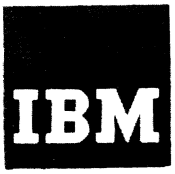
File No. S360-21
GC28-2000-5



Systems Reference Library

IBM Time Sharing System

Assembler Language

This publication describes the IBM Time Sharing System Assembler Language, a symbolic programming language. The language provides a convenient means for representing machine instructions and related data, especially as related to the Time Sharing System (TSS). The TSS Assembler Program processes the language and provides auxiliary functions for preparing and documenting a program; the Assembler includes facilities for processing the assembler macro language.

This manual is intended for applications programmers and any users who code in the assembler language.

Sixth Edition (April 1976)

This is a revision of, but does not make obsolete, the previous edition, GC28-2000-4. Editorial changes have been made throughout this publication. Also, much of the now outdated reference data that appeared in the Appendices has been deleted.

This edition is current with Release 2.0 of the IBM Time Sharing System/370 (TSS/370), and remains in effect for all subsequent versions or modifications of TSS unless otherwise noted. Significant changes or additions to this publication will be provided in new editions or Technical Newsletters.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form is provided at the back of this publication for reader's comments. If the form has been removed, comments may be addressed to IBM Corporation, Time Sharing System - - Dept 80M, 1133 Westchester Avenue, White Plains, New York 10604.

©Copyright International Business Machines Corporation 1966, 1969, 1970, 1971, 1976.

HOW TO USE THIS BOOK

This publication explains for the Time Sharing System user the instructions which direct the Assembler language program in converting source statements into object language. It also contains the syntactical rules you must follow in coding your program in Assembler language. Only assembler instructions are fully described in this book; the functions of machine-executable instructions (such as Load, Add Register, or Move Character) are explained in Principles of Operation, GA22-6821.

Since some of the terms used in this book are defined in earlier sections of the book, we suggest an initial read-through; after that, you will probably continue to rely on this publication as a reference source. This book is not a primer, and other Time Sharing System publications are directed more towards illustrating specific possibilities; this publication serves as the definitive grammar of the Assembler language.

The reader is presumed to have some knowledge of programming concepts as well as some understanding of machine operations, particularly storage addressing, data formats, and machine instruction formats and functions.

THIS BOOK CONTAINS

- The rules of the language: statement format and syntax, assembler limitations.
- Assembler (not machine-executable) instructions, such as DC, DS, ORG, CSECT, ENTRY. How and when to use them.
- The meaning of words defined specially for Assembler language, such as "term", "literal", "relocatable expression", "symbolic parameter".
- Macro language. How to use it and the rules for writing a macro definition.
- Appendixes of Assembler language characteristics.

SEE THESE BOOKS FOR

- A description of machine-executable instructions. Standard instructions may be found in the IBM Principles of Operation, GA22-6821.
- How to use TSS to assemble a program you've written. The Time Sharing System Assembler Programmer's Guide, GC28-2032, tells how and has several examples.
- System macro instructions available to you. These are described fully in Time Sharing System Assembler User Macro Instructions, GC28-2004.
- Special macro-writing techniques. Certain macro techniques that you may find useful are described in the System Programmer's Guide, GC28-2008.

Time Sharing System Concepts and Facilities, GC28-2003, is recommended as preliminary reading. The TSS Quick Guide, GX28-6400, provides a pocket-size summary of some of the tables in this book.

CONTENTS

SECTION 1: INTRODUCTION 1
Assembler Language 1
Machine Operation Codes 1
Assembler Operation Codes 1
Macro Instructions 1
Assembler Program 1
Virtual Storage Concept 1
Basic Functions 2
Programmer Aids 2
Operating System Relationships 3

SECTION 2: GENERAL INFORMATION 4
Assembler Language Coding Conventions 4
Input Sources 4
Punched Card Coding Form 4
Statement Boundaries -- Card Format 4
Continuation Lines -- Card Format 5
Character Sets -- Card Format 5
Statement Boundaries -- Keyboard Format 5
Continuation Lines -- Keyboard Format 5
Character Sets -- Keyboard Format 5
Statement Format 5
Identification Sequence Field 6
Caution When Changing Card-Origin Statements 6
Summary of Statement Format 7
Character Set 7
Assembler Language Structure 7
Terms And Expressions 9
Terms 9
Symbols 9
Self-Defining Terms 9
Location Counter Reference 11
Literals 11
Symbol Length Attribute Reference 12
Terms in Parentheses 12
Expressions 13
Evaluation of Expressions 13
Absolute and Relocatable Expressions 13

SECTION 3: ADDRESSING -- PROGRAM SECTIONING AND LINKING 15
Addressing 15
Addresses -- Explicit and Implied 15
Base Register Instructions 15
USING -- Use Base Address Register 15
DROP -- Drop Base Register 16
Programming With The USING Instruction 16
Relative Addressing 17
Program Sectioning and Linking 17
Control Sections 18
Control Section Location Assignment 18
First Control Section of Program 18
START -- Start Assembly 18
CSECT -- Identify Control Section 19
Unnamed Control Section 19
DSECT -- Identify Dummy Section 20
COM -- Define Common Control Sections 21
PSECT -- Define Prototype Control Section 21
External Dummy Sections 22
Attributes of Control Sections 22
Symbolic Linkages 22
ENTRY -- Identify Entry Point Symbol 23
EXTRN -- Identify External Symbol 23

Addressing External Control Sections 24

SECTION 4: MACHINE INSTRUCTIONS 25
Machine Instruction Statements 25
Instruction Alignment and Checking 25
Operand Fields and Subfields 25
Lengths -- Explicit and Implied 26
Machine Instruction Mnemonic Codes 27
Machine Instruction Examples 27
RR Format 27
RX Format 27
RS Format 28
SI Format 28
SS Format 28
Extended Mnemonic Codes 28

SECTION 5: ASSEMBLER INSTRUCTION STATEMENTS 31
Symbol Definition Instruction 31
EQU -- Equate Symbol 31
Data Definition Instructions 32
DC -- Define Constant 32
Operand Subfield 1: Duplication Factor 33
Operand Subfield 2: Type 33
Operand Subfield 3: Modifiers 34
Operand Subfield 4: Constant 36
DS -- Define Storage 43
Special Uses of the Duplication Factor 44
DXD -- Define External Dummy Section 44
CXD -- Cumulative Length External Dummy Section 45
CCW -- Define Channel Command Word 46
Listing Control Instructions 46
TITLE -- Identify Assembly Output 46
EJECT -- Start New Page 47
SPACE -- Space Listing 47
PRINT -- Print Optional Data 47
Program Control Instructions 48
ICTL -- Input Format Control 48
ISEQ -- Input Sequence Checking 49
PUNCH -- Punch a Card 49
REPRO -- Reproduce Following Card 49
ORG -- Set Location Counter 49
LTORG -- Begin Literal Pool 49
Special Addressing Consideration 50
Duplicate Literals 50
CNOP -- Conditional No Operation 50
COPY -- Copy Predefined Source Coding 51
END -- End Assembly 51

SECTION 6: INTRODUCTION TO MACRO LANGUAGE 52
Macro Instruction Statement 52
Macro Definition 52
Sources Of Macro Definitions 53
System Macro Instructions 53
Varying The Generated Statements 53
Variable Symbols 53
Types of Variable Symbols 53
Assigning Values to Variable Symbols 53
Global SET Symbols 53

SECTION 7: HOW TO PREPARE MACRO DEFINITIONS 54
MACRO -- Macro Definition Header 54
MEND -- Macro Definition Trailer 54
Macro Instruction Prototype 54
Statement Format 54
Model Statements 55
Free Apostrophes 55
Symbolic Parameters 56

Concatenating Symbolic Parameters With Other Characters or Other Symbolic Parameters . . . 57
Comments Statements . . . 58
Copy Statements . . . 58

SECTION 8: HOW TO WRITE MACRO INSTRUCTIONS . . . 59
Macro Instruction Operands . . . 59
Statement Format . . . 60
Omitted Operands . . . 60
Operand Sublists . . . 60
Inner Macro Instructions . . . 61
Levels Of Macro Instructions . . . 62

SECTION 9: HOW TO WRITE CONDITIONAL ASSEMBLY INSTRUCTIONS . . . 63
Set Symbols . . . 63
Defining SET Symbols . . . 63
Using Variable Symbols . . . 63
Attributes . . . 64
Type Attribute (T') . . . 64
Length (L') Scaling (S'), and Integer (I') Attributes . . . 65
Count Attribute (K') . . . 66
Number Attribute (N') . . . 66
Assigning Attributes to Symbols . . . 66
Sequence Symbols . . . 67
LCLA, LCLB, LCLC -- Define SET Symbols . . . 67
SETA -- SET Arithmetic . . . 68
Evaluation of Arithmetic Expressions . . . 68
Using SETA Symbols . . . 68
SETC -- SET Character . . . 69
Type Attribute . . . 70
Character Expression . . . 70
Substring Notation . . . 70
Using SETC Symbols . . . 71
SETB -- SET Binary . . . 72
Evaluation of Logical Expressions . . . 73
Using SETB Symbols . . . 73
AIF -- Conditional Branch . . . 73
AGO -- Unconditional Branch . . . 74
ANOP -- Assembly No Operation . . . 75
Conditional Assembly Elements . . . 75

SECTION 10: EXTENDED FEATURES OF MACRO LANGUAGE . . . 77
MEXIT -- Macro Definition Exit . . . 77
MNOTE -- Request for Error Message . . . 77
Global and Local Variable Symbols . . . 78
Defining Local and Global SET Symbols . . . 79
Using Global and Local SET Symbols . . . 79
Subscripted SET Symbols . . . 81
System Variable Symbols . . . 82
&SYSNDX -- Macro Instruction Index . . . 82
&SYSECT -- Current Control Section . . . 82
&SYSSTYP -- Current Control Section Type . . . 83
&SYSLIST -- Macro Instruction Operand . . . 84
&SYSPSCT -- Prototype Control Section Name . . . 84
&SYSDATE and &SYSTIME -- Date/Time Variables . . . 85
Keyword Macro Definitions and Instructions . . . 85
Keyword Prototype . . . 86
Keyword Macro Instruction . . . 86
Mixed-Mode Macro Definitions and Instructions . . . 87
Mixed-Mode Prototype . . . 88
Mixed-Mode Macro Instruction . . . 88
Macro Definition Compatibility . . . 88

APPENDIX A: ASSEMBLER INSTRUCTIONS 89

APPENDIX B: MACHINE INSTRUCTION FORMAT 92
Notes for Appendix B: 93

APPENDIX C: SUMMARY OF CONSTANTS 94

APPENDIX D: MACRO LANGUAGE SUMMARY 95

APPENDIX E: SAMPLE ASSEMBLY 101

INDEX 105

ILLUSTRATIONS

Figure 1. Coding Form 4
Figure 2. Assembler Language Structure -- Machine and Assembler 8
Figure 3. Multiple Base Register Assignment 17
Figure 4. Extended Mnemonic Codes (RX format) 29
Figure 5. Extended Mnemonic Codes (RR format) 30
Figure 6. Type Codes for Constants 34
Figure 7. Bit-Length Specification (Single Constant) 35
Figure 8. Bit-Length Specification (Multiple Constants) 35
Figure 9. Bit-Length Specification (Multiple Operands) 35
Figure 10. CNOP Alignment 51
Figure 11. Conditional Assembly Elements 76

Table 1. Details of Address Specification 26
Table 2. Details of Length Specification in SS Instructions . . 27
Table 3. Channel Command Word 46

Chart 1. Statements (Part 1 of 2) 95
Chart 2. Macro Language Elements 97
Chart 3. Expressions 98
Chart 4. Attributes 99
Chart 5. Variable Symbols 99

Computer programs may be expressed in machine language (language directly interpreted by the computer) or in a symbolic language, which is more meaningful to the programmer. The symbolic language must be translated into machine language by an associated processing program before the computer can execute the program.

Among symbolic programming languages, assembler languages are closest to machine language in form and content.

The assembler language discussed in this manual is a symbolic programming language for the IBM Time Sharing System (TSS). It enables the programmer to use all machine functions as if he were coding in machine language.

The assembler program translates symbolic instructions into machine language instructions, assigns storage locations, and performs auxiliary functions necessary to produce an executable machine language program.

ASSEMBLER LANGUAGE

The basis of the assembler language is a collection of mnemonic symbols which represent:

1. Machine language operation codes.

2. Operations (auxiliary functions) to be performed by the assembler program.

The language is augmented by other symbols, supplied by the programmer and used to represent storage addresses or data. Symbols are easier to remember and to code than their machine language equivalents. Use of symbols greatly reduces programming effort and error.

The programmer may also create a type of instruction called macro instructions, for which mnemonic symbols, supplied by the programmer, serve as operation codes.

Machine Operation Codes

The assembler language provides mnemonic machine instruction operation codes for all machine instructions in the Universal Instruction Set and provides extended mnemonic operation codes for the conditional branch instruction.

Assembler Operation Codes

The assembler language also contains mnemonic assembler instruction operation codes that specify auxiliary functions to be performed by the assembler program. These instructions to the assembler program, with a few exceptions, do not result in the generation of any machine language code by the assembler program.

Macro Instructions

The assembler language enables the programmer to define and use macro instructions that are represented by operation codes specifying sequences of machine and/or assembler instructions that accomplish the desired function.

Macro instructions used in preparing an assembler language source program are either: (a) system macro instructions, provided by IBM, that relate the object program to components of the operating system, or (b) macro instructions created by the programmer specifically for the program at hand or for incorporation in a library.

Programmer-created macro instructions simplify program writing and ensure that a standard sequence of instructions is used to accomplish a desired function.

For instance, the logic of a program may require repetitive execution of the same instruction sequence. Rather than code the entire sequence every time it is needed, the programmer creates a macro instruction that represents the sequence. Then whenever the sequence is needed, the programmer codes the macro instruction statement. During assembly the corresponding sequence of instructions is inserted in the object program.

The language and procedures for defining and using macro instructions are discussed in Part Two.

ASSEMBLER PROGRAM

The assembler program, or assembler, processes source statements that are written in the assembler language.

Virtual Storage Concept

TSS permits the concept of a "virtual storage", whose size is the maximum

addressing capability of the computer system.

The 24-bit addressing capability permits 12-bit base addressing (4096 base addresses) and 12-bit byte addressing (4096 byte addresses) -- a maximum of over 16 million addressable bytes. In programming, however, the user is usually restricted to addresses that represent physical storage on his machine. Such a program cannot address 16 million contiguous bytes directly, but must be structured as a series of overlays. For the Model 67, which has been modified for 32-bit addressing, the capacity of virtual storage is over four billion bytes.

The time sharing monitor assigns active programs to whatever physical storage is available. Automatic relocation techniques are used to distribute the logical program, as written by the programmer, into physical locations that are consistent with efficient operation of the system. These relocation techniques need not concern the programmer. He may write his program as if contiguous bytes of storage were available for each assembly.

Basic Functions

Processing involves translating source statements into machine language, assigning virtual storage locations to instructions and other elements of the program, and performing the auxiliary assembler program functions designated by the programmer. The output of the assembler program is the object program, a machine language equivalent of the source program. Upon the PRINT request of the programmer, the assembler program furnishes a printed listing of the source- and object-program statements, and presents additional information, such as error indications, that may be useful to him in analyzing his program. (Nonconversational users may arrange to have listings printed directly without the PRINT command by so specifying in the ASM command.) The object program is in the format required by the linkage and loading components of Time Sharing System.

The assembler uses virtual storage to allocate its working storage. Thus, language elements that are customarily limited by the capacity of internal tables are, in effect, limited only by the size of virtual storage available to the TSS Assembler.

PROGRAMMER AIDS

The auxiliary functions of the assembler program assist the programmer in checking and documenting programs, controlling address assignment, segmenting programs, defining data and symbols, generating macro instructions, and controlling the assembly program itself. Mnemonic operation codes for these functions are provided in the language.

Variety in Data Representation: The programmer selects decimal, binary, hexadecimal, or character representations of machine language binary values that best suit his purposes in writing source statements.

Base Register Address Calculation: The addressing scheme requires designation of a base register (containing a base address value) and a displacement value to specify a storage location (discussed in IBM Principles of Operation). The assembler assumes the clerical burden of calculating relative virtual storage addresses in these terms for the symbolic addresses used by the programmer. The programmer retains control of base register usage and the values entered therein.

Relocatability: The TSS assembler assigns virtual storage locations to a program. Physical storage locations are assigned to virtual storage components, when the program is executed, by a combination of linkage programming and automatic relocation features.

Sectioning and Linking: The assembler language and program provide facilities for partitioning an assembly into one or more parts, called control sections. Special control sections provide facilities for reenterable programs and the "common" data feature for FORTRAN.

The assembler allows symbols to be defined in one assembly and referred to in another, thus effecting a link between separately assembled programs. This permits reference to data and/or transfer of control between programs. Sectioning and linking is discussed in Section 3, under "Program Sectioning and Linking."

Program Listings: A listing of source program statements and resulting object program statements may be produced by the assembler for each source program. The programmer can, to some extent, control the form and content of the listing.

Conversational users will have selected listings automatically stored in a list data set which may be later obtained on an output device by issuing the PRINT command.

The user may, however, override this default by requesting that listings be printed out immediately at his terminal. Nonconversational users may choose to have listings put either in a list data set or immediately onto an output device. (Refer to IBM Time Sharing System Command System User's Guide, GC28-2001 for a full explanation of the listing data set maintenance process.)

Error Indications: As a source program is assembled, it is analyzed for actual or potential errors in the use of the assembler language. Detected errors are indicated in the program listing or in conversational mode during the actual assembly run.

OPERATING SYSTEM RELATIONSHIPS

The assembler, a component of TSS, functions under control of the operating system that provides the assembler with input/output, supplementary macro library, system macro library, and other services needed in assembling a source program. Similarly, the object program produced by the assembler will operate under control of TSS and will depend on it for input/output and other services. In writing the source program, the programmer must include statements requesting the desired functions from TSS. These statements are discussed in IBM Time Sharing System Assembler User Macro Instructions.

The assembler will create the proper linkage between the object program and the specified service components of the operating system. More specific information on operating system relationships is in Concepts and Facilities.

This section presents information about assembler language coding conventions, assembler source statement structure, addressing, and sectioning and linking of programs.

ASSEMBLER LANGUAGE CODING CONVENTIONS

Input Sources

A source program is a sequence of source statements that have either been punched into cards and entered by card reader, or typed at the keyboard of a remote terminal device; the statement formats differ slightly between the two sources. The card format is identical to that in other IBM assembler languages; the keyboard format was designed for use at typewriter-like terminal devices.

Punched Card Coding Form

Assembler language source statements may be written on the standard coding form, X28-6509 (Figure 1), provided by IBM. One line of coding on the form is punched into one card; vertical columns on the form correspond to card columns.

Space is provided for program identification and instructions to keypunch operators. The body of the form is composed of the statement field, columns 1-71, and the identification sequence field, columns 73-80. The identification sequence field, not part of a statement, is discussed under "Statement Format," below.

The entries (coding) that compose a statement occupy columns 1-71 of a statement line and, if needed, columns 16-71 of up to two successive continuation lines.

Statement Boundaries -- Card Format

Source statements are normally in columns 1-71 of statement lines and columns 16-71 of continuation lines. Therefore, columns 1, 71, and 16 are referred to as the "begin," "end," and "continue" columns. (This convention may be altered by use of the input format control (ICTL) assembler

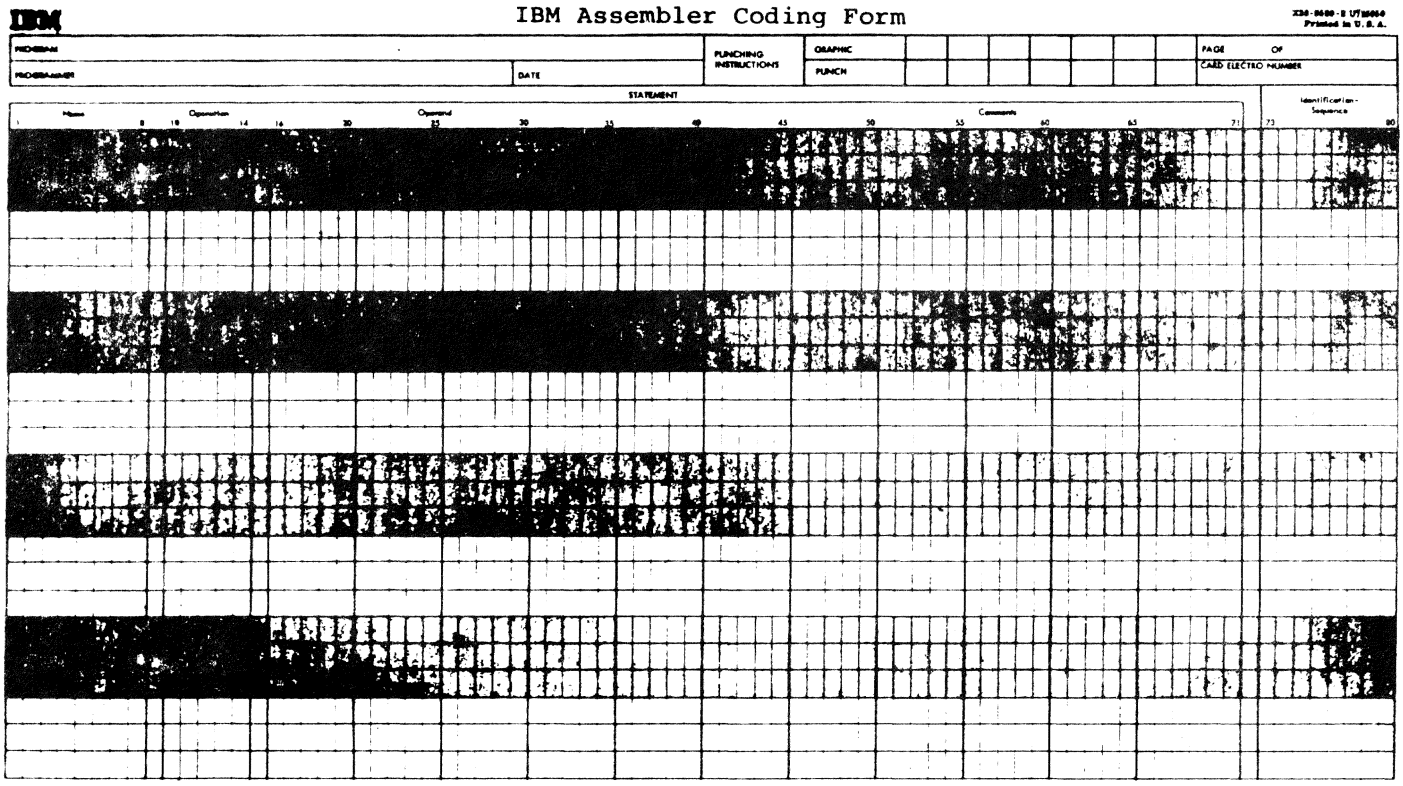


Figure 1. Coding Form

instruction, which will be discussed later.) The continuation character, if used, always immediately follows the end column.

Continuation Lines -- Card Format

When it is necessary to continue a statement on another line, these rules apply.

1. Enter a continuation character (not blank, and not part of the statement coding) in the column following the end column (normally column 72) of the statement line.

2. Continue the statement on the next line, starting in the continue column (normally column 16); all columns to the left of the continue column must be blank.

3. When more than one continuation line is needed, each line to be continued must have a character (not blank, and not part of the statement coding) entered in the column following the end column (normally column 72).

4. Only two continuation lines may be used for a normal statement. A macro instruction, however, may use as many as necessary.

Character Sets -- Card Format

Source statements may be entered into TSS from punched cards in three ways: from the installation's high-speed card reader, from an IBM 2780 Data Transmission Terminal located away from the central installation (remote job entry), or from an IBM 1056 Card Reader attached to a 1052 Printer-Keyboard.

Cards read into the central or remote job entry card readers must be in 029 keypunch code, which is converted to EBCDIC. Cards read into the 1056 Card Reader may be in either 1057 card punch code or 029 punch code.

To initiate punched card input from the 1056 Card Reader, the user must type in CA, CB, or C on the keyboard. CA transfers control from the keyboard to the reader and specifies conversion from 1057 card punch code to EBCDIC. CB transfers control to the reader and specifies conversion from 029 punch code to EBCDIC. C transfers control to the reader using the following convention: if keyboard mode was KA, CA will be the new mode; if KB was the keyboard mode, CB will be the new mode. If some cards were punched on the 1057 and others on the 029, the commands CA and CB may be inserted at any place in the deck where it is necessary to change the mode. (Further information on the use of C, CA, and CB will be found in Command System User's Guide and Terminal User's Guide.)

Statement Boundaries -- Keyboard Format

Source statements occupy the area between: (a) the column at which the command language interpreter releases the keyboard to the user and (b) the right-hand margin setting. This area is not restricted to 80 columns.

Continuation Lines -- Keyboard Format

When it is necessary to continue a statement that is being entered from a keyboard, the continuation character is typed at the point at which continuation is desired, followed immediately by a carrier return. (For example, on the IBM 1052 Printer-Keyboard, the continuation character is a hyphen.) The statement is continued at the first non-blank, non-tab character of the next line.

Character Sets -- Keyboard Format

KA and KB can be used to specify the character set to be used during keyboard input. With KA, the user indicates he wishes to use the full EBCDIC character set. With KB, the user specifies that the lower case characters (a-z and ! " ←) be translated into their upper case equivalents (A-Z and $ # @).

Statement Format

Statements may consist of one to four entries. They must be written in this order, left to right: name entry, operation entry, operand entry, and comments entry. These entries must be separated by one or more blanks (or a horizontal tab character from the keyboard).

For punched card input, the coding form provides an 8-character name field, a 5-character operation field, and a 56-character operand and/or comments field.

If the programmer wants to disregard these boundaries and write the name, operation, operand, and comment entries in other positions, he is subject to these rules:

1. On punched cards, the entries must not extend beyond statement boundaries (either the conventional boundaries, or as designated by the programmer via the ICTL instruction).

2. Entries must be in proper sequence.

3. Entries must be separated by one or more blanks.

4. A name entry must start in the begin column.

5. Name and operation entries must be completed in the first line of the statement, including at least one blank following the operation entry.

In the descriptions of the entries, below, the assembler regards a horizontal tab character in keyboard format as a single blank.

Name Entry: This symbol, usually optional, is created by the programmer to identify or label a statement. The symbol must consist of eight characters or less; the first character must be entered in the begin column. If the begin column is blank, the assembler assumes no name has been entered; no blanks may appear within the name entry.

Operation Entry: The mnemonic operation code specifies the machine operation, assembler, or macro instruction operation desired. An operation entry is mandatory and must start at least one position to the right of the begin column; it cannot appear in a continuation line. Valid mnemonic operation codes for machine and assembler operations, given in Appendixes B and C, consist of five characters or less for machine or assembler operation codes, and eight characters or less for macro instruction operation codes. No blanks may appear within the operation entry.

Operand Entry: This entry identifies and describes data (such as storage locations, masks, storage area lengths, or types of data) associated with the instruction.

Operands are required for all machine instructions and, depending on the needs of the instruction, one or more operands may be written. Operands must be separated by commas. No blanks may appear between operands and the commas that separate them. The operands may not contain embedded blanks except when character representation specifies a constant, a literal, or immediate data in an operand; e.g., C'AB D'.

Comments Entry: These are information items about the program that are to be inserted in the program listing. All valid characters (see "Character Set," below), including blanks, may be used in writing a comment. The entry may follow the operand entry and must be separated from it by a blank; comments entries cannot extend beyond the end column (normally column 71).

An entire line, or a series of lines, may be used for comments, by placing an asterisk in the begin column of each line. Also, continuation lines, described above, may be used.

In statements where an optional operand entry is omitted, or in statements which allow no operand but in which a comments entry is desired, the absence of the operand entry must be indicated by a comma preceded and followed by one or more blanks.

Name	Operation	Operand
	END	, COMMENT

Statement Example: A compare instruction is named by the symbol COMP; the operation entry (CR) is the mnemonic operation code for a register-to-register compare operation; and the two operands (5,6) designate two general registers whose contents are to be compared. The comments entry reminds the programmer that he is comparing "new sum" to "old."

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

Identification Sequence Field

For source statements that originate from punched cards, the identification sequence field of the coding form (columns 73-80) is used to enter the optional program identification and/or statement sequence characters. If the field, or a portion of it, is used, the program identification is punched in the statement cards and reproduced in the printed listing of the source program.

The programmer may number the cards in this field, to keep source statements in order, by punching appropriate characters into the cards. During assembly, he may request the assembler to verify this sequence by using the input sequence checking (ISEQ) assembler instruction (see Section 5, "Program Control Instructions").

Caution When Changing Card-Origin Statements

Source statements from punched cards may later be changed, using various commands of the TSS Text Editor (the Text Editor commands are described in Command System User's Guide).

On assembly, each source statement of punched card origin is treated as an 80-character record. Where the statement has been shortened to fewer than 80 characters by changing it with a Text Editor command after it has been stored, the assembler,

before further processing, pads the statement to 80 characters with trailing blanks. Where the statement has been changed to contain more than 80 characters, the assembler truncates the statement to 80 characters.

Care must be taken in changing a card-origin source statement so that, after padding or truncation by the assembler, the statement will still conform to the coding conventions discussed in this section. (An example might be a statement containing a sequence number in the identification sequence field, columns 73-80. The statement is shortened one character during text editing. The assembler pads with one trailing blank in column 80, leaving columns 72-79 containing the sequence number. Since column 72 is normally the continuation column, an error results if the next source statement is not a continuation line.)

Summary of Statement Format

Entries in a statement must always be separated by at least one blank and must be in this order: name, operation, operands, comment.

Every statement requires an operation entry; name and comment entries are optional. Operand entries are required for all machine instructions and most assembler instructions.

The name and operation entries must be completed in the first statement line, including at least one blank following the operation entry.

The name and operation entries must not contain blanks; operand entries must not have blanks preceding or following the commas that separate them.

A name entry must always start in the begin column.

If the column after the end column is blank, the next line must start a new statement. If the column after the end column is not blank, the following line will be treated as a continuation line.

All entries must be contained within the designated begin-, end-, and continue-column boundaries.

Character Set

Source statements use these characters:

Letters	A through Z $ # @
Digits	0 through 9
Special Characters	+ - , = . * ( ) ' / & blank

Any of the remainder of the 256 punch combinations may be designated anywhere that characters may appear between paired apostrophes, and in comments.

ASSEMBLER LANGUAGE STRUCTURE

A source statement is composed of:

- A name entry (usually optional)
- An operation entry (mandatory)
- An operand entry (usually required)

A name entry is:

- A symbol

An operation entry is:

- A mnenomic operation code representing a machine, assembler, or macro instruction.

An operand entry is:

- One or more operands composed of one or more expressions, which, in turn, are composed of a term or an arithmetic combination of terms.

Operands of machine instructions generally represent such things as storage locations, general registers, immediate data, or constant values. Operands of assembler instructions provide the information needed by the assembler program to perform the designated operation.

This structure is shown in Figure 2. Terms shown in the figure are classed as absolute or relocatable, depending on how program relocation affects them. The relocation consideration in the following discussions is the adjustment, by the loader or linkage editor, of the virtual storage assignments that were made by the assembler. This adjustment usually takes the form of a base increment that is added to the original virtual storage location assignments. A term is absolute if its value does not change when such an adjustment is made and is relocatable if its value changes upon relocation.

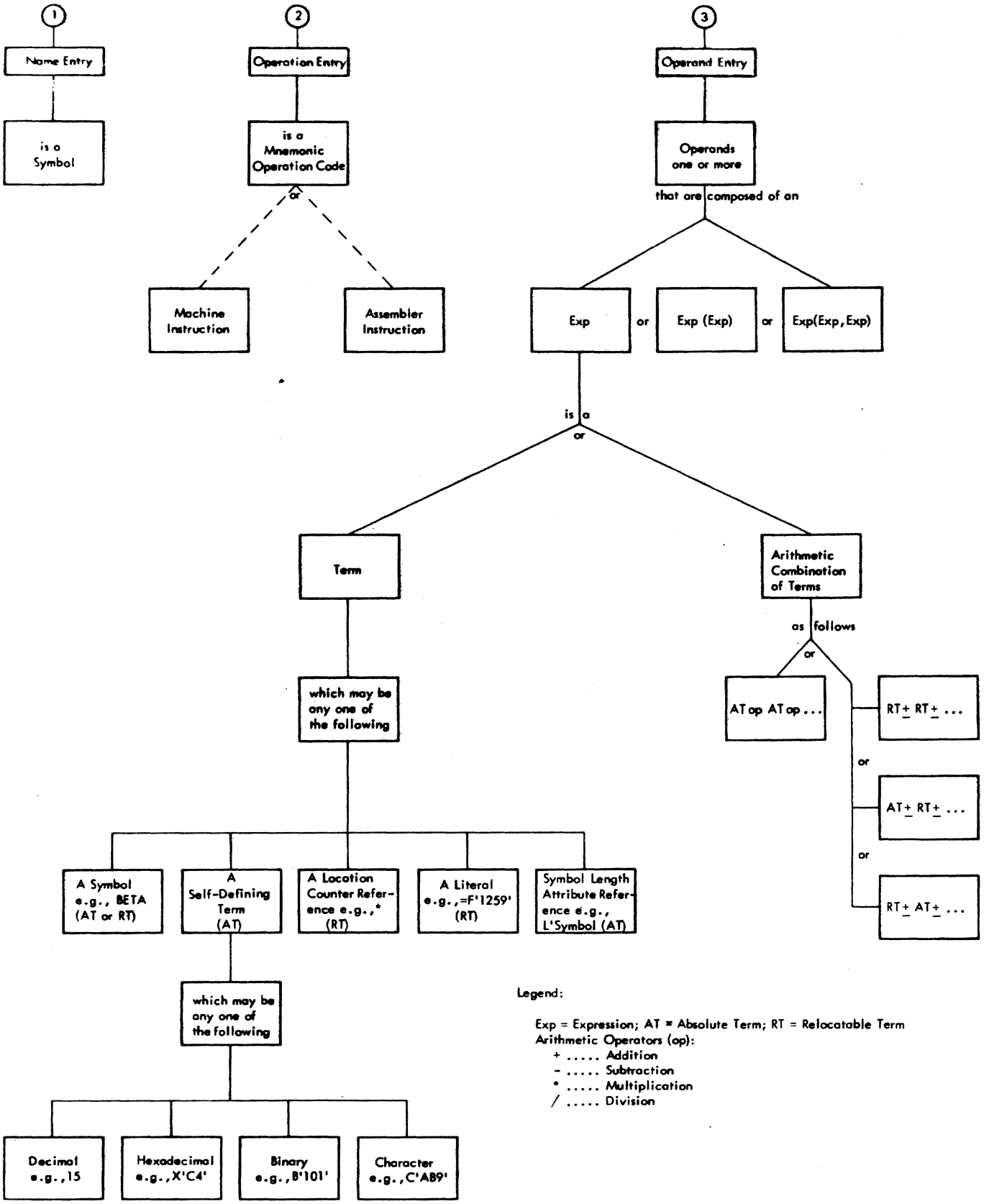


Figure 2. Assembler Language Structure -- Machine and Assembler

TERMS AND EXPRESSIONS

TERMS

Every term represents a value that may be assigned by the assembler program (symbols, symbol length attribute, location counter reference), or that may be inherent in the term itself (self-defining term literal). An arithmetic combination of terms is reduced to a single value by the assembler.

The following material discusses each type of term and the rules for its use.

Symbols

A symbol is a character or combination of characters that represents addresses or arbitrary values.

Symbols, through their use as names and in operands, provide the programmer with a way to name and refer to a program element. A symbol created by the programmer for use as a name entry and in an operand must conform to these rules:

1. The symbol must consist of not more than eight characters. The first character must be a letter, the other characters may be letters, digits, or any combination of the two.

2. No special characters may be included in a symbol.

3. No blanks are allowed in a symbol.

These are valid symbols:

READER	LOOP2	@B4
A23456	N	$A1
X4F2	S4	#56

These symbols are invalid, for the reasons noted:

256B	(first character is not alphabetic)
RECORDAREA2	(more than eight characters)
BCD*34	(contains a special character - *)
IN AREA	(contains a blank)

Defining Symbols: The assembler assigns a value to each symbol appearing as a name entry in a source statement. The values assigned to symbols that name storage areas, instructions, constants, and control sections are the addresses of the leftmost bytes of the storage fields containing the named items. Since the addresses of these items may change with program relocation, the symbols naming them are considered relocatable terms.

A symbol used as a name entry in the equate (EQU) assembler instruction is assigned the value designated in the operand entry of the instruction. Since the operand entry may represent a relocatable value or an absolute (nonchanging) value, the symbol is considered a relocatable, or absolute, term, depending upon the value it is equated to.

The assembled value of a symbol may not be negative and may not exceed $2^{24}-1$ except when using a 32-bit machine where its allocated value may be as high as $2^{32}-1$.

A symbol is defined when it appears as the name of a source statement. (A special case of symbol definition is discussed in Section 3, "Program Sectioning and Linking.")

Symbol definition also involves the assignment of a length attribute to the symbol. (The assembler program maintains an internal symbol table that has the values and attributes of symbols. When the assembler encounters a symbol in an operand, it refers to the table for the values associated with the symbol.) The symbol's length attribute is the size, in bytes, of the storage field whose address is represented by the symbol. For example, a symbol naming an instruction that occupies four bytes of storage has a length attribute of 4.

Previously Defined Symbols: Some instructions require that a symbol appearing in the operand entry be previously defined; that symbol, before its use in an operand, must have appeared as a name entry in a prior statement.

General Restrictions on Symbols: A symbol may be defined only once in an assembly; each symbol used as a statement name must be unique to that assembly. However, a symbol may be used in the name field more than once as a control section name (defined in the START, CSECT, or DSECT assembler statements, described in Section 3) because the coding of a control section may be suspended and then resumed at any subsequent point. The CSECT or DSECT statement that resumes the section must be named by the same symbol that initially named the section; thus, the symbol that names the section must be repeated. Such usage is not considered to be duplication of a symbol definition.

Self-Defining Terms

The value of a self-defining term is inherent in the term; for example, the decimal self-defining term 15 represents a value of fifteen. The value of a self-

defining term is absolute; it does not change when the program is relocated.

The four types of self-defining terms -- decimal, hexadecimal, binary, and character -- are used as the machine language binary value or bit configuration they represent. The type of term selected depends on what is being specified.

Using Self-Defining Terms: These terms, representing machine values or bit configurations, are used to specify program elements, such as immediate data, masks, registers, addresses, and address increments.

Self-defining terms are distinct from data constants or literals. When a self-defining term is used in a machine instruction statement, its value is assembled into the instruction. When a data constant or literal is specified in the operand of an instruction, its address is assembled into the instruction.

Decimal Self-Defining Term: A decimal self-defining term is simply an unsigned decimal number written as a sequence of digits; high-order 0s may be used (e.g., 007). Limitations on the value of a term depend on its use. For example, a decimal term that designates a general register should have a value between 0 and 15 inclusive; one that represents an address should not exceed the size of storage. In any case, the value of a decimal term may not exceed 4,294,967,295 ($2^{32}-1$). A decimal term is assembled as its binary equivalent. Some examples of decimal self-defining terms are 8, 147, 4092, and 00021.

Hexadecimal Self-Defining Term: A hexadecimal self-defining term is an unsigned hexadecimal number written as a sequence of hexadecimal digits. The digits must be enclosed in apostrophes and preceded by the letter X; for example, X'C49'.

Each hexadecimal digit is assembled as its 4-bit binary equivalent. Thus, a hexadecimal term used to represent an 8-bit mask would consist of two hexadecimal digits. The maximum value of a hexadecimal term is X'FFFFFFFF'.

The hexadecimal digits and their bit patterns are:

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

Binary Self-Defining Term: A binary self-defining term is written as an unsigned sequence of 1s and 0s enclosed in apostrophes and preceded by the letter B; for example, B'10001101'. This term would appear in storage as shown, occupying one byte. A binary term may have up to 32 bits. Binary representation is used primarily in designating bit patterns of masks or in logical operations.

The following example illustrates a binary term used as a mask in a test under mask (TM) instruction. The contents of GAMMA are to be tested, bit by bit, against the pattern of bits represented by the binary term.

Name	Operation	Operand
ALPHA	TM	GAMMA,B'10101101'

Character Self-Defining Term: A character self-defining term consists of one to four characters enclosed by apostrophes; it must be preceded by the letter C. All letters, decimal digits, and special characters may be used in a character term. Also, any of the remainder of the 256 punch combinations may be designated in a character self-defining term. Examples of character self-defining terms are:

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

Because apostrophes, in the assembler language, and ampersands, in the macro instruction language, are used as syntactic characters, the following rule must be observed when using these characters in a character term: For each apostrophe or ampersand desired in a character self-defining term, two apostrophes or ampersands must be written. For example, the character value A'# would be written as 'A''#', while an apostrophe followed by a blank and another single apostrophe would be written as ''' '''.

Each character in the character sequence is assembled as its 8-bit code equivalent. The two apostrophes or ampersands that must be used to represent a single apostrophe or ampersand within the character sequence are assembled as a single apostrophe or ampersand.

Location Counter: A location counter is used to assign storage addresses to program statements; it is the assembler's equivalent of the instruction counter in the computer. As each machine instruction or data area is assembled, the location counter is adjusted to the proper boundary for the item, if adjustment is necessary, and then

incremented by the length of the assembled item. Thus, it always points to the next available location. If the statement is named by a symbol, the value attribute of the symbol is the value of the location counter after boundary adjustment, but before addition of the length.

The assembler maintains a location counter for each control section of the program and manipulates each location counter as previously described. Source statements for each section are assigned addresses from the location counter for that section. Within each control section, the location counter controls the assignment of consecutively higher virtual storage locations. Thus, if a program has multiple control sections, all statements identified as belonging to the first control section will be assigned from the location counter for section 1; the statements for the second section from the location counter for section 2; etc. This procedure is followed whether the statements from different control sections are interspersed or written in control section sequence.

The location counter setting can be controlled by using the START and ORG assembler instructions, which are described in Sections 3 and 5. The counter affected by either of these assembler instructions is the counter for the control section in which it appears; the maximum value for the counter is $2^{24}-2$.

Location Counter Reference

The programmer may refer to the current value of the location counter at any place in a program by using an asterisk in an operand. The asterisk represents the location of the first byte of currently available storage (that is, after any required boundary adjustment). Using an asterisk in a machine instruction statement is the same as placing a symbol in the name field of the statement and then using that symbol as an operand of the statement. Because a location counter is maintained for each control section, a location counter reference designates the location counter for the section in which the reference appears.

A reference to the location counter may be made in a literal address constant (that is, the asterisk may be used in an address constant specified in literal form). The address of the instruction containing the literal is used for the value of the location counter. A location counter reference may not be used in a statement that requires the use of a predefined symbol, excepting the EQU and ORG assembler instructions.

Literals

This is a constant preceded by an equal sign (=), one of three basic ways to introduce data into a program.

A literal represents data, rather than a reference to data. The inclusion of a literal in a source statement directs the assembler to: (a) assemble the value specified by the literal, (b) store this value in a "literal pool," and (c) place the address of the storage field containing the value in the operand field of the assembled source statement.

Literals provide a means of entering constants (such as numbers for calculation and addresses, indexing factors, or words or phrases for printing out a message) into a program by specifying the constant in the operand of the instruction in which it is used. This may be done rather than using the DC (define constant, see Section 5) assembler instruction to enter the data into the program, and then using the name of the DC instruction in the operand.

Only one literal is allowed in a machine instruction statement and it may not be combined with any other terms; also, it may not be used as the receiving field of an instruction that modifies storage. A literal may not be specified in an address constant. (However, an address constant may be specified as a literal.)

This instruction shows one use of a literal:

```
┌─────┬─────────┬──────────────────────┐
│Name │Operation│Operand               │
├─────┼─────────┼──────────────────────┤
│GAMMA│L        │10,=F'274'            │
└─────┴─────────┴──────────────────────┘
```

The statement GAMMA is a load instruction using a literal as the second operand. When assembled, the second operand will be the address at which the value represented by F'274' is stored.

In general, literals may be used wherever a storage address is permitted as an operand. They may not, however, be used in any assembler instruction that requires the use of a previously defined symbol. Literals are considered relocatable because the address of the literal, rather than the literal itself, will be assembled in the statement that employs a literal. The assembler generates the literals, collects them, and places them in a specific storage area (explained in "Literal Pool," below).

A literal is not to be confused with the immediate data in an SI instruction; such

data is assembled into the instruction and is not preceded by an equal sign (=).

Literal Format: The assembler requires descriptions of the specified type of literal and of the literal itself. Those descriptions assist the assembler in assembling the literal correctly. The descriptive portion of the literal must indicate the format of the constant, and it may specify the constant's length.

The method of describing and specifying a constant as a literal is nearly identical to the method of specifying it in the operand of a DC assembler instruction. The major difference is that the literal must start with an equal sign (=) to indicate to the assembler that a literal follows. (Specification of a literal is covered in the DC assembler instruction operand format, in Section 5.) The type of literal designated in an instruction is not checked for correspondence with the operation code of the instruction.

Some examples of literals are:

```
=A(BETA) -- address constant literal
=F'1234' -- fixed point, 4-byte number
=C'ABC'  -- character literal
```

Literal Pool: The literals processed by the assembler are collected and placed in a special area called the literal pool; the location of a literal, rather than the literal itself, is assembled in a statement using a literal. The positioning of the literal pool may be controlled by the programmer, if he so desires. Unless otherwise specified, the literal pool is placed at the end of the first control section (CSECT). To facilitate writing reenterable programs, if the assembly contains one or more prototype control sections (PSECTs), literal address constants are placed in a pool at the end of the first prototype control section.

The programmer can also specify the creation of multiple literal pools. However, the sequence in which literals are ordered within the pool is controlled by the assembler. Literals that are eight bytes, or a multiple of eight are aligned at a doubleword boundary; 4-byte literals are aligned at a word boundary; 2-byte literals are aligned at a halfword; and literals with an odd number of bytes are aligned at the next available storage location. Further information on positioning literal pools is in Section 5, under "LTORG -- Begin Literal Pool."

Symbol Length Attribute Reference

The length attribute of a symbol may be used as a term. Reference to the attribute is made by coding L' followed by the symbol; for example, L'BETA. The length attribute of BETA will be substituted for the term. The following example illustrates the use of L'symbol in moving a character constant into either the high-order or low-order end of a storage field.

For ease in following the example, the length attributes of A1 and B2 are mentioned. However, keep in mind that the L'symbol term makes coding such as this possible in situations where lengths are unknown.

```
┌─────┬────────┬──────────────────────┐
|Name |Operation|Operand               |
├─────┼────────┼──────────────────────┤
|A1   |DS      |CL8                   |
|B2   |DC      |CL2'AB'               |
|HIORD|MVC     |A1(L'B2),B2           |
|LOORD|MVC     |A1+L'A1-L'B2(L'B2),B2 |
└─────┴────────┴──────────────────────┘
```

A1 names an 8-byte storage field and is assigned a length attribute of 8. B2 names a 2-byte character constant and is assigned a length attribute of 2. HIORD moves the contents of B2 into the leftmost two bytes of A1; the term L'B2 in parentheses provides the length specification required by the instruction. When the instruction is assembled, the length is placed in the proper field of the machine instruction.

LOORD moves the contents of B2 into the rightmost two bytes of A1. The combination of terms A1+L'A1-L'B2 results in the addition of the length of A1 to the beginning address of A1, and the subtraction of the length of B2 from this value. The result is the address of the seventh byte in field A1. The constant represented by B2 is moved into A1, starting at this address. L'B2 in parentheses provides length specification, as in HIORD.

Terms in Parentheses

These are reduced to a single value, so they, in effect, become a single term. Arithmetically combined terms, enclosed in parentheses, may be used in combination with terms outside the parentheses:

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

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

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

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

The innermost set of terms in parentheses is evaluated first. For compatibility with other IBM assemblers, expressions should be limited to five levels of parentheses; parentheses which occur as part of an operand format are not included in this five-level limit. Programs written expressly for the TSS assembler are not restricted in the number of levels of parentheses that may be used.

EXPRESSIONS

This subsection discusses the two types of expressions, absolute and relocatable, used in coding operand entries for source statements, together with the rules for determining these attributes of an expression.

An expression is composed of a single term or an arithmetic combination of terms, as shown in Figure 2. These are examples of valid expressions:

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

The rules for coding expressions are:

1. An expression may not start with an arithmetic operator, (+-/*). The expression -A+BETA is invalid; 0-A+BETA is valid.

2. An expression may not contain two successive terms or two operators.

3. For compatibility with other IBM assemblers, an expression should not consist of more than 16 terms or contain more than five levels of parentheses. For programs written expressly for the TSS assembler, there is no restriction on the number of terms or levels of parentheses.

4. A multiterm expression may not contain a literal.

Expressions containing more than five levels of parentheses produce warning messages, but are assembled correctly.

Evaluation of Expressions

A single term expression; for example, 29, BETA, *, L'ALPHA, takes on the value of the term involved.

A multiterm expression; for example, BETA+10, ENTRY-EXIT, 25*10+A/B, is reduced to a single value, as follows:

1. Each term is given its value.

2. Arithmetic operations are performed left to right. Multiplication and division are done before addition and subtraction; for example, A+B*C is evaluated as A+(B*C), not (A+B)*C. The computed result is the value of the expression.

3. Division always yields an integer result; for example, 1/2*10 yields 0; 10*1/2 yields 5.

4. Division by 0 is valid and yields a 0 result.

Parenthesized multiterm expressions used in an expression are processed before the rest of the terms in the expression; for example, in A+BETA*(CON-10), the term CON-10 is evaluated first and the resulting value is used in computing the final value of the expression.

Absolute and Relocatable Expressions

An expression is absolute if its value is unaffected by program relocation; it is relocatable if its value changes upon program relocation. The two types of expressions, absolute and relocatable, take on these characteristics from the terms composing them.

Absolute Expression: This may be either an absolute term or any arithmetic combination of absolute terms. An absolute term may be an absolute symbol, any of the self-defining terms, or a length attribute reference. All arithmetic operations are permitted between absolute terms, as indicated in Figure 2.

An absolute expression may contain relocatable terms (RT), alone or in combination with absolute terms (AT), under these conditions:

1. There must be an even number of relocatable terms in the expression.

2. The relocatable terms must be paired. Each pair must have the same relocatability attributes; that is, they appear in the same control section in this assembly (see "Program Sectioning and Linking," in Section 3). Each

pair must consist of terms with opposite signs. The paired terms do not have to be contiguous, for example, RT+AT-RT.

3. No relocatable term may enter into a multiply or divide operation. RT-RT* 10 is invalid; (RT-RT)*10 is valid.

The pairing of relocatable terms (with opposite signs and the same relocatability attribute) cancels the effect of relocation. Therefore, the value represented by the paired terms remains constant, regardless of program relocation. For example, in the absolute expression A-Y+X, A is an absolute term, and X and Y are relocatable with the same relocatability attribute. If A equals 50, Y equals 25, and X equals 10, the value of the expression would be 35. If X and Y are relocated by a factor of 100, their values would then be 125 and 110. However, the expression would still be evaluated as 35(50-125+110=35).

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

```
A-Y+X
A
A*A
X-Y+A
*-Y (A reference to the location counter
     must be paired with another relocat-
     able term from the same control sec-
     tion, that is, with the same reloc-
     atability attribute.)
```

Note that paired relocatable expressions cannot be successfully used in certain macro language statements of this book. In macro language, conditional or branching statements, such as AIF, SETA, and SETB statements, determine which of several desired, pre-stored lines of source code will be included in an assembled program. Since macro expansion (generation of these selected source code statements into a program) takes place prior to the assignment of location counter values, a paired relocatable expression in a conditional macro language statement will not have been resolved into an absolute expression.

Relocatable Expressions: The value of a relocatable expression would change by n if the program in which it appears is relocated n bytes away from its originally assigned area of storage. All relocatable expressions must have a positive value.

A relocatable expression may be a relocatable term and it may contain relocatable terms, alone or in combination with absolute terms, under these conditions:

1. There must be an odd number of relocatable terms.

2. All the relocatable terms but one must be paired (described in "Absolute Expression," above).

3. The unpaired term must not be directly preceded by a minus sign.

4. No relocatable term may enter into a multiply or divide operation.

A relocatable expression reduces to the single relocatable value of the odd relocatable term, adjusted by the values represented by the absolute terms and/or paired relocatable terms associated with that value. The relocatability attribute is that of the odd relocatable term.

For example, in W-X+W-10, W and X are relocatable terms with the same relocatability attribute. If initially, W equals 10 and X equals 5, the value of the expression is 5. However, upon relocation this value will change. If a relocation factor of 100 is applied, the value of the expression is 105. Note that the value of the paired terms W-X remains constant at 5 regardless of relocation. Thus, the new value of the expression, 105, is the result of the value of the odd term (W) adjusted by the values of W-X and 10.

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

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

ADDRESSING

The addressing technique requires the use of a base register, which contains the base address, and a displacement, which is added to the contents of the base register. The programmer may specify a symbolic address and request the assembler to determine its storage address in terms of a base register and a displacement. The programmer may rely on the assembler to perform this service for him by indicating which general registers are available for assignment, and what values the assembler may assume that each contains. The programmer may use as many or as few registers for this purpose as he desires. The only requirement is that, at the point of reference, a register containing an address from the same control section is available, and that this address is less than or equal to the address of the item to which the reference is being made. The difference between the two addresses may not exceed 4095 bytes.

ADDRESSES -- EXPLICIT AND IMPLIED

An address is composed of a displacement plus the contents of a base register. (In the case of RX instructions, the contents of an index register are also used to derive the address.)

The programmer writes an explicit address by specifying the displacement and the base register number. In designating explicit addresses, a base register must not be combined with a relocatable symbol.

He writes an implied address by specifying an absolute or relocatable address. The assembler has the facility to select a base register and compute a displacement, thereby generating an explicit address from an implied address, provided that the assembler has been informed: (a) what base registers are available to it, and (b) what each contains. The programmer conveys this information to the assembler through the USING and DROP instructions.

BASE REGISTER INSTRUCTIONS

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

To use symbols in the operand field of machine instruction statements, the programmer must: (a) indicate to the assembler, by a USING statement, that one or more general registers are available for use as base registers, (b) specify, by the USING statement what value each base register contains, and (c) load each base register with the value he has specified for it.

If implicit addressing is desired, a program must have at least one USING statement for each control section to be addressed.

The assembler's determination of base registers and displacements relieves the programmer of separating each address into a displacement value and a base address value. This feature of the assembler will eliminate a likely source of programming errors, thus reducing the time required to check out programs. To take advantage of this feature, the programmer uses the USING and DROP instructions described in this subsection. The principal discussion of this feature follows the descriptions of both instructions.

USING -- Use Base Address Register

This instruction indicates that one or more general registers are available for use as base registers. Also, this instruction states the base address values that the assembler may assume will be in the registers at object time. Note that USING does not load the specified registers; it is the programmer's responsibility to see that the base address values are placed into the registers. Suggested loading methods are described in the subsection "Programming With the USING Instruction," below. The format of the USING statement is:

Name	Operation	Operand
Blank	USING	From 2-17 expressions of the form v,r1,r2,r3,...,r16

Operand v must be an absolute or relocatable expression specifying a value that the assembler can use as a base address (no literals are permitted). The other operands must be absolute expressions. Operand r1 specifies the general register that can be assumed to contain the base address represented by operand v. Operands r2, r3, r4, ... specify registers that can be assumed to contain v+4096, v+8192, v+12288, ..., respectively. The values of operands r1, r2, r3, ..., r16 must be between 0 and 15. For example, the statement

Name	Operation	Operand
	USING	*,12,13

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

If the programmer changes the value in a base register currently being used and wishes the assembler to compute displacement from this value, the assembler must be told the new value by means of another USING statement. In the following sequence, the assembler first assumes that the value of ALPHA is in register 9; the second statement causes the assembler to assume that ALPHA+1000 is the value in register 9.

Name	Operation	Operand
	USING	ALPHA,9
	.	
	.	
	USING	ALPHA+1000,9

When a USING statement specifies general register 0 as a base register, the assembler places subsequent effective addresses less than 4096 in the displacement field, and uses 0 for the base register field. This process is the same as for any other base register. Note that the hardware of TSS will not actually reference register 0, but will use zero quantity as a base value. The user should not attempt to use general register 0 as a base register.

DROP -- Drop Base Register

The DROP instruction, specifying a previously available register that may no longer be used as a base register, has this format:

Name	Operation	Operand
Blank	DROP	Up to 16 absolute expressions of the form r1,r2, r3,...,r16, or blank

The expressions indicate general registers previously named in a USING statement that are now unavailable for base addressing. A blank operand field indicates that all registers previously defined as base registers are now unavailable for addressing. This statement, for example, prevents the assembler from using registers 7 and 11:

Name	Operation	Operand
	DROP	7,11

It is not necessary to use a DROP statement when the base address in a register is changed by a USING statement, nor are DROP statements needed at the end of the source program. A register made unavailable by a DROP instruction can be made available again by a subsequent USING instruction.

Note: If a comment is desired on a DROP statement which has no operands, a comma must be used to signify the missing operand.

PROGRAMMING WITH THE USING INSTRUCTION

The USING and DROP instructions may be used anywhere in a program, as often as needed, to indicate the general registers that are available for use as base registers, and the base address values the assembler may assume each contains at execution time. Whenever an address is specified in a machine instruction statement, the assembler determines whether there is an available register containing a suitable base address. A register is considered available for a relocatable address if it was loaded with a relocatable value that is in the same control section as the address. The base address is considered suitable only if it is less than or equal to the address of the item to which the reference

is made. The difference between the two addresses may not exceed 4095 bytes. In calculating the base register to be used, the assembler will always use the available register giving the smallest displacement. If there are two registers with the same value, the highest numbered register will be chosen.

If operand v of a USING statement specifies an absolute value, the assembler will use the associated base registers only for operands with absolute values. In the absence of a base register containing a suitable absolute value, an operand with an absolute value less than 4096 will be placed directly in the displacement field of the assembled instruction. Thus, for example, a base register does not have to be explicitly stated for the operands of shift-type instructions. The programmer is cautioned, however, that in TSS any virtual storage references derived from absolute values will be references to an area of virtual storage that is reserved for use by the system monitor.

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|BEGIN|BASR     |2,0                      |
|     |USING    |*,2                      |
|FIRST|.        |                         |
|     |.        |                         |
|LAST |.        |                         |
|     |END      |BEGIN                    |
```

In the preceding sequence, the BASR instruction loads register 2 with the address of the first storage location immediately following. In this case, it is the address of the instruction named FIRST. The USING instruction indicates to the assembler that register 2 contains this location. When employing this method, the USING instruction must immediately follow the BASR instruction. No other USING or load instructions are required if the location named LAST is within 4095 bytes of FIRST.

The BASR and LM instructions in Figure 3 load registers 2-5. The USING instruction indicates to the assembler that these registers are available as base registers for addressing a maximum of 16,384 consecutive bytes of storage, beginning with the location named HERE. The number of addressable bytes may be increased or decreased, altering the number of registers designated, by the USING and LM instructions and the number of address constants specified in the DC instruction.

```
|Name    |Operation|Operand                  |
|--------|---------|-------------------------|
|BEGIN   |BASR     |2,0                      |
|        |USING    |HERE,2,3,4,5             |
|HERE    |LM       |3,5,BASEADDR             |
|        |B        |FIRST                    |
|BASEADDR|DC       |A(HERE+4096,             |
|        |         | HERE+8192,HERE+12288)   |
|FIRST   |.        |                         |
|        |.        |                         |
|LAST    |.        |                         |
|        |END      |BEGIN                    |
```

Figure 3. Multiple Base Register Assignment

RELATIVE ADDRESSING

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

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|ALPHA|LR       |3,4                      |
|     |CR       |4,6                      |
|     |BCR      |1,14                     |
|BETA |AR       |2,3                      |
```

PROGRAM SECTIONING AND LINKING

It is often convenient, or necessary, to write a large program in sections. The sections may be assembled separately, then combined into one object program. The assembler provides facilities for creating multisectioned programs and symbolically linking separately assembled programs or program sections.

Sectioning a program is optional, and many programs can best be written without sectioning them. The programmer writing an unsectioned program need not concern himself with the subsequent discussion of program sections, which are called control sections. He need not employ the CSECT instruction, which is used to identify the control sections of a multisection program. Similarly, he need not concern himself with the discussion of symbolic linkages, if his

program neither requires a linkage to nor receives a linkage from another program. He may, however, wish to identify the program and/or specify a tentative starting location for it, both of which may be done by using the START instruction. He may also want to employ the dummy section feature obtained by using the DSECT instruction.

Note: Program sectioning and linking is closely related to the specification of base registers for each control section. Sectioning and linking examples are provided in this section under "Addressing External Control Sections."

CONTROL SECTIONS

The concept of program sectioning is a consideration at coding time, assembly time, and load time. To the programmer, a program is a logical unit. He may want to divide it into sections called control sections; if so, he writes it in such a way that control passes properly from one section to another, regardless of the relative physical position of the sections in storage. A control section is a block of coding whose virtual storage location assignments can be adjusted, independently of other coding, during linkage or loading without altering or impairing the operating logic of the program. A control section is normally identified by the CSECT, PSECT, or COM assembler instructions.

To the assembler there is no such thing as a program; instead, there is an assembly, which consists of one or more control sections. (However, the terms assembly and program are often used interchangeably.) An unsectioned program is treated as a single control section. To the linkage editor there are no programs, only control sections that must be fashioned into an object program.

The output of the assembler consists of the assembled control sections and a control dictionary. The control dictionary contains information that the linkage editor and the loader need to complete cross-referencing between control sections, as it combines them into an object program. The linkage editor and the loader can take control sections from various assemblies and combine them properly, with the help of the corresponding control dictionaries. Successful combination of separately assembled control sections depends on the techniques used to provide symbolic linkages between the control sections.

Regardless of the degree to which his program is sectioned, the programmer still knows the elements that comprise his virtual storage, because he has described them symbolically. He cannot, however, make any assumptions about the position or ordering of control sections, since their virtual storage location assignments may have been adjusted by the linkage editor or the loader, and their physical storage addresses may be constantly changing within the time sharing environment.

Control Section Location Assignment

Control section contents can be written in an intermixed manner because the assembler provides a location counter for each control section. Virtual storage locations are assigned consecutively within each control section, beginning at 0. The order in which different control sections appear in the assembly does not imply a similar order of program execution.

FIRST CONTROL SECTION OF PROGRAM

Normally, this contains the literals requested in the program, although their positioning can be altered. (Further explanation is in the discussion of the LTORG assembler instruction, below.)

START -- Start Assembly

The START instruction may be used to give a name to the first (or only) control section of a program; only one START instruction may be in an assembly. Also, it may be used to specify the initial virtual storage location counter value for the first control section. The START format is:

Name	Operation	Operand
A symbol or blank	START	A self-defining term or blank

If a symbol names the START instruction, the symbol is established as the name of the control section; if not, the control section is considered to be unnamed. All subsequent statements are assembled as part of that control section. The procedure continues until a different CSECT instruction identifying a control section, or a DSECT, PSECT or COM, instruction is encountered. A CSECT instruction named by the same symbol that names a START instruction is considered to identify the continuation of the control section first identified by the START. Similarly, an unnamed CSECT that occurs in a program initiated by an unnamed START is considered to identify the continuation of the unnamed control section.

The symbol in the name field is a valid relocatable symbol with a value that represents the address of the first byte of the control section and with a length attribute of 1.

The assembler uses the self-defining value specified by the operand as the starting value for the virtual storage location counter for the control section. The START instruction is, in effect, equivalent to a CSECT instruction followed by an ORG instruction. For example, either of these statements could be used to assign the name PROG2 to the first control section and to indicate an initial location counter value of 2040:

```
|Name |Operation|Operand                |
|-----|---------|-----------------------|
|PROG2|START    |2040                   |
|PROG2|START    |X'7F8'                 |
```

If the operand is omitted, the assembler sets the initial value of the location counter to 0. The location counter is set at the next doubleword boundary when the value of the START operand is not divisible by 8.

Note: The START instruction may not be preceded by any assembler language statement that affects or depends on the setting of the location counter.

CSECT -- Identify Control Section

The CSECT instruction identifies the beginning or the continuation of a control section, in this format:

```
|Name     |Operation|Operand                |
|---------|---------|-----------------------|
|A symbol |CSECT    |One or more attribute  |
|or blank |         |names, or blank        |
```

If a symbol names the CSECT instruction, the symbol is established as the name of the control section; otherwise, the section is considered to be unnamed. All statements following the CSECT are assembled as part of that control section until a statement identifying a different control section (another CSECT, PSECT, COM or DSECT instruction) is encountered.

The symbol in the name field is a valid relocatable symbol with a value that represents the address of the first byte of the control section and with a length attribute of 1.

When several CSECT statements with the same name appear within a program, the first is considered to identify the beginning of the control section; the rest identify the resumption of the section. Thus, statements from different control sections may be interspersed. They are properly assembled (assigned contiguous storage locations) as long as the statements from the various control sections are identified by the appropriate CSECT instructions.

The operand field may be used to assign attributes to the control section. Attributes of control sections are discussed later in this section.

Unnamed Control Section

The assembler will produce an unnamed control section if it encounters certain statements before any type of control section statement (CSECT, PSECT, COM, or DSECT) is encountered. The statements for which this will be done are those which assume that a location counter value is available. Such statements are:

1. Machine operation instructions
2. Macro instructions (but not instructions within macro definitions)
3. CCW
4. CNOP
5. CXD
6. DC, DS, and ORG
7. EQU
8. USING and DROP
9. LTORG
10. END
11. ENTRY

If an unnamed CSECT is not wanted, then the above statements should follow a CSECT, PSECT, COM, or DSECT instruction.

There can be only one unnamed control section in a program. If one is initiated and is then followed by a named control section, any subsequent unnamed CSECT statements are considered to resume the unnamed control section. If a programmer wants to write a small program that is unsectioned, the program need not contain a CSECT instruction.

DSECT -- Identify Dummy Section

A dummy section represents a control section that is assembled but is not part of the object program. A dummy section is a convenient means of describing the layout of a storage area without actually reserving the storage. (It is assumed that the storage is reserved either by some other part of this assembly or by another assembly.) The DSECT instruction identifies the beginning or resumption of a dummy section. More than one dummy section may be defined per assembly, but each must be named. This is the format of the DSECT instruction statement:

Name	Operation	Operand
A symbol	DSECT	Not used; should be blank

The symbol in the name field is a valid relocatable symbol with a value that represents the first byte of the section and with a length attribute of 1.

Program statements belonging to dummy sections may be interspersed throughout the program or may be written as a unit. In either case, the appropriate DSECT instruction should precede each set of statements. When multiple DSECT instructions with the same name are encountered, the first is considered to initiate the dummy section, and the rest to continue it.

Symbols that name statements in a dummy section may be in USING instructions. Therefore, they may be used in program elements (for example, machine instructions and data definitions) that specify storage addresses. An example illustrating the use of a dummy section appears under "Addressing Dummy Sections," below.

Note: A symbol that names a statement in a dummy section may be used in an A-type address constant only if it is paired with another symbol (with the opposite sign) from the same dummy section.

DUMMY SECTION LOCATION ASSIGNMENT: A location counter is used to determine the relative locations of named program elements in a dummy section. The location counter is always set to 0 at the beginning of the dummy section; the location values assigned to symbols that name statements in the dummy section are relative to the initial statement in the section.

ADDRESSING DUMMY SECTIONS: The programmer may wish to describe the format of an area whose storage location will not be determined until the program is executed. He can describe the format of the area in a dummy section, and he can use symbols defined in the dummy section as the operands of machine instructions. To effect references to the storage area, he does the following:

1. Provides a USING statement that specifies: (a) a general register that the assembler can assign to the machine instructions as a base register, and (b) a value from the dummy section that the assembler may assume the register contains.

2. Ensures that the same register is loaded with the actual address of the storage area.

The value assigned to symbols defined in a dummy section are relative to the initial statement of the section. Thus, all machine instructions which refer to names defined in the dummy section will, at execution time, refer to storage locations relative to the address loaded into the register.

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

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

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

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

COM -- Define Common Control Sections

The COM assembler instruction identifies and reserves common areas of storage that may be referred to by independent assemblies that have been linked and/or loaded for execution as one overall program. Appearance of another COM statement after the initial one indicates resumption of the previously defined blank or named common control section. One blank and any number of named common control sections can be designated in an assembly. The format is:

```
|Name    |Operation|Operand                |
|--------|---------|-----------------------|
|Symbol  |COM      |One or more attributes |
|or blank|         |or blank               |
```

The common area may be broken up into subfields through use of the DS and DC assembler instructions. Names of subfields are defined relative to the beginning of the common section, as in the DSECT control section.

No instructions or constants are assembled in the blank common control section. Data can be placed there only through execution of the program. Instructions and constants can be assembled in named common control sections. The rules governing the final structure of common control sections are described in Linkage Editor.

The operand field may be used to assign attributes to a common section. Attributes of control sections are discussed later in this section.

```
|Name    |Operation|Operand                |
|--------|---------|-----------------------|
|ASMBLY3 |CSECT    |PUBLIC,READONLY        |
|BEGIN   |BASR     |1,0                    |
|        |USING    |*,1                    |
|        |USING    |INAREA,2               |
|        |USING    |PRCTL,3                |
|        |USING    |NAMED,4                |
|*       |.        |                       |
|*       |.        |                       |
|NEWPG   |MVC      |CUTPUTB,TITLE          |
|        |MVC      |CUTPUTA,BLANKS         |
|        |MVI      |PRCTL,C'1'             |
|*       |.        |                       |
|*       |.        |                       |
|        |CLI      |INCODE,C'E'            |
|        |BE       |MVLINE                 |
|*       |.        |                       |
|*       |.        |                       |
|MVLINE  |MVC      |OUTPUTA,INPUTA         |
|        |MVC      |OUTPUTB,INPUTB         |
|        |MVI      |PRCTL,C' '             |
|*       |.        |                       |
|*       |.        |                       |
|INAREA  |DSECT    |                       |
|INCODE  |DS       |CL1                    |
|INPUTA  |DS       |CL20                   |
|INPUTB  |DS       |CL14                   |
|*       |.        |                       |
|*       |.        |                       |
|        |COM      |                       |
|PRCTL   |DS       |CL1                    |
|OUTPUTA |DS       |CL20                   |
|OUTPUTB |DS       |CL14                   |
|*       |.        |                       |
|*       |.        |                       |
|NAMED   |COM      |                       |
|TITLE   |DC       |CL14'ERROR MESSAGES'   |
|BLANKS  |DC       |CL20' '                |
|        |END      |BEGIN                  |
```

PSECT -- Define Prototype Control Section

Within TSS, a single copy of a commonly used, reenterable routine will appear to have different virtual storage location assignments to different users, although its physical disposition in storage remains unchanged. When control is transferred to a reenterable routine, the calling program must supply an address constant which reflects the virtual storage assignments of the calling program, so the reenterable routine may obtain data storage that is unique to the user.

This would ordinarily imply that a program that calls a reenterable routine knows all address constants which might be required within the hierarchy of reenterable programs. To minimize this clerical

burden, a prototype control section is defined for use by reenterable programs to simplify the handling of address constants and working storage. The format is:

Name	Operation	Operand
A symbol	PSECT	One or more attribute names, or blank

The contents of a prototype section are identical in every respect with those of a control section (CSECT). However, on linkage to the reenterable routine, a copy of the contents of the prototype section is made and then assigned to virtual storage locations within the domain of the calling program.

A reenterable program is then free to assemble all its working storage and address constants within a prototype section, and the user need not know any of the internal requirements of the routine he calls.

Communication of prototype section information is accomplished through use of the R-type address constant (see Section 5).

The operand field may be used to assign attributes to a prototype section.

EXTERNAL DUMMY SECTIONS

External dummy sections allow the programmer to define work area requests for several different programs and, at execution time, combine these requests into one block of storage accessible to each program. Several different programs may be assembled together, each with one or more external dummy sections. After the loader has processed these programs, the programmer, having issued a GETMAIN macro instruction, can dynamically allocate storage for the dummy sections in one block. External dummy sections are defined through the use of a Q-type DC instruction in combination with a DXD or a DSECT instruction. In order to allocate the correct amount of storage when the program is executed, the programmer must use the CXD instruction, described in Section 5, within one of the programs.

ATTRIBUTES OF CONTROL SECTIONS

To facilitate dynamic linkage and loading within TSS, it is often necessary to indicate that certain attributes are characteristic of the data or instructions within a control section. One or more of the following operands may be used in CSECT, PSECT, or COM statements to indicate which attributes are to be assigned to the section:

PUBLIC - Section contains shared public data or instructions.

READONLY - Section contains instructions or data that are never modified.

VARIABLE - Length of section may vary during program execution.

PRVLGD - When section is assigned space by loader, a protection key is to be assigned to it; only privileged system service routines have read or write access to it.

SYSTEM - Section may contain entry points to system subroutines whose entry point names begin with SYS.

Attributes may be specified singly or in combination, where meaningful. If a section is interrupted and resumed, as described above under CSECT, the final set of attributes for the section is determined by combining the attributes which appear on each of the various CSECT, PSECT, or COM statements. If no attributes are specified, the section is defined as a standard control, prototype, or common section.

SYMBOLIC LINKAGES

Symbols may be defined in one program and referred to in another, thus effecting symbolic linkages between independently assembled programs. The linkages can be effected only if the assembler is able to provide information about the linkage symbols to the dynamic loader, which resolves these linkage references at load time. The assembler places the necessary information in the control dictionary on the basis of the linkage symbols identified by the ENTRY and EXTRN instructions. Note that these symbolic linkages are described as linkages between independent assemblies; more specifically, they are linkages between independently assembled control sections.

In the program where the linkage symbol is defined (i.e., used as a name), it must also be identified to the assembler by the ENTRY assembler instruction. It is identified as a symbol that names an entry point, which means that another program will use that symbol to effect a branch operation or

a data reference. The assembler places this information in the control dictionary.

Similarly, the program that uses a symbol defined in some other program must identify it by the EXTRN assembler instruction. It is identified as an externally defined symbol (that is, defined in another program) that is used to effect linkage to the point of definition. The assembler places this information in the control dictionary.

Another way to obtain symbolic linkages is by using the V-type address constant. "Data Definition Instructions" in Section 5 contains the details pertinent to writing a V-type address constant. It is sufficient here to note that this constant may be considered an indirect linkage point. It is created from an externally defined symbol, but that symbol does not have to be identified by an EXTRN statement. The V-type address constant may be used for external branch references (that is, for effecting branches to other programs) or for external data references.

ENTRY -- Identify Entry Point Symbol

The ENTRY instruction identifies linkage symbols that are defined in this program but may be used by some other program. The length attribute of an external symbol is 1. The format of the ENTRY instruction statement is:

Name	Operation	Operand
Blank	ENTRY	One or more relocatable or absolute symbols, separated by commas, that also appear as statement names

The symbols in the ENTRY operand field may be used as operands by other programs. An ENTRY statement operand may not contain a symbol defined in a dummy section. The following example identifies the statements named SINE and COSINE as entry points to the program.

Name	Operation	Operand
	ENTRY	SINE,COSINE

If the ENTRY statement appears within a named CSECT, PSECT, or named common section, all the operands appearing in the ENTRY statement are associated with the name of the section. These entry names may then be referenced with R-type address constants in other programs.

Note: The ENTRY statement may not appear in a DSECT, unnamed common section, or an unnamed control section.

Note: The name of a control section cannot be identified by an ENTRY instruction. The assembler automatically places information on control section names in the control dictionary. Multiple declarations of the same ENTRY name, either through duplicate ENTRY statements or duplication of an operand within an ENTRY statement, do not cause multiple definitions to be entered in the output program module.

Note: An operand of an ENTRY statement must appear in the name field of another statement; however, a symbol declared in an ENTRY statement may not be the name of an EQU statement if the EQU contains an externally defined symbol in its operand field. Under this rule, the following code is incorrect:

Name	Operation	Operand
	ENTRY	A
	EXTRN	X
A	EQU	X

EXTRN -- Identify External Symbol

The EXTRN instruction identifies linkage symbols that are used by this program but defined in some other program. Each external symbol must be identified; this includes symbols that name control sections. The length attribute of an external symbol is 1. This is the format of the EXTRN instruction statement:

Name	Operation	Operand
Blank	EXTRN	One or more relocatable symbols, separated by commas

The symbols in the operand field may not appear as names of statements in this program. The following example identifies three external symbols that have been used as operands in this program but are defined in some other program.

Name	Operation	Operand
	EXTRN	RATEBL,PAYCALC,WITHCALC

An example that employs the EXTRN instruction appears under "Addressing External Control Sections," below.

Notes:

(1) Symbols appearing in the operand field of a V-type or R-type address constant need not be defined by an EXTRN statement.

(2) When external symbols are used in an expression, they may not be paired; each must be considered as having a unique relocatability attribute.

Addressing External Control Sections

A common way for a program to link to an external control section is to:

1. Create a V-type address constant with the name of the external symbol. If the external control section is reenterable, create an R-type constant with the name of the external symbol.

2. Load the constants into general registers and branch to the control section, via the register containing the V-type address constant.

For example, to link to the control section named SINE, this coding might be used:

```
|Name     |Operation|Operand              |
|---------|---------|---------------------|
|MAINPROG |CSECT    |                     |
|BEGIN    |BASR     |2,0                  |
|         |USING    |*,2                  |
|         |.        |                     |
|         |.        |                     |
|         |L        |13,RCON              |
|         |L        |15,VCON              |
|         |BASR     |14,15                |
|         |.        |                     |
|         |.        |                     |
|RCON     |DC       |R(SINE)              |
|VCON     |DC       |V(SINE)              |
|         |END      |BEGIN                |
```

To use an external symbol naming data, the programmer:

1. Identifies the external symbol with the EXTRN instruction and creates an address constant from the symbol.

2. Loads the constant into a general register and uses the register for base addressing.

For example, to use an area named RATETBL, which is in another control section, this coding might be used:

```
|Name     |Operation|Operand              |
|---------|---------|---------------------|
|MAINPROG |CSECT    |                     |
|BEGIN    |BASR     |2,0                  |
|         |USING    |*,2                  |
|         |.        |                     |
|         |.        |                     |
|         |EXTRN    |RATETBL              |
|         |.        |                     |
|         |.        |                     |
|         |L        |4,RATEADDR           |
|         |USING    |RATETBL,4            |
|         |A        |3,RATETBL            |
|         |.        |                     |
|         |.        |                     |
|RATEADDR |DC       |A(RATETBL)           |
|         |END      |BEGIN                |
```

Alternatively, a V-type address constant may be used to refer to externally named data:

```
|Name     |Operation|Operand              |
|---------|---------|---------------------|
|MAINPROG |CSECT    |                     |
|BEGIN    |BASR     |2,0                  |
|         |USING    |*,2                  |
|         |.        |                     |
|         |.        |                     |
|         |L        |4,RATEADDR           |
|         |USING    |RATETAB,4            |
|         |A        |3,RATE               |
|         |.        |                     |
|         |.        |                     |
|RATEADDR |DC       |V(RATETBL)           |
|RATETAB  |DSECT    |                     |
|RATE     |DS       |F                    |
|         |END      |BEGIN                |
```

This section discusses coding of the machine instructions represented in the assembler language. The functions of each machine instruction are discussed in Principles of Operation.

MACHINE INSTRUCTION STATEMENTS

Machine instructions may be represented symbolically as assembler language statements. The symbolic format of each varies according to the actual machine instruction format, of which there are five: RR, RX, RS, SI, and SS. Within each basic format, further variations are possible.

The symbolic format of a machine instruction is similar to, but does not duplicate, its actual format. A mnemonic operation code is written in the operation field; one or more operands are written in the operand field. Comments may be appended to a machine instruction statement, as explained in Section 1.

Any machine instruction statement may be named by a symbol that other assembler statements can use as an operand. The value attribute of the symbol is the address of the leftmost byte assigned to the assembled instruction. The length attribute of the symbol depends on the basic instruction format:

Basic Format	Length Attribute
RR	2
RX	4
RS	4
SI	4
SS	6

Instruction Alignment and Checking

All machine instructions are aligned automatically by the assembler on halfword boundaries. If any statement that causes information to be assembled requires alignment, the bytes skipped are filled with hexadecimal 0s. All expressions that specify storage addresses are checked to ensure that they refer to appropriate boundaries for the instructions in which they are used. Register numbers are also checked to make sure that they specify the proper registers:

1. Floating point instructions must specify floating point registers 0, 2, 4, or 6.

2. Double-shift, fullword multiply, and divide instructions must specify an even-numbered general register in the first operand.

OPERAND FIELDS AND SUBFIELDS

Some symbolic operands are written as a single field; other operands are written as a field followed by one or two subfields. For example, addresses consist of the contents of a base register and a displacement. An operand that specifies a base and displacement is written as a displacement field followed by a base register subfield, as 40(5). In the RX format, both an index register subfield and a base register subfield are written as 40(3,5). In the SS format, both a length subfield and a base register subfield are written as 40(21,5).

Two types of addressing formats for RX, RS, SI, and SS instructions are shown in Appendix B. In each case, the first type shows the method of specifying an address explicitly, as a base register and displacement. The second type indicates how to specify an implied address as an expression.

For example, a load multiple instruction (RS format) may have either of these symbolic operands:

```
R1,R3,D2(B2) - explicit address
R1,R3,S2 - implied address
```

While D2 and B2 must be represented by absolute expressions, S2 may be represented by either a relocatable or an absolute expression.

To use implied addresses, these rules must be observed:

1. The base register assembler instructions (USING and DROP) must be used.

2. An explicit base register designation must not accompany the implied address.

For example, assume that FIELD is a relocatable symbol which has been assigned a value of 7400. Assume also that the assembler has been notified (by a USING instruction) that general register 12 cur-

rently contains a relocatable value of 4096 and is available as a base register. The following example shows a machine instruction statement as it would be written in assembler language and as it would be assembled. Note that the value of D2 is the difference between 7400 and 4096, and that X2 is assembled as 0, since it was omitted. The assembled instruction is presented in hexadecimal:

Assembler statement:

```
ST      4,FIELD
```

Assembled instruction:

```
Op. Code R1  X2  B2  D2
50       4   0   C   CE8
```

An address may be specified explicitly as a base register and displacement (and index register for RX instructions) by the formats shown in the first column of Table 1. The address may be specified as an implied address by the formats shown in the second column. Observe that the two storage addresses required by the SS instructions are presented separately; an implied address may be used for one and an explicit address for the other.

Table 1. Details of Address Specification

Type	Explicit Address	Implied Address
RX	D2(X2,B2)	S2(X2)
	D2(0,B2)	S2
RS	D2(B2)	S2
SI	D1(B1)	S1
SS	D1(L1,B1)	S1(L1)
	D1(L,B1)	S1(L)
	D2(L2,B2)	S2(L2)

A comma must be written to separate operands. Parentheses must be written to enclose subfields, and a comma must be written to separate two subfields within parentheses. When parentheses are used to enclose one subfield and the subfield is omitted, the parentheses must be omitted. In the case of two subfields that are separated by a comma and enclosed by parentheses, these rules apply:

1. If both subfields are omitted, the separating comma and the parentheses must also be omitted.

   ```
   L 2,48(4,5)
   L 2,FIELD (implied address)
   ```

2. If the first subfield in the sequence is omitted, the comma that separates it from the second subfield is written; the parentheses must also be written.

   ```
   MVC 32(16,5),FIELD2
   MVC BETA(,5),FIELD2 (implied
    length)
   L 2,48(,5) (omitted index register)
   ```

3. If the second subfield in the sequence is omitted, the comma that separates it from the first subfield must be omitted; the parentheses must be written.

   ```
   MVC 32(16,5),FIELD2
   MVC FIELD1(16),FIELD2 (implied
    address)
   ```

Fields and subfields in a symbolic operand may be represented either by absolute or by relocatable expressions, depending on what the field requires. (An expression is defined as consisting of one term or a series of arithmetically combined terms.) Refer to Appendix B for a detailed description of field requirements.

Note: Blanks may not appear in an operand unless provided as characters in a character self-defining term or a character literal; blanks may not intervene between fields and the comma separators, between parentheses and fields, etc.

LENGTHS -- EXPLICIT AND IMPLIED

The length field in SS instructions can be explicit or implied. To imply a length, the programmer omits a length field from the operand. The omission indicates that the length field is either:

1. The length attribute of the expression specifying the displacement, if an explicit base and displacement have been written.

2. The length attribute of the expression specifying the effective address, if the base and displacement have been implied.

In either case, the length attribute for an expression is the length of the leftmost term in the expression.

By contrast, an explicit length is written by the programmer in the operand as an absolute expression. The explicit length overrides any implied length.

Whether the length is explicit or implied, it is always an effective length. The value inserted into the length field of the assembled instruction is one less than the effective length in the machine instruction statement.

Note: If a length field of 0 is desired, for use with an execute (EX) machine instruction, the length may be stated as 0.

To summarize, the length required in an SS instruction may be specified explicitly by the formats shown in the first column of Table 2, or may be implied by the formats shown in the second column. Observe that the two lengths required in one of the SS instruction formats are presented separately. An implied length may be used for one and an explicit length for the other. For the SS instruction format, which requires a single length field (L), the implicit length used is that of the first operand.

Table 2. Details of Length Specification in SS Instructions

Explicit Length	Implied Length
D1(L1,B1)	D1(,B1)
S1(L1)	S1
D1(L,B1)	D1(,B1)
S1(L)	S1
D2(L2,B2)	D2(,B2)
S2(L2)	S2

MACHINE INSTRUCTION MNEMONIC CODES

The mnemonic operation codes are easily remembered for indicating the functions of the instructions. The normal format of the code is shown below; the items in brackets are not necessarily present in all codes:

Verb[Modifier] [Data Type] [Machine Format]

The verb, which is usually one or two characters, specifies the function (A represents Add, and MV represents Move). The function may be further defined by a modifier (modifier L indicates a logical function, as in AL for Add Logical).

Mnemonic codes for functions involving data usually indicate the data types by letters that correspond to those for the data types in the DC assembler instruction (see Section 5). Furthermore, letters U and W have been added to indicate short and long unnormalized floating-point operations, respectively. For example, AE indicates Add Normalized Short, whereas AU indicates Add Unnormalized Short. Where applicable, fullword fixed-point data is implied if the data type is omitted.

The letters R and I are added to the codes to indicate, respectively, RR and SI machine instruction formats. Thus, AER indicates Add Normalized Short in the RR format. Functions involving character and decimal data types imply the SS format.

MACHINE INSTRUCTION EXAMPLES

The examples that follow are grouped according to machine instruction format. They illustrate the various symbolic operand formats. All symbols employed in the examples must be assumed to be defined elsewhere in the same assembly. All symbols that specify register numbers and lengths must be assumed to be equated elsewhere to absolute values.

Implied addressing, control section addressing, and the function of the USING assembler instruction were considered, together with examples of coding sequences, earlier in this section, under "Program Sectioning and Linking" and "Base Register Instructions."

RR Format

Name	Operation	Operand
ALPHA1	LR	1,2
ALPHA2	LR	REG1,REG2
BETA	SPM	15
GAMMA1	SVC	250
GAMMA2	SVC	TEN

The operands of ALPHA1, BETA, and GAMMA1 are decimal self-defining values, which are categorized as absolute expressions. The operands of ALPHA2 and GAMMA2 are symbols that are equated elsewhere to absolute values.

RX Format

Name	Operation	Operand
ALPHA1	L	1,39(4,10)
ALPHA2	L	REG1,39(4,TEN)
BETA1	L	2,ZETA(4)
BETA2	L	REG2,ZETA(REG4)
GAMMA1	L	2,ZETA
GAMMA2	L	REG2,ZETA
GAMMA3	L	2,=F'1000'
LAMBDA1	L	3,20(0,5)

Both ALPHA instructions specify explicit addresses; REG1 and TEN are absolute symbols. Both BETA instructions specify implied addresses, and both use index registers. Indexing is omitted from the GAMMA instructions. GAMMA1 and GAMMA2 specify implied addresses. The second operand of GAMMA3 is a literal. LAMBDA1 specifies no indexing.

RS Format

Name	Operation	Operand
ALPHA1	BXH	1,2,20(14)
ALPHA2	BXH	REG1,REG2,20(REGD)
ALPHA3	BXH	REG1,REG2,ZETA
ALPHA4	SLL	REG2,15
ALPHA5	SLL	REG2,0(15)

While ALPHA1 and ALPHA2 specify explicit addresses, ALPHA3 specifies an implied address. ALPHA4 is a shift instruction, shifting the contents of REG2 left 15 bit positions. ALPHA5 is a shift instruction, shifting the contents of REG2 left by the value contained in general register 15.

SI Format

Name	Operation	Operand
ALPHA1	CLI	40(9),X'40'
ALPHA2	CLI	40(REG9),TEN
BETA1	CLI	ZETA,TEN
BETA2	CLI	ZETA,C'A'
GAMMA1	SIO	40(9)
GAMMA2	SIO	0(9)
GAMMA3	SIO	40(0)
GAMMA4	SIO	ZETA

The ALPHA instructions and GAMMA1-GAMMA3 specify explicit addresses; the BETA instructions and GAMMA4 specify implied addresses. GAMMA2 specifies a displacement of 0. GAMMA3 does not specify a base register.

SS Format

Name	Operation	Operand
ALPHA1	AP	40(9,8),30(6,7)
ALPHA2	AP	40(NINE,REG8),30(L6,7)
ALPHA3	AP	FIELD2,FIELD1
ALPHA4	AP	FIELD2(9),FIELD1(6)
BETA	AP	FIELD2(9),FIELD1
GAMMA1	MVC	40(9,8),30(7)
GAMMA2	MVC	40(NINE,REG8),DEC(7)
GAMMA3	MVC	FIELD2,FIELD1
GAMMA4	MVC	FIELD2(9),FIELD1

ALPHA1, ALPHA2, GAMMA1, and GAMMA2 specify explicit lengths and addresses. ALPHA3 and GAMMA3 specify both implied length and implied addresses. ALPHA4 and GAMMA4 specify explicit length and implied addresses. BETA specifies an explicit length for FIELD2 and an implied length for FIELD1; both addresses are implied.

EXTENDED MNEMONIC CODES

For the convenience of the programmer, the assembler provides extended mnemonic codes, which allow conditional branches to be specified mnemonically, as well as through the use of the BC and BCR machine instructions. These extended mnemonic codes specify both the machine branch instruction and the condition on which the branch is to occur. The codes are not part of the universal set of machine instructions, but are translated by the assembler into the corresponding operation and condition combinations.

The allowable extended mnemonic codes and their operand formats are shown in Figures 4 and 5, together with their machine instruction equivalents. All extended mnemonics in Figure 4 are for instructions in the RX format. Figure 5 shows the extended mnemonics for instructions in the RR format. Note that the only difference between the operand fields of the extended mnemonics and those of their machine instruction equivalents is the absence of the R1 field and the comma that separates it from the rest of the operand field. The extended mnemonic list for the RX format, like the machine instruction list, shows explicit address formats only. Each address can also be specified as an implied address.

In the following examples, which illustrate the use of extended mnemonics, it is to be assumed that the symbol GO is defined elsewhere in the program.

Name	Operation	Operand
	B	40(3,6)
	B	40(0,6)
	BL	GO(3)
	BL	GO
	BR	4
	BER	3

The first two instructions specify an unconditional branch to an explicit address. The address in the first case is the sum of the contents of base register 6, the contents of index register 3, and the displacement 40; the address in the second instruction is not indexed. The third instruction specifies a branch on low to the address implied by GO as indexed by the contents of index register 3; the fourth instruction does not specify an index register. The fifth instruction is an unconditional branch to the address contained in register 4. The last instruction specifies a branch on equal to the address contained in register 3.

Extended Code	Meaning	Machine-Instruction
B D2(X2,B2)	Branch Unconditional	BC 15,D2(X2,B2)
NOP D2(X2,B2)	No Operation	BC 0,D2(X2,B2)
	Used After Compare Instructions	
BH D2(X2,B2)	Branch on High	BC 2,D2(X2,B2)
BL D2(X2,B2)	Branch on Low	BC 4,D2(X2,B2)
BE D2(X2,B2)	Branch on Equal	BC 8,D2(X2,B2)
BNH D2(X2,B2)	Branch on Not High	BC 13,D2(X2,B2)
BNL D2(X2,B2)	Branch on Not Low	BC 11,D2(X2,B2)
BNE D2(X2,B2)	Branch on Not Equal	BC 7,D2(X2,B2)
	Used After Arithmetic Expressions	
BO D2(X2,B2)	Branch on Overflow	BC 1,D2(X2,B2)
BP D2(X2,B2)	Branch on Plus	BC 2,D2(X2,B2)
BM D2(X2,B2)	Branch on Minus	BC 4,D2(X2,B2)
BZ D2(X2,B2)	Branch on Zero	BC 8,D2(X2,B2)
BNM D2(X2,B2)	Branch on Not Minus	BC 11,D2(X2,B2)
BNO D2(X2,B2)	Branch on No Overflow	BC 14,D2(X2,B2)
BNP D2(X2,B2)	Branch on Not Plus	BC 13,D2(X2,B2)
BNZ D2(X2,B2)	Branch on Not Zero	BC 7,D2(X2,B2)
	Used After Test Under Mask Instructions	
BO D2(X2,B2)	Branch if Ones	BC 1,D2(X2,B2)
BM D2(X2,B2)	Branch if Mixed	BC 4,D2(X2,B2)
BZ D2(X2,B2)	Branch if Zeros	BC 8,D2(X2,B2)
BNM D2(X2,B2)	Branch on Not Mixed	BC 11,D2(X2,B2)
BNO D2(X2,B2)	Branch on Not Ones	BC 14,D2(X2,B2)
BNZ D2(X2,B2)	Branch on Not Zeros	BC 7,D2(X2,B2)

Figure 4. Extended Mnemonic Codes (RX format)

Extended Code	Meaning	Machine-Instruction
BR R2	Branch Unconditional	BCR 15,R2
NOPR R2	No Operation	BCR 0,R2
Used After Compare Instructions		
BHR R2	Branch on High	BCR 2,R2
BLR R2	Branch on Low	BCR 4,R2
BER R2	Branch on Equal	BCR 8,R2
BNHR R2	Branch on Not High	BCR 13,R2
BNLR R2	Branch on Not Low	BCR 11,R2
BNER R2	Branch on Not Equal	BCR 7,R2
Used After Arithmetic Expressions		
BOR R2	Branch on Overflow	BCR 1,R2
BPR R2	Branch on Plus	BCR 2,R2
BMR R2	Branch on Minus	BCR 4,R2
BZR R2	Branch on Zero	BCR 8,R2
BNMR R2	Branch on Not Minus	BCR 11,R2
BNOR R2	Branch on No Overflow	BCR 14,R2
BHPR R2	Branch on Not Plus	BCR 13,R2
BNZR R2	Branch on Not Zero	BCR 7,R2
Used After Test Under Mask Instructions		
BOR R2	Branch if Ones	BCR 1,R2
BMR R2	Branch if Mixed	BCR 4,R2
BZR R2	Branch if Zeros	BCR 8,R2
BNMR R2	Branch on Not Mixed	BCR 11,R2
BNOR R2	Branch on Not Ones	BCR 14,R2
BNZR R2	Branch on Not Zeros	BCR 7,R2

Figure 5. Extended Mnemonic Codes (RR format)

Just as machine instructions request the computer to perform a sequence of operations during program execution time, so assembler instructions request the assembler to perform certain operations during the assembly. Assembler instruction statements, in contrast to machine instruction statements, do not always cause machine instructions to be included in the assembled program. Some, such as DS and DC, generate no instructions but do cause storage areas to be set aside for constants and other data. Others, such as EQU and SPACE, are effective only at assembly time; they generate nothing in the assembled program and have no effect on the location counter.

This is a list of all the assembler instructions:

Symbol Definition Instruction
EQU - Equate symbol

Data Definition Instructions
DC - Define constant
DS - Define storage
DXD - Define external dummy section
CXD - Cumulative length external dummy section
CCW - Define channel command word

* Program Sectioning and Linking Instructions
START - Start assembly
CSECT - Identify control section
DSECT - Identify dummy section
ENTRY - Identify entry point symbol
EXTRN - Identify external symbol
COM - Identify blank common control section
PSECT - Identify prototype section

* Base Register Instructions
USING - Use base address register
DROP - Drop base address register

* Discussed earlier in this section

Listing Control Instructions
TITLE - Identify assembly output
EJECT - Start new page
SPACE - Space listing
PRINT - Print optional data

Program Control Instructions
ICTL - Input format control
ISEQ - Input sequence checking
ORG - Set location counter
LTORG - Begin literal pool
CNOP - Conditional no operation
COPY - Copy predefined source coding
END - End assembly
PUNCH - Punch card
REPRO - Reproduce following card

SYMBOL DEFINITION INSTRUCTION

EQU -- Equate Symbol

The EQU instruction defines a symbol by assigning to it the attributes specified in the operand fields. The format of the EQU instruction statement is:

Name	Operation	Operand
Symbol, variable symbol, or blank	EQU	Previously defined symbol,length,type

The symbol in the name field is optional; if it is omitted, the statement is treated as a comment. A variable symbol is valid in the name field; it is possible to code

```
&NAME  EQU  *
```

where &NAME is undefined.

The first operand may be absolute (including negative), relocatable, or complex. Symbols in this field must be previously defined. This operand may not be omitted unless the name field is also omitted.

The second operand specifies an explicit length attribute. It consists of any absolute integer expression with a value from 1 to 65535, or a 1 - 2 byte self-defining term (hex, character, or binary).

The third operand specifies an explicit type attribute. It consists of any absolute integer expression with a value from 0 to 255, or a 1-byte self-defining term (hex, character, or binary).

The symbol in the name field is given the attributes explicitly specified by the second and third operands. If the second operand is omitted, the symbol in the name field will receive the length attribute of the first operand. If the first operand consists of an absolute value, the length attribute of the symbol is 1; otherwise, the length attribute is that of the leftmost (or only) term of the first operand.

If the type attribute is omitted, the symbol's type attribute is assigned as U (undefined).

The value attribute of the symbol in the name field is the value of the first operand.

Note: The second and third operands are optional; if they are specified, they must be written in the order shown above. If the length attribute is omitted, and the type attribute is specified, a comma must be used to indicate the missing operand.

The EQU instruction equates symbols to register numbers, immediate data, and other arbitrary values. These examples illustrate how this might be done:

Name	Operation	Operand
REG2	EQU	2 (general register)
TEST	EQU	X'3F' (immediate data)

In the following example, LENGTH is equated to EXP2. The length attribute is explicitly specified as 8, overriding the length attribute of EXP2. TYPE is equated to EXP3; the type attribute is C (character). DEFAULT is equated to VALUE. Because the second and third operands are omitted, the length and type attributes are determined by the attributes of VALUE.

Name	Operation	Operand
LENGTH	EQU	EXP2,8
TYPE	EQU	EXP3,,C'C'
DEFAULT	EQU	VALUE

To reduce programming time, the programmer can equate symbols to frequently used expressions, and then use the symbols as operands instead of the expressions. Thus, in the statement:

Name	Operation	Operand
FIELD	EQU	ALPHA-BETA+GAMMA

FIELD is defined as ALPHA-BETA+GAMMA and may be used in place of it. However, ALPHA, BETA, and GAMMA must have been previously defined.

DATA DEFINITION INSTRUCTIONS

The five data definition instruction statements are: define constant (DC), define storage (DS), define external dummy section (DXD), cumulative length external dummy section (CXD), and define channel command word (CCW). They are used to: (a) enter data constants into storage, (b) define and reserve areas of storage, (c) define storage for external dummy sections, (d) specify the cumulative length for external dummy sections, and (e) specify the contents of channel command words. The statements may be named by symbols so that other program statements can refer to the fields generated from them.

Discussion of the DC instruction is more extensive than that of the DS or DXD instructions, because the DS and DXD instructions are written in the same format as the DC instruction and may specify some or all of the information that the DC instruction provides. Only the function and treatment of the statements vary. Therefore, the DC instruction is presented first and discussed in more detail than either the DS or DXD instructions.

DC -- Define Constant

The DC instruction provides constant data in storage. It may specify one constant or a series of constants, thereby relieving the programmer of the necessity to write a separate data definition statement for each constant desired. Furthermore, a variety of constants may be specified: fixed-point, floating-point, decimal, hexadecimal, character, and storage addresses. (Data constants are generally called constants unless they are created from storage addresses, in which case they are called address constants.) The format of the DC instruction statement is:

Name	Operation	Operand
Symbol or blank	DC	One or more operands in format described below, each separated by comma

Each operand consists of four subfields; the first three describe the constant, and the fourth provides the constants. The second and fourth subfields must be specified; the first and third are optional. Note that more than one constant may be specified in the fourth subfield for most types of constants. All specified constants must be of the same type; the descriptive subfields that precede the constants apply to all of them. No blanks may occur within any subfields (unless provided as characters in a character constant or a character self-defining term), nor may they occur between the subfields of an operand. Similarly, blanks may not occur between

operands and the commas that separate them, when multiple operands are being specified.

The subfields of each DC operand are written in this sequence:

1	2	3	4
Duplication Factor	Type	Modifiers	Constants

Although the constants specified in an operand must have the same characteristics, each operand may specify different types of constants. For example, in a DC instruction with three operands, the first operand might specify four decimal constants, the second a floating-point constant, and the third a character constant.

The symbol that names the DC instruction is the name of the constant (or first constant if the instruction specifies more than one). Relative addressing (for example, SYMBOL+2) may be used to address the various constants if more than one has been specified, because the number of bytes allocated to each constant can be determined.

The value attribute of the symbol naming the DC instruction is the address of the leftmost byte (after alignment) of the first, or only, constant. The length attribute depends on two things: the type of constant being defined and the presence of a length specification. Implied lengths are assumed for the various constant types, in the absence of a length specification. If more than one constant is defined, the length attribute is the length in bytes (specified or implied) of the first constant.

Boundary alignment also varies according to the type of constant being specified and the presence of a length specification. Some constant types are only aligned to a byte boundary, but the DS instruction can be used to force any type of word boundary alignment for them. This is explained under "DS -- Define Storage." Other constants are aligned at various word boundaries (half, full, or double) in the absence of a length specification. If length is specified, no boundary alignment occurs for such constants.

Bytes that must be skipped to align the field at the proper boundary are not considered to be part of the constant. In other words, the location counter is incremented to reflect the proper boundary (if incrementing is necessary), before the address value is established. Thus, the symbol naming the constant will not receive a value that is the location of a skipped byte.

Any bytes skipped in aligning statements that do not cause information to be assembled are not set to hexadecimal 0. Thus bytes skipped to align a statement such as DC F'123' are set to hexadecimal 0, and bytes skipped to align a statement such as DS F are not set to hexadecimal 0.

Information concerning constants, presented in this section, is summarized in Appendix C.

LITERAL DEFINITIONS: The discussion of literals as machine instruction operands (in Section 2) referred the reader to the description of the DC operand for the method of writing a literal operand. All subsequent operand specifications are applicable to writing literals; the only differences are:

1. The literal is preceded by an equal sign.
2. Multiple operands may not be specified.
3. The duplication factor may not be 0.
4. S-type address constants may not be specified.

Examples of literals appear throughout the remainder of the DC instruction discussion.

Operand Subfield 1: Duplication Factor

The duplication factor may be omitted. If specified, it causes the constants to be generated as many times as indicated by the factor, which may be specified either by an unsigned decimal self-defining term or by a positive absolute expression that is enclosed by parentheses. All symbols in the expression must be previously defined.

The duplication factor is applied after the constant is assembled. A duplication factor of 0 is permitted, except in a literal, and achieves the same result as it would in a DS instruction (see "Forcing Alignment," under "DS -- Define Storage," later in this section).

Note: If duplication is specified for an address constant containing a location counter reference, the value of the location counter used in each duplication is incremented by the length of the constant.

Operand Subfield 2: Type

This subfield defines the type of constant being specified. From that specification, the assembler determines how it will interpret the constant and translate it into the appropriate machine format.

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
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
Q	Address	Space reserved for offset of external dummy section; normally a fullword
V	Address	Space reserved for external symbol addresses; each address normally a fullword
R	Address	Space reserved for address of control section of external symbol; each address normally a fullword

Figure 6. Type Codes for Constants

The type is specified by a single-letter code, as shown in Figure 6.

Further information about these constants is provided in the discussion of the constants themselves, under "Constant," below.

Operand Subfield 3: Modifiers

Modifiers describe the length in bytes desired for a constant (in contrast to an implied length), and the scaling and exponent for the constant. If multiple modifiers are written, they must appear in this sequence: length, scale, exponent. Each is written and used as described in the following text.

LENGTH MODIFIER: This is written as Ln, where n is either an unsigned decimal value or an absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. A paired relocatable expression cannot be used in a length modifier to form an absolute expression. The value of n represents the number of bytes of storage assembled for the constant. The maximum values permitted for the length modifiers supplied for the various types of constants are summarized in Appendix C. Also, this appendix indicates the implied length for each type of constant; the implied length is used unless a length modifier is present. A length modifier may be specified for any type of constant, but no boundary alignment will be provided when a length modifier is given.

Bit-Length Specification: The bit length of a constant is specified by L.n, where n is specified as above and represents the number of bits in storage into which the constant is to be assembled. The value of n may exceed 8 and is interpreted to mean an integral number of bytes plus so many bits. For example, L.20 is interpreted as a length of two bytes plus four bits.

Assembly of the first (or only) constant with bit-length specification starts on a byte boundary. The constant is placed in the high- or low-order end of the field, depending on the type of constant being specified. The constant is padded or truncated to fit the field. If the assembled length does not leave the location counter set at a byte boundary, and another bit-length constant does not follow, the remainder of the last byte used is filled with 0s. This leaves the location counter set at the next byte boundary. A fixed-point constant with a specified bit length of 13, as coded, and as it would appear in storage, is shown in Figure 7. Note that the constant has been padded on the left to bring it to its designated 13-bit length.

A reference to BLCON, with an implied length of two bytes, would cause both bytes to be referenced.

When bit-length specification is used in association with multiple constants (see "Constants," below), each succeeding constant in the list is assembled, starting at the next available bit. Figure 8 illustrates this.

The symbol used as a name entry in a DC assembler instruction takes on the length attribute of the first constant in the list; the implied length of BLMCON in Figure 8 is two bytes.

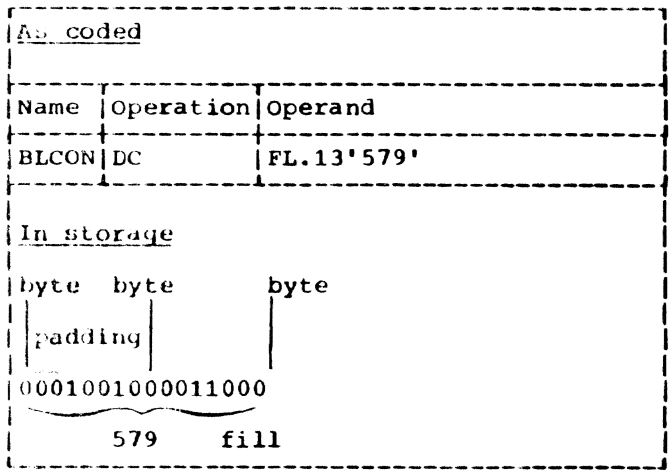


Figure 7. Bit-Length Specification (Single Constant)

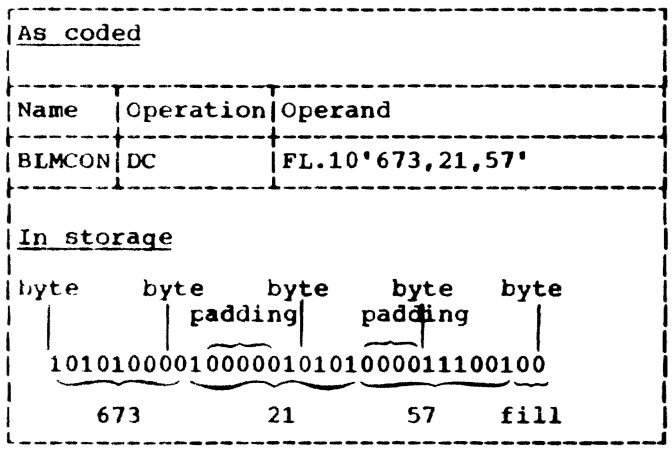


Figure 8. Bit-Length Specification (Multiple Constants)

If duplication is specified, filling occurs once, at the end of the field occupied by the duplicated constants.

When bit-length specification is used in association with multiple operands, assembly of the constants in each succeeding operand starts at the next available bit. Figure 9 illustrates this.

Three different types of constants have been specified in Figure 9, one to an operand. Note that the character constant 'AB', which normally would occupy 16 bits, is truncated on the right to fit the designated 10-bit field. Note that filling occurs only at the end of the field occupied by all the constants.

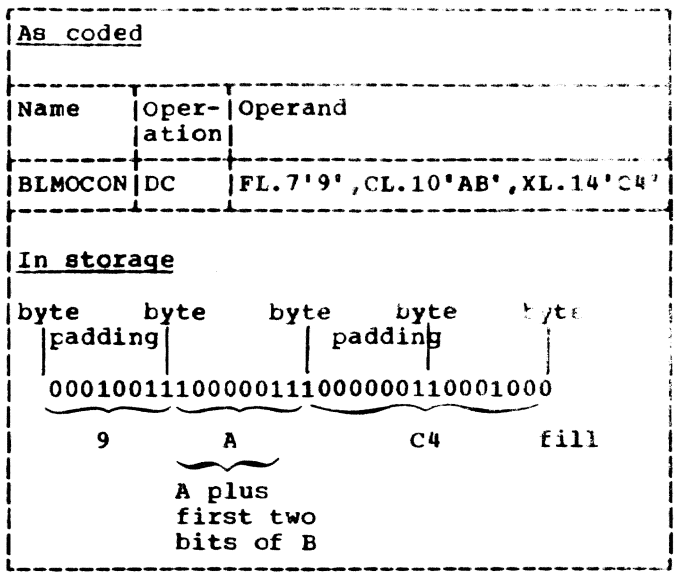


Figure 9. Bit-Length Specification (Multiple Operands)

SCALE MODIFIER: This modifier is written as Sn, where n is either a decimal value or an absolute expression enclosed by parentheses. Any symbol in the expression must be previously defined. The decimal self-defining term or the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for scale modifiers are summarized in Appendix C.

A scale modifier may be used with fixed-point (F, H) and floating-point (E, D) constants only. It is used to specify the desired amount of internal scaling.

Scale Modifier for Fixed-Point Constants: This modifier specifies the power of 2, by which the constant must be multiplied after it has been converted to its binary representation. Just as multiplication of a decimal number by a power of 10 causes the decimal point to move, multiplication of a binary number by a power of 2 causes the binary point to move. This multiplication has the effect of moving the binary point away from its assumed position in the binary field -- the assumed position being to the right of the rightmost position.

Thus, the scale modifier indicates either of these: (a) the number of binary positions to be occupied by the fractional portion of the binary number, or (b) the number of binary positions to be deleted from the integral portion of the binary number.

A positive scale x will shift the integral portion of the number x binary positions to the left, thereby reserving

the rightmost x binary positions for the fractional portion. A negative scale shifts the integral portion of the number to the right, thereby deleting rightmost integral positions. If a scale modifier does not accompany a fixed-point constant containing a fractional part, the fractional part is lost.

In all cases where positions are lost because of scaling (or the lack of scaling), rounding occurs in the leftmost bit of the lost portion. The rounding is reflected in the rightmost position saved.

Scale Modifier for Floating-Point Constants: Only a positive scale modifier may be used with a floating-point constant. It indicates the number of hexadecimal positions that the fraction is to be shifted to the right. Note that this shift is in terms of hexadecimal positions, each of which is four binary positions. (A positive scaling actually indicates that the point is to be moved to the left. However, a floating-point constant is always converted to a fraction, which is hexadecimally normalized. The point is assumed to be at the left of the leftmost position in the field; since the point cannot be moved left, the fraction is shifted right.)

Thus, scaling that is specified for a floating-point constant provides an assembled fraction that is unnormalized; that is, it contains hexadecimal 0s in the leftmost positions of the fraction. When the fraction is shifted, the exponent is adjusted accordingly to retain the correct magnitude. When hexadecimal positions are lost, rounding occurs in the leftmost hexadecimal position of the lost portion. The rounding is reflected in the rightmost hexadecimal position saved.

EXPONENT MODIFIER: This modifier is written as En, where n is either a decimal self-defining term or an absolute expression enclosed by parentheses. Any symbols in the expression must be previously defined. The decimal value of the parenthesized expression may be preceded by a sign; if none is present, a plus sign is assumed. The maximum values for exponent modifiers are summarized in Appendix C.

An exponent modifier may be used with fixed-point (F, H) and floating-point (E, D) constants only. The modifier denotes the power of 10 by which the constant is to be multiplied before its conversion to the proper internal format.

This modifier is not to be confused with the exponent of the constant itself, which is specified as part of the constant and is explained under "Constant," below. Both are denoted in the same fashion, as En. The exponent modifier affects each constant in the operand; the exponent written as part of the constant only pertains to that constant. Thus, a constant may be specified with an exponent of +2, and an exponent modifier of +5 may precede the constant. In effect, the constant has an exponent of +7.

Note that there is a maximum value, both positive and negative, listed in Appendix C, for exponents. This applies both to exponent modifier and exponents specified as part of the constant, or to their sum, if both are specified.

Operand Subfield 4: Constant

This subfield supplies the constants described by the subfields that precede them. A data constant (all types except A, Y, S, Q, V, and R) is enclosed by apostrophes. An address constant (types A, Y, S, Q, V, and R) is enclosed by parentheses. To specify two or more constants in the subfield, the constants must be separated by commas, and the entire sequence of constants must be enclosed by the appropriate delimiters (that is, apostrophes or parentheses). The format for specifying constants is one of these:

Single Constant	Multiple Constants*
'constant'	'constant,...,constant'
(constant)	(constant,...,constant)

*Not permitted for character, hexadecimal, and binary constants.

All constant types except character (C), hexadecimal (X), binary (B), packed decimal (P), and zoned decimal (Z), are aligned on the proper boundary, as shown in Appendix F, unless a length modifier is specified. In the presence of a length modifier, there is no boundary alignment. If an operand specifies more than one constant, any necessary alignment applies to the first constant only. Thus, for an operand that provides five fullword constants, the first would be aligned on a fullword boundary, and the rest would automatically fall on fullword boundaries.

The total storage requirement of an operand is the product of the length multiplied by the number of constants in the operand, multiplied by the duplication factor (if present), plus any bytes skipped for boundary alignment of the first constant. If more than one operand is present, the storage requirement is derived by summing the requirements for each operand.

If an address constant contains a location counter reference, the location counter value to be used is the storage address

of the first byte the constant will occupy. Thus, if several address constants in the same instruction refer to the location counter, the value of the location counter varies from constant to constant. Similarly, if a single constant is specified (and it is a location counter reference) with a duplication factor, the constant is duplicated with a varying location counter value.

The following text describes each constant type and provides examples.

Character Constant -- C: Any of the valid 256 punch combinations may be designated in a character constant. Only one character constant may be specified per operand. Since multiple constants within an operand are separated by commas, an attempt to specify two character constants would result in interpreting the comma separating them as a character.

Special consideration must be given to representing apostrophes and ampersands as characters. Each apostrophe or ampersand desired as a character in the constant must be represented by a pair of apostrophes or ampersands. Only one apostrophe or ampersand appears in storage.

The maximum length of a character constant is 256 bytes. No boundary alignment is performed. Each character is translated into one byte. Double apostrophes or double ampersands count as one character. If no length modifier is given, the size in bytes of the character constant is equal to the number of characters in the constant. If a length modifier is provided, the result varies, as follows:

1. If the number of characters in the constant exceeds the specified length, as many rightmost bytes and/or bits as necessary are dropped.

2. If the number of characters is less than the specified length, the excess rightmost bytes and/or bits are filled with blanks.

In this example, the length attribute of FIELD is 12:

```
|Name |Operation|Operand                  |
|-----+---------+-------------------------|
|FIELD|DC       |C'TOTAL IS 110'          |
```

However, in this example, the length attribute is 15, and three blanks appear in storage to the right of the 0:

```
|Name |Operation|Operand                  |
|-----+---------+-------------------------|
|FIELD|DC       |CL15'TOTAL IS 110'       |
```

In this example, the length attribute of FIELD is 12, although 13 characters appear in the operand; two ampersands count as one byte:

```
|Name |Operation|Operand                  |
|-----+---------+-------------------------|
|FIELD|DC       |C'TOTAL IS &&10'         |
```

Note that in this example, a length of 4 has been specified, but there are five characters in the constant:

```
|Name |Operation|Operand                  |
|-----+---------+-------------------------|
|FIELD|DC       |3CL4'ABCDE'              |
```

The generated constant would be

```
ABCDABCDABCD
```

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

```
ABCDE ABCDE ABCDE
```

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

```
|Name |Operation|Operand                  |
|-----+---------+-------------------------|
|     |MVC      |AREA(12),=3CL4'ABCDE'    |
```

Hexadecimal Constant -- X: A hexadecimal constant consists of one or more of the hexadecimal digits 0-9 and A-F. Only one hexadecimal constant may be specified per operand. The maximum length of a hexadecimal constant is 256 bytes (512 hexadecimal digits). No boundary alignment is performed.

Constants that contain an even number of hexadecimal digits are translated as one byte per pair of digits. If an odd number of digits is specified, the leftmost byte has the leftmost four bits filled with a hexadecimal 0; the rightmost four bits contain the odd (first) digit.

If no length modifier is given, the implied length of the constant is half the number of hexadecimal digits in the constant (assuming that a hexadecimal 0 is added to an odd number of digits). If a

length modifier is given, this is the handling of the constant:

1. If the number of hexadecimal digit pairs exceeds the specified length, the necessary leftmost bits (and/or bytes) are dropped.

2. If the number of hexadecimal digit pairs is less than the specified length, the necessary bits (and/or bytes) are added to the left and filled with hexadecimal 0s.

An 8-digit hexadecimal constant provides a convenient way to set the bit pattern of a full binary word. The constant in the following example would set the first and third bytes of a word to 1s:

```
|Name |Operation|Operand                |
|-----|---------|-----------------------|
|     |DS       |0F                     |
|TEST |DC       |X'FF00FF00'            |
```

The DS instruction sets the location counter to a fullword boundary.

The next example uses a hexadecimal constant as a literal and inserts 1s into bits 24 through 31 of register 5:

```
|Name |Operation|Operand                |
|-----|---------|-----------------------|
|     |IC       |5,=X'FF'               |
```

In this example, the digit A would be dropped, because five hexadecimal digits are specified for a length of two bytes:

```
|Name     |Operation|Operand              |
|---------|---------|---------------------|
|ALPHACON |DC       |3XL2'A6F4E'          |
```

The resulting constant would be 6F4E, which would occupy the specified two bytes. It would then be duplicated three times, as requested by the duplication factor. If X'A6F4E' had been specified, the resulting constant would have had a hexadecimal 0 in the leftmost position:

0A6F4E

Binary Constant -- B: A binary constant uses 1s and 0s enclosed in apostrophes; only one binary constant may be specified in an operand. Duplication and length may be specified. The maximum length of a binary constant is 256 bytes.

The implied length of a binary constant is the number of bytes occupied by the constant, including any padding necessary. Padding or truncation takes place on the left; the padding bit used is a 0.

In this example of the coding used to designate a binary constant, BCCN would have a length attribute of 1:

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|BCCN  |DC       |B'11011101'            |
|BTRUNC|DC       |BL1'100100011'         |
|BPAD  |DC       |BL1'101'               |
```

BTRUNC would assemble with the leftmost bit truncated: 00100011

BPAD would assemble with five 0s as padding: 00000101

Fixed-Point Constants -- F and H: A fixed-point constant is written as a decimal number, which may be followed by a decimal exponent if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number; it may be omitted, in which case the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified. Unless a scale modifier accompanies a mixed number or fraction, the fractional portion is lost, as explained under "Operand Subfield 3: Modifiers," above.

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent may be in the range -85 to +75. If an unsigned exponent is specified, a plus sign is assumed. The exponent causes the value of the constant to be adjusted by the specified power of 10 before the constant is converted to its binary form.

The number is converted to its binary equivalent and is assembled as a fullword or halfword, depending on whether the type is specified as F or H. It is aligned at the proper fullword or halfword boundary if a length is not specified. An implied length of four bytes is assumed for a fullword (F) and two bytes for a halfword (H). However, any length up to and including eight bytes may be specified for either

type of constant by a length modifier, in which case no boundary alignment occurs.

Maximum and minimum values, exclusive of scaling, for fixed-point constants are:

Length	Max	Min
8	$2^{63}-1$	-2^{63}
4	$2^{31}-1$	-2^{31}
2	$2^{15}-1$	-2^{15}
1	$2^{7}-1$	-2^{7}

The binary number occupies the rightmost portion of the field in which it is placed. The unoccupied portion (i.e., the leftmost bits) is filled with the sign; that is, the setting of the bit designating the sign is the setting for the bits in the unused portion of the field. If the value of the number exceeds the length, the necessary leftmost bits are dropped. A negative number is carried in 2s complement form.

If the rightmost portion of the number must be dropped as a result of scale modifiers, rounding occurs. Any duplication factor that is present is applied after the constant is converted to its binary format and assembled into the proper number of bytes.

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

Name	Operation	Operand
CONWRD	DC	3F'658474'

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

Name	Operation	Operand
HALFCON	DC	HS6'-25.93'

The next constant (3.50) is multiplied by 10 to the -2 before being converted to its binary format. The scale modifier reserves 12 bits for the fractional portion.

Name	Operation	Operand
FULLCON	DC	HS12'3.50E-2'

The same constant could be specified as a literal:

Name	Operation	Operand
	AH	7,=HS12'3.50E-2'

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

Name	Operation	Operand
THREECON	DC	FS4'10,25.3,100'

Floating-Point Constants -- E and D: A floating-point constant is written as a decimal number, which may be followed by a decimal exponent, if desired. The number may be an integer, a fraction, or a mixed number (i.e., one with integral and fractional portions). The format of the constant is:

1. The number is written as a signed or unsigned decimal value. The decimal point may be placed before, within, or after the number; it may be omitted, in which case the number is assumed to be an integer. A positive sign is assumed if an unsigned number is specified.

2. The exponent is optional. If specified, it is written immediately after the number as En, where n is an optionally signed decimal value specifying the exponent of the factor 10. The exponent may be in the range -85 to +75. If an unsigned exponent is specified, a plus sign is assumed.

Machine format for a floating-point number is in two parts: (a) the portion containing the exponent, which is sometimes called the characteristic, followed by (b) the portion containing the fraction, which is sometimes called the mantissa. Therefore, the number specified as a floating-point constant must be converted to a fraction before it can be translated into the proper format. For example, the constant 27.35E2 represents the number 27.35 multiplied by 10 to the 2nd. Represented as a fraction, it would be 0.2735 multiplied by 10 to the 4th, the exponent having been

modified to reflect the shifting of the decimal point. The exponent may also be affected by the presence of an exponent modifier (explained under "Operand Subfield 3: Modifiers.") Thus, the exponent is also altered before being translated into machine format. When the constant is converted into the proper exponent and fraction, each is translated into its binary equivalent and arranged in machine floating-point format.

The translated constant is placed in a fullword or a doubleword, depending on whether the type is specified as E or D. It is aligned at the proper word or doubleword boundary if a length is not specified. An implied length of four bytes is assumed for a fullword (E), and eight bytes is assumed for a doubleword (D). However, any length up to and including eight bytes may be specified for either type of constant by a length modifier, in which case no boundary alignment occurs.

Within the portion of the floating-point field allocated to the fraction, the hexadecimal point is assumed to be to the left of the leftmost hexadecimal digit, and the fraction occupies the leftmost portion of the field. The fraction is normalized (no leading hexadecimal 0s), unless scaling is specified. If the rightmost portion of the fraction must be dropped, because of length or scale modifiers, rounding will occur. Negative fractions are carried in true representation, not in the 2s complement form.

Any of the following statements could be used to specify 46.415 as a positive, fullword, floating-point constant; the last is a machine instruction statement with a literal operand. Note that the last two constants contain an exponent modifier.

```
┌─────┬─────────┬──────────────────────┐
│Name │Operation│Operand               │
├─────┼─────────┼──────────────────────┤
│     │DC       │E'46.415'             │
│     │DC       │E'46415E-3'           │
│     │DC       │E'+464.15E-1'         │
│     │DC       │E'+.46415E+2'         │
│     │DC       │EE2'.46415'           │
│     │AE       │6,=EE2'.46415'        │
└─────┴─────────┴──────────────────────┘
```

The following would each be generated as doubleword floating-point constants.

```
┌─────┬─────────┬──────────────────────┐
│Name │Operation│Operand               │
├─────┼─────────┼──────────────────────┤
│FLOAT│DC       │DE+4'+46,-3.729,+473' │
└─────┴─────────┴──────────────────────┘
```

Decimal Constants -- P and Z: A decimal constant is written as a signed or unsigned decimal value. If the sign is omitted, a plus sign is assumed. The decimal point may be written wherever desired, or it may be omitted. Scaling and exponent modifiers may not be specified for decimal constants. The maximum length of a decimal constant is 16 bytes; there is no word boundary alignment.

The placement of a decimal point in the definition does not affect the assembly of the constant in any way because, unlike fixed-point and floating-point constants, a decimal constant is not converted to its binary equivalent. The fact that a decimal constant is an integer, a fraction, or a mixed number is not pertinent to its generation. Further, the decimal point is not assembled into the constant. The programmer may determine proper decimal point alignment, either by defining his data so that the point is aligned or by selecting machine instructions that will operate on the data properly (that is, shift it for purposes of alignment).

If zoned decimal format (Z) is specified, each decimal digit is translated into one byte. The rightmost byte contains the sign, and the rightmost digit. For packed decimal format (P), each pair of decimal digits is translated into one byte. The rightmost digit and the sign are translated into the rightmost byte. The bit configuration for the digits is identical to the configurations for the hexadecimal digits 0-9, as shown in Section 3 under "Hexadecimal Self-Defining Value." For both packed and zoned decimals a plus sign is translated into the hexadecimal digit C, and a minus sign into the digit D.

If an even number of packed decimal digits is specified, one digit will be left unpaired, because the rightmost digit is paired with the sign. Therefore, in the leftmost byte, the leftmost four bits will be set to 0s, and the rightmost four bits will contain the odd (first) digit.

If no length modifier is given, the implied length for either constant is the number of bytes the constant occupies (taking into account the format, sign, and possible addition of 0 bits for packed decimals). If a length modifier is given, this is the handling of the constant:

1. If the constant requires fewer bytes than the length specifies, the necessary number of bytes is added to the left. For zoned decimal format, the decimal digit 0 is placed in each

added byte. For packed decimals, the bits of each added byte are set to 0.

2. If the constant requires more bytes than the length specifies, the necessary number of leftmost digits or pairs of digits is dropped, depending on which format is specified.

Examples of decimal constant definitions:

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|     |DC       |P'+1.25'                 |
|     |DC       |Z'-543'                  |
|     |DC       |Z'79.68'                 |
|     |DC       |PL3'79.68'               |
```

The following statement specifies both packed and zoned decimal constants. The length modifier applies to each constant in the first operand (that is, to each packed decimal constant). Note that a literal could not specify both operands.

```
|Name     |Operation|Operand                |
|---------|---------|-----------------------|
|DECIMALS |DC       |PL8'+25.8,-3874,       |
|         |         |+2.3',Z'+80,-3.72'     |
```

This example illustrates the use of a packed decimal literal:

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|     |UNPK     |OUTAREA,=PL2'+25'        |
```

ADDRESS CONSTANTS: An address constant is a virtual storage address or an absolute expression that is translated into a constant. Address constants are normally used for initializing base registers to facilitate the addressing of storage. Further, they provide the means of communicating between control sections of a multisection program. However, storage addressing and control section communication are also dependent on the USING assembler instruction and the loading of registers. These considerations are illustrated in Section 3 under "Programming With the USING Instruction."

An address constant, unlike other types of constants, is enclosed in parentheses. Two or more address constants specified in an operand are separated by commas, and the entire sequence is enclosed by parentheses. The six types of address constants are A, Y, S, Q, V, and R.

Complex Relocatable Expressions: A complex relocatable expression can only be used to specify an A-type or Y-type address constant. These expressions contain two or more unpaired relocatable terms and/or a negative relocatable term, in addition to any absolute or paired relocatable terms that may be present. In contrast to relocatable expressions, complex relocatable expressions may represent negative values. A complex relocatable expression might consist of external symbols (which cannot be paired) and designate an address in an independent assembly that is to be linked and loaded with the assembly containing the address constant.

A-Type Address Constant: This constant is specified as an absolute, relocatable, or complex relocatable expression. (Remember that an expression may be single term or multiterm.) The value of the expression is calculated as explained in Section 2, with one exception. The maximum value of the expression may be $2^{31}-1$. The implied length of an A-type constant is four bytes, and the value is placed in the rightmost position. The length that may be specified depends on the type of constant. A length of 1-4 bytes may be used for an absolute, relocatable, or complex expression. The programmer is cautioned that address constants which contain relocatable values may cause truncation of the relocated value if fewer than four bytes are allowed for the length of the constant.

In the following examples, the field generated from the statement named ACONST contains four constants, each of which occupies four bytes. Note that one has a location counter reference. The value of the location counter will be the address of the first byte allocated to the fourth constant. The second statement shows the same set of constants specified as literals (i.e., address constant literals).

```
|Name  |Operation|Operand                  |
|------|---------|-------------------------|
|ACONST|DC       |A(108,LOOP,              |
|      |         |END-STRT,*+4096)         |
|      |LM       |4, 7,=A(108,LOOP,        |
|      |         |END-STRT,*+4096)         |
```

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

Y-Type Address Constant: This constant has the characteristics and format of the A-type constant, except for:

1. The constant is assembled as a 16-bit value and is aligned to a halfword boundary.

2. The implied length is two bytes.

3. The maximum length of a Y-type address constant is two bytes. If length specification is used, a length of one or two bytes may be designated for a relocatable or complex expression and one to two bytes for an absolute expression.

WARNING: Specification of relocatable Y-type address constants two or fewer bytes in length should be avoided in programs to be executed under TSS control, since such constants do not permit the assignment of a complete virtual storage address.

Y-type address constants should not be used in programs to be executed under OS control.

S-Type Address Constant: This constant is used to store an address in base-displacement form. It may be specified in two ways:

1. As an absolute or relocatable expression, for example, S(BETA).

2. As two absolute expressions, the first of which represents the displacement value and the second the base register, e.g., S(400(13)).

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

If length specification is used, only two bytes may be specified. S-type address constants may be specified as literal.

Q-Type Address Constant: This constant is used to reserve storage for the offset of an external dummy section. This offset is added to the address of the block of storage allocated to the external dummy sections to address the desired section. The constant is specified as a relocatable symbol which is defined in a DXD or DSECT statement. The implied length of a Q-type address constant is 4 bytes, and the boundary alignment is to a fullword; a length of 1-4 bytes may be specified. No bit length specification is permitted. Q-type address constants may be specified in literals.

In the following example, the constant VALUE is defined in a DXD or DSECT statement. To address VALUE, the value of A is added to the base address of the block of storage allocated for external dummy sections.

```
|Name|Operation|Operand                    |
|----|---------|---------------------------|
|A   |DC       |Q(VALUE)                   |
```

V-Type Address Constant: This constant is used to reserve storage for the address of an external symbol. The constant is specified as one relocatable symbol, which need not be identified by an EXTRN statement. The symbol used is assumed to be an external symbol because it is supplied in a V-type address constant.

Specification of a symbol as the operand of a V-type constant does not constitute a definition of the symbol for this assembly. The implied length of a V-type address constant is four bytes, and boundary alignment is to a fullword. A length modifier may be used to specify a length of from one to four bytes, in which case no such boundary alignment occurs. The programmer is cautioned that address constants which contain relocatable values may cause truncation of the relocated value if fewer than four bytes are allowed for the length of the constant. In the following example 12 bytes will be reserved, because there are three symbols. The values of each assembled constant will be 0 until the program is loaded.

```
|Name  |Operation|Operand                  |
|------|---------|-------------------------|
|VCONST|DC       |V(SORT,MERGE,CALC)       |
```

R-Type Address Constant: This constant, when linking to a reenterable program, supplies the address of the control section that is required by the reenterable program for working storage, variable program data, and address constants unique to the calling program. The programmer need not be aware of the name given to the section by the reenterable program; he needs to know only the desired entry point within the reenterable program. Use of the R-type address constant causes the loader to supply the address of the control section as a function of the entry point name.

```
|Name  |Operation|Operand                  |
|------|---------|-------------------------|
|RCONST|DC       |R(SINE)                  |
```

In the example above, SINE is an entry point, defined by an ENTRY assembler instruction, in a reenterable routine. The prototype control section for the routine is named MATHSECT. The programmer who wishes to use the SINE entry needs to write only the entry name; the loader will provide the address of the prototype section MATHSECT in which SINE has been defined by the ENTRY instruction. The assembler will generate the equivalent of:

Name	Operation	Operand
RCONST	DC	V(MATHSECT)

The rules for specifying an R-type address constant are the same as those for V-type.

The assembler automatically associates the name of an entry point with the name of the CSECT, PSECT, or COM section in which it was defined (see the ENTRY assembler instruction in Section 3).

DS -- Define Storage

The DS instruction reserves areas of storage and assigns names to those areas. The use of this instruction is the preferred way of symbolically defining storage for work areas, input/output areas, etc. The size of a storage area that can be reserved by using the DS instruction is limited only by the maximum value of the location counter.

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

The format of the DS operand is identical to that of the DC operand; exactly the same subfields are employed and are written in the same sequence as in the DC operand. Although the formats are identical, there are two differences in the specification of subfields:

1. The specification of data (subfield 4) is optional in a DS operand, but is mandatory in a DC operand.

2. The maximum length that may be specified for character (C) and hexadecimal (X) field types is 65,535 bytes, rather than 256 bytes.

If a DS operand specifies a constant in subfield 4, and no length is specified in subfield 3, the assembler determines the length of the data and reserves the appropriate amount of storage. It does not assemble the constant. The ability to specify data and have the assembler calculate the storage area that would be required for such data is a convenience to the programmer. If he knows the general format of the data that will be placed in the storage area during program execution, he need only show it as the fourth subfield in a DS operand. The assembler then determines the correct amount of storage to be reserved, thus relieving the programmer of length considerations.

If the DS instruction is named by a symbol, its value attribute is the location of the leftmost byte of the reserved area. The length attribute of the symbol is the length (implied or explicit) of the type of data specified. Should the DS have a series of operands, the length attribute of the symbol is developed from the first item in the first operand. Any positioning required for aligning the storage area to the proper type of boundary is done before the address value is determined. Bytes skipped for alignment are not set to 0.

Each field type (for example, hexadecimal, character, floating point) is associated with certain characteristics (summarized in Appendix C). The associated characteristics will determine which field-type code the programmer selects for the DS operand and what other information he adds, such as a length specification or a duplication factor. For example, both E floating-point fields and F fixed-point fields have implied lengths of four bytes. The leftmost byte is aligned to a fullword boundary. Thus, either code could be specified, if it were desired to reserve four bytes of storage aligned to a fullword boundary. To obtain a length of eight bytes, either the E or F field type could be specified with a length modifier of 8. However, a duplication factor would have to be used to reserve a larger area, because the maximum length specification for either type is eight bytes. Note also that specifying length would cancel any special boundary alignment.

In contrast, packed and zoned decimal (P and Z), character (C), hexadecimal (X), and binary (B) fields have an implied length of one byte. Any of these codes should be accompanied by a length modifier, unless just one byte is to be reserved. Although no alignment occurs, the use of C and X fields permits greater latitude in length specifications; the maximum for either type is 65,535 bytes. (Note that this differs from the maximum for these types in a DC

instruction.) Unless a field of one byte is desired, the length must be either: (a) specified for the C, X, P, Z, or B field types, or (b) the data must be specified (as the fourth subfield), so that the assembler can calculate the length.

To define four 10-byte fields and one 100-byte field, the respective DS statements might be:

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|FIELD|DS       |4CL10                    |
|AREA |DS       |CL100                    |
```

Although FIELD might have been specified as one 40-byte field, the preceding definition has the advantage of providing FIELD with a length attribute of 10. This would be pertinent when using FIELD as an SS machine instruction operand.

Here are more examples of DS statements:

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|ONE  |DS       |CL80 (one 80-byte field; |
|     |         |length attribute, 80)    |
|TWO  |DS       |80C (80 1-byte fields;   |
|     |         |length attribute, 1)     |
|THREE|DS       |6F (six fullwords; length|
|     |         |attribute, 4)            |
|FOUR |DS       |D (one doubleword; length|
|     |         |attribute, 8)            |
|FIVE |DS       |4H (four halfwords;      |
|     |         |length attribute, 2)     |
```

Note: A DS statement causes the storage area to be reserved but not set to 0's. No assumption should be made as to the contents of the reserved area.

Special Uses of the Duplication Factor

FORCING ALIGNMENT: The location counter can be forced to a doubleword, fullword, or halfword boundary by using the appropriate field type (for example, D, F, or H) with a duplication factor of 0. This method may be used to obtain boundary alignment that otherwise would not be provided. For example, the following statements would set the location counter to the next doubleword boundary and then reserve storage space for a 128-byte field (whose leftmost byte would be on a doubleword boundary).

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|     |DS       |0D                       |
|AREA |DS       |CL128                    |
```

DEFINING FIELDS OF AN AREA: A DS instruction with a duplication factor of 0 can be used to assign a name to an area of storage without actually reserving the area. Additional DS and/or DC instructions may then be used to reserve the area and assign names to fields within the area (and generate constants if DC is used).

For example, assume that 80-character records are to be read into an area for processing, and that each record has this format:

```
Positions  5-10    Payroll number
Positions 11-30    Employee name
Positions 31-36    Date
Positions 47-54    Gross wages
Positions 55-62    Withholding tax
```

The following example illustrates how DS instructions might be used to assign a name to the record area, then define the fields of the area and allocate the storage for them. The first statement names the entire area by defining the symbol RDAREA; the statement gives RDAREA a length attribute of 80 bytes, but does not reserve any storage. Similarly, the fifth statement names a 6-byte area by defining the symbol DATE; the three subsequent statements actually define the fields of DATE and allocate storage for them. The second, ninth, and last statements are used for spacing purposes and, therefore, are not named.

```
|Name  |Operation|Operand                  |
|------|---------|-------------------------|
|RDAREA|DS       |0CL80                    |
|      |DS       |CL4                      |
|PAYNO |DS       |CL6                      |
|NAME  |DS       |CL20                     |
|DATE  |DS       |0CL6                     |
|DAY   |DS       |CL2                      |
|MONTH |DS       |CL2                      |
|YEAR  |DS       |CL2                      |
|      |DS       |CL10                     |
|GROSS |DS       |CL8                      |
|FEDTAX|DS       |CL8                      |
|      |DS       |CL18                     |
```

DXD -- Define External Dummy Section

The DXD instruction defines an external dummy section (also referred to as a Pseudo Register). When the assembler encounters a DXD instruction, it computes the amount of storage, and the alignment, required for each operand. This information is available to the loader, which will compute the total length of the external dummy sections. The format for the DXD instruction is:

Name	Operation	Operand
Symbol	DXD	Duplication factor, type,modifiers,constant

The name field, which must be specified, may not contain a sequence symbol. The symbol in the name field is a symbol which usually appears as a Q-type constant in the operand field of a DC statement in the program. The symbol has a length attribute which is calculated by the rules for DC and DS instructions.

The operand of a DXD may be written as if it were a DS instruction operand. The constant in the operand field is only used in determining the byte alignment and total length. Multiple operands and multiple constants are allowed but are only used to determine the length of the work area request. The storage specified in DXD instructions is not initialized to the value specified in the constant.

If more than one external dummy section with the same name is encountered by the loader, it uses the first section in computing length. If two or more identically named external dummy sections have different boundary alignments, the loader uses the first alignment encountered in computing total length.

An external dummy section may also be defined by a Q-type address constant which refers to a DSECT name. The alignment is doubleword and the length is the highest location counter value assigned to that DSECT.

CXD -- Cumulative Length External Dummy Section

The CXD instruction allocates a four-byte, fullword aligned area in storage which will contain, at execution time, the sum of the lengths of all external dummy sections plus the sum of the lengths of all bytes used in aligning external dummy sections. This sum is supplied by the loader. The instruction format is:

Name	Operation	Operand
Symbol or blank	CXD	Must be blank

The CXD instruction may appear anywhere within a program, or, if several programs are being combined, it may appear in each program. The symbol in the name field has the length attribute of 4.

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

ROUTINE A		
Name	**Operation**	**Operand**
ALPHA	DXD	2DL8
BETA	DXD	4FL4
OMEGA	CXD	
	.	
	.	
A1	DC	Q(ALPHA)
A2	DC	Q(BETA)
	.	
	.	
ROUTINE B		
Name	**Operation**	**Operand**
GAMMA	DXD	5D
DELTA	DXD	10F
	.	
	.	
A3	DC	Q(GAMMA)
A4	DC	Q(DELTA)
	.	
	.	
ROUTINE C		
Name	**Operation**	**Operand**
DELTA	DXD	10F
EPSILON	DXD	4H
	.	
	.	
A4	DC	Q(DELTA)
A5	DC	Q(EPSILON)
	.	
	.	

Each of the three routines is requesting a work area. Routine A requests 2 doublewords and 4 fullwords. Routine B request 5 doublewords and 10 fullwords. Routine C requests 10 fullwords and 4 halfwords. At the time these routines are loaded, the sum of the individual lengths, plus the length of any alignment bytes used, will be placed in the fullword storage area allocated by the CXD instruction labeled OMEGA. Routine A can then allocate the amount of storage that is specified in the CXD location. Note that routines B and C may communicate with each other by placing information in the dummy section named DELTA. No other interroutine communication is possible in the example shown because no other corresponding external dummy sections have been defined in routines A, B, or C.

CCW -- Define Channel Command Word

The CCW instruction provides a convenient way to define and generate an 8-byte channel command word aligned at a doubleword boundary. The internal machine format of a channel command word is shown in Table 3. The format of the CCW instruction statement is:

Name	Operation	Operand
Symbol or blank	CCW	Four operands, separated by commas, specifying contents of channel command word in format described in following text

The four operands must appear, written from left to right, as:

1. An absolute expression that specifies the command code. This expression's value is right-justified in byte 1.

2. An absolute, relocatable, or complex relocatable expression specifying the data address. The value of this expression is right-justified in bytes 2-4.

3. An absolute expression that specifies the flags for bits 32-36 and 0's for bits 37-39. The value of this expression is right-justified in byte 5 (byte 6 is set to 0).

4. An absolute expression that specifies the count. The value is right-justified in bytes 7-8.

This is an example of a CCW statement:

Name	Operation	Operand
	CCW	2,READAREA,X'48',80

The form of the third operand sets bits 37-39 to 0, as required. The bit pattern of this operand is:

32-35	36-39
0100	1000

A symbol appearing in the name field of the CCW instruction is assigned the address value of the leftmost byte of the channel command word. The length attribute of the symbol is 8.

Table 3. Channel Command Word

Byte	Bits	Usage
1	0-7	Command code
2-4	8-31	Data address
5	32-36	Flags
	37-39	Must be 0
6	40-47	Set to 0
7-8	48-63	Count

CAUTION: The CCW command does not provide a field large enough to contain a 32-bit address.

LISTING CONTROL INSTRUCTIONS

These instructions identify an assembly listing, provide blank lines in the listing, and designate how much detail is to be included. In no case are instructions or constants generated in the object program.

TITLE -- Identify Assembly Output

TITLE enables the programmer to identify the assembly output, and has the format:

Name	Operation	Operand
Name or blank	TITLE	Sequence of characters, enclosed in apostrophes

The name field may contain a name of from one to four alphabetic or numeric characters, in any combination. The contents of the name field are placed in the assembly output to serve as identification for punching output cards. Only the first TITLE statement in a program may have a name in the name field; for all subsequent TITLE statements the field should be blank.

The operand field may contain up to 100 characters, enclosed in apostrophes. The contents of the operand field are printed at the top of each assembly listing page.

If a character string containing at least one single apostrophe is to be substituted for an operand in a TITLE statement, it must meet the requirements described in Section 7, under "Free Apostrophes."

A program may contain more than one TITLE statement. Each statement provides the heading for pages in the assembly listing that follow it, until another TITLE statement is encountered. Each TITLE statement encountered causes the listing to be advanced to a new page (before the heading is printed). For example, assume that

this statement is the first TITLE statement to appear in a program:

Name	Operation	Operand
	TITLE	'FIRST HEADING'

then this heading appears at the top of each following page: FIRST HEADING

If this statement occurs later in the same program:

Name	Operation	Operand
	TITLE	'A NEW HEADING'

then each following page begins with this heading: A NEW HEADING

EJECT -- Start New Page

EJECT causes the next line of the listing to appear at the top of a new page. This instruction provides a convenient way to separate routines in the program listing. The format is:

Name	Operation	Operand
Blank	EJECT	Not used

If the next line of the listing normally appears at the top of a new page, the EJECT statement has no effect. Two EJECT statements may be used in succession to obtain a completely blank page.

SPACE -- Space Listing

This instruction inserts one or more blank lines in the listing. The format is:

Name	Operation	Operand
Blank	SPACE	Decimal value or blank

A decimal value specifies the number of blank lines to be inserted in the assembly listing; a blank operand causes one blank line to be inserted. If this value exceeds the number of lines remaining on the listing page, the statement will have the same effect as an EJECT statement.

PRINT -- Print Optional Data

The PRINT instruction controls what is assembled into the list data set. Through use of this instruction, lines or data may be included or omitted from the listing created during assembly. The object module created is not affected.

Note: The PRINT assembler instruction is not to be confused with the PRINT command. The instruction determines what lines or data are to be included or not in the listing; the command is necessary to cause the listing to be printed. (A special case, however, is when a nonconversational user elects to have his listing printed immediately on SYSOUT; in this case, any PRINT assembler instructions will affect the output listing just as it would if the list data set had been retained internally.)

The format for the PRINT instruction is:

Name	Operation	Operand
Blank	PRINT	One to three operands

One to three of these operands are used:

ON - Listing is printed.

OFF - No listing is printed.

GEN - All statements generated by macro instructions, backward AGO instructions, or statements in a copied element are printed.

FULLGEN - All statements generated by macro instructions, backward AGO statements, conditional instructions as they are encountered, or statements in a copied element are printed.

NOGEN - Statements generated by macro instructions, backward AGO instructions, or statements in a copied element are not printed. The macro instruction or COPY statement itself will appear in the listing.

DATA - Constants are printed in full in the listing.

NODATA - Only the first eight bytes (16 hexadecimal digits), or the first constant, whichever is shorter, of the assembled data is printed in the listing.

A program may contain any number of PRINT statements. One PRINT controls the printing of the assembly listing until another PRINT is encountered. If no PRINT assembler instruction is issued, the following is assumed:

Name	Operation	Operand
	PRINT	ON,NODATA,GEN

For example, if

Name	Operation	Operand
	DC	XL256'00'

appears in a program, 256 bytes of 0's are assembled. If

Name	Operation	Operand
	PRINT	DATA

is the last PRINT to appear before the DC statement, the 256 bytes of 0's are printed in the assembly listing. However, if

Name	Operation	Operand
	PRINT	NODATA

is the last PRINT before the DC statement, only eight bytes of 0's are printed in the assembly listing.

Whenever an operand is omitted, the assembler assumes the operand specification made in the most recent PRINT statement.

PROGRAM CONTROL INSTRUCTIONS

Program Control instructions:

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

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

ICTL -- Input Format Control

This instruction allows the programmer to alter the normal format of source statements that originate on punched cards; it has no effect on statements entered through a keyboard. ICTL must precede all other statements in the source program that originate from cards; it may be used only once. The format of the ICTL instruction statement is:

Name	Operation	Operand
Blank	ICTL	1-3 decimal values of the form b,e,c

Operand b specifies the begin column of the source statement. It must always be specified, and must be from 1-40, inclusive.

Operand e specifies the end column of the last source statement. The end column, when specified, must be from 41-80, inclusive; when not specified, it is assumed to be 71. The column after the end column is used to indicate whether the next card is a continuation card.

Operand c specifies the continue column of the source statement. The continue column, when specified, must be from 2-40 and must be greater than b. If the continue column is not specified, or if column 80 is specified as the end column, the assembler assumes that there are no continuation cards and that all statements are on a single card.

If no ICTL statement is used in the source program, the assembler assumes that 1, 71, and 16 are the begin, end, and continue columns, respectively. An error in the specification of b causes the statement to be ignored. If e is in error, the end column defaults to 71. If the specification for c is in error, the continue column defaults to 16; if this default conflicts with the begin or end column specifications, continuation cards will not be allowed.

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

Name	Operation	Operand
	ICTL	25

ISEQ -- Input Sequence Checking

ISEQ checks the sequence of source statements that originate from cards. The format of the ISEQ instruction statement is:

Name	Operation	Operand
Blank	ISEQ	Two decimal values of the form 1, r, or blank

The operands 1 and r specify the leftmost and rightmost columns of the field in the input cards to be checked. Operand r must be equal to or greater than operand 1. Columns to be checked must not be between the "begin" and "end" columns.

Sequence checking begins with the first card following the ISEQ statement. Comparison of adjacent cards makes use of the 8-bit internal collating sequence. ISEQ with a blank operand terminates the operation; checking may be resumed with another ISEQ.

Sequence checking is performed only on statements in the source program. Statements inserted by the COPY assembler instruction or generated by a macro instruction are not checked for sequence.

PUNCH -- Punch a Card

The PUNCH assembler instruction is in the language for compatibility with other IBM assembler languages. It will produce a line in the printed listing only.

REPRO -- Reproduce Following Card

The REPRO assembler instruction, too, is in the language for compatibility with other IBM assemblers. It will produce lines in the printed listing only.

ORG -- Set Location Counter

ORG alters the setting of the location counter for the current control section. The format is:

Name	Operation	Operand
Blank	ORG	Expression, or blank

Any symbols in the expression must have been previously defined and must not be external. If the expression is relocatable, it must not be complex; the unpaired relocatable symbol must be defined in the same control section in which the ORG statement appears. Absolute expressions must not be negative.

The location counter is set to the value of the expression in the operand. An absolute value sets the location counter the specified number of virtual storage locations higher than the beginning of the control section, which is always 0. If the operand is omitted, the location counter is set to a location that is one byte higher than the maximum location assigned for the control section up to this point.

An ORG statement must not be used to specify a location below the beginning of the control section in which it appears. For example,

Name	Operation	Operand
	ORG	*-500

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

If the location counter is to be reset to a value that is one byte beyond the highest location yet assigned (in the control section), this statement should be used:

Name	Operation	Operand
	ORG	

If previous ORG statements have reduced the location counter for the purpose of redefining a portion of the current section, an ORG statement with an omitted operand can be used to terminate the effects of such statements and restore the location counter to its highest setting.

LTORG -- Begin Literal Pool

The LTORG instruction causes all literals thus far encountered in the source program to be assembled at appropriate boundaries, starting at the first doubleword boundary following the LTORG statement. The format of the instruction is:

Name	Operation	Operand
Symbol or blank	LTORG	Not used

The symbol represents the address of the first byte of the literal pool; it has a length attribute of 1.

Special Addressing Consideration

Any literals used after the last LTORG statement are placed at the end of the first control section. If there are no LTORG statements, all literals used in the program are placed at the end of the first control section. In these circumstances, the programmer must ensure that the first control section is always addressable. This means that the base address register for the first control section should not be changed through use in subsequent control sections. If the programmer does not want to reserve a register for this purpose, he may place an LTORG statement at the end of each control section, thereby ensuring that all literals appearing in that section are addressable.

In programs which define one or more prototype control sections (PSECTs), literal address constants are placed in a separate literal pool at the end of the first prototype control section. In these circumstances, the programmer also must ensure that the first prototype control section is always addressable.

Duplicate Literals

If duplicate literals occur within the range controlled by one LTORG statement, only one literal is stored. Literals are considered duplicates only if their specifications are identical. A literal will be stored even if it appears to duplicate another literal, if it is an A-type address constant containing any reference to the location counter.

These examples illustrate how the assembler stores pairs of literals, if the placement of each pair is controlled by the same LTORG statement:

```
X'F0'
C'0'        Both are stored

XL3'0'
HL3'0'      Both are stored

A(*+4)
A(*+4)      Both are stored

X'FFFF'
X'FFFF'     Identical; the first is stored
```

CNOP -- Conditional No Operation

This instruction allows the programmer to align an instruction at a specific word boundary. If any bytes must be skipped to align the instruction properly, the assembler ensures an unbroken instruction flow by generating no-operation instructions. This facility is useful in creating calling sequences consisting of a linkage to a subroutine followed by parameters such as channel command words (CCW).

CNOP ensures the alignment of the location counter setting to a halfword, fullword, or doubleword boundary. If the location counter is already properly aligned, the CNOP instruction has no effect. If the specified alignment requires the location counter to be incremented, one to three no-operation instructions are generated. Each uses two bytes.

This is the format of the CNOP instruction:

Name	Operation	Operand
Blank	CNOP	Two absolute expressions of the form b,w

Any symbols used in the expressions in the operand field must have been previously defined.

Operand b specifies at which byte in a word or doubleword the location counter is to be set; b can be 0, 2, 4, or 6. Operand w specifies whether byte b is in a fullword (w=4) or doubleword (w=8). These pairs of b and w are valid:

b,w	Specifies
0,4	Beginning of word
2,4	Middle of word
0,8	Beginning of doubleword
2,8	Second halfword of doubleword
4,8	Middle (third halfword) of doubleword
6,8	Fourth halfword of doubleword

The position in a doubleword that each of these pairs specifies is shown in Figure 10. Note that both 0,4 and 2,4 specify two locations in a doubleword.

Assuming that the location counter is currently aligned at a doubleword boundary, the CNOP instruction in this sequence

Name	Operation	Operand
	CNOP	0,8
	BASR	2,14

has no effect; it is printed in the assembly listing. However, this sequence

Name	Operation	Operand
	CNOP	6,8
	BASR	2,14

causes three branch-on-conditions (NO-OPs) to be generated, thus aligning the BASR instruction at the last halfword in a doubleword, as follows:

Name	Operation	Operand
	BCR	0,0
	BCR	0,0
	BCR	0,0
	BASR	2,14

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

Figure 10. CNOP Alignment

After the BASR instruction is generated, the location counter is at a doubleword boundary, thereby ensuring an unbroken instruction flow.

COPY -- Copy Predefined Source Coding

The COPY instruction obtains source language coding from a library and includes it in the program currently being assembled. This is the format:

Name	Operation	Operand
Blank	COPY	One symbol

The operand is a symbol that identifies the section of coding to be copied. The assembler inserts the requested coding immediately after the COPY statement is encountered. The requested coding must not contain another COPY statement. If identical COPY statements are encountered, the coding they request is brought into the program each time. The control section in effect at the time of the COPY statement is not automatically resumed after the copied element; therefore, if the copied element contains one or more control section statements, all coding following the COPY statement (until another control section statement is encountered) will be considered as part of the last control section in the copied element.

END -- End Assembly

END terminates the assembly of a program. It may also designate a point in the program, or in a separately assembled program, to which control may be transferred after the program is loaded. The END instruction must always be the last statement in the source program, in this format:

Name	Operation	Operand
Blank	END	Relocatable or absolute expression, or blank

The operand specifies the point to which control is transferred when loading is complete. This point is usually the first machine instruction in the program, as shown in this sequence:

Name	Operation	Operand
NAME	CSECT	
AREA	DS	50F
BEGIN	BASR	2,0
	USING	*,2
	.	
	.	
	.	
	END	BEGIN

SECTION 6: INTRODUCTION TO MACRO LANGUAGE

Macro language is an extension of the TSS Assembler language.

The macro language provides the programmer with a convenient way to generate a desired, previously prepared sequence of assembler language statements with only one statement, a macro instruction.

The sequence of generated statements is determined in a previously written definition and, when invoked with a macro instruction, placed in line with his code. This definition may be written only once, stored, and used repeatedly in one or more programs. The definition may be created by a programmer for his sole use, or it may be shared by many users in a system.

This facility simplifies coding programs, reduces the chance of programming errors, and ensures that standard sequences of assembler language statements are used to accomplish desired functions.

An additional facility, called conditional assembly, allows specification of assembler language statements, which may or may not be assembled, depending upon conditions evaluated at assembly time. These conditions are usually tests of values, which may be defined, set, changed, and tested during the course of the assembly itself. The conditional assembly facility can be used without macro instruction statements.

MACRO INSTRUCTION STATEMENT

A macro instruction statement, or simply a macro instruction, is a source program statement that is processed by the assembler, just as assembler language statements are source program statements that are processed by the assembler.

The assembler generates a sequence of assembler language statements for each occurrence of the same macro instruction. The generated statements are then processed like any other assembler language statement.

Three types of macro instructions may be written: positional, keyword, and mixed-mode.

Positional macro instructions permit the programmer to write the operands of a macro instruction in a fixed order. Keyword macro instructions permit him to write the operands in a variable order. Mixed-mode macro instructions permit him to use the features of both positional and keyword macro instructions in the same macro instruction.

MACRO DEFINITION

Before a macro instruction can be assembled, a macro definition must be available to the assembler.

A macro definition is a set of statements that provides the assembler with: (a) the mnemonic operation code and the format of the macro instruction, and (b) the sequence of statements the assembler generates when the macro instruction appears in the source program.

Every macro definition consists of a macro definition header statement, a macro instruction prototype statement, one or more model statements, and a macro definition trailer statement. In addition, COPY statements MEXIT, MNOTE, and conditional assembly instructions may be used.

Header and trailer statements indicate to the assembler the beginning and end of a macro definition.

The prototype statement specifies the mnemonic operation code and format of the macro instruction.

Model statements are used by the assembler to generate the assembler language statements that replace each occurrence of the macro instruction.

COPY statements can be used to copy model statements, MEXIT, MNOTE, or conditional assembly instructions from a system library into a macro definition.

MEXIT can be used to terminate processing of a macro definition.

MNOTE can be used to generate an error message when the rules for writing a particular macro instruction are violated.

The conditional assembly instructions may be used to vary the sequence of statements generated for each occurrence of a macro instruction. Conditional assembly instructions may also be used outside macro definitions, that is, among the assembler language statements in the program.

SOURCES OF MACRO DEFINITIONS

Macro definitions may be secured from three sources and are searched for in the following order:

1. User source statements
2. Supplementary macro library
3. System macro library

The same macro definition may be made available to more than one source program by placing the macro definition in a macro library. A macro library is a collection of macro definitions that can be used by several assembler language programs. Once a macro definition has been placed in a macro library, it may be used by writing its corresponding macro instruction in a source program, if the user has access to that library.

SYSTEM MACRO INSTRUCTIONS

The macro instructions that correspond to macro definitions prepared by IBM are called system macro instructions; they are described in Assembler User Macro Instructions.

VARYING THE GENERATED STATEMENTS

Each time a macro instruction appears in the source program, it is replaced by the same sequence of assembler language statements, unless one or more conditional assembly instructions appear in the macro definition. Conditional assembly instructions are used to vary the number and format of the generated statements.

VARIABLE SYMBOLS

A variable symbol is assigned different values by either the programmer or the assembler. When the assembler uses a macro definition to determine what statements are to replace a macro instruction, variable symbols in the model statement are replaced with the values assigned to them. By changing the values assigned to variable symbols, the programmer can change parts of the generated statements.

A variable symbol is written as an ampersand followed by from one through seven letters and/or digits, the first of which must be a letter. Elsewhere, two ampersands must be used to represent an ampersand.

Types of Variable Symbols

There are three types of variable symbols: symbolic parameters, system variable symbols, and SET symbols. SET symbols are further broken down into SETA, SETB, and SETC symbols. The three types of variable symbols differ in the way they are assigned values.

Assigning Values to Variable Symbols

Symbolic parameters are assigned values by the programmer each time he writes a macro instruction.

System variable symbols are assigned values by the assembler each time it processes a macro instruction.

SET symbols are assigned values by the programmer's use of conditional assembly instructions.

Global SET Symbols

The values assigned to SET symbols in one macro definition may be used to vary the statements that appear in other macro definitions. All SET symbols used for this purpose must be defined by the programmer as global SET symbols. All other SET symbols (that is, those which may be used to vary statements that appear in the same macro definition) must be defined by the programmer as local SET symbols. Local SET symbols and the other variable symbols (that is, symbolic parameters and system variable symbols) are local variable symbols. Global SET symbols are global variable symbols.

SECTION 7: HOW TO PREPARE MACRO DEFINITIONS

A macro definition, which must appear in the source program before any macro instruction that references it, consists of:

- A macro definition header statement.
- A macro instruction prototype statement.
- Zero or more model statements, COPY statements, MEXIT, MNOTE, or conditional assembly instructions.
- A macro definition trailer statement.

Except for MEXIT, MNOTE, and conditional assembly instructions, this section describes all statements that may be used to prepare macro definitions. Conditional assembly instructions are described in Section 9. MEXIT and MNOTE instructions are described in Section 10.

MACRO -- Macro Definition Header

The macro definition header statement, indicating the beginning of a macro definition, must be the first statement, with this format:

Name	Operation	Operand
Blank	MACRO	Blank

MEND -- Macro Definition Trailer

The trailer statement, indicating the end of a macro definition, must be the last statement, with this format:

Name	Operation	Operand
Blank	MEND	Blank

MACRO INSTRUCTION PROTOTYPE

The macro instruction prototype statement, called the prototype statement, specifies the mnemonic operation code and the format of all macro instructions that refer to the macro definition. It must be the second statement of every macro definition, with this format:

Name	Operation	Operand
Symbolic parameter or blank	Symbol	Zero or more symbolic parameters, separated by commas

The symbolic parameters (see "Model Statements," below) are used in the macro definition to represent the name field and operands of the corresponding macro instruction. The name field of the prototype statement may be blank, or it may contain a symbolic parameter.

The symbol in the operation field is the mnemonic operation code that must appear in all macro instructions that refer to a macro definition. The mnemonic operation code must not be the same as the mnemonic operation code of another macro definition in the source program or of an assembler instruction. If a macro definition in a source program has the same mnemonic as a machine operation, the macro definition is used during assembly. If the macro definition is stored in a library, however, the machine operation will be used.

The operand field may contain 0 or more symbolic parameters separated by commas.

This is a prototype statement:

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

STATEMENT FORMAT

The prototype statement may be written in a format different from that used for other assembler language statements. The normal format was described in Section 2, under "Assembler Language Coding Conventions." The alternate format described here allows the programmer to write an operand on each line, and intersperse operands and comments in the statement.

In the alternate format, as in the normal, the name and operation fields must appear on the first line of the statement, and at least one blank must follow the operation field on that line. Both types of statement format may be used in the same prototype statement.

The rules for using the alternate format for punched cards are:

1. If an operand is followed by a comma and a blank, and the column after the end column contains a nonblank character, the operand field may be continued on the next line, starting in the continue column. More than one operand may appear on the same line.

2. Comments may appear after the blank that indicates the end of an operand, up to and including the end column.

3. If, in the next card, the operand field starts after the continue column, the information entered on that line is considered to be comments, and the operand field of the previous card is considered terminated. Any subsequent continuation lines are considered comments.

Note: A prototype statement may be written on as many continuation lines as there are operands.

The rules for using the alternate statement format at a keyboard are:

1. If an operand is followed by a comma and either a blank or a horizontal tab, and the remaining text contains a keyboard continuation character prior to the return, the operand field may be continued on the next line, starting in the first column of the statement area. If this form of continuation is used, an operand must appear on a single line. More than one operand, however, may appear on the same line.

2. Comments may appear after the blank or horizontal tab that indicates the end of an operand, up to the column containing the return.

3. If text contains the keyboard continuation character prior to the return, and the next line starts in the first column of the statement area, the information entered on the next line is considered to be an operand. However, if the first column of the statement area is blank or contains a horizontal tab character, the information entered on the next line is considered to be comments, and the operand field is considered terminated. Any subsequent continuation lines are considered to contain only commands.

4. A prototype statement may be written on as many continuation lines as there are operands.

The following examples illustrate: (a) the normal statement format, (b) the alternate statement format, and (c) the combination of both statement formats.

Name	Operation	Operand	Comments
NAME1	OP1	OPERAND1,OPERAND2,OPERAND3 THIS IS THE NORMAL STATEMENT FORMAT	X X
NAME2	OP2	OPERAND1, THIS IS THE ALOPERAND2,OPERAND3 TERNATE STATEMENT FORMAT	X X
NAME3	OP3	OPERAND1, THIS IS A COMBOPERAND2,OPERAND3,OPERAND4,OPERAND5 INATION OF BOTH STATEMENT FORMATS	X X X

MODEL STATEMENTS

These are the macro definition statements from which the desired sequences of assembler language statements are generated. One or more model statements may follow the prototype statement. A model statement consists of one to four fields (left to right): name, operation, operand, and comments.

The name field may be blank, or it may contain a symbol or symbolic parameter.

The operation field may contain: any machine instruction mnemonic operation code; any assembler mnemonic operation code except COPY, END, ICTL, ISEQ, START, MACRO, and MEND; or the mnemonic operation code of a user macro instruction or of a system macro instruction. Variable symbols may not be used to generate the following assembler mnemonic operation codes: COPY, END, ICTL, ISEQ, REPRO, START, MACRO, and MEND.

Variable symbols may not be used in the name field nor in the operand fields of the following instructions: COPY, END, ICTL, ISEQ, REPRO, and START.

The operand field may contain symbols, symbolic parameters, or other combinations of characters, with the exception noted under "Free Apostrophes" below.

The comments field may contain any combination of characters.

Free Apostrophes

A free apostrophe is:

- A single apostrophe not immediately preceded or followed by another apostrophe.

- The last apostrophe in a sequence containing an odd number of consecutive apostrophes.

A macro instruction operand may include free apostrophes. If a macro definition is to treat such an operand in an apostrophe delimited field, the character positions which may contain free apostrophes must be doubled by using concatenated substring notations. Substring notation is discussed in Section 9.

Apostrophe delimited fields appear as operands of TITLE (Section 5), SETC (Section 9), MNOTE (Section 10) instructions, or as terms of character relations in the operand fields of AIF or SETB instructions (Section 9).

SYMBOLIC PARAMETERS

A symbolic parameter is a type of variable symbol that is assigned values by the programmer when he writes a macro instruction. The programmer may vary statements that are generated for each occurrence of a macro instruction by varying the values assigned to symbolic parameters.

A symbolic parameter consists of an ampersand followed by from one to seven letters and/or digits, the first of which must be a letter. Elsewhere, two ampersands must be used to represent an ampersand. The programmer should not use &SYS as the first four characters of a symbolic parameter.

These are valid symbolic parameters:

```
&READER     &LOOP2
&A23456     &N
&X4F2       &S4
```

These are invalid symbolic parameters:

CARDAREA	(first character is not ampersand)
&256B	(first character after ampersand is not letter)
&AREA2456	(more than seven characters after ampersand)
&BCD%34	(contains special character other than initial ampersand)
&IN AREA	(contains special character (blank) other than initial ampersand)

Any symbolic parameters in a model statement must appear in the prototype statement of the macro definition.

The following is an example of a macro definition. Note that the symbolic parameters in the model statements appear in the prototype statement.

	Name	Operation	Operand
Header		MACRO	
Prototype	&NAME	MOVE	&TO,&FROM
Model	&NAME	ST	2,SAVE
Model		L	2,&FROM
Model		ST	2,&TO
Model		L	2,SAVE
Trailer		MEND	

Symbolic parameters in model statements are replaced by the characters of the macro instruction that correspond to the symbolic parameters.

In the following example, the characters HERE, FIELDA, and FIELDB of the MOVE macro instruction correspond to the symbolic parameters &NAME, &TO, and &FROM, respectively, of the MOVE prototype statement.

Name	Operation	Operand
HERE	MOVE	FIELDA,FIELDB

Any occurrence of the symbolic parameters &NAME, &TO, and &FROM in a model statement will be replaced by the characters HERE, FIEIDA, and FIELDB. If the preceding macro instruction were used in a source program, these assembler language statements would be generated:

Name	Operation	Operand
HERE	ST	2,SAVE
	L	2,FIELDB
	ST	2,FIELDA
	L	2,SAVE

The example below illustrates another use of the MOVE macro instruction, using operands that are different from those that appear in the preceding example.

	Name	Operation	Operand
Macro	LABEL	MOVE	IN,OUT
Generated	LABEL	ST	2,SAVE
Generated		L	2,OUT
Generated		ST	2,IN
Generated		L	2,SAVE

If a symbolic parameter appears in the comments field of a model statement, it is not replaced by the corresponding characters of the macro instruction.

Concatenating Symbolic Parameters With Other Characters or Other Symbolic Parameters

If a symbolic parameter in a model statement is immediately preceded or followed by other characters or another symbolic parameter, the characters that correspond to the symbolic parameter are combined, in the generated statement, with the other characters or the characters that correspond to the other symbolic parameter. This process is called concatenation.

The macro definition, macro instruction, and generated statements in the following example illustrate these rules.

	Name	Operation	Operand
Header		MACRO	
Prototype	&NAME	MOVE	&TY,&P,&TO,&FROM
Model	&NAME	ST&TY	2,SAVEAREA
Model		L&TY	2,&P&FROM
Model		ST&TY	2,&P&TO
Model		L&TY	2,SAVEAREA
Trailer		MEND	
Macro	HERE	MOVE	D,FIELD,A,B
Generated	HERE	STD	2,SAVEAREA
Generated		LD	2,FIELDB
Generated		STD	2,FIELDA
Generated		LD	2,SAVEAREA

Symbolic parameter &TY is used in each of the four model statements to vary the mnemonic operation code of each of the generated statements. The character D in the macro instruction corresponds to symbolic parameter &TY. Since &TY is preceded by other characters (that is, ST and L) in the model statements, the character that corresponds to &TY (that is, D) is concatenated with the other characters to form the operation fields of the generated statements.

Symbolic parameters &P, &TO, and &FROM are used in two of the model statements to vary part of the operand fields of the corresponding generated statements. The characters FIELD, A, and B correspond to the symbolic parameters &P, &TO, and &FROM. Since &P is followed by &FROM in the second model statement, the characters that correspond to them (that is, FIELD and B) are concatenated to form part of the operand field of the second generated statement. Similarly, FIELD and A are concatenated to form part of the operand field of the third generated statement.

If the programmer wants to concatenate a symbolic parameter with a letter, digit, left parenthesis, or period following the symbolic parameter, he must immediately follow the symbolic parameter with a period. A period is optional if the symbolic parameter is to be concatenated with another symbolic parameter, or with a special character other than a left parenthesis or another period that follows it.

If a symbolic parameter is immediately followed by a period, the symbolic parameter and the period are replaced by the characters that correspond to the symbolic parameter. A period that immediately follows a symbolic parameter does not appear in the generated statement.

The following macro definition, macro instruction, and generated statements illustrate these rules.

	Name	Operation	Operand
Header		MACRO	
Prototype	&NAME	MOVE	&P,&S,&R1,&R2
Model	&NAME	ST	&R1,&S.(&R2)
Model		L	&R1,&P.B
Model		ST	&R1,&P.A
Model		L	&R1,&S.(&R2)
Trailer		MEND	
Macro	HERE	MOVE	FIELD,SAVE,2,4
Generated	HERE	ST	2,SAVE(4)
Generated		L	2,FIELDB
Generated		ST	2,FIELDA
Generated		L	2,SAVE(4)

The symbolic parameter &P is used in the second and third model statements to vary part of the operand field of each of the corresponding generated statements. FIELD of the macro instruction corresponds to &P. Since &P is to be concatenated with letters (B and A) in each of the statements, a period immediately follows &P in each of the model statements. The period does not appear in the generated statements.

Similarly, symbolic parameter &S is used in the first and fourth model statements to vary the operand fields of the corresponding generated statements. &S is followed by a period in each of the model statements because it is to be concatenated with a left parenthesis. The period does not appear in the generated statements.

Comments Statements

A model statement may be a comments statement. A comments statement consists of an asterisk in the begin column, followed by comments. The comments statement is used by the assembler to generate an assembler language comments statement, just as other model statements are used by the assembler to generate assembler language statements. No variable symbol substitution is performed.

The programmer may also write, in a macro definition, comments statements that are not to be generated. These statements must have a period in the begin column, immediately followed by an asterisk and the comments.

The first statement in the following example will be used by the assembler to generate a comments statement; the second statement will not.

```
|Name |Operation|Operand                          |
|-----|---------|---------------------------------|
| *THIS STATEMENT WILL BE GENERATED                |
|.*THIS ONE WILL NOT BE GENERATED                  |
```

COPY STATEMENTS

COPY statements may be used to copy model statements and MEXIT, MNOTE, and conditional assembly instructions into a macro definition, just as COPY may be used outside macro definitions to copy source statements into an assembler language program.

The format of this statement is:

```
|Name |Operation|Operand                          |
|-----|---------|---------------------------------|
|Blank|COPY     |A symbol                         |
```

The symbol in the operand field identifies the section of coding to be copied. Any statement that may be used in a macro definition may be part of the copied coding, except MACRO, MEND, COPY, and prototype statements.

Statements to be copied are secured either from a supplementary macro library or the system macro library. If available, the supplementary macro library is first searched for the statements to be copied. If the statements are not in the supplementary macro library, the system macro library is then searched.

A COPY statement is not a model statement, since it is not used by the assembler to generate a COPY statement.

The format of a macro instruction is:

Name	Operation	Operand
Symbol or blank	Mnemonic operation code	0 or more operands, separated by commas

The name field of the macro instruction may contain a symbol. The symbol will not be defined unless a symbolic parameter appears in the name field of the prototype and the same parameter appears in the name field of a generated model statement.

The operation field contains the mnemonic operation code of the macro instruction. The mnemonic must be the same as the mnemonic of a macro definition in the source program or in the macro library. The macro definition with the same mnemonic is used by the assembler to process the macro instruction. If a macro definition in the source program and one in the macro library have the same mnemonic, the macro definition in the source program is used.

The placement and order of the operands in the macro instruction are determined by the placement and order of the symbolic parameters in the operand field of the prototype statement.

MACRO INSTRUCTION OPERANDS

Any combination of up to 255 characters may be used as a macro instruction operand, provided the following rules concerning apostrophes, parentheses, equal signs, ampersands, commas, and blanks are observed.

Paired Apostrophes: An operand may contain one or more quoted strings, or sequences of characters that begin and end with apostrophes and contain even numbers of apostrophes.

The first quoted string starts with the first apostrophe in the operand. Subsequent quoted strings start with the first apostrophe after the apostrophe that ends the previous quoted string.

The first and last apostrophes of a quoted string are called paired apostrophes. In the following example, the first and fourth and the fifth and eighth apostrophes are paired apostrophes.

```
'A''B'C'D'''
```

An apostrophe immediately followed by a letter and immediately preceded by the letter L, which in turn is preceded by any special character other than &, and not within a quoted string, is not considered to be the first apostrophe of a quoted string. An example is (L'X).

Paired Parentheses: There must be an equal number of left and right parentheses; paired parentheses are a left parenthesis and a following right parenthesis, without any intervening parentheses. If there is more than one pair, each additional pair is determined by removing any pairs already recognized and reapplying the above rule for paired parentheses. For instance, in the following example the first and fourth, the second and third, and the fifth and sixth parentheses are paired parentheses: (A(B)C)D(E)

A parenthesis that appears between paired apostrophes is not considered in determining paired parentheses. For instance, the middle parenthesis is not considered in this example: (')')

Equal Signs: An equal sign can occur only as the first character in an operand or between paired apostrophes or paired parentheses. These illustrate the rule:

```
=F'32'
'C=D'
E(F=G)
```

Ampersands: Each sequence of consecutive ampersands must be an even number of ampersands, except as noted under "Inner Macro Instructions," below. This example illustrates the rule:

```
&&123&&&&
```

Commas: A comma indicates the end of an operand, unless it is placed between paired apostrophes or paired parentheses. This example illustrates the rule:

```
(A,B)C',
```

Blanks: A blank indicates the end of the operand field, unless it is placed between paired apostrophes, except as noted under "Statement Format," below. This example illustrates the rule:

```
'A B C'
```

These are valid macro instruction operands:

SYMBOL	A+2
123	(TO(8),FROM)
X'189A'	0(2,3)
*	=F'4096'
L'NAME	AB&&9
'TEN = 10'	'PARENTHESIS IS) '
'QUOTE IS'''	'COMMA IS,'

These are invalid macro instruction operands:

W'NAME	(odd number of apostrophes)
5A)B	(number of left parentheses does not equal number of right parentheses)
(15 B)	(blank not placed between apostrophes)
'ONE' IS '1'	(blank not placed between paired apostrophes)

STATEMENT FORMAT

Macro instructions may use the same alternate format that can be used for prototype statements. If the alternate format is used, a blank does not always indicate the end of the operand field. The alternate format was described in Section 7 under "Macro Instruction Prototype."

OMITTED OPERANDS

If an operand that appears in the prototype statement is omitted from the macro instruction, the comma that would have separated it from the next operand must be present. If the last operand is omitted from a macro instruction, the comma separating the last operand from the previous operand may be omitted.

The following example shows a macro instruction preceded by its corresponding prototype statement; the operands that correspond to the third and sixth operands of the prototype statement are omitted.

Name	Operation	Operand
	EXAMPLE	&A,&B,&C,&D,&E,&F
	EXAMPLE	17,*+4,,AREA,FIELD(6)

If the symbolic parameter that corresponds to an omitted operand is used in a model statement, a null character value replaces the symbolic parameter in the generated statement. In effect, the symbolic parameter is removed.

For example, the first statement below is a model statement that contains the symbolic parameter &C. If the operand that corresponds to &C were omitted from the macro instruction, the second statement would be generated from the model statement.

Name	Operation	Operand
	MVC	THERE&C.25,THIS
	MVC	THERE25,THIS

OPERAND SUBLISTS

An operand of a macro instruction may be a sublist that provides the programmer with a convenient way to refer to: (a) a collection of macro instruction operands as a single operand, or (b) a single operand in a collection of operands.

A sublist consists of one or more operands separated by commas and enclosed in paired parentheses. The entire sublist, including the parentheses, is considered to be one macro instruction operand.

If a macro instruction is written in the alternate statement format, each sublist operand may be written on a separate line; the macro instruction may be written on as many lines as there are operands, including sublist operands.

If &P1 is a symbolic parameter in a prototype statement, and the corresponding operand of a macro instruction is a sublist, &P1(n) may be used in a model statement to refer to the nth operand of the sublist, where n may be a decimal integer. (n may also be any arithmetic expression allowed in the SETA instruction that is described in Section 9.)

If &P1 is a symbolic parameter in a prototype statement, and the corresponding operand of a macro instruction is not a sublist, &P1(n) refers to the operand as a whole, regardless of the value of n.

For example, consider these macro definition, macro instruction, and generated statements:

	Name	Operation	Operand
Header		MACRO	
Prototype		TOTAL	&NUM,&REG,&AREA
Model		L	&REG,&NUM(1)
Model		A	&REG,&NUM(2)
Model		A	&REG,&NUM(3)
Model		ST	&REG,&AREA
Trailer		MEND	
Macro		TOTAL	(A,B,C),6,SUM
Generated		L	6,A
Generated		A	6,B
Generated		A	6,C
Generated		ST	6,SUM

The operand of the macro instruction that corresponds to symbolic parameter &NUM is a sublist. One of the operands in the sublist is referred to in the operand field of three model statements. For example, &NUM(1) refers to the first operand in the sublist corresponding to symbolic parameter &NUM; the first operand is A. Therefore, A replaces &NUM(1) to form part of the generated statement.

Note: When referring to an operand in a sublist, the left parenthesis of the sublist notation must immediately follow the last character, &NUM(1), of the symbolic parameter.

A period should not be placed between the left parenthesis and the last character of the symbolic parameter. A period may be used between these two characters only when the programmer wants to concatenate the left parenthesis with the characters that the symbolic parameter represents. The following example shows what would be generated if a period appeared between the left parenthesis and the last character of the symbolic parameter in the first model statement of the preceding example.

	Name	Operation	Operand
Prototype		TOTAL	&NUM,&REG,&AREA
Model		L	&REG,&NUM.(1)
Macro		TOTAL	(A,B,C),6,SUM
Generated		L	6,(A,B,C) (1)

The symbolic parameter &NUM is used in the operand field of the model statement. The characters (A,B,C) of the macro instruction correspond to &NUM. Since &NUM is immediately followed by a period, &NUM and the period are replaced by (A,B,C); the period is not in the generated statement. The resulting generated statement is an invalid assembler language statement.

INNER MACRO INSTRUCTIONS

A macro instruction may be used as a model statement in a macro definition; then the macro instruction is called an inner macro instruction. A macro instruction that is not used as a model statement is called an outer macro instruction.

Any symbolic parameters used in an inner macro instruction are replaced by the corresponding characters of the outer macro instruction. The macro definition corresponding to an inner macro instruction is used to generate the statements that replace the inner macro instruction.

The TOTAL macro instruction of the previous example is used as an inner macro instruction in the following example. The inner macro instruction contains two symbolic parameters, &S and &T. Characters (X,Y,Z) and J of the macro instruction correspond to &S and &T. Therefore, these characters replace the symbolic parameters in the operand field of the inner macro instruction.

	Name	Operation	Operand
Header		MACRO	
Prototype		COMP	&R1,&R2,&S,&T,&U
Model		SR	&R1,&R2
Model		C	&R1,&T
Model		BNE	&U
Inner		TOTAL	&S,12,&T
Model	&U	A	&R1,&T
Trailer		MEND	
Macro	K	COMP	10,11,(X,Y,Z),J,K
Generated		SR	10,11
Generated		C	10,J
Generated		BNE	K
Generated		L	12,X
Generated		A	12,Y
Generated		A	12,Z
Generated		ST	12,J
Generated	K	A	10,J

The assembler then uses the macro definition that corresponds to the inner macro instruction to generate statements to replace the inner macro instruction. The fourth through seventh statements are generated for the inner macro instruction.

Note: An ampersand that is part of a symbolic parameter is not considered in determining whether a macro instruction operand contains an even number of consecutive ampersands.

LEVELS OF MACRO INSTRUCTIONS

A macro definition that corresponds to an outer macro instruction may contain any number of inner macro instructions. The outer macro instruction is considered a first-level macro instruction. Each inner macro instruction is considered a second-level macro instruction.

The macro definition that corresponds to a second-level macro instruction may contain any number of inner macro instructions. These are considered third-level macro instructions, etc.

The conditional assembly instructions allow the programmer to: (a) define and assign values to SET symbols that can be used to vary parts of generated statements, and (b) vary the sequence of generated statements. Thus the programmer can use these instructions to generate many different sequences of statements from the same macro definition.

Nine of the 12 conditional assembly instructions are described in this section:

LCLA	SETA	AIF
LCLB	SETB	AGO
LCLC	SETC	ANOP

The other three, GBLA, GBLB, and GBLC, are described in Section 10.

All conditional assembly instructions may be used anywhere in an assembler language source program; the primary use is in macro definitions.

The LCLA, LCLB, and LCLC instructions may be used to define and assign initial values to SET symbols.

The SETA, SETB, and SETC instructions may be used to assign arithmetic, binary, and character values, respectively, to SET symbols. The SETB instruction is described below, after the SETA and SETC instructions, because the operand field of the SETB instruction is a combination of the operand fields of the SETA and SETC instructions.

The AIF, AGO, and ANOP instructions may be used in conjunction with sequence symbols to vary the sequence in which statements are processed by the assembler. The programmer can test attributes assigned by the assembler to macro instruction operands to determine which statements are to be processed.

Examples illustrating the use of each conditional assembly instruction are included throughout this section, and a chart summarizing the elements that can be used in each instruction appears at the end of this section.

SET SYMBOLS

SET symbols are one type of variable symbol; symbolic parameters, in Section 7, are another type. SET symbols differ from symbolic parameters in: (a) where they can be used in an assembler language source program, (b) how they are assigned values, and (c) whether the values assigned to them can be changed.

Symbolic parameters can only be used in macro definitions; SET symbols can be used inside and outside macro definitions.

Symbolic parameters are assigned values when the programmer writes a macro instruction; SET symbols are assigned values when he writes SETA, SETB, and SETC conditional assembly instructions.

Each symbolic parameter is assigned a single value for one use of a macro definition, whereas the values assigned to each SETA, SETB, and SETC symbol can change during one use of a macro definition.

Defining SET Symbols

SET symbols must be defined by the programmer before they are used. When a SET symbol is defined, it is assigned an initial value. New values may be assigned by the SETA, SETB, and SETC instructions. A SET symbol is defined when it appears in the operand field of an LCLA, LCLB, or LCLC instruction.

Using Variable Symbols

SETA, SETB, and SETC instructions may be used to change the values assigned to SETA, SETB, and SETC symbols. When a SET symbol appears in the name, operation, or operand field of a statement, the current value of the SET symbol replaces the SET symbol in the statement.

For example, if &A is a symbolic parameter and the corresponding characters of the macro instruction are the symbol HERE, then HERE replaces each occurrence of &A in the macro definition. However, if &A is a SET symbol, the value assigned to &A can be changed, and a different value can replace each occurrence of &A in the macro definition. The same variable symbol may not be used as a symbolic parameter and as a SET symbol in the same macro definition. The rule is illustrated by:

Name	Operation	Operand
&NAME	MOVE	&TO,&FROM

If the statement above is a prototype statement, &NAME, &TO, and &FROM may not be used as SET symbols in the macro definition.

The same variable symbol may not be used as two different types of SET symbols in the same macro definition. Similarly, the same variable symbol may not be used as two different types of SET symbols outside macro definitions. For example, if &A is a SETA symbol in a macro definition, it cannot be used as a SETC symbol in that definition. Similarly, if &A is a SETA symbol outside macro definitions, it cannot be used as a SETC symbol outside macro definitions.

The same variable symbol may be used in two or more macro definitions and outside macro definitions. In that case, the variable symbol will be considered a different variable symbol each time it is used. For example, if &A is a variable symbol (either SET symbol or symbolic parameter) in one macro definition, it can be used as a variable symbol in another definition. Similarly, if &A is a variable symbol in a macro definition, it can be used as a SET symbol outside macro definitions.

All variable symbols may be concatenated with other characters in the same way that symbolic parameters may be concatenated with other characters. The rules for this concatenation are in Section 7, under "Model Statements."

Variable symbols in macro instructions are replaced by the values assigned to them, immediately prior to processing the definition. If a SET symbol is used in the operand field of a macro instruction, and the value assigned to the SET symbol is equivalent to the sublist notation, the operand is not considered a sublist.

ATTRIBUTES

The assembler assigns attributes to macro instruction operands, all literals, and all symbols defined in the program. The six attributes, type, length, scaling, integer, count, and number, are discussed below.

All attributes of macro instruction operands may be referred to within macro definitions. However, only the type, length, scaling, and integer attributes of symbols may be referred to outside macro definitions. Attributes of symbols appearing in the name field of generated statements may not be referred to outside macro definitions; they may be referred to within macro definitions only when the symbol is previously defined. The notations associated with the attributes are:

Type	T'
Length	L'
Scaling	S'
Integer	I'
Count	K'
Number	N'

This is how the programmer may refer to an attribute:

1. In a statement that is outside macro definitions, he may write the notation for the attribute, immediately followed by a symbol; for example, L'NAME refers to the length attribute of the symbol NAME.

2. In a statement that is in a macro definition, he may write the notation for the attribute, immediately followed by a symbolic parameter; e.g., L'&NAME refers to the length attribute of the macro instruction operand that corresponds to symbolic parameter &NAME; L'&NAME(2) refers to the length attribute of the second operand in the sublist that corresponds to symbolic parameter &NAME.

   If a macro instruction operand is a sublist, the programmer may refer to any of the attributes of each member in the list. However, the only attributes defined for the entire sublist are number and count, and any reference to type, length, scaling or integer attributes of an entire sublist parameter is invalid. For example, L'&NAME(2), referring to the length attribute of the second member of a symbolic parameter which is a sublist, would be valid; L'&NAME, referring to the sublist itself, would not be valid.

If an outer macro instruction operand is a symbol or a literal, the operand attributes are the same as the corresponding attributes of the symbol or literal. The symbol must appear in the name field of an assembler language statement or in the operand field of an EXTRN statement. The statement must be outside macro definitions and must not contain any variable symbols. If an inner macro instruction operand is a symbolic parameter, the operand attributes are the same as the attributes of the corresponding outer macro instruction operand.

Type Attribute (T')

The type attribute of a macro instruction operand, a literal, or a symbol is a letter. This list defines the letters that

are used for symbols that name DC and DS statements, and for outer macro instruction operands that are either literals or symbols that name DC or DS statements:

A	A-type address constant, implied length, aligned
B	Binary constant
C	Character constant
D	Long floating-point constant, implied length, aligned
E	Short floating-point constant, implied length, aligned
F	Fullword fixed-point constant, implied length, aligned
G	Fixed-point constant, explicit length
H	Halfword fixed-point constant, implied length, aligned
K	Floating-point constant, explicit length
P	Packed decimal constant
Q	Q-type address constant, implied length, aligned
R	A-, Q-, R-, S-, V-, or Y-type address constant, explicit length
S	S-type address constant, implied length, aligned
V	V-type address constant, implied length, aligned
X	Hexadecimal constant
Y	Y-type address constant, implied length, aligned
Z	Zoned decimal constant
#	R-type address constant, implied length, aligned

These letters are used for symbols and outer macro instruction operands that are symbols which: (a) name statements other than DC or DS statements, or (b) appear in the operand field of an EXTRN statement:

I	Machine instruction
J	Control section name
M	Macro instruction
T	External symbol
W	CCW assembler instruction

These letters are used only for inner and outer macro instruction operands:

N	Self-defining term
O	Omitted operand

U = Undefined is used for: (a) symbols whose attributes are not available, and (b) inner and outer macro instruction operands, symbols, or literals that cannot be assigned any of the above letters. The U is also assigned to symbols that name EQU statements.

The programmer may refer to a type attribute in the operand field of a SETC instruction, or in character relations in the operand fields of SETB or AIF instructions.

Length (L') Scaling (S'), and Integer (I') Attributes

These attributes of macro instruction operands, literals, and symbols are numeric values.

The length attribute of a symbol (or of a macro instruction operand that is a symbol) was described in Section 2. The length attribute of symbols or macro instruction operands with attributes of M, O, T, or U is 0. If N is the type attribute, the length attribute is 1.

Scaling and integer attributes are provided for fixed-point, floating-point, and decimal literals, and for symbols that name fixed-point, floating-point and decimal fields.

Fixed-Point: The scaling attribute of a fixed-point number is the value of the scale modifier subfield of the DC statement. The integer attribute of a fixed-point number is a function of the length and scaling attributes, as I'=8*L'-S'-1.

Floating-Point: The scaling attribute of a floating-point number is the number of hexadecimal 0s in the leftmost portion of the fraction. The integer attribute of a floating-point number is the number of significant hexadecimal digits in the fraction.

Decimal: The scaling attribute of a decimal number is the number of decimal digits to the right of the decimal point. The integer attribute of a decimal number is a function of the length and scaling attributes. For a packed decimal number, I'=2*L'-S'-1; for a zoned decimal number, I'=L'-S'.

Scaling and integer attributes are available for symbols and macro instruction operands only if their type attributes are H, F, and G (fixed-point); D, E, and K (floating-point); or P and Z (decimal).

The programmer may refer to the length, scaling, and integer attributes in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB or AIF instructions.

Count Attribute (K')

The count attribute is a value equal to the number of characters in the macro instruction operand, excluding commas. If the operand is a sublist, the count attribute includes the beginning and ending parentheses and the commas within the sublist. The count attribute of an omitted operand is 0. This attribute is assigned to macro instruction operands only.

If the macro instruction operand contains variable symbols, the characters that replace the variable symbols, rather than the variable symbols themselves, are used to determine the count attribute.

The programmer may refer to the count attribute in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB and AIF instructions that are parts of a macro definition.

Number Attribute (N')

The programmer may refer to the number attribute of macro instruction operands only.

The number attribute is a value equal to the number of operands in an operand sublist. The number of operands in an operand sublist is equal to 1 plus the number of commas that indicate the end of an operand in the sublist.

These examples illustrate the rule:

```
(A,B,C,D,E)     5 operands
(A,,C,D,E)      5 operands
(A,B,C,D)       4 operands
(,B,C,D,E)      5 operands
(A,B,C,D,)      5 operands
(A,B,C,D,,)     6 operands
```

If the macro instruction operand is not a sublist, the number attribute is 1; if the operand is omitted, it is 0.

The programmer may refer to the number attribute in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB and AIF instructions that are part of a macro definition.

Assigning Attributes to Symbols

The integer attribute is computed from the length and scaling attributes.

Fixed-Point: The integer attribute of a fixed-point number is equal to eight times the length attribute of the number, minus the scaling attribute, minus 1; that is, I'=8*L'-S'-1.

Each of the following statements defines a fixed-point field. The length attribute of HALFCON is 2, the scaling attribute is 6, and the integer attribute is 9. The length attribute of ONECON is 4, the scaling attribute is 8, and the integer attribute is 23.

```
|Name    |Operation|Operand                |
|--------|---------|-----------------------|
|HALFCON |DC       |HS6'-25.93'            |
|ONECON  |DC       |FS8'100.3E-2'          |
```

Floating-Point: This integer attribute is equal to two times the difference between the length attribute of the number and 1, minus the scaling attribute; that is, I'=2*(L'-1)-S'.

Each of the following statements defines a floating-point field. The length attribute of SHORT is 4, the scaling attribute is 2, and the integer attribute is 4. The length attribute of LONG is 8, the scaling attribute is 5, and the integer attribute is 9.

```
|Name |Operation|Operand                  |
|-----|---------|-------------------------|
|SHORT|DC       |ES2'46.415'              |
|LONG |DC       |DS5'-3.729'              |
```

Decimal: The integer attribute of a packed decimal number is equal to two times the length attribute of the number, minus the scaling attribute, minus 1; that is, I'=2*L'-S'-1. The integer attribute of a zoned decimal number is equal to the difference between the length attribute and the scaling attribute; that is, I'=L'-S'.

This table identifies the characteristics of the names that are in the statements defining the decimal fields:

Name	Length	Scaling	Integer
FIRST	2	2	1
SECOND	3	0	3
THIRD	4	2	2
FOURTH	3	2	3

```
|Name   |Operation|Operand                |
|-------|---------|-----------------------|
|FIRST  |DC       |P'+1.25'               |
|SECOND |DC       |Z'-543'                |
|THIRD  |DC       |Z'79.68'               |
|FOURTH |DC       |P'79.68'               |
```

SEQUENCE SYMBOLS

Name fields may contain sequence symbols that provide the programmer with the ability to vary the sequence in which statements are processed by the assembler.

A sequence symbol is used in the operand field of an AIF or AGO statement to refer to the statement named by the sequence symbol. A sequence symbol may be used in the name field of any statement that does not contain a symbol or SET symbol; exceptions are prototype statements and MACRO, LCLA, LCLB, LCLC, GBLA, GBLB, and GBLC instructions.

A sequence symbol consists of a period followed by one through seven letters and/or digits; a letter must be first.

These are valid sequence symbols:

```
.READER     .A23456

.LOOP2      .X4F2

.N          .S4
```

These are invalid sequence symbols:

```
CARDAREA    (first character is not
            period)

.246B       (first character after period
            is not letter)

.AREA2345   (more than seven characters
            after period)

.BCD%84     (contains special character
            other than initial period)

.IN AREA    (contains special character
            (blank) other than initial
            period)
```

If a sequence symbol appears in the name field of a macro instruction, and the corresponding prototype statement contains a symbolic parameter in the name field, the sequence symbol does not replace the symbolic parameter wherever it is used in the macro definition.

This example illustrates the rule:

```
   |Name |Operation|Operand                |
   |-----|---------|-----------------------|
   |     |MACRO    |                       |
1  |&NAME|MOVE     |&TO,&FROM              |
2  |&NAME|ST       |2,SAVEAREA             |
   |     |L        |2,&FROM                |
   |     |ST       |2,&TO                  |
   |     |L        |2,SAVEAREA             |
   |     |MEND     |                       |
   |     |         |                       |
3  |.SYM |MOVE     |FIELDA,FIELDB          |
   |     |         |                       |
4  |     |ST       |2,SAVEAREA             |
   |     |L        |2,FIELDB               |
   |     |ST       |2,FIELDA               |
   |     |L        |2,SAVEAREA             |
```

The symbolic parameter &NAME is used in the name field of the prototype statement (1) and the first model statement (2). In the macro instruction (3) a sequence symbol (.SYM) corresponds to the symbolic parameter &NAME that is not replaced by .SYM; therefore, the generated statement (4) does not contain an entry in the same field.

Sequence symbols appearing within a macro definition are, in effect, "local" symbols that may be duplicated in other macro definitions.

LCLA, LCLB, LCLC -- Define SET Symbols

The format of these instructions is:

```
|Name |Operation|Operand                |
|-----|---------|-----------------------|
|Blank|LCLA,    |One or more variable   |
|     |LCLB,    |symbols to be used as  |
|     |or LCLC  |SET symbols, separated |
|     |         |by commas              |
```

The LCLA, LCLB, and LCLC instructions are used to define and assign initial values to SETA, SETB, and SETC symbols. SETA, SETB, and SETC are assigned initial values of 0, 0, and null character value, respectively.

The programmer should not define any SET symbol whose first four characters are &SYS.

An LCLA, LCLB, or LCLC instruction that is part of a macro definition must occur before any of the symbols which it defines are referenced by SET statements.

SETA -- SET Arithmetic

This instruction may be used to assign an arithmetic value to a SETA symbol. The format of this instruction is:

Name	Operation	Operand
SETA symbol	SETA	Arithmetic expression

The expression in the operand field is evaluated as a signed 32-bit arithmetic value which is assigned to the SETA symbol in the name field. The minimum and maximum allowable values of the expression are -2^{31} and $+2^{31}-1$.

The expression may consist of one term or an arithmetic combination of terms. Terms that may be used alone or in combination are self-defining terms, variable symbols, and the length, scaling, integer, count, and number attributes.

A variable symbol may not represent a relocatable symbol (even if paired to form an absolute expression) when used in the operand of a SETA instruction.

Note: A SETC variable symbol may appear in a SETA expression only if the value of the SETC variable is from one to eight decimal digits; decimal digits will be converted to a positive arithmetic value.

The arithmetic operators that may be used to combine the terms of an expression are + (add), - (subtract), * (multiply), and / (divide). An expression may not contain two terms or two operators in succession, nor may it begin with an operator.

These are valid operand fields of SETA instructions:

```
&AREA+X'2D'     I'&N/25
&BETA*10        &EXIT-S'&ENTRY+1
L'&HERE+32      29
```

These are invalid operand fields of SETA instructions:

```
&AREAX'C'       (two terms in succession)

&FIELD+-        (two operators in
                succession)

-&DELTA*2       (begins with operator)

*+32            (begins with operator; two
                operators in succession)

NAME/15         (NAME is not valid term)
```

Evaluation of Arithmetic Expressions

The procedure used to evaluate the arithmetic expression in the operand field of a SETA instruction is the same as that used to evaluate arithmetic expressions in assembler language statements. The only difference between the two types of arithmetic expressions is the terms that are allowed in each expression. This is the evaluation procedure:

1. Each term is given its numerical value.

2. The arithmetic operations are performed from left to right; multiplication and division are performed before addition and subtraction.

3. The computed result is the value assigned to the SETA symbol in the name field.

The arithmetic expression in the operand field of a SETA instruction may contain one or more sequences of arithmetically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence. These examples of SETA instruction operand fields contain parenthesized sequences of terms:

```
(I'&HERE+32)*29

&AREA+X'2D'/(&EXIT-S'&ENTRY+1)

&BETA*10*(I'&N/25/(&EXIT-S'&ENTRY+1))
```

The parenthesized portions of an arithmetic expression are evaluated before the other terms. Parenthesized terms that are within other parenthesized sequences are evaluated first.

Using SETA Symbols

The arithmetic value assigned to a SETA symbol is substituted for the SETA symbol when it is used in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of SETB and AIF instructions. If SETA is used in any other statement, the arithmetic value is converted to an unsigned integer, with leading 0's removed; when the value is 0, it is converted to a single 0.

This illustrates the rule:

```
   ┌─────┬─────────┬──────────────────────┐
   │Name │Operation│Operand               │
   ├─────┼─────────┼──────────────────────┤
   │     │MACRO    │                      │
   │&NAME│MOVE     │&TO,&FROM             │
   │     │LCLA     │&A,&B,&C,&D           │
1  │&A   │SETA     │10                    │
2  │&B   │SETA     │12                    │
3  │&C   │SETA     │&A-&B                 │
4  │&D   │SETA     │&A+&C                 │
   │&NAME│ST       │2,SAVEAREA            │
5  │     │L        │2,&FROM&C             │
6  │     │ST       │2,&TO&D               │
   │     │L        │2,SAVEAREA            │
   │     │MEND     │                      │
   │     │         │                      │
   │HERE │MOVE     │FIELDA,FIELDB         │
   │     │         │                      │
   │HERE │ST       │2,SAVEAREA            │
   │     │L        │2,FIELDB2             │
   │     │ST       │2,FIELDA8             │
   │     │L        │2,SAVEAREA            │
   └─────┴─────────┴──────────────────────┘
```

Statements 1 and 2 assign values +10 and +12 to SETA symbols &A and &B. Statement 3 assigns value -2 to SETA symbol &C. When &C is used in statement 5, value -2 is converted to unsigned integer 2. When &C is used in statement 4, value -2 is used, and &D is assigned value +8. When &D is used in statement 6, value +8 is converted to unsigned integer 8.

This example shows how the value assigned to a SETA symbol in a macro definition may be changed:

```
   ┌─────┬─────────┬──────────────────────┐
   │Name │Operation│Operand               │
   ├─────┼─────────┼──────────────────────┤
   │     │MACRO    │                      │
   │&NAME│MOVE     │&TO&FROM              │
   │     │LCLA     │&A                    │
1  │&A   │SETA     │5                     │
   │&NAME│ST       │2,SAVEAREA            │
2  │     │L        │2,&FROM&A             │
3  │&A   │SETA     │8                     │
4  │     │ST       │2,&TO&A               │
   │     │L        │2,SAVEAREA            │
   │     │MEND     │                      │
   │     │         │                      │
   │HERE │MOVE     │FIELDA,FIELDB         │
   │     │         │                      │
   │HERE │ST       │2,SAVEAREA            │
   │     │L        │2,FIELDB5             │
   │     │ST       │2,FIELDA8             │
   │     │L        │2,SAVEAREA            │
   └─────┴─────────┴──────────────────────┘
```

Statement 1 assigns value +5 to SETA symbol &A. In statement 2, &A is converted to unsigned integer 5. Statement 3 assigns value +8 to &A. In statement 4, therefore, &A is converted to unsigned integer 8, instead of 5.

A SETA symbol used with a symbolic parameter to refer to an operand in an operand sublist must have been assigned a positive value. Expressions that may be used in operand fields of SETA instructions may be used to refer to operands in operand sublists (described in Section 8, under "Operand Sublists").

The following macro definition may be used to add the last operand in an operand sublist to the first operand in an operand sublist and store the result at the first operand. A sample macro instruction and the generated statements follow the macro definition.

```
   ┌─────┬─────────┬──────────────────────┐
   │Name │Operation│Operand               │
   ├─────┼─────────┼──────────────────────┤
   │     │MACRO    │                      │
1  │     │ADDX     │&NUMBER,&REG          │
   │     │LCLA     │&LAST                 │
2  │&LAST│SETA     │N'&NUMBER             │
   │     │L        │&REG,&NUMBER (1)      │
3  │     │A        │&REG,&NUMBER (&LAST)  │
   │     │ST       │&REG,&NUMBER (1)      │
   │     │MEND     │                      │
   │     │         │                      │
4  │     │ADDX     │(A,B,C,D,E),3         │
   │     │         │                      │
   │     │L        │3,A                   │
   │     │A        │3,E                   │
   │     │ST       │3,A                   │
   └─────┴─────────┴──────────────────────┘
```

&NUMBER is the first symbolic parameter in the operand field of the prototype statement (1). The corresponding characters (A,B,C,D,E) of the macro instruction (4) are a sublist. Statement 2 assigns to &LAST arithmetic value +5, which is equal to the number of operands in the sublist. Therefore, in statement 3, &NUMBER(&LAST) is replaced by the fifth operand of the sublist.

SETC -- SET Character

SETC assigns a character value to a SETC symbol. The format of this instruction is:

```
┌─────┬─────────┬──────────────────────┐
│Name  │Operation│Operand               │
├─────┼─────────┼──────────────────────┤
│A SETC│SETC     │One operand, of the form│
│symbol│         │described below       │
└─────┴─────────┴──────────────────────┘
```

The operand field may consist of the type attribute, a character expression, a substring notation, or a concatenation of substring notations and character expressions. A SETA symbol may appear in the operand of a SETC statement. The result is the character representation of the decimal value, unsigned, with leading 0's removed; if the value is 0, one decimal 0 is used.

If a character string containing at least one single apostrophe is to be substituted for an operand in a SETC instruction, it must meet the requirements described in Section 7, under "Free Apostrophes."

Type Attribute

If the character value assigned to a SETC symbol is a type attribute, the attribute must appear alone in the operand field. The following example assigns to &TYPE the type attribute (T') of the macro instruction operand that corresponds to the symbolic parameter &ABC.

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|&TYPE |SETC     |T'&ABC                 |
```

Character Expression

A character expression consists of any combination of characters enclosed in apostrophes; the enclosed value in the operand field is assigned to the SETC symbol in the name field. The maximum value size that can be assigned to a SETC symbol is eight characters.

Evaluation of Character Expressions: This statement assigns the character value AB%4 to SETC symbol &ALPHA:

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|&ALPHA|SETC     |'AB%4'                 |
```

More than one character expression may be concatenated into a single expression by placing a period between the terminating apostrophe of one expression and the opening apostrophe of the next expression. For example, either of the following statements may be used to assign the character value ABCDEF to the SETC symbol &BETA.

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|&BETA |SETC     |'ABCDEF'               |
|&BETA |SETC     |'ABC'.'DEF'            |
```

Two apostrophes must be used to represent an apostrophe that is part of a character expression, as in this statement:

```
|Name   |Operation|Operand                |
|-------|---------|-----------------------|
|&LENGTH|SETC     |'L''SYMBOL'            |
```

Variable symbols may be concatenated with other characters in the operand field of a SETC instruction, according to the general rules for concatenating symbolic parameters with other characters (see Section 7).

If &ALPHA has been assigned the character value AB%4, the following statement may be used to assign the character value AB%4RST to the variable symbol &GAMMA.

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|&GAMMA|SETC     |'&ALPHA.RST'           |
```

Two ampersands must be used to represent an ampersand that is not part of a variable symbol. Only one ampersand becomes part of the character value assigned to the SETC symbol. This statement assigns value HALF& to the SETC symbol &AND:

```
|Name  |Operation|Operand                |
|------|---------|-----------------------|
|&AND  |SETC     |'HALF&&'               |
```

Substring Notation

The character value assigned to a SETC symbol may be a substring character value, to permit the programmer to assign part of a character value to a SETC symbol. If the programmer wants to assign part of a character value to a SETC symbol, he must indicate to the assembler in the operand field of a SETC instruction: (a) the character value itself, and (b) the part of the character value he wants to assign to the SETC symbol. The combination of (a) and (b) is a substring notation; the value is referred to as a substring character value.

Substring notation consists of a character expression, immediately followed by two arithmetic expressions that are separated by a comma and enclosed in parentheses; the expressions may be any expression that is allowed in the operand field of a SETA instruction. The first expression indicates the first character in the character expression that is to be assigned to the SETC symbol in the name field. The second expression indicates the number of consecutive characters in the character expression (starting with the character indicated by the first expression) that are to be assigned to the SETC symbol.

The maximum size substring character value that can be assigned to a SETC symbol is eight characters; 255 characters is the maximum size character expression that the

substring character value can be chosen from.

These are valid substring notations:

```
'&ALPHA'(2,5)
'AB%4'(&AREA+2,1)
'&ALPHA.RST'(6,&A)
'ABC&GAMMA'(&A,&AREA+2)
```

These are invalid substring notations:

'&BETA' (4,6)	(blanks between character value and arithmetic expressions)
'L''SYMBOL' (142-&XYZ)	(only one arithmetic expression)
'AB%1&ALPHA' (8&FIELD*2)	(arithmetic expressions not separated by comma)
'BETA'4,6	(arithmetic expressions not enclosed in parentheses)

Using SETC Symbols

The character value assigned to a SETC symbol is substituted for the SETC symbol when it is used in the name, operation, or operand field of a statement. Consider these macro definition, macro instruction, and generated statements:

```
   |Name    |Operation|Operand              |
   |--------|---------|---------------------|
   |        |MACRO    |                     |
   |&NAME   |MOVE     |&TO,&FROM            |
   |        |LCLC     |&PREFIX              |
1  |&PREFIX |SETC     |'FIELD'              |
   |&NAME   |ST       |2,SAVEAREA           |
2  |        |L        |2,&PREFIX&FROM       |
3  |        |ST       |2,&PREFIX&TO         |
   |        |L        |2,SAVEAREA           |
   |        |MEND     |                     |
   |        |         |                     |
   |HERE    |MOVE     |A,B                  |
   |        |         |                     |
   |HERE    |ST       |2,SAVEAREA           |
   |        |L        |2,FIELDB             |
   |        |ST       |2,FIELDA             |
   |        |L        |2,SAVEAREA           |
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX; in 2 and 3, &PREFIX is replaced by FIELD.

This is how the value assigned to a SETC symbol may be changed in a macro definition:

```
   |Name    |Operation|Operand              |
   |--------|---------|---------------------|
   |        |MACRO    |                     |
   |&NAME   |MOVE     |&TO,&FROM            |
   |        |LCLC     |&PREFIX              |
1  |&PREFIX |SETC     |'FIELD'              |
   |&NAME   |ST       |2,SAVEAREA           |
2  |        |L        |2,&PREFIX&FROM       |
3  |&PREFIX |SETC     |'AREA'               |
4  |        |ST       |2,&PREFIX&TO         |
   |        |L        |2,SAVEAREA           |
   |        |MEND     |                     |
   |        |         |                     |
   |HERE    |MOVE     |A,B                  |
   |        |         |                     |
   |HERE    |ST       |2,SAVEAREA           |
   |        |L        |2,FIELDB             |
   |        |ST       |2,AREAA              |
   |        |L        |2,SAVEAREA           |
```

Statement 1 assigns the character value FIELD to the SETC symbol &PREFIX. Therefore, &PREFIX is replaced by FIELD in statement 2. Statement 3 assigns the character value AREA to &PREFIX. Therefore, &PREFIX is replaced by AREA, instead of FIELD, in statement 4.

Here is an illustration of a substring notation used as the operand field of a SETC instruction:

```
   |Name    |Operation|Operand              |
   |--------|---------|---------------------|
   |        |MACRO    |                     |
   |&NAME   |MOVE     |&TO,&FROM            |
   |        |LCLC     |&PREFIX              |
1  |&PREFIX |SETC     |'&TO'(1,5)           |
   |&NAME   |ST       |2,SAVEAREA           |
2  |        |L        |2,&PREFIX&FROM       |
   |        |ST       |2,&TO                |
   |        |L        |2,SAVEAREA           |
   |        |MEND     |                     |
   |        |         |                     |
   |HERE    |MOVE     |FIELDA,B             |
   |        |         |                     |
   |HERE    |ST       |2,SAVEAREA           |
   |        |L        |2,FIELDB             |
   |        |ST       |2,FIELDA             |
   |        |L        |2,SAVEAREA           |
```

Statement 1 assigns the substring character value FIELD (the first five characters corresponding to symbolic parameter &TO) to the SETC symbol &PREFIX. Therefore, FIELD replaces &PREFIX in statement 2.

Concatenating Substring Notations and Character Expressions: Substring notations may be concatenated with character expressions in the operand field of a SETC instruction. If a substring notation follows a character expression, both may be concatenated by a

period between the terminating apostrophe of the character expression and the opening apostrophe of the substring notation.

For example, if &ALPHA has been assigned the character value AB%4 and &BETA has value ABCDEF, this statement assigns value AB%4BCD to &GAMMA:

```
|Name   |Operation|Operand                  |
|-------|---------|-------------------------|
|&GAMMA |SETC     |'&ALPHA'.'&BETA'(2,3)    |
```

If a substring notation precedes a character expression or another substring notation, both may be concatenated by writing the opening apostrophe of the second item immediately after the closing parenthesis of a substring notation and the opening apostrophe of the next item in the operand field.

If the character value AB%4 is assigned to &ALPHA, and value 5RS to &ABC, either of these statements may be used to assign value AB%45RS to &WORD:

```
|Name  |Operation|Operand                    |
|------|---------|---------------------------|
|&WORD |SETC     |'&ALPHA'(1,4)'&ABC'        |
|&WORD |SETC     |'&ALPHA'(1,4)'&ABC'(1,3)   |
```

When a SETC symbol is used in the operand field of a SETA instruction, the character value assigned to the SETC symbol must be from one to eight decimal digits, which will be converted to a positive arithmetic value when the SETC symbol is replaced in the SETC instruction.

When a SETA symbol is used in the operand field of a SETC statement, the arithmetic value is converted to an unsigned integer with leading 0s removed; if the value is 0, it is converted to a single 0.

SETB -- SET Binary

The SETB instruction assigns binary 0 or 1 to a SETB symbol. The format of this instruction is:

```
|Name   |Operation|Operand                  |
|-------|---------|-------------------------|
|SETB   |SETB     |0 or 1, or logical       |
|symbol |         |expression enclosed in   |
|       |         |parentheses              |
```

A logical expression is evaluated to determine if it is true or false; binary 1 or 0, corresponding to true or false, is then assigned to the SETB symbol in the name field.

A logical expression consists of one term or a logical combination of terms. The terms that may be used alone or in combination with each other are arithmetic relations, character relations, and SETB symbols. The logical operators that combine the terms are AND, OR, and NOT.

A logical expression may contain two operators in succession only if the first operator is either AND or OR, and the second operator is NOT. A logical expression may begin with operator NOT, not with operators AND or OR.

An arithmetic relation consists of two arithmetic expressions connected by a relational operator. A character relation consists of two character values connected by a relational operator. The relational operators are EQ (equal), NE (not equal), LT (less than), GT (greater than), IE (less than or equal), and GE (greater than or equal).

Any expression that may be used in the operand field of a SETB instruction may be used as an arithmetic expression in the operand field of a SETB instruction. Any expression that may be used in the operand field of a SETC instruction may be used as a character value in the operand field of a SETB instruction, including substring and type attribute notations. The maximum size of the character values that can be compared is 255 characters. If the two character values are of different lengths, the shorter one will always compare "less than" the larger one.

Relational operators may be preceded immediately by one or more blanks, a closing parenthesis, or a closing apostrophe. They may be followed immediately by one or more blanks, an opening parenthesis, an opening apostrophe, or an ampersand that introduces a variable symbol. Each relation may or may not be enclosed in parentheses; it must be separated from the logical operators by at least one blank. A relation enclosed in parentheses need not be separated by any blanks from the logical operators; however, blanks may be optionally placed between logical operators and relations enclosed in parentheses.

If a character string containing at least one single apostrophe is to be substituted for a variable symbol in a SETB instruction, it must meet the requirements described in Section 7, under "Free Apostrophes."

These are valid operand fields of SETB instructions:

```
1
(&AREA+2 GT 29)
('AB%4' EQ '&ALPHA')
(T'&ABC NE T'&XYZ)
(T'&P12 EQ 'F')
(&AREA+2 GT 29 OR &B)
(NOT &B AND &AREA+X'2D' GT 29)
```

These are invalid operand fields of SETB instructions:

&B	(not enclosed in parentheses)
(T'&P12 EQ 'F' &B)	(two terms in succession)
('AB%4' EQ 'ALPHA' NOT &B)	(NOT operator must be preceded by AND or OR)
(AND T'&P12 EQ 'F')	(expression begins with AND)

Evaluation of Logical Expressions

This procedure evaluates a logical expression in the operand field of a SETB instruction:

1. Each term (arithmetic relation, character relation, or SETB symbol) is evaluated and given its logical value (true or false).
2. The logical operations are performed from left to right. However, NOTs are performed before ANDs and ANDs before ORs.
3. The computed result is the value assigned to the SETB symbol in the name field.

The logical expression in the operand field of a SETB instruction may contain one or more sequences of logically combined terms that are enclosed in parentheses. A sequence of parenthesized terms may appear within another parenthesized sequence. These are examples:

```
(NOT (&B AND &AREA+X'2D' GT 29))
(&B AND (T'&P12 EQ 'F' OR &B))
```

The parenthesized portion or portions of a logical expression are evaluated before other terms are evaluated. If a sequence of parenthesized terms appears within another parenthesized sequence, the innermost sequence is evaluated first.

Using SETB Symbols

The logical value assigned to a SETB symbol is used for the SETB symbol appearing in the operand field of an AIF instruction or another SETB instruction.

If a SETB symbol is used in the operand field of a SETA instruction, or in arithmetic relations in the operand fields of AIF and SETB instructions, binary 1 (true) and 0 (false) are converted to arithmetic values +1 and +0.

If a SETB symbol is used in the operand field of a SETC instruction, in character relations in the operand fields of AIF and SETB instructions, or in any other statement, binary 1 (true) and 0 (false) are converted to character values 1 and 0. L'&TO EQ 4 is assumed to be true, and S'&TO EQ 0 is assumed to be false.

```
   |Name  |Operation|Operand                  |
   |------|---------|-------------------------|
   |      |MACRO    |                         |
   |&NAME |MOVE     |&TO,&FROM                |
   |      |LCLA     |&A1                      |
   |      |LCLB     |&B1,&B2                  |
   |      |LCLC     |&C1                      |
1  |&B1   |SETB     |(L'&TO EQ 4)             |
2  |&B2   |SETB     |(S'&TO EQ 0)             |
3  |&A1   |SETA     |&B1                      |
4  |&C1   |SETC     |'&B2'                    |
   |      |ST       |2,SAVEAREA               |
   |      |L        |2,&FROM&A1               |
   |      |ST       |2,&TO&C1                 |
   |      |L        |2,SAVEAREA               |
   |      |MEND     |                         |
   |      |         |                         |
   |HERE  |MOVE     |FIELDA,FIELDB            |
   |      |         |                         |
   |HERE  |ST       |2,SAVEAREA               |
   |      |L        |2,FIELDB1                |
   |      |ST       |2,FIELDA0                |
   |      |L        |2,SAVEAREA               |
```

Because the operand field of statement 1 is true, &B1 is assigned the binary value 1. Therefore, the arithmetic value +1 is substituted for &B1 in statement 3. Because the operand field of statement 2 is false, &B2 is assigned the binary value 0; character value 0 is substituted for &B2 in statement 4.

AIF -- Conditional Branch

The AIF instruction conditionally alters the sequence in which source program statements are processed by the assembler. This format is used:

Name	Operation	Operand
Sequence symbol or blank	AIF	Logical expression enclosed in parentheses, immediately followed by sequence symbol

Any logical expression that may be used in the operand field of a SETB instruction may be used in the operand field of an AIF instruction. The sequence symbol in the operand field must immediately follow the closing parenthesis of the logical expression.

The logical expression in the operand field is evaluated to determine if it is true or false. If true, the statement named by the sequence symbol in the operand field is the next statement processed by the assembler. If false, the next sequential statement is processed by the assembler.

The statement named by the sequence symbol may precede or follow the AIF instruction.

If an AIF instruction is in a macro definition, the sequence symbol in the operand field must appear in the name field of a statement in the definition. If AIF appears outside macro definitions, the sequence symbol in the operand field must appear in the name field of a statement outside macro definitions, and any attributes used in the logical expression must be those of previously defined symbols.

These are valid operand fields of AIF instructions:

```
(&AREA+X'2D' GT 29).READER
(T'&P12 EQ 'F').THERE
```

These are invalid operand fields of AIF instructions:

```
(T'&ABC NE T'&XYZ)          (no sequence
                             symbol)
.X4F2                       (no logical
                             expression)
(T'&ABC NE T'&XYZ) .X4F2    (blanks between
                             logical expres-
                             sion and
                             sequence sym-
                             bol)
```

The following macro definition may be used to generate the statements needed to move a fullword fixed-point number from one storage area to another. The statements will be generated only if the type attribute of both storage areas is the letter F.

```
   |Name |Operation|Operand
   |-----|---------|----------------------
   |     |MACRO    |
   |&N   |MOVE     |&T,&F
1  |     |AIF      |(T'&T NE T'&F).END
2  |     |AIF      |(T'&T NE 'F').END
3  |&N   |ST       |2,SAVEAREA
   |     |L        |2,&F
   |     |ST       |2,&T
   |     |L        |2,SAVEAREA
4  |.END |MEND     |
```

The logical expression in the operand field of statement 1 is true if the type attributes of the two macro instruction operands are not equal. If the attributes are equal, the expression is false. Therefore, if the attributes are not equal, statement 4 (named by the sequence symbol .END) is the next statement processed by the assembler. If the attributes are equal, statement 2 (the next sequential statement) is processed.

The expression in the operand of statement 2 is true if the type attribute of the first macro instruction operand is not F. If the attribute is F, the expression is false. Therefore, if the attribute is not F and the expression is true, statement 4 (named by the sequence symbol .END) is the next statement processed by the assembler. If the attribute is F, statement 3 (the next sequential statement) is processed.

If a character string containing at least one single apostrophe is to be substituted for a variable symbol in an AIF statement, it must meet the requirements described in Section 7, under "Free Apostrophes."

Note: AIFB is an instruction, identical in format and function with AIF, which can be used to provide compatibility with the IBM OS Assembler.

AGO -- Unconditional Branch

The AGO instruction unconditionally alters the sequence in which source program statements are processed by the assembler. This is the format:

Name	Operation	Operand
Sequence symbol or blank	AGO	Sequence symbol

The statement named by the sequence symbol in the operand field is the next statement processed by the assembler.

The statement named by the sequence symbol may precede or follow the AGO instruction. If an AGO instruction is part of a macro definition, the sequence symbol in the operand field must appear in the name field of a statement that is in that definition. If an AGO appears outside macro definitions, the sequence symbol in the operand field must appear in the name field of a statement outside macro definitions.

This illustrates the use of the AGO instruction:

```
   |Name   |Operation|Operand                |
   |-------|---------|-----------------------|
   |       |MACRO    |                       |
   |&NAME  |MOVE     |&T,&F                  |
1  |       |AIF      |(T'&T EQ 'F').FIRST    |
2  |       |AGO      |.END                   |
3  |.FIRST |AIF      |(T'&T NE T'&F).END     |
   |&NAME  |ST       |2,SAVEAREA             |
   |       |L        |2,&F                   |
   |       |ST       |2,&T                   |
   |       |L        |2,SAVEAREA             |
4  |.END   |MEND     |                       |
```

Statement 1 determines if the type attribute of the first macro instruction operand is the letter F. If the attribute is F, statement 3 is the next statement processed by the assembler. Otherwise, statement 2 is processed next. Statement 2 indicates to the assembler that statement 4 (named by sequence symbol .END) is the next statement.

Note: AGOB is an instruction, identical in format and function with AGO, which can be used to provide compatibility with the IBM OS Assembler.

ANOP -- Assembly No Operation

The ANOP instruction facilitates conditional and unconditional branching to statements named by symbols or variable symbols.

The format of this instruction is:

```
|Name     |Operation|Operand                |
|---------|---------|-----------------------|
|Sequence |ANOP     |blank                  |
|symbol   |         |                       |
```

If the programmer wants to use an AIF or AGO to branch to another statement, he must place a sequence symbol in the name field of the statement to which he wants to branch. However, if he has already entered a symbol or variable symbol in the name field of that statement, he cannot place a sequence symbol in the name field. Instead, he must place an ANOP before the statement and then branch to the ANOP. This has the same effect as branching to the statement immediately after the ANOP.

This illustrates the use of the ANOP instruction:

```
   |Name   |Operation|Operand                |
   |-------|---------|-----------------------|
   |       |MACRO    |                       |
   |&NAME  |MOVE     |&T,&F                  |
   |       |LCLC     |&TYPE                  |
1  |       |AIF      |(T'&T EQ 'F').FTYPE    |
2  |&TYPE  |SETC     |'E'                    |
3  |.FTYPE |ANOP     |                       |
4  |&NAME  |ST&TYPE  |2,SAVEAREA             |
   |       |L&TYPE   |2,&F                   |
   |       |ST&TYPE  |2,&T                   |
   |       |L&TYPE   |2,SAVEAREA             |
   |       |MEND     |                       |
```

Statement 1 determines if the type attribute of the first macro instruction operand is the letter F. If the attribute is not F, statement 2 is the next statement processed by the assembler. Otherwise, statement 4 is processed next. However, since there is a variable symbol (&NAME) in the name field of statement 4, the required sequence symbol (.FTYPE) cannot be placed in the name field. Therefore, an ANOP (statement 3) must be placed before statement 4.

Then, if the attribute of the first operand is F, the next statement processed by the assembler is the statement named by sequence symbol .FTYPE. The value of &TYPE retains its initial null character value because the SETC instruction is not processed. Since .FTYPE names an ANOP instruction, the next statement processed by the assembler is statement 4, which follows the ANOP.

CONDITIONAL ASSEMBLY ELEMENTS

The elements that can be used in each conditional assembly instruction are summarized in Figure 11. Each row in this chart indicates which elements can be used in a single conditional assembly instruction. Each column indicates the conditional assembly instructions in which a particular element can be used.

The intersection of a column and a row indicates whether an element can be used in an instruction; if so, in what fields of the instruction the element can be used. For example, the intersection of the first row and the first column of the chart indicates that symbolic parameters can be used in the operand field of SETA instructions.

	Variable Symbols				Attributes						
		SET Symbols									
	S.P.	SETA	SETB	SETC	T'	L'	S'	I'	K'	N'	S.S.
SETA	O	N,O	O	O		O	O	O	O	O	
SETB	O	O	N,O	O	O[1]	O[2]	O[2]	O[2]	O[2]	O[2]	
SETC	O	O	O	N,O	O						
AIF	O	O	O	O	O[1]	O[2]	O[2]	O[2]	O[2]	O[2]	N,O
AGO											N,O
ANOP											N

[1]Only in character relations
[2]Only in arithmetic relations

Abbreviations

N is Name	T' is Type Attribute	K'	is Count Attribute
O is Operand	L' is Length Attribute	N'	is Number Attribute
S.P. is Symbolic Parameter	S' is Scaling Attribute	S.S.	is Sequence Symbol
	I' is Integer Attribute		

Figure 11. Conditional Assembly Elements

The extended features of the macro language allow the programmer to:

1. Terminate processing of a macro definition.
2. Generate error messages.
3. Define global SET symbols.
4. Define subscripted SET symbols.
5. Use system variable symbols.
6. Prepare keyword and mixed-mode macro definitions and write keyword and mixed-mode macro instructions.
7. Use other macro definitions.

MEXIT -- MACRO DEFINITION EXIT

The MEXIT instruction, used only in a macro definition, indicates to the assembler that it should terminate processing of a macro definition. The format is:

Name	Operation	Operand
Sequence symbol or blank	MEXIT	Blank

If the assembler processes the MEXIT instruction in a macro definition corresponding to an outer macro instruction, the next statement processed by the assembler is the next statement outside macro definitions.

If the assembler processes the MEXIT in a macro definition corresponding to a second- or third-level macro instruction, the next statement processed is the next statement after the second- or third-level macro instruction in the macro definition.

MEXIT should not be confused with MEND, which indicates the end of a macro definition. MEND must be the last statement of every macro definition, including those that contain one or more MEXIT instructions.

This illustrates the use of the MEXIT instruction:

```
        Name    Operation  Operand
                MACRO
        &NAME   MOVE       &T,&F
1               AIF        (T'&T EQ 'F').OK
2               MEXIT
3       .OK     ANOP
        &NAME   ST         2,SAVEAREA
                L          2,&F
                ST         2,&T
                L          2,SAVEAREA
                MEND
```

Statement 1 determines if the type attribute of the first macro instruction operand is the letter F. If the type is F, the assembler processes the remainder of the macro definition, starting with statement 3. Otherwise, statement 2 is processed next. Statement 2 indicates to the assembler that processing of the macro definition is to be terminated.

MNOTE -- Request for Error Message

The MNOTE instruction requests the assembler to generate an error message or comments line. This instruction is principally used with conditional assembly statements, either in macro definitions or assembler language source code. The format is:

Name	Operation	Operand
Sequence symbol or blank	MNOTE	Severity code, followed by comma, followed by any combination of characters enclosed in apostrophes

If the first operand field is an absolute or null value, the MNOTE causes an error message to be generated, and the value is treated as a severity code. This severity code may be a decimal integer from 0 through 255. A null value is treated as a value 0. A severity code value of 0 remains 0, and the message is not included in the error count. A severity code value of 1 remains 1 and causes the error count to be incremented and a W to appear on the error message. A severity code value of 2 or greater is set to 2 and causes the error count to be incremented and an E to appear on the error message. The maximum number of characters allowed in the error message character string is 226.

An asterisk in the first operand indicates that the MNOTE is to generate a comments line. The error level and count are unaffected by such a statement. A maximum of 100 characters is allowed in the following character string.

When an MNOTE instruction is processed by the assembler, the characters enclosed in apostrophes are provided in the source program listing, in the same way as other error messages. When the assembler encounters an MNOTE instruction in assembler language source code in a conversational assembly, it will interpret processing to prompt the user for corrections.

Two apostrophes must be used to represent an apostrophe enclosed in apostrophes in the operand field of an MNOTE instruction; one apostrophe will be listed for each pair of apostrophes in the operand field.

Any variable symbols used in the operand field of an MNOTE instruction will be replaced by the values assigned to them.

Two ampersands must be used to represent an ampersand that is not part of a variable symbol in the operand field of an MNOTE statement; one ampersand will be listed for each pair of ampersands in the operand field.

If a character string containing at least one single apostrophe is to be substituted for an operand in an MNOTE instruction, it must meet the requirements described in Section 7, under "Free Apostrophe."

This example illustrates the use of MNOTE in a macro definition. The MNOTE instruction is valid anywhere in an assembler language source program.

```
   |Name |Operation|Operand                  |
   |-----|---------|-------------------------|
   |     |MACRO    |                         |
   |&NAME|MOVE     |&T,&F                    |
1  |     |AIF      |(T'&T NE T'&F).M1        |
2  |     |AIF      |(T'&T NE 'F').M2         |
3  |&NAME|ST       |2,SAVEAREA               |
   |     |L        |2,&F                     |
   |     |ST       |2,&T                     |
   |     |L        |2,SAVEAREA               |
   |     |MEXIT    |                         |
4  |.M1  |MNOTE    |'TYPE NOT SAME'          |
   |     |MEXIT    |                         |
5  |.M2  |MNOTE    |'TYPE NOT F'             |
   |     |MEND     |                         |
```

Statement 1 determines if the type attributes of both macro instruction operands are the same. If they are, statement 2 is the next statement processed by the assembler. Otherwise, statement 4 is the next statement processed. Statement 4 causes an error message -- TYPE NOT SAME -- to be printed in the source program listing.

Statement 2 determines if the type attribute of the first macro instruction operand is the letter F. If the type is F, statement 3 is the next statement processed. Otherwise, statement 5 is next. Statement 5 causes an error message -- TYPE NOT F -- to be printed in the source program listing.

GLOBAL AND LOCAL VARIABLE SYMBOLS

These are local variable symbols:

- Symbolic parameters
- Local SET symbols
- System variable symbols

Global SET symbols are the only global variable symbols.

The GBLA, GBLB, and GBLC instructions define global SET symbols, just as the LCLA, LCLB, and LCLC instructions define the SET symbols described in Section 9. SET symbols defined by LCLA, LCLB, and LCLC instructions are referred to as local SET symbols.

Global SET symbols communicate values developed outside a macro definition to statements which are within macro definitions. Communication from macro definitions to macro definitions is also permissible. However, local SET symbols communicate values between statements in the same macro definition, or between statements outside macro definitions.

If a local SET symbol is defined in two or more macro definitions, or in a macro definition and outside macro definitions, the SET symbol is considered to be a different SET symbol in each case. However, a global SET symbol is the same SET symbol in each place it is defined.

A SET symbol must be defined as a global SET symbol in each macro definition in which it is to be used as a global SET symbol. A SET symbol must be defined as a global SET symbol outside macro definitions, if it is to be used outside.

If the same SET symbol is defined as a global SET symbol in one or more places, and as a local SET symbol elsewhere, it is considered the same symbol wherever it is defined as a global SET symbol, and a different symbol wherever it is defined as a local SET symbol.

Defining Local and Global SET Symbols

Local SET symbols are defined when they appear in the operand field of an LCLA, LCLB, or LCLC instruction. These instructions were discussed in Section 9, under "Defining SET Symbols."

Global SET symbols are defined when they appear in the operand field of a GBLA, GBLB, or GBLC instruction. The formats of these instructions are:

Name	Operation	Operand
Blank	GBLA, GBLB, or GBLC	One or more variable symbols to be used as SET symbols, separated by commas

The GBLA, GBLB, and GBLC instructions define global SETA, SETB, and SETC symbols and assign the same initial values as the corresponding types of local SET symbols. However, a global SET symbol is assigned an initial value by only the first GBLA, GBLB, or GBLC instruction processed in which that SET symbol appears. Subsequent GBLA, GBLB, or GBLC instructions processed by the assembler do not affect the value assigned to the SET symbol. The programmer should not define any global SET symbols whose first four characters are &SYS.

If a GBLA, GBLB, or GBLC instruction is part of a macro definition, it must occur before any of the symbols which it defines are referenced by SET statements.

A GBLA, GBLB, or GBLC instruction must precede any macro instructions which generate references to the global instruction.

Using Global and Local SET Symbols

The following examples illustrate the use of global and local SET symbols; each example consists of two parts. The first part is an assembler language source program. The second part shows the statements that would be generated by the assembler after it processed the statements in the source program.

Example 1: The same SET symbol can be used to communicate: (a) values between statements in the same macro definitions, and (b) different values between statements outside macro definitions.

&A is defined as a local SETA symbol in a macro definition (statement 1) and outside macro definitions (statement 4). &A is used twice within the macro definition (statements 2 and 3) and twice outside macro definitions (statements 5 and 6).

Since &A is a local SETA symbol in the macro definition and outside macro definitions, it is one SETA symbol in the macro definition, and another SETA symbol outside macro definitions. Therefore, statement 3 (in the macro definition) does not affect the value used for &A in statements 5 and 6 (outside macro definitions).

	Name	Operation	Operand
		MACRO	
	&NAME	LOADA	
1		LCLA	&A
2	&NAME	LR	15,&A
3	&A	SETA	&A+1
		MEND	
4		LCLA	&A
	FIRST	LOADA	
5		LR	15,&A
		LOADA	
6		LR	15,&A
		END	FIRST
	FIRST	LR	15,0
		LR	15,0
		LR	15,0
		LR	15,0
		END	FIRST

Example 2: This example illustrates the use of global SET symbols to communicate a value developed outside a macro definition to a statement which is within a macro definition.

	Name	Operation	Operand
		MACRO	
	&NAME	LOADA	
1		GBLA	&A
2	&NAME	LR	15,&A
		MEND	
3		GBLA	&A
	FIRST	LOADA	
4	&A	SETA	&A+1
5		LR	15,&A
		LOADA	
6	&A	SETA	&A+1
7		LR	15,&A
		END	FIRST
	FIRST	LR	15,0
		LR	15,1
		LR	15,1
		LR	15,2
		END	FIRST

&A is defined as a global SETA symbol in a macro definition (statement 1) and outside macro definitions (statement 3). &A is used once within the macro definition

(statement 2) and 4 times outside macro definitions (statements 4, 5, 6, and 7).

Global SET symbols may be used to communicate values from macro definitions to macro definitions, or from statements outside of macro definitions to statements within macro definitions. Global SET symbols may not be used to communicate values from within macro definitions to statements outside macro definitions.

Example 3: The same SET symbol can be used to communicate (a) values between statements in one macro definition, and (b) different values between statements in different macro definitions.

&A is defined as a local SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statements 2, 3, 5, and 6).

Since &A is a local SETA symbol in each macro definition, it is one SETA symbol in one macro definition, and another SETA symbol in the other. Therefore, statement 3 (in one macro definition) does not affect the value used for &A in statement 5 (in the other macro definition). Similarly, statement 6 does not affect the value used for &A in statement 2.

```
   |Name  |Operation|Operand                |
   |------|---------|-----------------------|
   |      |MACRO    |                       |
   |&NAME |LOADA    |                       |
1  |      |LCLA     |&A                     |
2  |&NAME |LR       |15,&A                  |
3  |&A    |SETA     |&A+1                   |
   |      |MEND     |                       |
   |      |         |                       |
   |      |MACRO    |                       |
   |      |LOADB    |                       |
4  |      |LCLA     |&A                     |
5  |      |LR       |15,&A                  |
6  |&A    |SETA     |&A+1                   |
   |      |MEND     |                       |
   |      |         |                       |
   |FIRST |LOADA    |                       |
   |      |LOADB    |                       |
   |      |LOADA    |                       |
   |      |LOADB    |                       |
   |      |END      |FIRST                  |
   |------|---------|-----------------------|
   |FIRST |LR       |15,0                   |
   |      |LR       |15,0                   |
   |      |LR       |15,0                   |
   |      |LR       |15,0                   |
   |      |END      |FIRST                  |
```

Example 4: A SET symbol can be used to communicate values between statements that are parts of two different macro definitions.

&A is defined as a global SETA symbol in two different macro definitions (statements 1 and 4). &A is used twice within each macro definition (statements 2, 3, 5 and 6).

```
   |Name  |Operation|Operand                |
   |------|---------|-----------------------|
   |      |MACRO    |                       |
   |&NAME |LOADA    |                       |
1  |      |GBLA     |&A                     |
2  |&NAME |LR       |15,&A                  |
3  |&A    |SETA     |&A+1                   |
   |      |MEND     |                       |
   |      |         |                       |
   |      |MACRO    |                       |
   |      |LOADB    |                       |
4  |      |GBLA     |&A                     |
5  |      |LR       |15,&A                  |
6  |&A    |SETA     |&A+1                   |
   |      |MEND     |                       |
   |      |         |                       |
   |FIRST |LOADA    |                       |
   |      |LOADB    |                       |
   |      |LOADA    |                       |
   |      |LOADB    |                       |
   |      |END      |FIRST                  |
   |------|---------|-----------------------|
   |FIRST |LR       |15,0                   |
   |      |LR       |15,1                   |
   |      |LR       |15,2                   |
   |      |LR       |15,3                   |
   |      |END      |FIRST                  |
```

Since &A is a global SETA in each macro definition, it is the same SETA symbol in each. Therefore, statement 3 (in one macro definition) affects the value used for &A in statement 5 (in the other macro definition). Similarly, statement 6 affects the value used for &A in statement 2.

Example 5: The same SET symbol can be used to communicate: (a) values between statements in two different macro definitions, and (b) different values between statements outside macro definitions.

&A is defined as a global SETA in two different macro definitions (statements 1 and 4), but it is defined as a local SETA outside macro definitions (statement 7). &A is used twice within each macro definition and twice outside macro definitions (statements 2, 3, 5, 6, 8 and 9).

Since &A is a global SETA in each macro definition, it is the same SETA in each. However, since &A is a local SETA outside macro definitions, it is a different SETA symbol outside.

Therefore, statement 3 (in one macro definition) affects the value used for &A in statement 5 (in the other macro definition), but it does not affect the value used for &A in statements 8 and 9 (outside

macro definitions). Similarly, statement 6 affects the value used for &A in statement 2, but it does not affect the value used for &A in statements 8 and 9.

```
   |Name |Operation|Operand                |
   |-----|---------|-----------------------|
   |     |MACRO    |                       |
   |&NAME|LOADA    |                       |
1  |     |GBLA     |&A                     |
2  |&NAME|LR       |15,&A                  |
3  |&A   |SETA     |&A+1                   |
   |     |MEND     |                       |
   |     |         |                       |
   |     |MACRO    |                       |
   |     |LOADB    |                       |
4  |     |GBLA     |&A                     |
5  |     |LR       |15,&A                  |
6  |&A   |SETA     |&A+1                   |
   |     |MEND     |                       |
   |     |         |                       |
7  |     |LCLA     |&A                     |
   |FIRST|LOADA    |                       |
   |     |LOADB    |                       |
8  |     |LR       |15,&A                  |
   |     |LOADA    |                       |
   |     |LOADB    |                       |
9  |     |LR       |15,&A                  |
   |     |END      |FIRST                  |
   |-----|---------|-----------------------|
   |FIRST|LR       |15,0                   |
   |     |LR       |15,1                   |
   |     |LR       |15,0                   |
   |     |LR       |15,2                   |
   |     |LR       |15,3                   |
   |     |LR       |15,0                   |
   |     |END      |FIRST                  |
```

Subscripted SET Symbols

Both global and local SET symbols may be defined as subscripted SET symbols. The local SET symbols defined in Section 9 were all unsubscripted SET symbols.

Subscripted SET symbols provide the programmer with a convenient way to use one SET symbol plus a subscript to refer to many arithmetic, binary, or character values. A subscripted symbol consists of a SET symbol immediately followed by a subscript that is enclosed in parentheses. The subscript may be any arithmetic expression that is allowed in the operand field of a SETA statement. The subscript of a variable symbol may not be an attribute of a subscripted variable symbol.

These are valid subscripted SET symbols:

```
&READER(17)
&A23456(&S4)
&X4F2(25+&A2)
```

These are invalid subscripted SET symbols:

```
&X4F2        (no subscript)
 (25)        (no SET symbol)
&X4F2 (25)   (subscript does not
              immediately
              follow SET symbol)
```

Defining Subscripted SET Symbols: If the programmer wants to use a subscripted SET symbol, he must write, in a GBLA, GBLB, GBLC, LCLA, LCLB, or LCLC instruction, a SET symbol immediately followed by a decimal integer enclosed in parentheses. The decimal integer, called a dimension, indicates the number of SET variables associated with the SET symbol. Every variable associated with a SET symbol is assigned an initial value that is the same as the initial value assigned to the corresponding type of unsubscripted SET symbol.

If a subscripted SET symbol is defined as global, the same dimension must be used with the SET symbol each time it is defined as global. The maximum dimension that can be used with SETA, SETB, and SETC symbols is 255.

The following statements define the global SET symbols &SBOX, &WBOX, and &PSW, and the local SET symbol &TSW. &SBOX has 50 arithmetic variables associated with it, &WBOX has 20 character variables, and &PSW and &TSW have 230 binary variables each.

```
|Name |Operation|Operand                |
|-----|---------|-----------------------|
|     |GBLA     |&SBOX(50)              |
|     |GBLC     |&WBOX(20)              |
|     |GBLB     |&PSW(230)              |
|     |LCLB     |&TSW(230)              |
```

Using Subscripted SET Symbols: After the programmer has associated a number of SET variables with a SET symbol, he may assign values to each of the variables and use them in other statements.

If the statements in the previous example were parts of a macro definition (and &A was defined as a SETA in the same definition), these statements could be parts of the same macro definition.

```
   |Name     |Operation|Operand              |
   |---------|---------|---------------------|
1  |&A       |SETA     |5                    |
2  |&PSW(&A) |SETB     |(6 LT 2)             |
3  |&TSW(9)  |SETB     |(&PSW(&A))           |
4  |         |A        |2,=F'&SBOX(45)'      |
5  |         |CLI      |AREA,C'&WBOX(17)'    |
```

Statement 1 assigns the arithmetic value 5 to the unsubscripted SETA symbol &A. Statements 2 and 3 then assign the binary

value 0 to subscripted SETB symbols &PSW(5) and &TSW(9). Statements 4 and 5 generate statements that add the value assigned to &SBOX(45) to general register 2, and compare the value assigned to &WBOX(17) to the value stored at AREA.

SYSTEM VARIABLE SYMBOLS

System variable symbols are variable symbols to which the assembler automatically assigns values. The five local system variable symbols (&SYSNDX, &SYSECT, &SYSSTYP, &SYSLIST, and &SYSPSCT) may be used in the name, operation, and operand fields of statements in macro definitions, but not in statements outside macro definitions. They may not be defined as symbolic parameters or SET symbols, nor may they be assigned values by SETA, SETB, and SETC. &SYSDATE and &SYSTIME are global system variable symbols to which the assembler assigns values. They may be used in macro definitions and in assembler language source code. They may not be defined or altered.

&SYSNDX -- Macro Instruction Index

The system variable symbol &SYSNDX may be concatenated with other characters to create unique names for statements generated from the same model statement.

&SYSNDX is assigned the 4-digit number 0001 for the first macro instruction processed by the assembler, and it is incremented by one for each subsequent inner and outer macro instruction processed.

If &SYSNDX is used in a model statement, SETC or MNOTE instruction, or a character relation in a SETB or AIF instruction, the value substituted for &SYSNDX is the 4-digit number of the macro instruction being processed, including leading 0s. If &SYSNDX appears in arithmetic expressions (in the operand field of a SETA instruction), the value used for &SYSNDX is an arithmetic value.

Throughout one use of a macro definition, the value of &SYSNDX may be considered a constant, independent of any inner macro instruction in that definition.

The accompanying example illustrates these rules. It is assumed that the first macro instruction processed, OUTER1, is the 106th macro instruction processed by the assembler. Statement 7 is the 106th macro instruction processed. Therefore, &SYSNDX is assigned 0106 for that macro instruction. The number 0106 is substituted for &SYSNDX when it is used in statements 4 and 6. Statement 4 assigns the character value 0106 to the SETC symbol &NDXNUM and statement 6 creates the unique name B0106.

Statement 5 is the 107th macro instruction processed. &SYSNDX is assigned the number 0107 for that macro instruction. 0107 is substituted for &SYSNDX in statements 1 and 3. 0106 is substituted for the global SETC symbol &NDXNUM in statement 2.

Statement 8 is the 108th macro instruction processed. Each occurrence of &SYSNDX is replaced by 0108. Statement 6 creates the unique name B0108.

```
   |Name     |Operation|Operand             |
   |---------|---------|--------------------|
   |         |MACRO    |                    |
   |         |INNER1   |                    |
   |         |GBLC     |&NDXNUM             |
1  |A&SYSNDX |SR       |2,5                 |
   |         |CR       |2,5                 |
2  |         |BE       |B&NDXNUM            |
3  |         |B        |A&SYSNDX            |
   |         |MEND     |                    |
   |         |         |                    |
   |         |MACRO    |                    |
   |&NAME    |OUTER1   |                    |
   |         |GBLC     |&NDXNUM             |
4  |&NDXNUM  |SETC     |'&SYSNDX'           |
   |&NAME    |SR       |2,4                 |
   |         |AR       |2,6                 |
5  |         |INNER1   |                    |
6  |B&SYSNDX |S        |2,=F'1000'          |
   |         |MEND     |                    |
   |---------|---------|--------------------|
7  |ALPHA    |OUTER1   |                    |
8  |BETA     |OUTER1   |                    |
   |---------|---------|--------------------|
   |ALPHA    |SR       |2,4                 |
   |         |AR       |2,6                 |
   |A0107    |SR       |2,5                 |
   |         |CR       |2,5                 |
   |         |BE       |B0106               |
   |         |B        |A0107               |
   |B0106    |S        |2,=F'1000'          |
   |BETA     |SR       |2,4                 |
   |         |AR       |2,6                 |
   |A0109    |SR       |2,5                 |
   |         |CR       |2,5                 |
   |         |BE       |B0108               |
   |         |B        |A0109               |
   |B0108    |S        |2,=F'1000'          |
```

When statement 5 processes the 108th macro instruction, statement 5 becomes the 109th macro instruction processed. Each occurrence of &SYSNDX is replaced by 0109. Statement 1 creates the unique name A0109.

&SYSECT -- Current Control Section

The system variable symbol &SYSECT represents the name of the control section in which a macro instruction appears. For each inner and outer macro instruction processed, &SYSECT is assigned a value that is the name of the control section in which the macro instruction appears.

When &SYSECT is used in a macro definition, the value substituted for &SYSECT is the name of the last CSECT, DSECT, or START statement that occurs before the macro instruction. If no named CSECT, DSECT, or START statements occur before a macro instruction, &SYSECT is assigned a null character value for that macro instruction. CSECT or DSECT statements processed in a macro definition affect the value of &SYSECT of any subsequent inner macro instructions in that definition, and for any other outer and inner macro instructions. Throughout the use of a macro definition, the value of &SYSECT may be considered a constant, independent of any CSECT or DSECT statements or inner macro instructions in that definition.

The next example illustrates these rules.

```
   |Name     |Operation|Operand             |
   |---------|---------|--------------------|
   |         |MACRO    |                    |
   |         |INNER    |&INCSECT            |
1  |&INCSECT |CSECT    |                    |
2  |         |DC       |A(&SYSECT)          |
   |         |MEND     |                    |
   |         |         |                    |
   |         |MACRO    |                    |
   |         |OUTER1   |                    |
3  |CSOUT1   |CSECT    |                    |
   |         |DS       |100C                |
4  |         |INNER    |INA                 |
5  |         |INNER    |INB                 |
6  |         |DC       |A(&SYSECT)          |
   |         |MEND     |                    |
   |         |         |                    |
   |         |MACRO    |                    |
   |         |OUTER2   |                    |
7  |         |DC       |A(&SYSECT)          |
   |         |MEND     |                    |
   |---------|---------|--------------------|
8  |MAINPROG |CSECT    |                    |
   |         |DS       |200C                |
9  |         |OUTER1   |                    |
10 |         |OUTER2   |                    |
   |---------|---------|--------------------|
   |MAINPROG |CSECT    |                    |
   |         |DS       |200C                |
   |CSOUT1   |CSECT    |                    |
   |         |DS       |100C                |
   |INA      |CSECT    |                    |
   |         |DC       |A(CSOUT1)           |
   |INB      |CSECT    |                    |
   |         |DC       |A(INA)              |
   |         |DC       |A(MAINPROG)         |
   |         |DC       |A(INB)              |
```

Statement 8 is the last CSECT, DSECT, or START statement processed before statement 9. Therefore, &SYSECT is assigned the value MAINPROG for macro instruction OUTER1 in statement 9. MAINPROG is substituted for &SYSECT when it appears in statement 6.

Statement 3 is the last CSECT, DSECT, or START statement processed before statement 4. Therefore, &SYSECT is assigned CSOUT1 for macro instruction INNER in statement 4. CSOUT1 is substituted for &SYSECT when it appears before statement 2.

Statement 1 is used to generate a CSECT statement for statement 4. This is the last CSECT, DSECT, or START statement that appears before statement 5. Therefore, &SYSECT is assigned INA for macro instruction INNER in statement 5. INA is substituted for &SYSECT when it appears in statement 2.

Statement 1 is used to generate a CSECT statement for statement 5. This is the last CSECT, DSECT, or START statement that appears before statement 10. Therefore, &SYSECT is assigned INB for macro instruction OUTER2 in statement 10. INB is substituted for &SYSECT when it appears in statement 7.

&SYSSTYP -- Current Control Section Type

The system variable symbol &SYSSTYP represents the mnemonic operation which defined the name of the control, prototype, dummy, or common section in which a macro instruction appears.

When &SYSSTYP is used in a macro definition, its substituted value is one of the character string values CSECT, PSECT, DSECT, or COM. The value used will correspond to the mnemonic operation which defined the name represented by system variable symbol &SYSECT. A section defined by the START instruction is considered to be a CSECT. &SYSSTYP enables the macro language programmer to resume the current control section after it is interrupted, without requiring the section type to be supplied as a parameter.

```
   |Name     |Operation|Operand             |
   |---------|---------|--------------------|
   |         |MACRO    |                    |
   |&LOC     |OUTLINE  |&RTNE               |
1  |PRO      |PSECT    |                    |
2  |&LOC     |DC       |A(&RTNE)            |
3  |&SYSECT  |&SYSSTYP |                    |
   |         |MEND     |                    |
   |---------|---------|--------------------|
   |MAINPROG |CSECT    |                    |
   |         |I        |1,ADDR              |
   |ADDR     |OUTLINE  |SUBR                |
   |         |I        |2,X                 |
   |---------|---------|--------------------|
   |MAINPROG |CSECT    |                    |
   |         |I        |1,ADDR              |
   |PRO      |PSECT    |                    |
   |ADDR     |DC       |A(SUBR)             |
   |MAINPROG |CSECT    |                    |
   |         |L        |2,X                 |
```

In the example above, statement 1 is a model statement to initiate the prototype control section PRO. Statement 2 generates an address constant of the entry point of subroutine SUBR. Statement 3 reinstates the interrupted control section MAINPROG. The macro instruction does not require that the name or type of the original control section be stated as a parameter.

&SYSLIST -- Macro Instruction Operand

System variable symbol &SYSLIST provides the programmer with an alternative to symbolic parameters for referring to macro instruction operands.

&SYSLIST and symbolic parameters may be used in the same macro definition. &SYSLIST also may be used in place of symbolic parameters.

&SYSLIST(n) may be used to refer to the nth macro instruction operand. In addition, if the nth operand is a sublist, then &SYSLIST(n,m) may be used to refer to the mth operand in the sublist, where n and m may be any arithmetic expressions, greater than zero, allowed in the operand field of a SETA statement.

The type, length, scaling, integer, and count attributes of &SYSLIST(n) and &SYSLIST(m,m), and the number attributes of &SYSLIST(n) and &SYSLIST, may be used in conditional assembly instructions. N'&SYSLIST may be used to refer to the total number of operands in a macro instruction statement. N'&SYSLIST(n) may be used to refer to the number of operands in a sublist. If the nth operand is omitted, N' is 0; if the nth operand is not a sublist, N' is 1.

The following procedure is used to evaluate N'&SYSLIST:

A sublist is considered to be one operand.

The number of operands equals 1 plus the number of commas indicating the end of an operand.

Attributes were discussed in Section 7 under "Attributes."

	Name	Operation	Operand
Header		MACRO	
Prototype		TOTAL	NUM, REG, AREA
Model		L	&SYSLIST(2), &SYSLIST(1,1)
Model		A	&SYSLIST(2), &SYSLIST(1,2)
Model		A	&SYSLIST(2), &SYSLIST(1,3)
Model		ST	&SYSLIST(2), &SYSLIST(3)
Trailer		MEND	
Macro		TOTAL	(A,B,C),6,SUM
Generated		L	6,A
Generated		A	6,B
Generated		A	6,C
Generated		ST	6,SUM

The generated statements in the above examples are exactly the same as the generated statements of the example under "Operand Sublists" of Section 8, "How to Write Macro Instructions."

&SYSPSCT -- Prototype Control Section Name

The system variable symbol &SYSPSCT represents the name of the first prototype control section defined in the source program.

When &SYSPSCT is used, the value substituted for &SYSPSCT is the name of the first PSECT statement in the source program. If no PSECT statement has occurred prior to the use of &SYSPSCT, the value assigned to &SYSPSCT is that of a null character string. &SYSPSCT may be used when writing reenterable programs. It provides the macro language programmer with the ability to generate instructions within a prototype control section without requiring that the name of the PSECT be supplied as an explicit parameter. Thus, the macro instruction OUTLINE, from the previous example, might be rewritten as shown below, where PRO was the first PSECT declared in the program.

```
|Name     |Operation|Operand                |
|---------|---------|-----------------------|
|         |MACRO    |                       |
|&LOC     |OUTLINE  |&RTNE                  |
|&SYSPSCT |PSECT    |                       |
|&LOC     |DC       |A(&RTNE)               |
|&SYSECT  |&SYSSTYP |                       |
|         |MEND     |                       |
|---------|---------|-----------------------|
|MAINPROG |CSECT    |                       |
|         |L        |1,ADDR                 |
|ADDR     |OUTLINE  |SUBR                   |
|         |L        |2,X                    |
|---------|---------|-----------------------|
|MAINPROG |CSECT    |                       |
|         |L        |1,ADDR                 |
|PRO      |PSECT    |                       |
|ADDR     |DC       |A(SUBR)                |
|MAINPROG |CSECT    |                       |
|         |L        |2,X                    |
```

&SYSDATE and &SYSTIME -- Date/Time Variables

The global system variables &SYSDATE and &SYSTIME represent the date and time as set by the system during assembly. These symbols are available in open code as well as in macro definitions. The values for these symbols are set only once during an assembly; they cannot be changed by the programmer. GBLC statements are not required to refer to these variables, and a duplicate symbol diagnostic will be issued if these symbols are defined.

When either &SYSDATE or &SYSTIME are encountered in a source program, the assembler will substitute the value from the version identification for the assembly.

&SYSDATE is a GBLC variable symbol of the form:

mo/da/yr

where mo is the month, da is the day, and yr is the year. &SYSTIME is a GBLC variable symbol of the form

hh:mm:ss

where hh is the hour (24 hour clock), mm is minutes, and ss is seconds.

In the following example, the programmer is using the GATWR macro instruction to write the date and time of this program's assembly onto his SYSOUT. &SYSDATE and &SYSTIME are set by the assembler when first encountered; they are both eight characters in length. When the GATWR macro instruction is executed, the output will consist of statements 1 and 2.

```
     |Name     |Operation|Operand                |
     |---------|---------|-----------------------|
     |         |CSECT    |                       |
     |         |  .      |                       |
     |         |  .      |                       |
     |         |  .      |                       |
     |         |GATWR    |DATE,LENGTH1           |
     |         |GATWR    |TIME,LENGTH2           |
     |         |  .      |                       |
     |         |  .      |                       |
     |         |  .      |                       |
     |DATE     |DC       |C'&SYSDATE'            |
     |LENGTH1  |DC       |F'8'                   |
     |TIME     |DC       |C'&SYSTIME'            |
     |LENGTH2  |DC       |F'8'                   |
     |         |  .      |                       |
     |         |  .      |                       |
     |         |  .      |                       |
     |         |END      |                       |
     |---------|---------|-----------------------|
1    |03/18/71
2    |09:30:15
```

In the following example, the macro TODAY is defined to obtain the date from the &SYSDATE variable. &M is set to the month, &D is set to the day, and &Y is set to the year. When the macro definition is referenced, registers 3, 4, and 5 will contain the month, day, and year respectively.

```
          |Name|Operation|Operand                |
          |----|---------|-----------------------|
Header    |    |MACRO    |                       |
Prototype |    |TODAY    |&R1,&R2,&R3            |
Model     |    |LCLC     |&M,&D,&Y               |
Model     |&M  |SETC     |'&SYSDATE'(1,2)        |
Model     |&D  |SETC     |'&SYSDATE'(4,2)        |
Model     |&Y  |SETC     |'&SYSDATE'(7,2)        |
Model     |    |LA       |&R1,&M                 |
Model     |    |LA       |&R2,&D                 |
Model     |    |LA       |&R3,&Y                 |
Trailer   |    |MEND     |                       |
          |----|---------|-----------------------|
Macro     |    |TODAY    |3,4,5                  |
Generated |    |LA       |3,03                   |
Generated |    |LA       |4,18                   |
Generated |    |LA       |5,71                   |
```

KEYWORD MACRO DEFINITIONS AND INSTRUCTIONS

Keyword macro definitions provide the programmer with an alternative way of preparing macro definitions.

A keyword macro definition reduces the number of operands in each macro instruction that corresponds to the definition; the operands may be written in any order.

The macro instructions that correspond to the macro definitions (positional macro instructions and positional macro defini-

tions), described in Section 7, require that the operands be written in the same order as the corresponding symbolic parameters in the operand field of the prototype statement.

In a keyword macro definition, the programmer can assign standard values to any symbolic parameters appearing in the operand field of the prototype statement. The standard value assigned to a symbolic parameter is substituted for the symbolic parameter if the programmer does not write anything in the operand field of the macro instruction to correspond to the symbolic parameter. When a keyword macro instruction is written, the programmer need write only one operand for each symbolic parameter whose value he wants to change.

Keyword macro definitions are prepared the same way as positional macro definitions, except that the prototype statement is written differently, and &SYSLIST may not be used in the definition. The rules for preparing positional macro definitions are in Section 7.

Keyword Prototype

The format of this statement is:

Name	Operation	Operand
Symbolic parameter or blank	Symbol	One or more operands of form described below, separated by commas

Each operand must consist of a symbolic parameter, immediately followed by an equal sign and optionally followed by a standard value. A standard value that is part of an operand must immediately follow the equal sign.

Anything that may be used as an operand in a macro instruction, except variable symbols, may be used as a standard value in a keyword prototype statement. The rules for forming valid macro instruction operands were detailed in Section 8.

These are valid keyword prototype operands:

```
&READER=
&LOOP2=SYMBOL
&S4==F'4096'
```

These are invalid keyword prototype operands:

```
CARDAREA          (no symbolic parameter)
&TYPE             (no equal sign)
&TWO =123         (equal sign does not
                   immediately follow
                   symbolic parameter)
&AREA= X'189A'    (standard value does not
                   immediately follow
                   equal sign)
```

The following keyword prototype statement contains a symbolic parameter in the name field, and four operands in the operand field; the first two operands contain standard values. The mnemonic operation code is MOVE.

Name	Operation	Operand
&N	MOVE	&R=2,&A=S,&T=,&F=

Keyword Macro Instruction

After a programmer has prepared a keyword macro definition, he may use it by writing a keyword macro instruction.

The format of a keyword macro instruction is:

Name	Operation	Operand
Symbol, sequence symbol, or blank	Mnemonic operation code	0 or more operands of form described below, separated by commas

Each operand consists of a keyword immediately followed by an equal sign and an optional value. Any valid operand in a positional macro instruction may be used as a value in a keyword macro instruction. The rules for forming valid positional macro instruction operands were detailed in Section 8.

A keyword consists of one through seven letters and digits, the first of which must be a letter. The keyword part of each keyword macro instruction operand must correspond to one of the symbolic parameters appearing in the operand field of the keyword prototype statement. A keyword corresponds to a symbolic parameter, if the characters of the keyword are identical to the characters of the symbolic parameter that follow the ampersand.

These are valid keyword macro instruction operands:

```
LOOP2=SYMBOL
S4==F'4096'
TO=
```

These are invalid keyword macro instruction operands:

```
&X4F2=0(2,3)      (keyword does not begin
                   with a letter)
CARDAREA=A+2      (keyword is more than
                   seven characters)
=(TO(8),(FROM)    (no keyword)
```

The operands in a keyword macro instruction may be written in any order. If an operand appeared in a keyword prototype statement, a corresponding operand need not appear in the keyword macro instruction. If an operand is omitted, the comma that would have separated it from the next operand need not be written.

The following rules are used to replace the symbolic parameters in the statements of a keyword macro definition:

1. If a symbolic parameter appears in the name field of the prototype statement, and the name field of the macro instruction contains a symbol, the symbolic parameter is replaced by the symbol. If the name field of the macro instruction is blank or contains a sequence symbol, the symbolic parameter is replaced by a null character value.

2. If a symbolic parameter appears in the operand field of the prototype statement, and the macro instruction contains a keyword that corresponds to the symbolic parameter, the value assigned to the keyword replaces the symbolic parameter.

3. If a symbolic parameter was assigned a standard value by a prototype statement, and the macro instruction does not contain a keyword that corresponds to the symbolic parameter, the standard value assigned to the symbolic parameter replaces the symbolic parameter. Otherwise, the symbolic parameter is replaced by a null character value.

Note: If a standard value is a self-defining term, the type attribute assigned to the standard value is N. If a standard value is omitted, the type attribute assigned is O. All other standard values are assigned the type attribute appropriate to their type.

The following keyword macro definition, keyword macro instruction, and generated statements illustrate these rules.

Statement 1 assigns standard values 2 and S to symbolic parameters &R and &A. Statement 6 assigns values FA, FB, and THERE to keywords T, F, and A. The symbol HERE is used in the name field of statement 6.

Since a symbolic parameter (&N) appears in the name field of the prototype statement (1), and the corresponding characters (HERE) of the macro instruction (6) are a symbol, &N is replaced by HERE (in 2).

```
   |Name |Operation|Operand                |
   |-----|---------|-----------------------|
   |     |MACRO    |                       |
1  |&N   |MOVE     |&R=2,&A=S,&T=,&F=      |
2  |&N   |ST       |&R,&A                  |
3  |     |L        |&R,&F                  |
4  |     |ST       |&R,&T                  |
5  |     |L        |&R,&A                  |
   |     |MEND     |                       |
   |-----|---------|-----------------------|
6  |HERE |MOVE     |T=FA,F=FB,A=THERE      |
   |-----|---------|-----------------------|
   |HERE |ST       |2,THERE                |
   |     |L        |2,FB                   |
   |     |ST       |2,FA                   |
   |     |L        |2,THERE                |
```

Since &T appears in the operand field (statement 1), and statement 6 contains the keyword (T) that corresponds to &T, the value assigned to T (FA) replaces &T (in 4). Similarly, FB and THERE replace &F and &A (in 3 and in 2 and 5). Note that the value assigned to &A in statement 6 is used instead of the value assigned to &A in statement 1.

Since &R appears in the operand field of statement 1, and statement 6 does not contain a corresponding keyword, the value assigned to &R (2) replaces &R (in 2, 3, 4, and 5).

Operand Sublists: The value assigned to a keyword and the standard value assigned to a symbolic parameter may be an operand sublist. Any valid operand sublist in a positional macro instruction may be used as a value in a keyword macro instruction and as a standard value in a keyword prototype statement. The rules for forming valid operand sublists were detailed in Section 8 under "Operand Sublists."

Keyword Inner Macro Instructions: Keyword and positional inner macro instructions may be used as model statements in either keyword or positional macro definitions.

MIXED-MODE MACRO DEFINITIONS AND INSTRUCTIONS

Mixed-mode macro definitions allow the programmer to use the features of keyword and positional macro definitions in the same macro definition.

Mixed-mode macro definitions are prepared in the same way as positional macro

definitions, except that the prototype statement is written differently. &SYSLIST may be used to reference the positional operands. The rules for preparing positional macro definitions are in Section 7.

Mixed-Mode Prototype

The format of this statement is:

Name	Operation	Operand
Symbolic parameter or blank	Symbol	Two or more operands of form described below, separated by commas

The operands must be valid operands of positional and keyword prototype statements. All positional operands must precede the first keyword operand. The rules for forming positional operands were discussed in Section 7 under "Macro Instruction Prototype."

The following sample mixed-mode prototype statement contains three positional operands and two keyword operands.

Name	Operation	Operand
&N	MOVE	&TY,&P,&R,&TO=,&F=

Mixed-Mode Macro Instruction

The format for this is:

Name	Operation	Operand
Symbol, sequence symbol, or blank	Mnemonic operation code	0 or more operands of form described below, separated by commas

The operand field consists of two parts. The first part corresponds to the positional prototype operands and is written in the same way as the operand field of a positional macro instruction. The rules for writing positional macro in structions are in Section 8.

The second part of the operand field corresponds to the keyword prototype operands and is written in the same way as the operand field of a keyword macro instruction.

The following mixed-mode macro definition, mixed-mode macro instruction, and generated statements illustrate these facilities.

	Name	Operation	Operand
		MACRO	
1	&N	MOVE	&TY,&P,&R,&TO=,&F=
	&N	ST&TY	&R,SAVE
		L&TY	&R,&P&F
		ST&TY	&R,&P&TO
		L&TY	&R,SAVE
		MEND	
2	HERE	MOVE	H,,2,F=FB,TO=FA
	HERE	STH	2,SAVE
		LH	2,FB
		STH	2,FA
		LH	2,SAVE

The prototype statement (1) contains three positional operands (&TY, &P, and &R) and two keyword operands (&TO and &F). In the macro instruction (2), the positional operands are written in the same order as the positional operands in the prototype statement (the second operand is omitted). The keyword operands are written in an order that is different from the order of keyword operands in the prototype statement.

Mixed-mode inner macro instructions may be used as model statements in mixed-mode, keyword, and positional macro definitions. Keyword and positional inner macro instructions may be used as model statements in mixed-mode macro definitions.

MACRO DEFINITION COMPATIBILITY

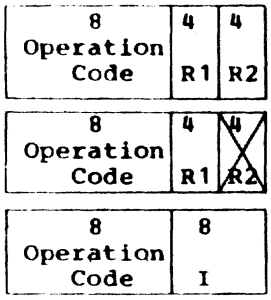
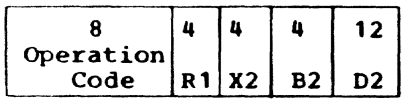
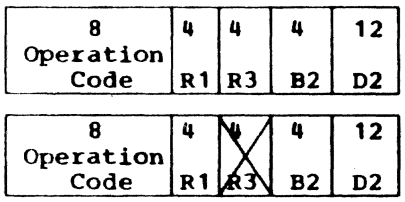
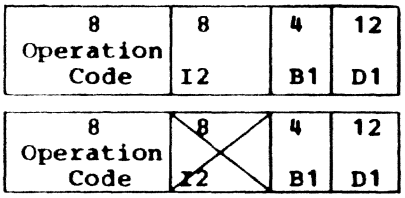
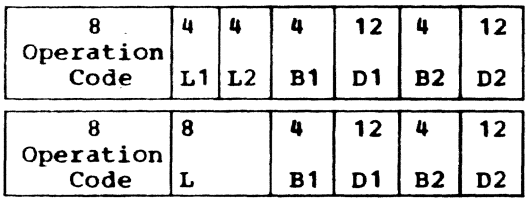
Macro definitions prepared for use with other IBM assemblers having macro language facilities may be used with the TSS assembler, provided that all SET symbols are defined in an appropriate LCLB, GBLA, GBLB, or GBLC statement. The AIFB and AGOB instructions will be processed by the TSS assembler in the same way that AIF and AGO are processed.

Name	Operation	Operands	Specified as
[symbol]	CCW	command code	1 byte, absolute expression, right justified.
		,data address	Absolute, relocatable, or complex expression.
		,flag values	Absolute expression.
		,count	Absolute expression.
	CNOP	byte alignment, word type {double / single}	b=n, where n=0, 2, 4, or 6 w=n, where n=4 or 8
[symbol]	COM	same as CSECT	Default = standard common section.
	COPY	symbol name	Name of area to be copied.
[symbol]	CSECT	[public storage] [,read only storage] [,variable section length] [,privileged section] [,section includes SYS entry points]	PUBLIC ,READONLY ,VARIABLE ,PRVLGD ,SYSTEM
[symbol]	CXD	none	
[symbol]	DC	[duplication factor] constant type [length {bits / bytes}] [scale] [exponent] constant[,...]	See Appendix C. One or more additional operands, separated by comma(s), may be specified.
	DROP	[reg1 [,reg2,...,reg16]]	Absolute value. Default = all currently active base registers.
[symbol]	DS	[duplication factor] constant type [length {bits / bytes}] [scale] [exponent]	Same as DC except as noted below. See Appendix C.
		1. Specification of 'constant' operand, optional. 2. 'Constant' operand reserves space but does not store data. 3. Maximum length for C and X constants, 65,535 bytes. 4. Duplication factor of 0 forces alignment to assumed alignments in DC.	

Name	Operation	Operands	Specified as
[symbol]	DSECT	none	
symbol	DXD	[duplication factor] constant type [length {bits / bytes}] [scale] [exponent] [constant]	Same as DC except as noted below.
		1. 'Constant' operand does not cause initialization to value specified. 2. 'Scale' and 'Exponent' operands may be specified. 3. Multiple operands and constants only used to determine length.	
	EJECT	none	
	END	[control transfer point]	Relocatable or absolute expression. Default = first instruction of CSECT.
	ENTRY	[entry point][,...]	Relocatable symbols.
[symbol]	EQU	[previously defined symbol] [,length] [,type]	 absolute integer (1-65535) absolute integer (0-255)
	EXTRN	external symbol [,...]	Relocatable symbols.
	ICTL	[beginning source column] [,ending source column] [,continue column]	b=decimal digit, range, 1-40; default=1. e=decimal digit, range, 41-80; default=71. c=decimal digit, range, 2-40; default=16.
	ISEQ	[sequence field - left col] [sequence field - right col]	Decimal digits, default = no sequence check.
[symbol]	LTORG	none	
	ORG	[new location counter address]	Expression; default=current location counter positicn+1.
	PRINT	printing option {listing / no listing}	ON OFF
		[,macro gen {executable / all / none}]	FULIGEN GEN NOGEN
		[,constants {print full constant / print 8 bytes or less}]	DATA NODATA
symbol	PSECT	same as CSECT	Same as CSECT.

Name	Operation	Operands	Specified as
	SPACE	[no. of lines to be spaced]	Decimal digits, default=1.
[symbol]	START	[initial location ctr. address]	Self-defining term, default=0.
[symbol]	TITLE	'characters'	To 100 characters.
		Note: Symbol may be from 1 to 4 alphameric characters.	
	USING	base value, reg1[,reg2,...,reg16]	Absolute or relocatable value. Absolute expression.

APPENDIX B: MACHINE INSTRUCTION FORMAT

	Basic Machine Format	Assembler Operand Field Format	Applicable Instructions
RR		R1,R2 R1 I	All RR instructions except SPM and SVC SPM SVC
	(See notes 1,6, and 8)		
RX		R1,D2(X2,B2) R1,S2(X2)	All RX instructions
	(See notes 1-4, and 7)		
RS		R1,R3,D2(B2) R1,R3,S2 R1,D2(B2) R1,S2	BXH,BXLE,LM,STM All shift instructions
	(See notes 1-3, 7, and 8)		
SI		D1(B1),I2 S1,I2 D1(B1) S1	All SI instructions except LPSW,SSM, HIO,SIO,TIO,TCH,TS LPSW,SSM,HIO,SIO, TIO,TCH,TS
	(See notes 2, 3, and 6-8)		
SS		D1(L1,B1),D2(L2,B2) S1(L1),S2(L2) D1(L,B1),D2(B2) S1(L),S2	PACK,UNPK,MVO,AP, CP,DP,MP,SP,ZAP NC,OC,XC,CLC,MVC,MVN, MVZ,TR,TRT,ED,EDMK
	(See notes 2, 3, 5, and 7)		

Notes for Appendix B:

1. R1, R2, and R3 are absolute expressions that specify general or floating-point registers. General registers are 0 through 15; floating-point registers are 0, 2, 4, and 6.

2. D1 and D2 are absolute expressions that specify displacements; a value from 0 to 4095 may be specified.

3. B1 and B2 are absolute expressions that specify base registers; registers are 0 - 15.

4. X2 is an abolute expression that specifies an index register; registers 1-15 may be used as index registers. If a base register and no index register is desired, X2 may be either specified as 0 or omitted, with a comma preceding B2.

   Examples: L 2,48(0,5)
             L 2,48(,5)

5. L, L1, and L2 are absolute expressions that specify field lengths; L value can be specified from 0 to 256; L1 and L2 values from 0 to 16. The assembled value will be one less than the specified value except that if 0 is specified, 0 will be assembled. L may be defaulted; the length assigned will be that of the first operand. L1 and L2 may be defaulted; however, a comma must precede B1 and B2. If an explicit base and displacement have been written, the defaulted (or implied) length will be the length attribute of the expression specifying the displacement. If the base and displacement have been implied, the defaulted length will be the length attribute of the expression specifying the effective address.

6. I and I2 are absolute expressions that provide immediate data; the value can be from 0 to 255.

7. S1 and S2 are absolute or relocatable expressions that specify an address.

8. RR, RS, and SI instruction fields that are crossed out in the machine formats are not examined during instruction execution. The fields are not written in the symbolic operand, but are assembled as binary 0s.

APPENDIX C: SUMMARY OF CONSTANTS

Type	Implied Length (Bytes)	Alignment	Length Modifier Range	Specified by	Number of Constants Per Operand	Range for Exponents	Range for Scale	Truncation/ Padding Side
C	as needed	byte	.1 to 256 (1)	characters	one			right
X	as needed	byte	.1 to 256 (1)	hexadecimal digits	one			left
B	as needed	byte	.1 to 256	binary digits	one			left
F	4	word	.1 to 8	decimal digits	multiple	-85 to +75	-187 to +346	left
H	2	halfword	.1 to 8	decimal digits	multiple	-85 to +75	-187 to +346	left
E	4	word	.1 to 8	decimal digits	multiple	-85 to +75	0 to 2L-2 (2)	right
D	8	doubleword	.1 to 8	decimal digits	multiple	-85 to +75	0 to 2L-2 (2)	right
P	as needed	byte	.1 to 16	decimal digits	multiple			left
Z	as needed	byte	.1 to 16	decimal digits	multiple			left
A	4	word	.1 to 4 (3)	any expression	multiple			left
V	4	word	1 to 4 (3)	relocatable symbol	multiple			
R	4	word	1 to 4 (3)	an external symbol	multiple			
S	2	halfword	2 only	one absolute or relocatable expression or two absolute expressions: exp (exp)	multiple			
Y	2	halfword	.1 to 2 (3)	any expression	multiple			left
Q	4	word	1 to 4 (4)	symbol naming a DXD or DSECT	multiple			left

(1) In a DS assembler instruction C and X type constants may have length specification to 65535.
(2) L is length of constant. Negative scaling is not permitted.
(3) Bit length specification permitted with absolute expressions only. Relocatable values may be 1, 2, 3, or 4 bytes only.
(4) No bit length specification permitted.

These charts summarize the macro language described in the second half of this manual.

1. Chart 1 describes the name and operand fields of each statement.
2. Chart 2 indicates which macro language elements may be used in the name and operand fields of each statement.
3. Chart 3 is a summary of the expressions that may be used in macro language statements.
4. Chart 4 is a summary of the attributes that may be used in each expression.
5. Chart 5 is a summary of the variable symbols that may be used in each expression.

Chart 1. Statements (Part 1 of 2)

Instruction	Name Field	Operand Field
AGO,ABOB	Sequence symbol or blank	Sequence symbol
AIF,AIFB	Sequence symbol or blank	Logical expression enclosed in parentheses, immediately followed by a sequence symbol
ANOP	Sequence symbol	Blank
COPY	Sequence symbol or blank	Symbol
GBLA,GBLB,GBLC	Blank	One or more variable symbols to be used as SET symbols, separated by commas[3]
LCLA,LCLB,LCLC	Blank	One or more variable symbols to be used as SET symbols, separated by commas[3]
MACRO[1]	Blank	Blank
MEND[1]	Sequence symbol or blank	Blank
MEXIT[1]	Sequence symbol or blank	Blank
MNOTE	Sequence symbol or blank	Severity code, followed by a comma, followed by any combination of characters enclosed in apostrophes
SETA	SETA symbol	Arithmetic expression
SETB	SETB symbol	0 or 1, or a logical expression enclosed in parentheses
SETC	SETC symbol	Type attribute, character expression substring notation, or concatenation of character expressions and substring notations
Model statement (any assembler language mnemonic operation code, except COPY, END, ICTL, ISEQ, MACRO, and START)[5]	Symbol, variable symbol, sequence symbol or blank, or concatenation of variable symbols and other characters that is equivalent to symbol	Any combination of characters (including variable symbols)

Chart 1. Statements (Part 2 of 2)

Instruction	Name Field	Operand Field
Prototype statement	Symbolic parameter or blank	0 or more operands that are symbolic parameters, separated by commas, followed by 0 or more operands (separated by commas) in the form of symbolic parameter, equal sign, optional standard value
Macro instruction statement	Symbol, or variable symbol, sequence symbol or blank, or concatenation of variable symbols and other characters that is equivalent to symbol	0 or more positional operands separated by commas, followed by 0 or more keyword operands (separated by commas) in the form of keyword, equal sign, value[2]
Assembler language statement[4] [5]	Symbol, or variable symbol, sequence symbol or blank, or concatenation of variable symbols and other characters that is equivalent to symbol	Any combination of characters (including variable symbols)

[1]May be used only 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]SET symbols may be defined as subscripted SET symbols.

[4]Variable symbols may not be used to generate the following mnemonic operation codes, nor may variable symbols be used in the name and operand fields of these instructions: COPY, END, ICTL, ISEQ, REPRO, and START. Variable symbols may not be used to generate a macro instruction mnemonic operation code.

[5]The line following a REPRO statement may not contain variable symbols.

Chart 2. Macro Language Elements

Statement	Variable Symbols: Symbolic Parameter	Variable Symbols: Global SET Symbols: SETA	Variable Symbols: Global SET Symbols: SETB	Variable Symbols: Global SET Symbols: SETC	Variable Symbols: Local SET Symbols: SETA	Variable Symbols: Local SET Symbols: SETB	Variable Symbols: Local SET Symbols: SETC	System Variable Symbols: &SYSNDX	System Variable Symbols: &SYSECT	System Variable Symbols: &SYSLIST	Attributes: Type	Attributes: Length	Attributes: Scaling	Attributes: Integer	Attributes: Count	Attributes: Number	Sequence Symbol
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
COPY																	Name
Inner Macro[1]	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name Operand							Name
SETA	Operand[2]	Name Operand	Operand[2]	Operand[2]	Name Operand	Operand[2]	Operand[2]	Operand		Operand[2]		Operand	Operand	Operand	Operand	Operand	
SETB	Operand[2]	Operand[2]	Name Operand	Operand[2]	Operand[2]	Name Operand	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	
SETC	Operand	Operand[7]	Operand[2]	Name Operand	Operand[7]	Operand[2]	Name Operand	Operand	Operand	Operand	Operand						
AIF	Operand[2]	Operand[2]	Operand	Operand[2]	Operand[2]	Operand	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Operand[2]	Name Operand
AGO																	Name Operand
AREP																	Name
MEXIT																	Name
MNOTE	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand	Operand							Name
MEND																	Name
Outer Macro[1]		Name Operand	Name Operand	Name Operand	Name Operand	Name Operand	Name 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.

Chart 3. Expressions

Expression	Arithmetic Expressions	Character Expressions	Logical Expressions
May contain	1. Self-defining terms 2. Length, scaling, integer, count, and number attributes 3. SETA and SETB symbols 4. SETC symbols whose value is 1-8 decimal digits 5. Symbolic parameters, if corresponding operand is self-defining term 6. &SYSLIST(n), if corresponding operand is self-defining term 7. &SYSLIST(n,m), if corresponding operand is self-defining term 8. &SYSNDX	1. Any combination of characters enclosed in quotation marks 2. Any variable symbol enclosed in apostrophes 3. Concatenation of variable symbols and other characters enclosed in apostrophes 4. Request for type attribute.	1. SETB symbols 2. Arithmetic relations[2] 3. Character relations[2]
Operators are	+,-,*, and / parentheses permitted	Concatenation, with a period (.)	AND, OR, and NOT parentheses permitted
Range of values	-2^{31} to $+2^{31}-1$	0 to 255 characters	0 (false) or 1 (true)
May be used in	1. SETA operands 2. Arithmetic relations 3. Subscripted SET symbols 4. &SYSLIST 5. Substring notation 6. Sublist notation	1. SETC operands[3] 2. Character relations[2]	1. SETB operands 2. AIF operands

[1]An arithmetic relation consists of two arithmetic expressions related by operators GT, LT, EQ, NE, GE, or LE.

[2]A character relation consists of two character expressions related by operators GT, LT, EQ, NE, GE, or LE. Type attribute notation and the substring notation also may be used in character relations. Maximum size of character expression that can be compared is 255 characters. If the two character expressions are of unequal size, the smaller will always compare less than the larger.

[3]Maximum of eight characters will be assigned.

Chart 4. Attributes

Attribute	Notation	May be used with	Produces pertinent value when T' is	May be used in
Type	T'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m), inside macro definitions	--	1. SETC operand fields 2. Character relations
Length	L'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m), inside macro definitions	Any letter except L, M,N,O,T, and U	Arithmetic expressions
Scaling	S'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m), inside macro definitions	H,F,G,D,E,K,P, and Z	Arithmetic expressions
Integer	I'	Symbols outside macro definitions; symbolic parameters, &SYSLIST(n), and &SYSLIST(n,m), inside macro definitions	H,F,G,D,E,K,P, and Z	Arithmetic expressions
Count	K'	Symbolic parameters corresponding to macro instruction operands &SYSLIST(n) and &SYSLIST(n,m), inside macro definitions	Any letter except L	Arithmetic expressions
Number	N'	Symbolic parameters &SYSLIST and &SYSLIST(n), inside macro definitions	Any letter except I	Arithmetic expressions

Chart 5. Variable Symbols

Variable Symbol	Defined by	Initialized or set to	Value changed by	May be used in
Symbolic[1] parameter	Prototype statement	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions
SETA	ICLA or GBLA instruction	0	SETA instruction	1. Arithmetic expressions 2. Character expressions

Variable Symbol	Defined by	Initialized or set to	Value changed by	May be used in
SETB	ICLB or GBLB instruction	0	SETB instruction	1. Arithmetic expressions 2. Character expressions 3. Logical expressions
SETC	ICLC or GBLC instruction	Null character value	SETC instruction	1. Arithmetic expressions, if value is self-defining term 2. Character expressions
&SYSNDX[1]	Assembler	Macro instruction index	(Constant throughout definition; unique for each macro instruction)	1. Arithmetic expressions 2. Character expressions
&SYSECT[1]	Assembler	Control section in which macro instruction appears	(Constant throughout definition; set by CSECT, DSECT, and START)	Character expressions
&SYSLIST[1]	Assembler	Not applicable	Not applicable	N'SYSLIST in arithmetic expressions
&SYSLIST(n)[1] &SYSLIST (n,m)[1]	Assembler	Corresponding macro instruction operand	(Constant throughout definition)	1. Arithmetic expressions if operand is self-defining term 2. Character expressions
&SYSSTYP[1]	Assembler	Mnemonic operation of statement which defined symbol represented by &SYSECT	(Constant throughout definition set by CSECT, PSECT, COM, DSECT, and START)	Character expressions
&SYSPSCT[1]	Assembler	Name of first PSECT defined within assembly; null string if no PSECT is defined	(Constant throughout definition; set by first PSECT statement)	Character expressions
&SYSDATE	Assembler	Assembler date stamp	(Constant throughout assembly)	1. Character expressions 2. Macro definitions
&SYSTIME	Assembler	Assembler time stamp	(Constant throughout assembly)	1. Character expressions 2. Macro definitions

[1]May be used only in macro definitions.

Given:

1. A TABLE with 15 entries, each 16 bytes long, having this format:

Number of Items	Switches	Address	Name
3 bytes	1 byte	4 bytes	8 bytes

2. A LIST of items, each 16 bytes long, having this format:

Name	Switches	Number of Items	Address
8 bytes	1 byte	3 bytes	4 bytes

Find: Any of the items in the list which occur in the table and put the switches, number of items, and address from that LIST entry into the corresponding TABLE entry. If the LIST item does not occur in the TABLE, turn on the first bit in the switches-byte of the LIST entry.

The TABLE entries have been sorted by their name.

```
EXAM     TITLE 'SAMPLE ASSEMBLY'
*
*        THIS IS THE MACRO DEFINITION
*
         MACRO
         MOVE &TO,&FROM
.*
.*       DEFINE SETC SYMBOL
.*
         LCLC &TYPE
.*
.*       CHECK NUMBER OF OPERANDS
.*
         AIF   (N'&SYSLIST NE 2).ERROR1
.*
.*       CHECK TYPE ATTRIBUTES OF OPERANDS
.*
         AIF   (T'&TO NE T'&FROM).ERROR2
         AIF   (T'&TO EQ 'C' OR T'&TO EQ 'G' OR T'&TO EQ 'K').TYPECGK
         AIF   (T'&TO EQ 'D' OR T'&TO EQ 'E' OR T'&TO EQ 'H').TYPEDEH
         AIF   (T'&TO EQ 'F').MOVE
         AGO   .ERROR3
.TYPEDEH ANOP
.*
.*       ASSIGN TYPE ATTRIBUTE TO SETC SYMBOL
.*
&TYPE    SETC T'&TO
.MOVE    ANOP
*        NEXT TWO STATEMENTS GENERATED FOR MOVE MACRO
         L&TYPE   2,&FROM
         ST&TYPE  2,&TO
         MEXIT
.*
.*       CHECK LENGTH ATTRIBUTES OF OPERANDS
.*
.TYPECGK AIF   (L'&TO NE L'&FROM OR L'&TO GT 256).ERROR4
*        NEXT STATEMENT GENERATED FOR MOVE MACRO
         MVC   &TO,&FROM
```

```
         MEXIT
.*
.*       ERROR MESSAGES FOR INVALID MOVE MACRO INSTRUCTIONS
.*
.ERROR1  MNOTE 1, 'IMPROPER NUMBER OF OPERANDS, NO STATEMENTS GENERATED'
         MEXIT
.ERROR2  MNOTE 1, 'OPERAND TYPES DIFFERENT, NO STATEMENTS GENERATED'
         MEXIT
.ERROR3  MNOTE 1, 'IMPROPER OPERAND TYPES, NO STATEMENTS GENERATED'
         MEXIT
.ERROR4  MNOTE 1, 'IMPROPER OPERAND LENGTHS, NO STATEMENTS GENERATED'
         MEND
*
*        MAIN ROUTINE
*
ARGSRCH  CSECT PUBLIC,READONLY
BEGIN    STM   R14,R12,12(R13)     SAVE CALLER'S REGISTERS
         LR    R14,R13
         L     R13,72(R13)         SECURE PSECT COVER FOR THIS ROUTINE
         USING RSAVE,R13
         ST    R14,RSAVE+4         SAVE CALLER'S SAVE AREA COVER
         LR    R12,R15             ESTABLISH ADDRESSABILITY OF PROGRAM
         USING BEGIN,R12             AND TELL THE ASSEMBLER
         LM    R5,R7,=A(LISTAREA,16,LISTEND)  LOAD LIST AREA PARAMETERS
         USING LIST,R5               REGISTER 5 POINTS TO THE LIST
MORE     BAS   R14,SEARCH          FIND LIST ENTRY IN TABLE
         TM    SWITCH,NONE         CHECK TO SEE IF NAME WAS FOUND
         BO    NOTTHERE              BRANCH IF NOT
         USING TABLE,R1            REGISTER 1 NOW POINTS TO TABLE ENTRY
         MOVE  TSWITCH,LSWITCH                        MOVE FUNCTIONS
```

Following the macro instruction MOVE, the macro definition might go through the following sequence:

```
         LCLC  &TYPE
         AIF   (2 NE 2).ERROR1
         AIF   (T'TSWITCH NE T'LSWITCH).ERROR2

         AIF   (T'TSWITCH EQ 'C' OR T'TSWITCH EQ 'G' OR T'TSWITCH EQ 'K'
).TYPECGK
.TYPECGK AIF   L'TSWITCH NE L'LSWITCH OR L'TSWITCH GT 256).ERROR4
```

As a result, the following instruction would be assembled into the object program in place of the macro instruction MOVE:

```
         MVC   TSWITCH,LSWITCH
```

```
         MOVE  TNUMBER,LNUMBER                        FROM LIST ENTRY
```

Again, the MOVE macro instruction and how the macro definition would process it:

```
         LCLC  &TYPE
         AIF   (2 NE 2).ERROR1
         AIF   (T'TNUMBER NE T'LNUMBER).ERROR2

         AIF   (T'TNUMBER EQ 'C' OR T'TNUMBER EQ 'G' OR T'TNUMBER EQ 'K'
).TYPECGK
.TYPECGK AIF   (L'TNUMBER NE L'LNUMBER OR I'TNUMBER GT 256).ERROR4
```

This would be substituted in the assembled code in place of the macro instruction:

```
         MVC   TNUMBER,LNUMBER
```

```
         MOVE  TADDRESS,LADDRESS                      TO TABLE ENTRY
```

Once again, the MOVE macro definition would process the macro instruction:

```
         LCLC  &TYPE
         AIF   (2 NE 2).ERROR1
         AIF   (T'TADDRESS NE T'LADDRESS).ERROR2
```

```
         AIF   (T'TADDRESS EQ 'C' OR T'TADDRESS EQ 'G' OR T'TADDRESS EQ
'K').TYPECGK
         AIF   T'TADDRESS EQ 'D' OR T'TADDRESS EQ 'E' OR T'TADDRESS EQ
'H').TYPEDEH
         AIF   (T'TADDRESS EQ 'F').MOVE

.MOVE    ANOP

These lines of code would be generated, replacing the MOVE macro instruction:
         L     2,LADDRESS
         ST    2,TADDRESS

The assembler continues with the following source statements:
         BXLE  R5,R6,MORE          LOOP THROUGH THE LIST
         B     EXIT                SUBROUTINE COMPLETION
NOTTHERE OI    LSWITCH,NONE        TURN ON SWITCH IN LIST ENTRY
         BXLE  R5,R6,MORE          LOOP THROUGH THE LIST
EXIT     L     R13,RSAVE+4         RETURN TO CALLER
         L     R14,12(R13)         RESTORE EXIT ADDRESS
         OI    11(R13),1           SET LINKAGE BIT
         LM    R2,R12,28(R13)      MULTIPLE REGISTER RESTORE
         BR    R14                 RETURN TO CALLER
NONE     EQU   X'80'
*
*        BINARY SEARCH ROUTINE
*
SEARCH   NI    SWITCH,255-NONE     TURN OFF NOT-FOUND  SWITCH
         LM    R1,R3,=F'128,4,128'                   LOAD TABLE PARAMETERS
         LA    R1,TABLAREA-16(R1) GET ADDRESS OF MIDDLE TABLE ENTRY
LOOP     SRL   R3,1                DIVIDE INCREMENT BY 2
         CLC   LNAME,TNAME         COMPARE LIST ENTRY WITH TABLE ENTRY
         BH    HIGHER                BRANCH IF SHOULD BE HIGHER IN TABLE
         BCR   8,R14               EXIT IF FOUND
         SR    R1,R3               OTHERWISE IT IS LOWER IN THE TABLE
*                  SO  SUBTRACT INCREMENT
         BCT   R2,LOOP             LOOP 4 TIMES
         B     NOTFOUND            ARGUMENT IS NOT IN THE TABLE
HIGHER   AR    R1,R3               ADD INCREMENT
         BCT   R2,LOOP             LOOP 4 TIMES
NOTFOUND OI    SWITCH,NONE         TURN ON NOT-FOUND  SWITCH
         BR    R14                 EXIT

All literals, except literal address constants, go here:
         =F'128,4,128'
SAP      PSECT
         ENTRY BEGIN
RSAVE    DC    F'76'               USER'S SAVE AREA
         DC    18F'0'
SWITCH   DS    X                   FOUND SWITCH FOR BINARY SEARCH
*
*        THIS IS THE TABLE
*
         DS    OD
TABLAREA DC    XL8'0',CL8'ALPHA'
         DC    XL8'0',CL8'BETA'
         DC    XL8'0',CL8'DELTA'
         DC    XL8'0',CL8'EPSILON'
         DC    XL8'0',CL8'ETA'
         DC    XL8'0',CL8'GAMMA'
         DC    XL8'0',CL8'IOTA'
         DC    XL8'0',CL8'KAPPA'
         DC    XL8'0',CL8'LAMBDA'
         DC    XL8'0',CL8'MU'
         DC    XL8'0',CL8'NU'
         DC    XL8'0',CL8'OMICRON'
         DC    XL8'0',CL8'PHI'
         DC    XL8'0',CL8'SIGMA'
```

```
         DC    XL8'0',CL8'ZETA'
*
*        THIS IS THE LIST
*
LISTAREA DC    CL8'LAMBDA',X'0A',FL3'29',A(BEGIN)
         DC    CL8'ZETA',X'05',FL3'5',A(LOOP)
         DC    CL8'THETA',X'02',FL3'45',A(BEGIN)
         DC    CL8'TAU',X'00',FL3'0',A(1)
         DC    CL8'LIST',X'1F',FL3'456',A(0)
LISTEND  DC    CL8'ALPHA',X'00',FL3'1',A(123)
*
*        THESE ARE THE SYMBOLIC REGISTERS
*
R1       EQU   1
R2       EQU   2
R3       EQU   3
R5       EQU   5
R6       EQU   6
R7       EQU   7
R12      EQU   12
R13      EQU   13
R14      EQU   14
R15      EQU   15

Address constant literals go here:
               =A(LISTAREA,16,LISTEND)
*
*              THIS IS THE FORMAT DEFINITION OF LIST ENTRYS
*
LIST     DSECT
LNAME    DS    CL8
LSWITCH  DS    C
LNUMBER  DS    FL3
LADDRESS DS    F
*
*        THIS IS THE FORMAT DEFINITION OF TABLE ENTRYS
*
TABLE    DSECT
TNUMBER  DS    FL3
TSWITCH  DS    C
TADDRESS DS    F
TNAME    DS    CL8
         END
```

&SYS, restrictions on use 56
&SYSDATE (see date variable)
&SYSECT
(see current control section name)
&SYSLIST
(see macro instruction operand)
&SYSNDX
(see macro instruction index)
&SYSPSCT
(see prototype control section name)
&SYSSTYP
(see current control section type)
&SYSTIME (time variable)

Absolute term 9
Address constants 41-43
A-type 41,65
complex relocatable expressions 41
literals not allowed 11
Q-type 42
R-type 42-43,65
S-type 42,65
V-type 42,65
Y-type 41-42,65
Addressing
caution, CCW command 46
dummy sections 19-20
explicit 15
external control sections 24
external dummy sections 22
implied 15
relative 17
AGO instruction
example 75
format of 74
inside macro definitions 75
operand field of 75
outside macro definitions 75
sequence symbol in 74-75
use of 74
AGOB insturction (see AGO instruction)
AIF instruction
example of 74
format of 74
inside macro definitions 74
invalid operand fields of 74
logical expression in 74
operand field of 74
outside macro definitions 74
sequence symbols in 74
use of 73-74
valid operand fields of 74
AIFB instruction (see AIF instruction)
Alignment, boundary
CNOP instruction for 50-51
machine instruction 25
Ampersands in
character expressions 70
macro instruction operands 59
MNOTE instruction 74-75
symbolic parameters 54,56-57
variable symbols 53
ANOP instruction
example of 75
format of 75
sequence symbol in 75
use of 75
Apostrophes, free 55-56
Apostrophes in
character expressions 70
macro instruction operands 59
MNOTE instruction 78
Arithmetic expressions
arithmetic relations 72
evaluation procedure 68
operand sublists 69
operators allowed 68
parenthesized terms in evaluation of 68
SETA instruction 68-69
SETB instruction 72-73
Substring notation 70-71
terms allowed 68
Arithmetic relations 72
Assembler instructions
statement 30-51

Assembler language
comparison chart 95-100
macro language, relation to 52
statement format 5-6,7
structure 7-8
Assembler program
basic functions 2
output 47
Assembly, terminating an 51
Assembly no operation (see ANOP instruction)
Asterisk
MNOTE instruction 78
Attributes (see also specific attributes)
assignment to control sections 22
how referred to 64
inner macro instruction operands 64
kinds of 64-67
notations 64
operand sublists 64
outer macro instruction operands 64
summary chart of 99
use of 64-66
A-type address constant 41

Base registers
address calculation 2,25
DROP instruction 16
loading of 15
USING instruction 15-16
Binary constant 38,65
Binary self-defining term 10
Blanks
logical expressions 72
macro instruction operands 59

CCW instruction 46
Channel command word, defining 46

Character constant 37-38,65
Character expressions
  ampersands in 70
  character relations 72
  examples of 70
  periods and 70
  SETB instructions 72-73
  SETC instructions 69-70
Character relation 72
Character self-defining term 10
Character set 7
CNOP instruction 50-51
Coding form 4
COM instruction 21
Commas, macro instruction operands 59
Comments statements
  example of 6,58
  model statements 55
  not generated 58
Comparison chart 95-100
Complex relocatable expressions 41
Concatenation
  character expressions 70,71-72
  defined 57
  examples of 57
  substring notations 71-72
Conditional assembly elements, summary
 charts of 76
Conditional assembly instructions
  how to write 63-76
  summary of 76
  use of 63
    (see also specific instructions)
Conditional branch
  (see AIF instruction)
Conditional branch instruction 28
Constants
  defining
    (see DC instructions)
  summary of 94
    (see also specific types)
Continuation lines 5
Control dictionary 18
Control section location assignment 18
Control sections
  attributes of 22
  blank COMMON 21
  CSECT instruction 19
  defined 18
  DSECT instruction 20-21
  first control section, properties of 18
  identification of 18
  maximum location counter value 11
  PSECT instruction 21-22
  START instruction 18-19
  unnamed 19
COPY instruction 51
COPY statements in macro definitions
  format of 58
  model statements, contrasted 58
  operand field of 58
  use of 58
Count attribute
  defined 66
  operand sublists 66
  use of 66
  variable symbols 66
CSECT instruction 19
Current control section name
 (&SYSECT) 82-83,100
  affected by CSECT, DSECT, START 82-83
  example of 82-83
  use of 82-83
Current control section type
 (&SYSSTYP) 83-84,98
CXD instruction 45

Data definition instructions 32
  channel command words 46
  constants 32-43
    storages 42-43
Date variable (&SYSDATE) 85,100
DC instruction 31-43
  constant operand subfield 36
    address-constants (see address
     constants)
    binary constant 38
    character constant 37
    decimal-constants 40-41
    fixed-point constants 38-39
    floating-point constants 39-40
    hexadecimal constant 37-38
    type codes for 34
  duplication factor operand subfield 33
  operand subfield modifiers 34
  type operand subfield 33-34
    bit length specification 34-35
    exponent modifier 36
    length modifier 34
    scale modifier 35
Decimal constants 40-41,65
  length modifier 40-41
  length, maximum 40
  packed 40
  zoned 40
Decimal field, integer attribute of 65
Decimal self-defining terms 10
Defining constants
  (see DC instruction)
Defining storage
  (see DC instruction, DS instruction)
Defining symbols 9-11
Dimension, subscripted SET symbols 81
Double-shift instruction 25
DROP instruction 16
DS instruction 43-44
  defining areas 44
  forcing alignment 44
DSECT instruction 20
Dummy section location assignment 20
Duplication factor 33,35
  defining fields of an area 44
  forcing alignment 44
DXD instruc 44-45

Effective address, length 26
EJECT instruction 47
END instruction 51
ENTRY instruction 23
Entry point symbol, identification of 23
EQU instruction 31-32

Equal signs, as macro instruction operands 59
Error message
(see MNOTE instruction)
Explicit addressing 15
length 26-27
Exponent modifiers 36
Expressions 13-14
absolute 13
evaluation 13
relocatable 14
summary chart of 98
Extended mnemonic codes
RR format 30
RX format 29
External control section, addressing of 24
External dummy section
allocation for 22
definition of 22
description of 22
External symbol, identification of 23-24
EXTRN instruction 23-24

First control section 18
Fixed-point constants 35-36,38-39,65
format 39
positioning of 38
scaling 35
values, minimum and maximum 38
Fixed-point field, integer attribute of 65
Floating-point constants 39-40,65
alignment 40
format 40
scale modifiers 40
Floating-point field, integer attribute of 65
Format control, input 48

GBLA instruction
format of 79
inside macro definitions 79-80
outside macro definitions 79-80
use of 79-81
GBLB instruction
format of 79
inside macro definitions 79-80
outside macro definitions 79-80
use of 79-81
GBLC instruction
format of 79
inside macro definitions 79-80
outside macro definitions 79-80
use of 79-81
General register 0, base register usage 16
Global SET symbols
defining 79
examples of 79-81
local SET symbols, compared 79
using 79
Global system variable symbols 78-82
definition of 78
types of 78
Global variable symbols
types of 78
(see also global SET symbols, subscripted SET symbols)

Hexadecimal constants 37-38,65
Hexadecimal self-defining terms 10

I'
(see integer attribute)
ICTL instruction 48
Identification-sequence field 6
Identify dummy section 22
Identifying blank common control section 21
Implied addressing 15-16
length 26
Implied length specification 26-27
Inner macro instruction 61-62
defined 61
example of 61
symbolic parameters in 61-62
Instruction alignment 25
Integer attribute 65-66
decimal fields 65
defined 65
examples of 65
fixed-point fields 65
floating-point fields 65
notation 65
restrictions on use 66
symbols 65-66
use of 65-66
ISEQ instruction 49

K'
(see count attribute)
Keyboard statement formats
boundaries 5
continuation lines 5
Keyword
defined 85
keyword macro-instruction 85-87
symbolic parameter and 86
Keyword, inner macro instructions used in 87
Keyword macro definition
positional macro definitions, compared 85-86
use 86
Keyword macro instruction
example of 86-87
format of 86
keywords in 86
operand sublists in 87
operands
invalid examples 86-87
valid examples 86
Keyword prototype statement 86
example of 86
format of 86
operands 86
invalid examples 86
valid examples 86
standard values 86

I'
(see length attribute)

LCLA instruction
format of 67
use of 67
LCLB instruction
format of 67
use of 67
LCLC instruction
format of 67
use of 67
Length attribute
defined 65
examples 65
notation 65
restrictions on use 65
symbols 12,65
use of 65-66
Length modifier 35-36
bit-length specification 34
length subfield 34
Lengths explicit and implied 26-27
Linkage symbols
(see also ENTRY instruction, EXTERNAL instruction)
entry point symbol 23
external symbol 23-24
Listing control instructions 46-48
Listing, spacing 47
Literal pools 12,49
Literals 11-12
character 26
DC instruction, used in 12
duplicate 50
format 12
literal pool, beginning 49-50
literal pools, multiple 12
Local SET symbols
defining 79
examples of 79-81
global SET symbols, compared 79
using 79-81
Local system variable symbols
(see also local SET symbols)
(see also subscripted SET symbols)
types of 78
Location counter 10-11,33,36,45,50
maximum value 11
references to 11
Logical expressions
AIF instructions 73-74
arithmetic relations 72
blanks in 72
character relation 72
evaluation of 73
invalid examples of 73
logical operators in 72
parenthesized terms in
evaluation of 73
examples of 73
relation operators in 72
SETB instructions 72
terms allowed in 72
valid examples of 73
LTORG instruction 49

Machine-instruction examples and format
RR 25,27
RS 26,28
RX 26,27
SI 26,28
SS 26,28

Machine-instruction mnemonic codes 27

Machine-instructions
alignment and checking 25
literals, limits on 11
mnemonic operation codes 27-28
operand fields and subfields 25-26
symbolic operand formats 25
MACRO
format of 54
use 54
Macro language
extended features of 74-88
relation to assembler language 52
summary 95-100
Macro definition
defined 52
example of 55-56
how to prepare 54-58
keyword 52
(see also keyword macro definition)
mixed-mode 52
(see also mixed-mode macro definition)
placement in source program 54
use 52
Macro definition exit
(see MEXIT instruction)
Macro definition header statement
(see MACRO)
Macro definition trailer statement
(see MEND)
Macro instruction
defined 52
example of 61
format of 59
how to write 59-61
levels of 62
mnemonic operation code 59
name field of 59
omitted operands 60
example 60
operand field of 59-60
operand sublists 60-61
operands
ampersands 59
blanks 59
commas 59
equal signs 59
paired apostrophes 59
paired parentheses 59
operation field of 59
statement format 60
types of 52
used as model statement 61
Macro instruction index (&SYSNDX) 82,100
AIF instruction 82
arithmetic expressions 82
character relation 82
example 82
SETA instruction 82
SETB instruction 82
SETC instruction 82
use of 82

Macro instruction operand
(&SYSLIST) 84,100
attributes of 84
use of 84
(see also symbolic parameters)
Macro instruction prototype
statement 52,54
Macro instruction statement
(see macro instruction)
MEND
format of 54
MEXIT instruction, contrasted 77
use of 54,77
MEXIT instruction
example of 77
format of 77
MEND, contrasted 77
use of 77
Mixed-mode macro definitions
positional macro definitions,
contrasted 87-88
use 88
Mixed-mode macro instruction 87-88
example of 88
format of 88
operand field of 88
Mixed-mode prototype statement
example of 88
format of 88
operands of 88
Mnemonic operation codes 27
extended 28-30
machine instruction 27
RR format 27
RS format 28
RX format 27
SI format 28
SS format 28
MNOTE instruction
ampersands in 77-78
apostrophes in 77-78
asterisk in 78
error message 77-78
example of 77-78
operand field of 77-78
severity code 77-78
use of 77-78
Model statements
comments field of 55
comments statements 55
defined 52,55
name field of 55
operand field of 55
operation field of 55
use of 55

N'
(see number attribute)
Name entries 6
Number attribute
defined 66
example of 66
notation 66
operand sublist 66

Operand sublist
alternate statement format 60-61
defined 60-61
example of 60-61
use of 60-61
Operands
entries 6
fields 25-26
subfields 25-26
symbolic 25-26
Operating system 3
ORG instruction 49
Outer macro instruction defined 62

Paired apostrophes 59
Paired parentheses 59
Parentheses in
arithmetic expressions 70
logical expressions 69
macro instruction operands 57
operand fields and subfields 25-26
Period in
character expressions 67
comments statements 58
concatenation 57
sequence symbols 64
Positional macro definition
(see macro definition)
Positional macro instruction
(see macro instruction)
Previously defined symbols 9
PRINT instruction 47
Program control instructions 45-48
Program listings 2
Program sectioning and linking 17-18
Prototype control section name
(&SYSPSCT) 84,100
Prototype statement
example of 54
format of 54
keyword
(see keyword prototype statement)
mixed-mode
(see mixed-mode prototype statement)
name field of 54
operand field of 55
operation field of 54
statement format 54-55
symbolic parameters in 54,56-57
use of 54-55
PSECT instruction 21-22
PUNCH instruction 49

Q-type address constant 42

Relative addressing 17
Relocatability 2,7
attributes 14
program, general register 0 16
Relocatable expressions 14
in USING instructions 15-16
Relocatable terms 9
in relocatable expressions 14
pairing of 14
REPRO instruction 49

RR extended mnemonic codes 30
RR machine-instruction format 25,27
  length attribute 25
  symbolic operands 25
RS machine-instruction format 25,28
  address specification 26
  length attribute 25
  symbolic operands 25
R-type address constant 42-43
RX extended mnemonic codes 29
RX machine-instruction format 25,27
  address specification 26
  length attribute 25
  symbolic operands 25

S'
  (see scaling attribute)
Sample assembly 101-104
Scale modifier 35-36
  fixed-point constants 35-36
  floating-point constant 36
Scaling attribute
  decimal fields 65
  defined 65
  examples of 65
  fixed-point fields 65
  floating-point fields 65
  notation 65
  restrictions on use 65-66
  symbols 65
  use of 65-66
Self-defining terms 9-10
  (see also specific terms)
Sequence checking 49
Sequence symbols
  AGO instruction 74-75
  AIF instruction 73-74
  ANOP instruction 75
  how to write 67
  invalid examples of 67
  use of 67
  valid examples of 67
Set symbols
  (see also local SET symbols)
  (see also global SET symbols)
  (see also subscripted SET symbols)
  assigning values to 63
  defining 63
  symbolic parameters, contrasted 63
  use 63-64
SET variable 78
SETA instruction
  examples of 68
  format of 68
  operand field of 68
    evaluation procedure 68
    operators allowed 68
    parenthesized terms 68
    terms allowed 68
    valid examples of 68
  operand sublist 69
    example 69
SETA symbol
  assigning values to 69
  defining 63
  SETA instruction 68
  SETC instruction 69
  using 68-69
SETB instruction
  example of 73
  format of 72
  logical expression in 72
    arithmetic relations 72
    blanks in 72
    character relations 72
    evaluation of 73
    operators allowed 72
    terms allowed 72
  operand field of 73
    invalid examples of 73
    valid examples of 73
SETB symbol
  AIF instruction 73-74
  assigning values to 63
  defining 63
  SETA instruction 73
  SETB instruction 73
  SETC instruction 73
  using 73
SETC instruction
  apostrophes 70
  character expressions in 70
    ampersands 70
    periods 70
  concatenation in
    character expressions 71-72
    substring notations 71-72
  examples of 70
  format of 69
  operand field of 69
  substring notations in 70-71
    arithmetic expressions in 70
    character expressions in 70
    invalid examples of 71
    valid examples of 71
  type attribute in 70
SETC symbol
  assigning values to 71
  defining 63
  SETA instruction 70
  using 71-72
Severity code in MNOTE instruction 77
SI machine-instruction format 25,28
  address specification 26
  length attribute 25
  symbolic operands 25
SPACE instruction 47
SS machine-instruction format 25,28
  address specification 26
  length attribute 25
  length field 26
  symbolic operands 25
Standard value
  attributes of 87
  keyword prototype statement 86
START instruction
  positioning of 19
  unnamed control sections 18-19
Statements 4-7
  boundaries 4
  examples 6
  macro instructions 53
  prototype 54
Storage, defining
  (see DS instruction)

S-type address constant 42
Sublist
(see operand sublist)
Subscripted SET symbols
defining 81
dimension of 81
examples 81
how to write 81-82
invalid examples of 81
subscript of 81
using 81-82
valid examples of 81
Substring notation
arithmetic expressions in 70
character expression in 70
defined 70
how to write 70-71
invalid example of 71
SETA instruction 70
SETC instruction 70
valid examples of 71
Symbol definition, EQU instruction for 31-32
Symbolic linkages 22-24
Symbolic operand formats 25-27
Symbolic parameter
concatenation of 57
defined 56
how to write 56
invalid examples of 56
replaced by 56-57
valid example of 56
Symbols
defining 9-11
length attributes 9,11
length, maximum 9
previously defined 9
restrictions 9
System macro instructions defined 53
System variable symbols 82-83
(see also specific system variable symbols)
assigned values by assembler 82-83
defined 82

T'
(see type attribute)
Terms
expressions composed of 9
in parentheses 12-13
pairing of 14
Time variable (&SYSTIME) 85,100
TITLE instruction 46-47
Type attribute
defined 64-65
literals 65
macro instruction operands 65
notation 65
symbols 65
use 65

Unconditional branch
(see AGO instruction)
Unconditional branch instruction 28
Unnamed control section 19
USING instruction 15-16,17

Variable symbols
assigning values to 53
defined 53
how to write 53
types of 53
use 53
(see also specific variable symbols)
Virtual storage concept 1-2
V-type address constant 42
V-type address constant 41-42

GC28-2000-5

ASSEMBLER LANGUAGE Printed in U.S.A. GC28-2000-5

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
[U.S.A. only]

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
[International]