions. An ordinary symbol has one alphabetic character followed by zero to seven alphameric characters.

outer macro instruction: A macro instruction that is specified in open code. Contrast with inner macro instruction.

★ overflow: That portion of the result of an operation that exceeds the capacity of the intended unit of storage.

★ overlay: The technique of repeatedly using the same blocks of internal storage during different stages of a program. When one routine is no longer needed in storage, another routine can replace all part of it.

★ padding: A technique used to fill a block with dummy data.

paired parentheses: A left parenthesis and a right parenthesis that belong to the same level of nesting in an expression; the left parenthesis must appear before its matching right parenthesis. If parentheses are nested within paired parentheses, the nested parentheses must be paired.

paired relocatable terms: Two relocatable terms in an expression with the same relocatability attribute that have different signs after all unary operations have been performed. Paired relocatable terms have an absolute value.

★ parameter:
1. A variable that is given a constant value for a specific purpose or process.
2. See keyword parameter, name field parameter, positional parameter, symbolic parameter.

point of substitution: Any place in an assembler language statement, particularly a model statement, into which values can be substituted at pre-assembly time. Variable symbols represent points of substitution.

positional operand: An operand in a macro instruction that assigns a value to the corresponding positional parameter declared in the prototype statement of the called macro definition.

positional parameter: A symbolic parameter that occupies a fixed position relative to the other positional parameters declared in the same macro prototype statement.

pre-assembly time: The time at which the assembler process macro definitions and performs conditional assembly operations.

private code: An unnamed control section.

★ program:
1. A series of actions proposed in order to achieve a certain result.
2. Loosely, a routine.
3. To design, write, and test a program as in (1).
4. Loosely, to write a routine.
5. See computer program, object program, source program.

program fetch time:
1. The time at which a program (in the form of load modules or phases) is loaded into virtual storage for execution.
2. See also fetch (2), fetch (3).

★ program library: A collection of available computer programs and routines.

programmer macro definition: Loosely, a source macro definition.

prototype statement: Same as macro prototype statement.

★ pushdown list: A list that is constructed and maintained so that the next item to be retrieved and removed is the most recently stored item in the list, that is, last in, first out. Synonymous with pushdown stack.

pushdown stack: Same as pushdown list.

quoted string: A character string enclosed by apostrophes that is used in a macro instruction operand to represent a value that can include blanks. The enclosing apostrophes are part of the value represented. Contrast with character expression.

★ radix: In positional representation, that integer, if it exists, by which the significance of the digit place must be multiplied to give the significance of the next higher digit place. For example, in decimal notation, the radix of each place is ten.

★ radix complement: A complement obtained by subtracting each digit from one less than its radix, then adding one to the least significant digit, executing all carries required. For example, tens complement in decimal notation, twos complement in binary notation.

read-only: A type of access to data that allows it to read but not modified.

real storage: The storage of a IBM System/370 computer from which the central processing unit can directly obtain instructions and data and to which it can directly return results. Real storage can occupy all or part of main storage. Contrast with virtual storage.

recursive: Pertaining to a process in which each step makes use of the results of earlier steps.
NOTE: In assembler programming, the inner macro instruction that calls the macro definition within which it is nested performs a recursive call.

★ register:
1. A device capable of storing a specified amount of data such as one word.
2. See base register, index register.

relation: The comparison of two expressions to see if the value of one is equal to, less than, or greater than the value of the other.

relational operator: An operator that can be used in an arithmetic or character relation to indicate the comparison to be performed between the terms in the relation. The relational operators are: EQ (equal), GE (greater than or equal to), GT (greater than), LE (less to or equal to), LT (less than), NE (not equal to).

★ relative address: The number that specifies the difference between the absolute address and the base address. Synonymous with displacement.

relocatability attribute: An attribute that identifies the control section to which a relocatable expression belongs. Two relocatable expressions have the same relocatability attribute if the unpaired term in each of them belongs to the same control section.

relocatable expression: An assembly-time expression whose value is affected by program relocation. A relocatable expression can represent a relocatable address.

relocatable term: A term whose value is affected by program relocation.

★ relocate: In computer programming, to move a routine from one portion of storage to another and to adjust the necessary address references so that the routine, in its new location, can be executed.

relocation: The modification of address constants to compensate for a change in origin of a module, program, or control section.

★ rounding: Same as roundoff.

roundoff: To delete the least significant digit or digits of a numeral and to adjust the part retained in accordance with some rule.

★ routine:
1. An ordered set of instructions that may have some general or frequent use.
2. See subroutine.

scale modifier: A subfield in the operand of the DC assembler instruction that indicates the number of digits in the object code to be occupied by the fractional portion of a fixed-point or floating-point constant.

scaling attribute: An attribute that indicates the number of digit positions occupied by the fractional portion of fixed-point, decimal, and floating-point constants in their object code form.

scope:
1. In assembler programming, that part of a source program in which a variable symbol can communicate its value.
2. See global scope, local scope.

self-defining term: An absolute term whose value is implicit in the specification of the term itself.

sequence symbol: A symbol used as a branching label for conditional assembly instructions. It consists of a period followed by one to seven alphameric characters, the first of which must be alphabetic.

SET symbol: A variable symbol used to communicate values during conditional assembly processing. It must be declared to have either a global or local scope.

severity code: A code assigned by the assembler to an error detected in a source module. A severity code can also be specified and assigned to an error message generated by the MNOTE instruction.

★ sign bit: A binary digit occupying the sign position.

sign position: A position, normally located at one end of a numeral, that contains an indication of the algebraic sign of the number.

★ significant digit: A digit that is needed for a certain purpose, particularly one that must be kept to preserve a specific accuracy or precision.

★ source language: The language from which a statement is translated.

source macro definition: A macro definition included in a source module. A source macro definition can be entered into a program library; it then becomes a library macro definition.

source module: A sequence of statements in the assembler language that constitutes the input to a single execution of the assembler.

★ source program: A computer program written in a source language. Contrast with object program.

★ space character: A normally nonprinting graphic character used to separate words. Synonymous with blank character. Contrast with null character.

★ special character: A graphic character that is neither a letter, nor a digit, nor a space character.

★ statement:
1. In computer programming, a meaningful expression or generalized instruction in a source language.
2. See job control statement, linkage editor control statement, comments statement, model statement.

★ storage:
1. Pertaining to a device into which data can be entered, in which they can be held, and from which they can be retrieved at a later time.
2. Loosely, any device that can store data.
3. See main storage, real storage, virtual storage.

★ storage allocation:
1. The assignment of blocks of data to specified blocks of storage.
2. See dynamic storage allocation.

★ <u>storage protection</u>: An arrangement for preventing access to storage for either reading, or writing, or both.

<u>storage stack</u>: Loosely, a pushdown list.

★ <u>string</u>:
1. A linear sequence of entities such as characters or physical elements.
2. See bit string, character string, null string.

<u>sublist</u>: A macro instruction operand that contains one or more entries separated by commas and enclosed in parentheses.

★ <u>subroutine</u>:
1. A routine that can be part of another routine.
2. See closed subroutine, open subroutine.

<u>subscript</u>: One or more elements, enclosed in parentheses, that appear immediately after a variable symbol or character expression. The value of a subscript indicates a position in the array or string of values represented by the variable symbol or character expression.

<u>subscripted &SYSLIST</u>: The system variable symbol &SYSLIST immediately followed by either one subscript or two subscripts separated by commas, and enclosed in parentheses. The value of the first subscript indicates the position of a positional operand in a macro instruction and the value of the second subscript indicates the position of the entry in the sublist of the positional operand indicated by the first subscript.

<u>subscripted SET symbol</u>:
1. A SET symbol that is immediately followed by a subscript. A subscripted SET symbol must be declared with an allowable dimension before it can be used. The value of the subscript indicates the position of the value given to the subscripted symbol in the array represented by the SET symbol.
2. See also dimensioned SET symbol.

<u>subscripted symbolic parameter</u>: A symbolic parameter that is immediately followed by a subscript. The value of the subscript indicates the position of the entry in the sublist in the macro instruction operand referred to by the symbolic parameter.

<u>substitution</u>: The action taken by the assembler when it replaces a variable symbol with a value, for example, during the expansion of a macro definition.

<u>substring</u>:
1. A character string that has been extracted from a character expression.
2. See also substring notation.

<u>substring notation</u>: A character expression immediately followed by two subscripts, separated by a comma, and enclosed in parentheses. It can be used only in conditional assembly instructions. The value of the first subscript indicates the position of the character within the character expression that begins the substring. The value of the second subscript represents the number of characters to be extracted from the character expression.

★ <u>switch</u>: A device or programming technique for making a selection, for example, a conditional jump.

★ <u>symbol</u>:
1. A representation of something by reason of relationship, association, or convention.
2. See mnemonic symbol, ordinary symbol, sequence symbol, SET symbol, variable symbol.

★ <u>symbolic address</u>: An address expressed in symbols convenient to the computer programmer.

★ <u>symbolic coding</u>: Coding that uses machine instructions with symbolic addresses. NOTE: In assembler programming, any instruction can contain symbolic addresses. In addition, any other portion of an instruction may be represented with symbols, for example, labels, registers, lengths and immediate data.

<u>symbolic parameter</u>:
1. A variable symbol declared in the prototype statement of a macro definition. A symbolic parameter is usually assigned a value from the corresponding operand in the macro instruction that calls the macro definition.
2. See also keyword parameter, name field parameter, positional parameter.

<u>system loader</u>: See loader (2).

<u>system macro definition</u>: Loosely, a library macro definition supplied by IBM.

<u>system macro instruction</u>: Loosely, a macro instruction that calls for the processing of an IBM-supplied library macro definition, for example, the ATTACH macro.

<u>system variable symbol</u>: A variable symbol that always begins with the characters

&SYS. The system variable symbols do not have to be declared, because the assembler assigns them read-only values automatically according to specific rules.

term:
1. The smallest part of an expression that can be assigned a value.
2. See absolute term, arithmetic term, logical term, relocatable term.

*translate: To transform statements from one language to another without significantly changing the meaning.

*truncate: To terminate a computational process in accordance with some rule, for example, to end the evaluation of a power series at a specified term.
NOTE: In assembler programming, the object code for data constants can be truncated by the assembler.

*twos complement: The radix complement in binary notation.

type attribute (T'): An attribute that distinguishes one form of data from another, for example, fixed-point constants from floating-point constants or machine instructions from macro instructions.

unary operator: An arithmetic operator having only one term. The unary operators that can be used in absolute or relocatable, and arithmetic expressions are: positive (+) and negative (-).

unnamed control section: A control section that is initiated in one of the following three ways:
1. By an unnamed START instruction.
2. By an unnamed CSECT instruction, if no unnamed START instruction appears before the CSECT instruction.
3. By any instruction that affects the setting of the location counter.

* variable: A quantity that can assume any of a given set of values.

variable symbol: In assembler programming, a symbol, used in macro and conditional assembly processing, that can assume any of a given set of values. It consists of an ampersand (&) followed by one to seven alphameric characters, the first of which must be alphabetic.
NOTE: All variable symbols must be declared except the system variable symbols.

virtual storage: Address space appearing to the user as real storage from which instructions and data are mapped into real storage locations. The size of virtual storage is limited only by the addressing scheme of the computing system rather than by the actual number of real storage locations. Contrast with real storage.

warning message: An indication that a possible error has been detected. The assembler does not assign a severity code to this type of error. Contrast with error message.

word:
* 1. A character string or bit string considered as an entity.
* 2. See computer word.
3. See doubleword, fullword, halfword.

* word length: A measure of the size of a word, usually specified in units such as characters or binary digits.
NOTE: In assembler programming, the word, or fullword, contains 32 bits (binary digits) or 4 bytes.

wrap-around: Loosely, the overflow of the location counter when the value assigned to it exceeds $2^{24}-1$

This page left blank intentionally.

Index

B

C

D

E

F

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

This page left blank intentionally.

GC33-4010-5

OS/VS—DOS/VS—VM/370 Assembler Language (File No. S370-21) (OS/VS, DOS/VS, VM/370) Printed in U.S.A. GC33-4010-5

IBM®

IBM

Technical Newsletter

This Newsletter No. GN20-9372
Date 28 December 1981

Base Publication No. GC33-4010-5
File No. S370-21

Prerequisite Newsletters None

OS/VS-DOS/VSE-VM/370 Assembler Language

This technical newsletter, a part of Release 4 of OS/VS1, Release 3 of OS/VS2, Release 2 of VM/370, and DOS/VSE, provides replacement pages for the subject publication. These replacement pages remain in effect for any subsequent releases unless specifically altered. Pages to be inserted and/or removed are:

- cover, ii
- vii, viii
- xi, xii
- 135
- 139, 140
- 167-170
- 189, 190
- 223, 224
- 279, 280
- 285, 286
- 291, 292
- 297, 298
- 305, 306
- 327-330
- 333, 334
- 391-408
- 447, 448

A change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

This technical newsletter contains maintenance changes.

Note: Please file this cover letter at the back of the manual to provide a record of changes.

IBM Corporation, P.O. Box 50020, Programming Publishing, San Jose, California 95150

Printed in U.S.A.